Hochschule Reutlingen

Reutlingen University

–Studiengang Mechatronik–

Bachelor Thesis

Sensor-based Navigation of a Robot under ROS with Triangulation Scanner

Aabed Mohamed Sayed Abdelhamid Solayman

Matrikelnummer: 761224

Betreuer: Prof. Dr. rer. nat. Stefan Mack

Zweitbetreuer: Prof. Dr.-ING. Christoph Haslach

Abgabedatum: 01.09.2020

Declaration

Ich versichere, dass ich diese Thesis/Arbeit ohne fremde Hilfe selbständig verfasst, keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie alle wörtlichen oder sinngemäß übernommenen Stellen in der Arbeit gekennzeichnet habe. Die Arbeit wurde noch keiner Kommission zur Prüfung vorgelegt und verletzt in keiner Weise Rechte Dritter.

Acknowledgment

I would like to express my deepest appreciation to my academic supervisor Prof. Dr. rer. nat. Stefan Mack for his kind direction and support when it was needed the most.

I'm extremely grateful to my family for their continuous and unparalleled love, help and support. I express my sincere gratitude to both women, who have given me everything they could; my mom and my fiancee.

Contents

1 Introduction

1.1 Motivation

Mobile robots have received much attention in the past few years. The number of applications of mobile robots, especially in the industry, has increased enormously. The autonomous navigation is one of the main challenges in the design and implementation of mobile robots. The approaches developed for the navigation of mobile robots are very diverse. Most of these approaches are based on several hardware components, such as embedded laser scanners, odometry sensors, and other components. Thus, developing an approach where only one sensor is used instead of several sensors will decrease the robot's costs.
Furthermore, the development of mobile robots is a complex process. The motivation behind this project is to develop a maintainable robot navigation approach that can be transferable and adaptable to other robots and hardware components. This can be done by using ROS, which will be explained in later chapters.

1.2 Problem Statement

This project is divided into several hardware and software components. The main element is the PC, which can be seen as a master. Other elements, such as a triangulation laser scanner, a lego boost robot, localization algorithms, and motion control algorithms can be viewed as slaves.
First, a ROS system must be installed and created on the PC. Then a simple Lego Boost robot must be assembled, including the robot's connection to the PC. A triangulation laser scanner must be later integrated into the system. The next problem is the implementation of the localization algorithms. Lastly, the motion control algorithms must be implemented and integrated into the system.

1.3 Aim

The final aim of this project is to build a ROS system consisting of a Lego Boost robot, a triangulation sensor, a localization and control algorithm. All these components are represented by individual ROS nodes. The triangulation sensor is fixed in the room, and the controller lets the robot follow a user-predefined path in the room.

2 Theoretical Background

The core of the field of mechatronics is the combination of both hardware and software. A fundamental aspect in the field of mechatronics is the modularisation. Modularisation takes place both on the hardware and on the software level. The hardware modules used in this project are a differential driver robot and a triangulation laser scanner. On the other side, ROS is responsible for the implementation of the software modules. Before going deep into the implementation of the project and the methods used to solve the problem, both software and hardware modules will be explained.

2.1 Robots Operating System ROS

Designing robots in general, especially mobile robots, is considered as one of the essential elements of the industrial revolutions over the past decades. However, designing a robot is a complicated procedure. The main reason for that complicity is that robots' platforms consist of two main components, namely the hardware and the software applications.

The hardware of robots could be, i.e., sensors, motors, and actuators. On the other hand, software applications are responsible for doing a specific functionality, such as navigation, localization, or object recognition [1].

2.1.1 Robot Software Platforms

In order to simplify the robots' design procedure, scientists and researches were trying to develop so-called "Robot Software Platforms." In the literature, there seems to be no specific definition of Robot Software Platforms. However, robots software platforms can be interpreted as a middleware between the hardware components and the software modules, allowing developers to design seperated software modules, that could run on different hardware components without reprogramming the developed software module according to the different low-level drivers of each component [2][3] [1]. For example, a software module could be developed for navigation. By integrating this module in a robot software platform, it can run on different robots without dealing with each robot's underlying complexity. Using a proper robot software platform depends on each robot's purpose and requirements, as each platform has its advantages and disadvantages. One of the most used robot software platforms is ROS. ROS stands for Robot Operating System, which is confusing, since ROS is not an operating system by itself. Instead, it runs on an operating system like Linux and can be seen as a layer between an operating system and robot's applications, as shown in figure 2.1 [3].

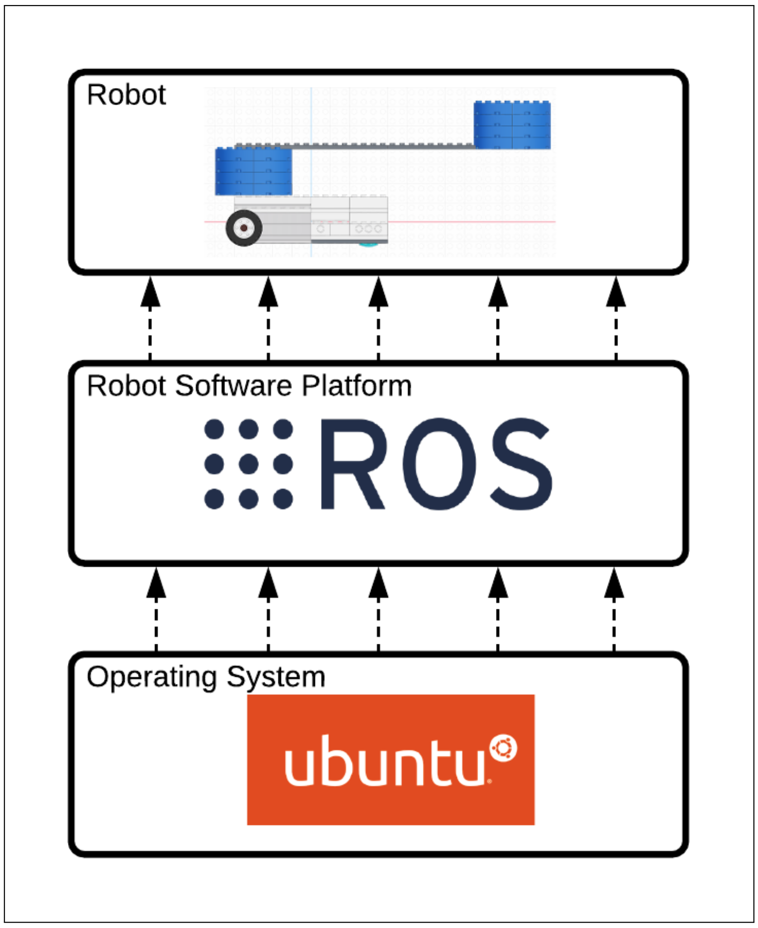


Figure 2.1: ROS as middleware between robots and operating systems

ROS has been chosen over other platforms for the implementation of the application developed in this thesis because of its ecosystem and the enormous amount of packages that are ready to be used and integrated into the application. Another advantage of ROS is its simplicity to learn. One of the ROS disadvantages is that it is not a real-time robot software platform, as it is built on Linux. However, realtime applications can be developed on ROS2, which is the next version of ROS. ROS versions are always tied to Ubuntu LTS (Long Term Support) versions and are released simultaneously, namely every two years. The used version of ROS in this project is ROS Melodic, which runs on Ubuntu 18.04. The next version of ROS is ROS Neotic, which runs on Ubuntu 20.04. ROS Noetic would be the last version of ROS. After that, ROS2 will be used instead.

2.1.2 ROS Working Principle

ROS working principle is explained in the literature in different ways. As explained in [1], ROS is based on a publish/subscribe concept, where processes communicate with each other via a message system. The best way to explain how ROS works is to analyze this publish/subscribe concept and explain the message system's main components and how messages are exchanged. This subsection explains the working principle of the ROS based on its description given in [3].

ROS applications consist of different processes, where each process is responsible for specific functionality such as navigation, path planning, or processing sensor data and so forth. These processes are called nodes. These nodes exchange information with each other in the form of messages. Each node can contain a publisher, that sends messages to a topic, or a subscriber, that receives messages from a topic, or both. A topic can be interpreted as a container that contains messages in nodes. TCPROS is the protocol that is responsible for exchanging the messages between different nodes in ROS, and it is based on a TCP/IP protocol. The central node in any ROS application is the master node, which is responsible for matching the information of the nodes with each other. In figure 2.2, the steps needed for establishing a communication between a publisher node and a subscriber node is displayed. First, the subscriber sends its information to the master node. This information contains, for example, the URI of the node, and the topics in the node. Next, the publisher sends its information to the master. The master then passes the publisher's information to the subscriber. The subscriber now recognizes the publisher, and the communication is established. The task of the master node is only to exchange the nodes' information. Once the nodes recognize each other, the master node role is completed. ROS nodes also contain other information other than publishers and subscribers, such as parameters, service servers, and action servers.

As described in [4], ROS has its own build system, which is called catkin. In each ROS project, a file called "CMakeLists.txt" must be defined. This file contains the dependencies

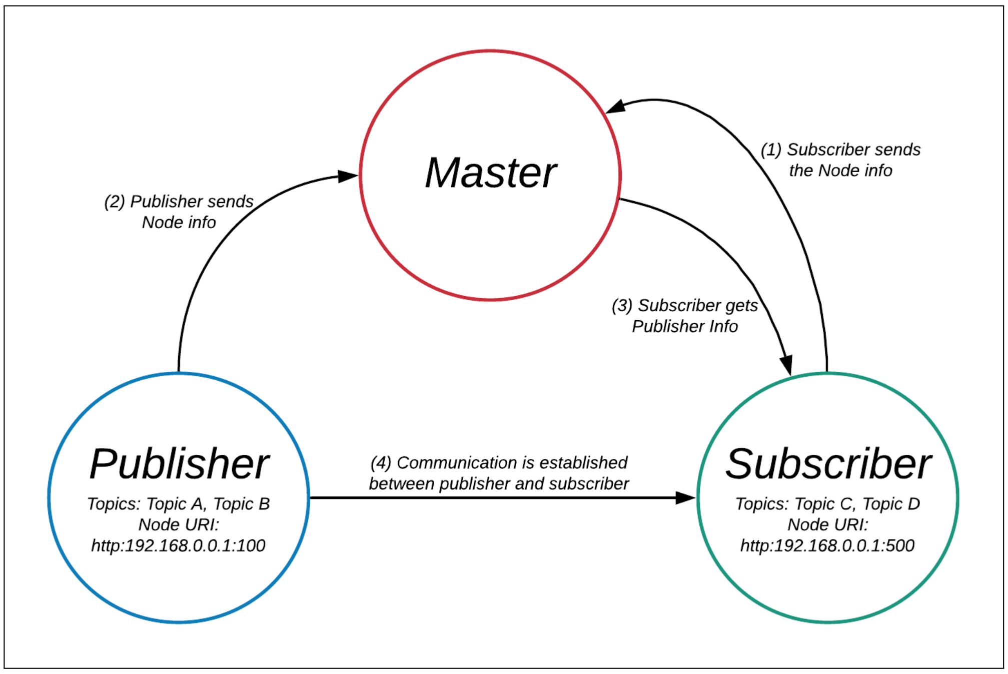

Master
(1) Subscriber sends the Node info
(2) Publisher sends Node info
(3) Subscriber gets Publisher Info
Publisher
Topics: Topic A, Topic B
Node URI:
http:192.168.0.0.1:100
(4) Communication is established between publisher and subscriber
Subscriber
Topics: Topic C, Topic D
Node URI:
http:192.168.0.0.1:500

Figure 2.2: ROS architecture design

of the project that are needed for the build process. Catkin is also responsible for creating the workspace, in which all project components are installed. Inside the catkin workspace, the project packages are defined. Packages in ROS can be interpreted as libraries. Each package contains files, in which nodes, messages, services, and functionalities are defined.

In summary, it is important to know that ROS's core feature is the modularization in terms of software. Modularization has many advantages, such as code reuse and working on loosely coupled processes.

2.2 Differential Driver Robots

The definition of mobile robots is usually associated with autonomous movement. As explained in [5], mobile robots are autonomously moving robots that can reach a specific goal. Therefore, an essential property of mobile robots is that they can plan their motion independently. Mobile robots are differentiated based on the environment they are used in, such as air, ground, and water [6]. Thus, the definition of mobile robots does not just include wheeled mobile robots, but generally, any robot that can move autonomously, such as drones, wheeled robots, or even legged robots. As shown in figure 2.3, wheeled mobile robots can be classified into three main categories: car-like mobile robots, omnidirectional mobile robots, and differential mobile robots [5].

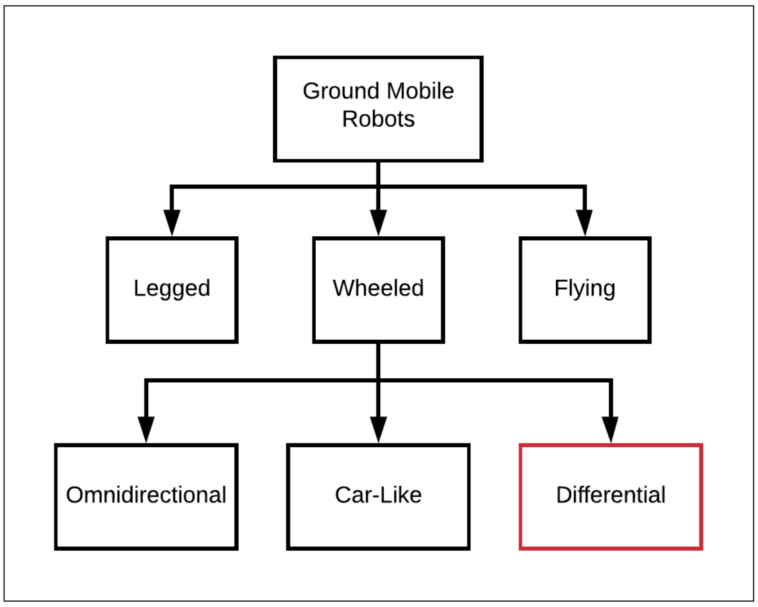


Figure 2.3: Different types of ground mobile robots

The difference between these categories relies on the driving system of each robot. This subsection gives a theoretical explanation of the differential mobile robots since it is the type of mobile robot used for the project.

