Autonomous Container Handling

A solution for handling containers using state-of-the-art autonomous ground vehicles.

Magnus Tomren Ekløff & Kristoffer Syrdal Tronstad

Supervisors:
Morten Hallquist Rudolfsen (University of Agder) & Teodor Nilsen Aune (Red Rock)

This Master's Thesis is carried out as a part of the education at the University of Agder and is therefore approved as a part of this education. However, this does not imply that the University answers for the methods that are used or the conclusions that are drawn.

University of Agder, 2019
Faculty of Engineering and Science
Department of Engineering Sciences

Abstract

This Master's thesis is carried out as a part of the Mechatronics Master's program at University of Agder. The thesis describes the development of an autonomous ground vehicle prototype for moving containers, as well as presenting a state-of-the-art of the container industry.

First the theory utilized in the methodology of the thesis is presented. Then the method's used to gather the desired results is described. Lastly the results are presented and discussed, before a conclusion is drawn.

The project follows the V-model methodology and started with the design of a system overview and selection of suitable hardware and software. Robot Operating System (ROS) was used as the main software in this thesis. The software was first tested as individual modules on the selected hardware before implementation into larger sub-systems for integration testing. Finally, all the sub-systems was combined into the complete system for the final testing.

A 1:14 scale RC truck was assembled and used as the base of the prototype. Remote access was set up with a wireless hotspot to control the prototype with an external computer. The first step to autonomous driving was to drive the prototype manually and develop the low-level control for Ackermann steering. Secondly an indoor localization system based on ArUco tracking and an extended Kalman filter was created to estimate the pose of the prototype. The low-level control and localization was combined into the *navigation stack* in ROS. The ROS navigation stack also includes a local and global path planner, in addition to a map for the prototype to navigate in. Simultaneous localization and mapping (SLAM) was implemented to provide mapping capabilities.

The path planning was tested with a point-to-point driving test width dynamic obstacle avoidance. Finally, the full autonomous capabilities of the prototype was showcased by a demonstration program where the prototype was given a set of user defined waypoints. The prototype proceed to drive through the waypoints in a continuous loop whilst being able to avoid several dynamic obstacles.

Acknowledgements

This thesis is a part of the Master's program in Mechatronics, at the Department of Engineering Sciences, at the University of Agder. The task in hand proved to be challenging, containing key aspects of the mechatronics and robotics field. The thesis members are pleased with the final results, after five months of dedicated work.

We would like to thank our supervisor, Morten Rudolfsen for his much appreciated inputs, as well as Teodor Aune and Dr. Andreas Klausen from Red Rock.

We would also like to thank Red Rock for providing the hardware utilized in this thesis to develop the prototype.

Magnus Tomren Ekløff	Kristoffer Syrdal Tronstad
Magnus Tomren Ekløff	Kristoffer Syrdal Tronstad
Signature	*Signature*
24.05.2019	24.05.2019
Date	*Date*

24. mai, 2019

Contents

List of Figures

List of Tables

1. Introduction

The demand for autonomous systems is ever increasing in the modern world. Autonomous vehicles has become a reality and pave the way for future systems where computers take over the art of driving, executing tasks and navigating.

1.1 Background

Shipping containers revolutionised the global movement of goods. The next revolution in container handling is totally automated container ports and supply chain. Container ports are an ideal place to implement an autonomous system. The operational area is limited to a fully controlled environment, enabling installation of specialized infrastructure. Unlike autonomous cars, the decision making can be implemented in layers extending from the automated machinery all the way to the core software of the container terminal. The container terminals have safe access control systems built in accordance with well-defined industrial safety standards which again will enable the automated system to work parallel to humans in a safe manner.

The world biggest container port is the Yangshan Deep Water Port in Shanghai with an annual Twenty-foot equivalent unit (TEU) of approximately 40 million [1]. One tenth of the port is fully automated with a TEU of 4 million. In comparison the container port in Kristiansand, Norway have an annual TEU of 50 000 [2].

Only 3% of the shipping terminals in the world was semi or fully automated by the year 2018 [1]. It is estimated that the automated container market will experience an increase of 20% by the year 2023. The port of Qingdao began automated container handling in the year 2017, capable of handling 1 million TEU. The entire system cost $468 million to install. The investment reduced the number of workers from 60 to 9, while increasing the efficiency by 30%. The investment became profitable a mere 10 month after it was opened [1].

Autonomous container handling is the future within the shipping industry. Three percent of the big container ports in the world have installed systems of autonomous container cranes, gantry cranes and custom-made trucks. These systems are not suited for smaller container ports, due to complexity and price. As of today, there is no existing commercially automated system for smaller container ports.

1.2 State-of-the-Art

This section will discuss the highest level of general development within the container handling industry.

The shipping industry is undergoing rapid changes. A common trend across the industry is supply chain visibility, implementation of new technology, increasing rate levels and more efficient transportation management. Container ships are increasing their capacity which demands more efficient container handling in the container ports. Several container ports worldwide have implemented semi or fully automated container handling solutions. Automated ports make supply chain management efficient and help to increase rate levels, shipping time and revenue.

Small high-tech companies are joining forces with global shipping firms. Carriers adopt new technology which again boost automated online resources, increased visibility in the supply chain and enables seamless shipping solutions worldwide [3].

For the container terminal operators, the automation is not just about handling more cargo. Automated systems make the container ports increase the efficiency on the most limiting factor: space. Intelligent systems know the supply chain and stack the containers in the most efficient manner possible [4].

An automated container port requires minimum two types of hardware: An overhead crane to unload the vessel and a vehicle to move the container to the desired location. There exist a wide variety of automated cranes and different kinds of autonomous container handling vehicles on the marked. Autonomous ground vehicles (AGV), straddle carriers and gantry cranes are the solutions used on big automated ports to transport and unload containers. The hardware is well suited for big automated ports, as of today there is few solutions to automate small container ports.

Kalmar is currently the leading company when it comes to delivering fully automated port systems. They deliver cranes, AGV, straddle carriers and gantry cranes as well as a full software package called Kalmar OneTerminal. There exist several other companies that deliverers container handling hardware and software solutions like Hyundai, Toyota and KoneCranes. A container port software normally consists of three layers of automation: A Terminal Operating System (TOS), Equipment Control System (ECS) and Equipment Automation.

There exist two possible container handling solutions for small container ports. A reach stacker or a straddle carrier as illustraded in Figure 1.1 and Figure 1.2. The two vehicles have their own pros and cons and the choice of handling system depends on several criteria like the shape and size of the terminal, productivity and annual TEU [5].

Figure 1.1: Reach Stacker [6]

Figure 1.2: Straddle Carrier [7]

The straddle carrier is meant for small to medium size ports. It transports containers from ship to shore cranes to the terminal or loading area. A straddle carrier can handle two 20 feet container at the time and is capable of stacking one over two as well as stacking three containers high. It is only capable of stacking along one row and therefor it requires a driving lane on each side of the container stack. The straddle carrier is capable of delivering containers to road trucks, but not to rail cars. The footprint of the straddle carrier is quite big, and they are high. This makes them relatively slow and they can not access conventional warehouses.

The reach stacker is highly practical in small container ports and operations that require great flexibility. Every fourth container shipped in the world today is moved by a reach stacker container handler. [8] The reach stacker can perform several tasks on the container port like loading and unloading small vessels, transport containers, load trucks or rail cars and drive indoors in warehouses and stacking containers in the yard. The reach stacker is capable of stacking container in block stacks three to four deep and four to five containers in height. The reach stacker can manoeuvre while holding a container, which facilitates accurate positioning. The visibility is also much better than a forklift solution.

In 2019 there is no autonomous reach stacker available on the marked. However there exist one electric reach stacker, the XCMG XCS45-EV. There is no electric straddle carrier available, however Kalmar deliverers a hybrid straddle carrier equipped with a sensor suit meant for total automation.

Based on the trends within the shipping industry, there will be an increasing demand for autonomous container handlers and smart shipping solutions. Big container ports have already implemented autonomous systems, the next phase is to implement the technology to smaller ports. The reach stacker is without doubt the most effective container handling solution on small ports due to the versatile design and there should be an interest in the shipping industry to develop a fully automated reach stacker.

1.3 Objective

The main objective of this thesis is to develop a state-of-the-art autonomous ground vehicle prototype for container handling on shipping yards. The vehicle should be able to autonomously navigate from one location to another while avoiding obstacles. A sensor suite has to be configured and solutions enabling autonomous navigation has to be investigated.

Red Rock requests a working prototype in 1:14 scale, that will function as a platform to showcase their product to potential investors and buyers. The prototype must be able to navigate in a closed area, gather information about its surrounding to determine its position, generate a path from current location to the new desired location, and finally drive autonomously to the goal location while avoiding dynamic obstacles.

Due to the shear size of the project, no previous experience with Robot Operating System (ROS) and available prototype vehicle, it was decided in accordance with Red Rock and the project supervisors to focus on the autonomous base of the container handler. The loading and unloading of container using the container spreader is not considered a part of this project.

1.4 V-Model

The software in this master thesis is developed according to the V-Model. The V-Model is a software development method where the development happens in a sequential manner. Each development stage is directly associated with the testing phase. The sequence of development is listed in a chronological order below:

- Requirement analysis
- System design
- Module design
- Software design
- Module testing
- System integration testing
- System test

The V-Model approach is described in detail in Section 3.1, in the *Method*-chapter.

1.5 Thesis Structure

The report is composed of six chapters: Introduction, theory, method, results, discussion and conclusion. In addition, there is an appendix with code and data sheets. The task, background and state-of-the-art within autonomous container handling is presented in the introduction. The theory chapter contains all relevant theory applied in the thesis. The method chapter describes the practical usage of the theory and how its implemented in the software and prototype. The results chapter presents all the results obtained in the project. The discussion chapter discuss improvements, alternative hardware, issues with hardware and software and other relevant aspect of the thesis. The final chapter is the conclusion chapter, which summarize the entire thesis and draws conclusions.

2. Theory

This chapter covers the relevant theory regarding the content of the thesis and contains theory about: Operating system, Robot Operating System (ROS), Ackermann kinematics, sensors, localization, GPS, extended Kalman filter, Simultaneous localization and mapping (SLAM) and the navigation stack.

2.1 Container Port Architecture

Container ports consists of a quay and a terminal. The containers are unloaded from the ship situated at the quay by an overhanging crane. There is an area separating the quay from the container terminal. The terminal is the area where the containers are stored, stacked and sent away either by a truck or railway. Figure 2.1 illustrates a typical container port architecture.

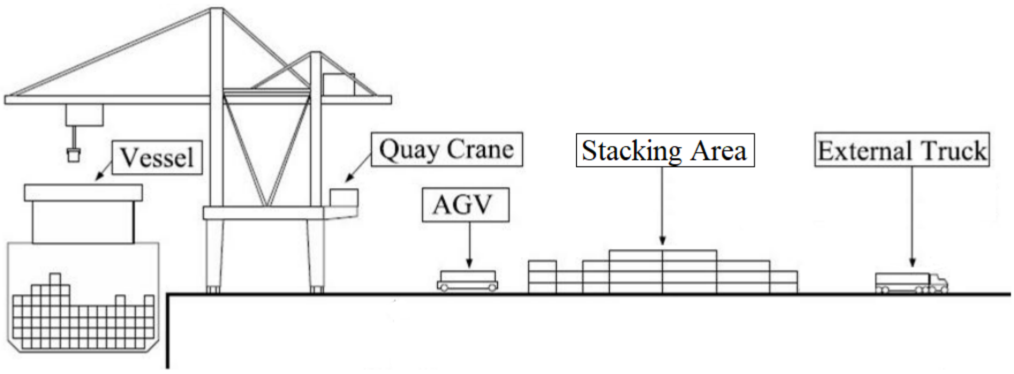


Figure 2.1: Container Port Architecture [9]

Small container ports tend to have a less systematic outlay, with container stacked where ever there is free space rather than a continuous stack.

2.2 Operating System

This section contains information about the three levels of container port automation. The three levels are: Terminal operating system (TOS), equipment control system (ECS) and equipment automation. Note that autonomous container handlers refers to several different automated container handling solutions in this chapter on a general basis, like cranes, autonomous ground vehicle, straddle carriers and reach stacker's. An overview of the container port automation is displayed in Figure 2.2 on page 8.

2.2.1 Terminal Operating System

A port with automated container solutions relies on a terminal operating system (TOS). The TOS is the key part of the supply chain and aims to coordinate the movement and storage of cargo. The system uses technology to monitor the flow of containers in, out and around the container port [10]. Data from various sources are sent to a central database in real time. The database provides information about gods status and location of the autonomous container handlers.

The TOS system enables efficient use of resources like space, labour, equipment and workload. Every task is monitored from high level vessel planning down to container handling in the port. TOS has two main functions from the ECS system perspective [11]:

- Maintain a correct container inventory based on information received from the ECS
- Plan the storage location of containers and provide job orders to the ECS

2.2.2 Equipment Control System

The equipment control system (ECS) monitors and controls all events and processes at equipment level. The TOS dictates which container the autonomous container handler is supposed to move, the location of the container and where it is supposed to be moved. The ECS system receives the information from TOS and then provides the vehicle or crane with a global path and information about the container like serial number, colour and other useful information. The ECS keep track of safety features and vehicles coordination. The interaction of different equipment is also coordinated by the ECS.

The communication between the ECS and TOS contain the following information [11]:

- Submit and confirm work order
- Update status and location of equipment
- Job concluded or interrupted
- Area status update

2.2.3 Equipment Automation

This thesis is focusing on the equipment automation part of the terminal operating system. The equipment automation is the control system implemented on the autonomous container handler. Sensor data is processed at the equipment automation level and container handling commands are executed. The autonomous container handler calculate local paths, avoids obstacles, lifts the container and move to the desired position. The following functions are typically performed at equipment level:

- Receiving work order
- Calculate path
- Avoid obstacles
- Control container movement

- Simultaneous localization and mapping (SLAM)
- Validating and confirming work order

A graphical representation of the full scale operation system is illustrated in Figure 2.2.

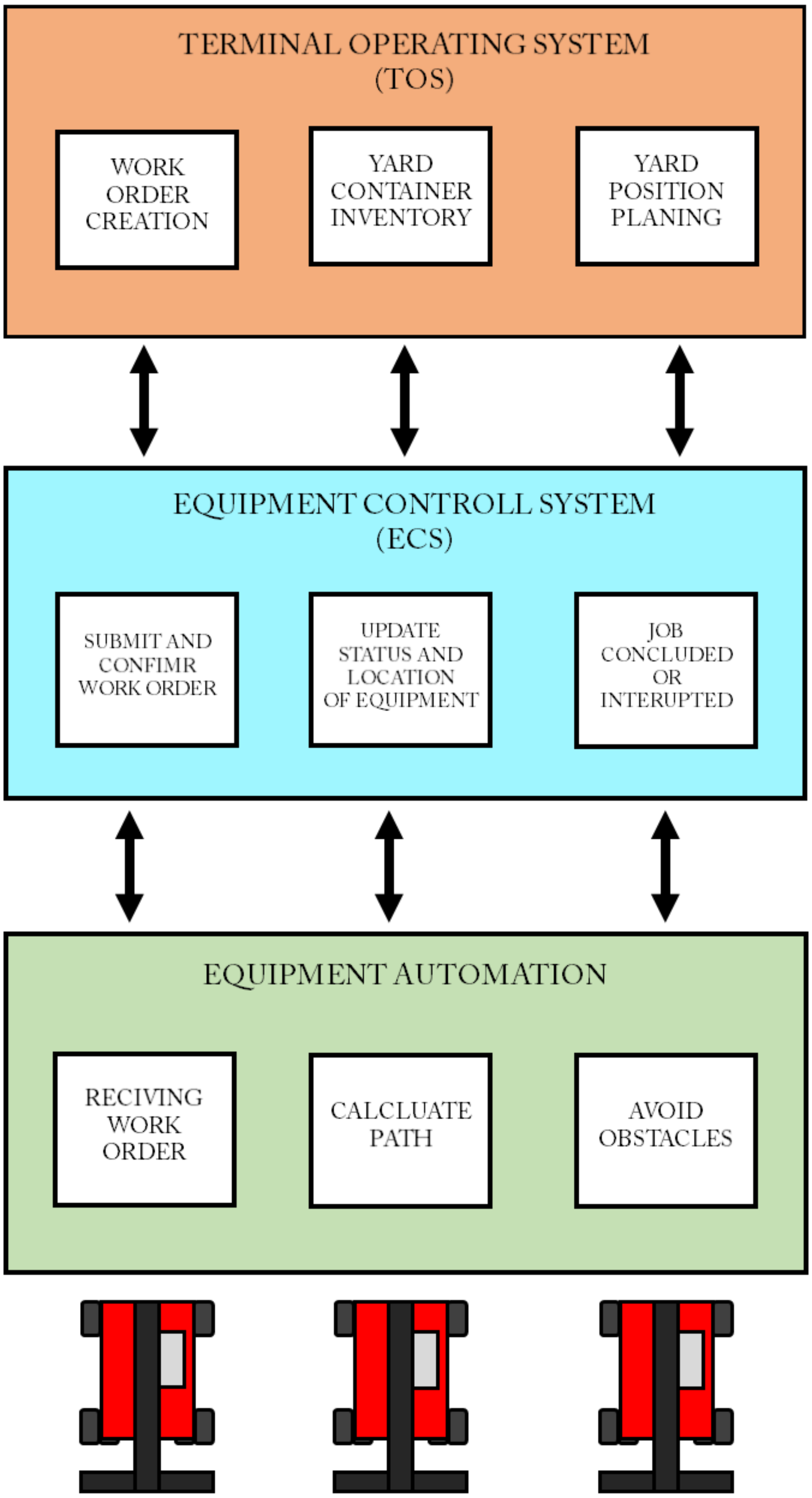


Figure 2.2: Full Scale Operation System

2.3 Robot Operating System

Robot Operating System, or ROS for short, is a robotics middleware that provides a framework for robot development [12]. The system provides open source libraries and tools to help software developers creating robot applications. It provides support for hardware, drivers, visualizers and more.

The reason for using ROS is that it is open source and supported by user's worldwide that provides code and insight into projects. Instead of reinventing the wheel every time a new project is started, the ROS frameworks help development by providing drivers, libraries and managing how the code is developed. The ROS framework has proven to be an effective way of boosting robot development.

2.3.1 Nodes

ROS enables communication between multiple computers with different programming languages. ROS is constructed by several *nodes*. A node is a executable process that performs some sort of computation [13]. A robot usually consists of many nodes, one for each process. The ROS-core handles the communication between the nodes and establishes the connection between them. For example, one node control a laser rangefinder, another node control the motor and a final node performs localization.

One of the main benefits of having a system composed of several nodes is fault tolerance, as crashes and system faults are isolated to individual nodes and not necessary crashing the entire system. In addition, the code is broken down to a more modular design reducing the complexity of scripts.

2.3.2 Topics

A *topic* is a communication bus which nodes exchange messages [14]. In general, nodes are not aware of what nodes they are communicating with. Instead, nodes *subscribe* to the relevant topic to receive data. Nodes that generate data *publish* it to the relevant topic. There could be multiple publishers and subscribers to one topic.

2.3.3 Messages

Nodes communicate with each other using *messages* [15]. A node publishes a message to a topic in order to send information. For another node to receive that information, it has to subscribe to the same topic.

nav_msg/Odometry

The most common message in ROS for mobile robotic is the `nav_msg/Odometry`-message. The message is standard for communicating robot pose changes and is configured for 6 degrees of freedom (DOF). The message is a combination of a *header*, a `Pose`-message and a `Twist`-message from the `geometry_msg`-type. A typical odometry message is generated as described below:

Start by defining the variable as a `nav_msg/Odometry`-message:

```
odom = nav_msg/Odometry()
```

The header contains a *time stamp*, in addition to both the parent and child-frame ID.

```
odom.header.stamp = rospy.Time.now()
odom.header.frame_id = "odom"
odom.child_frame_id = "base_link"
```

The `Pose`-message is a point in (x, y, z)-coordinate and an angle in quaternion-space.

```
odom.pose.pose.x = x
odom.pose.pose.y = y
odom.pose.pose.z = z
odom.pose.pose.theta = Quaternion(theta)
```

Finally, the `Twist`-message contains the linear and angular velocity in (x, y, z)-direction.

```
odom.twist.twist.linear.x = vx
odom.twist.twist.linear.y = vy
odom.twist.twist.linear.z = vz
odom.twist.twist.angular.x = rvx
odom.twist.twist.angular.y = rvy
odom.twist.twist.angular.z = rvz
odom.twist.twist.angular.w = rvw
```

All of the variables above have to be assigned a value to create an odometry message.

2.3.4 ROS Qt Graph

ROS has an integrated feature to generate a graphical overview of the software in *Qt*, an open-source widget for creating graphical user interfaces, called *rqt_graph*. The graphical overview displays the network of nodes, topics and messages. An example graph is illustrated in Figure 2.3.

Figure 2.3: RQt-graph Example from *Turtle Sim*

The graph consists of two nodes: `/teleop_turtle` and `/turtlesim`, where `/teleop_turtle` is publishing the `/command_velocity`-topic under the `/turtle1` main topic. `/turtlesim` is subscribing to the same topic, thus receiving the desired information (a velocity command in this case).

To launch the viewer, simply execute `rqt_graph` in the terminal.

2.3.5 Unified Robot Description Format

Unified Robot Description Format, or URDF, is the standard robot format in ROS. The format consists mainly of two components: *links* and *joints*. The links are the physical components in the model, for instance a wheel or chassis. Joints describes how links moves relative to each other. A joint has a parent-link and a child-link. The documentation of a `.urdf`-file is based on *XML*-language.

2.3.6 RViz

RViz is a 3D-visualization tool in the ROS framework [16]. The program visualizes node data from ROS. For instance, a point cloud from a depth camera, a map generated by a robot or path planning. The tool is very helpful for simulation, testing and development. Figure 2.4 show the default view in RViz.

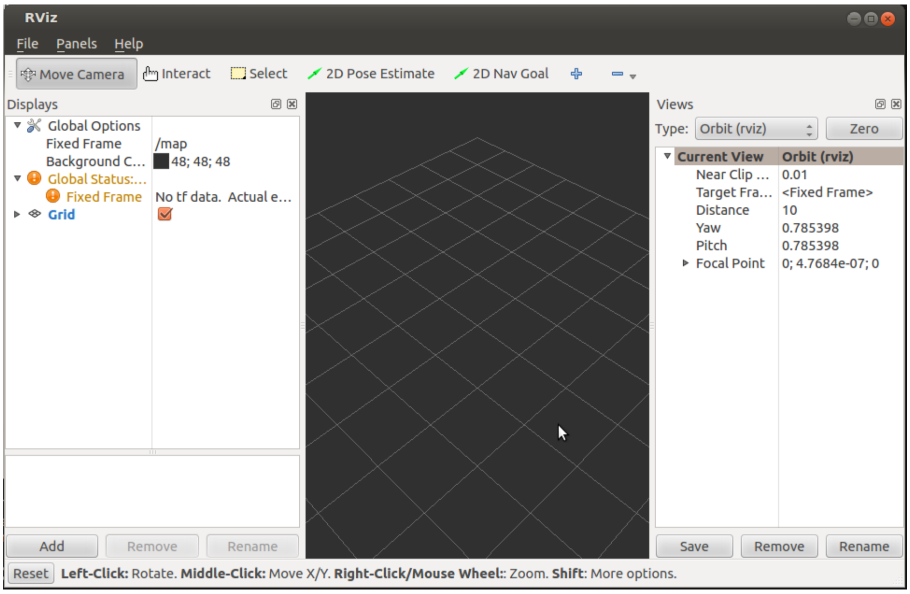


Figure 2.4: Default View in RViz

Furthermore, RViz works as an HMI for ROS. The program could for an instance be used to publish the initial pose of a robot, publish a goal pose for a robot and publish waypoints.

2.3.7 Useful ROS Commands

The following subsection lists several useful ROS commands. Table 2.1 contains the bash-commands and a short description of their function.

Table 2.1: Useful ROS Commands

Command	**Description**
roscore	Runs ROS master-node
catkin_make	Compiles workspace
roscd <package>	Changes directory to a ROS-package
catkin_create_pkg <package name>	Creates a ROS-package
roslaunch <package> <launch-file>	Run launch file
rosrun <package> <executable>	Run individual ROS-nodes
rostopic list	List all the active topics
rostopic echo <topic>	Prints message being published to topic
rostopic pub <topic> <message-type>	Publish message to topic

2.4 Kinematics

The kinematics of a car-like robot could be broken down into two frames: the *fixed global frame* and the *dynamic local frame* [17]. If the motion of the local frame is known, it could be transformed to the global frame. Thus, the position could be calculated in global world coordinates.

2.4.1 Ackermann Steering

The intention of Ackermann steering is to avoid the front tires from slipping when following a curve shaped path. Each wheel has the axle arranged as the radius of circles with a common centre point, the instantaneous centre of rotations (ICR). The outer wheel has a greater radius than the inner wheel. An approximation to a perfect Ackermann steering is obtained by moving the steering pivot points inwards. Ackermann steering provides a fairly accurate dead reckoning solution and is often the solution of choice for big outdoor autonomous equipment like a reach stacker. Figure 2.5 displays the basic concept of Ackermann steering. The Ackermann steering formula is displayed in Equation (2.1).

The Ackermann steering allows the vehicle to drive in a circle with a common centre of rotation, the kinematics can therefore be approximated by those of a tricycle.

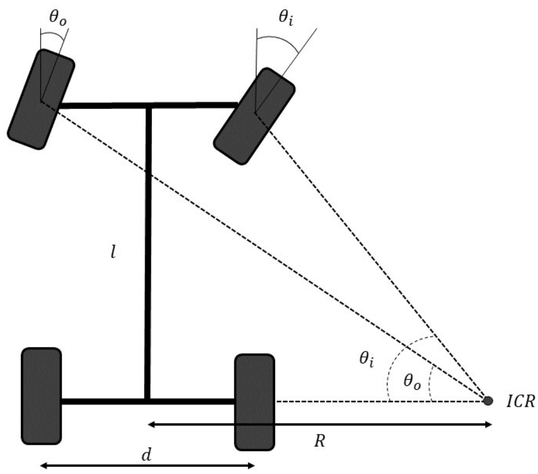


Figure 2.5: Ackermann Steering

$$\cot\theta_i - \cot\theta_o = \frac{d}{l} \tag{2.1}$$

Where:
d: Lateral wheel separation.
l: Longitudinal wheel separation.
θ_i: Relative steering angle of inner wheel.
θ_o: Relative steering angle of outer wheel.

The vehicle is travelling at relatively low speed, it is therefore assumed that the front wheels rolls without experiencing slip. Ackermann steering dictates that the required steering torque will increase with increase in steering angle. In parallel steering the trend is opposite, thus positive feedback, which is not desirable.

The forward kinematics is used to predict the future pose of the vehicle. The model is simplified to the three-wheeled tricycle model displayed in Figure 2.6. The velocity is described by Equation (2.2) in x-direction and Equation (2.3) in y-direction:

$$\dot{x} = u_1 \cdot \cos\theta \tag{2.2}$$

$$\dot{y} = u_1 \cdot \sin\theta \tag{2.3}$$

Where:
$\dot{x}$: Velocity in x-direction
$\dot{y}$: Velocity in y-direction
u_1: Tangential velocity

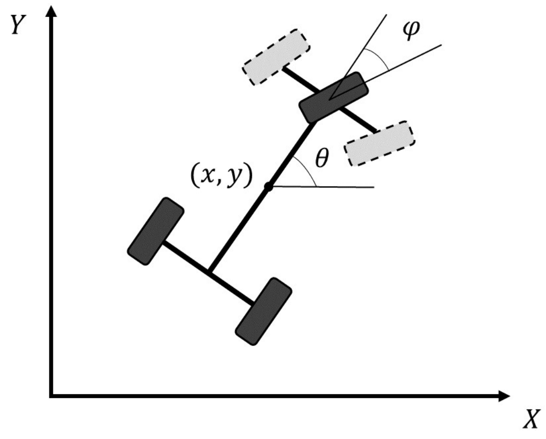


Figure 2.6: Inverse Kinematics

The rotational velocity $\dot{\theta}$ around the instantaneous centre of rotation (ICR) is defined by Equation (2.4):

$$\dot{\theta} = \frac{u_1}{l} \cdot \tan\phi \tag{2.4}$$

Where:
$\dot{\theta}$: Angular velocity around ICR
l: Longitudinal wheel separation
u_1: Tangential velocity
ϕ: Steering angle
θ: Vehicle orientation

2.5 Deadband Compensation

Deadband compensation eliminate odometry errors at low speed. The error in the odometry data is corrected by measuring the deadband in the motor and drive chain. A deadband compensation is executed in order to make the robot behave more like an ideal linear plant. The motor and gearbox are subjected to two kinds of friction: Coulomb and viscous. Coulomb friction is mostly static and will counteract the initiation of movement. Viscous friction increases with higher velocity.

The coulomb friction is determined by increasing the motors duty cycle gradually until the wheels starts to rotate. The coulomb friction is visible as a bias in the compensation graph.

There is no need for measuring the actual friction when compensating for the viscous friction. It is enough to register a known input duty cycle and measure the output velocity. A plot could then be created using linear regression on each of the two halves (-1 to 0 and 0 to 1) which is illustrated in Figure 2.7.

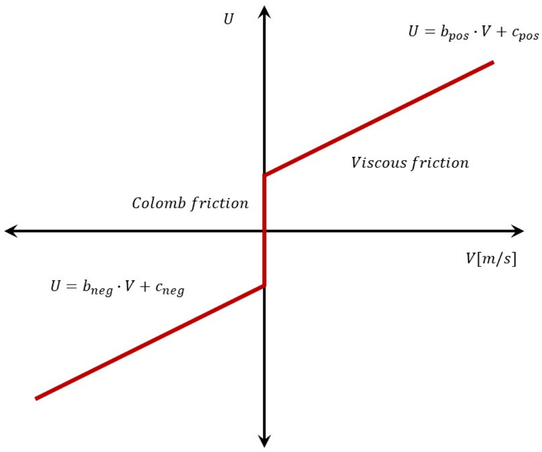


Figure 2.7: Friction Plot

A motor has two individual frictions, depending on the direction of movement. Equation (2.5) is used to calculate the duty cycle:

$$U = \begin{cases} b_{pos} \cdot V + c_{pos} & V > 0 \\ 0 & V = 0 \\ b_{neg} \cdot V + c_{neg} & V < 0 \end{cases} \tag{2.5}$$

Where:
V: Desired velocity
U: Duty cycle signal written to the motor
b_{pos}: Viscous friction coefficient for the positive velocity
b_{neg}: Viscous friction coefficient for the negative velocity
c_{pos}: Coulomb friction coefficient for the positive velocity
c_{neg}: Coulomb friction coefficient for the negative velocity

2.6 Camera Calibration

Most cameras add some kind of distortions to an image. Distortion is a deviation from rectilinear projection, that cause straight lines to appear curved.

2.6.1 Distortion

The two most common types of distortion are: *radial* distortion and *tangential* distortion [18]. The radial distorting causes straight lines to appear curved. Radial distortion appears both negative and positive. Tangential distortion occurs when the lens-plane is not parallel with the image-plane, thus making objects in the lower part of the image seem closer and objects in the upper part seem to be further away. Figure 2.8 illustrates both radial and tangential distortion.

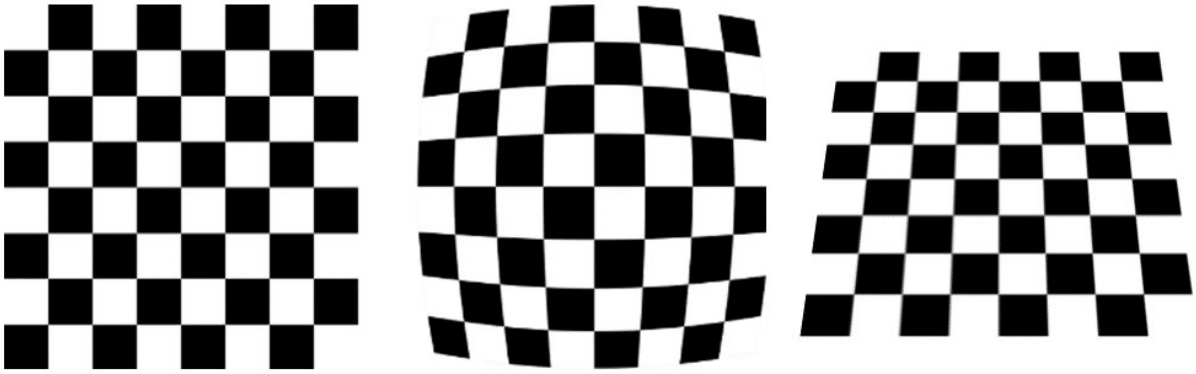


Figure 2.8: No Distortion vs. Positive Radial Distortion vs. Tangential Distortion [18]

The effect of radial distortion becomes greater, further away from the centre of the image-plane. Equation (2.6) and Equation (2.7) show how radial distortion is presented in equations:

$$x_{distorted} = x(1 + k_1 r^2 + k_2 r^4 + k_3 r^6) \tag{2.6}$$

$$y_{distorted} = y(1 + k_1 r^2 + k_2 r^4 + k_3 r^6) \tag{2.7}$$

Where:
$k_{1,2,3}$: Radial distortion parameters

The tangential distortion is presented in Equation (2.8) and Equation (2.9):

$$x_{distortion} = x + [2p_1 xy + p_2(r^2 2x^2)] \tag{2.8}$$

$$y_{distortion} = y + [p_1(r^2 + 2y^2) + 2p_2 xy] \tag{2.9}$$

Where:
$p_{1,2}$: Tangential distortion parameters

To summarise, there is five parameters influencing the camera distortion:

$$[k_1 \quad k_2 \quad k_3 \quad p_1 \quad p_2]$$

Furthermore, some additional information is required: The intrinsic and extrinsic parameters of the camera. Intrinsic parameters are specific to a camera and they include information about the focal length (f_x, f_y) and optical centre (c_x, c_y) of the camera. The parameters are used to create the camera matrix, which compensate for the distortion created by a lens with a specific characteristic. The dimension of the camera matrix is 3×3:

$$\text{Camera Matrix} = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix}$$

2.7 Depth Camera

Acquiring depth information from the scene is one of the most crucial problems in modern computer vision. Computer vision is increasingly popular in industrial applications and is used in a broad variety of fields such as industrial automation, safety systems, measuring equipment, 3D recognition and augmented reality [19]. The classical depth measurement methods are stereo vision, time-of-flight and structured light.

2.7.1 Time-of-Flight

Time-of-flight or TOF is a method where light is actively illuminating the scene. The light is reflected back and captured by a charge-coupled device (CCD)-sensor. The distances within the image can then be determined by calculating the phase shift of the returned light or by calculating the time the light spent travelling to the object in the scene and back to the sensor. The disadvantage of the TOF method is the relatively high price, low resolution and complexity of device [20].

The advantages of this method is that the system is compact and that the illuminating source can be placed right next to the image sensor. The TOF system does not require high computational power compared to stereo vision that demand complex correlation algorithms to measure distances. TOF cameras are suited for real time applications because they are capable of measure distances in an entire scene in one image frame.

There are several disadvantages with TOF cameras, especially when it comes to background light and interference. A normal TOF camera emits approximately 1 watt of IR light per square meter and the sun emits approximately 1050 watts of IR light per square meters. Interfering light can cause problems outdoors. TOF cameras are also prone to interference. Multiple reflections could cause measurement errors. The light is illuminating the entire scene and for a phase difference device this can cause reflection problems. The light reaches the object through several paths due to reflective surfaces, this causes the measurements to be greater than the actual distance. Direct TOF images experiences problems when light is reflected from a specular surface.

2.7.2 Stereo Vision

Stereo vision is based on two camera sensors separated by a certain baseline. 3D information from the scene is obtained by examining the relative position to an object from two different vantage points. The relative depth information is obtained in a disparity map, which refers to the apparent horizontal pixel coordinate difference in the stereo images. The disparity values are inversely proportional to the scene depth at the corresponding pixel location.

It is possible to combine stereo vision with a structural light source to increase the accuracy of the camera. The Intel RealSense is an example of a active stereo camera. An IR emitter projects a pattern of structural light in order to simplify stereo matching [21].

2.7.3 Structural Light

The structural light stereo camera experiences problems in measuring depth if there is another IR source interfering the projection. Indirect illumination is a problem occurring on reflecting surfaces, because dots are projected on other parts of the scene. Object situated with a flat angle relative to the camera could also result in lack of depth information.

2.8 IMU

Inertial measurement unit or IMU, is a chip containing three gyroscopes and accelerometers mounted orthogonal on each of the three axis. The inertial measurement unit works by detecting linear acceleration using the accelerometers and rotational rate using the gyroscopes.

2.8.1 Accelerometer

Acceleration is detected by measuring deflection and thereby forces acting on a microscopic mass-spring system. By integrating the acceleration measurement twice, the position change is known. The acceleration measurement is in a scale where the value 1 corresponds to 1**g** in forward direction and -1 corresponds to 1**g** of acceleration in the negative direction.

Accelerometers detect collisions and other events which are likely to disturb the position. When used to estimate position, the values have to be integrated twice which is a process that is extremely noise sensitive. The accelerometer requires known orientation. It is not possible to distinguish gravity from other acceleration affecting the robot, so the estimate of the orientation will only be good if the other accelerations are small compared to gravity. Accelerometers tend to obtain distorted values due to external forces as gravitational forces in motion; which then accumulates as noise in the system. Accelerometers react quickly but accumulates error over time due to accelerometer jitters and noise. Accelerometers are therefore not reliable for inertial measurement systems alone.

2.8.2 Gyroscope

A gyroscope is a device used to determine orientation by measuring angular velocity and integrate it. The gyroscope is used to reduce the uncertainty in orientation. The gyroscope chip measures deflection caused by rotation of a small oscillating micro electro-mechanical system (MEMS). The deflection is measured by use of the theory behind the Coriolis effect. The Coriolis effect states that an inertial force acts on a mass which is moving relative to a rotating frame of reference. The main problem with a gyro is that the angle estimates drifts over time because it only sense changes and have no fixed frame of reference. It is also sensitive to noise and bias.

2.8.3 Combining Gyroscope & Accelerometer Data

Sensor fusion is combining sensor data such that the resulting information has less uncertainty than individual measurements. Accelerometers and gyroscopes can obtain accurate sensor readings when combined. The accelerometer is useful to calculate the position of an object moving at relatively constant velocity. Since the accelerometer is prone to noise and disturbances due to accelerations other than gravity and vibrations, it is not reliable alone. On the contrary a gyroscope is used to measure angular velocity, which could be integrated into angular position and thereby the position of a robot. Due to inaccurate measurements, a position error will accumulate over time because the gyroscope only sense changes and have no fixed frame of reference.

The accelerometer and gyroscopes properties complement each other in a way that they can be used to calibrate each other. The long-term accuracy of a gyroscope combined with the short-term accuracy of the accelerometer improves the overall accuracy. The addition of accelerometer data allows the bias in a gyroscope to be minimized, this reduce propagating errors and improves orientation readings. Accelerometers sense directional changes with respect to gravity which can orient a gyroscope to calculate angular displacement with higher accuracy.