Differentially driven mobile robots are used in various applications, such as internal logistics or automated gastronomy, due to its design simplicity. As shown in figure 2.4, the drive system of differentially driven mobile robots consists of two wheels placed on the right and the left side of the robot, where each wheel is controlled by a separate motor [5]. Moreover, most of the time, there is one more omnidirectional wheel or a caster wheel that provides the stability of the robot. This wheel enables the robot to move even on uneven surfaces. The speed and direction of the right and left wheels determine the robot's motion. By using different wheel speeds and directions, the motion of differentially driven robots can be explained as an arc around a point, which is the Instantaneous Center of Rotation (ICR), as shown in figure 2.4 [7]. The properties of the ICR, namely, the distance between the robot and the ICR, can be determined by knowing the velocity of each wheel and the distance between the wheels. This kinematic model is explained in [7] as follows:

$$\dot{\theta} = \frac{v_L}{L} = \frac{v_R}{R} \tag{2.1}$$

$$\dot{\theta} = \frac{v_R - v_L}{W} \tag{2.2}$$

$$\dot{x} = v \cdot cos(\theta) \tag{2.3}$$

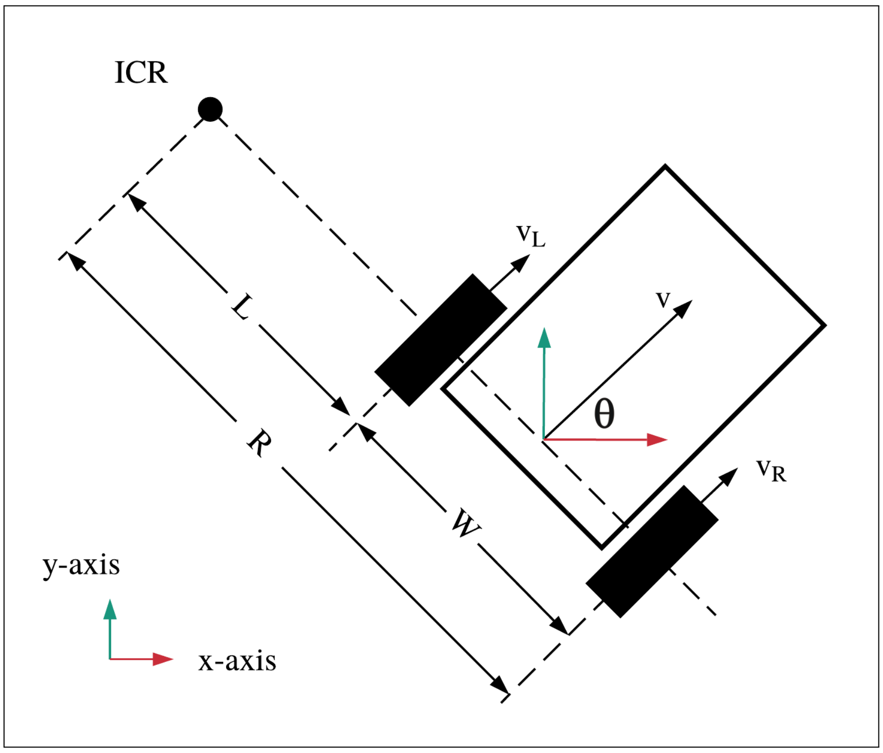


Figure 2.4: Basic design of a differential driven mobile robot

$$\dot{y} = v \cdot sin(\theta) \tag{2.4}$$

$$\dot{\theta} = \frac{v_\Delta}{W} \tag{2.5}$$

where v_L, v_R are the velocities of left and right wheels, respectively

L, R are the distance from IPC to the left and right wheels, respectively

W is the distance between the left and right wheels

$v_\Delta = v_R - v_L$

$v = \frac{1}{2} \cdot (v_R + v_L)$

By using these equations, a control algorithm can be designed to achieve a specific goal such as, following a trajectory, or following a line, or moving to a specific point. The implementation of the control algorithm is explained in a later chapter.

2.3 Triangulation Laser Scanner

Another module of the project is the laser scanner. The laser scanner is used to localize the differential driver robot position and orientation. The implementation of the localization process is discussed in later chapters. In this section, the advantage of the triangulation laser scanner compared to other types of sensors is discussed as well as the working principle of the specific sensor used.

2.3.1 Localization Sensors and Methods

Wheel Odometry:
The position of the robot can be estimated by using rotary encoders. The rotary encoders are embedded in each wheel. The idea behind rotary encoders is to calculate the number of revolutions of the wheel. By knowing the radius of the wheel, the moved distance can be then calculated. The wheel odometry method is simple and is used in many applications. However, several problems occur while implementing this method. As described in [8], the main problem is the position draft due to the wheel slippage. This leads to inaccurate position estimation.

GPS:
Global Positioning System (GPS) is another localization method that can be used. It is used in different applications as transportations, agriculture, and aviation. Using a GPS receiver, the position of the robot can be determined with the help of four satellites. The main problem with using a GPS sensor is its measurement accuracy. For example, Adafruit Ultimate GPS has an accuracy of 3 meters and costs around 40€[9]. Another sensor is SparkFun GPS-RTK2, which has an accuracy of 2.5 meters and costs around 200 €[10]. Another problem is the reduction of accuracy in indoor applications. Thus, GPS sensor-based applications are used outdoors due to week signals from the GPS satellites.

Ultrasonic Sensors:
Ultrasonic sensors measure the distance to an obstacle by using ultrasonic waves. As described in [8], the major disadvantage of ultrasonic sensors is the dependence on the material and the orientation of the measured object surface. Another disadvantage of the ultrasonic sensors is the poor lateral resolution (also called angular resolution). Additionally, the noise of the environment affects the accuracy of the measurement. In figure 2.5 is an example of ultrasonic sensors.

Laser Sensors:
Laser sensors are widely used for navigation and localization purposes. The application purpose of laser sensors is similar to ultrasonic sensors, where the distances to objects are

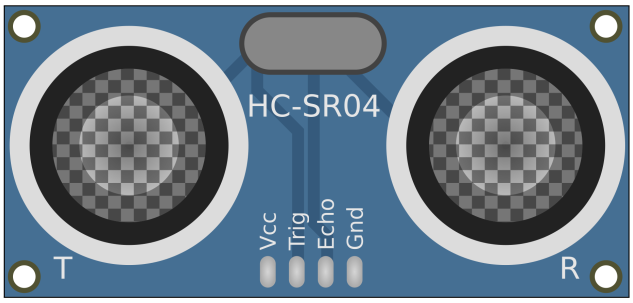


Figure 2.5: Ultrasonic sensor

detected. Laser sensors, however, as the name indicates, use a laser instead of ultrasonic waves. Laser sensors can be specified into two main types, namely position-based (triangulation) or time-based (LIDAR) sensors. LIDAR sensors calculate the distance to an object by measuring the time the laser took from the laser source to the object and back. An example of LIDAR sensors is the Sick 2D-Laserscanner LMS100, which costs around 3000 €.

Triangulation Laser scanners use another technique to measure the distance to objects. It does not use the time of flight but the direction of the reflected light to determine the range of an object. In this project, the triangulation laser scanner YDLIDAR X2 which is displayed in figure 2.6, is used. It costs around 70 €, which is much cheaper than LIDAR sensors. The working principle of the triangulation laser scanner will be discussed in the following subsection.

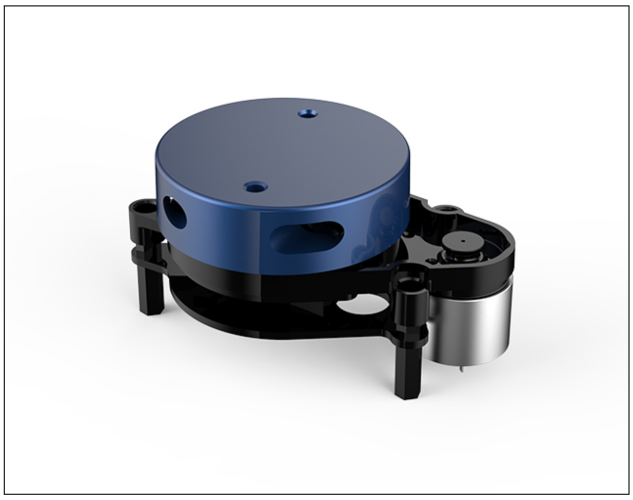


Figure 2.6: YDLIDAR X2 triangulation laser scanner. ©YDLIDAR All rights reserved

As explained in [8], the following table describes the advantages and disadvantages of each sensor.

Sensor	Advantages	Disadvantage
Wheel Odometry	Simple implementation Cost-effective	Position drift inaccuracy
GPS	Good for outdoor applications	Low accuracy Expensive
Ultrasonic	Cost-effective	Low accuracy due to poor lateral resolution
Laser sensor	Cost-effective Good accuracy level	Accuracy depends on reflection properties of the surface

Table 2.1: Comparison between different types of sensors

2.3.2 Triangulation Laser Scanner Working Principle

The triangulation laser scanner consists of three main components, namely a laser, an optical lens, and and a position sensitive photo diode (PSD) also called ”line detector” since the photodiodes form a line. When the laser hits an object, it reflects and travels to the line detector throw an optical lens, as displayed in figure 2.7.

The distance to this object is then calculated using the line detector. That is why it is called a position-based sensor. The YDLIDAR X2 triangulation laser scanner rotates around its axis and scans the surrounding area with a scan angle of 360°. The angle resolution is 0.72°. The angle resolution is the angle difference between two consecutive scans. These specifications are essential for the localization process and will be discussed in later chapters. In appendix A the datasheet of the YDLIDAR X2 triangulation sensor is appended.

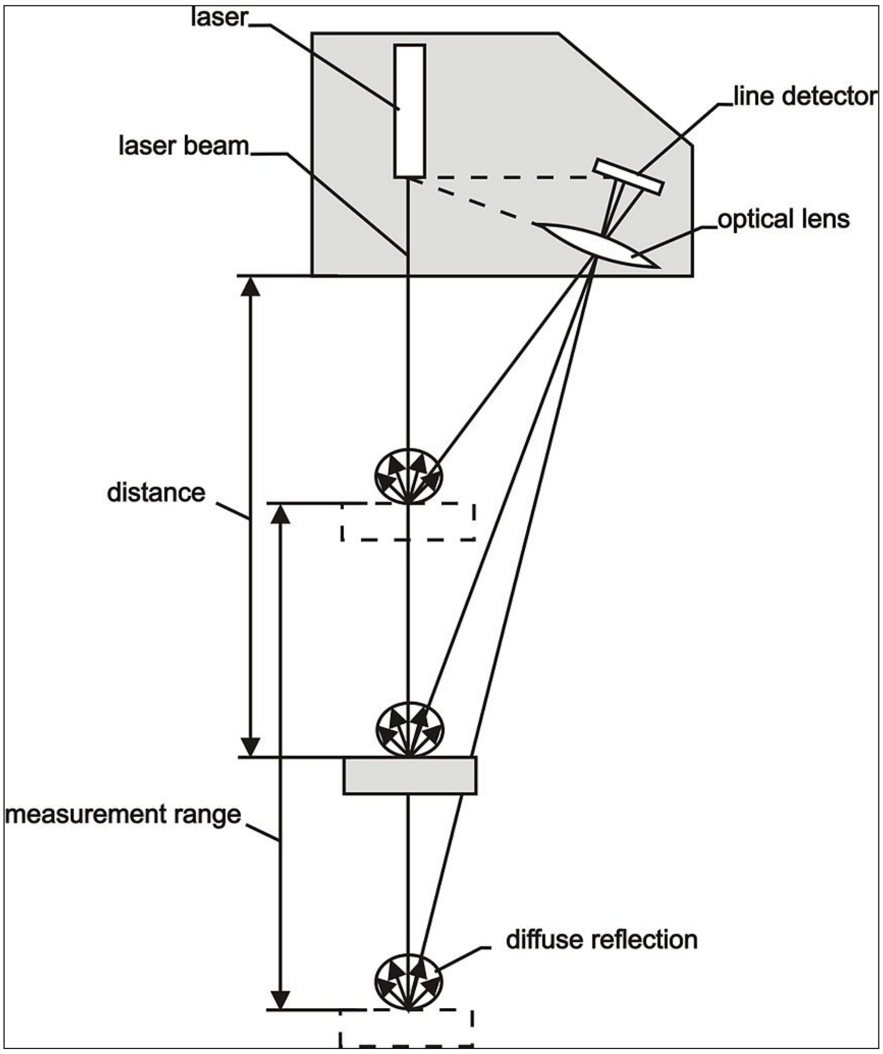


Figure 2.7: Triangulation principle [11].

3 Methodology and Implementation

As mentioned previously, this project consists of several components that run in unison to each other. Each component can be thought of as a building block to form a wall. To successfully build the wall, each block must be carefully placed so that it effectively supports the next. These components are ROS, Lego Boost robot, Localization algorithms, and motion control algorithm.
In this chapter, the implementation and methodology of each component will be explained.

3.1 ROS Software Architecture

For a better understanding of the developed ROS software, it is important to describe its architectural design and explain the connection between its different components. This section describes the overall ROS software architecture, as mentioned in [2] and also the architecture of the implemented software.

In general, the architecture of any software which is based on ROS can be divided into two main layers. The first layer is the ROS computation graph. This computation graph contains nodes, topics, messages, and other components. The second layer is the ROS filesystem, which contains all files and packages of the software. A general example of the architecture design of a software-based on ROS is shown in 3.1. The components of each layer of the software are described and discussed in detail in the following subsections.