2.9 LiDAR

Light detection and ranging (LiDAR) is a method used to detect and measure distance to objects. The LiDAR sensor emits a laser beam and measures differences in return time and wavelengths to map physical features with high resolution. The point cloud from the LiDAR provides information about the obstacles surrounding the robot.

The equation for measuring laser beam distance D is shown in Equation (2.10):

$$D = \frac{t_d \cdot c}{2} \tag{2.10}$$

Where:
t_d: Flight time
c: Speed of light
D: Distance travelled

2.10 Localization

This section explains the theory behind the localization method used to navigate a full-scale reach stacker. The reach stacker requires a redundant and precise GPS system, in order to navigate correctly and avoid collisions. The localization system implemented on the down-scaled prototype is also elaborated.

2.10.1 Real Time Kinematic GPS

The full-scale reach stacker rely on a Real Time Kinematic (RTK) GPS system for navigation. The localization system should be similar to the systems used on excavators and other type of heavy equipment. The accuracy should be within 100 mm, in order to avoid collisions and position the reach stacker within reach of the container [22].

A high accuracy and redundancy are achieved by implementing a positioning system relying on two signals. The two signals originate from the American GPS system and a radio transceiver that transcends correctional signals from one or several local base stations situated on the outskirts of the container port. The base stations are located on an absolute position with known world coordinates. The base stations receive the same signal with the same error as the GPS receivers mounted on the vehicle, however the error is used to calibrate the system since the base stations absolute position is known. The calibration messages are sent through radio link and is used to correct the position in real time. The system is capable of pinpointing the position with an error of a few centimetres [22]. The RTK system could be combined with IMU and odometry data through an extended Kalman filter to achieve higher precision and redundancy.

A radio modem broadcasting low cost signals is the preferred real time signal for the RTK GPS. The radio signal is commonly in the Ultra High Frequency (UHF) band and most countries provide frequencies allocated specifically for RTK purposes [23]. RTK is accurate up to about 20 km from the base station.

The navigation system might experience problems with obtain the GPS fix quickly. The solution to this problem is to implement an assisted Global Positioning System (aGPS). The RTK system relies on custom base stations while aGPS uses ordinary cell phone towers to estimate the position and signal correction [24]. The system can reduce the time to first fix (TTFF) significantly and is used in cases of weak signals, that are only temporally available.

2.10.2 GPS Navigation

The coordinate system that is most applicable to position the full-scale reach stacker is the Universal Transverse Mercator (UTM) [25]. The UTM projection uses a two-dimensional coordinate system to output position information. The UTM is not a single map projection, the surface of the earth is divided into 60 equal zones. The only region of the world that is not uniform is located in Norway, one region at Svalbard and one region south west in Norway is extended.

The position of the mechatronics lab at UiA Grimstad has the following UTM coordinate: 32V 475183 6465986. 32V is the UTM region of southern Norway. 475183 is the Easting in meters and 6465986 is the Northing in meters. The origin is located in the bottom left corner of the grid zone. Figure 2.9 shows how the zones in Europe is divided.

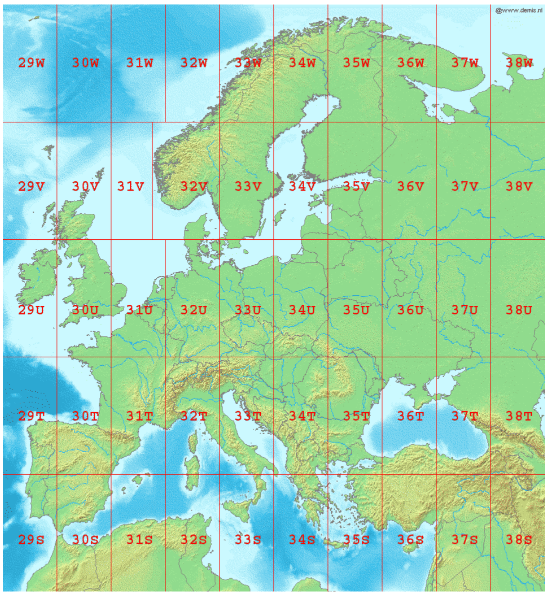


Figure 2.9: Grid Zones Europe [26]

Due to the fact that the prototype is tested in a controlled indoor environment, the satellite signal is blocked by the building infrastructure. An artificial GPS signal has to be created in order to verify the pose of the vehicle. The artificial GPS signal is created with a camera and an ArUco marker.

2.10.3 ArUco Markers

Augmented reality markers created by the University of Córdoba, more commonly known as *ArUco* markers, are often used for pose estimation, which is of great importance in robot localization. The process is based on correspondences between points in the real environment and the 2D-image projection [27].

The ArUco marker method uses binary square fiducial markers composed of a wide black border and a inner binary matrix which determines its identifier (ID).

An ArUco marker could have various matrix sizes, however a 4 × 4 or 6 × 6 matrix is the most common. Figure 2.10 shows some examples of ArUco markers.

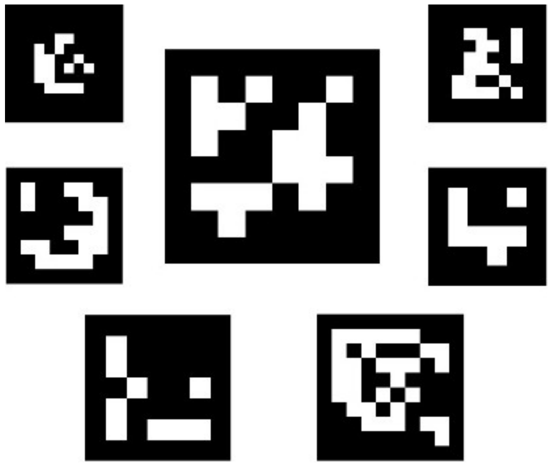


Figure 2.10: Examples of ArUco Markers [27]

2.10.4 ArUco Detection

Given an image with several ArUco markers, the detection software has to return a list of detected markers. Each detected marker displays the position of the four corners in the image and the marker ID.

The marker detection consists of two steps:

- Detection of marker candidates. The image is analyzed and square shaped marker candidates is detected. The detection process starts with an adaptive image thresholding, to segment the markers. The contours are extracted and shapes that does not approximate a square is discarded. Additional filtering is applied to remove contours to close to each other and to big or to small contours.
- When the detection of the square shapes is complete, the software has to determine if the shape actually is a marker by analyzing the inner codification. The step begins with extracting the marker bits of each marker. The extraction starts with a perspective transformation to obtain the marker in its canonical form. Otsu's method is applied to separate white and black bits. The image is divided into cells depending on the marker size and the amount of black or white pixels is counted to determine if the cell is a black or white bit. The bits are analyzed in order to determine if the marker exists in the ArUco library. Then the algorithm calculates the pose of the marker relative to the camera in 6DOF.

2.10.5 Odometry

Odometry is the use of motion data to estimate the change in position over time. Data from the motor count and the steering servo position is fed through the kinematic equations. An approximation of the robot pose is calculated by repeatedly computing the distance moved and the change in direction. The odometry is only valid in small time windows due to accumulating errors from the integration of the velocity. However, the odometry can be fused together with IMU measurements and GPS data and used over small intervals to increase the accuracy and redundancy of the pose estimate. The odometry equations listed below are based on the Ackermann kinematics from Section 2.4.1. The tangential velocity is calculated in Equation (2.11):

$$v_s = \frac{m_{vel}}{i} \cdot r \tag{2.11}$$

Where:
i: Gearbox ratio
r: Wheel radius [m]
v_s: Tangential velocity [m/s]
m_{vel}: Motor velocity [rad/s]

The velocity in local x-direction is calculated in Equation (2.12):

$$\dot{x}_l = v_s \cdot cos(\phi) \tag{2.12}$$

Where:
$\dot{x}_l$: Local velocity in x-direction [m/s]
ϕ: Steering angle of centre wheel [rad]
v_s: Tangential velocity [m/s]

There is assumed no slip in y-direction, thus $\dot{y}_l = 0$. The change in heading is calculated in Equation (2.13):

$$\dot{\theta} = \frac{tan(\phi)}{L} \cdot \dot{x}_l \tag{2.13}$$

Where:
L: Length of wheel base [m]
$\dot{x}_l$: Local velocity in x-direction [m/s]
ϕ: Steering angle of centre wheel [rad]
$\dot{\theta}$: Global change in angle, relative to origin [rad]

The global velocity in x-direction is calculated i Equation (2.14):

$$\dot{x}_g = \dot{x}_l \cdot cos(\theta) \tag{2.14}$$

Where:
θ: Global heading angle [rad]
$\dot{x}_l$: Local velocity in x-direction [m/s]
$\dot{x_g}$: Global velocity in x-direction [m/s]

The global velocity in y-direction is calculated in Equation (2.15):

$$\dot{y}_g = \dot{x}_l \cdot sin(\theta) \tag{2.15}$$

Where:
θ: Global heading angle [rad]
$\dot{x}_l$: Local velocity in x-direction [m/s]
$\dot{y_g}$: Global velocity in y-direction [m/s]

2.11 Kalman Filter

A Kalman filter is an algorithm that predicts future state of a system based on the previous states. A series of observed measurements containing statistical noise are feed into the filter that outputs an estimate of the unknown variables [28]. The joint probability distribution is calculated for each variable at each time frame.

There are numerous control applications for the Kalman filter like guidance and navigation of vehicles. The algorithm functions in two main steps, the prediction step and the update step. In the prediction step, the filter makes an estimate of the current state and their uncertainties. Then the measurement of the sensors is observed by the filter with a certain amount of error, including noise. The estimate is then updated with a weighted average. The estimate with highest certainty receives the highest weight [29].

The Kalman filter is recursive, which means that it uses one or more of its outputs as an input in a feedback loop. The filter functions in real time and can handle time delays and discrete signals by using the present input and the previous state and its uncertainty matrix, no additional past information is required.

2.11.1 Example GPS Application with Kalman Filter

Consider the problem of localizing the reach stacker container handler on a container port. The reach stacker is equipped with a GPS sensor, to determine its position. The GPS sensor signal contains noise with values jumping around with an error of several meters from the actual position [29]. There is need for additional data input, so encoders are equipped on the wheels to use odometry for dead reckoning. The dead reckoning provides a smooth signal, but it drifts over time. The Kalman filter is implemented to predict and update the position.

The reach stacker has an old position which is modified in accordance with the kinematics of the vehicle. A new position is predicted with an additional new covariance. The covariance might be proportional to the velocity of the vehicle, a higher speed results in bigger position errors due to for instance wheel slip. Next is the updating phase where the GPS position is obtained, with a certain uncertainty. The GPS signal covariance is relative to the previous phase and it affects how much the new measurement impacts the updated prediction. In an ideal case the odometry drifts and is updated by the GPS estimate that pulls the estimate back towards the actual position without disturbing to a point where the position estimate becomes noisy and jumps around.

2.11.2 Covariance Matrix

Any robot using some kind of sensor fusion needs to know the accuracy of the sensor data in order to weigh the data in a proper manner. There is two types of errors affecting the accuracy of data: systematic and non-systematic errors. Systematic errors do not depend on the environment surrounding the robot and may for an instance originate from a bias in the IMU data. A non-systematic error depends on the environment and changes dramatically with changing environment.

The non-systematic errors are expressed in terms of the covariance matrix. The diagonal values are variances and all the other values are covariances. The variance is calculated using the formula in Equation (2.16):

$$\sigma_x^2 = \frac{\sum_{i=1}^{N} \left(\overline{x} - x_i\right)^2}{N} \tag{2.16}$$

Where:
N: Number of measurements
$\overline{x}$: Average of measurements

The covariance is calculated using the formula in Equation (2.17):

$$\sigma_x \sigma_y = \frac{\sum_{i=1}^{N} \left(\overline{x} - x_i\right)\left(\overline{y} - y_i\right)}{N} \tag{2.17}$$

An arbitrary covariance matrix will look like the matrix below:

$$\begin{bmatrix} \sigma_x^2 & \sigma_x \sigma_y \\ \sigma_y \sigma_x & \sigma_y^2 \end{bmatrix} \tag{2.18}$$

2.12 Simultaneous Localization and Mapping

Simultaneous localization and mapping or *SLAM*, is the problem of constructing and updating a map while keeping track of the robots position within the map. It is a search-based approach where the robot moves around and explores the surroundings while mapping the surrounding landmarks. The distance to surrounding landmarks is measured either by a LiDAR or a stereo camera. The most frequently used SLAM-packages in ROS is: `hector_slam` and `gmapping`.

The required computational power of SLAM is quite high; however it is reduced by mapping in 2D and by implementing odometry data and IMU measurements to estimate the motion of the LiDAR. In SLAM a single point consists of a pose and a map.

2.12.1 Particle Filter SLAM

Both `gmapping` and `hector_slam` utilizes Rao Blackwellized particle filter with scan matching, also known as the grid map based fast SLAM algorithm [30]. The `gmapping` algorithm requires odometry data to solve grid-based SLAM. The particle filter is used to calculate the trajectory of the robot and the map is based on observations from the LiDAR and odometry. The algorithm works in two steps; First the trajectory of the robot is calculated from odometry and LiDAR data. Then the map is computed since the posterior trajectory and observations are known [31].

Each particle represents a potential trajectory. An individual map is calculated for each particle. The particle with highest probability is chosen as a reference and the associated map is outputted by the algorithm. Scan matching is implemented to match observations with the map constructed in the previous position, thus providing the most likely pose of the robot.

The `hector_slam`-package in ROS, applies the Gauss-Newton approach before the scan matching is conducted [32]. The approach presents the measurement as Gaussian distributions, thus ”smothering” the sampled data and generating a map with less noise. Fast scan matching enables it to function without odometry data. This is practical for robots which can not provide odometry data or have inaccurate odometry measurements. The algorithm requires a high update rate to accommodate the gaps in data, due to missing odometry. A grid map discrete the observed surroundings into an occupancy grid map, with a threshold that marks the cells either as occupied or free. Each cell is given a value between 0 and 1, depending on the probability for it being occupied. This approach decides the localization of the robot iterative over time.

A disadvantage of `hector_slam` is that it has poor performance in areas without many distinct landmarks [31]. `hector_slam` does not provide any explicit loop closing abilities, however the algorithm manages to close the loop in many robot applications.

2.13 Navigation Stack

The basic concept of the *navigation stack* is that it takes information from the robots odometry, sensor streams and a goal pose and outputs a velocity command which is sent to the mobile base [33].

2.13.1 Navigation Stack Setup

In ROS, the navigation of a mobile base is handled by the *navigation stack*. The navigation stack has support for navigation in three dimensions, however this thesis is based on navigation in two dimensions. As pre-requisite, the robot must have a `tf`-compliant transform tree, publishing sensor data using correct ROS-message types, in addition to be configured for the shape and dynamics of the robot. Figure 2.11 show a high-level view of the navigation stack.

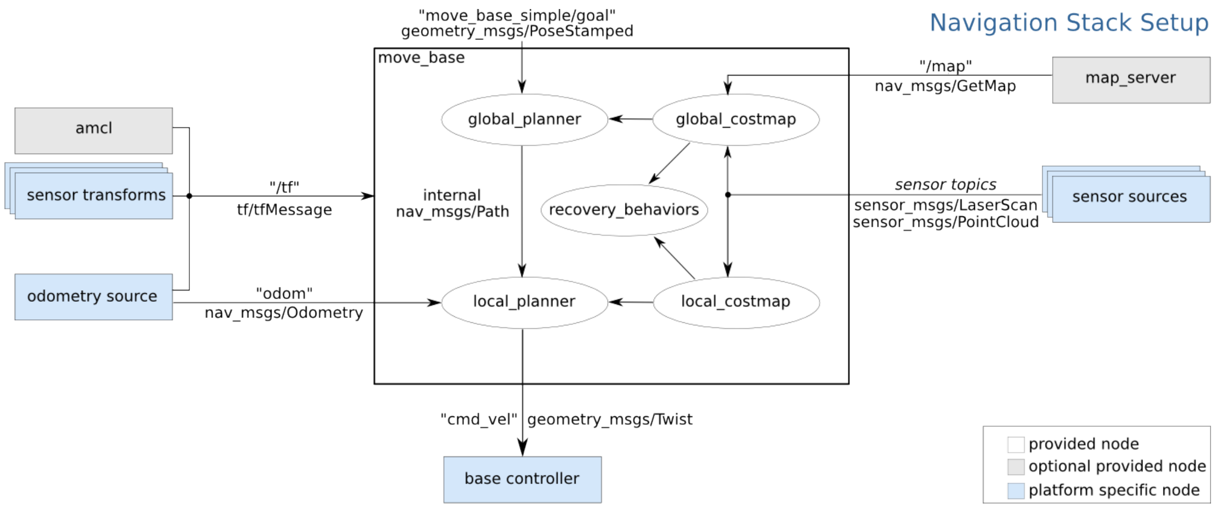


Figure 2.11: Navigation Stack High-Level View [34]

2.13.2 Transform Configuration

The transform configurations in ROS is handled by the tf-package, which keep track of multiple coordinate frames relative to a base frame. The tf-package stores the relationship of frames in a tree structure which makes it simple to get an overview of the system [35].

Figure 2.12 illustrates an example of a transform tree. The transform tree shows that the transformation between the map-frame and odom-frame is handled by the amcl-node and the transformation between the odom-frame and base_link-frame is handled by the ackermann_odometry-node.

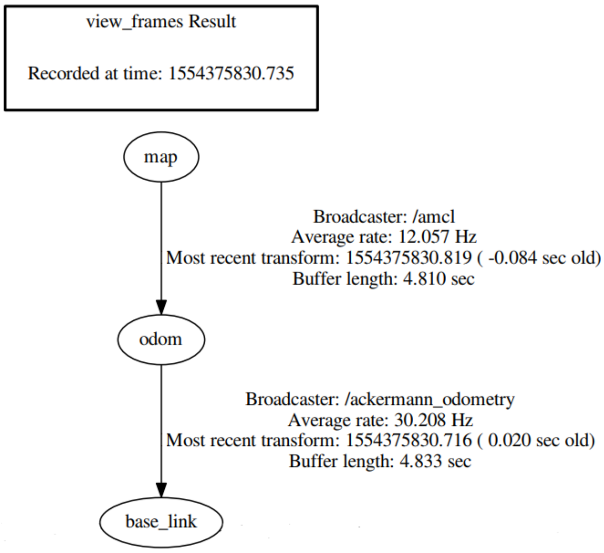


Figure 2.12: Example of tf-tree

2.13.3 Move Base

The standard way of moving a mobile base in ROS, is through the move_base-node [34]. As seen in Figure 2.11, the move_base-node receives both static and dynamic input transforms as well as odometry, sensor and map inputs. Move base uses the global and local planner in addition to the costmap parameters to calculate a velocity command which is sent to the base controller through the /cmd_vel-topic.

2.13.4 Occupancy Grid

The SLAM generated map is represented as an occupancy grid. The map is divided into small cells which are labeled as either undiscovered, walkable or occupied. The resolution could be altered by changing the dimension of each cell. High-resolution SLAM results in a computational heavy map. Figure 2.13 show an occupancy grid generated with SLAM. The walkable areas are represented by white, black is occupied and grey is unknown.

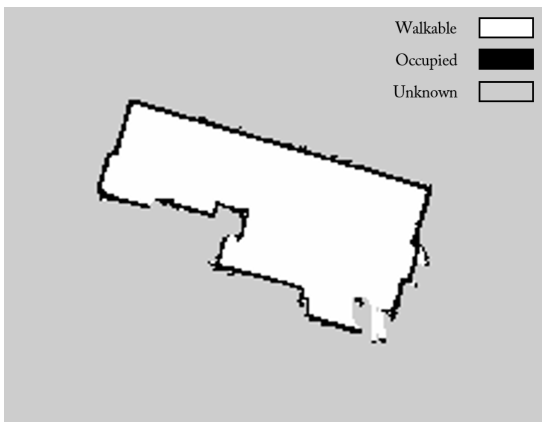


Figure 2.13: SLAM Generated Occupancy Grid

2.13.5 Costmap

The costmap is placed as a layer above the occupancy grid map and contains information about obstacles in the environment. Sensor data is obtained from the LiDAR and odometry and obstacles are inflated to a size equal to the inscribed radius of the robot. The robot is therefore configured to never cross the inflated area with the centre of the robot. The costmap subscribes to sensor data topics and updates itself automatically. The input data is used to insert obstacles to the map or clear obstacles. Figure 2.14 show a costmap overlay on the previously shown occupancy grid.

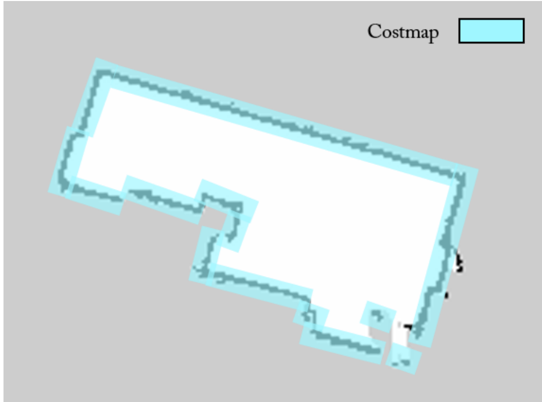


Figure 2.14: Costmap Overlay on Occupancy Grid

2.13.6 Obstacle Avoidance

Obstacle avoidance is achieved as a part of the overall trajectory optimization. Trajectory optimization is concerned with finding the minimum cost trajection, thus avoiding obstacles which have a high cost in the map. Ideally the cost value should be infinite, however this would require optimizing *hard constraints*. A better solution is to use soft constraint with a quadratic penalty term ensuring a finite cost [36]. Figure 2.15 shows an example of a penalty term with a minimum allowed distance to an obstacle set to 0.2 meters.

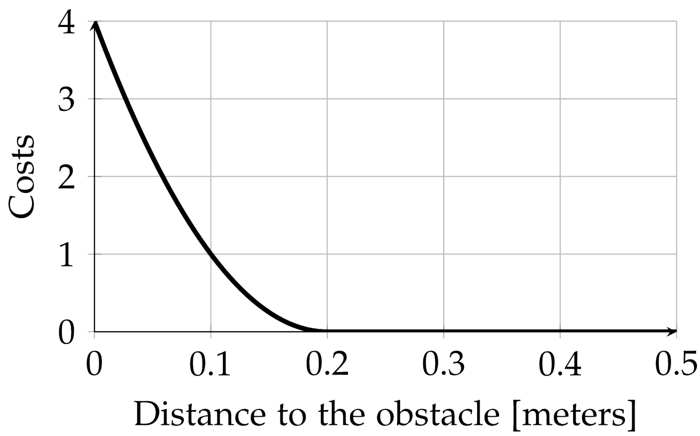


Figure 2.15: Exemplary Penalty Graph [36]

A typical discrete trajectory is composed of multiple robot poses over time. The planner arranges each consecutive pose according to the discretization interval. To avoid the obstacle the distance between the planned pose and obstacle has to be found. Figure 2.16 illustrates an example where the trajectory consists of eight poses and the discretization interval of `dt_ref`.

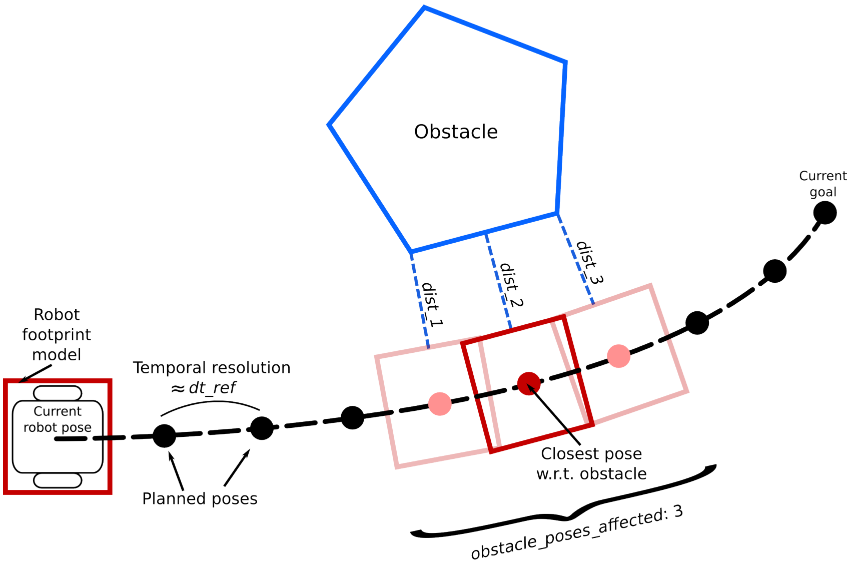


Figure 2.16: Example of Robot Trajectory Around Obstacle [36]

The trajectory optimization places the poses on the planned trajectory closest to the obstacle. This only applies to a subset of poses affected by the obstacle, in this case three. The other poses are placed by the global planner.

2.13.7 Adaptive Monte Carlo Localization

Adaptive Monte Carlo Localization, or AMCL, is a localization algorithm that track the pose of the robot. The approach uses a particle filter to track the pose within a known map. The map is created priorly by the SLAM algorithm. The `amcl`-node in ROS requires a laser-based map, LiDAR data and a `tf`-message to output a pose estimate.

When the `amcl`-node is started, it initializes a particle filter according to the provided parameters. During movement the algorithm resamples and try to estimate the pose by using Bayesian estimation, where the particles are compared to the features of the map. The pose uncertainty is large during the first scans, and is visualized as a cloud of vectors in RViz. The bigger the uncertainty, the bigger the vector cloud appears. During movement the algorithm resamples, shifts the particles and predicts the new state. When the surroundings are recognized by the filter, the pose uncertainty decreases.

2.13.8 Dynamic Window Approach

The *Dynamic Window Approach* (DWA) is the algorithm utilized by the default local planner in ROS [37]. If nothing is specified in the `move_base`-node, DWA will be utilized. The `dwa_local_planner` provides a controller that executes local path planning for a mobile base. The algorithm utilizes a map and a global plan to generate a local kinematic trajectory for the robot. A value function is created locally around the robot represented as a grid map and the cost for traversing through the grid cells.

The basic function of the DWA-algorithm is as follows [37]:

- Discretely sample (dx, dy, $d\theta$) in the robot's control space.
- For each sampled velocity, perform forward simulation from the robot's current state to predict what would happen if the sampled velocity were applied for a short period of time.
- Evaluate each trajectory resulting from the forward simulation by: proximity to obstacles, proximity to the goal, proximity to the global path, and speed. Discard illegal trajectories (those that collide with obstacles).
- Pick the highest-scoring trajectory and send the associated velocity command to the mobile base.
- Clear memory and repeat.

2.13.9 Time-Elastic-Band

The `teb_local_planner` [38] is a 2D local path planner utilizing the *time-elastic-band* (TEB) algorithm to generate a local path. The global path planner dictates the trajectory and goal pose of the robot and the local planner optimises the trajectory during the movement [39]. The trajectory is optimised in order to avoid obstacles in highly dynamic environments like for an instance a container port. The algorithm improves trajectory execution time and move with compliance to the kinodynamic constraints. The `teb_local_planner` is meant for nonholonomic vehicles. Figure 4.24 show an example of a TEB generated path around three simulated obstacles in RViz.

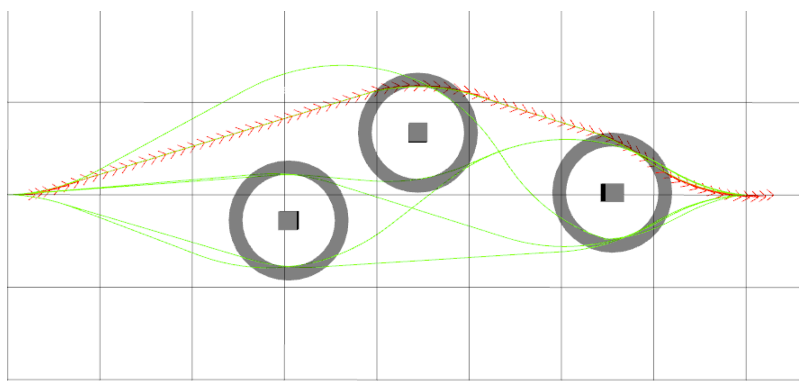


Figure 2.17: Simulated `teb_local_planner` with Several Obstacles

The planner reduces the required computing power by restricting the search space to an optimal local area [40]. However, the path is usually non-convex, meaning that there exist several path options due to the presence of obstacles. The `teb_local_planner` adhere to the navigation stack by providing `Twist`-messages containing translational and angular velocity.

It is worth noticing that an angular velocity equal to zero $\omega = 0$ results in an infinite turning radius r which leads to a zero in steering angle $\phi = 0$. For a non-zero angular velocity the turning radius r is computed by $r = v/\omega$, the steering angle is derived by $\phi = tan^{-1}(wheelbase/r)$. The steering angle is therefore not defined for zero velocity. The `teb_local_planner` deals with the problem by setting the steering angle to zero by default if the linear velocity is zero.

3. Methods

This chapter elaborate for the methodology used to obtain the results presented in this thesis.

3.1 V-Model Approach

The V-Model is used to develop the software system. Figure 3.1 show an illustration of a typical V-Model.

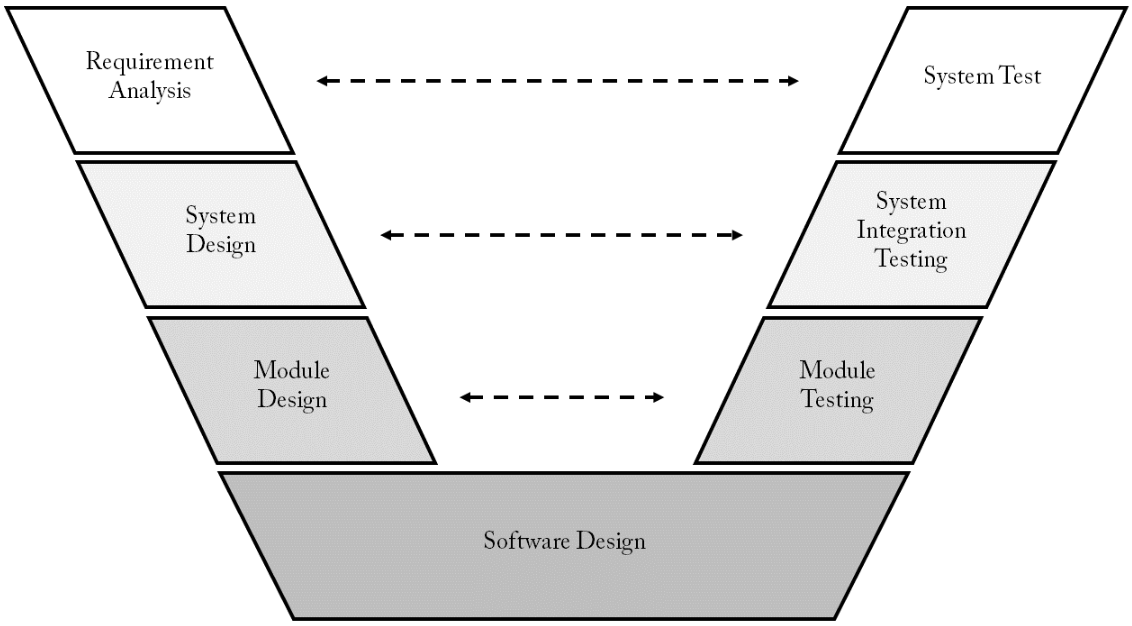


Figure 3.1: V-Model

The V-Model is supposed to follow a chronological order. The module design and software design however were carried out as a joint effort. Some modules where added in a later stage of the project, this demonstrates the modular capabilities of Robot Operating System (ROS).

The software is built in a universal modular manner, enabling fast implementation on different hardware. The software functions on a small prototype as well as a full-scale reach stacker. The universal design allows for reuse of code and further development by third party developers [41].

3.1.1 Requirement Analysis

The requirement analysis is the start of the project. Red Rock requested the features they wanted in the software and the prototype. It was important with detailed communication to understand the ”costumers” requirement and expectations. The software must enable autonomous driving and be scalable, meaning that the software could be implemented on a scale prototype as well as the full-scale container handler. The prototype has to navigate autonomously, avoid obstacles, map the dynamic environment and update the map. The prototype hardware was researched and purchased early i order to start development rapidly.

3.1.2 System Design

The system overview was developed in the design phase. It was decided to use ROS Kinetic Kame as the overall operating system. The simplest way for ROS to communicate with the hardware was via USB-ports. WiFi was utilized to enable communication with the stationary computers handling the tracking system. WiFi was also used to connect to the prototype and to perform manual control or changes in software during testing. The system design is explained further in Section 4.1.

3.1.3 Module Design

The software is structured in modules. Some modules were pre-developed by members of the ROS community while other modules was developed during this thesis. Each module can be isolated and tested. The modules could also be implemented in other compatible system. The pre-developed ROS modules are listed in Table 3.1 in Section 3.4.3.

3.1.4 Software Design

The software design phase stitches every module together to a functioning software suite. ROS has specific guidelines and standards on communication and data processing. By following the standard in each individual module, implementation to a full software suite was carried out without any major problems. The software suite was revised several times and optimized for better performance. Some modules where added in post.

3.1.5 Module Testing

The hardware was bought early and available at the start of the project. The ROS compatibility of each hardware component was tested before implementation. Every module was tested before implementation in the overall software. Incompatible software or hardware was eliminated at an early stage and troubleshooting was made less cumbersome.

3.1.6 System Integration Testing

The system integration testing revolves around communication and coexistence. The ROS communication functions flawlessly due to the correct topics and message types being used. Some USB connections experienced coexistence due to the hardware being routed through a USB-hub into the single USB-port on the Jetson. This caused some minor issues.

3.1.7 System Testing

The system testing is associated with the system design phase in the beginning of the project. This phase tests the entire system at once, or at least a collection of the modules functioning together. It also unveils problems that is related to external factors.

3.2 Sensor Package

The full-scale reach stacker must be equipped with a perception system that enables it to function autonomously. The sensors are used to gather information about the vehicle and the surrounding environment. The sensors are divided into two categories:

- Proprioceptive sensors is responsible for sensing the vehicle's internal states like inertial measurement unit and wheel encoders.
- Exteroceptive sensors are responsible for sensing the surrounding states like LiDAR, cameras, ultrasonic and RADAR.

There are two sensor packages on the reach stacker. The first package is used to drive the vehicle around and positioning it relative to a container that is supposed to be moved. The second package is used to position the container spreader directly above the container to enable lifting. This thesis will focus on the vehicle positioning-package.

3.2.1 Vehicle Positioning-Package

The vehicle positioning-package is equipped with seven different sensors: LiDAR, encoders, GPS, stereo camera, IMU, compass and ultrasonic sensors. Figure 3.2 show the layout of the sensor package. Please note that the range and spread of the sensors are just for illustration.

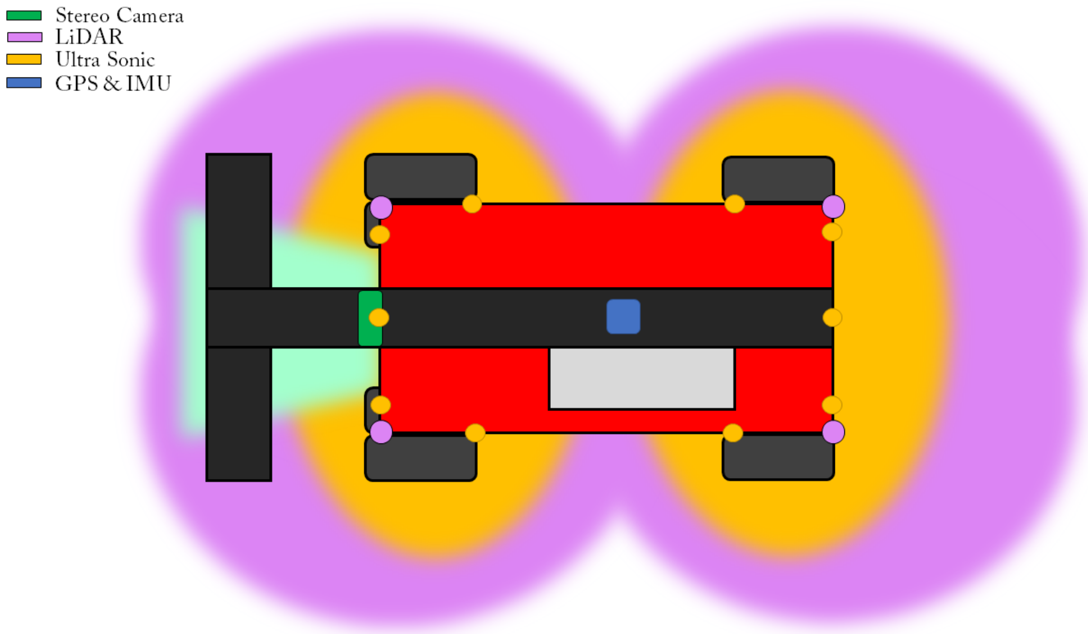


Figure 3.2: Vehicle Positioning-Package

The LiDAR are situated on each of the four corners of the reach stacker with 180 degrees line of sight. The LiDAR point cloud is stitched together to give a two-dimensional picture of the objects surrounding the vehicle.

Encoders will measure the steering angle, which is used in the kinematic equations to calculate the pose and velocity of the vehicle. In addition, there will be mounted one encoder for each wheel in order to measure the odometry more accurate.

An IMU will be placed at the centre of rotation of the reach stacker. The IMU is used for pose-estimation and localization. The IMU contains accelerometers, gyros and a compass. The compass outputs a fixed heading and is therefore useful to find the orientation of the vehicle.

The GPS provides an absolute measurement of position and is useful to pinpoint the position of the vehicle and the velocity.

The stereo camera is situated in the front of the reach stacker and is used in several applications. The camera detects humans and other obstacles. The camera can detect containers and read the serial number in order to verify that the container is the one that is supposed to be transported. The camera can also detect ArUco markers strategically placed around the container port to verify that the pose estimate is correct. The container port could be configured with a grid of lines the reach stacker could use camera vision to follow. The grid could be used to improve accuracy and redundancy of the system.

Road markings are not visible under a layer of snow and ice or could be worn out or covered by dust. This problem could be solved by a magnetic strip embedded in the asphalt. A magnetic sensor could then be used to follow the line or sense information about an area.

Ten ultrasound sensors are strategically placed around the reach stacker and functions as an electrical bumper. The ultrasound sensors are useful at low speeds and detect if any object comes within a certain range. The sensors are practical during thigh manoeuvres and during lifting procedures.

3.2.2 Container Spreader Package

The second sensor package is placed on the container spreader and gives feedback on the position relative to the container. The package consists of two cameras and two ultrasound sensors. The cameras are mounted on the diagonal on each side of the spreader. The two ultrasound sensors are situated on each side of the spreader. Figure 3.3 show an illustration of where the sensors could be placed.

The cameras look down on the container and detects two intersecting edges on each side. The spreader adjusts to the correct length and position. The ultrasound sensors measure the distance between the spreader and the container. A height difference between the two ultrasound sensors would originate from an angle offset. The spreader is tilted until the distance offset is zero, and the boom is lowered until the container is locked in position.

Figure 3.3: Container Spreader-Package

3.2.3 Down-Scaled Sensor Package

A down-scaled sensor package was created for prototyping purposes due to both restrictions in space and budget. The package consists of a LiDAR, depth camera, IMU and ultrasonic sensors. In addition, there is situated an ArUco marker on top of the prototype, in order to track the position. A voltmeter with an embedded buzzer is connected to the main LiPo battery in order to measure the voltage on each battery cell.

The LiDAR is placed in the front of the prototype and has an approximate 230 degree view angle, due to the laser beam being obstructed by the truck frame. The objects obstructing the LiDAR is not within the LiDAR range and is there for not visible on the scan. The main application of the LiDAR is obstacle avoidance and mapping.

For redundancy, a stereo camera could be mounted in the front of the prototype to provide additional inputs to the obstacle avoidance. It could also be used in navigation and object recognition.

An IMU is utilized to further improve the localization. The IMU uses three gyroscopes and three accelerometers to adjust the position estimate, the magnetometers are not used due to magnetic interference. The IMU is placed in the vehicle centre in a horizontal position.

The ultrasonic sensor could be situated at the rear bumper of the prototype and functions as an electric bumper. The sensor would sense a possible collision when performing a K-turn or if the prototype gets lost.

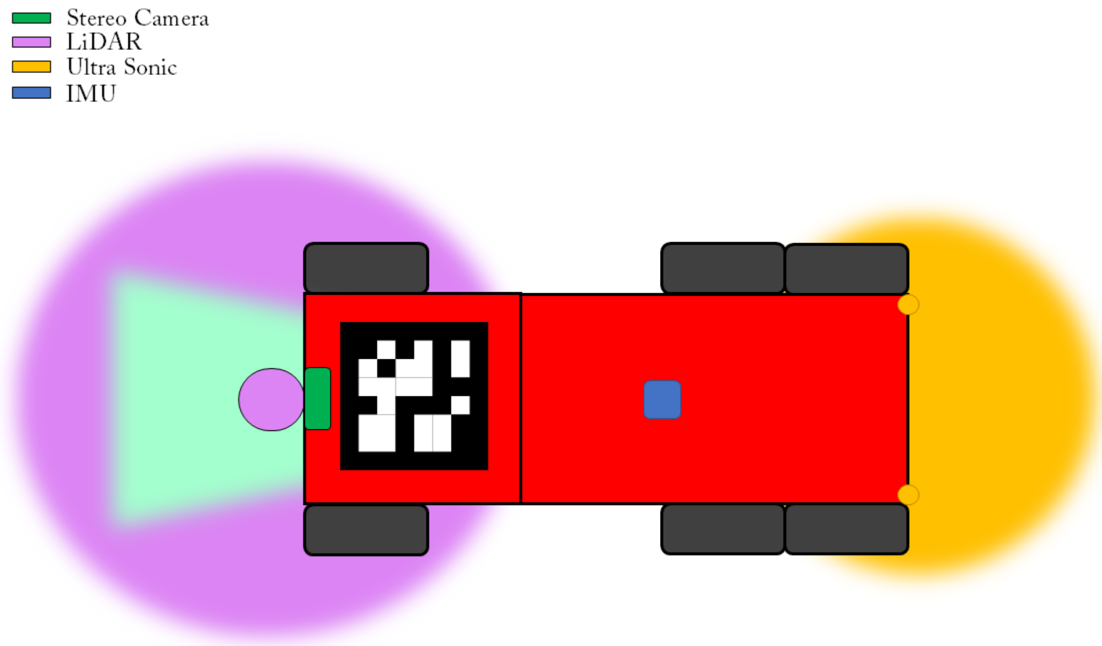


Figure 3.4: Down-Scaled Sensor Package

3.3 Hardware Setup

This section contains information about the prototype hardware. The full-scale reach stacker has to rely on industry grade hardware, however the components selected in this section is meant for prototyping and indoor robotics.

Red Rock requested a 1:14 scale model of a reach stacker. This would obviously be the best platform to use in a prototype. However, the manufacturer of the reach stacker model had ceased production and it was not possible to buy the model, neither new or used. The second-best option was then to buy a 1:14 scale truck with the possibility to attach a trailer that could carry a 40 feet container.

3.3.1 NVIDIA Jetson TX2 Developer Kit

The Jetson TX2 Developer Kit provides a fast and easy way to develop software and test it on the desired hardware. It is ideal for deep learning; computer vision and GPU computing [42]. Figure 3.5 shows an image of the Jetson developer kit.

Figure 3.5: NVIDIA Jetson TX2 Development Kit [42]

3.3.2 SLAMTEC RPLiDAR A3

SLAMTEC's RPLiDAR A3 is an ultra-thin 2D LiDAR designed both for indoor and outdoor applications. With its 16000 samples per second and 25 meter range, it is accurate and versatile [43]. The LiDAR is mounted in front of the robot in order to view the environment ahead. Figure 3.6 shows an image of the LiDAR.

Figure 3.6: SLAMTEC RPLiDAR A3 [43]

3.3.3 Intel RealSense D435 & D435i

Intel® RealSense™ Depth Camera D435i is a depth sensing camera with the addition of an inertial measurement unit (IMU). In ROS, the IMU input is used to improve dead reckoning accuracy. The RealSense camera has the ability to measure distance due to the two IR image sensors. The active IR emitter projects a coherent pattern of points and the distance is measured by triangulation [44]. There will be one camera unit without IMU, fixed in place to detect the ArUco marker placed on the robot. In addition, the prototype has a camera with IMU mounted in front, which is meant for future applications. Figure 3.7 shows an image of the RealSense camera.

Figure 3.7: Intel® RealSense™ Depth Camera D435 [44]

3.3.4 Tamiya RC Truck

The prototype is built on the base of a Tamiya 1:14 scale RC Mercedes Benz truck. The kit was assembled without the drivers cabin and a platform was 3D printed and mounted to the frame. The platform has enough room to attach the Jetson TX2 developer board and several different sensors and other electrical components. The truck has a 3-stage gearbox and servo driven Ackermann steering. The truck is driven in second gear, with a gear ratio of 17.761:1. Figure 3.8 shows the Tamiya RC truck with the drivers cabin.

Figure 3.8: Tamiya Mercedes-Benz Actros 3363 [45]

3.3.5 VESC

A Vedders Electronic Speed Controller (VESC) is used to control the drive train of the prototype. The VESC is an advanced open-sourced ESC [46]. The VESC is connected directly to the Jetson with an USB-cable. The ROS community has developed several different pre-built packages, to configure the VESC for robot applications.

Figure 3.9: VESC [47]

3.3.6 SkyRC BLDC Motor

The original brush motor was replaced with a SkyRC BLDC brushless motor [48] in order to control the drive train with a VESC. Figure 3.10 shows an image of the SkyRC BLDC. The motor is a three-phase motor with 2 poles, the motor will have six poles since the pole pattern have to be repeated for each phase.

Figure 3.10: SkyRC Ares Pro V2 Competition 540, BLDC [49]

3.3.7 Power HD-9001MG Servo Motor

A Power HD-9001MG servo motor is utilized to steer the prototype. This servo is commonly used in RC vehicles and aeroplanes and operate at a voltage between 5-6V. The no load velocity at 5V is 60°/0.140 sec. The servo is able to rotate ±90° and has a holding torque of 0.96 Nm.

Figure 3.11: Power HD-9001MG Servo Motor [50]

3.3.8 Arduino Uno

Arduino in an open-source electronics platform which utilized its own Arduino programming language based on Processing and *C++*. The Arduino board is able to read sensor inputs and write outputs. The Arduino Uno is based on the ATmega328P single chip micro-controller and is powered by an USB-cable. Figure 3.12 show an illustration of the Arduino Uno.

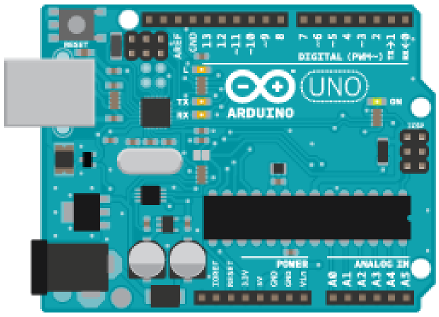


Figure 3.12: Arduino Uno [51]

3.3.9 Adafruit Servo Driver

The Adafruit servo driver uses I2C to communicate from the Arduino to the servo motor. It uses an external power supply to power up-to 12 servo motors, however for the setup in this thesis, it only powers the steering servo.

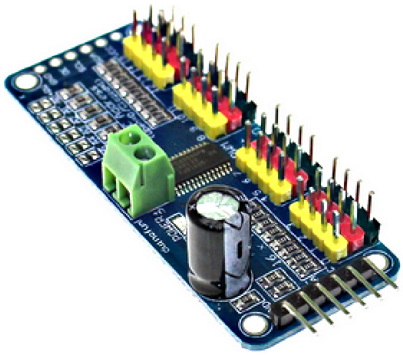


Figure 3.13: Adafruit Servo Driver [52]

3.3.10 SparkFun 9DOF Razor IMU

SparkFun 9DOF Razor IMU M0 was selected as IMU for the prototype, due to the USB connection. The IMU has 9DOF obtained from 3 accelerometers, 3 gyroscopes and 3 magnetometers placed orthogonal in relation to each other. The board has a small Atmel SAMD21 Arduino-compatible 32-bit micro controller integrated. The IMU is pre-programmed with a Arduino bootloader and the core had to be flashed with an Arduino program to enable ROS communication. Figure 3.14 show an image of the IMU. The IMU's data sheet is attached in Appendix E.5.

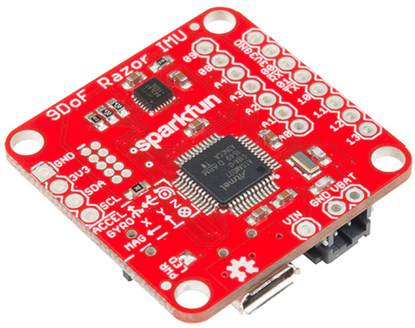


Figure 3.14: SparkFun 9DOF Razor IMU M0 [53]

3.3.11 Prototype Build

A 1:14 scale Tamiya RC truck was built as prototype for testing hardware and software. The truck came as a kit and was assembled using the provided construction manual [54] (see reference link for full construction manual). The cabin of the truck was removed and replaced with a 3D printed platform (CAD drawing in Appendix E.7) to house the batteries, NVIDIA Jetson, LiDAR, VESC, Intel RealSense, Arduino board, servo driver, IMU, a USB-hub and the ArUco marker on top. The final CAD model and physical model is displayed in Figure 3.15 and Figure 3.16.

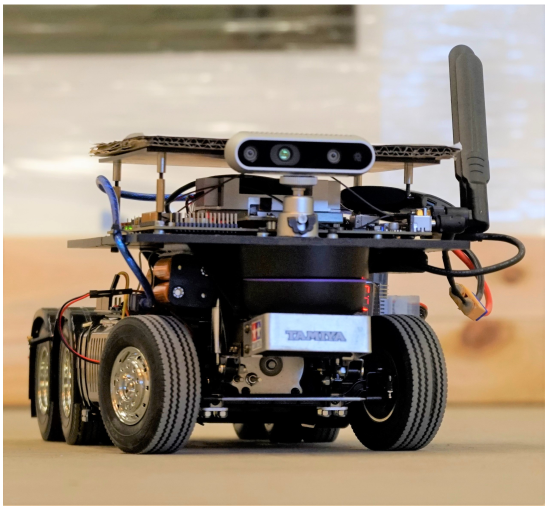


Figure 3.15: Photo of Prototype

The RPLiDAR is mounted upside down under the 3D printed platform and a stereo camera is situated on top of the platform on a ball head. The Jetson developer board is situated behind the camera, with an ArUco marker placed above it. The USB dongle and all the batteries are mounted underneath the platform. A bumper was mad from aluminium and mounted to the frame in order to shield the LiDAR in a collision. The IMU was mounted in the middle of the truck. There are attached more photos of the prototype in Appendix F.

Figure 3.16: CAD Render of Prototype

Electrical Connections

The USB-connections of the prototype is displayed in Figure 3.17. The Jetson TX2 has only one USB-port, thus all communication has to go through a USB-hub. The VESC and Jetson TX2 are powered by two different batteries and the rest of the hardware is powered by the USB-cable.

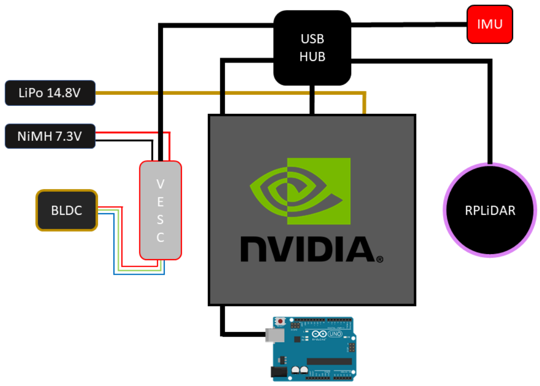


Figure 3.17: USB & Battery Connections

An Adafruit with an external power supply functions as the servo driver connected to the steering mechanism. The Adafruit is connected to the Arduino as illustrated in Figure 3.18.

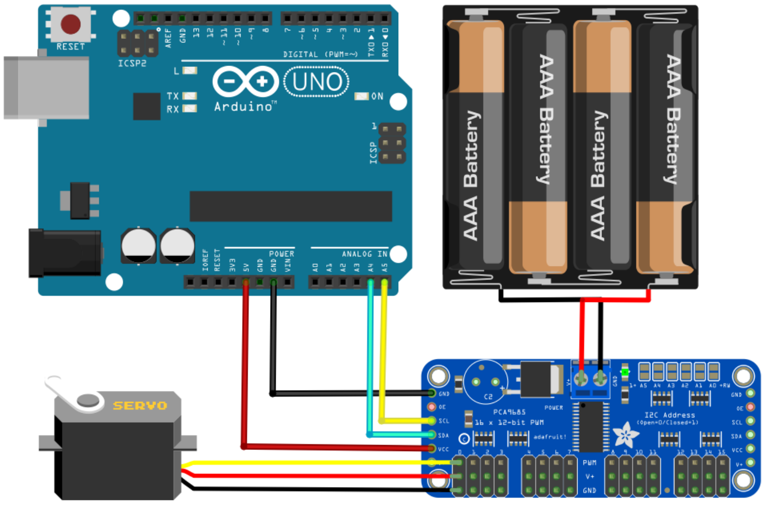


Figure 3.18: Arduino Circuit Diagram

3.4 Software

This section describes the software setup on the NVIDIA Jetson TX2 Developer Kit and the stationary computer(s) managing the ArUco detection. The software is developed on the bases of the prototype hardware, however there are several of the software components that could be utilized on a full-scale system due to a general structure and function of the codes.

3.4.1 JetPack

The Jetson JetPack 3.3 was flashed from a host computer running Ubuntu 16.04. JetPack includes the desired Ubuntu OS, CUDA graphics compiler, TensorFlow and OpenCV.

The installation was initialized by downloading JetPack 3.3 from NVIDIA's developer page [55]. The packages was downloaded and the kernel on the Jetson was flashed by following the installer guide.

3.4.2 OpenCV

Open Source Computer Vision Library, or *OpenCV*, is an open source computer vision software library. OpenCV was built to provide a common infrastructure for computer vision applications and accelerate the usage of machine perception [56].

More than 2500 optimized algorithms are included in the library, which includes state-of-the-art computer vision algorithms. These algorithms can be used to identify objects, track moving objects and pose estimation of markers. With its community of more than 47 000 users, information and codes supplements are easy to find.

OpenCV for Ubuntu with Python integration was installed by following the guide on the OpenCV installation page [57]. OpenCV is mainly used in the ArUco tracking, however it could be utilized by the on-board camera on the prototype in computer vision applications.

3.4.3 ROS

The ROS Melodic is the newest ROS release and was first used in the project, however the software caused several problems. The software did not have the same support as the older versions within the ROS community. The decision then fell on ROS Kinetic Kame, the reason for choosing Kinetic is that most available packages in the ROS community is supported. ROS Kinetic was installed as described on the ROS-installation wiki-page [58].

ROS Packages

Several ROS packages had to be downloaded and complied to be able to communicate with the different hardware. In addition a couple of packages was needed for the navigation and localization of the prototype. Table 3.1 shows a list of the packages downloaded in addition to a description of their function. The references in the end of the description is a link to the GitHub-repository the packages was cloned from.

Table 3.1: ROS Packages

Packaged	**Description**
`teleop_twist_keyboard`	Keyboard teleoperation control [59]
`rplidar_ros`	Communication with RPLiDAR A3 [60]
`vesc`	Communication with the VESC [61]
`rosserial`	Serial Communication with Arduino [62]
`razor_imu_9dof`	Communication with the Razor IMU [63]
`robot_localization`	Package used for localization in ROS [64]
`navigation`	Package for navigation in ROS [65]
`hector_slam`	Package containing `hector_slam` [66]
`teb_local_planner`	Package containing `teb_local_planer` [67]
`follow_waypoints`	Package for following waypoints in navigation [68]

ROS Package Installation

This subsection show an example of how to download a general ROS package and how to compile it.

The first step is to create a source folder in the work directory of the *catkin workspace*, utilized in all ROS applications:

```
$ mkdir <catkin workspace name>/src
```

Then change directory to the source folder and clone the GitHub-repository of the desired package:

```
$ git clone <GitHub-repository URL>
```

Lastly, the workspace has to be compiled with the new package(s) downloaded. To compile, run the following command in the root-directory of the workspace:

```
$ catkin_make
```

Wait for the compiler to finish. The package is now installed and ready to be utilized.

3.4.4 Intel RealSense SDK & intel-ros

The Intel RealSense cameras has to have *intel-ros* installed in order to be able to communicate with ROS. The installation was compleated by following the steps in the README-file on the GitHub-page [69]. The installation also include *librealsense* [70] which is the main software developer kit (SDK) needed to run the camera. The RealSense package was also installed on the stationary computer handling the localization.

3.5 Wireless Communication

The Jetson was configured as a WiFi hotspot in order to control and monitor the processes on the prototype, as well as enabling connection to several node computers. However, for the Jetson to act as a hotspot a parameter in the `/sys/module/bcmdhd/parameters/op_mode`-file had to be changed from `"0"` to `"2"`. A hotspot connection was created in the Ubuntu WiFi configuration. The hotspot configuration is useful when the operator wants to have manual control over the prototype and for the node computer to transmit the necessary localization data. ROS has a simple way of sharing messages and topics wirelessly.

The NVIDIA Jetson was set as a *ROS MASTER* by using the following command in the terminal:

```
$ export ROS_MASTER_URI=http://<MASTER_IP_ADDRESS>:11311
```

The master uniform resource identifier (URI) is set to the IP address of the Jetson as a safety measure. The URI prevent the system from stopping or losing control if the wireless communication to other ROS nodes are lost.

All the node computers in the ROS system will communicate with each other, through the master NVIDIA Jetson, by typing the command above in the terminal. Finally, all the computers have to export their own IP address using:

```
$ export ROS_IP=<COMPUTER_IP>
```

With this setup, only the master has to run *roscore* and all the topics generated on the node computers is accessible for every other node computers connected to the NVIDIA Jetson hotspot.

3.6 Manual Driving & Low-Level Control

The first step toward autonomous driving is to drive the prototype manually. A laptop was used to control the prototype using the ROS-package `teleop_twist_keyboard` [71]. The teleop-package is based upon the `Twist`-message in ROS, which is meant to control differential driven robots. The Ackermann steering on the prototype require its own *low-level control* to convert the `Twist`-message into duty-cycle controlling the VESC and an angle value sent to the steering servo.

The `lowlvlcontrol`-node subscribes to the `/cmd_vel`-topic which is the standard topic in ROS for velocity commands. The keyboard teleop-node published to the `/cmd_vel`-topic by default. `Twist`-messages have a command-signal ranging from -1 to 1, which means that the low-level control program have to convert the `Twist`-message into the range of the servo and VESC. The servo signal was converted using the formula in Equation (3.1).

$$C_{out} = C_{in} \cdot \frac{R_s}{2} + a \tag{3.1}$$

Where:
C_{out}: Servo command
R_s: Servo range
a: Centre value of servo
C_{in}: Twist message input

The converted servo command is published to the `/servo_cmd`-topic.

The VESC already have an input of ± 1, thus it do not require conversion, however when the deadband in the system was identified, the signal had to be compensated, thus the VESC commands was implemented in the low-level control.

3.7 Camera Calibration

The script listed in Appendix B.2 was used to remove radial and tangential distortion in the camera. The camera calibration is a pre-generated script included in the *ArUco Tacker*-package from *OpenCV*. The code was downloaded from GitHub [72].

The code utilizes input images in addition to information about the checker board's geometry, like grid size and size of the squares in the grid. The input images have to be taken from different angles so that the different types of distortion can be detected. Figure 3.19 show an example of an input image.

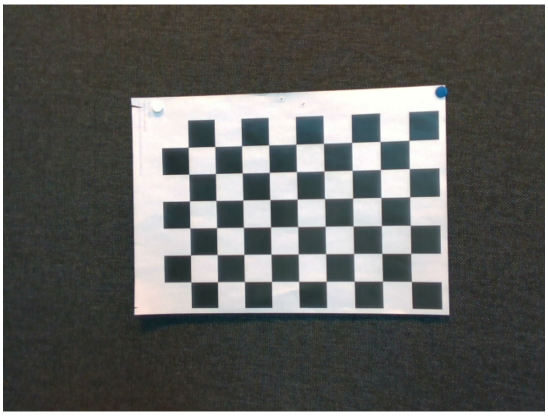


Figure 3.19: Example of Input Image for Camera Calibration

In order to find the pattern in the checker board the `cv.findChessboardCorners()`-function was used. The code inputs the grid dimensions, in this case a 9×6 grid. The function returns the corner points and a variable named: `retval`, which will be *True* if the board pattern was obtained. The corners are placed in an order from left-to-right, top-to-bottom.

Once the corners is located, their accuracy is increased using `cv.cornerSubPix()` and a pattern is drawn using `cv.drawChessboardCorners()`.

3.8 Localization

Localization is the problem of making a robot know its own position relative to a frame of reference.

3.8.1 Artificial GPS

The full-scale reach stacker is tracked with a GPS system. It is not possible to use GPS indoors, therefor the prototype have to relay on another kind of tracker to manoeuvre autonomously. The UiA Motion Lab has a Qualizys system consisting of 17 high frame rate infrared cameras. The cameras are capable of track retro reflective spheres with high accuracy. The idea was initially to create a similar tracking system using two infrared cameras and retro reflective spheres. Three spheres would be placed in a non-uniform triangle.

By using image segmentation, the spheres could be tracked by Hugh transformation and the distance to each sphere and the length between them could be calculated. Based on the information the pose of the vehicle could be calculated. The same system has frequently been built with Microsoft Kinect cameras. However, the production of Kinect camera has ceased, and another type of camera had to be selected. It was decided to use an Intel RealSense due to the low price, small size and the support within the ROS community.

The *librealsense*-package was downloaded and the RealSense camera was launched in ROS. When the IR video stream was displayed it was made clear that the RealSense camera projects infrared points rather than a continues infrared illumination. This proved to be a problem since the retro reflective spheres did not light up in the same way as in the continuous IR illumination originating from the Kinect camera. It was therefore impossible to locate the spheres.

Eventually the reflective spheres were abandoned and it was decided to utilize an ArUco tracking system to determine the pose of the prototype.

3.8.2 ArUco Detection

A python script was developed in order to detect the ArUco markers. The script initializes by calibrating the camera using the same images as described in Section 3.7. The scripts create a subscriber to the RGB camera-topic generated by the `ros-realsense` launch-file. Then the RGB image is converted into grey-scale and the algorithm search through the image looking for ArUco markers.

The marker detection is performed in the ArUco module with the `detectmarkers()` function. This function is the back bone of the module due to the fact that all the other functionality is based on the previously detected markers returned by the `detectmarkers()` function.

The parameters of the `detectmarkers()` function are;

- The first parameter is the input image containing markers.
- Second parameter the dictionary object.
- The third parameter is storing the detected markers in the `markerCorners` and `markerIds` structures. `markerCorners` is the list of corners on the detected markers. Four corners are returned for each marker, in their original order. `markerIds` is a list containing all the detected markers in the image.
- The fourth parameter is the object. This object contains all the parameters that are possible to customise.
- The fifth parameter, `rejectedCandidates`, is a list of marker candidates that did not contain a valid codification.

The `drawDetectedMarkers()` function serves as a method of visually inspect if the marker detection is functioning properly. The function displays a green square around the detected ArUco marker with an ID tag.

3.8.3 ArUco Pose Estimation

After the ArUco marker has been detected it is possible to obtain the marker pose using the corners of the detected marker in addition to the camera matrix and distortion coefficients. The `cameraMatrix` and `distcoeffs` are the camera parameters. The camera matrix consists of 3×3 elements with camera centre coordinates (intrinsic parameters) and focal distance. The distortion coefficients are a vector of five elements that models the camera distortion. Finally the corners are an output from the `detectMarkers()`-function. These values are then fed into the `estimatePoseSingleMarkers()`-function which outputs the `rvec` and `tvec`. These variables are then used to publish the ArUco marker's translation and rotation relative to the camera.

The full python script for both the detection and pose estimation is listed in Appendix B.3 and is based on a ArUco tracking script downloaded from GitHub [72].

3.8.4 Depth Camera

As an attempt to improve the distance measurements of the ArUco marker, the Intel RealSense's integrated depth camera was utilized.

The depth camera stream is published to a ROS-topic in the same way as the RGB camera. The pixel coordinates of the corners in the ArUco tracking script is used to create a dynamic *region of interest*, or ROI for short, in which only the ArUco marker is located. By creating the ROI, the depth measurement is only taken from the area where the marker is actually located, thus reducing measurement noise. To increase the accuracy further the average distance to an even smaller area within the ROI was used as the distance measurement. The script is listed in Appendix B.4.

3.8.5 Camera Placement

The camera is placed in a position overviewing the configuration space of the prototype. It is important that the camera cowers as much space as possible with an angle that enables tracking of the ArUco marker. The formulas shown in Equation (3.2), (3.3), (3.4) & (3.5) is used to calculate the camera angle, position and viewing area based on Figure 3.20 [73].

$$\theta_1 = \tan^{-1}\left(\frac{D}{H-h}\right) \tag{3.2}$$

$$\theta_2 = \tan^{-1}\left(\frac{d_2}{H}\right) \tag{3.3}$$

$$\theta_3 = \theta_1 - \theta_2 \tag{3.4}$$

$$d_2 = H \cdot \tan(\theta_2) \tag{3.5}$$

Several parameters could be decided, like height of camera H, camera vertical view angle θ_3 and target height h. By deciding the blind area d_2, it is possible to calculate the maximum view distance D.

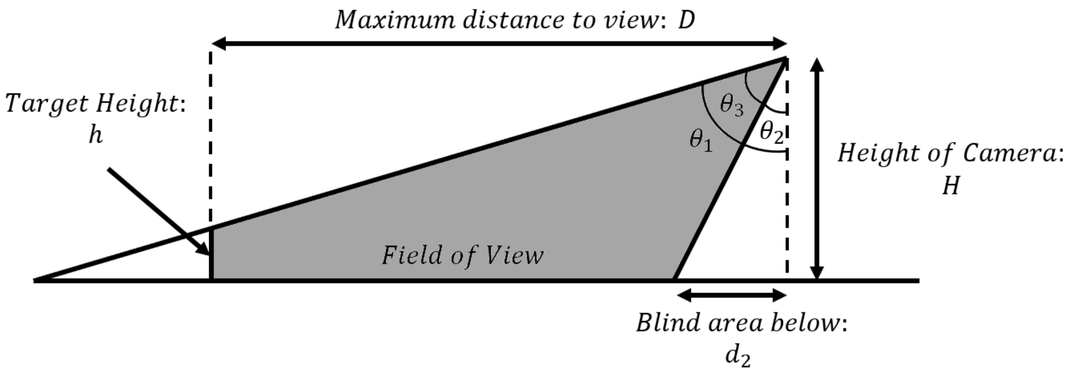


Figure 3.20: Camera Field of View

3.8.6 Odometry

To calculate the odometry of the Ackermann-model a script was created based on the equations from Section 2.10.5. The script subscribes to the measured motor speed and steering angle from the servo topics and published the calculated odometry to the `/odom`-topic. The script is listed in Appendix B.5.

3.8.7 IMU

The SparkFun Razor 9DOF IMU is built on an Arduino based developer board with an USB-connection. For ROS to recognize the IMU as an independent node, a special Arduino script had to be flashed onto the IMU. The installation setup procedure was completed by following the `README.md`-file on the GitHub-page [63].

The hardware version utilized in this thesis is the SparkFun "9DOF Razor IMU M0" version "SEN-14001". The version number had to be *uncommented* in the Arduino firmware-file to make the software compatible with the hardware.

3.9 Extended Kalman Filter Configuration

For the localization of the prototype described in Section 3.8, the different sensor data was combined using an extended Kalman filter in ROS named `ekf_localization` [74]. The EKF node combines the wheel odometry, ArUco pose and IMU measurements into a single odometry value. The extended Kalman filter is configured for 6DOF, however the prototype operates in 3DOF, this is resolved by zeroing out the inactive matrix values.

Appendix D.1 contains all the configuration parameters for the `ekf_localization`-node.

3.9.1 Wheel Odometry Covariance

To use the wheel odometry in the `ekf_localization`-node, the input has to be a `OdometryWithCovarianceStapmed`-message, which means the covariance matrix has to be included in the message.

The covariance matrix was calculated by driving the prototype manually and tune the gains; k_x, k_y and k_{yaw} until the results was satisfactory. The matrix is a 6×6-matrix since it is configured for translation in x, y, and z-direction, in addition to rotation in x, y, and z-direction. The covariance matrix for the odometry is displayed bellow:

$$\begin{bmatrix} k_x|\Delta X| & 0 & 0 & 0 & 0 & 0 \\ 0 & k_y|\Delta Y| & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & k_{yaw}|\Delta yaw| \end{bmatrix}$$

Where k_x, k_y and k_{yaw} are the covariance-gains. The prototype operates in 3DOF, thus translation in z-direction, roll and pitch is zeroed out.

3.9.2 ArUco Pose Covariance

In order to calculate the covariance matrix of the measured ArUco pose, an ArUco marker was fixed to a spot and the pose was measured. Since the marker is static, the ”noise” of the measurement is used to calculate the covariance matrix of the marker localization.

A script was created to track the ArUco marker and generate a graph, exporting the measured data in addition to calculating the covariance matrix. The relevant data is the position in x and y-direction as well as the angle of the marker. The script is listed in Appendix B.6.

The script samples the pose of the marker at 10 Hz for 60 seconds (600 samples). All the sampled data is stored in an array similar to the one displayed bellow:

$$A = \begin{bmatrix} x_1 & y_1 & \theta_1 \\ \vdots & \vdots & \vdots \\ x_{n-1} & y_{n-1} & \theta_{n-1} \\ x_n & y_n & \theta_n \end{bmatrix}$$

The covariance matrix is calculated by finding the deviation matrix using the formula in Equation (3.6) and then multiplying it by the transposed deviation matrix as shown in Equation (3.7):

$$a = A - \begin{bmatrix} 1 & 1 & \dots & 1 \\ \vdots & \vdots & \ddots & \vdots \\ 1 & 1 & \dots & 1 \end{bmatrix} \cdot A \cdot \frac{1}{n} \tag{3.6}$$

$$\text{Covariance Matrix} = a^T a \tag{3.7}$$

The covariance matrix is then implemented into the message sent to the `ekf_localization`-node.

3.9.3 Parameter Tuning

Several parameters in the EKF filter was tuned to improve the results. Initially, each of the inputs dimensions had to be set in the `config`-matrix, which is displayed below:

$$\texttt{config}\text{-matrix} = \begin{bmatrix} x & y & z \\ roll & pitch & yaw \\ v_x & v_y & v_z \\ v_{roll} & v_{pitch} & v_{yaw} \\ a_x & a_y & a_z \end{bmatrix}$$

The wheel odometry and ArUco pose will only provide measurements for x and y-position and *yaw* (z-rotation), thus these values was set to `true` and the rest was set to `false` in the `config`-matrix. For the IMU, the *yaw velocity* and x, y and z-acceleration was used in the `config`-matrix.

Each of the sensor messages has the option of being integrated differentially by setting the `_differential`-parameter to true or false. If the parameter is set to *true*, for measurement at time t, the previous measurement is subtracted at time t-1, and the resulting value is converted to the velocity [75]. If several measurements have an absolute pose information, these measurements may get out of sync and cause oscillations in the filter. Integrating differentially will avoid this scenario. Hence the IMU and camera pose was set to differential.

The _relative parameter was also tuned. If the parameter is set to true, the sensor data would be fused relative to its first measurement. This is useful in order to make the state estimation always start in (0, 0, 0). For this use-case, both the wheel odometry and IMU was set to true.

3.10 Navigation Stack

A ROS *navigation stack* was built to handle the navigation of the prototype. The pre-requisites were the prototypes transform configurations and sensor streams. Furthermore, the stack contains a global and local costmap. These features was implemented to the node commonly named `move_base`.

3.10.1 Transform Configuration

The `tf`-package in ROS was utilized to configure the transforms. First, the static transformations was configured. The main frame of a robot is commonly named "*base_link*", which refers to the URFD naming where a *link* is a rigid body. The "*base_link*" has its origin in the centre of rotation of the prototype. Further the other *links* which will transmit or receive data was defined relative to the *base_link* with a *(x, y, z, yaw, pitch, roll)*-transformation. The final inputs of the transform configuration is the frame parent and child ID. Below, the static transform from the *base_link* to the LiDAR-frame is displayed. The input values are given in meters and radians; hence the LiDAR is placed 0.35 m in x-direction and 0.1 m in z-direction from the centre of the base. It is also rotated upside down, thus the roll values was set to 3.14 rad = 180^o.

```
<node pkg="tf" type="static_transform_publisher" name="base_to_laser"
args="0.35 0 0.1 0 0 3.14 base_link laser"/>
```

The two final transforms used in the navigation stack was the dynamic transforms between the `/map`-frame and `/odom`-frame, and the transform between `/odom`-frame and `base_link`. These transforms are used by various nodes. The transforms between the `/map`-frame and `/odom`-frame is mainly used by the `amcl`-node described in Section 3.10.5. The transform between the `/odom`-frame and the `base_link` are mostly used by the `ackermann_odometry`-node explained in Section 3.8.6. The *launch*-file for the static transform configurations are listed in Appendix C.1.

3.10.2 Costmap

The costmap in ROS navigation is an occupancy grid type map. In ROS three `.yaml`-files was created with parameters for common, local and global costmap. The parameters are listed in Appendix D.2, D.3 & D.4.

For the common parameters, the `map_type` was defined as: *costmap*, the footprint of the prototype was defined by giving the coordinate to each of the prototypes four corners relative to the centre of rotation. The topic and frame of the LiDAR scan was the final common parameter to be configured.

The local and global-costmap individual parameters was set. For the local-costmap, the *global_frame* was set to the /odom-topic, as for the global-costmap, it was set to the /map-topic. The robot_base_frame was set to /base_link for both.

3.10.3 Time-Elastic-Band Local Planner

The planing and navigation of a car-like robot is not directly supported by the navigation stack, however teb_local_planer could be manipulated to support plans that are feasible for Ackermann steered vehicles [76]. The teb_local_planer could then be configured to support front wheeled steered vehicles, as the prototype, as well as rear wheel steered vehicles like the full-scale reach stacker.

Path planing supported by car-like robots was achieved by extending the nonholonomic constraint by a minimum bound on the turn radius response by satisfying $r_{min} < v/\omega$. The min_turning_radius parameter was set. The Twist-messages from the navigation stack was converted into messages containing the steering angle and the linear velocity, which is handled by the ackermann_odometry-node. Differential driven robots have recovery behaviour provided by the navigation stack, this allows them to rotate around its own axis. Car-like robots must move forward or backwards to steer, so the recovery behaviour has to be turned off or replaced. This is further described in Section 3.10.8

The steering angle problem was fixed by teb_local_planner which executes the steering angle calculations automatically by changing the parameter cmd_angle_instead_rotvel to *true* and by specifying the wheelbase wheelbase in meters [76]. The angular velocity was then substituted by the steering angle. If the vehicle is supposed to utilize rear wheeled steering, like for an instance a reach stacker, the cmd_angle_instead_rotvel parameter has to be negative.

The planner is configurable by changing the parameters of the configuration file listed in Appendix D.5, Table 3.2 show a list of the parameters changed and their function.

Table 3.2: teb_local_planner Parameters

Parameter	**Value**	**Description**
odom_topic	/odom	Define what topic is to be used for odometry
map_frame	/map	Define what topic is to be used for the map
max_vel_x	0.3	Maximum /cmd_vel in linear.x
min_turning_radius	0.63	The minimum turning radius for the planner
footprint_model	polygon	The shape the planer uses for the prototype
free_goal_vel	False	The prototype has to stop in goal, not coast
min_obstacle_dist	0.2	Minimum distance from obstacle
weight_kinematics_nh	1000	Weight for *nonholonomic* kinematics of robot
weight_kinematics_forward_drive	100	Favour forward driving, reducing reversing
weight_kinematics_turning_radius	100	Make path weight turning radius
max_number_classes	4	The amount of path generated/considered
enable_homotopy_class_planning	False	Disabling parallel planning

3.10.4 Global Planner

The global_planner-package [77] was used as the *global planner* in the navigation stack. This package adheres to the nav_core and move_base-package. No parameters was changed, thus the default settings was utilized.

3.10.5 Adaptive Monte Carlo Localization

Adaptive Monte Carlo Localization, or *AMCL* [78], was used to improve the localization of the prototype by combining the pose from the ArUco tracking-system with the laser scan-matching provided by the amcl-node. The parameters are based on a default template for a differential drive robot. Initially the *global costmap* and the provided map would not align when the planner was launched and would drift apart during movement. The drift problem was solved by setting the odom_model_type to diff-corrected instead of diff, in addition to adding the odom_alpha-values. The launch-file is attached in Appendix C.2.

3.10.6 Mapping

To achieve the best possible localization of the prototype, the map had to correspond well with the actual LiDAR reading. ROS has a pre-installed SLAM package, named gmapping [79], in addition there is another commonly used package named hector_slam [80].

Both packages have a similar setup. They require to know the static transforms between the physical components, in addition to the dynamic transform between the map and the odometry-frame. The launch-file used to run these nodes are listed in Appendix C.3 for gmapping and Appendix C.4 for hector_slam.

3.10.7 Move Base

The move_base-node is used to drive the prototype. The node handles the map and path planning. In the move_base launch-file the map_server-node and amcl-node is first launched. Then the parameters for the costmaps and teb_local_planner is loaded and teb_local_planner is started. The launch-file is attached in Appendix C.5.

3.10.8 Recovery Behaviour

When driving autonomously, the prototype might get stuck. A *recovery behaviour* was implemented into the navigation stack to unstuck the prototype.