3.1.1 ROS Filesystem

The top component of the filesystem is the catkin workspace, which contains all used ROS packages. Beneath the catkin workspace come the packages. The packages provide different tools and libraries, which could be used and configured to accomplish the desired functionalities. Each package contains messages, data types, services, and most importantly, executable

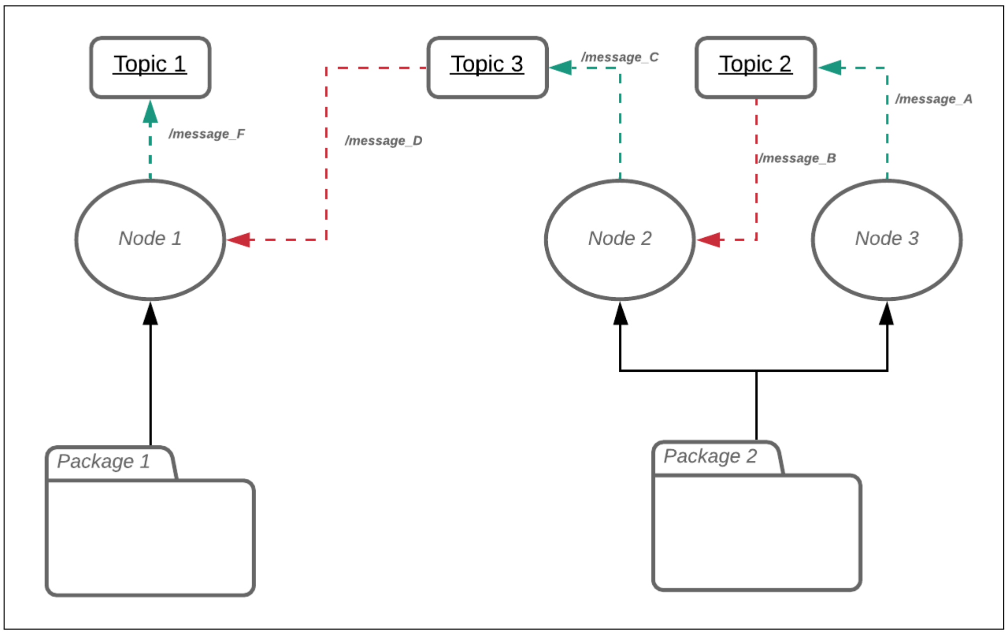


Figure 3.1: ROS architecture design

files and launch files [2]. The executable files written either in C++ or in Python, are used to initialize ROS nodes. The launch files are used to configure and launch specific nodes. Since ROS is an open-source tool, all its packages can be found through the ROS Software Browser. During installation, there are some default packages that are recommended while downloading ROS. Other than this, all remaining packages must be downloaded or cloned individually from the Software Browser or alternatively be made from scratch. The following figure 3.2 clearly demonstrates all the packages used in this project and the nodes that they include. The directory structure of the whole system is displayed in appendix B

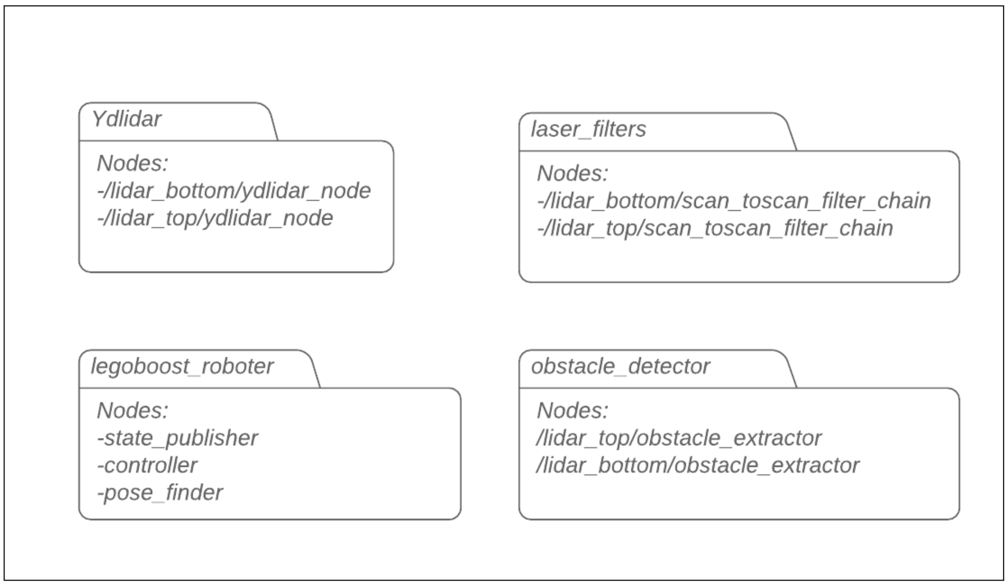

Ydlidar
Nodes:
-/lidar_bottom/ydlidar_node
-/lidar_top/ydlidar_node
laser_filters
Nodes:
-/lidar_bottom/scan_toscan_filter_chain
-/lidar_top/scan_toscan_filter_chain
legoboost_roboter
Nodes:
-state_publisher
-controller
-pose_finder
obstacle_detector
Nodes:
/lidar_top/obstacle_extractor
/lidar_bottom/obstacle_extractor

Figure 3.2: ROS packages of the project

Packages needed to be installed to support the default ROS packages are:

1. **Ydlidar Package:** This package should be installed first. It supports the YdlidarX2 triangulation laser scanner, which is developed by Shenzhen EAI Technology Co. It contains all the executable files that process the laser scanner data and publishes it as a LaserScan message to a topic. The nodes created in these packages are the /lidar-bottom/ydlidar-node and the /lidar-top/ydlidar-node, which contain information about the bottom and top laser scanners, respectively. Further information about the nodes is discussed in the computation graph section 3.1.2.

2. **The Laser-Filters Package:** As mentioned in [12], this package is responsible for adjusting messages of type LaserScan obtained from the triangulation laser scanner by using predefined filters so that it is suitable for later processing. Another option is to use pointCloud messages, but this option is not required as the laser scanner publishes messages of type laserscan and not point-cloud. This package includes a node called scan-toscan-filter-chain, which is a pipeline of filters defined by the user. It is useful for instances such as removing laser scan points of the wall to concentrate only on the robot laser scan data.

3. **The Obstacle Detector Package:** As seen in [13], this package is initially used for obstacle detection and tracking by using laser scan data from laser scanners. However, in our case, this package is used to detect the robot. The node used in this package is called the obstacle-extractor node. This package also has a user-defined message type called "Obstacle" which includes the position of the obstacle in the XYZ plane. It also has nodes to visualize the obstacles in rviz.

4. **The Legoboost-Roboter Package:** Unlike the other packages, this one is made from scratch and contains all the information about the robot. This package consists of three main components; the controller, pose detector, and state publisher, each of which has a specific function. The state publisher consists of a node responsible for publishing the robots' frames and contains its URDF file. The pose detector is essentially a node that calculates the position and orientation of the robot using data collected from the obstacle collector. And lastly, the controller, which has a node that is responsible for the motion control of the robot.

3.1.2 ROS Computation Graph

This section will go more in-depth on the computation graph layer for a better understanding of the software architecture as described in [2]. The computation graph is responsible for the connection between these components and the way the information is exchanged between these components. ROS node is one of the main components of the computation graph. It is responsible for computing a specific functionality or task. ROS Nodes are connected together through topics or services. They could subscribe to topics to get the information from these topics and use this information to perform a specific task. They can also publish information to topics to post-process this information. Information that is exchanged between topics is called a message. Messages types can be either built-in, for example String or Bool, or can be defined by the user. Figure 3.3 shows the nodes used in this software and how they are interconnected. Any item within a square notion that this item is a topic, however, an item within a circle relates to a node. This figure is generated by the rqt tool from ROS. This is a framework that is used for GUI purposes.

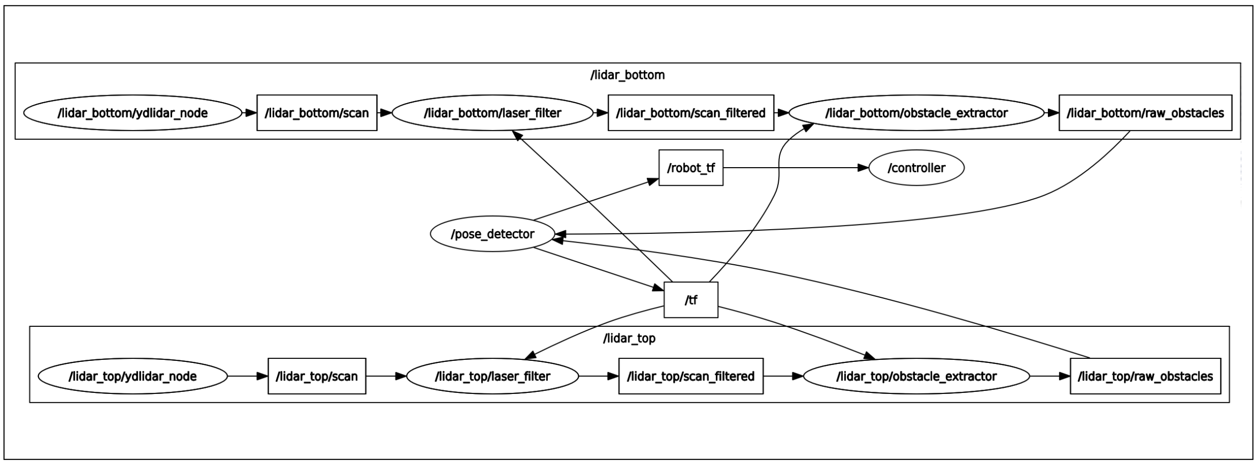


Figure 3.3: Relation between the nodes used in the project

Since there are two laser scanners and the data of each laser scanner must be processed individually, it is essential to launch separate nodes for each one. This can be done by using the attribute ”ns” in the launch file. By using this attribute, an instance of the node can be created in a user-defined namespace so that separate nodes are initiated. Figure 3.3 shows these two namespaces, which are named ”lidar_bottom” and ”lidar_top”.

As described in the previous figure, the nodes are:

1. **State-Publisher:** This node is responsible for publishing the joint states and trans-

formers of the robot.

2. **lidar_bottom/ydlidar_node and lidar_top/ydlidar_node:** These nodes publish messages of type LaserScan to the topics "lidar_bottom/scan" and "lidar_top/scan" respectively. As defined in [14], a LaserScan message in ROS is defined in sensor_msgs package and it contains information about laser scanner scans such as the ranges of each angle as well as the scan time and maximum and minimum range.

3. **lidar_bottom/scan_filter and lidar_top/scan_filter**: These nodes subscribe to the scan topics which are published from the ydlidar_node and publish a filtered LaserScan message to the topics "lidar_bottom/filtered_scan" and "lidar_top/filtered_scan". The scan_filter nodes in each laser scanner use LaserScanBoxFilter, which exclude scan points that are outside a defined room[12]. The room is defined by two points in an XYZ-space. This is important to exclude the scan points of the room's walls. By doing this, only the scan points of the robot are going to be considered. Figure 3.4 shows the plan view of the room, and the position of the laser scanners as well as the coordinates of the room. The XY coordinate of the laser scanners is (0,0), and the room's length as well as the room's width, is 1.6 meters. The laser scanner is placed 15 cm apart from the wall in both x and y directions. It's therefore concluded that P_{min} = (0.15,-0.15) and P_{max} = (-1.45,1.45).

4. **lidar_top/obstacle_extractor and lidar_bottom/obstacle_extractor:** subscribe to the scan_filter topic of each laser scanner and publish messages of type Obstacles to the topics "/lidar_top/raw_obstacles" and "/lidar_bottom/raw_obstacles."

5. **Pose_Detector:** This node subscribes to the raw_obstacle topic of each laser scanner and publishes the pose of the robot as a transformer message to the topic "robot_tf".

6. **Controller:** This node subscribes to the robot pose and calculates the velocities of the left and right wheels. Then it sends velocity commands to the control unit.
The implementation of obstacle extractor nodes, as well as the pose_detector node and controller node is discussed in further sections.

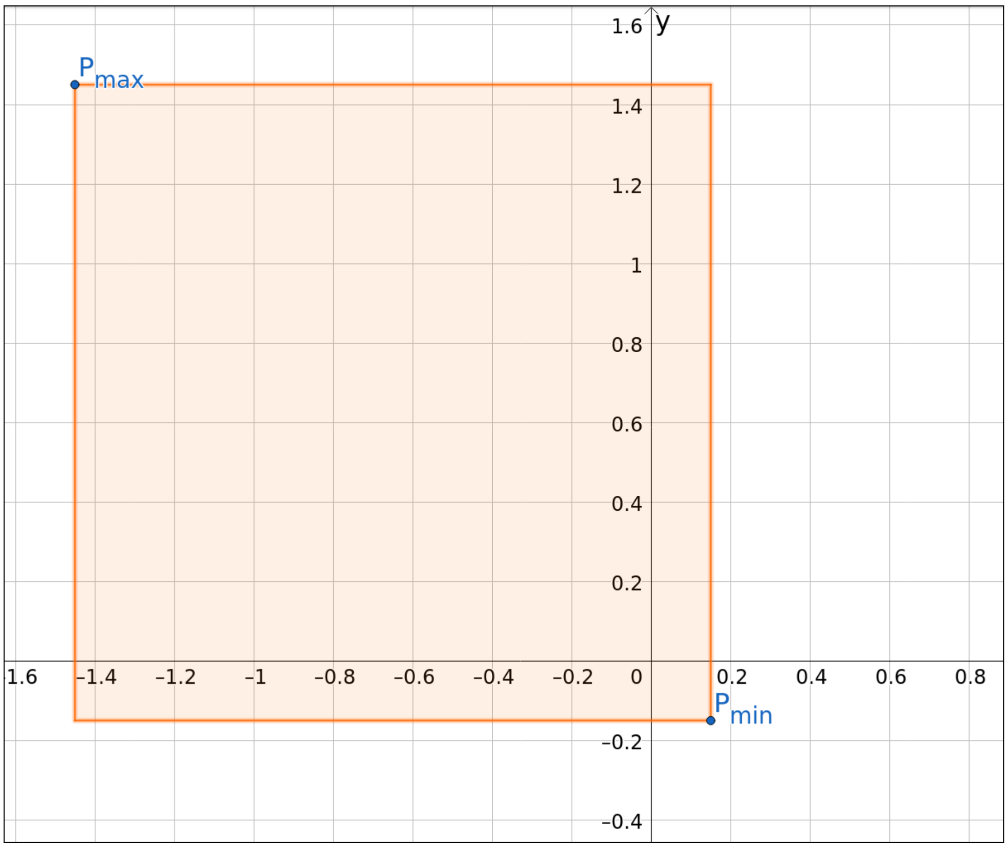


Figure 3.4: Plan view of the room

3.2 Model Architecture and Visualization

The design and visualization of the required model are two other building blocks that are essential for the successful completion of the project. Visualizing the components of the model using visualization tools that are provided by the ROS is important for tracking results and debugging the robot's motion. The designed model consists of three main components, namely the robot, the laser scanners, and the room. In the first subsection, the specifications and requirements of each component are explained in detail. Based on these requirements and specifications, the design of each component is going to be explained. In the second subsection, the visualization of the model in rviz is explained as well as the implementation of the robot's URDF.