ROS comes with several recovery behaviours implemented, rotate_recovery, clear_costmap_recovery and move_slow_and_clear [33]. These behaviours are pretty self-explanatory, where rotate_recovery make the robot rotate in place (works only on differential drive robots), clear_costmap_recovery clears the entire costmap and start gathering data for a new costmap. move_slow_and_clear forces the robot to move slowly and clear non-existing objects from the costmap.

rotate_recovery is the default behaviour in the navigation stack, however this was changed to clear_costmap_recovery, since the prototype is not a differential driven robot.

3.11 Autonomous Driving

The autonomous driving was executed by combining the localization and navigation stack. By launching the move_base-file in ROS, the prototype is able to navigate to a goal position. The goal position is determined by using *2D Nav Goal*-tool in RViz or by publishing a geometry_msg/PoseStamped-message to the /move_base_simple/goal-topic. Furthermore, the path can be configured to intersect waypoints by using the *Publish Point*-tool in RViz or by publishing a geometry_msg/PointStamped-message to the /clicked_point-topic. Lastly, to help with the localization of the prototype, an initial pose could be assigned in RViz using the *2D Pose Estimate*-tool or by publishing a geometry_msg/PoseWithCovarianceStamped-message to the /initialpose-topic.

3.12 Continuous Autonomous Driving Through Waypoints

A program making the prototype continuously drive through user defined waypoints was developed to showcase the prototype's autonomous capabilities. The program uses move_base with the teb_local_planner to navigate as before, however in the program the "goal" is set by using the follow_waypoints-node [81]. The node has a list of waypoints situated in a map. When the command is executed the prototype move to each waypoint in the order it was assigned. The user assigns goals by using the *2D Pose Estimation Tool* in RViz. When the waypoints are assigned, the prototype will navigate to each waypoint when the following command is executed in the terminal window:

```
$ rostopic pub /path_ready std_msgs/Empty -1
```

The follow_waypoints-node does not enable continuous driving alone, thus a program was developed to enable the robot to drive in a continuous loop.

The continuous loop ability was enabled by implementing a while-loop into the *path executions class* of the script. The while-loop run as long as a counter n is less or equal to the length of the waypoints-list. Secondly if the counter n exceeds the number of paths, it resets itself, thus forcing the while-loop to restart and the first waypoint will be set as the current goal.

The continuous-loop script is attached in Appendix B.8.

4. Results

This chapter present the result of the previously described method followed in this thesis. Please note that the source codes was made as modular as possible in order to have a set of packages which could be rearranged and utilized for future applications.

Due to the confidentiality of this thesis, the code is uploaded to a private GitHub repository which can only be accessed by receiving an invitation. To request an invitation to the repository, please send an email with the GitHub account to:

`magnus.tomren@gmail.com`

A video demonstrating the prototype was made. The video demonstrates: SLAM, point-to-point driving and continuous driving. The test in the video also showcases the prototypes ability to avoid dynamic obstacles being placed and removed. The video is uploaded to YouTube and is accessed through the following link:

`https://www.youtube.com/watch?v=TfkkGDr1rkw`

4.1 System Architecture

The system architecture of an autonomous vehicle is generally divided into three categories: *perception, planning* and *actuation* [82].

Perception uses internal and external sensors to understand the surroundings. The planning uses the output of the perception and usually some type of map to generate a plan for where the vehicle is heading.

A planner usually consists of a *global* and a *local* planner. The global planner keep track of where the vehicle is heading and the current position. A local planner is used to avoid dynamic obstacles and obstacles not present in the initial map. The local planner communicates directly to the *low-level control* which connects the planning and the acting of the vehicle. The actuation is the part of the system which control the physical vehicle based on the inputs provided by the previous systems.

Figure 4.1 displays the general system architecture of the prototype.

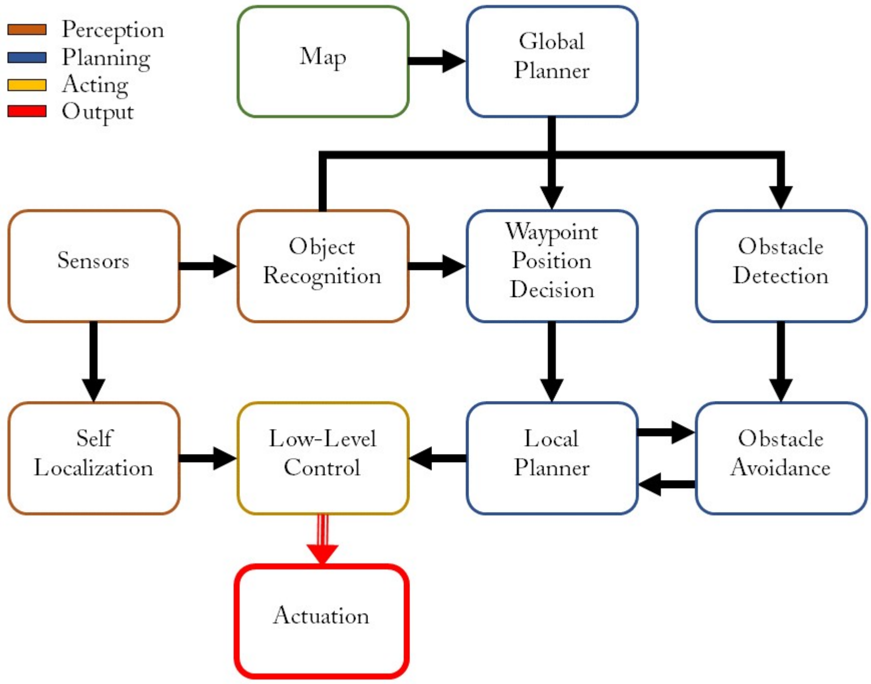


Figure 4.1: System Architecture for Autonomous Vehicle

Figure 4.2 displays a sketch of the indoor localization-system based on the ArUco tracker.

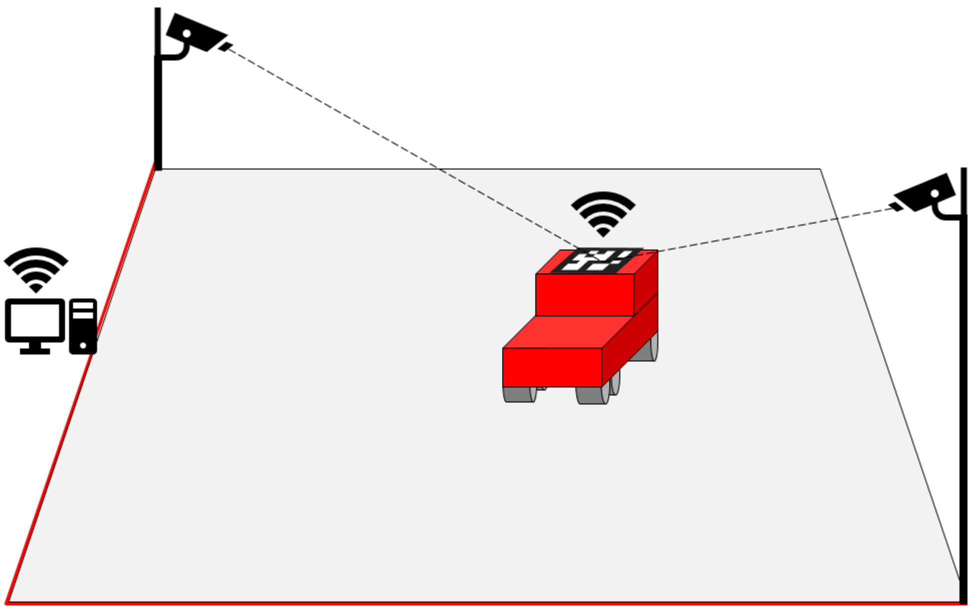


Figure 4.2: Sketch of System Overview

In the RQt-graph (attached to the very last page), the entire ROS communication is visualized.

4.2 Camera Calibration

Figure 4.3 displays the result of the camera calibration using the script described in Section 3.7.

By locating each corner in the grid, the script generates the calibrated camera matrix which was equal to:

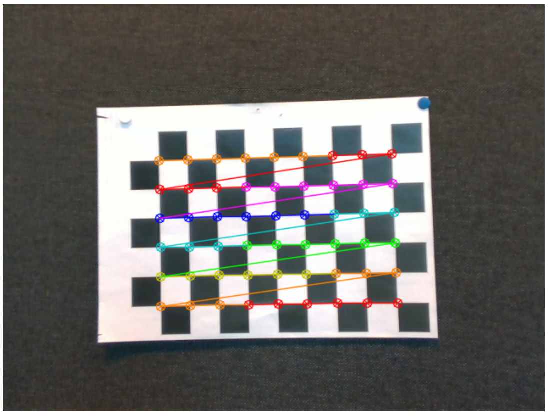


Figure 4.3: Camera Calibration Result

$$\text{Camera Matrix} = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 724.7 & 0.0 & 377.8 \\ 0.0 & 2777.9 & 320.1 \\ 0.0 & 0.0 & 1.0 \end{bmatrix}$$

4.3 ArUco Marker Detection

The ArUco tracker worked rather well and was able to detect a 20 × 20 mm marker from a distance of 15 meters. Figure 4.4 show a screen-shot of the viewfinder generated by the ArUco tracker code described in Section 3.8.3, detecting a marker.

When the angle of the marker was too shallow (Approximately 30^o), the algorithm had trouble detecting the marker. Various lighting-conditions also caused some issues to the tracker's accuracy. However, these problems where only present in extreme conditions not experienced in the controlled testing area.

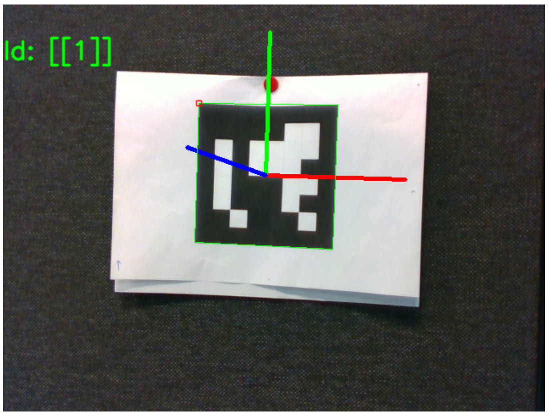


Figure 4.4: ArUco Marker ID 1 Detected

4.4 Depth Camera Measurement

When the ArUco tracker managed to detect a marker, the region of interest (ROI) in Figure 4.5 was generated. The image is binary, with the white area corresponding to a value received and black area corresponding to no data received.

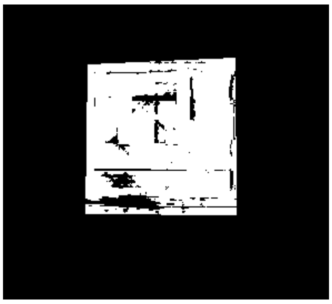


Figure 4.5: Depth Image ROI Generated by ArUco Marker

The RGB-camera was implemented as an output to show only the ROI. This made it simpler to locate the depth camera and understand where the measurement was taken. Figure 4.6 show the RGB ROI, where the blue square inside is the area is where the depth data is gathered.

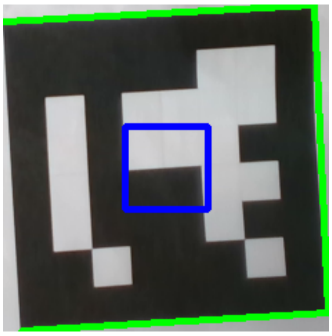


Figure 4.6: RGB ROI

The code utilized to gather ArUco data is described in Section 3.8.4.

4.5 Manual Driving & Deadband Compensation

The manual driving of the prototype was tested using the keyboard teleoperation described in Section 3.6, on a laptop connected to the prototype wirelessly. It was discovered that there was a significant deadband present when driving the prototype forward and in reveres. The deadband was identified by incrementing the motor duty cycle until the prototype started moving. The test resulted in a deadband of ± 0.05 in the duty cycle.

A set of `if`-statements was implemented in the low-level control to compensate for the deadband, in order to prevent the prototype from jerking at small duty cycles. The `if`-statements are listed below:

```
if duty_cycle > 0:
    duty_cycle = duty_cycle + 0.05

if duty_cycle < 0:
    duty_cycle = duty_cycle - 0.05

if duty_cycle == 0:
    duty_cycle = 0
```

4.6 Odometry

The accuracy of the odometry script was tested and the results was visualized in RViz to confirm that the generated turning radius corresponded to the actual values measured with the prototype. Table 4.1 shows the results from the simulated and measured maximum turning radius and Figure 4.7 show a plot from the simulated odometry in RViz.

Table 4.1: Ackermann Odometry Turning Radius

Simulated [m]	**Measured [m]**
0.628	0.63

The steering angle, α, was tuned until the simulated results was close to the measured results. The maximum steering angle of the prototype was eventually measured to $0.5\,\text{rad} = 29^o$.

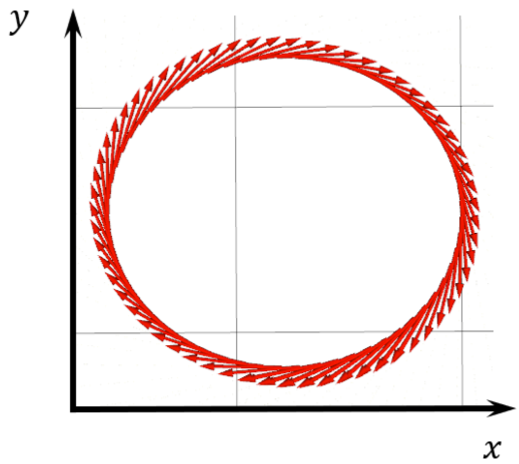


Figure 4.7: Odometry Mapping in RViz

The prototype was manually driven 3 meters in x-direction and 1 meter in y-direction. Table 4.2 show the simulated and measured results.

Table 4.2: Ackermann Odometry Turning Radius

Direction	**Simulated [m]**	**Measured [m]**
x	2.98	3.00
y	0.90	1.00

Figure 4.8 shows the simulated results in RViz.

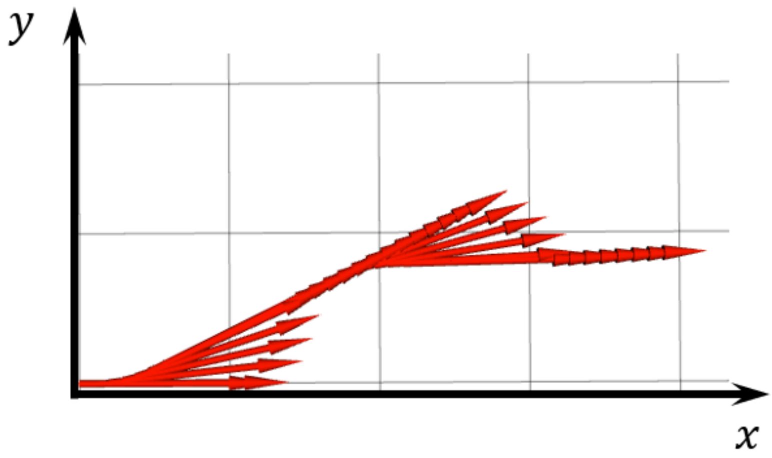


Figure 4.8: Odometry Test in RViz

The short-term accuracy of the wheel odometry was deemed sufficient.

4.7 Transform Configuration

The static transforms was configured as shown in Figure 4.9, where x-direction is red, y-direction is green and z-direction is blue. The odometry-frame is placed on the rear axle of the truck and the IMU and `base_link` is placed in the centre of rotation. The IMU is mounted upside-down to enable USB-port connection. By configuring the transform to the physical position and orientation, the output of the IMU orientation is correct relative to the rest of the truck. The same goes for the ArUco marker on top of the prototype, which is rotated 90 degrees relative to the local x-direction of the model. Finally, the LiDAR is mounted upside-down, by rotating the frame 180 degrees in the transform-node the laser scan is not inverted.

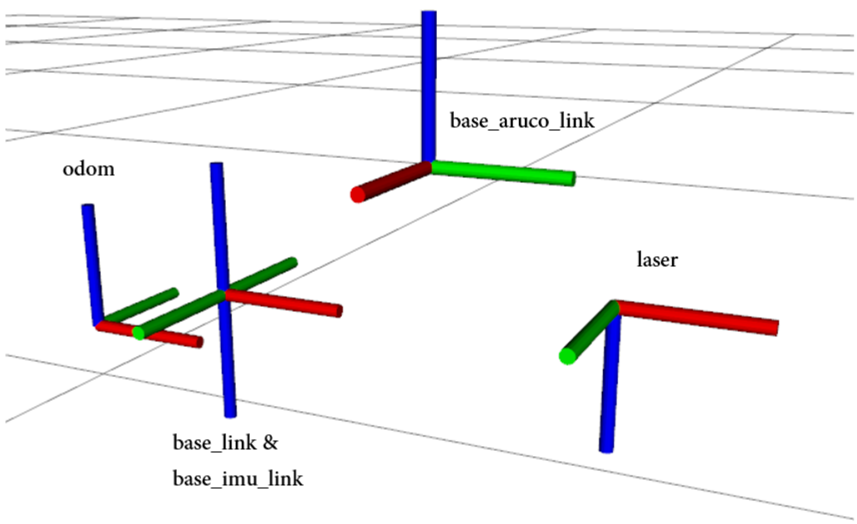


Figure 4.9: Static Transform Configurations Viewed in RViz

4.8 Localization

A test of the localization system was conducted. The test was setup up with the stationary cameras looking down on the test area. The prototype was manually driven around and the position was registered. The ArUco tracker data was combined with the wheel odometry in an extended Kalman filter-node to estimate an accurate pose of the prototype.

4.8.1 ArUco Tracking

The ArUco tracking uses the RGB camera and the ArUco tracking code described in Section 4.3. The depth camera measures the distance to the midpoint of the physical marker as described in Section 3.8.4. The depth camera was presumed to measure distance more accurately than the RGB camera. However, the depth camera experienced noise. Figure 4.10 show how much the dept camera measurement varies over time compared to the RGB ArUco tracker. The tracker provides one pose constantly, while the measurement from the depth camera jumps back and forth with ±0.25 m, which is not sufficient.

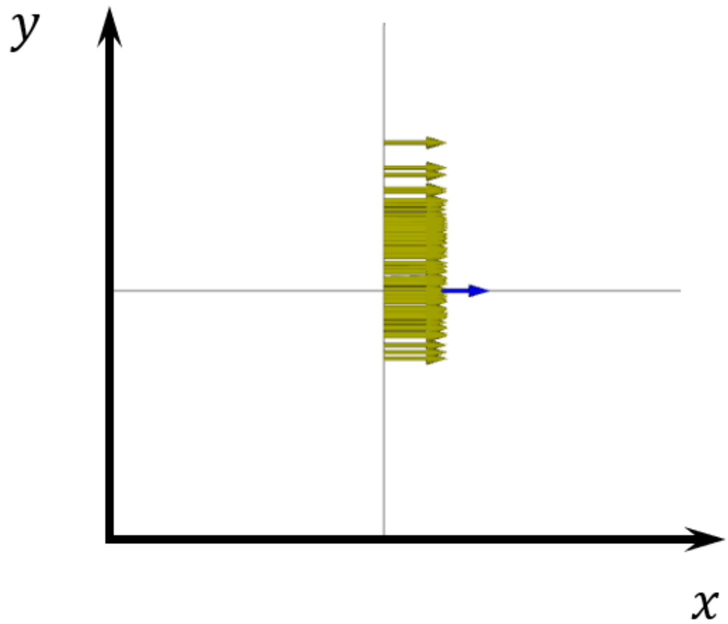


Figure 4.10: Pose by ArUco Tracker (Blue) & Depth Camera (Yellow)

The depth camera approach had to be abounded and the ArUco tracker uses only the RGB sensor for tracking.

A *Odometry*-message was generated using the odometry script listed in Appendix B.7. The odometry script utilizes the values from the ArUco-tracker script described in Section 4.3. The odometry script generates a relative pose of the marker, meaning that the position where the ArUco marker is located when the localization is initialized is set to the initial pose of the marker.

Figure 4.11 show the output of the Intel RealSense camera when localizing the prototype. The ArUco marker is recognized and the (x, y)-coordinate and pose is further used in the localization of the robot.

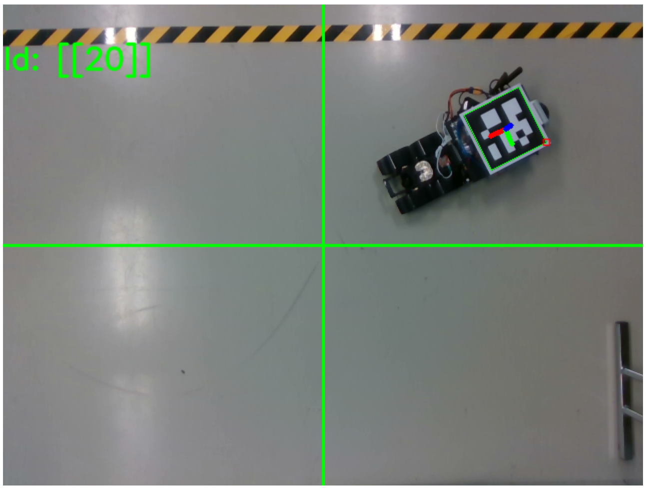


Figure 4.11: Localization Test Viewed from Camera

The optimal position to obtain the greatest field-of-view is calculated from the equations in Section 3.8.5. A camera mounted obliquely cover more area than a camera facing straight down. The test results of the oblique camera were not good due to shallow angle relative to the flat mounted ArUco markers. The camera had trouble detecting the marker and the varied lighting condition also affected the result. Figure 4.12 and 4.13 show two screenshots from the test where the ArUco marker is not detected and next image showing that the marker is detected. Mounting the camera higher would improve the result.

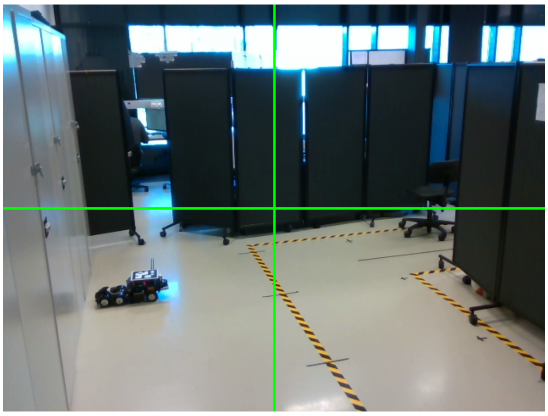


Figure 4.12: ArUco Not Detected

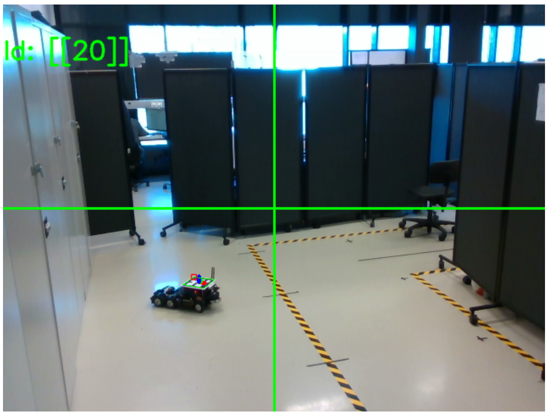


Figure 4.13: ArUco Detected

Facing the camera straight down obtained the best results. Both in tracking the ArUco and measuring its position within the camera frame, however the field of view is more limited than the obliquely mounted camera.

4.9 Extended Kalman Filter Configuration

This section presents the results of the extended Kalman filter configuration, including covariance gains for the wheel odometry, the generated covariance matrix for the ArUco tracker and tuning of the `ekf_localization`-node parameters.

4.9.1 Wheel Odometry

The prototype was driven around manually to gather odometry data in order to tune the covariance matrix described in Section 3.9.1. Due to the backlash in the steering and other sources of error the prototype drifts to either left or right, thus the k_y and k_{yaw} was adjusted until the results was satisfactory. Moreover, the distance in x-direction was pretty accurate which made the k_x-gain quite small. The final gains are listed in Table 4.3.

Table 4.3: Wheel Odometry Covariance Gains

Variable	Value
k_x	0.1
k_y	0.5
k_{yaw}	0.5

4.9.2 ArUco Pose

The ArUco pose measurement is not static. The pose measurement noise was outputted from the script described in Section 3.9.2 and plotted over a time span of 60 seconds. The pose measurement plot is shown in Figure 4.14. The ArUco marker was situated 2 meters away from an arbitrary place in the camera frame. The plot show that the measurement is not static and has some noise, however these peaks are small, with only a few millimeters of inaccuracy, which will not affect the overall performance of the tracking.

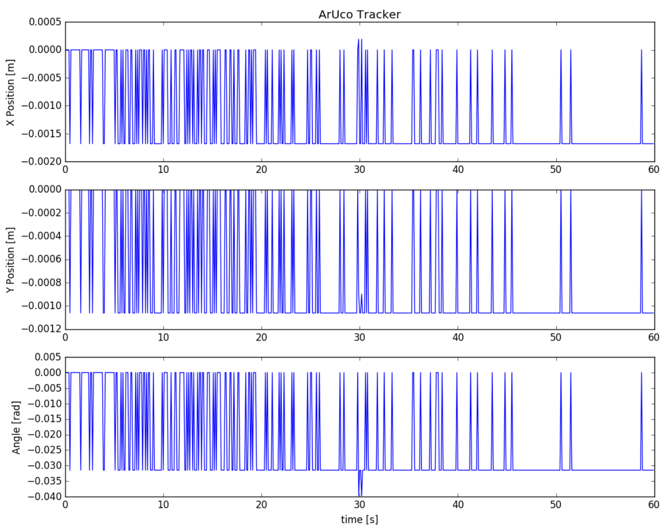


Figure 4.14: ArUco Pose Measurements

The covariance matrix of the ArUco pose measurement was calculated from the gathered data and is presented below:

$$\text{Covaraince Matrix} = \begin{bmatrix} 0.00126893 & 0.00080084 & 0.02373705 \\ 0.00080084 & 0.0005075 & 0.01506908 \\ 0.02373705 & 0.01506908 & 0.44778692 \end{bmatrix}$$

The variances are the diagonal values and the covariances values are the non-diagonals.

4.9.3 Parameter Tuning

To further improve the results of the `ekf_localization`-node, the ArUco tracker and odometry data was compared. Figure 4.15 show the ArUco tracker in red versus the wheel odometry in green. As seen, there is some error in the odometry relative to the tracker. To improve the odometry measurement, the maximum steering angle was decreased from 0.5 rad $= 29^o$ to 0.45 rad $= 26^o$. The result after tuning is shown in Figure 4.16.

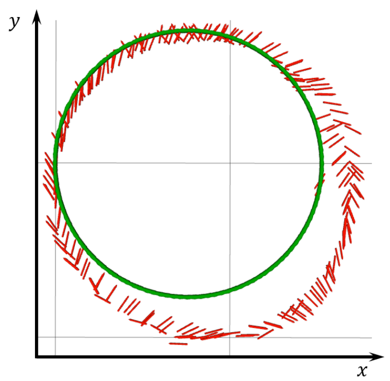


Figure 4.15: ArUco Tracking (Red) vs. Odometry Before Tuning (Green

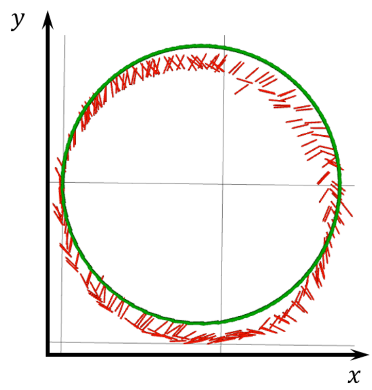


Figure 4.16: ArUco Tracking (Red) vs. Odometry After Tuning (Green)

Figure 4.17 shows a test where the prototype drives in a straight line. The steering inaccuracy causes the prototype to drift to the left. The green arrows are the wheel odometry going straight. The yellow arrows are the ArUco tracker which clearly show that the prototype is drifting. Lastly, the red arrows are the output from the `ekf_localization`-node which has a bias towards the more accurate ArUco tracker which is the desired result. From this test it is obvious that the ArUco tracker registers the drift and the extended Kalman filter helps to improve the position estimate.

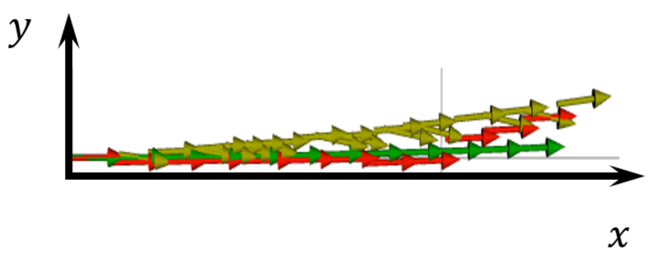


Figure 4.17: Localization Drift Test

4.10 Mapping

The mapping abilities was configured in several tests. The ROS integrated `gmapping`-package was the first to be implemented and tested on the prototype. The `gmapping` algorithm encountered several problems and results was not satisfactory. The problems where mainly due to the algorithm relaying too much on the inaccurate odometry data. This problem is further discussed in Section 5.5. The `gmapping` algorithm was abandoned and `hector_slam` was implemented as the SLAM-algorithm.

4.10.1 Short Distance Mapping

Initially a small closed-off area in the Machine Hall at UiA was mapped to ensure that `hector_slam` worked as expected. The *scan matching*-based algorithm worked immediately and obtained an accurate map of the surroundings. Figure 4.18, show the map generated with `hector_slam` in addition to the driven path (green line) originating from odometry data.

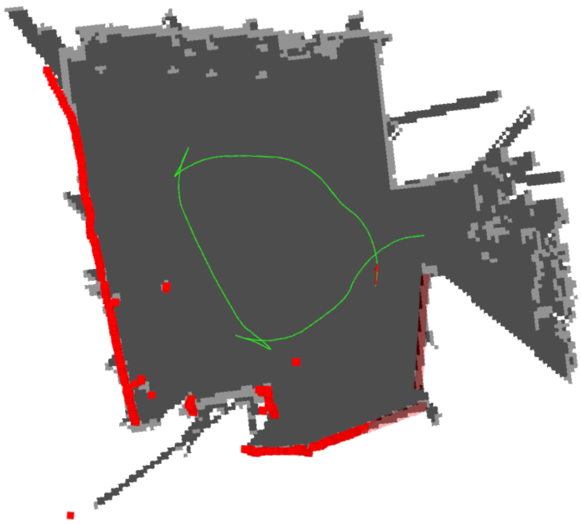


Figure 4.18: SLAM of Test Area in the Machine Hall

4.10.2 Loop Closing

An important aspect of a SLAM-algorithm is the *loop-closing* ability. As mentioned in the theory, hector SLAM does not provide an explicit loop closing ability, but manages to create a continuous loop in many robot applications. The loop closing was tested for `hector_slam` by driving the robot manually around a hallway shaped like a rectangle in the A3 building at UiA. Figure 4.19 shows the floor plan of the hallway. The orange area is the hallway driven by the prototype. The long stretch in the hallway is roughly 30 meters long.

Figure 4.19: Building Plan of Hallway A3, UiA [83]

Figure 4.20 show the generated map of the hallway. The resulting map was accurate and demonstrates the ability to construct a continuous map with `hector_slam`.

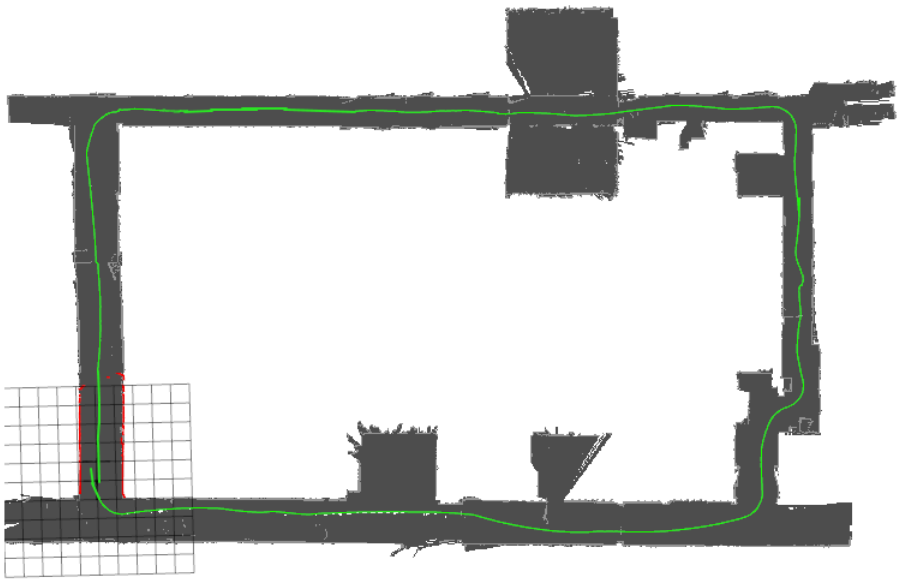


Figure 4.20: SLAM of Hallway A3, UiA

4.10.3 Long-Distance

Finally, a long-distance test was conducted. The `hector_slam` algorithm was tested in the longest continuous hallway at campus (main straight is approximately 50 m long). The robot was driven to the end of one of the side hallways, executed a K-turn and drove out the same hallway, before continuing through to the next hallway. The result is presented in Figure 4.21.

The map appears to have some angular offset in the third corner, which does not appear to be exactly 90 degrees. In the area marked with 1 it is possible to see the legs of several lockers placed against the wall. In Area 2 the lockers was situated directly on the floor. Area 3 there is a window at the end of the hallway, which the LiDAR see straight trough. Thus, the lines that appears to "grow" out of the map. Area 4 shows the K-turn carried out at the corner. In Area 5 the robot drove over 2 doorsteps, which lead to inaccuracies in the map. However, the result was quite good and the prototype is able to SLAM areas much bigger than the configured area at Red Rock, with high accuracy.

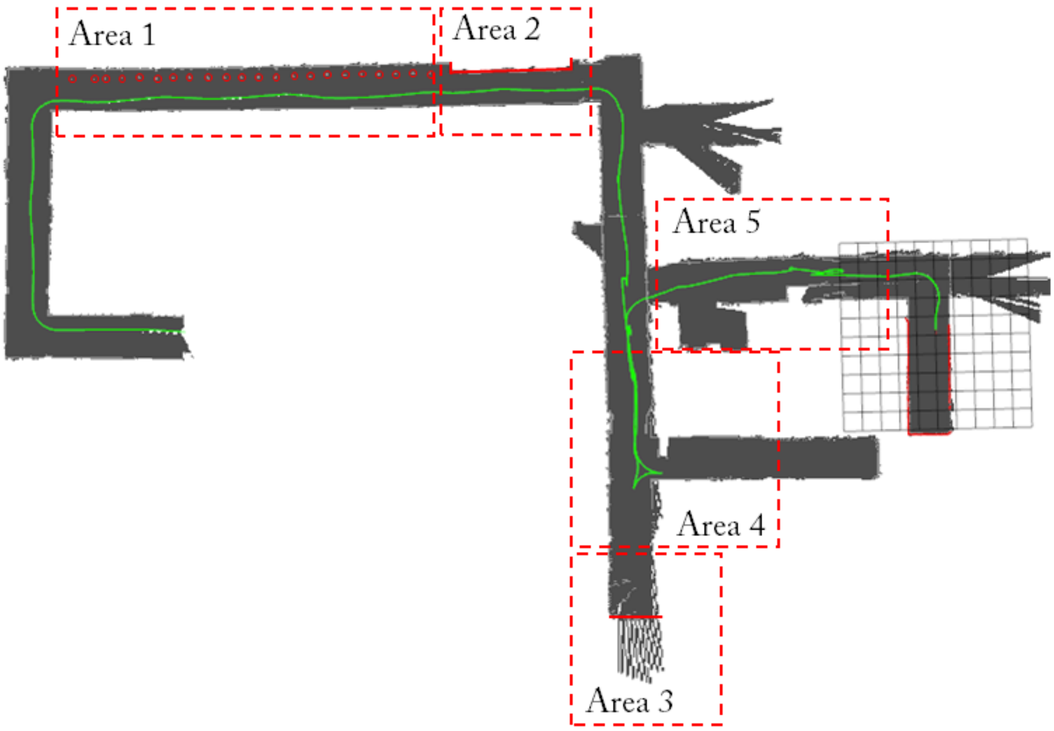


Figure 4.21: SLAM of Hallway A1, UiA

4.11 Point-to-Point Driving

The point-to-point driving test was carried out in an open environment without obstacles to make it as simple as possible. The goal of this test was to confirm that the communication with RViz and the low-level control through the `move_base`-node worked as expected. The prototype was given an initial pose by using *2D Pose Estimation*. The *global frame* was set to `/map` in a known map and the robot was supposed to navigate to a goal pose by using *2D Nav Goal* with the global frame set to `/odom` in RViz.

The first test was carried out using the default local planner in ROS, the `dwa_local_planner`. Initially it worked when the prototype was set to drive in a straight line. However, when the prototype was set to turn, it did not take the Ackermann kinematics into consideration. This resulted in the prototype turning the wheels to the maximum angle while not moving forward.

A Twist-message, giving a value to the angular.z-parameter, will make a differential drive robot rotate in place, this is not the case for Ackermann steered robots. After discovering the kinematic problem, the dwa_local_planner was abounded.

The teb_local_planner, was tested. This planner can be manipulated to function with Ackermann steered robots. The planner managed to implement the steering radius of the prototype and added K-turns to the planned path, thus completing the straight-line test and making the prototype able to turn around.

4.12 Path Planning

After testing the planner's ability to drive from point-to-point without any obstacles, a test for verifying how the planner handles known static obstacles and unknown dynamic obstacles was executed.

4.12.1 Simulation Test of TEB

A simulation test of the teb_local_planner algorithm was conducted before implementing it in the navigation stack. The planner was tested in a simulated environment containing obstacles in RViz. The simulated path goes from one point to another in a horizontal line. Three tests were conducted: The first test contained one obstacle, the second test contained three obstacles. The third test placed the obstacles in a manner that required a K-turn. The algorithm plots every possible trajectory and chooses the trajectory with shortest execution time with a certain distance to the obstacles in order to avoid a collision.

The result from the test with one obstacle is displayed in Figure 4.22 and 4.23. There are two paths available, one of the paths is longer than the other. The path with red arrows is the chosen path and it is obvious that it is the shortest path, with shortest execution time. The two plots in Figure 4.23 contains the translational velocity in m/s and rotational velocity in rad/s from the chosen path.

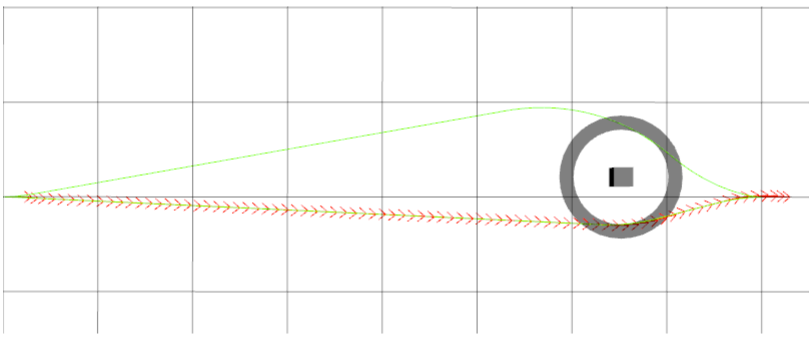


Figure 4.22: Simulated teb_local_planner with One Obstacle

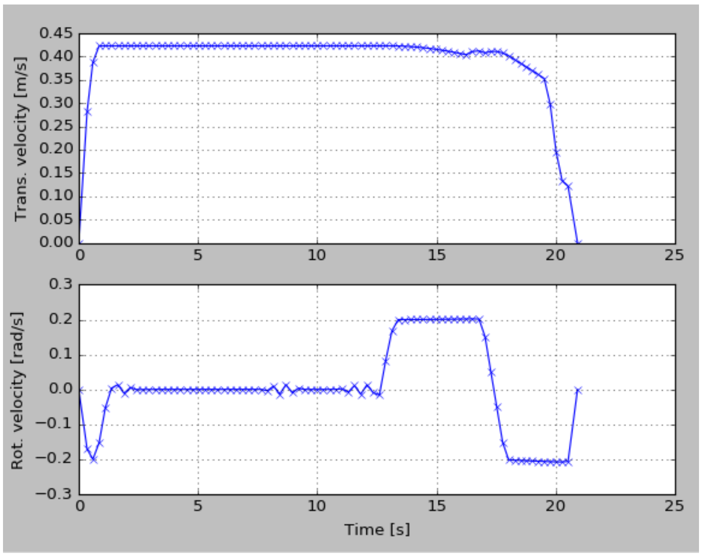


Figure 4.23: Velocity Graph from Simulated `teb_local_planner` with One Obstacle

The result from the test with three obstacles is displayed in Figure 4.24 and 4.25. The algorithm finds five possible paths and chooses the path with red arrows. The two plots in Figure 4.25 contains the translational velocity in m/s and rotational velocity in rad/s from the chosen path.

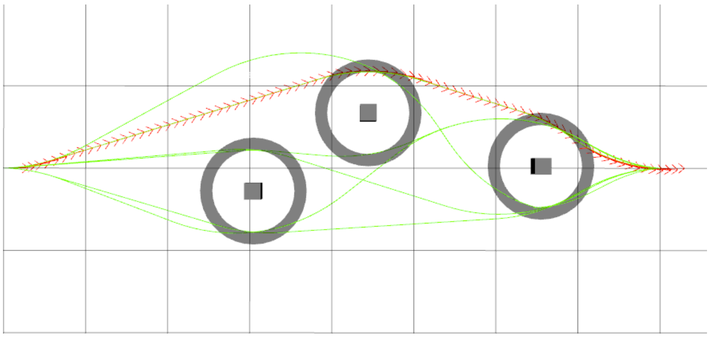


Figure 4.24: Simulated `teb_local_planner` with Several Obstacles

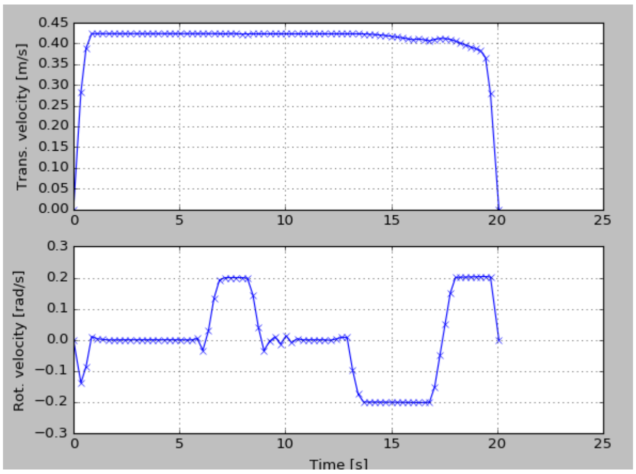


Figure 4.25: Velocity Graph from Simulated `teb_local_planner` with Several Obstacles

The result from the K-turn test is displayed in Figure 4.26 and 4.27. The planner makes two paths with a K-turn. One of the paths is slightly longer than the other. The path with red arrows is the chosen path and it is the shortest path, with shortest execution time. The K-turn is visible in Figure 4.26.

It is observed in the translational velocity plot in Figure 4.27 that the velocity is negative. The simulated vehicle had to reverse and then steer around the obstacle to reach the goal position.

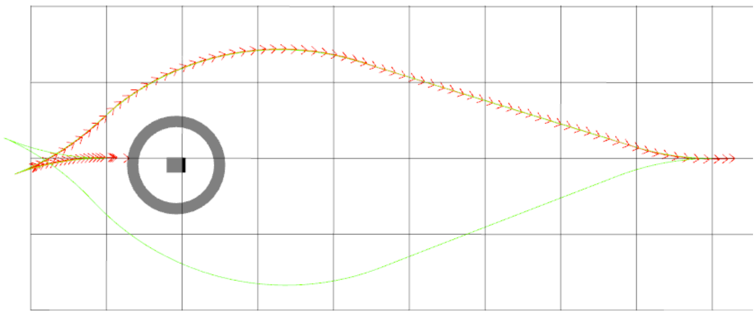


Figure 4.26: Simulated Path in `teb_local_planner` with K-turn

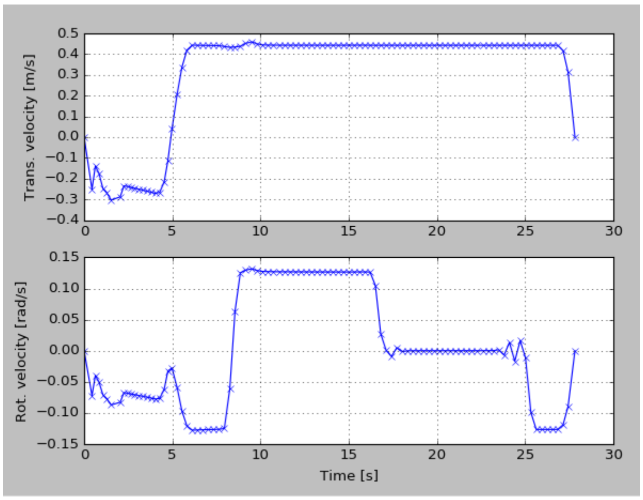


Figure 4.27: Velocity Graph from Simulated Path in `teb_local_planner` with K-turn

4.12.2 Real World Test of TEB

In this test, a known map with obstacles was provided, in addition to one unknown "dynamic" obstacle represented by a cardboard box. The prototype was localized in the map and given a goal destination behind the obstacle. The global path was generated and `teb_local_planner` provided the local path in real-time as the truck started to drive. The obstacle was detected and added to the local costmap when the robot was within a range of 2 m. The local planner calculated a new path around the obstacle and preceded to drive around it.

Figure 4.28 show the initial pose of the prototype with the local costmap (blue and purple pixels) is detecting the walls surrounding the prototype's footprint model (dark green rectangle) using the LiDAR scan (red particles) while the prototype starts to move towards the goal position (green arrow) set to the left in the map. In Figure 4.29 the prototype detect the dynamic obstacle and re-plan a path around it (dark blue line). The red arrows are the discrete pose of the prototype.

Figure 4.28: Goal Received

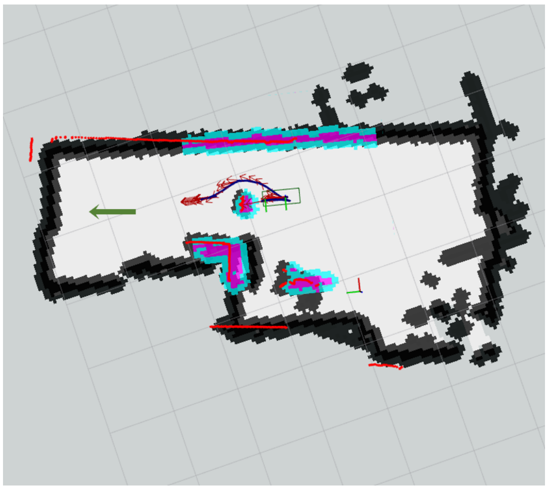


Figure 4.29: Re-Planning Around Obstacle

In Figure 4.30 it could be observed that the prototype avoided the obstacle and proceeded to the first goal. When the prototype reached the first goal pose, a second goal pose was given. Figure 4.31 show the prototype reversing into a K-turn to turn around and proceed to the second goal pose.

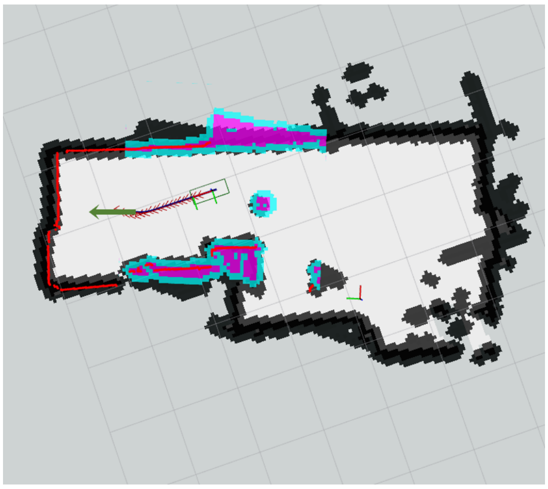


Figure 4.30: Avoided Obstacle

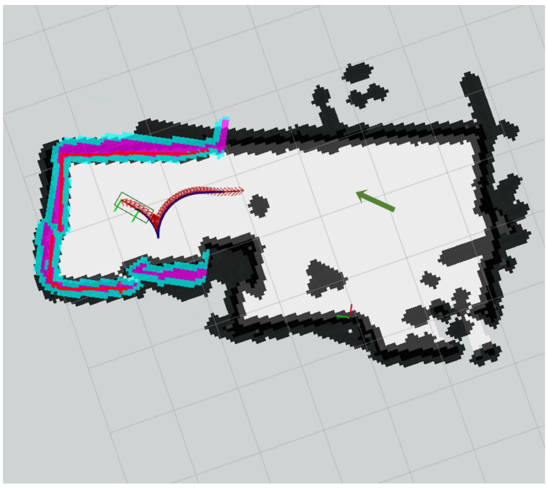


Figure 4.31: K-turn to Reach New Goal

Figure 4.32 show the path planner creating two K-turns for the prototype to turn 180 degrees and Figure 4.33 show the prototype just before the final goal was reached.

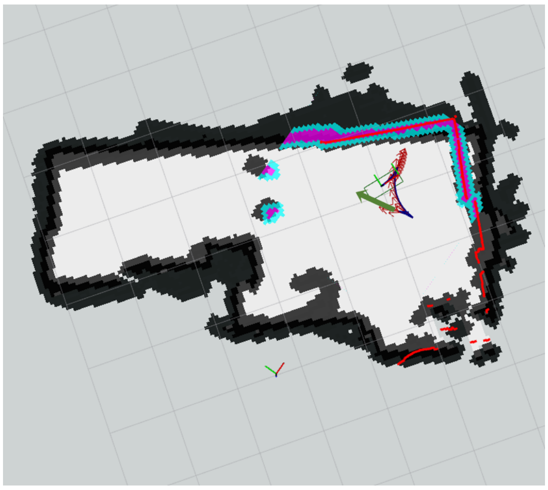


Figure 4.32: Turning Around

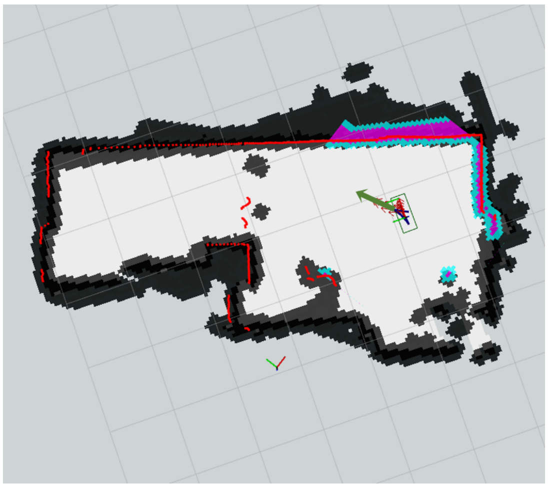


Figure 4.33: Reached Final Goal

Please note that all the screen captures from RViz was taken in *real-time*, thus they do not always show the optimal illustration of the scenario. The local planner iterates the path at 5Hz and is therefore continuously making small adjustments to the path.

4.13 Continuous Autonomous Driving Through Waypoints

The `follow_waypoints`-node was tested. The test was carried out in the small testing area in the Machine Hall at UiA. The waypoints were assigned in clockwise direction, thus making the prototype drive in a loop. Figure 4.34 show the setup of the waypoints in the map of the testing area.

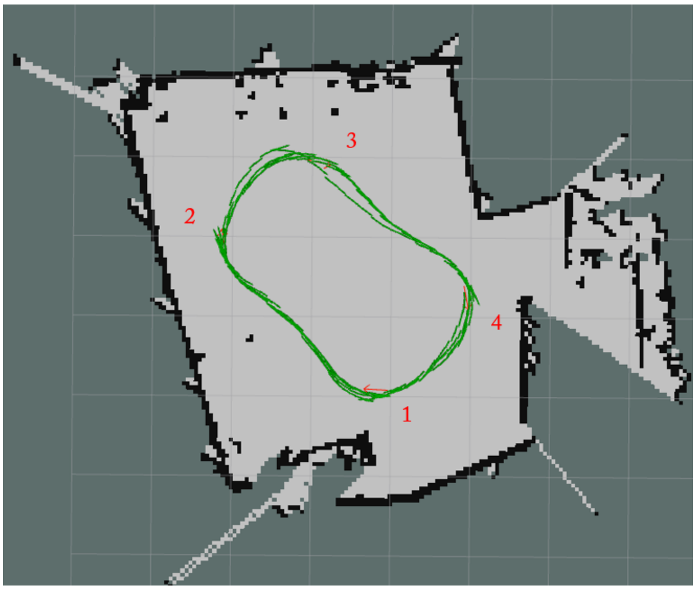


Figure 4.34: Path Driven Through Waypoints

A test of the continuous driving was conducted. The `while`-loop approached worked as expected and the prototype drove until it was stopped manually. It is also seen from Figure 4.34 that the prototype drives different paths for each loop. This is due to the algorithm always trying to drive the path that takes the shortest time and slight difference in position at each lap.

To make the prototype drive smoother and not spend time on getting the orientation of the waypoint vector right, the goal tolerance was increases. The prototype is configured to drive within a radius of 20 cm of the waypoint and not to mimic the direction of the waypoint vector. The tolerance is the reason for why the prototype did now drive completely through the third waypoint. This could be tuned further to reach a more specific result.

A second test with a dynamic obstacle was also conducted. The prototype drove three times through the waypoints. First without obstacle, secondly with an obstacle placed in *OB1* and lastly the obstacle was moved to *OB2*. As seen from the result in Figure 4.35, the prototype manage to re-plan the path around the dynamic obstacle and continued driving to the next waypoint.

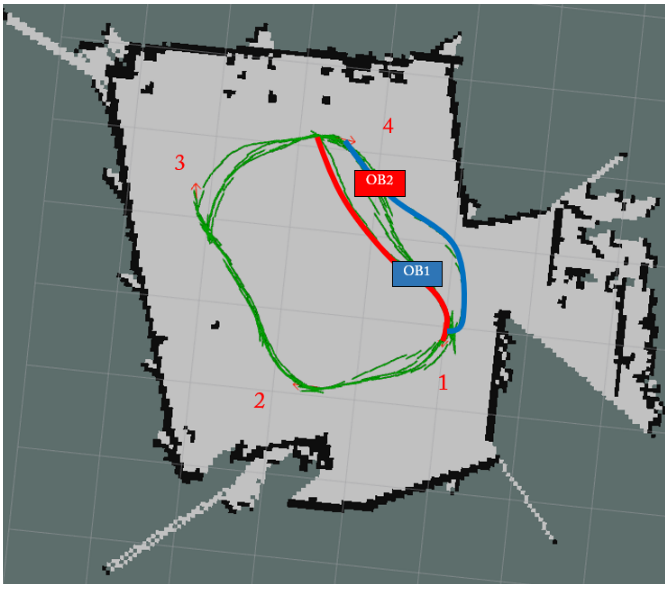


Figure 4.35: Obstacle Avoidance Through Waypoints

The path highlighted in blue is the path the prototype drove to avoid the first obstacle, *OB1*. The obstacle was then moved to *OB2* and the prototype drove the path highlighted in red.

4.14 User Manual

To simplify the somewhat intricate start-up procedure of the prototype, a user manual with a step-by-step explanation on how to use the prototype and start the different modes, in addition to troubleshooting and known issues was created. The user manual is attached in Appendix A.

5. Discussion

This chapter discuss the theory, methodology and results obtained in the master thesis, in addition to elaborating improvements and further work.

5.1 V-Model

It was decided to use the V-Model approach as a software development guideline in the beginning of the project. The methodology was systematic and worked relatively well. The V-Model is highly disciplined and not prone to changes. Some requirements changed during the scope of the project, like changes in the tracking system and the use of a truck rather than a reach stacker as a base for the prototype.

The V-Model approach was not followed explicitly in the module design and software design phase. The V-model states that every module should be finished when the software is stitched together, in the software design phase. This was not the case in this project, mainly due to the use of ROS. ROS enables modules to be tested together at a low level. The module design and software design phase were therefore more of a joint effort, where modules was tested together in a subsystem. The entire software was revised and tested again. Functionalities where added along the scope of the project until the entire software functioned in the desired manner.

It could be argued that the V-Model was not the right methodology for this specific project. The V-Model is not prone to frequent changes and the fact that ROS allows low-level software testing and implementation. The Incremental Model [84] would probably function better, due to the model combining the elements of the waterfall model with the iterative philosophy of prototyping.

5.2 Software & Hardware Issues

The fact that choosing the newest software and hardware is not always a best option was learnt the hard way in this project. A lot of time was spent making the ROS operating system work and make components work together.

5.2.1 Ubuntu & ROS Issues

A virtual machine running Ubuntu was used in the beginning of the project. The virtual machine was not suited for the project due to limited processor usage and difficulties to connect USB components. The hard drive was partitioned, and the newest software Ubuntu 18.04 was installed. However, the newest software had limited support within the ROS community, so the software had to be downgraded to Ubuntu 16.04. The hard drive also encountered some problems with the partition, leading to problems in the Windows partition of the hard drive.

The ROS distribution Melodic Morenia was first used in the project. The version was launched on May 23rd, 2018. The software experienced a lot of compatibility problems and was downgraded to ROS Kinetic Kame from May 2016 which worked well.

5.2.2 Hardware Issues

In future work it would be desirable to use an external SSD-drive to run Ubuntu. A hard drive partition could possible lead to problems with existing software. The software on the external hard drive could just be booted from any computer. Unless a project serves the purpose of evolving the newest software and has to use cutting edge technology, it is often more convenient to use an older software edition with more documentation and help available.

Some of the components used in this project was relatively new, which again lead to compatibility problems. There is also a low amount of available help and information related to new hardware. The components that experienced problems was the IMU, Intel RealSense camera and the VESC. The IMU has USB connection and an integrated Arduino board and the ROS compatible firmware had to be flashed to the IMU kernel. This proved to be a time-consuming problem, the software had to be altered and debugged to obtain compatibility. The VESC controls the brush less motor. The VESC has a servo out cable that was intended to be used to control the servo motor used in the steering. However, the firmware was the latest release and not ROS compatible, a lot of effort was made to flash the kernel with an earlier firmware, without results. The solution was then to drive the servo using an additional Arduino board and a servo driver board from Adafruit.

The newest Intel RealSense camera was implemented, and several issues became apparent. The camera did not work together with ROS on one of the computers. There was also some firmware problems and a variety of smaller compatibility issues.

The prototype was initially supposed to be tracked by retro reflective spheres and it was therefor decided to use an IR camera. It became apparent that the RealSense camera only emits a point cloud which does not enable sphere tracking. It was therefor decided to track ArUco markers with the RGB sensor. A better suited camera should replace the RealSense.

5.3 Prototype

The main difference between the optimal full-scale system and the down scaled prototype is the kinematics. The turning radius is far greater on the prototype truck than on a reach stacker, in addition the reach stacker's steering wheels are situated on the rear axle. This is not a major difference from the prototype truck alone. It simply means that the kinematics is inverse. The prototype could have been driven backwards, but this configuration would not allow the prototype truck to pull a trailer.

5.3.1 Steering

The steering on the prototype appears to be one of the main sources of positioning error. The steering appears to have some backlash and is wobbly. This became apparent when driving the prototype in a straight line forward, the prototype would start to deviate from the coarse. The steering made the prototype deviate 6 cm to the left when travelling a distance of one meter. The steering link was adjusted; however, the prototype still experienced a drift of 1 cm in y-direction, for every meter driven in x-direction.

The drift result in an error in the measured odometry. However, the Kalman filter and `amcl`-node compensates for it. Steering components machined in aluminium is a possible solution to the drift problem.

5.3.2 Ackermann Steering vs. Differential Drive

For prototyping purpose and proof-of-concept, a differential driven base would have been easier and faster to build and program. The Ackermann steering was used due to the fact that the full-scale reach stacker rely on the same principle. Differential driven robots are easy to control and program. Differential driven robots are more common in the ROS community and a lot more information is available compared to car like robots. Ackermann drive was challenging to configure but gave valuable insight and information that could help further development of the Red Rock autonomous reach stacker.

5.4 Localization

The ArUco tracker script had the world origin in the centre of the frame, which resulted in the ArUco odometry being negative relative to the wheel odometry. This made localization difficult and caused problems.

The Intel RealSense was not suited for the project and should be replaced with a high accuracy RGB camera meant for computer vision purposes. The RealSense was originally bought to replace the Kinect in the IR tracking. However, the RealSense did not work in the tracking of retro reflective markers due to it relying on structural light rather than complete illumination. It was therefor decided to track the ArUco marker instead.

The RGB camera track the ArUco marker and the depth camera calculate the distance to the centre of the marker. The RGB camera tracked the marker well and locked on to the marker immediately, however the depth camera on the Intel RealSense had highly oscillating behaviour and the ArUco tracker functioned best by only relying on the RGB camera on the RealSense. The camera system was considered to be excessive due to the prototype navigating with high accuracy using LiDAR, odometry and the `amcl`-node. In addition, the camera system has to be mounted and configured for each area the prototype is used. This would make it cumbersome to bring the robot to exhibitions or other places for showcasing the product.

The ArUco tracker works and was meant to simulate a GPS signal. With another camera system and further work it would function with high accuracy. The full-scale reach stacker relies on GPS and the prototype code has been configured for future implementation of a UTM-GPS signal.

5.5 Mapping

The ability to perform mapping with simultaneous localization and mapping (SLAM) is an important feature enabling fast implementation of the robot in a new environment. A map could be constructed by hand from blueprints, however there are often problem with the approach. Many buildings do not comply with the blueprints generated by the architects. Most buildings also have furniture and machines present, which alter the robot's perception of the environment. Mapping is truly one of the core elements of a completely autonomous robot.

The particle filter-based SLAM utilized by the `gmapping`-package did not function properly on the prototype. The algorithm managed to provide SLAM data, however the resulting map was just a cluster and did not make any sense. The main reason for the bad result is the Ackermann steering. The prototype steering is quite wobbly and has some backlash. The odometry from the kinematic equations are also not accurate due to the accuracy of the steering mechanism.

The ArUco tracking system could possibly have provided accurate odometry data to the `gmapping` SLAM algorithm. However, the camera system would then have to be installed and configured for the respective area. The camera system would also have a limited range, of about 10 meters. This would pose an issue when obtaining SLAM in larger areas.

The `hector_slam` worked well and provided god maps of the environment. A higher updating frequency significantly increased the accuracy of the map.

5.6 Local planner

Two different local planners was tested in this thesis: `dwa_planner` and `teb_local_planner`. In this section the performance of the planners is discussed and evaluated.

5.6.1 Dynamic Window Approach Local Planner

The `dwa_local_planer` does not take into account non-holonomic kinematics of Ackermann steered vehicles. This resulted in the prototype simply turning its wheels with out any forward motion when it was commanded to make a turn. DWA was therefore replaced by TEB.

5.6.2 Time-Elastic-Band Local Planner

The map updating frequency was configured to 0.2 seconds. During testing of the TEB planer there was a constant warning message stating that the map update took 2.7 seconds, making the prototype drive blind for 2.7 seconds. The delay caused the prototype to collide with obstacles and caused problems with following the path.

The problem was initially resolved by changing the local costmap's cell size resolution from 0.05 m to 0.2 m, in addition to changing the local cost map size from 6×6 m to 4×4 m. The changes decreased the computation time and the desired update frequency was achieved. Figure 5.1 show the initial local costmap resolution and Figure 5.2 show the downgraded resolution used for testing. The low resolution caused obstacles to appear lager, resulting in the prototype not being able to navigate through seemingly open areas.

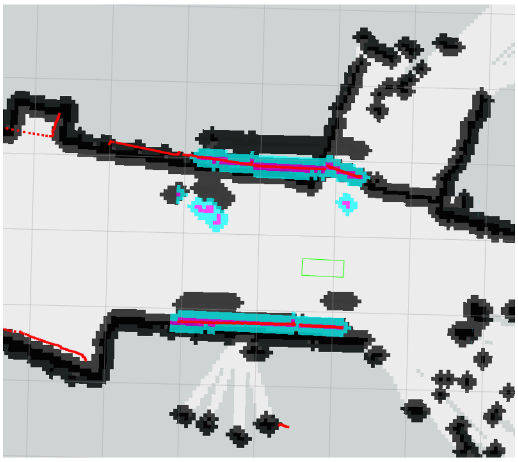


Figure 5.1: High Resolution Local Costmap

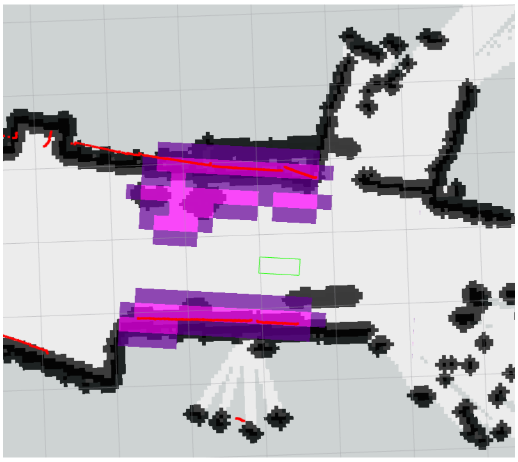


Figure 5.2: Low Resolution Local Costmap

Moreover, `teb_local_planner` does not have a linear behaviour due to its parallel planning in distinctive topologies (homotopy class). Since multiple trajectories are optimized at once, the process requires allot of CPU resources, thus the time complexity increases. By disabling the parallel planning, the performance increases significantly. The parallel planning is disabled by changing `enable_homotopy_class_planning` from `True` to `False`. Disabling the parallel planning reduced the loop time from 2.7 seconds down to 0.2 seconds. The prototype was then able to function with the initial resolution without warnings.

5.7 Prototype Getting Stuck

During testing the prototype sometimes ”got stuck” for no obvious reason. From the local costmap there was usually space for it to continue on the path or just simply reverse to where it came from. However the prototype went into recovery behaviour, which did not always solve the problem (this issue is addressed in Section 5.8.2). A solution to the problem was to manual drive the prototype in a short time period, forcing it to reverse and then letting it continue on its path.

5.8 Improvements & Further Work

This section presents possible improvements and further work.

5.8.1 Path Planning

A local path planning algorithm specially designed for the kinematics of a full size reach stacker would probably be an advantage. The TEB planer could be configured for car like robots with rear wheel steering. The planner could also be based on other algorithms for instance the dynamic window approach to name one.

In further work different global planners could be experimented with. In this thesis the default ROS `global_planner` worked well, and was therefor used. However both the `sbpl_lattice_planner` [85] and `carrot_planner` [86] are widely used and could be considered in further work.

5.8.2 Recovery Behaviour

The default recovery behaviour of the navigation stack, is rotating behaviour. The feature only works with differential driven robots. The clear costmap behaviour is a good solution, however it is not ideal. One recovery behaviour which could be developed would be to make the prototype back-track by reversing the path it just had driven, in addition to clearing the costmap for obstacle which is not present anymore. A second solution could be to turn around by using several small K-turns. This behaviour mimics the already exciting rotating behaviour and aims to solve the problem in a similar fashion.

5.8.3 Reversing Sensors

The ultrasound sensor should be implemented in further work. The sensor would function as additional redundancy. The prototype keeps track of the surrounding obstacles with the LiDAR scan, however there could be situations where dynamic obstacles suddenly appears behind the prototype. An ultrasound sensor would also be useful in situations where the prototype loses track of its position. This could solve some of the issues with the prototype getting stuck and be useful in the recovery behaviour. Most ideally the prototype could have featured a 360 degree field of view LiDAR, however due to the RC truck design it was not possible to achieve.

5.8.4 Multithreading & GPU Acceleration

The ROS package implemented on the prototype does not support multithreading or GPU acceleration. The path planning algorithm is using most of the computing power and is running on one core of the processor. The full potential of the NVIDIA Jetson is therefore not utilized. In further work the workload should be moved to the GPU.

The main advantage of using the GPU is that the local costmap could be increased in size, without suffering from reduced loop time. By increasing the size of the local costmap the prototype would be able to plan further ahead, thus improving the overall performance of the autonomous driving.

5.8.5 Container Detection

Although the sensor suit for the container spreader was defined in this thesis, a prototype was not developed. However, it should be possible to detect a container using edge detection in an image from the camera mounted on the container spread.

Differentiate between the three ISO sizes of containers (10 feet, 20 feet and 40 feet), is fairly simple by creating a rectangular bounding box around the detected container and then calculating the ratio between the length and width of the rectangle. The containers have the same width, the ratio of a 10 feet container is close to 1:1, thus a 20 feet container have a ratio of 2:1 and a 40 feet container have a ratio of 4:1.

5.8.6 Large Scale Prototype

The next prototyping step should be to; design a container spreader, test the prototype outside with a GPS sensor and increase the scale of the prototype. If possible, the prototype should be as close to a reach stacker as possible to obtain more accurate results. With the increased size and amount of sensors, the computing power should also be increased, as it was seen to cause issues in the small scale testing.

Outdoor Localization

The `geonav_transform`-package [87] in ROS could be used to implement a GPS sensor in the navigation stack. The package transforms 2D geographic coordinates to local x and y-coordinates. The GPS data is converted to the UTM coordinate system. The `geonav_transform`-node then publishes a transform from `/utm` to `/map`.

The package utilizes the `geonav_transform`-method to convert from geographic to local coordinates. The `geonav_transform_node` receives an odometry message from the GPS sensor containing sensor frame orientation and velocity. The message is transformed to a new odometry message containing information in the UTM frame and odometry frame. The `tf`-library is used to broadcast two transforms: `/utm` → `/odom` and `/odom` → `/base_link`.

The information from the `geonav_transform`-package is intended to be utilised parallel to IMU's and then fused together in the `ekf_localization`-node. The `ekf_localization`-node is used to estimate the pose of a robot with measurements originating from different sources through an extended Kalman filter.

The process is listed below [74]:

- Convert GPS data to UTM coordinates.
- Use UTM coordinate, EKF output and IMU data to generate a static transform T from UTM grid to robot world frame.
- Transform all future GPS data using T.
- Feed output back into the extended Kalman filter.

6. Conclusion

This thesis covers the process of constructing and testing a functional autonomous ground vehicle prototype for moving containers. An indoor localization system was successfully designed to precisely estimate the pose of the prototype and enabling autonomous navigation in a map generated using simultaneous localization and mapping (SLAM). The prototype is able to sense dynamic obstacles and avoid them.

The reach stacker is the optimal container handling solution for small container ports due to the versatile design. The vehicle is agile and has the ability to perform all the tasks required in container handling operations. The container spreader is flexible which makes the positioning of the base less important.

After several design revisions, the indoor localization system was developed and tested. The odometry was successfully combined with the ArUco tracker's pose estimation and the IMU measurement in an extended Kalman filter. The resulting odometry signal was used in combination with scan matching in an Adaptive Monte Carlo Localization-algorithm, to provide the final pose estimation for the prototype. The system manages to adjust for the drift in wheel odometry created by mechanical backlash in the steering.

The prototype is able to provide real time SLAM. The `gmapping` did not function as intended due to inaccurate odometry data causing problems when generating a map and estimating pose. Moreover, the `hector_slam`-algorithm worked well and provided a high accuracy map, due to it relying on the LiDAR measurement and scan-matching.

The `teb_local_planer` have a good performance keeping track of the path and avoiding obstacles. The `teb_local_planner` obtained the best results with parallel planning disabled. Continuous driving was implemented to enable the prototype to drive in a loop through user-defined waypoints, whilst avoiding dynamic obstacles. The prototype would have obtained better path planing performance with multithreading and GPU acceleration, the CPU has limited computing resources, thus restraining the size of the local costmap.

Lastly it could be concluded that a fully autonomous prototype has been successfully developed, with a scalable and modular software package.

Bibliography

[1] iContainers. *The future of automation at terminals and ports.* https://www.icontainers.com/us/2018/10/09/the-future-of-automation-at-terminals-and-ports/. accessed 2019-05-21.

[2] Port of Kristiansand. *Containers.* https://www.portofkristiansand.no/container. accessed 2019-01-10.

[3] More Than Shipping. *SHIPPING INDUSTRY TRENDS TO LOOK OUT FOR IN 2019.* https://www.morethanshipping.com/shipping-industry-trends-to-look-out-for-in-2019/. accessed 2019-02-17.

[4] FORTUNE. *U.S. Ports Take Baby Steps in Automation as Rest of the World Sprints.* http://fortune.com/2018/01/30/port-automation-robots-container-ships/. accessed 2019-02-17.

[5] Port Strategy. *STRADS VERSUS STACKERS.* https://www.portstrategy.com/news101/port-operations/cargo-handling/strads_and_stackers_get_to_grips_with_key_issues. accessed 2019-02-17.

[6] kisspng. *Reach Stacker.* https://www.kisspng.com/png-forklift-reach-stacker-tire-intermodal-container-m-4658197/download-png.html. accessed 2019-02-04.

[7] konecranesusa. *Straddle Carrier.* https://www.konecranesusa.com/sites/default/files/gallery/jpglarge_3d_boxrunner.jpg. accessed 2019-01-10.

[8] Kalmar. *Autonomous container terminals vs. self-driving cars: differences and similarities.* https://www.kalmarglobal.com/news--insights/2018/20181219_autonomous-container-terminals-vs.-self-driving-cars-differences-and-similarities/. accessed 2019-01-10.

[9] Amir Hossein Gharehgozli. “Sea Container Terminals: New Technologies, OR models, and Emerging Research Areas”. ERIM Report dissertation. Rotterdam School of Management, 2014.

[10] RBS EMEA UG. *Container Terminal Operating System.* https://www.rbs-emea.com/glossary/container-terminal-operating-system-tos/. accessed 2019-04-04.

[11] PEMA. *Container Terminal Automation.* https://www.pema.org/wp-content/uploads/downloads/2016/06/PEMA-IP12-Container-Terminal-Automation.pdf. accessed 2019-04-04.

[12] Open Source Robot Foundation. *ROS, Documentation.* http://wiki.ros.org/. accessed 2019-01-25.

[13] Open Source Robot Foundation. *ROS, Nodes.* http://wiki.ros.org/Nodes. accessed 2019-01-25.

[14] Open Source Robot Foundation. *ROS, Topics.* http://wiki.ros.org/Topics. accessed 2019-01-25.

[15] Open Source Robot Foundation. *ROS, Messages.* http://wiki.ros.org/Messages. accessed 2019-01-25.

[16] Open Source Robot Foundation. *rviz*. http://wiki.ros.org/rviz. accessed 2019-02-14.

[17] Open Source Robot Foundation. *QA - Get odometry from car-like robot*. https://answers.ros.org/question/235519/get-odometry-from-car-like-robot/. accessed 2019-02-25.

[18] OpenCV. *Camera Calibration*. https://docs.opencv.org/3.4/dc/dbb/tutorial_py_calibration.html. accessed 2019-02-11.

[19] Hindawi. *Depth Measurement Based on Infrared Coded Structured Light*. https://www.hindawi.com/journals/js/2014/852621/. accessed 2019-02-19.

[20] Miles Hansard. *Time of Flight Cameras: Principles, Methods, and Applications*. https://hal.inria.fr/file/index/docid/725654/filename/TOF.pdf. accessed 2019-02-19.

[21] Fei-Fei Li. *Lecture 9 10: Stereo Vision*. http://vision.stanford.edu/teaching/cs131_fall1415/lectures/lecture9_10_stereo_cs131.pdf. accessed 2019-02-19.

[22] Teknisk Ukeblad. *Slik styres gravemaskinen til siste centimeter*. https://www.tu.no/artikler/slik-styres-gravemaskinen-til-siste-centimeter/258240. accessed 2019-02-06.

[23] Hermann Eul. *Navigation and Communication 2006*. http://wpnc.net/fileadmin/WPNC06/CFP_WPNC06.pdf. accessed 2019-02-20.

[24] DANIEL RUBINO. *GPS vs. aGPS: A Quick Tutorial*. https://www.windowscentral.com/gps-vs-agps-quick-tutorial. accessed 2019-02-20.

[25] Øystein B. Dick. *UTM*. https://snl.no/UTM. accessed 2019-02-16.

[26] Wikipedia. *File:LA2-Europe-UTM-zones.png*. https://en.wikipedia.org/wiki/Universal_Transverse_Mercator_coordinate_system#/media/File:LA2-Europe-UTM-zones.png. accessed 2019-02-16.

[27] doxygen. *Detection of ArUco Markers*. https://docs.opencv.org/3.1.0/d5/dae/tutorial_aruco_detection.html. accessed 2019-02-04.

[28] Bilgin's Blog. *Kalman Filter For Dummies*. http://bilgin.esme.org/BitsAndBytes/KalmanFilterforDummies. accessed 2019-03-21.

[29] Bzarg. *How a Kalman filter works, in pictures*. http://www.bzarg.com/p/how-a-kalman-filter-works-in-pictures/. accessed 2019-03-22.

[30] Peng Wang. *A loop closure improvement method of Gmapping for low cost and resolution laser scanner*. https://www.sciencedirect.com/science/article/pii/S2405896316308278. accessed 2019-03-22.

[31] Kamarulzaman Kamarudin. *Performance Analysis of the Microsoft Kinect Sensor for 2D Simultaneous Localization and Mapping (SLAM) Techniques*. https://www.ncbi.nlm.nih.gov/pmc/articles/PMC4299068/. accessed 2019-03-22.

[32] S. Kohlbrecher et al. "A Flexible and Scalable SLAM System with Full 3D Motion Estimation". In: *Proc. IEEE International Symposium on Safety, Security and Rescue Robotics (SSRR)*. IEEE. Nov. 2011.

[33] Open Source Robot Foundation. *ROS - Navigation*. http://wiki.ros.org/navigation. accessed 2019-02-28.