3.2.1 Model Design

System Requirements:

The first requirement of the system concerns the robot and laser scanner design. These two components must have a design that facilitates the localization process of the robot for the triangulation laser scanners to detect the position as well as the orientation of the robot. The second requirement is that the room design must be rectangular or square-shaped. Besides, the floor of the room must be a whiteboard, and the laser scanners must be placed at a corner of the room. Lastly, the area of the room must be large enough for the robot to navigate within, yet not too large to ensure precise detection of the robot pose.

System Specifications:
Some of the robot specifications are already known since they are closely tied to the Lego Move Hub specifications because it is the central control unit of the robot. However, other specifications must be included that are based on the requirements discussed above. These are listed below.

1. Communication Protocol: Move Hub uses a Bluetooth Low Energy (BLE) processor for the communication between the control unit and the PC. The Bluetooth communication is based on GATT protocol.

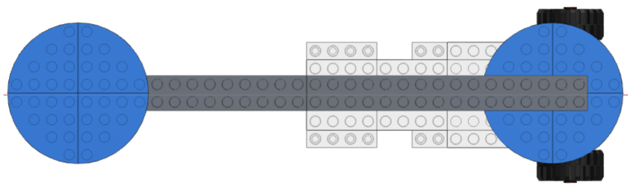
2. Steering system: Since Move Hub has two separated motors, each controlled individually, then the steering system of the robot is differentially steered. This steering system is nonholonomic.

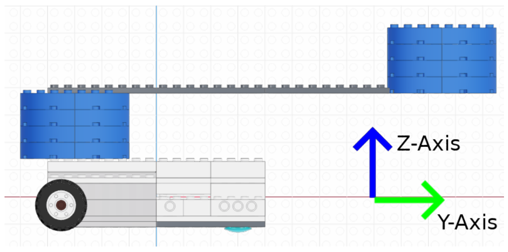
3. Robot design: The design of the robot is related to the first requirement mentioned previously. After several trials with different designs, a specific design was achieved as follows:

    - The robot has two cylinders with the same radius. One of these cylinders is placed at the front and the other at the back of the robot.
    - The cylinders are placed in different YZ planes.
    - The cylinders are wrapped with papers in order to decrease data loss. This papers are diffusely reflective.

    The following figure 3.5 shows the design of the robot. The design is made by Mecabricks.com, which is a web service to design models using Lego bricks.

4. Laser scanners design: The system has two laser scanners each determine a different cylinder. The laser scanners are placed on each other with the same orientation so that the x and y-axis of both laser scanners are identical. The laser scanner which is placed on top is on the same scanning plane as the cylinder which is placed on top.


(a) Plan view



(b) Side view

Figure 3.5: Plan and side view of the robot.

The same applies to the other laser scanner.

5. Room design: The fifth specification is the design of the room and is related to the second requirement mentioned in the requirements above. To design the room, it's important to consider two aspects. The first one is the laser scanner's specifications, which are discussed in section 2.3. The second aspect is the ability of the obstacle detector algorithm, which is used to detect the cylinders. By considering these aspects, the maximum area of the room can be determined as follows.
   The angle increment of the laser scanner is 0.72 $\frac{degrees}{step}$. To detect the circular obstacles an algorithm called "Obstacle Detector" is used. The implementation of this algorithm will be discussed in 3.3.1. However, for using the obstacle detector algorithm at least five laser beams of the object must be valid. The first and the last beam values are the tangents of the cylinder. The minimum valid angle between those tangents indicates the maximum range between the laser scanner and the obstacle. This angle is calculated using equation 3.1

$$\theta_{min} = i \cdot (R_{min} - 1) \tag{3.1}$$

where i = angle increment

R_{min} = minimum number of ranges

The challenge is to find this maximum length using this angle. This can be done by utilizing basic geometry rules using the angle found. The following figure 3.6 shows circle B, which has a radius r. The tangents of the circle are connected to the same point A. Theta is the angle between the tangents.

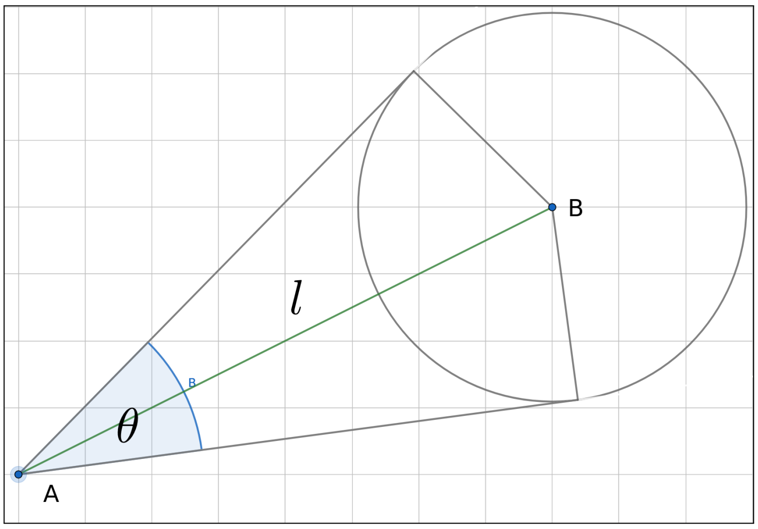


Figure 3.6: The Relation of the length between external point and center of a circle and the angle between circle tangents to this point

The length l can be calculated as follows:

$$l = \frac{r}{sin(\frac{\theta}{2})} \tag{3.2}$$

by substituting theta from equation 3.1 in equation 3.2

$$l_{max} = \frac{r}{sin(\frac{i \cdot (R_{min}-1)}{2})} \tag{3.3}$$

where $i = 0.72\ \frac{degree}{step}$

$R_{min} = 5$ steps

From this equation the room dimensions can be specified. By using cylinders with radius $r = 62$ mm. Then $l_{max} = 2.5$ m. The maximum distance in the room is the diagonal. The length of each side of the room could be calculated by using the following equation:

$$a = \frac{l_{max}}{\sqrt{2}} = 1.768m \tag{3.4}$$

Nevertheless, the range of error must be calculated for the previous analysis. Since the used triangulation laser scanner has a relative error of 1.5% for any distance between 0.5 m and 6m, and since there are 2 different laser scanners, then FRAGE(Error calculation)

3.2.2 Model Visualization

Visualizing the robot and the other components of the system using visualization tools that are provided by ROS is important for watching the results and monitoring the robot motion. The visualization tool used in ROS applications is called RViz, which is a GUI used to subscribe to topics and visualize the data in these topics. RViz contains specific display types for specific message types. For example, LaserScan display type displays the data from a sensor_msg::LaserScan message. However new display types could be added by using user-defined plugins. For example an obstacle display type is added to Rviz using a user-defined plugin. Another display type is the Robot Model, which represents the robot pose depending on its URDF. The URDF is an XML file which describes the joints and the links of the robot and the relation between them [15].

The next step is to write the URDF file of the LegoBoost robot. It simply consists of 4 links and 3 joints. The first joint is the base joint, which connects between two links, namely the base footprint which is the parent link, and the base link which is the child link. As defined in [16], the base footprint demonstrates the position of the robot at the ground level, while The base link is the position of the center of mass of the robot which is placed above the base footprint. These two links are connected through a fixed joint named base joint. The base joint is fixed since it doesn't have any degree of freedom. The second joint is the right wheel joint, which connects the base link to the right wheel link. The last joint is the left wheel joint, which connects between the base link and the left wheel link. The hierarchy of the URDF and the connection between the different links and joints is shown in figure 3.7. This figure is generated by urdf_to_graphviz tool.

Each link in the URDF file can contain visual properties. The visual properties could be used for a better visualization of the robot. The geometry of both right and left wheel links are implemented in the URDF as cylinders with the same radius of the robot's wheels. The base link geometry is implemented as a box with the same dimensions of the robot.
After implementing the URDF file, the next step is to set up the Rviz tool and add all the required display types, which are:

- Robot model: This displays the URDF of the robot.
- LaserScan: This displays the scan data of the laser scanners. A LaserScan display type

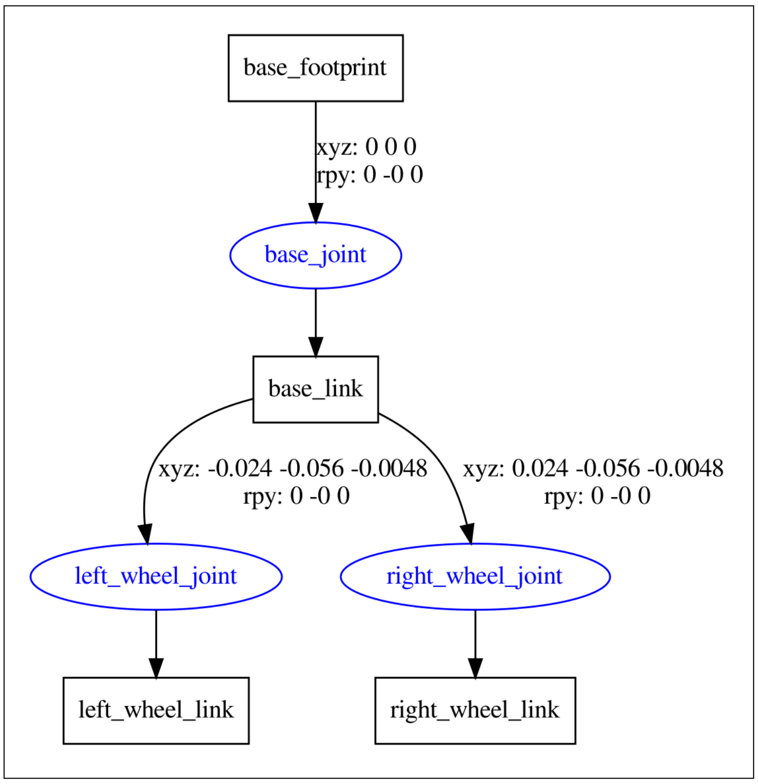


Figure 3.7: Graphical demonstration of LegoBoost URDF

is added for each laser scanner.

- Obstacles: This displays the cylinders on the robot. An obstacle is added to each cylinder. Each cylinder is colored differently, namely red for the top cylinder and blue for the bottom cylinder.
- TF: This displays the TF transform hierarchy of the model.

The final visualization of the whole model in RViz is shown in figure 3.8. The frames of the laser scanners as well as the robot are demonstrated. The red and white points demonstrate the laser scans of the laser scanners and the red-green cylinder shows the back cylinder of the robot. Lastly, the grey-blue cylinder demonstrates the front cylinder of the robot.

3.3 Robot Localization

Robot localization is another essential building block in the project and the core of the thesis. Under robot localization, two things must be considered. First, is localizing the position of the robot, and second, localizing the orientation of the robot. Both the orientation and the position represent the pose of the robot. Localization in mobile robots context is usually performed by using SLAM algorithms, where a LIDAR or a camera is integrated into the

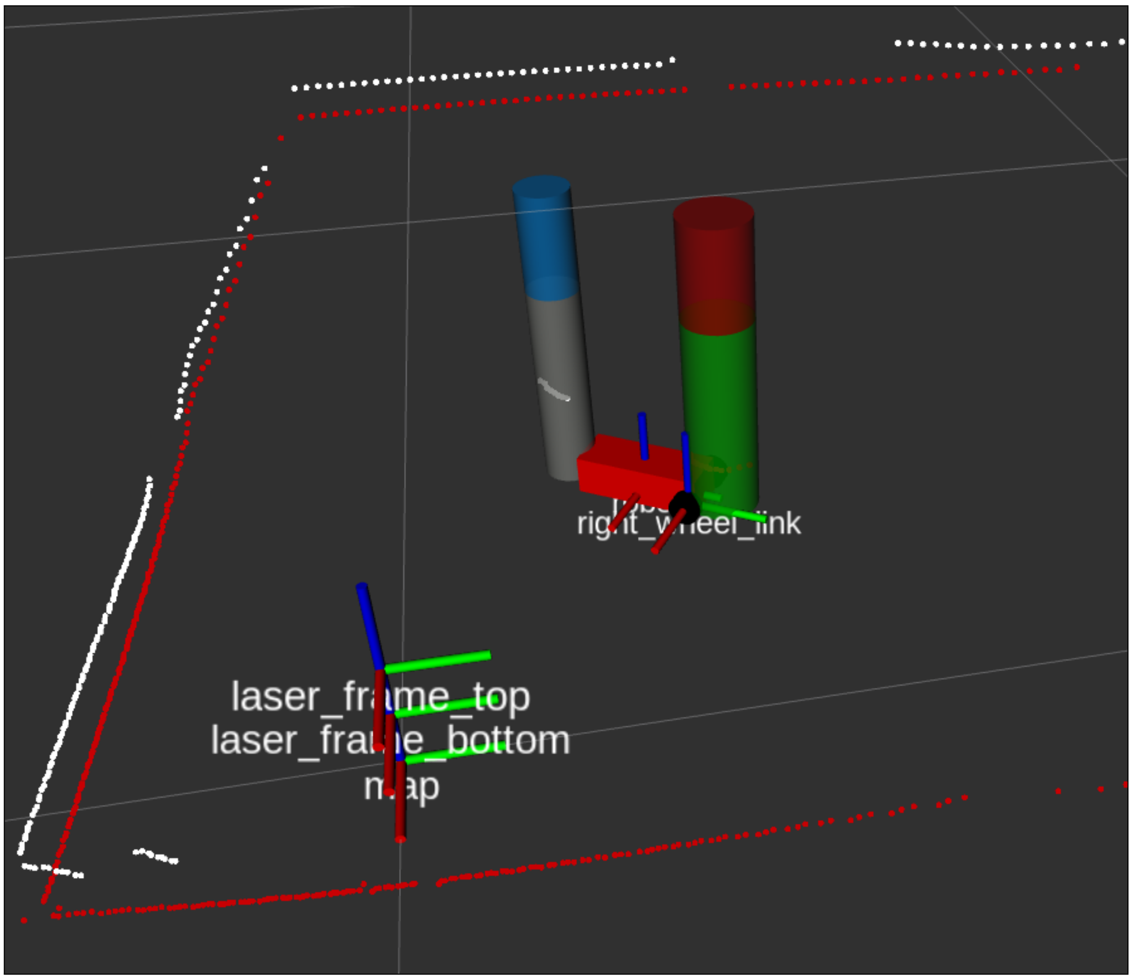


Figure 3.8: The model visualization in RViz

robot to scan the surrounding environment, and the SLAM algorithms are used to build a map of this environment and localize the pose of the robot depending on this map and other inputs like the odometry of the robot [3].

In contrast to the SLAM algorithms, the method used in this thesis is different since the laser scanner are not integrated into the robot but fixed in a specific place in the room. From the perspective of the laser scanner, the surrounding environment is still, and only the robot is moving. Using a fixed sensor in the room has an advantage, namely the cost efficiency of the solution by saving the battery of the robot and also by saving the costs of other robot components such as odometry sensors or IMU sensors. In this section, the used methods, which have been used for robot localization, are going to be discussed. First, localization using the obstacle detector approach is discussed, then using image processing is discussed, and last using the laser scan matcher approach.

3.3.1 Circular Object Detection

Obstacles detection algorithms are essential in robot navigation. Such algorithms are used,e.g., during navigation to avoid obstacles. The obstacle detection algorithm developed in [17] is used for detecting circle-shaped obstacles by using circles geometric properties and the polynomial regression of laser scan data. The main reason for choosing this algorithm is its fascinating efficiency and speed. According to the applied tests mentioned in [17], the accuracy rate of the circles' detection is 92.53%, and the average executing time is 16 ms per frame, which is around 50 Hz. This is more than twice the publishing rate of the X2 laser scanner, which is 20 Hz. This is the advantage of ROS, where a node could be reused and adjusted depending on the requirements or the problem. The main idea of using this algorithm is to detect the cylinders that are placed on the robot. The pose of the robot can be calculated by detecting the exact position of these cylinders in cartesian coordinates form. The implementation of this approach is divided into three main steps. The first step is to find a geometrical solution to the problem. The second step is tuning the obstacle extractor node parameters to fit with the specifications of the robot. The last step is to implement this geometrical solution in an executable file, i.e., C++ or Python.
The geometrical problem can be divided into two main problems. The first problem is finding the position of the robot. The position here means the middle point between the front and back cylinders in the XY plane. The second problem is finding the orientation of the robot. In other words, the main question is how to transform the coordinates of the laser scanner to the coordinates of the robot. Figure 3.9 shows an example of the robot coordinates P_{Robot} relative to the laser scanner P_{Sensor}. The points C_f and C_b represent the center of the front and the center of the back cylinders, respectively. The dotted coordinates indicate the laser scanner coordinates after transforming them into the robot's coordinates.

The position of the robot can be obtained by applying the midpoint formula as follows:

$$P_{Robot,y} = \frac{C_{f,y} + C_{b,y}}{2}, \quad P_{Robot,x} = \frac{C_{f,x} + C_{b,x}}{2} \tag{3.5}$$

To find the orientation of the robot, the angle θ must be calculated. By using the trigonometric function:

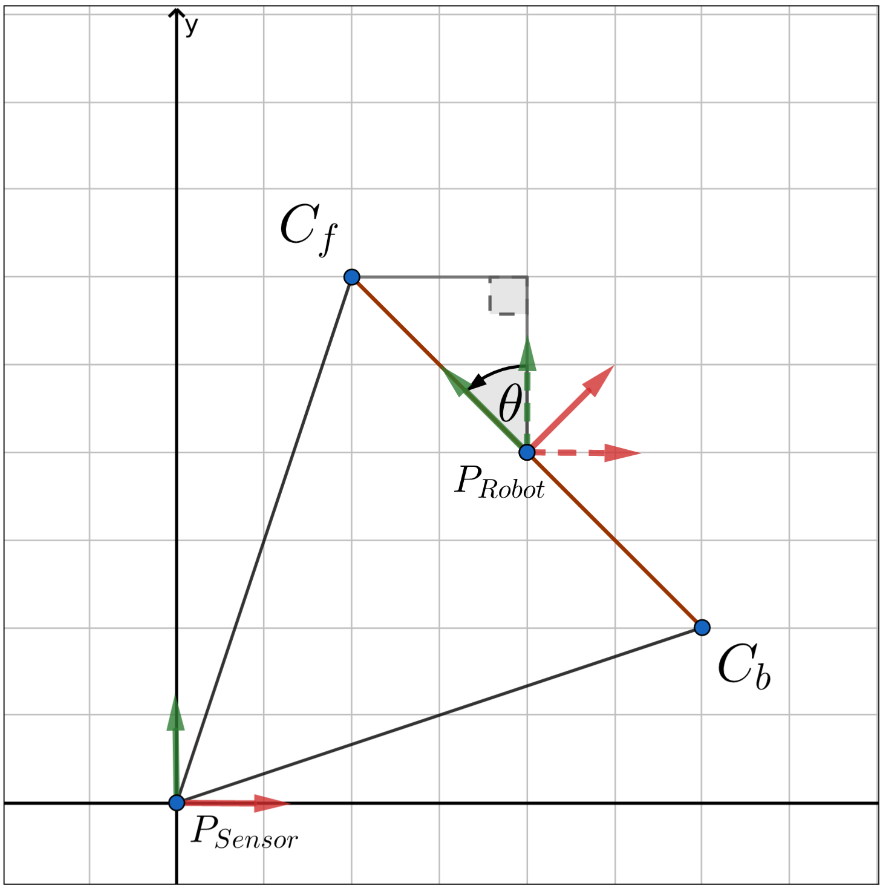


Figure 3.9: Robot and Sensor coordinates in the room

$$tan(\theta) = \frac{opposite}{adjacent}$$

then,

$$\theta = arctan(\frac{C_{f,y} - P_{Robot,y}}{C_{f,x} - P_{Robot,x}}) \tag{3.6}$$

After determining the geometrical solution of the problem and deriving the required equations to find the pose of the robot, the next step is to implement this solution in a ROS node. The first thing to be considered in the implementation is setting the obstacle extractor node parameters. This node is responsible for extracting the cylinders' position and publish them for further processing. The definition of each parameter is extracted from [13] and listed below:

By considering the requirements discussed in 3.2.1, the minimum_group_points parameter is set to 5. To find a proper value of the parameter maximum_group_distance, the maximum length of the room calculated in section 3.2.1 must be considered. At this distance, the triangulation laser scanner detects only five points of the cylinder. The distance between

Parameter	Definition	Value
active	Active/Sleep mode	true
use_scan	Use laser scan messages	true
min_group_points	Minimum number of points comprising a group to be further processed	5
max_group_distance	If the distance between two points is greater than this value, start a new group	0.06
max_merge_separation	If the distance between obstacles is smaller than this value, consider merging them	0.3
radius_enlargement	Artificially enlarge the circles' radius by this value	0

Table 3.1: The definition of obstacle_extractor node parameters

any two points, in this case, can be calculated using the formula of a circle's cord as follows:

$$c = r \cdot crd(\alpha)$$

$$c = 2r \cdot sin(\frac{\alpha}{2}) \tag{3.7}$$

where α is the angle between two points from circle center perspective.

c is the distance between these two points.

By using equation 3.1, the angle α can be calculated as follows:

$$\alpha = \frac{180 - (R_{min} - 1) \cdot i}{R_{min} - 1} \tag{3.8}$$

By substituting 3.8 in 3.7, then

$$c = 2r \cdot sin(\frac{1}{2} \cdot \frac{180 - (R_{min} - 1) \cdot i}{R_{min} - 1})$$

This can be simplified into

$$c = 2r \cdot sin(\frac{1}{2} \cdot (\frac{180}{R_{min} - 1} - i)) \tag{3.9}$$

By substituing r, R_{min} and i with the values discussed in 3.2.1, then $c \geq 0.0467$ m.

The value of the parameter max_merge_separation is the minimum distance between the cylinders. Since the distance between the cylinders is 0.216 m, then the value of max_merge_separation must be at least 0.216 m. The radius_enlargement is only for visualization purposes. This can be set to 0.

After setting the parameters of the obstacle_extractor node, the last step of the implementation is implementing the pose_detector node. The first thing to be considered is the programming language used for the implementation of the node. The high performance and high speed of pose calculations are essential in order to publish the pose of the robot at a proper rate to the controller. The higher the rate is, the more efficient the controller performance and the faster the robot would be. Since the publishing rate of this node is 100 Hz and also many calculations are taking place, this node is programmed in C++, since it is faster than Python.

3.3.2 Image Processing

Image processing and computer vision are commonly used in robotics for many applications, i.e., object recognition or avoiding obstacles. In this approach, the pose of the robot is detected using image processing. Image processing functionalities in ROS can be implemented using different libraries. One of the most powerful and used libraries in image processing is OpenCV. OpenCV is a C++ library. However, it uses a python wrapper. A python wrapper is used to implement the classes and methods of the C++ library in Python. In other words, these libraries are not available in Python as Python modules but as runtime libraries. These runtime libraries are created beforehand for the specific platform using the corresponding C++ modules. As mentioned in [18], OpenCV uses python modules to implement image processing solutions by using Numpy library. That means, OpenCV is highly associated with Numpy. The main idea of this approach is to convert the data in the sensor_msgs::LaserScan messages into sensor_msgs::Images. The sensor_msgs::Images message contains information like the height, the width and the encoding of an image.The converted images could be then processed using OpenCV library to detect the cylinders which are placed on the robot by their shape or size and then detecting the pose of the robot. It is important to know that ROS images are not compatible with OpenCV. That is because the format of each image is different,i.e, images in ROS are stored in arrays, while in OpenCV the images are stored in Numpy arrays. To process ROS Images in OpenCV, the cvBridge package must be installed. This package works as a link between ROS and OpenCV and converts ROS Images into OpenCV Images and vice versa [2].

The computational graph displayed in 3.10 is similar to the one discussed in 3.1.2, however this approach is using different nodes for the robot's localization. These nodes are:

- **ImageConverter:** The purpose of this node is to convert the LaserScan messages to messages of type sensor_msgs::Images.
- **Pose_Detector:** This node subscribes to "converted_images" topic, which has messages of type sensor_msgs::Images and convert the images in these messages into OpenCV images. Then, The OpenCV images should be processed to find the pose of the robot. This pose should then be published to the controller. However, this node hasn't been implemented. The reason for that is discussed at the end of this subsection.

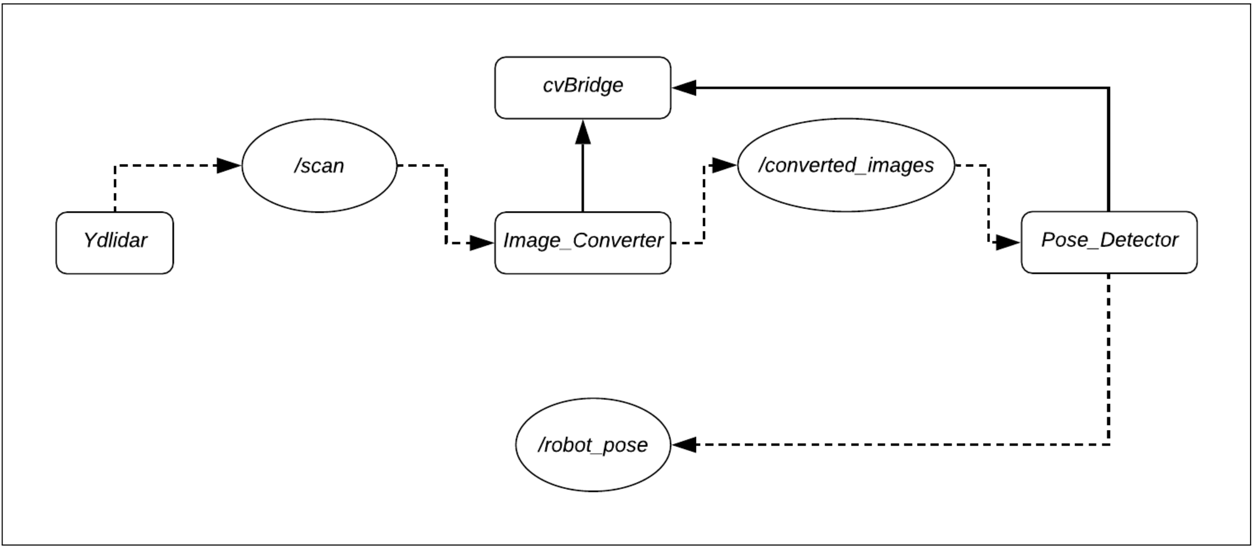


Figure 3.10: Computational graph of OpenCV method

The laser scanner generates a 2D point cloud, which is the content of the LaserScann Message. In this message, a measured distance is assigned for each scan angle. The ImageConverter node converts this point cloud into a 2x2 matrix, which is a binary image matrix. The pixels of the image matrix with the value "True" correspond to the XY-coordinates of the points in the point cloud.
To simplify the implementation of the ImageConverter node, the following activity diagram shown in figure 3.11 was designed. This step is essential to understand the flow control of the node. First, it subscribes to the "scan" topic to get the ranges of the LaserScan message. Then, the ranges are stored in a numpy array. Since some ranges may have invalid values,