[34] Open Source Robot Foundation. *move_base*. http://wiki.ros.org/move_base. accessed 2019-03-28.

[35] Tully Foote. "tf: The transform library". In: *Technologies for Practical Robot Applications (TePRA), 2013 IEEE International Conference on*. Open-Source Software workshop. Apr. 2013, pp. 1–6. DOI: 10.1109/TePRA.2013.6556373.

[36] Open Source Robot Foundation. *Obstacle Avoidance and Robot Footprint Model*. http://wiki.ros.org/teb_local_planner/Tutorials/Obstacle%20Avoidance%20and%20Robot%20Footprint%20Model. accessed 2019-04-05.

[37] Open Source Robot Foundation. *dwa_local_planner*. http://wiki.ros.org/dwa_local_planner. accessed 2019-04-05.

[38] Open Source Robot Foundation. *teb_local_planner*. http://wiki.ros.org/teb_local_planner. accessed 2019-02-25.

[39] Christoph Rösmann, Frank Hoffmann, and Torsten Bertram. "Integrated online trajectory planning and optimization in distinctive topologies". In: *Robotics and Autonomous Systems* 88 (2017), pp. 142–153. ISSN: 0921-8890. DOI: https://doi.org/10.1016/j.robot.2016.11.007. URL: http://www.sciencedirect.com/science/article/pii/S0921889016300495.

[40] Christoph Rösmann. "Integrated Online Trajectory Planning and Optimization in Distinctive Topologies". Paper. Technical University of Dortmund, 2016.

[41] tutorialspoint. *SDLC - V-Model*. https://www.tutorialspoint.com/sdlc/sdlc_v_model.htm. accessed 2019-01-10.

[42] NVIDIA. *Harness AI at the Edge with the Jetson TX2 Developer Kit*. https://developer.nvidia.com/embedded/buy/jetson-tx2-devkit. accessed 2018-12-13.

[43] SLAMTEC. *RPLiDAR A3*. http://www.slamtec.com/en/lidar/A3. accessed 2019-01-11.

[44] Intel®. *Intel® RealSense™ Depth Camera D435i*. https://click.intel.com/intel-realsense-depth-camera-d435i-imu.html. accessed 2019-01-11.

[45] kisspng. *Mercedes-Benz Actros*. https://www.kisspng.com/png-mercedes-benz-actros-mercedes-benz-c-class-car-tru-3586601/download-png.html. accessed 2019-02-07.

[46] Benjamin Vedder. *VESC – Open Source ESC*. http://vedder.se/2015/01/vesc-open-source-esc/. accessed 2019-02-07.

[47] FlipSky. *Mini FSESC4.20 50A base on VESC® 4.12 with Aluminum Anodized Heat Sink*. https://flipsky.net/collections/electronic-products/products/mini-fsesc4-20-50a-base-on-vesc-widely-used-in-eskateboard-escooter-ebike. accessed 2019-04-05.

[48] SkyRC. *540ARES PRO V2*. https://www.skyrc.com/AresPro. accessed 2019-02-07.

[49] Elefun. *SkyRC Ares Pro V2 1/10 Sensor 2860KV 13.5T MOD*. https://www.elefun.no/p/prod.aspx?v=41833. accessed 2019-04-05.

[50] Hobby Gulf. *POWER HD METAL GEAR SERVO 9.8KG / 56G HD-9001MG*. https://www.hobbygulf.com/power-hd-metal-gear-servo-9-8kg-56g-hd-9001mg.html. accessed 2019-04-05.

[51] Arduino. *Arduino Homepage*. https://www.arduino.cc/. accessed 2019-05-04.

[52] Adafruit. *Adafruit 16-Channel 12-bit PWM/Servo Driver - I2C interface - PCA9685*. https://www.arduino.cc/. accessed 2019-05-04.

[53] SparkFun. *SparkFun 9DoF Razor IMU M0*. https://www.sparkfun.com/products/14001. accessed 2019-03-15.

[54] Tamiya. *Mercedes-Benz Actros 3363 6x4 Assembly Manual*. https://d1hu0eys0tj9xi.cloudfront.net/media/files/56352ml-1051-2981.pdf. accessed 2019-01-11.

[55] NVIDIA. *JetPack 3.3 Release Notes*. https://developer.nvidia.com/embedded/jetpack-3_3. accessed 2019-01-08.

[56] OpenCV. *OpenCV*. https://opencv.org/. accessed 2019-02-13.

[57] OpenCV. *Install OpenCV-Python in Ubuntu*. https://docs.opencv.org/3.4.1/d2/de6/tutorial_py_setup_in_ubuntu.htmll. accessed 2019-01-19.

[58] Open Source Robot Foundation. *Ubuntu install of ROS Kinetic*. http://wiki.ros.org/kinetic/Installation/Ubuntu. accessed 2019-01-08.

[59] ros-teleop. *teleop_twist_keyboard*. https://github.com/ros-teleop/teleop_twist_keyboard. accessed 2019-01-12.

[60] RoboPeak. *rplidar_ros*. https://github.com/robopeak/rplidar_ros. accessed 2019-01-11.

[61] raess1. *Vesc-ROS-FW-3.33*. https://github.com/brentyi/vesc. accessed 2019-01-28.

[62] Open Source Robot Foundation. *rosserial*. http://wiki.ros.org/rosserial. accessed 2019-02-04.

[63] KristofRobot. *ROS Razor 9DoF IMU*. https://github.com/KristofRobot/razor_imu_9dof. accessed 2019-03-19.

[64] Charles River Analytics, Inc. *robot_localization*. https://github.com/cra-ros-pkg/robot_localization. accessed 2019-02-28.

[65] Open Source Robot Foundation. *ros_navigation*. https://github.com/ros-planning/navigation. accessed 2019-02-28.

[66] Stefan Kohlbreche. *hector_slam*. https://github.com/ros-planning/navigation. accessed 2019-03-28.

[67] rst-tu-dortmund. *teb_local_planner*. https://github.com/rst-tu-dortmund/teb_local_planner. accessed 2019-03-28.

[68] danielsnider. *follow_waypoints*. https://github.com/danielsnider/follow_waypoints. accessed 2019-04-27.

[69] Intel ROS. *ROS Wrapper for Intel® RealSense™ Devices*. https://github.com/intel-ros/realsense. accessed 2019-01-11.

[70] Intel. *Intel® RealSense™ SDK*. https://github.com/IntelRealSense/librealsense. accessed 2019-01-11.

[71] Open Source Robot Foundation. *teleop_twist_keyboard*. http://wiki.ros.org/teleop_twist_keyboard. accessed 2019-01-12.

[72] OpenCV. *ArUco Tracker*. https://github.com/njanirudh/Aruco_Tracker. accessed 2019-02-11.

[73] CCTV Infomation. *Lenses for Video Motion Detection Systems*. https://www.cctv-information.co.uk/i/Lenses_for_Video_Motion_Detection_Systems. accessed 2019-03-14.

[74] Tom Moore. *Working with the robot_localization Package*. https://roscon.ros.org/2015/presentations/robot_localization.pdf. accessed 2019-02-16.

[75] Tom Moore. *State Estimation Nodes*. http://docs.ros.org/melodic/api/robot_localization/html/state_estimation_nodes.html. accessed 2019-03-21.

[76] Open Source Robot Foundation. *Planning for car-like robots*. http://wiki.ros.org/teb_local_planner/Tutorials/Planning%20for%20car-like%20robots. accessed 2019-02-28.

[77] Open Source Robot Foundation. *global_planner*. http://wiki.ros.org/global_planner?distro=melodic. accessed 2019-02-13.

[78] Open Source Robot Foundation. *amcl*. http://wiki.ros.org/amcl?distro=melodic. accessed 2019-02-13.

[79] Open Source Robot Foundation. *gmapping*. http://wiki.ros.org/gmapping. accessed 2019-03-15.

[80] Open Source Robot Foundation. *hectro_slam*. http://wiki.ros.org/hector_slam. accessed 2019-03-25.

[81] Open Source Robot Foundation. *follow_waypoints*. http://wiki.ros.org/follow_waypoints. accessed 2019-04-27.

[82] Scott Pendelton. *Multi-class Driverless Vehicle Cooperation for Mobility-on-Demand*. https://www.researchgate.net/publication/290304568_Multi-class_driverless_vehicle_cooperation_for_mobility-on-demand. accessed 2019-03-23.

[83] MazeMap. *MazeMap*. https://use.mazemap.com/. accessed 2019-04-05.

[84] Amir Ghahrai. *Incremental Model*. https://www.testingexcellence.com/incremental-model/. accessed 2019-05-23.

[85] Open Source Robot Foundation. *sbpl_lattice_planner*. http://wiki.ros.org/sbpl_lattice_planner. accessed 2019-04-24.

[86] Open Source Robot Foundation. *carrot_planner*. http://wiki.ros.org/carrot_planner. accessed 2019-04-24.

[87] Open Source Robot Foundation. *geonav_transform*. http://wiki.ros.org/geonav_transform. accessed 2019-02-20.

A. User Manual

This Appendix includes the user manual for the prototype developed in this thesis. The user manual contains start-up procedure, trouble shooting and known issues.

A.1 Prototype Start-Up

To initiate the prototype the following steps has to be performed:

- Connect each of the three batteries. The 4S LiPo battery has to be connected to the Jetson's power-input. The 7.5V NiMH battery has to be connected to the VESC. Finally connect the alkaline battery package to the servo driver.
- Make sure that every USB cable is connected to the USB-hub
- Press the power button on the Jetson and wait few seconds for it to boot
- When the Jetson has booted, it will pop-up as an *wireless hotspot* on your Ubuntu laptops network manager. Connect to the hotspot to gain access to the prototype.
- When the connection is established, the user can utilise *Remmina Remote Desktop Client* to stream the display output on the Jetson to the laptop connected to the hotspot
- On the Ubuntu laptop, open a new terminal (`ctrl + alt + t`)
- Check laptop IP using `ifconfig` in the terminal
- Export as ROS MASTER URI and ROS IP using `export ROS_MASTER_URI=http://<MASTER IP ADDRESS>:11311` and `export ROS_IP=<IP ADDRESSE>`
- In the terminal, run the command `roscore` to start the ROS Master-node
- In Remmina, open a new terminal on the Jetson and run the command `roslaunch rr_racer bringup.launch` to start all the hardware communication (Arduino, VESC, RPLiDAR, IMU), in addition to launch the `static_transform`-node and the Ackermann transform publisher node.
- The prototype is now initiated and ready to be driven around manually.

After running the `bringup.launch`-file the user can start generate a map using SLAM, or start an autonomous driving sequence to a single goal or drive continuous through user defined waypoints.

A.1.1 SLAM

To start SLAM with `hector_slam`, the following command has to be executed in a new terminal window:

```
$ roslaunch rr_racer hector_mapping.launch
```

The user will be greeted with a RViz-window setup to with the necessary displays for SLAM.

A.1.2 Point-to-Point Autonomous Driving

For the prototype to navigate autonomously in a provided known map from an initial pose, to a user defined goal, the following command had to be executed in a new terminal window:

```
$ roslaunch rr_racer move_base_teb.launch
```

The user will be greeted with a RViz-wind setup to with the necessary displays for autonomous driving. The user has to provide the prototype with an rough estimate in the map with the *2D Pose Estimation*-tool. The goal pose can now be provided by changing the *Fixed Frame*-topic to the `/odom`-topic in the top left corner, under *Global Options*, and using the *2D Nav Goal*-tool and click on the map.

A.1.3 Continuous Autonomous Driving

To let the prototype run continuously through user defined waypoints with dynamic obstacle avoidance, the user has to execute the following command in a new terminal window:

```
$ roslaunch rr_racer continious_waypoints.launch
```

Then the user will be greeted with a RViz-window setup to with the necessary displays for continuous autonomous driving. The user can now provide as many waypoints as desired, as long as they are spaced evenly throughout the map, with the *2D Pose Estimation*-tool. Finally the user has to execute the following command in another terminal window to start the continuous driving:

```
$ rostopic pub /path_ready std_msgs/Empty -1
```

The prototype will start to move after a few seconds and its progress can be monitored in RViz.

A.2 Indoor Localization Start-Up

To start the indoor camera-based localization system, the camera has to be connected to a computer via a USB 3.0-port. Secondly the computes has to connect to the WiFi-hotspot created by the NVIDIA Jetson on the prototype. Then run the following command in the terminal on the computer to launch the RealSense-node:

```
$ roslaunch realsense2_viewer rs_camera.launch
```

When the node is started, the ArUco tracker-node run the python-script by changing directory into the *script*-folder and execute:

```
$ python aruco_pub.py
```

Lastly, the relative odometry generation node has to be started by running the script from the terminal:

```
$ python odom_relative_gen.py
```

If everything has been stated correctly, the computer should now publish the generated odometry to the correct topic.

A.3 Battery & Charging

The prototype has three batteries: one 4S LiPo, one 7.5V NiHM and a 4x AA-battery pack. Both the LiPo and NiHM is rechargeable, and should be charged frequently. The AA-battery pack, which provides power to the steering servo, is not rechargeable. If the servo does not turn the steering-wheels, it could mean that the AA battery-pack is discharged.

A.4 Known Issues

This section will describe the know issues with the prototype and how to solve them.

A.4.1 Recovery Behaviour

Sometimes the prototype ”get stuck” due to obstacle in the costmap, this forces it into *recovery behaviour*. As of right the recovery behaviour does not work as intended, due to there is not implemented any Ackermann specific behaviours. To unstuck the prototype, simply run the manual teleoperation and force it into reverse. By doing this a couple of times, the prototype tends to clear the costmap enough for it to resume autonomous driving.

A.4.2 Steering Servo Power Supply

The power supply for the steering servo has a tendency to loosen over time. In addition it also seem to have poor contact, thus require the user to squeeze the connection to ensure good contact.

A.4.3 ttyAMC

Due to the Jetson only having a single USB-port, a USB-hub is utilized. The IMU and VESC are both connected to the Jetson through the USB-hub, which sometime caused problems for the `ttyAMC`-ports they are provided in the source files. By default, the IMU is set to `ttyAMC1` and the VESC to `ttyACM2`. If the `bringup.launch` has an error, try swapping the `ttyAMC`-port of the IMU with the VESC. The ports can be changed for the VESC in the `vesc_driver_node.launch` and the `my_razor.yaml` parameter-file for the IMU.

B. Scripts

This Appendix include all the scripts utilized in this thesis.

B.1 Low-Level Control

```
#!/usr/bin/env python
#
# Author:   jksllk
# Version:  04.03.19
#
#       Converts Twist messages into
#       VESC and servo commands
#
#
# Update:
#
#       Added friction compensation
#
# ------------------------------------

import rospy, time
from std_msgs.msg import Float64, UInt16
from geometry_msgs.msg import Twist


global vesc_max, vesc_min, servo_max, servo_min, k_m, b
# Motor max/min
vesc_max = 0.3
vesc_min = -0.3

# Servo max/min
servo_max = 350
servo_min = 100

# Motor friction
k_m = 2.4
b = 0.06


```

```python
class LowLvlCtrl():
    # Low level control for RC Truck in ROS
    def __init__(self):
        rospy.loginfo("Setting up the node..")
        rospy.init_node("lowlvlctrl")

        # Create Publishers
        self.vesc_pub=rospy.Publisher('commands/motor/duty_cycle', Float64, queue_size=10)
        self.servo_pub=rospy.Publisher('/servo_pos', UInt16, queue_size=10)

        # Create Subscriber to /cmd_vel
        self.twist_sub=rospy.Subscriber("/cmd_vel", Twist, self.set_actuator_from_cmdvel)
        rospy.loginfo("> Subscirber correctly initizlized")

        # Save last time we got a reference
        self._last_cmd = time.time()
        self._timeout_s = 5

        rospy.loginfo("Initizlization complete")

    # SERVO CONTROL
    def servo_value_out(self, servo_in):
        """
        Given an input refreance in [-1, 1], it converts it in the actual range
        """
        servo_in=-servo_in
        center_val = 226
        range = 350
        half_range = 0.5*range

        self.servo_out = int(servo_in*half_range + center_val)
        # Set max min range
        if self.servo_out > servo_max:
            self.servo_out = servo_max

        if self.servo_out < servo_min:
            self.servo_out = servo_min

```

```
        return self.servo_out


    # VESC CONTROL
    def vesc_value_out(self, vesc_in):
        """
        Given an input refreance in [-1, 1], it converts it in the actual range

        Compensate for measured motor friction
        """
        # Compensate for friciton
        if vesc_in > 0:
            self.vesc_out = k_m*0.1*vesc_in + b

        if vesc_in == 0:
            self.vesc_out = 0

        if vesc_in < 0:
            self.vesc_out = k_m*0.1*vesc_in - b

        # Set max min range
        if self.vesc_out > vesc_max:
            self.vesc_out = vesc_max

        if self.vesc_out < vesc_min:
            self.vesc_out = vesc_min

        return self.vesc_out


    def set_actuator_from_cmdvel(self, message):
        """
        Get a Twist message from cmd_vel, assuming max input is 1
        """
        # Save time
        self._last_cmd = time.time()

        # Convert vel into servo
        servo_msg = self.servo_value_out(message.angular.z) # positive rgt
        # Convert vel into VESC
```

```
        vesc_msg = self.vesc_value_out(message.linear.x)

        # Publish the message using a function
        self.vesc_pub.publish(vesc_msg)
        self.servo_pub.publish(servo_msg)


    def run(self):
        # Set control rate
        rate = rospy.Rate(30)

        while not rospy.is_shutdown():
            #print(self._last_cmd)
            rate.sleep()


"""
Execute the main file
"""
if __name__ == '__main__':
    lowlvlctrl = LowLvlCtrl()
    lowlvlctrl.run()
```

B.2 OpenCV Camera Calibration

```
import numpy as np
import cv2
import glob


WAIT_TIME = 1000

# termination criteria
criteria = (cv2.TERM_CRITERIA_EPS + cv2.TERM_CRITERIA_MAX_ITER,
    27, 0.001) # Square size 25mm

# prepare object points, like (0,0,0), (1,0,0), (2,0,0)
    ....,(6,5,0)
objp = np.zeros((9*6,3), np.float32)
            # 6x9
objp[:,:2] = np.mgrid[0:6,0:9].T.reshape(-1,2)
            # 6x9

# Arrays to store object points and image points from all the
    images.
objpoints = [] # 3d point in real world space
imgpoints = [] # 2d points in image plane.

images = glob.glob('calib_images/rs_2/*.png')

for fname in images:
    img = cv2.imread(fname)
    gray = cv2.cvtColor(img,cv2.COLOR_BGR2GRAY)

    # Find the chess board corners
    ret, corners = cv2.findChessboardCorners(gray, (9,6),None) #
        9,6 Grid size of checker board

    # If found, add object points, image points (after refining
        them)
    if ret == True:
        objpoints.append(objp)

        corners2 = cv2.cornerSubPix(gray,corners,(11,11),(-1,-1),
            criteria)
```

```
        imgpoints.append(corners2)

        # Draw and display the corners
        img = cv2.drawChessboardCorners(img, (6,9), corners2,ret)
            # 6,9 grid
        cv2.imshow('img',img)
        cv2.waitKey(WAIT_TIME)

cv2.destroyAllWindows()
ret, mtx, dist, rvecs, tvecs = cv2.calibrateCamera(objpoints,
    imgpoints, gray.shape[::-1],None,None)

print(ret)
print(mtx)
print(dist)
print(rvecs)
print(tvecs)
cv_file = cv2.FileStorage("calib_images/test.yaml", cv2.
    FILE_STORAGE_WRITE)
cv_file.write("camera_matrix", mtx)
cv_file.write("dist_coeff", dist)
# note you *release* you don't close() a FileStorage object
cv_file.release()
```

B.3 ArUco Marker Detection

```python
#!/usr/bin/env python
# Author: jksllk
# Version: 18.03.19
#
#       First run roscore and roslaunch realsense2_camera rs_rgbd.launch
#       to initialize the camera nodes.
#
#       This script will detect ArUco markers and publish its pixel coordinates
#       to a topic.
#
# ----------------------------------------------------------------------------------------
import roslib
import sys
import rospy
import cv2
import numpy as np
from numpy import *
from cv_bridge import CvBridge, CvBridgeError
from sensor_msgs.msg import Image
import cv2.aruco as aruco
import glob
import sys, tty, termios, time
from std_msgs.msg import Int32, Float32
import matplotlib.pyplot as plt

# Create the ArUco Tracker class
class ArucoTracker(object):
    # Initialize nodes, publishers and subsrcibers
    def __init__(self):
        self.bridge_object = CvBridge()
        # Create a subsrciber to the RGB camera topic
        self.image_sub = rospy.Subscriber("/camera/color/image_raw", Image, self.camera_callback)
        # Create a publisher to publish ArUco corner coordinates
        self.x1_pub = rospy.Publisher("/aruco/corner1/x", Int32, queue_size=10)
```

```
        self.y1_pub = rospy.Publisher("/aruco/corner1/y", Int32, queue_size=10)
        self.x2_pub = rospy.Publisher("/aruco/corner2/x", Int32, queue_size=10)
        self.y2_pub = rospy.Publisher("/aruco/corner2/y", Int32, queue_size=10)
        self.x3_pub = rospy.Publisher("/aruco/corner3/x", Int32, queue_size=10)
        self.y3_pub = rospy.Publisher("/aruco/corner3/y", Int32, queue_size=10)
        self.x4_pub = rospy.Publisher("/aruco/corner4/x", Int32, queue_size=10)
        self.y4_pub = rospy.Publisher("/aruco/corner4/y", Int32, queue_size=10)
        # Create publisher for ArUco translation
        self.tx_pub = rospy.Publisher("/aruco/trans/x", Float32, queue_size=10)
        self.ty_pub = rospy.Publisher("/aruco/trans/y", Float32, queue_size=10)
        self.tz_pub = rospy.Publisher("/aruco/trans/z", Float32, queue_size=10)
        # Create publisher for ArUco rotation
        self.rx_pub = rospy.Publisher("/aruco/rot/x", Float32, queue_size=10)
        self.ry_pub = rospy.Publisher("/aruco/rot/y", Float32, queue_size=10)
        self.rz_pub = rospy.Publisher("/aruco/rot/z", Float32, queue_size=10)
        # Create pixel publisher
        self.cx_pub = rospy.Publisher("/aruco/pixel/x", Float32, queue_size=10)
        self.cy_pub = rospy.Publisher("/aruco/pixel/y", Float32, queue_size=10)
        rospy.loginfo("Node Initialized")
        rospy.loginfo(">> Tracking ArUco")

    # Callback function for RGB camera
    def camera_callback(self, data):
        try:
            # Select brg8 because its the OpenCV encoding by default
            cv_image = self.bridge_object.imgmsg_to_cv2(data,
```

```
                desired_encoding="bgr8")

        except CvBridgeError as e:
            print(e)

        # Set cap to cv image from topic
        cap = cv_image

        while (True):
            # Operations on the frame come here
            gray = cv2.cvtColor(cap, cv2.COLOR_BGR2GRAY)                 # Converte to gray-scale
            aruco_dict = aruco.Dictionary_get(aruco.DICT_6X6_250)        # Find 6x6 ArUco Code
            parameters = aruco.DetectorParameters_create()               # Detet parametes
            # Lists of ids and the corners beloning to each id
            corners, ids, rejectedImgPoints = aruco.detectMarkers(gray, aruco_dict, parameters=parameters)
            font = cv2.FONT_HERSHEY_SIMPLEX     # Font for displaying text (below)
            if np.all(ids != None):
                # Estimate pose of each marker and return the values rvet and tvec---different from camera coefficients
                rvec, tvec, _ = aruco.estimatePoseSingleMarkers(corners[0], 0.18, mtx, dist) # 0.18 = maker side length
                cor=aruco.drawAxis(cap, mtx, dist, rvec[0], tvec[0], 0.05)    # Draw Axis
                aruco.drawDetectedMarkers(cap, corners)                 # Draw A square around the markers
                #rospy.loginfo(corners)         # Top left(xy), top right (xy), bottom right (xy), bottom left (xy)
                rot_mtx = np.zeros(shape=(3,3))
                cv2.Rodrigues(rvec,rot_mtx)
                rect = cv2.minAreaRect(corners[0])
                box = cv2.boxPoints(rect)
                area = cv2.contourArea(box)
                # Find pixe coordinates of center to ArUco
```

```
            C = cv2.moments(box)
            px = int(C["m10"] / C["m00"])
            py = int(C["m01"] / C["m00"])
            #print(px,py)
            # Draw ID
            cv2.putText(cap, "Id: " + str(ids), (0,64), font,
                1, (0,255,0),2,cv2.LINE_AA)
            # Generate list from array
            a=corners
            b = a[0][0]
            c = b.ravel()
            d = list(c)

            # Get translation values
            tx=tvec[0][0][0]
            ty=tvec[0][0][1]
            tz=tvec[0][0][2]
            # Get rotation values
            rx=rvec[0][0][0]
            ry=rvec[0][0][1]
            rz=rvec[0][0][2]

            # Publish corners to topic
            self.x1_pub.publish(int(d[0]))
            self.y1_pub.publish(int(d[1]))
            self.x2_pub.publish(int(d[2]))
            self.y2_pub.publish(int(d[3]))
            self.x3_pub.publish(int(d[4]))
            self.y3_pub.publish(int(d[5]))
            self.x4_pub.publish(int(d[6]))
            self.y4_pub.publish(int(d[7]))
            #Publish translation values top topic
            self.tx_pub.publish(tx)
            self.ty_pub.publish(ty)
            self.tz_pub.publish(tz)
            #Publish rotation values top topic
            self.rx_pub.publish(rx)
            self.ry_pub.publish(ry)
            self.rz_pub.publish(rz)
            # Publish ArUco's center piexl coordinates
            self.cx_pub.publish(px)
            self.cy_pub.publish(py)
```

```
                #print(tx,ty)

            # Create grid in image
            cv2.line(cv_image, (0,240),(640,240), (0, 255, 0), 2)
            cv2.line(cv_image, (320,0),(320,480), (0, 255, 0), 2)
            # Show image
            cv2.imshow("Image Window", cv_image)
            cv2.waitKey(1)
            break

# Main function
def main():
    rospy.sleep(0.1)
    rospy.init_node("aruco_tracker_node", anonymous=True)
    aruco_tacker_object=ArucoTracker()
    rate = rospy.Rate(10)  # 10 Hz refresh rate
    try:
        while not rospy.is_shutdown():
            rate.sleep()

    except KeyboardInterrupt:
        print("Shutting down")


if __name__ == '__main__':
    # Frist run camera calibration
    # Termination criteria
    criteria = (cv2.TERM_CRITERIA_EPS + cv2.TERM_CRITERIA_MAX_ITER, 27, 0.001) # 25 mm sqare
    # Prepare object points, like (0,0,0), (1,0,0), (2,0,0) ....,(6,5,0)
    objp = np.zeros((9*6,3), np.float32)                          # 9x6
    objp[:,:2] = np.mgrid[0:6,0:9].T.reshape(-1,2)                # 9x6
    # Arrays to store object points and image points from all the images.
    objpoints = [] # 3d point in real world space
    imgpoints = [] # 2d points in image plane.
    images = glob.glob('calib_images/rs_2/*.png')

    for fname in images:
```

```
        img = cv2.imread(fname)
        gray = cv2.cvtColor(img,cv2.COLOR_BGR2GRAY)
        # Find the chess board corners
        ret, corners = cv2.findChessboardCorners(gray, (6,9),None)      # 9x6
        # If found, add object points, image points (after refining them)

        if ret == True:
            objpoints.append(objp)
            corners2 = cv2.cornerSubPix(gray,corners,(11,11),(-1,-1),criteria)
            imgpoints.append(corners2)
            # Draw and display the corners
            img = cv2.drawChessboardCorners(img, (6,9), corners2,ret)           # 9x6

    ret, mtx, dist, rvecs, tvecs = cv2.calibrateCamera(objpoints,imgpoints, gray.shape[::-1],None,None)

    # Run main function
    main()
```

B.4 ArUco Depth Measure

```
#!/usr/bin/env python
# Author: jksllk
# Version: 21.02.19
#
#       This script will recive pixel coordinates by subsrcibing to the
#       different topics and give actual depth from RealSense Camera.
#
# ----------------------------------------------------------------------
import roslib
import sys
import rospy
import cv2
import cv2.aruco as aruco
from cv_bridge import CvBridge, CvBridgeError
import numpy as np
from numpy import *
import glob
import sys, tty, termios, time
from sensor_msgs.msg import Image
from std_msgs.msg import Int32, Float32

# Define global values
global sqr, edg
sqr = 5    # Define size of sqare in pixels to probe depth data from ROI
edg = 0    # Reduce ArUco area in pixels

# Create Depth Camera class
class DepthCamera(object):
    # Initialize node, pusblishers and subsrcibers
    def __init__(self):
        rospy.sleep(1)
        # Define CvBridge objects
        self.bridge_depth_object=CvBridge()
        self.bridge_rgb_object=CvBridge()
        # Create pixel coordinate subsrcibers
```

```
        self.x1_sub=rospy.Subscriber("/aruco/corner1/x", Int32, self.x1)
        self.y1_sub=rospy.Subscriber("/aruco/corner1/y", Int32, self.y1)
        self.x2_sub=rospy.Subscriber("/aruco/corner2/x", Int32, self.x2)
        self.y2_sub=rospy.Subscriber("/aruco/corner2/y", Int32, self.y2)
        self.x3_sub=rospy.Subscriber("/aruco/corner3/x", Int32, self.x3)
        self.y3_sub=rospy.Subscriber("/aruco/corner3/y", Int32, self.y3)
        self.x4_sub=rospy.Subscriber("/aruco/corner4/x", Int32, self.x4)
        self.y4_sub=rospy.Subscriber("/aruco/corner4/y", Int32, self.y4)
        # Create a subsrciber to the Depth camera topic
        self.depth_image_sub=rospy.Subscriber("/camera/depth/image_rect_raw", Image, self.depth_camera_callback)
        self.rgb_image_sub=rospy.Subscriber("camera/color/image_raw", Image, self.rgb_callback)
        # Create publisher to publish ArUco depth (z) from depth camera_callback
        self.depth_pub=rospy.Publisher("/aruco/depth/avg_depth", Float32, queue_size=10)
        rospy.loginfo("Node Initialized")
        rospy.loginfo(">> Measuring Distance to ArUco")


    # Get values from subsrcibers
    def x1(self, msg):
        global x_1
        x_1=msg.data-edg

    def x2(self, msg):
        global x_2
        x_2 = msg.data-edg

    def x3(self, msg):
        global x_3
        x_3=msg.data-edg

```

```
    def x4(self, msg):
        global x_4
        x_4 = msg.data-edg

    def y1(self, msg):
        global y_1
        y_1=msg.data-edg

    def y2(self, msg):
        global y_2
        y_2 = msg.data-edg

    def y3(self, msg):
        global y_3
        y_3=msg.data-edg

    def y4(self, msg):
        global y_4
        y_4 = msg.data-edg


    # Region of intrest
    def region_of_intrest(self, image, p1, p2, p3, p4):
        # Create a polygon from corners
        polygon = np.array([[(p1), (p2), (p3), (p4)]])
        # Create bounding rectangle
        rect = cv2.boundingRect(polygon)
        x,y,w,h = rect
        # Crop image to rectangle
        croped = image[y:y+h, x:x+w].copy()
        return croped


    # Get depth camera data
    def depth_camera_callback(self, data):
        try:
            # Select brg8 because its the OpenCV encoding by
               default
            cv_depth_image=self.bridge_depth_object.imgmsg_to_cv2
               (data, desired_encoding="32FC1")

        except CvBridgeError as e:
```

```
            print(e)

        # Create region of intrest from function
        roi_img=self.region_of_intrest(cv_depth_image, (x_1, y_1)
           ,(x_2, y_2),(x_3, y_3),(x_4, y_4))
        # Get size of image. The size varies with where the
           marker is located
        height, width = roi_img.shape[:2]
        xmin=(width/2)-sqr
        xmax=(width/2)+sqr
        ymin=(height/2)-sqr
        ymax=(height/2)+sqr
        # Find depth value from ROI
        roi_val=roi_img[xmin:xmax, ymin:ymax]
        # Use only non-zero values. Remove noise
        val_non_zero=roi_val[roi_val != 0]
        # Calculate average distance over ROI
        avg_dist=sum(val_non_zero)/len(val_non_zero)
        #print(avg_dist)
        # Publish the average depth of ArUco marker
        self.depth_pub.publish(avg_dist)


    # Show RGB image to ilustrate where the camera is measuring
    def rgb_callback(self, data):
        try:
            # Select brg8 because its the OpenCV encoding by
               default
            rgb_img=self.bridge_rgb_object.imgmsg_to_cv2(data,
               desired_encoding="bgr8")

        except CvBridgeError as e:
            print(e)

        polygon = np.array([[(x_1, y_1),(x_2, y_2),(x_3, y_3),(
           x_4, y_4)]])
        # Draw polygon around ArUco
        cv2.polylines(rgb_img, polygon, False, (0, 255, 0),
           3)
        # Crop the bounding rect
        rect = cv2.boundingRect(polygon)
        x,y,w,h = rect
```

```
        rgb_croped = rgb_img[y:y+h, x:x+w].copy()
        # Get height and width date from image
        height, width = rgb_croped.shape[:2]
        # Create a square to read depth data from
        xmin=(width/2)-sqr
        xmax=(width/2)+sqr
        ymin=(height/2)-sqr
        ymax=(height/2)+sqr
        # Draw rectangle        Top left        Bottom rigth        Color    line thickness
        cv2.rectangle(rgb_croped, (xmin, ymin), (xmax, ymax), (255, 0, 0), 3)
        # View RGM image
        cv2.imshow("RGB", rgb_croped)
        cv2.waitKey(1)


if __name__=='__main__':
    # Sleep for 1 secon to allow data to be recived
    rospy.init_node("depth_camera_node", anonymous=True)
    depth_camera_object=DepthCamera()
    rate=rospy.Rate(10)  # 10 Hz refresh rate
    try:
        while not rospy.is_shutdown():
            rate.sleep()

    except KeyboardInterrupt:
        print("Shutting down")
    cv2.destroyAllWindowws()
```

B.5 Ackermann Odometry

```
#!/usr/bin/env python
#
# Author:   jksllk
# Version   30.03.19
#
#       Calculate Ackermann odometry from
#       Servo and VESC and publish transform
#
# ---------------------------------------------------------------

import math
from math import sin, cos, pi, tan
import time
import rospy
import tf
from std_msgs.msg import Float32, UInt16
from vesc_msgs.msg import VescStateStamped
from nav_msgs.msg import Odometry
from geometry_msgs.msg import Point, Pose, Quaternion, Twist,
   Vector3

# Define global paramerers
global r, i, L, dt, cov_x, cov_y, rcov_z
r=0.0425      # Wheel radius
i=17.761      # Gear ratio
L=0.34        # Wheel base
dt = 30       # Sample rate

# Covariance tuning
cov_x    = 0.1
cov_y    = 0.5
rcov_z   = 0.5


alpha = 0
m_vel = 0


def steering_angle_callback(msg):
```

```
    global alpha
    alpha = -(msg.data-224)*(0.5/125)   # Angle is 27 deg = 0.47 rad


def motor_vel_callback(msg):
    global m_vel
    m_vel = (msg.state.speed)*(2*pi/60) # Convert from rpm to rad/s


def main():
    # Init Node
    rospy.init_node("ackermann_odometry")
    rospy.loginfo("Initizalized Node")
    # Create Publishers
    odom_pub = rospy.Publisher("/odom", Odometry, queue_size=50)
    odom_broadcaster = tf.TransformBroadcaster()

    # Creat Subscribers
    sub_servo = rospy.Subscriber("/servo_pos", UInt16, steering_angle_callback)
    sub_vel = rospy.Subscriber("/sensors/core", VescStateStamped, motor_vel_callback)

    current_time = rospy.Time.now()
    last_time = rospy.Time.now()
    # Wait for data to be reviced
    rospy.sleep(1)
    rate = rospy.Rate(dt)

    # Set initial pose
    x_g = 0
    y_g = 0
    x_l = 0
    y_l = 0
    theta = 0

    rospy.loginfo(">> Process Started")
    while not rospy.is_shutdown():
        # Truning radius
        #R = L/(tan(alpha)+0.00000001)
```

```
        #print(R)
        # Tangential velocity
        v_s = (m_vel/i)*r

        #-- Local
        # Vel of robot center point in robot frame
        xdot_l = v_s*cos(alpha)
        ydot_l = 0 # No slip

        #-- Global
        theta_dot = (tan(alpha)/L)*xdot_l
        theta += theta_dot/dt

        xdot_g = xdot_l*cos(theta)
        ydot_g = xdot_l*sin(theta)

        x_l += xdot_l/dt
        y_l += ydot_l/dt
        x_g += xdot_g/dt
        y_g += ydot_g/dt

        # since all odometry is 6DOF we'll need a quaternion
           created from yaw
        odom_quat = tf.transformations.quaternion_from_euler(0,
           0, theta)

        # first, we'll publish the transform over tf
        odom_broadcaster.sendTransform(
            (x_g, y_g, 0),       # position
            odom_quat,           # ang
            rospy.Time.now(),    # time
            "base_link",         # child
            "odom"               # parrent
        )

        # next, we'll publish the odometry message over ROS
        odom = Odometry()
        odom.header.stamp = rospy.Time.now()
        odom.header.frame_id = "odom"

        # set the position
        odom.pose.pose = Pose(Point(x_g, y_g, 0.), Quaternion(*
```

```
            odom_quat))
        odom.pose.covariance =  [cov_x, 0,      0, 0, 0, 0,
                                 0,     cov_y,  0, 0, 0, 0,
                                 0,     0,      0, 0, 0, 0,
                                 0,     0,      0, 0, 0, 0,
                                 0,     0,      0, 0, 0, 0,
                                 0,     0,      0, 0, 0, rcov_z]

        # set the velocity
        odom.child_frame_id = "base_link"
        odom.twist.twist = Twist(Vector3(xdot_g, ydot_g, 0),
            Vector3(0, 0, theta_dot))

        # publish the message
        odom_pub.publish(odom)
        #print(odom)
        last_time = current_time

        rate.sleep()


if __name__ == '__main__':
    main()
```