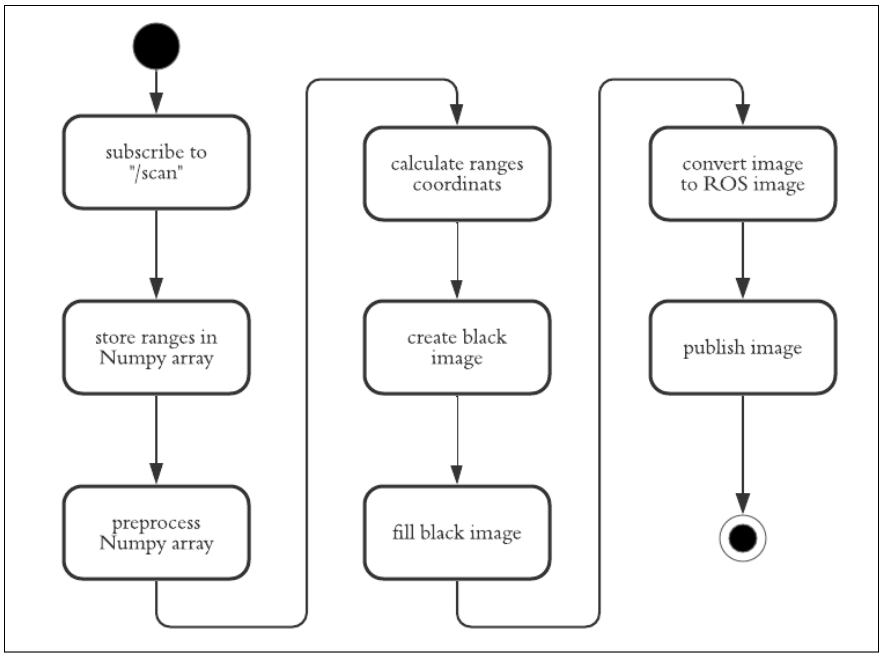


Figure 3.11: Activity diagram of the ImageConverter node

the ranges array must be preprocessed to delete these values. The next step is to calculate the exact position of each range in the XY plane. This can be done as follows:
Since the angle increment i of the laser scanner is known, then the position of each range can be represented as a point in polar coordinates as (r_n, θ_n), where $n \subset [0, \frac{360°}{i}]$. This point can be represented in Cartesian coordinates as (x_n, y_n) where:

$$x_n = r_n \cdot cos(\theta_n), \quad y_n = r_n \cdot sin(\theta_n) \tag{3.10}$$

By vectorizing the previous formula, the coordinates of each range point in XY plane can be represented as follows:

$$\begin{pmatrix} x_0 \\ \dots \\ x_n \end{pmatrix} = \begin{pmatrix} r_0 cos(\theta_0) \\ \vdots \\ r_n cos(\theta_n) \end{pmatrix}, \quad \begin{pmatrix} y_0 \\ \dots \\ y_n \end{pmatrix} = \begin{pmatrix} r_0 sin(\theta_0) \\ \vdots \\ r_n sin(\theta_n) \end{pmatrix} \tag{3.11}$$

The next step is to create a blank black image using OpenCV. After that being done, the color of corresponding pixels in this image to the positions from equation 3.11 must be changed. As Pixels are integer values, the positions calculated in 3.11 must be changed to integer values. This process has intelligibly a huge impact on the resolution of the localization process. To explain this impact, the following example is considered. Assuming the points P_1

and P_2, which are detected from the cylinder are next to each other, where a is the distance between those points. According to 3.9, $a = 0.018$ m, when the cylinder is detected by 10 points. Thus, it follows that the resolution of the image must be at least $1\frac{pixel}{mm}$. Since the room width and length are 1.5 m, then The size of the image can be calculated as follows:

$$\text{image size} = Resolution^2 \cdot width \cdot length = 2250 \cdot 10^3 \text{ pixels}$$

The next step in the activity diagram in figure 3.11 is to convert the openCV image into ROS image. The last step is then to publish the ROS image at a proper rate. Since, the Ydlidar laser scanner publishes LaserScan messages at rate of 20 Hz, the rate of this node must be greater than 20 Hz. This means each image must be processed withn 50 ms. This is a very short time to process the images and localize the robot. Thus, this approach is not used for localizing the robot. However, using image processing approach could be useful, if a camera is used instead of the laser scanner. By using a camera, objects would be easier to detect directly.

3.3.3 Laser Scan Matcher

Another approach to localize the pose of the robot is using the Canonical Scan Matcher (CSM) implementation. The CSM is a library in C that is used for scan matching problems, i.e., in mobile robot applications, and was developed by Andrea Censi [19]. As described in [20], CSM introduces a new method to compute the covariance of the ICP (Iterative Closes/Corresponding Point) algorithms, which are used for comparing different scans to each other and match these scans so that the difference between them is minimized. Fortunately, the CSM library is implemented into ROS as a package called laser scan matcher. This is the core algorithm of the well-known SLAM localization. To understand how the laser scan matcher package works, the following example is considered. Assuming a mobile robot is moving in a room, and a laser scanner is standing on the top of the robot. While the robot moves, the laser scanner scans the surrounding area and publishes scan frames. These scan frames are used as an input to the laser scan matcher node. The laser scan matcher node tries then to compare these scan frames to each other. By comparing the scan frames repetitively, the laser scan matcher node can estimate the difference between the scan frames and determine the pose of the robot. In this section, the implementation of the laser scan matcher package, as well as its disadvantages, are explained.

The following example is to be considered in order to understand the implementation of the laser scan matcher package. This example demonstrates the implementation of a robot, where a laser scanner is set on top of this robot and moving with it. This is different from the approach used in this thesis, where the laser scanner and the robot are not moving together. However, it would help to understand the concept of this approach.

According to figure 3.12, the laser scan matcher node publishes the pose of the robot to both topics tf and pose2D. For estimating the pose of the robot, the laser scan matcher node subscribes to several topics, as explained in [21].

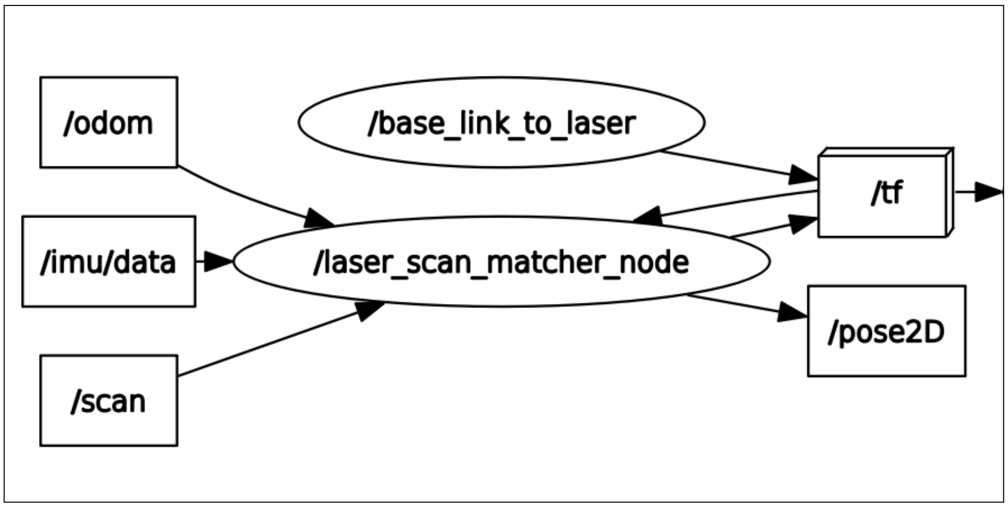


Figure 3.12: Computational graph of laser scan matcher

The first topic is "/scan", which contains the LaserScan messages. The second topic needed for the pose calculation is "tf". The "tf" topic contains the transformation messages between the laser scanner frame and the robot's base frame as well as the transformation messages between the robot's base frame and the world frame. The pose of the robot is computed relative to the world frame, which is a fixed frame in the map. These two topics are essential for pose calculation. However, other topics can be used to improve the accuracy of the laser scan matcher, such as 'IMU/data,' which contains data of an IMU sensor. Another topic that is used to improve the accuracy of the process is 'Odom'. This topic has the wheel odometry data, which is calculated using wheel encoders. Additionally, the IMU sensor could be used for a more accurate calculation of the pose. However, these supplementary topics are not used in the implementation of the robot, since Legoboost robot does not have such sensors.

The challenge momentarily is to adjust this example to fit with the aim of this thesis, where the laser scanner is not integrated into the robot.

In order for this to be done, two things must be modified. First, the fixed frame must be changed to be the laser scanner frame. Now, the pose of the robot is calculated relative to the laser scanner. The second thing is the scan messages. The laser scan matcher compares laser frames iteratively, and by matching these frames, the pose is calculated. This means that in order to calculate the pose of the robot, the scan plane must not be constant and must change. Since the laser scanner scans both the room and the robot, and the room's scan points are constant because the laser scanner is not moving, the whole scan frame is almost constant because the scan points of the room are way more the scan points of the robot. Thus, the scan points of the room must be excluded from the scan message using scan filters, as explained in section 3.1.2. However, excluding the room points would lead to another problem, namely the very few amount of the scan points, which cannot provide the laser scan matcher with enough data to calculate the pose of the robot. Thus, this approach is not used to localize the pose of the robot.

3.4 Motion Control and System Integration

The last component in the project is the motion control algorithm. As previously mentioned, the aim of the project at the end is to build a system where the robot follows a predefined path. This path can be interpreted as a straight line, a trajectory, or even only a pose in the room. For each kind of path, there is a specific control algorithm. The control algorithm implemented in this project is the "following a line" algorithm. However, in the future, other control algorithms can be implemented in separate nodes. This is the advantage of ROS, where all components are loosely coupled and can be implemented and activated separately. In this section, the implementation of the "following a line" algorithm is explained as described in [7]. After the implementation of the motion control, the integration of the ROS system including all nodes will be discussed.

3.4.1 Motion Control

Since two separate motors control the robot's movement, the kinematics of a differential drive robot must be considered. As explained previously, the speeds of the individual wheels determine both the angular velocity and the linear velocity of the robot.
The line that the robot has to follow can be described with the following equation:

$$ax + by + c = 0 \tag{3.12}$$

As mentioned in [7], in order to follow this line, two controllers must be considered. The first controller controls the robot's distance to the line, while the second controller controls the steering angle. The two controllers are then added together to determine the robot's overall motion, namely the speeds of the individual wheels.

The distance controller can be described as follows:

$$d = \frac{(a, b, c)}{\sqrt{a^2 + b^2}} \tag{3.13}$$

where d is the distance to the line.

The steering angle controller can be described as follows:

$$\theta^* = tan^{-1}(\frac{a}{b}) \tag{3.14}$$

By adding both controllers in 3.13 and 3.14, the controller can be described as follows:

$$\alpha = -K_d + K_h \cdot (\theta^* - \theta) \tag{3.15}$$

where K_d is the distance parameter
K_h is the heading parameter

The controller in 3.15 takes the position and orientation of the robot as input. The output of the controller is the angular velocity. Using the equations 2.2 and 2.5, the velocities of the right and left wheels can be calculated as follows:

$$V_L = \frac{1 - W \cdot \alpha}{R} \tag{3.16}$$

$$V_R = \frac{1 + W \cdot \alpha}{R} \tag{3.17}$$

The controller node is implemented in Python. It subscribes to the pose of the robot and does not publish any data. By implementing the node in Python, Python's powerful MatplotLib library can be used to plot the results. In order to adjust the controllers' parameters, a simulation in Simulink has been developed as displayed in figure 3.13. The simulation also helps to verify the solution and is used generally for debugging purposes.

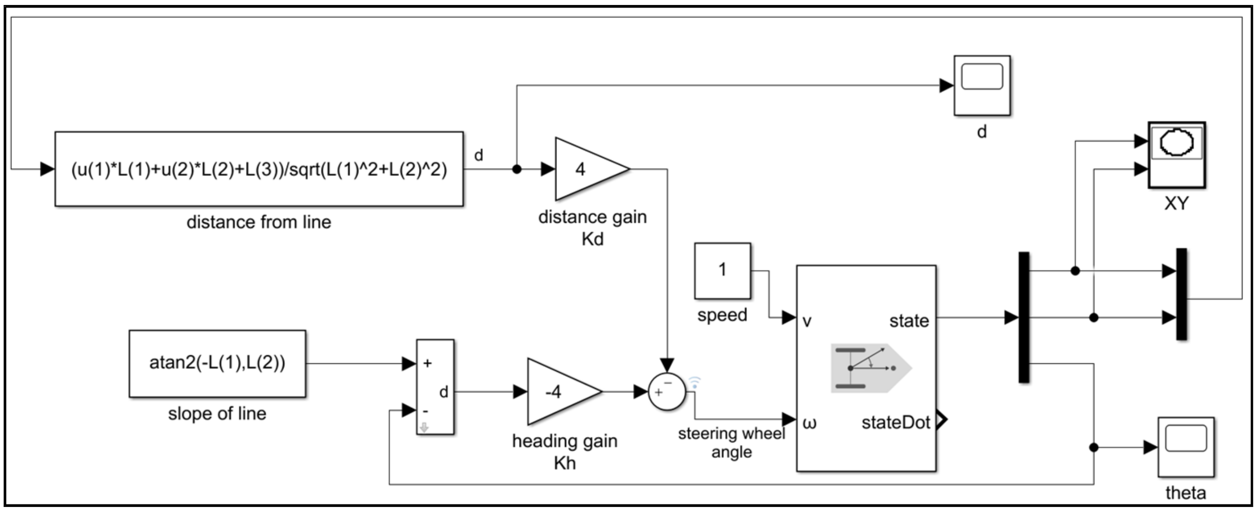
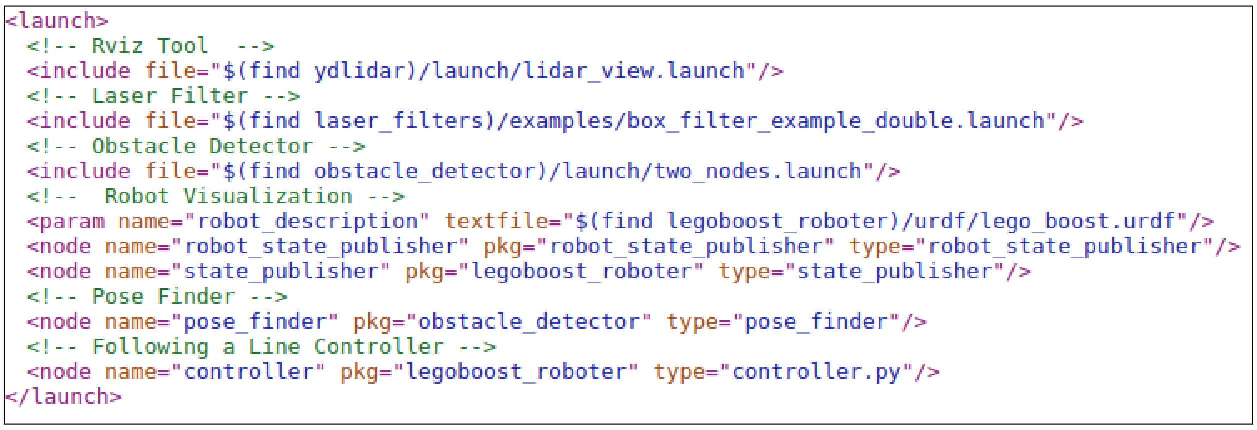


Figure 3.13: Simulation of following a line algorithm in Simulink

3.4.2 System Integration

After implementing the controller, the final step is to integrate the system components into one system. Instead of launching each node separately in the terminal, ROS provides the possibility of launching several nodes using ROS launch files. In the ROS launch file, all nodes needed for the project will be included. This is the mastery of ROS as the user can then adjust or even extend the software. The configuration of the launch file is demonstrated in figure 3.14. Launch files are basically XML configuration files.



```
<launch>
  <!-- Rviz Tool  -->
  <include file="$(find ydlidar)/launch/lidar_view.launch"/>
  <!-- Laser Filter -->
  <include file="$(find laser_filters)/examples/box_filter_example_double.launch"/>
  <!-- Obstacle Detector -->
  <include file="$(find obstacle_detector)/launch/two_nodes.launch"/>
  <!--  Robot Visualization -->
  <param name="robot_description" textfile="$(find legoboost_roboter)/urdf/lego_boost.urdf"/>
  <node name="robot_state_publisher" pkg="robot_state_publisher" type="robot_state_publisher"/>
  <node name="state_publisher" pkg="legoboost_roboter" type="state_publisher"/>
  <!-- Pose Finder -->
  <node name="pose_finder" pkg="obstacle_detector" type="pose_finder"/>
  <!-- Following a Line Controller -->
  <node name="controller" pkg="legoboost_roboter" type="controller.py"/>
</launch>
```

Figure 3.14: Launch file of the system

4 Results and Future Work

In the previous chapters, different components of the project have been discussed. After setting up the ROS system on the computer, the laser scanner was put into operation, and the robot was built. Then several localization algorithms were used and tested. Lastly, the control was implemented, and the whole system was integrated all together. As explained before, the robot must follow a pre-defined line. In this chapter, the results of the control algorithm will be discussed. Finally, improvements and future work will be discussed.

4.1 Control Results

As explained before, the robot follows a given line, which can be described with equation 3.12. In order to test the control algorithm, three different equations will be used. These equations are:

$$y + x = 0 \tag{4.1}$$

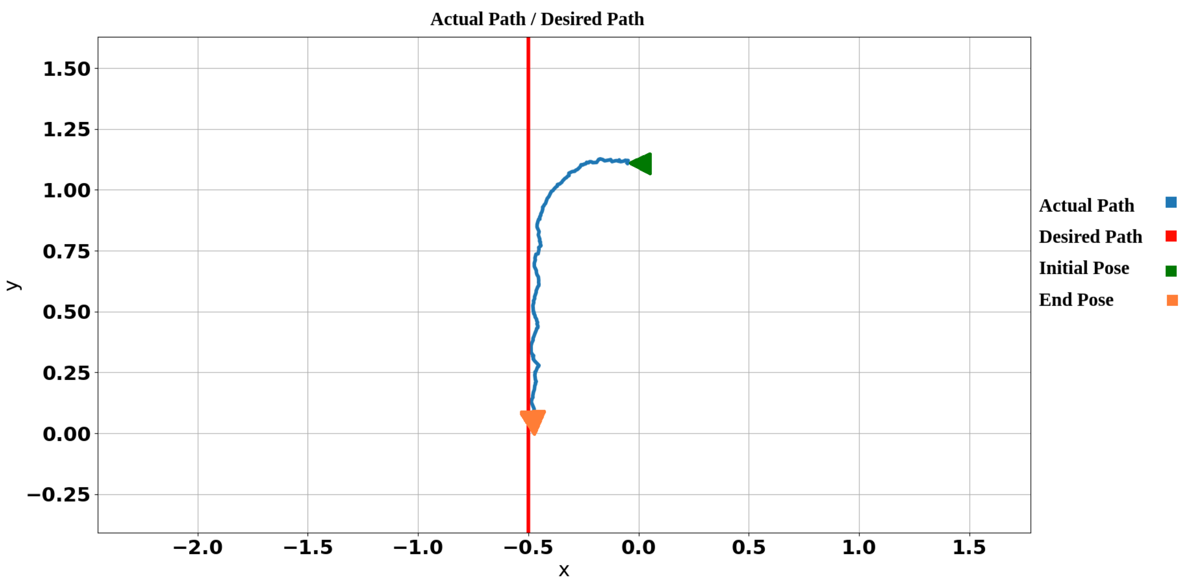
$$y - 0.5 = 0 \tag{4.2}$$

$$x + 0.5 = 0 \tag{4.3}$$

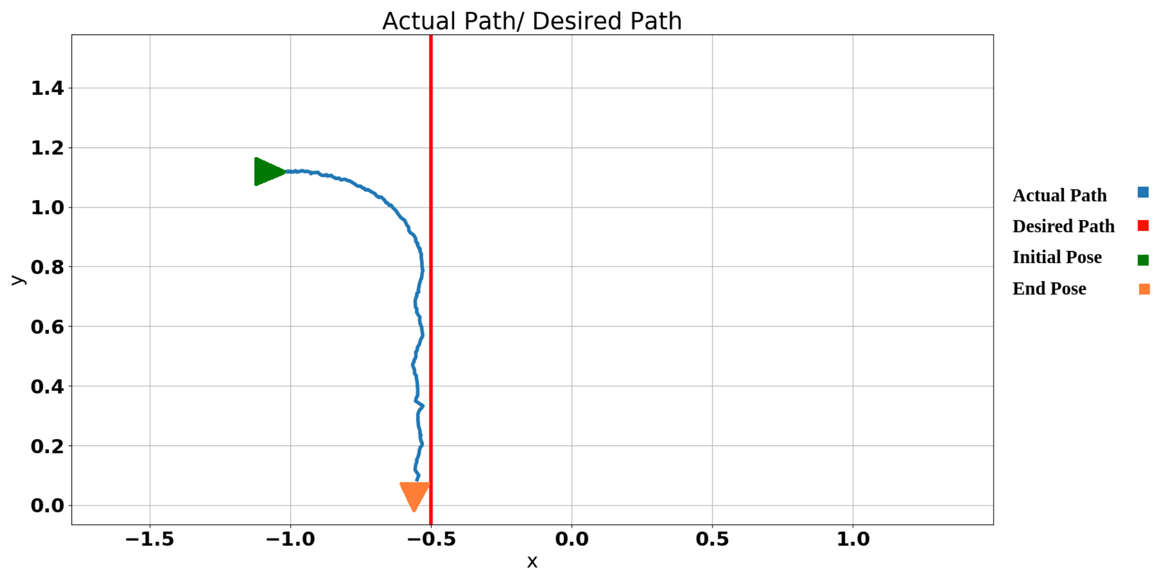
Each equation corresponds to a different line. Equation 4.3 corresponds to a straight line that is parallel to the Y-Axis. This line will be labeled as Path A. Path B is a straight line that is parallel to the X-Axis, which corresponds to equation 4.2. Lastly, Path C, which is a sloped line that corresponds to equation 4.1. For each equation, different initial poses will be tested. The poses are noted as following (x, y, θ), where x and y correspond to the position of the robot, and θ corresponds to the orientation of the robot. The different pathes are plotted in runtime using MatplotLib library.

Path A:

As displayed in figure 4.1, two different initial poses have been considered. The first initial pose is $(-0, 1.15, 180°)$. The second initial pose is $(-0, 1.15, 180°)$. In both cases, the robot successfully controlled its motion to follow the given line.



(a) First initial pose

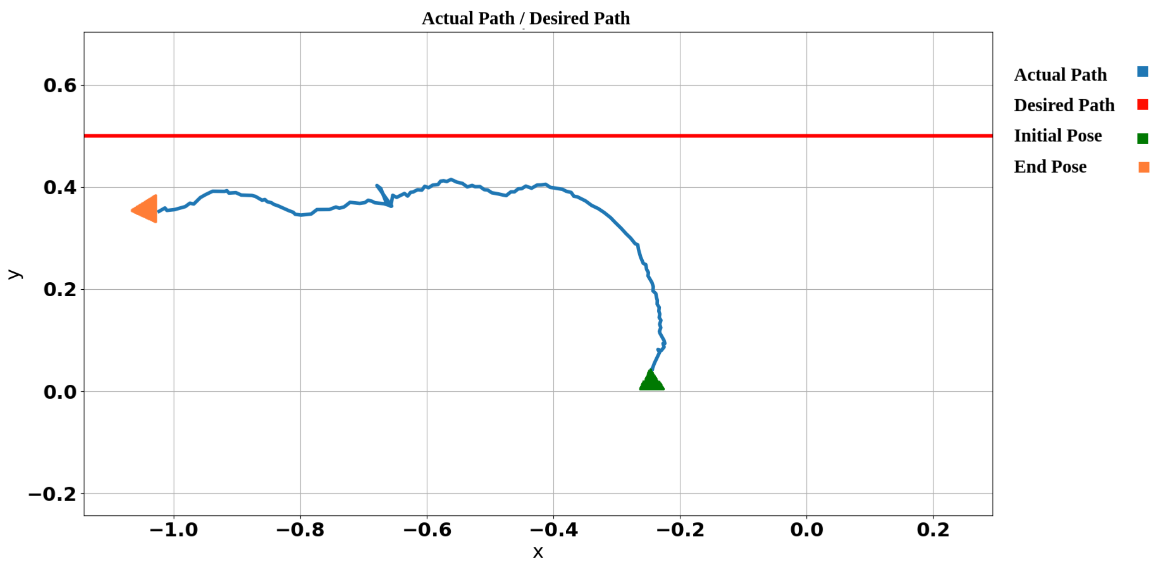


(b) Second initial pose

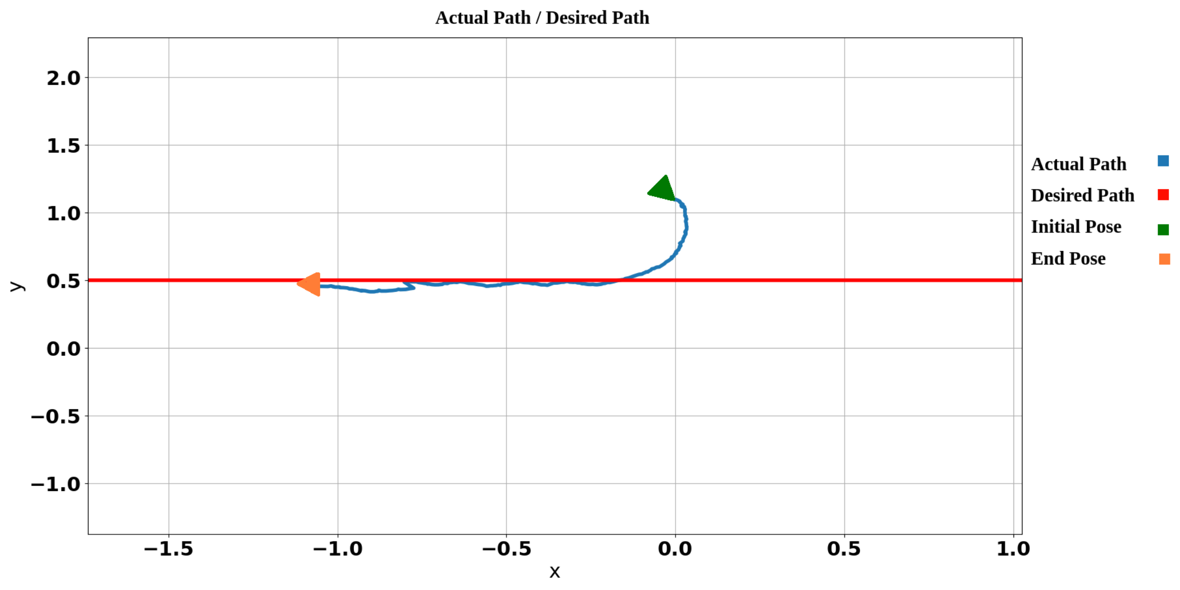
Figure 4.1: Robot motion control: Path A.

Path B:

The same as in Path A, two different initial poses have been considered. The first initial pose is $(-0.21, 0.05, 90°)$. The second initial pose is $(-0.05, 1.1, 315°)$. In both cases, the robot successfully controlled its motion to follow the given line. The motion of the robot in both cases is displayed in figure 4.2



(a) First initial pose



(b) Second initial pose

Figure 4.2: Robot motion control: Path B.