B.6 ArUco Marker Covariance Generator

```
#!/usr/bin/env python
# Author: jksllk
# Version: 23.04.19
#
#       Run aruco_pub.py first
#
#
#       This script will track an ArUco marker and plot the position
#       in addition to genetating the Covaraince matrix of the tracker
#
# ---------------------------------------------------------------------------------------------

import sys
import time
import rospy
import math
import numpy as np
import matplotlib.pyplot as plt
from std_msgs.msg import Int32, Float32
from geometry_msgs.msg import Point, Pose, Quaternion, Vector3, Twist
from nav_msgs.msg import Odometry
from array import *

# Define global values
global dt
dt = 10 # rate


# Initialize variables
x         = 0
y         = 0
theta     = 0
tx        = 0
ty        = 0
xt        = 0
yt        = 0
```

```
theta_t  = 0

# Define values of variables from subscribers
def trans_x(msg):
    global tx
    tx = msg.data
    #print(tx)

def trans_y(msg):
    global ty
    ty = msg.data
    #print(ty)

def trans_z(msg):
    global tz
    tz = msg.data
    #print(tz)

def rot_x(msg):
    global rx
    rx = msg.data
    #print(rx)


# Generate Covariance
def covar_generator():
        rospy.init_node("covar_generation_node", anonymous=True)
        # Define subscribers to recive translation coordinate
           from topics
        # Get xyz coordinates from ArUco Tracker
        trans_x_sub=rospy.Subscriber("/aruco/trans/x", Float32,
           trans_x)
        trans_y_sub=rospy.Subscriber("/aruco/trans/y", Float32,
           trans_y)
        trans_z_sub=rospy.Subscriber("/aruco/trans/z", Float32,
           trans_z)
        # Only rotation which is nesessary is rotation about the
           x axis
        rot_x_sub=rospy.Subscriber("/aruco/rot/x", Float32, rot_x
           )
        # Sleep for 1 seconds to allow data to be recived
        rospy.sleep(1)
```

```
        # Define rate of script
        rate=rospy.Rate(dt) # 10 Hz
        # Coordinate message, position and orientation
        # Initialize variables
        xtt       = tx
        ytt       = -ty
        theta_tt  = rx
        x         = 0
        y         = 0
        theta     = 0
        xdot      = 0
        ydot      = 0
        theta_dot = 0
        t = 0
        tmax = 60*dt
        A = [] # Create empty matrix
        rospy.loginfo("Node Initialized")
        rospy.loginfo(">> Gathering Sample Data")
        try:
            while not rospy.is_shutdown():
                current_time = rospy.Time.now()
                # Set current values
                xt = tx
                yt = -ty
                theta_t = rx
                # Find change in position, by subtracting current
                    from previous
                xdot = xt - xtt
                ydot = yt - ytt
                theta_dot = theta_t - theta_tt
                # Update previous value
                xtt = xt
                ytt = yt
                theta_tt = theta_t
                # Update position
                x += xdot
                y += ydot
                # Angle is taken from the rotation of the ArUco
                theta += theta_dot
                data = [x, y, theta]
                #print(data)
                # Sample data
```

```
            x_i, y_i, th_i = data
            # Appender add to end of array
            A.append([x_i, y_i, th_i, t])

            # Count time up
            t += 1
            """
            # Progress bar
            prcnt = t
            print(prcnt)
            sys.stdout.write('\r[{0}] {1}%'.format('='*(prcnt/3), prcnt))
            sys.stdout.flush()
            """
            # Quit loop when time is up!
            if t >= tmax:
                break

            rate.sleep()

    except KeyboardInterrupt:
        print("Shutting down")

    # Done sampling
    rospy.loginfo(">>>> Done!")
    A = np.array(A)
    X  = A[:, 0]
    Y  = A[:, 1]
    TH = A[:, 2]
    T =  A[:, 3]/dt # Convert from samples to time

    print(A.shape)  # Print shape to veryfi size
    #rospy.loginfo("Printed Data Array")

    # -- Create subplots and save figure
    plt.subplot(3, 1, 1)
    plt.plot(T,X, '-')
    plt.title('ArUco Tracker')
    plt.ylabel('X Position [m]')

    plt.subplot(3, 1, 2)
    plt.plot(T,Y, '-')
```

```
        plt.ylabel('Y Position [m]')

        plt.subplot(3, 1, 3)
        plt.plot(T,TH, '-')
        plt.xlabel('time [s]')
        plt.ylabel('Angle [rad]')
        #plt.savefig('/home/jksllk/Desktop/Results/matplotlib/
           aruco_covar.png')
        plt.show()
        rospy.loginfo("Ploted and Saved Graph")

        # Remove the time column (T) from A
        A = np.delete(A, 3, 1)

        # Deviation matrix
        AA = np.dot(np.ones([len(A), len(A)]), A)
        AN = np.dot(AA, 1/(len(A)))
        a = A - AN
        # Covariance matrix
        rospy.loginfo("Calculate Covaraince Matrix")
        at = a.transpose()
        cov = np.dot(at, a)
        print(cov)



# CHECK IF NAME == MAIN
if __name__ == '__main__':
    covar_generator()
```

B.7 ArUco Tracker Odometry Generator

```
#!/usr/bin/env python
# Author: jksllk
# Version: 27.03.19
#
#       Script creating a odometry message from the topics generated
#       by the ArUco tracker and depth camera
#      (Relative, Starts in zero regardless of where it is)
#
# ----------------------------------------------------------------------
import rospy
import math
import numpy as np
import matplotlib.pyplot as plt
import tf
from std_msgs.msg import Int32, Float32
from geometry_msgs.msg import Point, Pose, Quaternion, Vector3, Twist
from nav_msgs.msg import Odometry
# Define global values
global H, h, cov_x, cov_x, rcov_z
H       = 0.7     # [m] Height camera is palced, constant
h       = 0.2     # [m] Heigth of car, constant
# Covariance tuning
cov_x   = 0.00126893
cov_y   = 0.0005075
rcov_z  = 0.44778692

# Initialize variables
x        = 0
y        = 0
theta    = 0
tx       = 0
ty       = 0
xt       = 0
yt       = 0
theta_t  = 0

```

```
# Define values of variables from subscribers
def pixel_x(msg):
    global px
    px = msg.data
    #print(px)

def pixel_y(msg):
    global py
    py = msg.data
    #print(py)

def trans_x(msg):
    global tx
    tx = msg.data
    #print(tx)

def trans_y(msg):
    global ty
    ty = msg.data
    #print(ty)

def trans_z(msg):
    global tz
    tz = msg.data
    #print(tz)

def depth_z(msg):
    global depth_z
    depth_z = (msg.data)/1000 # Converte to meters
    #print(depth_z)

def rot_x(msg):
    global rx
    rx = msg.data
    #print(rx)

def twist_callback(msg):
    global aruco_twist
    aruco_twist = msg

# Generate 2D pose message
def odom_generator():
```

```
    rospy.init_node("odom_generation_node", anonymous=True)
    # Define subscribers to recive translation coordinate from topics
    # Get xyz coordinates from ArUco Tracker
    pixel_x_sub=rospy.Subscriber("/aruco/pixel/x", Float32, pixel_x)
    pixel_y_sub=rospy.Subscriber("/aruco/pixel/y", Float32, pixel_y)
    trans_x_sub=rospy.Subscriber("/aruco/trans/x", Float32, trans_x)
    trans_y_sub=rospy.Subscriber("/aruco/trans/y", Float32, trans_y)
    trans_z_sub=rospy.Subscriber("/aruco/trans/z", Float32, trans_z)
    # Subscribe to depth camera topic
    depth_z_sub=rospy.Subscriber("/aruco/depth/avg_depth", Float32, depth_z)
    # Only rotation which is nesessary is rotation about the x axis
    rot_x_sub=rospy.Subscriber("/aruco/rot/x", Float32, rot_x)
    # /cmd_vel twist
    twist_sub=rospy.Subscriber("/cmd_vel", Twist, twist_callback)
    # Create publisher to publish coordinate message
    odom_msg_pub=rospy.Publisher("/cam1/aruco/odom", Odometry, queue_size=10)
    # Sleep for 1 seconds to allow data to be recived
    rospy.sleep(1)
    # Define rate of script

    rate=rospy.Rate(10) # 10 Hz
    # Coordinate message, position and orientation
    # Initialize variables
    xtt       = tz
    ytt       = -tx
    theta_tt  = rx
    x         = 0
    y         = 0
    theta     = 0
    xdot      = 0
    ydot      = 0
```

```
        theta_dot = 0
        rospy.loginfo("Node Initialized")
        rospy.loginfo(">> Generating Odometry Message")
        try:
            while not rospy.is_shutdown():
                current_time = rospy.Time.now()
                # Using Pytagoras to calculate x coordinate
                #y = np.sqrt(np.square(depth_z)-np.square(H-h))
                # Set current values
                xt = tz
                yt = -tx
                theta_t = rx
                #print(xt, yt, theta_t)
                # Find change in position, by subtracting current
                    from previous
                xdot = xt - xtt
                ydot = yt - ytt
                theta_dot = theta_t - theta_tt
                #print(xdot, ydot, theta_dot)
                # Update previous value
                xtt = xt
                ytt = yt
                theta_tt = theta_t
                #print(xdot, ydot, theta_dot)
                # Update position
                x += xdot
                y += ydot
                #print(x, y)
                # Angle is taken from the rotation of the ArUco
                theta += theta_dot
                # Publish the pose message
                odom = Odometry()
                aruco_twist = Twist()
                #aruco_twist =  # ?
                odom.header.stamp = current_time
                odom.header.frame_id = "odom"
                odom.child_frame_id = "base_aruco_link"
                odom.pose.pose.position.x = x
                odom.pose.pose.position.y = y
                odom.pose.pose.position.z = 0
                odom_quat=tf.transformations.
                   quaternion_from_euler(0, 0, theta)
```

```
                odom.pose.pose.orientation = Quaternion(*
                   odom_quat)
                odom.pose.covariance = [cov_x,  0,      0, 0, 0,
                   0,
                                        0,      cov_y,  0, 0, 0,
                                           0,
                                        0,      0,      0, 0, 0,
                                           0,
                                        0,      0,      0, 0, 0,
                                           0,
                                        0,      0,      0, 0, 0,
                                           0,
                                        0,      0,      0, 0, 0,
                                           rcov_z]

                odom.twist.twist = aruco_twist
                odom_msg_pub.publish(odom)
                #print(odom)
                rate.sleep()

        except KeyboardInterrupt:
            print("Shutting down")


if __name__ == '__main__':
    odom_generator()
```

B.8 Follow Waypoints Continuously

```python
#!/usr/bin/env python
#
# Edit:
#       Make path run contuniously
#
# -----------------------------------------------------------------------

import threading
import rospy
import actionlib

from smach import State,StateMachine
from move_base_msgs.msg import MoveBaseAction, MoveBaseGoal
from geometry_msgs.msg import PoseWithCovarianceStamped, PoseArray
from std_msgs.msg import Empty

waypoints = []
n = 0

class FollowPath(State):
    def __init__(self):
        State.__init__(self, outcomes=['success'], input_keys=['waypoints'])
        self.frame_id = rospy.get_param('~goal_frame_id','map')
        # Get a move_base action client
        self.client = actionlib.SimpleActionClient('move_base', MoveBaseAction)
        rospy.loginfo('Connecting to move_base...')
        self.client.wait_for_server()
        rospy.loginfo('Connected to move_base.')

    def execute(self, userdata):
        global waypoints, n
        # Execute waypoints each in sequence
        while n <= len(waypoints):
            for waypoint in waypoints:
                # Break if preempted
                if waypoints == []:
```

```
                    rospy.loginfo('The waypoint queue has been reset.')
                    break
                # Otherwise publish next waypoint as goal
                goal = MoveBaseGoal()
                goal.target_pose.header.frame_id = self.frame_id
                goal.target_pose.pose.position = waypoint.pose.pose.position
                goal.target_pose.pose.orientation = waypoint.pose.pose.orientation
                rospy.loginfo('Executing move_base goal to position (x,y): %s, %s' %
                        (waypoint.pose.pose.position.x, waypoint.pose.pose.position.y))
                rospy.loginfo("To cancel the goal: 'rostopic pub -1 /move_base/cancel actionlib_msgs/GoalID -- {}'")
                self.client.send_goal(goal)
                self.client.wait_for_result()

                n=n+1

                if n>len(waypoints):
                    n = 0

                print(n)



        return 'success'

def convert_PoseWithCovArray_to_PoseArray(waypoints):
    """Used to publish waypoints as pose array so that you can see them in rviz, etc."""
    poses = PoseArray()
    poses.header.frame_id = 'map'
    poses.poses = [pose.pose.pose for pose in waypoints]
    return poses

class GetPath(State):
    def __init__(self):
        State.__init__(self, outcomes=['success'], input_keys=['
```

```
waypoints'], output_keys=['waypoints'])
        # Create publsher to publish waypoints as pose array so that you can see them in rviz, etc.
        self.poseArray_publisher = rospy.Publisher('/waypoints', PoseArray, queue_size=1)

        # Start thread to listen for reset messages to clear the waypoint queue
        def wait_for_path_reset():
            """thread worker function"""
            global waypoints
            while not rospy.is_shutdown():
                data = rospy.wait_for_message('/path_reset', Empty)
                rospy.loginfo('Recieved path RESET message')
                self.initialize_path_queue()
                rospy.sleep(3) # Wait 3 seconds because `rostopic echo` latches
                               # for three seconds and wait_for_message() in a
                               # loop will see it again.
        reset_thread = threading.Thread(target=wait_for_path_reset)
        reset_thread.start()

    def initialize_path_queue(self):
        global waypoints
        waypoints = [] # the waypoint queue
        # publish empty waypoint queue as pose array so that you can see them the change in rviz, etc.
        self.poseArray_publisher.publish(convert_PoseWithCovArray_to_PoseArray(waypoints))

    def execute(self, userdata):
        global waypoints
        self.initialize_path_queue()
        self.path_ready = False

        # Start thread to listen for when the path is ready (this function will end then)
        def wait_for_path_ready():
            """thread worker function"""
```

```
            data = rospy.wait_for_message('/path_ready', Empty)
            rospy.loginfo('Recieved path READY message')
            self.path_ready = True
        ready_thread = threading.Thread(target=wait_for_path_ready)
        ready_thread.start()

        topic = "/initialpose"
        rospy.loginfo("Waiting to recieve waypoints via Pose msg on topic %s" % topic)
        rospy.loginfo("To start following waypoints: 'rostopic pub /path_ready std_msgs/Empty -1'")

        # Wait for published waypoints
        while not self.path_ready:
            try:
                pose = rospy.wait_for_message(topic, PoseWithCovarianceStamped, timeout=1)
            except rospy.ROSException as e:
                if 'timeout exceeded' in e.message:
                    continue  # no new waypoint within timeout, looping...
                else:
                    raise e
            rospy.loginfo("Recieved new waypoint")
            waypoints.append(pose)
            # publish waypoint queue as pose array so that you can see them in rviz, etc.
            self.poseArray_publisher.publish(convert_PoseWithCovArray_to_PoseArray(waypoints))

        # Path is ready! return success and move on to the next state (FOLLOW_PATH)
        return 'success'

class PathComplete(State):
    def __init__(self):
        State.__init__(self, outcomes=['success'])

    def execute(self, userdata):
        rospy.loginfo('###############################')
        rospy.loginfo('##### REACHED FINISH GATE #####')
```

```
        rospy.loginfo('###############################')
        return 'success'

def main():
    rospy.init_node('follow_waypoints')

    sm = StateMachine(outcomes=['success'])

    with sm:
        StateMachine.add('GET_PATH', GetPath(),
                                   transitions={'success':'FOLLOW_PATH'},
                                   remapping={'waypoints':'waypoints'})
        StateMachine.add('FOLLOW_PATH', FollowPath(),
                                   transitions={'success':'PATH_COMPLETE'
                                      },
                                   remapping={'waypoints':'waypoints'})
        StateMachine.add('PATH_COMPLETE', PathComplete(),
                                   transitions={'success':'GET_PATH'})

    outcome = sm.execute()
```

C. Launch-Files

This Appendix include all the launch-files utilized in this thesis.

C.1 Static Transform Configurations

```xml
<!-- -*- mode: XML -*- -->
<!-- Center of Rotation/base_link is Set to the IMU Location -->
<launch>

    <!-- Odometry to base_link -->
    <!-- node pkg="tf" type="static_transform_publisher" name="odom_to_basefootprint"
        args="0.0 0.0 0.0 0 0 0.0 /odom /base_link 40" /-->

    <!-- base_link to base_footprint -->
    <node pkg="tf" type="static_transform_publisher" name="base_footprint_to_base_link"
        args="0.0 0.0 0.0 3.14 0 0  /base_link /base_footprint 40" />


    <!-- IMU is rotated 180 degrees-->
    <node pkg="tf" type="static_transform_publisher" name="base_link_to_imu"
        args="0 0 0 0 0 0 /base_link /base_imu_link 50"/>

    <!-- ArUco-->
    <node pkg="tf" type="static_transform_publisher" name="base_link_to_aruco"
        args="0.17 0 0.11 0 0 3.14 /base_link /base_aruco_link 50"/>

    <!-- LiDAR -->
    <node pkg="tf" type="static_transform_publisher" name="base_link_to_laser"
    args="0.3 0 0.03 3.14 0 3.14 /base_link /laser 100"/>


</launch>
```

C.2 AMCL

```
<launch>

 <!-- AMCL -->
 <node pkg="amcl" type="amcl" name="amcl" output="screen">
    <!--param name ="/use_sim_time" value="true"/-->
    <remap from="scan" to="/scan"/>
    <param name="odom_frame_id" value="odom"/>
    <param name="base_frame_id" value="base_link" />
    <param name="global_frame_id" value="map" />
    <param name="odom_model_type" value="diff-corrected"/>
    <param name="odom_alpha5" value="0.1"/>
    <param name="initial_pose_x" value="0.0"/>
    <param name="initial_pose_y" value="0.0"/>
    <param name="initial_pose_a" value="0.0"/>
    <param name="transform_tolerance" value="0.2" />
    <param name="gui_publish_rate" value="10.0"/>
    <param name="laser_max_beams" value="60"/>
    <param name="min_particles" value="500"/>
    <param name="max_particles" value="5000"/>
    <param name="kld_err" value="0.05"/>
    <param name="kld_z" value="0.99"/>
    <param name="odom_alpha1" value="0.2"/>
    <param name="odom_alpha2" value="0.4"/>
    <param name="odom_alpha3" value="0.6"/>
    <param name="odom_alpha4" value="0.4"/>
    <param name="laser_min_range" value="0.15"/>
    <param name="laser_max_range" value="16.0"/>
    <param name="laser_z_hit" value="0.5"/>
    <param name="laser_z_short" value="0.05"/>
    <param name="laser_z_max" value="0.05"/>
    <param name="laser_z_rand" value="0.5"/>
    <param name="laser_sigma_hit" value="0.2"/>
    <param name="laser_lambda_short" value="0.1"/>
    <param name="laser_lambda_short" value="0.1"/>
    <param name="laser_model_type" value="likelihood_field"/>
    <param name="laser_likelihood_max_dist" value="2.0"/>
    <param name="update_min_d" value="0.2"/>
    <param name="update_min_a" value="0.5"/>
    <param name="odom_frame_id" value="odom"/>
    <param name="resample_interval" value="1"/>
```

```
    <param name="transform_tolerance" value="0.2"/>
    <param name="recovery_alpha_slow" value="0.0"/>
    <param name="recovery_alpha_fast" value="0.0"/>
  </node>

</launch>
```

C.3 gmapping

```
<launch>
  <arg name="scan_topic"  default="scan" />
  <arg name="base_frame"  default="base_link"/>
  <arg name="odom_frame"  default="odom"/>

  <node pkg="gmapping" type="slam_gmapping" name="slam_gmapping"
     output="screen">
    <param name="base_frame" value="$(arg base_frame)"/>
    <param name="odom_frame" value="$(arg odom_frame)"/>
    <param name="map_update_interval" value="0.01"/>
    <param name="maxUrange" value="4.0"/>
    <param name="maxRange" value="5.0"/>
    <param name="sigma" value="0.05"/>
    <param name="kernelSize" value="3"/>
    <param name="lstep" value="0.05"/>
    <param name="astep" value="0.05"/>
    <param name="iterations" value="5"/>
    <param name="lsigma" value="0.075"/>
    <param name="ogain" value="3.0"/>
    <param name="lskip" value="0"/>
    <param name="minimumScore" value="30"/>
    <param name="srr" value="0.01"/>
    <param name="srt" value="0.02"/>
    <param name="str" value="0.01"/>
    <param name="stt" value="0.02"/>
    <param name="linearUpdate" value="0.05"/>
    <param name="angularUpdate" value="0.0436"/>
    <param name="temporalUpdate" value="-1.0"/>
    <param name="resampleThreshold" value="0.5"/>
    <param name="particles" value="8"/>
  <!--
    <param name="xmin" value="-50.0"/>
    <param name="ymin" value="-50.0"/>
    <param name="xmax" value="50.0"/>
    <param name="ymax" value="50.0"/>
  make the starting size small for the benefit of the Android
     client's memory...
  -->
    <param name="xmin" value="-1.0"/>
    <param name="ymin" value="-1.0"/>
```

```
    <param name="xmax" value="1.0"/>
    <param name="ymax" value="1.0"/>

    <param name="delta" value="0.05"/>
    <param name="llsamplerange" value="0.01"/>
    <param name="llsamplestep" value="0.01"/>
    <param name="lasamplerange" value="0.005"/>
    <param name="lasamplestep" value="0.005"/>
    <remap from="scan" to="$(arg scan_topic)"/>
  </node>
</launch>
```

C.4 hector_slam

```
<launch>

    <!-- Launch Default RPLiDAR A3 Node -->
    <!--include file="$(find rplidar_ros)/launch/rplidar_a3.launch"/-->

    <!--node pkg="tf" type="static_transform_publisher" name="map_to_odom"
        args="0.0 0.0 0.0 0 0 0.0 /map /odom 40" /-->

    <!--node pkg="tf" type="static_transform_publisher" name="odom_to_basefootprint"
        args="0.0 0.0 0.0 0 0 0.0 /odom /base_footprint 40" />

    <node pkg="tf" type="static_transform_publisher" name="base_footprint_to_base_link"
        args="0.0 0.0 0.0 0 0 0 /base_footprint /base_link 40" /-->

    <!-- Publish Static Transform for the LiDAR -->
    <!--node pkg="tf" type="static_transform_publisher" name="base_link_to_laser"
    args="0 0 0 3.14 0 0 /base_link /laser 40"/-->

    <include file="$(find hector_mapping)/launch/mapping_default.launch" />

    <node pkg="rviz" type="rviz" name="rviz" args="-d $(find rr_racer)/rviz/hector_mapping.rviz" />

    <include file="$(find hector_geotiff)/launch/geotiff_mapper.launch" />

</launch>
```

C.5 move_base

```
<launch>
  <!-- Global Parameters -->
  <!--param name="/use_sim_time" value="true"/-->

  <!-- Map Server -->
  <node name="map_server" pkg="map_server" type="map_server" args
     ="$(find rr_racer)/maps/rauland_up.yaml"/>

  <!-- AMCL Global Planner -->
  <include file="$(find rr_racer)/launch/amcl_teb.launch"/>

  <!-- move_base -->
  <node pkg="move_base" type="move_base" respawn="false" name="
     move_base" output="screen">
    <rosparam file="$(find rr_racer)/param/navigation/
       costmap_common_params.yaml" command="load" ns="
       global_costmap" />
    <rosparam file="$(find rr_racer)/param/navigation/
       costmap_common_params.yaml" command="load" ns="
       local_costmap" />
    <rosparam file="$(find rr_racer)/param/navigation/
       local_costmap_params.yaml" command="load" />
    <rosparam file="$(find rr_racer)/param/navigation/
       global_costmap_params.yaml" command="load" />
    <rosparam file="$(find rr_racer)/param/navigation/
       base_local_planner_teb2.yaml" command="load" />

    <param name="base_local_planner" value="teb_local_planner/
       TebLocalPlannerROS" />
    <param name="controller_frequency" value="10.0" />
  </node>

</launch>
```

D. Parameters

This Appendix contains the `.yaml`-files which launch-files access to get different parameters.

D.1 ekf_localization

```
# The frequency, in Hz, at which the filter will output a position estimate. Note that the filter will not begin
# computation until it receives at least one message from one of the inputs. It will then run continuously at the
# frequency specified here, regardless of whether it receives more measurements. Defaults to 30 if unspecified.
frequency: 10

# The period, in seconds, after which we consider a sensor to have timed out. In this event, we carry out a predict
# cycle on the EKF without correcting it. This parameter can be thought of as the minimum frequency with which the
# filter will generate new output. Defaults to 1 / frequency if not specified.
sensor_timeout: 0.1

# ekf_localization_node and ukf_localization_node both use a 3D omnidirectional motion model. If this parameter is
# set to true, no 3D information will be used in your state estimate. Use this if you are operating in a planar
# environment and want to ignore the effect of small variations in the ground plane that might otherwise be detected
# by, for example, an IMU. Defaults to false if unspecified.
two_d_mode: true

# Use this parameter to provide an offset to the transform generated by ekf_localization_node. This can be used for
# future dating the transform, which is required for interaction with some other packages. Defaults to 0.0 if
# unspecified.
transform_time_offset: 0.0

```

```
# Use this parameter to provide specify how long the tf listener should wait for a transform to become available.
# Defaults to 0.0 if unspecified.
transform_timeout: 0.0

# If you're having trouble, try setting this to true, and then echo the /diagnostics_agg topic to see if the node is
# unhappy with any settings or data.
print_diagnostics: true

# Debug settings. Not for the faint of heart. Outputs a ludicrous amount of information to the file specified by
# debug_out_file. I hope you like matrices! Please note that setting this to true will have strongly deleterious
# effects on the performance of the node. Defaults to false if unspecified.
debug: false

# Defaults to "robot_localization_debug.txt" if unspecified. Please specify the full path.
#debug_out_file: /home/nvidia/RR_RACER_ws/file.txt

# Whether to broadcast the transformation over the /tf topic. Defaults to true if unspecified.
publish_tf: true

# Whether to publish the acceleration state. Defaults to false if unspecified.
publish_acceleration: false

# REP-105 (http://www.ros.org/reps/rep-0105.html) specifies four principal coordinate frames: base_link, odom, map, and
# earth. base_link is the coordinate frame that is affixed to the robot. Both odom and map are world-fixed frames.
# The robot's position in the odom frame will drift over time, but is accurate in the short term and should be
# continuous. The odom frame is therefore the best frame for executing local motion plans. The map frame, like the odom
# frame, is a world-fixed coordinate frame, and while it contains the most globally accurate position estimate for your
# robot, it is subject to discrete jumps, e.g., due to the fusion of GPS data or a correction from a map-based
```

```
# localization node. The earth frame is used to relate multiple map frames by giving them a common reference frame.
# ekf_localization_node and ukf_localization_node are not concerned with the earth frame.
# Here is how to use the following settings:
# 1. Set the map_frame, odom_frame, and base_link frames to the appropriate frame names for your system.
#     1a. If your system does not have a map_frame, just remove it, and make sure "world_frame" is set to the value of
#         odom_frame.
# 2. If you are fusing continuous position data such as wheel encoder odometry, visual odometry, or IMU data, set
#   "world_frame" to your odom_frame value. This is the default behavior for robot_localization's state estimation nodes.
# 3. If you are fusing global absolute position data that is subject to discrete jumps (e.g., GPS or position updates
# from landmark observations) then:
#     3a. Set your "world_frame" to your map_frame value
#     3b. MAKE SURE something else is generating the odom->base_link transform. Note that this can even be another state
#         estimation node from robot_localization! However, that instance should *not* fuse the global data.
map_frame: map              # Defaults to "map" if unspecified
odom_frame: odom            # Defaults to "odom" if unspecified
base_link_frame: base_link  # Defaults to "base_link" if unspecified
world_frame: odom           # Defaults to the value of odom_frame if unspecified

# The filter accepts an arbitrary number of inputs from each input message type (nav_msgs/Odometry,
# geometry_msgs/PoseWithCovarianceStamped, geometry_msgs/TwistWithCovarianceStamped,
# sensor_msgs/Imu). To add an input, simply append the next number in the sequence to its "base" name, e.g., odom0,
# odom1, twist0, twist1, imu0, imu1, imu2, etc. The value should be the topic name. These parameters obviously have no
# default values, and must be specified.


# Pose from camera 1
odom0: /cam1/aruco/odom
```

```
# Camera gives x,y,yaw
odom0_config: [true,  true,  false,
               false, false, true,
               false, false, false,
               false, false, false,
               false, false, false]
odom0_differential: true
odom0_relative: true
odom0_queue_size: 5
odom0_rejection_threshold: 2  # Note the difference in parameter name
odom0_nodelay: false




odom1: /odom_ackermann

# Each sensor reading updates some or all of the filter's state. These options give you greater control over which
# values from each measurement are fed to the filter. For example, if you have an odometry message as input, but only
# want to use its Z position value, then set the entire vector to false, except for the third entry. The order of the
# values is: x,     y,      z,
#            roll,  pitch,  yaw,
#            vx,    vy,     vz,
#            vroll, vpitch, vyaw,
#            ax,    ay,     az.
#
#Note that not some message types do not provide some of the state variables estimated by the filter. For example, a TwistWithCovarianceStamped message
# has no pose information, so the first six values would be meaningless in that case. Each vector defaults to all false
# if unspecified, effectively making this parameter required for each sensor.
odom1_config: [true,  true,  false,
               false, false, true,  # Steering angle is not accurate enough
               false, false, false,
               false, false, false,
```

```
                false, false, false]

# If you have high-frequency data or are running with a low
    frequency parameter value, then you may want to increase
# the size of the subscription queue so that more measurements
    are fused.
odom1_queue_size: 5

# [ADVANCED] Large messages in ROS can exhibit strange behavior
    when they arrive at a high frequency. This is a result
# of Nagle's algorithm. This option tells the ROS subscriber to
    use the tcpNoDelay option, which disables Nagle's
# algorithm.
odom1_nodelay: false

# [ADVANCED] When measuring one pose variable with two sensors, a
     situation can arise in which both sensors under-
# report their covariances. This can lead to the filter rapidly
    jumping back and forth between each measurement as they
# arrive. In these cases, it often makes sense to (a) correct the
     measurement covariances, or (b) if velocity is also
# measured by one of the sensors, let one sensor measure pose,
    and the other velocity. However, doing (a) or (b) isn't
# always feasible, and so we expose the differential parameter.
    When differential mode is enabled, all absolute pose
# data is converted to velocity data by differentiating the
    absolute pose measurements. These velocities are then
# integrated as usual. NOTE: this only applies to sensors that
    provide pose measurements; setting differential to true
# for twist measurements has no effect.
odom1_differential: false

# [ADVANCED] When the node starts, if this parameter is true,
    then the first measurement is treated as a "zero point"
# for all future measurements. While you can achieve the same
    effect with the differential paremeter, the key
# difference is that the relative parameter doesn't cause the
    measurement to be converted to a velocity before
# integrating it. If you simply want your measurements to start
    at 0 for a given sensor, set this to true.
odom1_relative: false

```

```
# [ADVANCED] If your data is subject to outliers, use these threshold settings, expressed as Mahalanobis distances, to
# control how far away from the current vehicle state a sensor measurement is permitted to be. Each defaults to
# numeric_limits<double>::max() if unspecified. It is strongly recommended that these parameters be removed if not
# required. Data is specified at the level of pose and twist variables, rather than for each variable in isolation.
# For messages that have both pose and twist data, the parameter specifies to which part of the message we are applying
# the thresholds.
odom1_pose_rejection_threshold: 5
odom1_twist_rejection_threshold: 1


# Pose from Camera 2
#odom2: /cam2/aruco/odom
#odom2_config: [true,  true,  false,
#               false, false, true,
#               false, false, false,
#               false, false, false,
#               false, false, false]
#odom2_differential: false
#odom2_relative: false
#odom2_queue_size: 5
#odom2_rejection_threshold: 2  # Note the difference in parameter name
#odom2_nodelay: false


# values is: x,     y,      z,
#            roll,  pitch,  yaw,
#            vx,    vy,     vz,
#            vroll, vpitch, vyaw,
#            ax,    ay,     az.
#
imu0: /imu
imu0_config: [false, false, false,
              false, false, false,
              false, false, false,
              false, false, true,
              true,  true,  true]  # false, false, false]
```

```
imu0_nodelay: false
imu0_differential: true
imu0_relative: true
imu0_queue_size: 10
imu0_pose_rejection_threshold: 0.8      # Note the difference in parameter names
imu0_twist_rejection_threshold: 0.8             #
imu0_linear_acceleration_rejection_threshold: 0.8  #

# [ADVANCED] Some IMUs automatically remove acceleration due to gravity, and others don't. If yours doesn't, please set
# this to true, and *make sure* your data conforms to REP-103, specifically, that the data is in ENU frame.
imu0_remove_gravitational_acceleration: true

# [ADVANCED]  The EKF and UKF models follow a standard predict/correct cycle. During prediction, if there is no
# acceleration reference, the velocity at time t+1 is simply predicted to be the same as the velocity at time t. During
# correction, this predicted value is fused with the measured value to produce the new velocity estimate. This can be
# problematic, as the final velocity will effectively be a weighted average of the old velocity and the new one. When
# this velocity is the integrated into a new pose, the result can be sluggish covergence. This effect is especially
# noticeable with LIDAR data during rotations. To get around it, users can try inflating the process_noise_covariance
# for the velocity variable in question, or decrease the variance of the variable in question in the measurement
# itself. In addition, users can also take advantage of the control command being issued to the robot at the time we
# make the prediction. If control is used, it will get converted into an acceleration term, which will be used during
# predicition. Note that if an acceleration measurement for the variable in question is available from one of the
# inputs, the control term will be ignored.
# Whether or not we use the control input during predicition. Defaults to false.
use_control: false
# Whether the input (assumed to be cmd_vel) is a geometry_msgs/Twist or geometry_msgs/TwistStamped message. Defaults to
# false.
```

```
stamped_control: false
# The last issued control command will be used in prediction for this period. Defaults to 0.2.
control_timeout: 0.2
# Which velocities are being controlled. Order is vx, vy, vz, vroll, vpitch, vyaw.
control_config: [true, false, false, false, false, true]
# Places limits on how large the acceleration term will be. Should match your robot's kinematics.
acceleration_limits: [1.3, 0.0, 0.0, 0.0, 0.0, 3.4]
# Acceleration and deceleration limits are not always the same for robots.
deceleration_limits: [1.3, 0.0, 0.0, 0.0, 0.0, 4.5]
# If your robot cannot instantaneously reach its acceleration limit, the permitted change can be controlled with these
# gains
acceleration_gains: [0.8, 0.0, 0.0, 0.0, 0.0, 0.9]
# If your robot cannot instantaneously reach its deceleration limit, the permitted change can be controlled with these
# gains
deceleration_gains: [1.0, 0.0, 0.0, 0.0, 0.0, 1.0]

# [ADVANCED] The process noise covariance matrix can be difficult to tune, and can vary for each application, so it is
# exposed as a configuration parameter. This matrix represents the noise we add to the total error after each
# prediction step. The better the omnidirectional motion model matches your system, the smaller these values can be.
# However, if users find that a given variable is slow to converge, one approach is to increase the
# process_noise_covariance diagonal value for the variable in question, which will cause the filter's predicted error
# to be larger, which will cause the filter to trust the incoming measurement more during correction. The values are
# ordered as x, y, z, roll, pitch, yaw, vx, vy, vz, vroll, vpitch, vyaw, ax, ay, az. Defaults to the matrix below if
# unspecified.
process_noise_covariance: [0.05, 0,    0,    0,    0,    0,    0,     0,     0,    0,    0,    0,    0,    0,    0,
                           0,    0.05, 0,    0,    0,    0,    0,     0,     0,    0,    0,    0,    0,    0,    0,
```

```
                                        0,    0,    0.06, 0,    0,    0,    0,
                                            0,    0,    0,    0,    0,
                                          0,    0,    0,
                                        0,    0,    0,    0.03, 0,    0,    0,
                                            0,    0,    0,    0,    0,
                                          0,    0,    0,
                                        0,    0,    0,    0,    0.03, 0,    0,
                                            0,    0,    0,    0,    0,
                                          0,    0,    0,
                                        0,    0,    0,    0,    0,    0.06, 0,
                                            0,    0,    0,    0,    0,
                                          0,    0,    0,
                                        0,    0,    0,    0,    0,    0,
                                           0.025, 0,    0,    0,    0,    0,
                                          0,    0,    0,
                                        0,    0,    0,    0,    0,    0,    0,
                                            0.025, 0,    0,    0,    0,
                                          0,    0,    0,
                                        0,    0,    0,    0,    0,    0,    0,
                                            0,    0.04, 0,    0,    0,
                                          0,    0,    0,
                                        0,    0,    0,    0,    0,    0,    0,
                                            0,    0,    0.01, 0,    0,
                                          0,    0,    0,
                                        0,    0,    0,    0,    0,    0,    0,
                                            0,    0,    0,    0.01, 0,
                                          0,    0,    0,
                                        0,    0,    0,    0,    0,    0,    0,
                                            0,    0,    0,    0,    0.02,
                                        0,    0,    0,
                                        0,    0,    0,    0,    0,    0,    0,
                                            0,    0,    0,    0,    0,
                                          0.01, 0,    0,
                                        0,    0,    0,    0,    0,    0,    0,
                                            0,    0,    0,    0,    0,
                                          0,    0.01, 0,
                                        0,    0,    0,    0,    0,    0,    0,
                                            0,    0,    0,    0,    0,
                                          0,    0,    0.015]

# [ADVANCED] This represents the initial value for the state
    estimate error covariance matrix. Setting a diagonal
```

```
# value (variance) to a large value will result in rapid convergence for initial measurements of the variable in
# question. Users should take care not to use large values for variables that will not be measured directly. The values
# are ordered as x, y, z, roll, pitch, yaw, vx, vy, vz, vroll, vpitch, vyaw, ax, ay, az. Defaults to the matrix below
#if unspecified.
initial_estimate_covariance: [1e-9, 0,    0,    0,    0,    0,    0,    0,    0,    0,     0,     0,     0,    0,    0,
                              0,    1e-9, 0,    0,    0,    0,    0,    0,    0,    0,     0,     0,     0,    0,    0,
                              0,    0,    1e-9, 0,    0,    0,    0,    0,    0,    0,     0,     0,     0,    0,    0,
                              0,    0,    0,    1e-9, 0,    0,    0,    0,    0,    0,     0,     0,     0,    0,    0,
                              0,    0,    0,    0,    1e-9, 0,    0,    0,    0,    0,     0,     0,     0,    0,    0,
                              0,    0,    0,    0,    0,    1e-9, 0,    0,    0,    0,     0,     0,     0,    0,    0,
                              0,    0,    0,    0,    0,    0,    1e-9, 0,    0,    0,     0,     0,     0,    0,    0,
                              0,    0,    0,    0,    0,    0,    0,    1e-9, 0,    0,     0,     0,     0,    0,    0,
                              0,    0,    0,    0,    0,    0,    0,    0,    1e-9, 0,     0,     0,     0,    0,    0,
                              0,    0,    0,    0,    0,    0,    0,    0,    0,    1e-9,  0,     0,     0,    0,    0,
                              0,    0,    0,    0,    0,    0,    0,    0,    0,    0,     1e-9,  0,     0,    0,    0,
                              0,    0,    0,    0,    0,    0,    0,    0,    0,    0,     0,     1e-9,  0,    0,    0,
```

```
                                   0,    0,    0,    0,    0,    0,
                                         0,    0,    0,    0,     0,
                                          0,     1e-9, 0,    0,
                                   0,    0,    0,    0,    0,    0,
                                         0,    0,    0,    0,     0,
                                          0,     0,    1e-9, 0,
                                   0,    0,    0,    0,    0,    0,
                                         0,    0,    0,    0,     0,
                                          0,     0,    0,    1e-9]
```

D.2 Common Costmap

```
# Costmap common
#
# Version: 19.03.19
# Changes made:
#
#   ...
#
map_type: costmap
origin_z: 0.0

obstacle_range: 2
raytrace_range: 2
#footprint: [[x0, y0], [x1, y1], ... [xn, yn]]
#robot_radius: ir_of_robot
#robot_radius: 0.5 # distance a circular robot should be clear of
    the obstacle
footprint:  [[-0.2, 0.1], [0.33, 0.1], [0.33, -0.1], [-0.2,
   -0.1]]
inflation_radius: 0.1

observation_sources: laser_scan_sensor #point_cloud_sensor

# marking - add obstacle information to cost map
# clearing - clear obstacle information to cost map
laser_scan_sensor: {sensor_frame: laser, data_type: LaserScan,
   topic: scan, marking: true, clearing: true}

#point_cloud_sensor: {sensor_frame: frame_name, data_type:
   PointCloud, topic: topic_name, marking: true, clearing: true
```

D.3 Local Costmap

```
#
# Version: 07.04.19
# Changes made:
#
#   Reduced resolution to help with process speed
#
# -----------------------------------------------
# Local Costmap Params
local_costmap:
  global_frame: odom
  robot_base_frame: base_link
  update_frequency: 5
  publish_frequency: 2
  static_map: false
  rolling_window: true
  width: 2.5
  height: 2.5
  resolution: 0.05
```

D.4 Global Costmap

```yaml
# Global Costmap Params
#
# Version: 29.03.19
# Chages made:
#
#   Selected true on static_map
#
global_costmap:
  global_frame: map
  robot_base_frame: base_link
  publish_frequency: 5
  static_map: true
```

D.5 teb_local_planer

```
# Version: 12.04.19
#
# Changes made:
#
#   Added viapoint capabilities, homotopy
#
TebLocalPlannerROS:

 # Mics
 odom_topic: odom
 map_frame: /map

 # Trajectory
 teb_autosize: True
 dt_ref: 0.3
 dt_hysteresis: 0.2
 global_plan_overwrite_orientation: True
 max_global_plan_lookahead_dist: 2.0
 feasibility_check_no_poses: 5
 goal_plan_viapoint: 1 # NEW

 # Robot
 max_vel_x: 0.2
 max_vel_x_backwards: 0.2
 max_vel_theta: 2
 acc_lim_x: 1
 acc_lim_theta: 2
 wheelbase: 0.33 # NEW TEST
 cmd_angle_instead_rotvel: True # NEW TEST
 min_turning_radius: 0.63
 footprint_model: # types: "point", "circular", "two_circles", "line", "polygon"
   type: "polygon"
   vertices: [[-0.2, 0.1], [0.33, 0.1], [0.33, -0.1], [-0.2, -0.1]]

 # GoalTolerance
 xy_goal_tolerance: 0.2
 yaw_goal_tolerance: 0.34 #20 deg
 free_goal_vel: False
```

```

# Obstacles
min_obstacle_dist: 0.1
include_costmap_obstacles: True
costmap_obstacles_behind_robot_dist: 1.0
obstacle_poses_affected: 30
costmap_converter_plugin: ""
costmap_converter_spin_thread: False
costmap_converter_rate: 5

# Optimization
no_inner_iterations: 5
no_outer_iterations: 4
optimization_activate: True
optimization_verbose: False
penalty_epsilon: 0.1
weight_max_vel_x: 2
weight_max_vel_theta: 1
weight_acc_lim_x: 1
weight_acc_lim_theta: 1
weight_kinematics_nh: 1000
weight_kinematics_forward_drive: 10 # 1, NEW TEST
weight_kinematics_turning_radius: 100
weight_optimaltime: 1
weight_obstacle: 50
weight_dynamic_obstacle: 100 # not in use yet
weigth_viapoint: 1000            # NEW
alternative_time_cost: False # not in use yet
allow_init_with_backward_motion: True # NEW TEST


# Homotopy Class Planner
enable_homotopy_class_planning: False #True
enable_multithreading: True
simple_exploration: False
max_number_classes: 4
roadmap_graph_no_samples: 15
roadmap_graph_area_width: 5
h_signature_prescaler: 0.5
h_signature_threshold: 0.1
obstacle_keypoint_offset: 0.1
obstacle_heading_threshold: 0.45
```

```
visualize_hc_graph: False
```

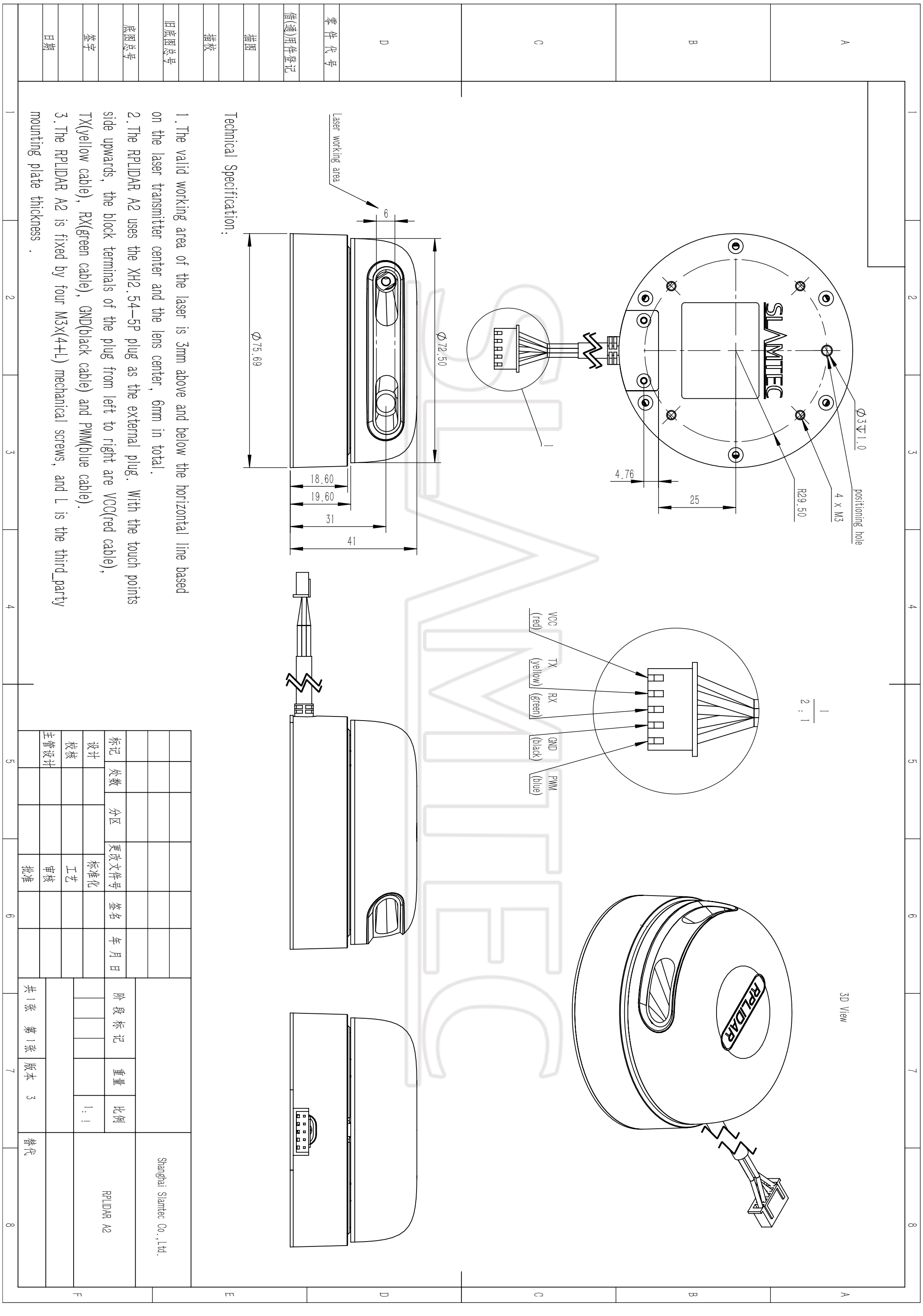
E. Data Sheets

This Appendix include all the data sheets for the components utilized in this thesis

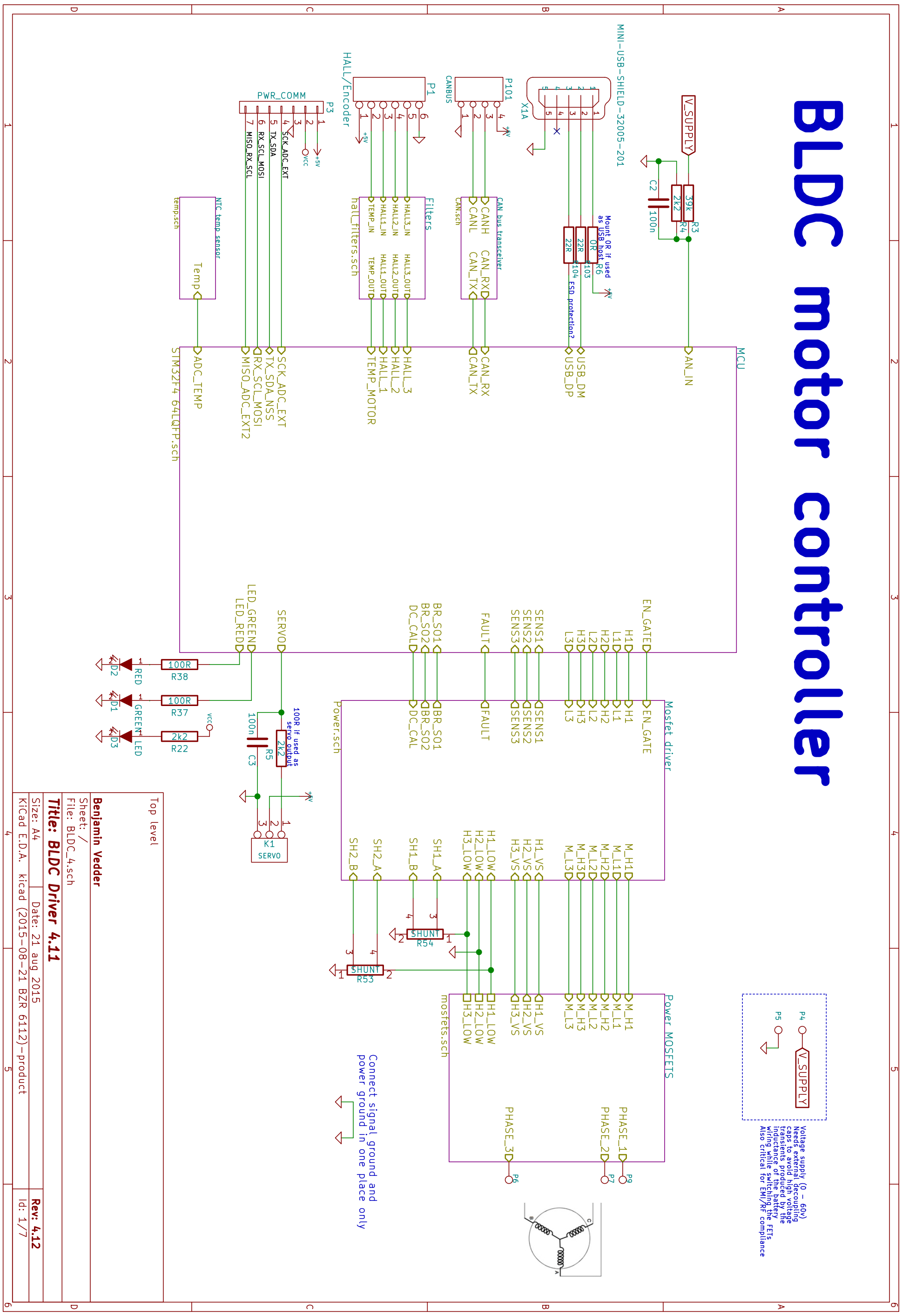
E.1 Intel® RealSense™ D435(i)

Use Environment	Indoor/Outdoor
Depth Technology	Active IR Stereo (Global Shutter)
Main Intel® RealSense™ component	Intel® RealSense™ Vision Processor D4 Intel® RealSense™ module D430
Depth Field of View (FOV)—(Horizontal × Vertical × Diagonal)	85.2° x 58° x 94° (+/- 3°)
Depth Stream Output Resolution	Up to 1280 x 720
Depth Stream Output Frame Rate	Up to 90 fps
Minimum Depth Distance (Min-Z)	0.1m
Sensor Shutter Type	Global shutter
Maximum Range	Approx.10 meters; Varies depending on calibration, scene, and lighting condition
RGB Sensor Resolution and Frame Rate	1920 x 1080 at 30 fps
RGB Sensor FOV (Horizontal x Vertical x Diagonal)	69.4° x 42.5° x 77° (+/- 3°)
Camera Dimension (Length x Depth x Height)	90 mm x 25 mm x 25 mm
Connectors	USB 3.0 Type - C
Mounting Mechanism	One 1/4-20 UNC thread mounting point Two M3 thread mounting points

E.2 SLAMTEC RPLiDAR A3



E.3 VESC


BLDC motor controller
MCU
Mosfet driver
Power MOSFETS
Benjamin Vedder
Title: BLDC Driver 4.11
Size: A4
Date: 21 aug 2015
Rev: 4.12
Connect signal ground and power ground in one place only

E.4 Servo

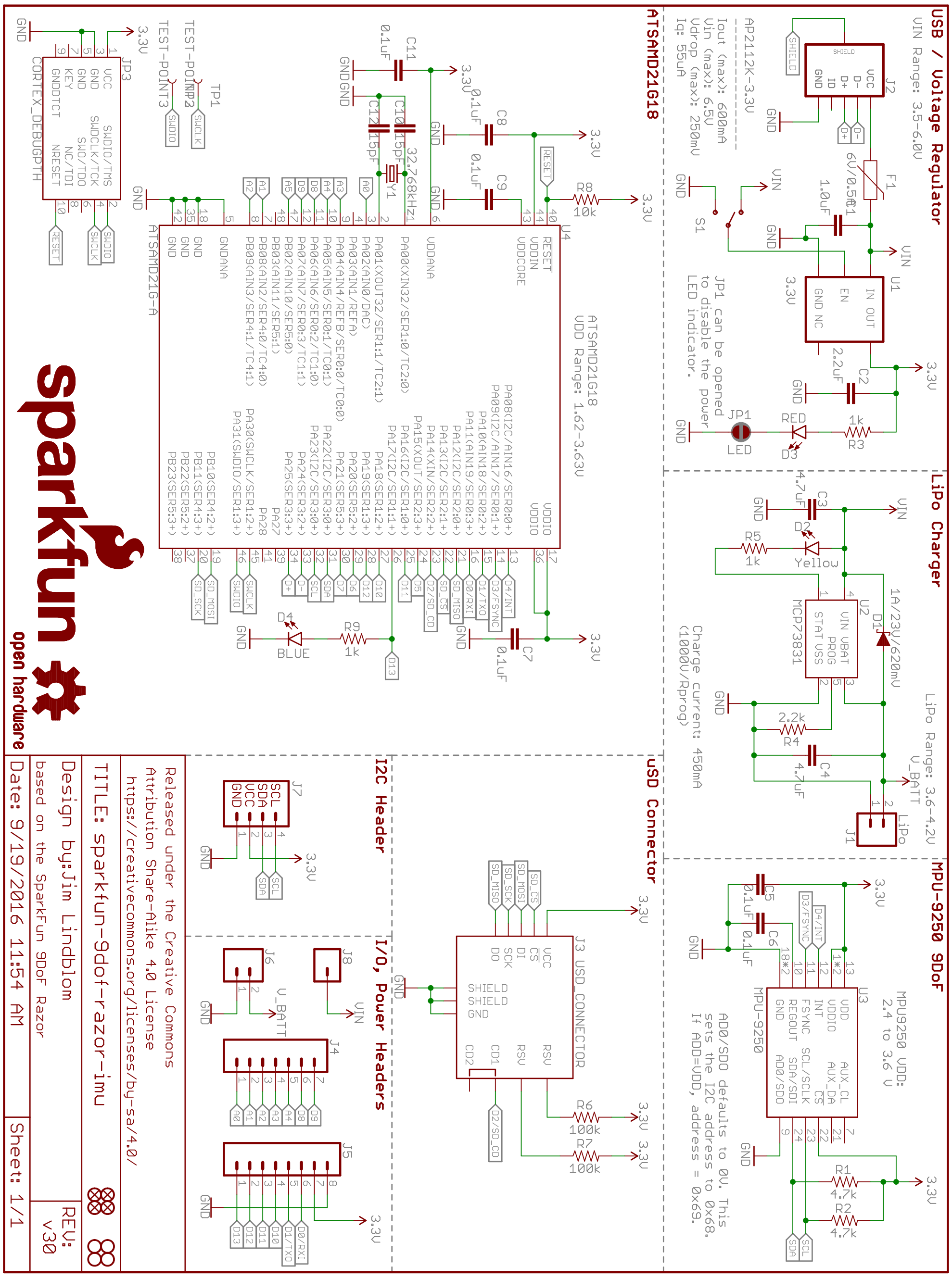
Power HD HD-9001MG - Standard Servo

Specifications

Modulation:	Analog
Torque:	**4.8V:** 119.40 oz-in (8.60 kg-cm) **6.0V:** 136.10 oz-in (9.80 kg-cm)
Speed:	**4.8V:** 0.16 sec/60° **6.0V:** 0.14 sec/60°
Weight:	1.98 oz (56.0 g)
Dimensions:	**Length:** 1.65 in (41.9 mm) **Width:** 0.81 in (20.6 mm) **Height:** 1.65 in (41.9 mm)
Motor Type:	(add)
Gear Type:	Metal
Rotation/Support:	Dual Bearings
Rotational Range:	(add)
Pulse Cycle:	(add)
Pulse Width:	(add)
Connector Type:	(add)

Brand:	Power HD
Product Number:	HD-9001MG
Typical Price:	12.00 USD
Compare:	add+

E.5 SparkFun 9DoF Razor IMU M0

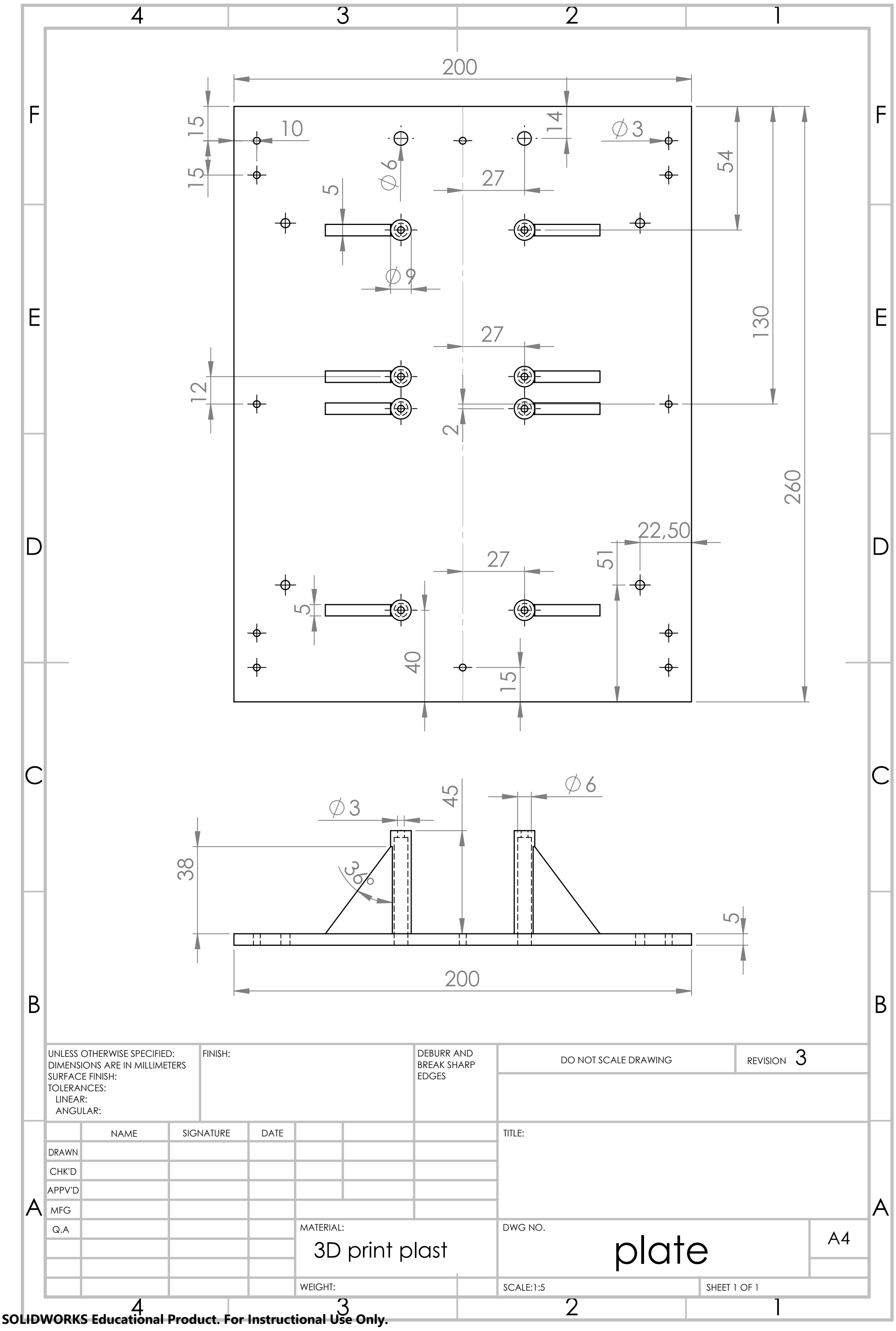


E.6 Tamiya Mercedes-Benz Actros 3363 6x4 GS - Kit

The construction manual for the kit can be accessed by following the link below:

`https://d1hu0eys0tj9xi.cloudfront.net/media/files/56348ml-779-3f46.pdf`

E.7 3D Printed Platform



F. Photos of Prototype

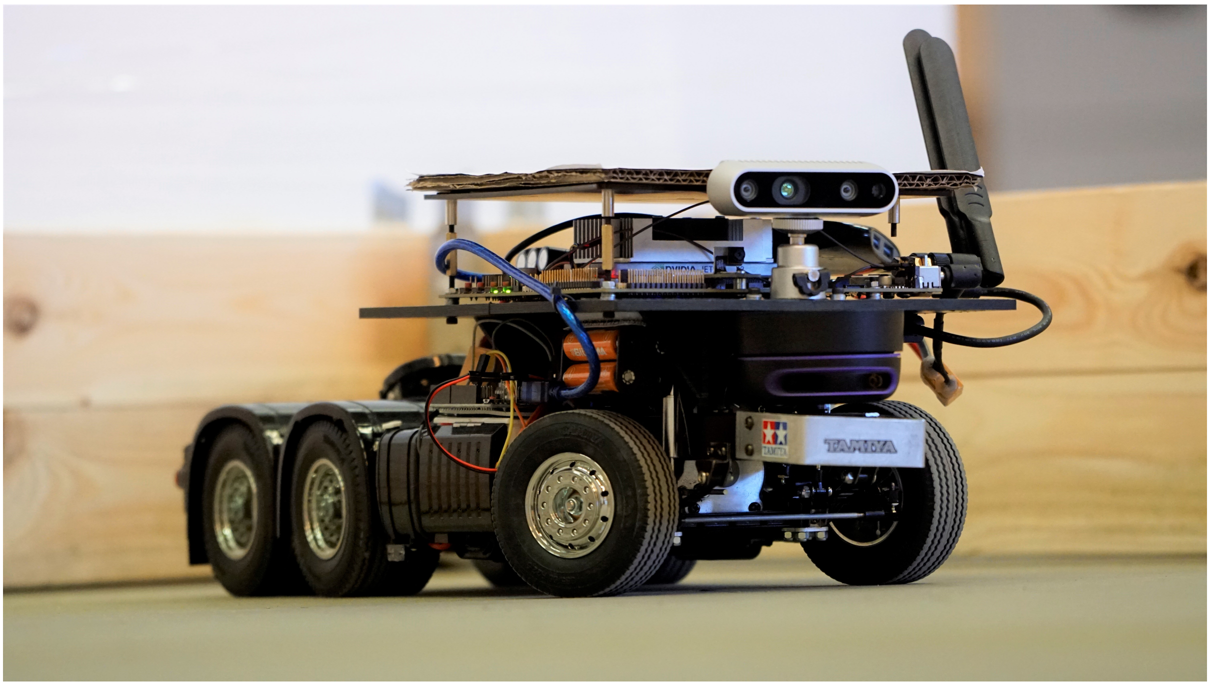


Figure F.1: Right Front View of Prototype

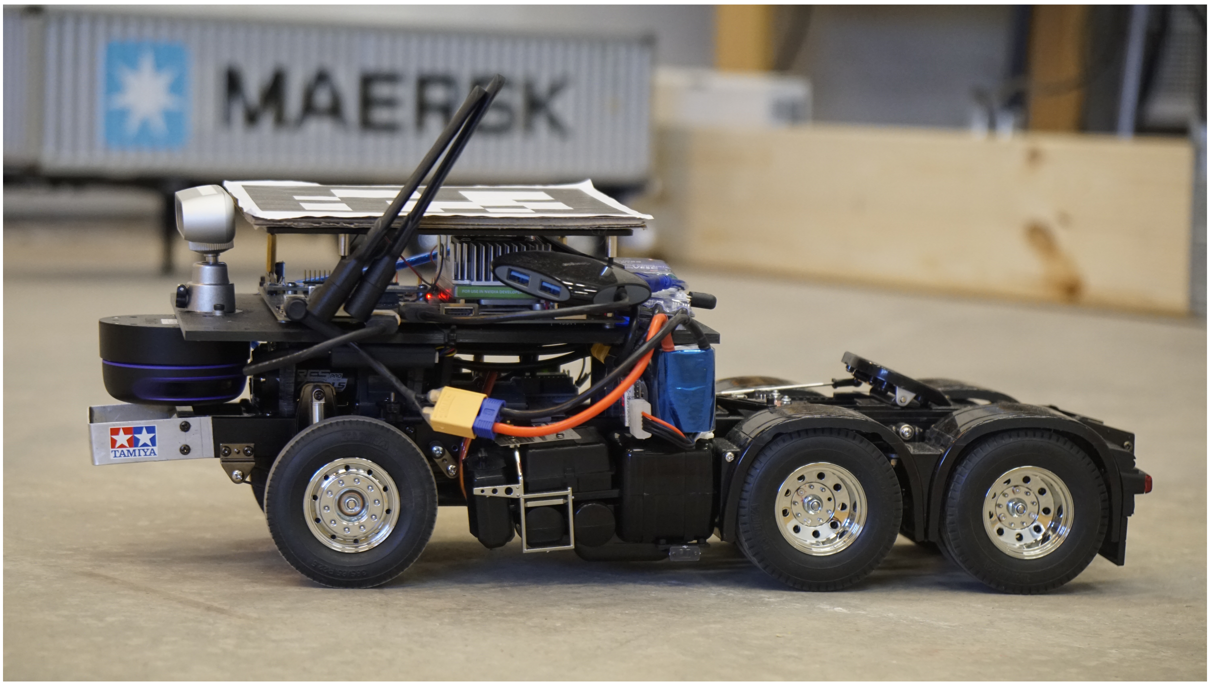


Figure F.2: Side View of Prototype

Figure F.3: Prototype with Trailer

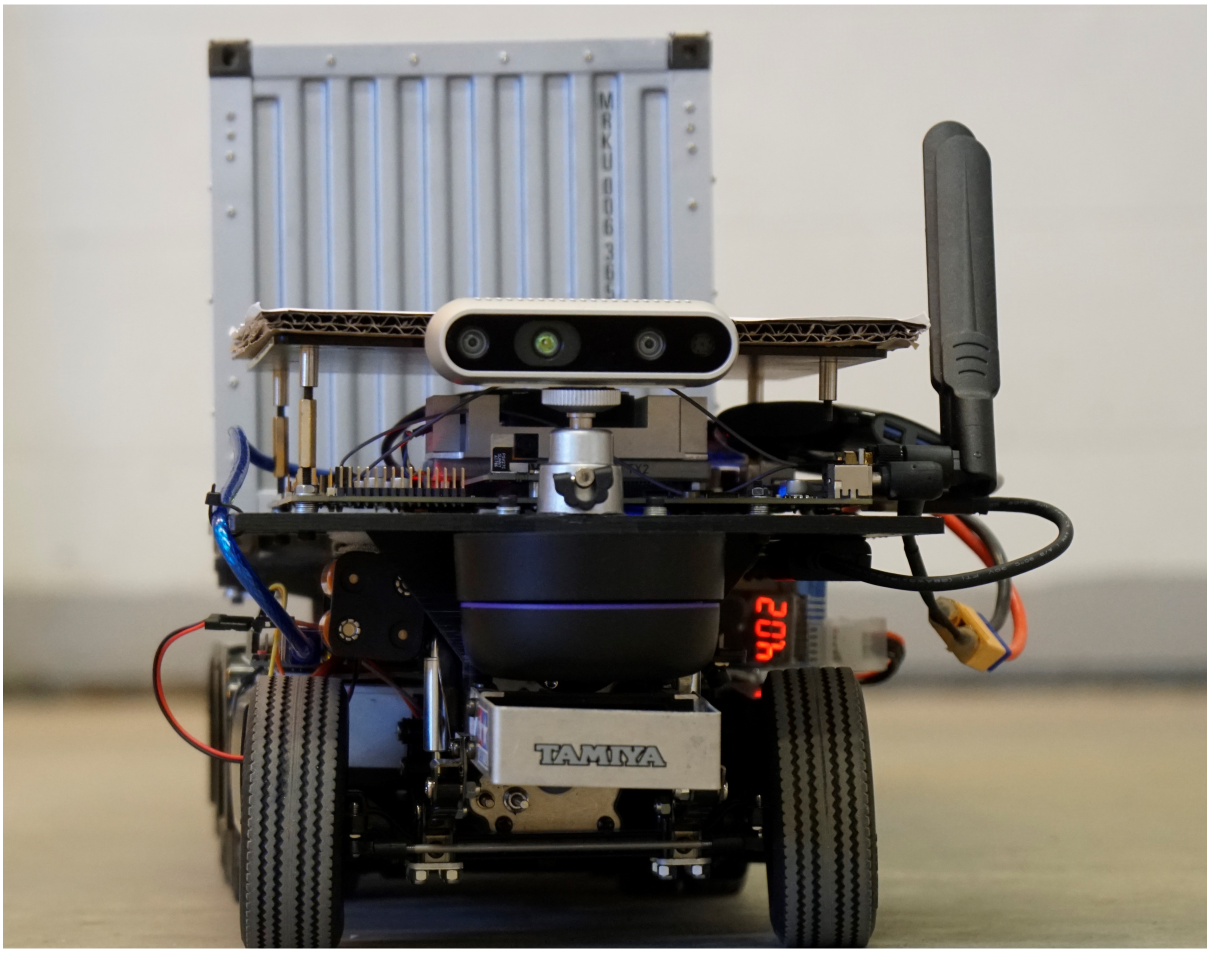


Figure F.4: Front View of Prototype

Figure F.5: Prototype with Trailer Detached

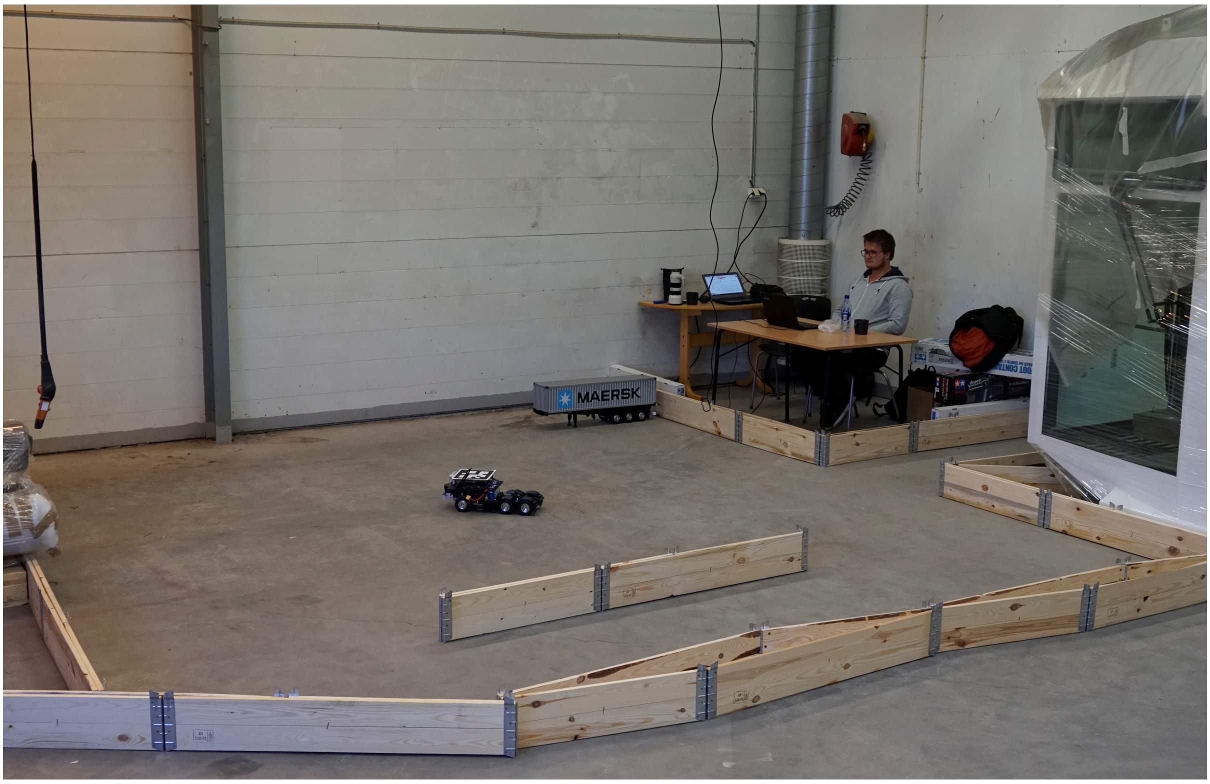
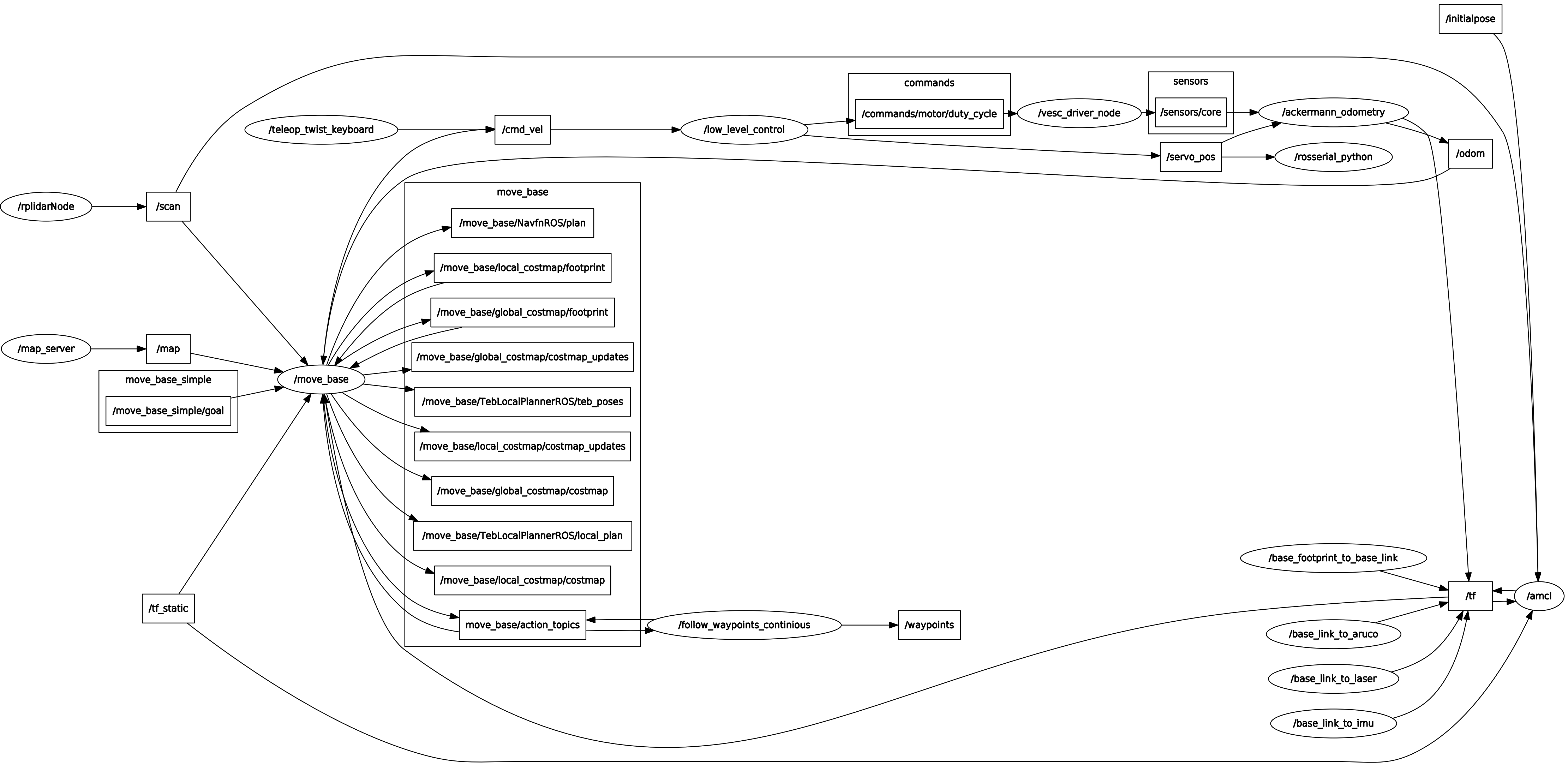


Figure F.6: Testing Area at Red Rock



/rplidarNode
/scan
/map_server
/map
move_base_simple
/move_base_simple/goal
/tf_static
/teleop_twist_keyboard
/cmd_vel
/move_base
move_base
/move_base/NavfnROS/plan
/move_base/local_costmap/footprint
/move_base/global_costmap/footprint
/move_base/global_costmap/costmap_updates
/move_base/TebLocalPlannerROS/teb_poses
/move_base/local_costmap/costmap_updates
/move_base/global_costmap/costmap
/move_base/TebLocalPlannerROS/local_plan
/move_base/local_costmap/costmap
move_base/action_topics
/follow_waypoints_continious
/waypoints
/low_level_control
commands
/commands/motor/duty_cycle
/vesc_driver_node
sensors
/sensors/core
/servo_pos
/rosserial_python
/ackermann_odometry
/odom
/initialpose
/base_footprint_to_base_link
/base_link_to_aruco
/base_link_to_laser
/base_link_to_imu
/tf
/amcl