Path C:

As displayed in 4.3, the initial pose of the robot is $(-0.25, 0, 90°)$. The robot follows the given line.

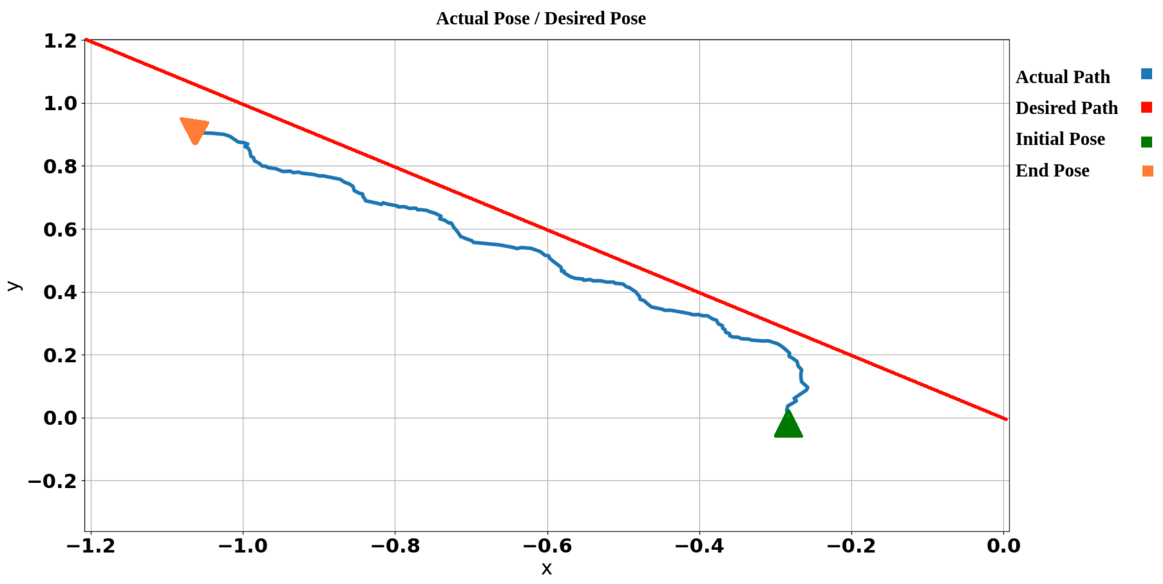


Figure 4.3: Robot motion control: Path C

As displayed in the previous figures, the robot was able to follow the pre-defined lines as expected. However, since two different proportional controllers are used, there is an offset to the given path. This offset occurs due to the inaccuracy of the proportional controller. Another problem that can be observed is the oscillations of the robot motion after the distance is minimized. This problem can be solved by adding a hysteresis.

Although the performance of the control algorithm was not ideal, this approach provides several advantages. The most important advantage is the extensibility of the software. In other words, other nodes and components can be added easily to the software to improve the results or even to add new features. Another advantage of this approach is the reusability of the nodes. The control algorithm node, for example, can be used to any other robot. The presented results show the feasibility. An optimization of the control system will improve the behaviour considerably, but unfortunately could not be tackled in the time frame of this work.

4.2 Future Work

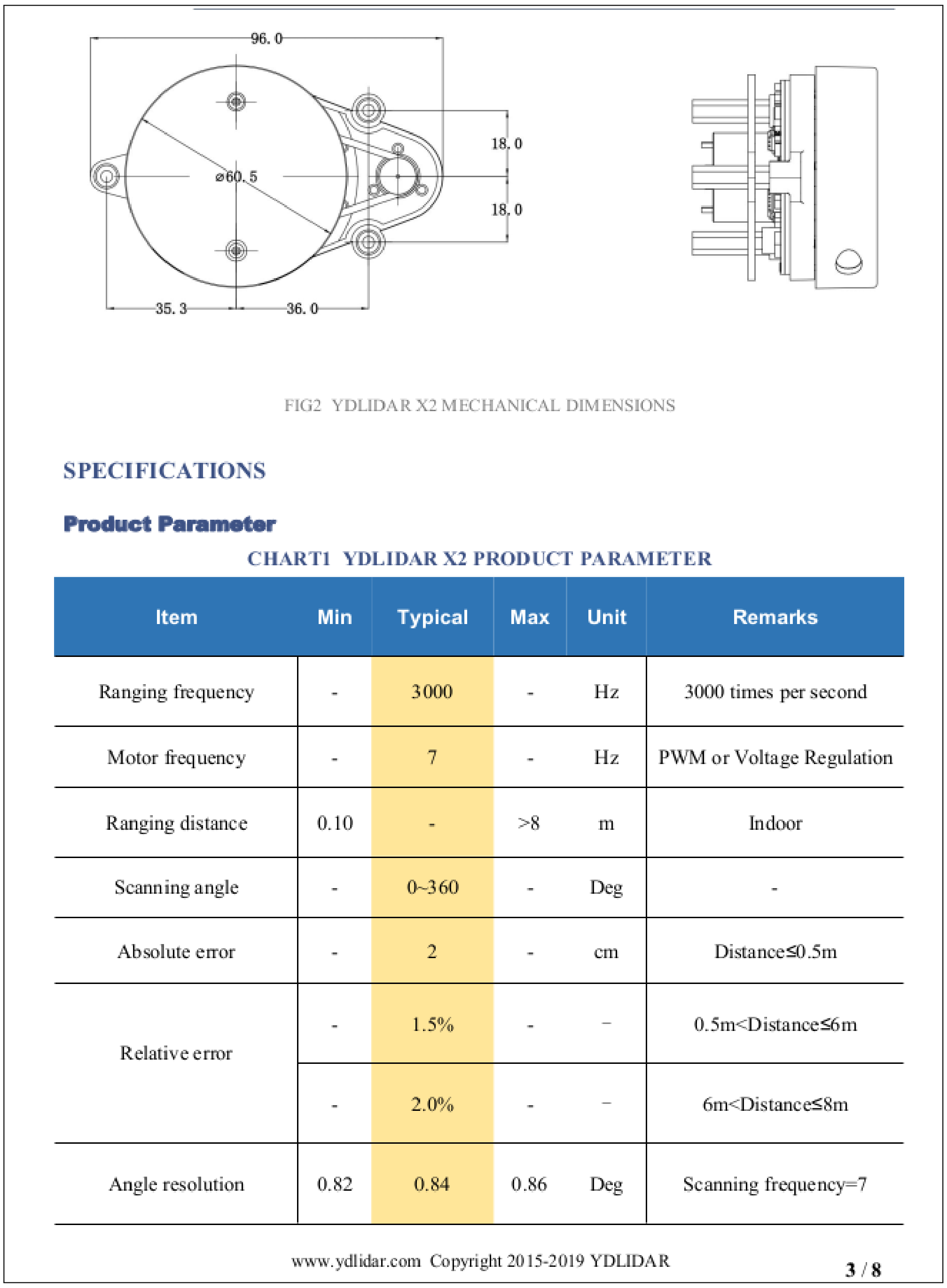
As explained before, since this project based on ROS, other components can be added and integrated into the software. Thus, many different applications and ideas can be added. The next step is to add other control algorithms, i.e., to let the robot follow a trajectory or to make it move to a specific pose. Many of these control algorithms are explained in [7]. Moreover, the whole system can be simulated using Gazebo. Gazebo is a simulation tool used in the field of robotics to simulate robots. Furthermore, the system can also be extended to contain more than one robot at the same time.

5 Summary

In conclusion, this thesis presents a new approach for mobile robot localization and navigation. This approach is different than other used approaches as it uses only a laser scanner. The laser scanner is fixed in the room. The idea behind that is to scan the surrounding area and detect the pose of the robot based on the scan information. The software used in this thesis was developed under ROS, which has different advantages as code extensibility and reusability. The project contains separate modules, namely the motion control, the localization, the laser scanner and the robot. In order to localize the robot, three different methods have been tested. After comparing the three methods, the circular object detection method has been used due to its efficiency and accuracy. Finally, a control algorithm was used to let the robot move along a pre-defined line. In order to test the controller, different pre-defined paths were given as input to the controller. Given that this was only a preliminary attempt to use this approach, it is important to improve the controller algorithm to achieve better performance. Furthermore, it is recommended to simulate the whole system using simulation tools like Gazebo, which is usually used for robotics applications developed under ROS.

Appendices

A YDLIDAR X2 Datasheet



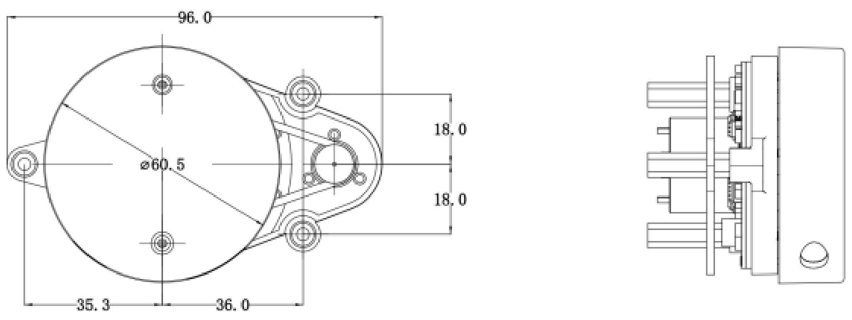


FIG2 YDLIDAR X2 MECHANICAL DIMENSIONS

SPECIFICATIONS

Product Parameter

CHART1 YDLIDAR X2 PRODUCT PARAMETER

Item	Min	Typical	Max	Unit	Remarks
Ranging frequency	-	3000	-	Hz	3000 times per second
Motor frequency	-	7	-	Hz	PWM or Voltage Regulation
Ranging distance	0.10	-	>8	m	Indoor
Scanning angle	-	0~360	-	Deg	-
Absolute error	-	2	-	cm	Distance≤0.5m
Relative error	-	1.5%	-	–	0.5m<Distance≤6m
	-	2.0%	-	–	6m<Distance≤8m
Angle resolution	0.82	0.84	0.86	Deg	Scanning frequency=7

www.ydlidar.com Copyright 2015-2019 YDLIDAR

3 / 8

Figure A.1: YDLIDAR X2 triangulation laser scanner. ©YDLIDAR All rights reserved

B Directory Structure

```
src
├── CMakeLists.txt
├── laser_filters
│   ├── CMakeLists.txt
│   ├── examples
│   │   ├── box_filter_example_double.launch
│   │   ├── box_filter_example.launch
│   │   └── box_filter.yaml
│   ├── package.xml
│   ├── src
│   │   ├── box_filter.cpp
│   │   └── laser_scan_filters.cpp
├── legoboost_roboter
│   ├── CMakeLists.txt
│   ├── include
│   │   └── legoboost_roboter
│   ├── launch
│   │   ├── load_LB_line_follower.launch
│   │   └── load_legoboost.launch
│   ├── msg
│   │   ├── CircleObstacle.msg
│   │   ├── Obstacles.msg
│   │   └── SegmentObstacle.msg
│   ├── package.xml
│   ├── src
│   │   ├── controller
│   │   ├── pose_finder
│   │   └── state_publisher
│   └── urdf
│       └── lego_boost.urdf
```

```
├── obstacle_detector
│   ├── CMakeLists.txt
│   ├── include
│   │   └── obstacle_detector
│   ├── launch
│   │   ├── demo.launch
│   │   ├── nodes.launch
│   │   └── two_nodes.launch
│   ├── msg
│   │   ├── CircleObstacle.msg
│   │   ├── Obstacles.msg
│   │   └── SegmentObstacle.msg
│   ├── package.xml
│   └── src
│       ├── obstacle_extractor.cpp
│       ├── obstacle_publisher.cpp
│       └── pose_finder
└── ydlidar_ros
    ├── CMakeLists.txt
    ├── launch
    │   ├── display.launch
    │   ├── lidar.launch
    │   ├── lidar.rviz
    │   ├── lidar_view.launch
    │   └── two_lidars.launch
    ├── package.xml
    ├── src
    │   ├── ydlidar_client.cpp
    │   └── ydlidar_node.cpp
    ├── urdf
        └── ydlidar.urdf
```

Bibliography

[1] Murat Calis. *Roboter mit ROS: Bots konstruieren und mit Open Source programmieren.* dpunkt.verlag, January 2020.

[2] Lentin Joseph. *Learning Robotics using Python - Second Edition.* June 2018.

[3] Pyo Yoonseok, Lim Darby, Cho Hancheol, and Jung Leon. *ROS Robot Programming (English).* ROBOTIS, 2017.

[4] Catkin Conceptional Overview - ROS Wiki. http://wiki.ros.org/catkin/conceptual_overview, Accesed Date: 18.06.2020.

[5] Spyros G. Tzafestas. *Introduction to Mobile Robot Control.* Elsevier, October 2013.

[6] Francisco Rubio, Francisco Valero, and Carlos Llopis-Albert. A review of mobile robots: Concepts, methods, theoretical framework, and applications. *International Journal of Advanced Robotic Systems*, March 2019.

[7] Peter Corke. *Robotics, Vision and Control: Fundamental Algorithms In MATLABÂ® Second, Completely Revised, Extended And Updated Edition.* Springer, May 2017.

[8] Mohammad O. A. Aqel, Mohammad H. Marhaban, M. Iqbal Saripan, and Napsiah Bt. Ismail. Review of visual odometry: types, approaches, challenges, and applications. *SpringerPlus*, 5(1):1897, October 2016.

[9] Adafruit Industries. Adafruit Ultimate GPS with USB. https://www.adafruit.com/product/4279, Accessed Date: 31.07.2020.

[10] SparkFun GPS-RTK2 Board - SparkFun Electronics. https://www.sparkfun.com/products/15136, Accessed Date: 31.07.2020.

[11] K.Willms. English: Triangulation, November 2012.

https://commons.wikimedia.org/wiki/File:Triangulation_englisch.jpg, Date Accessed: 01.08.2020.

[12] Laser Filters - ROS Wiki. http://wiki.ros.org/laser_filters, Accessed Date: 21.04.2020.

[13] Mateusz Przybyla. Obstacle_detector, May 2020. https://github.com/tysik/obstacle_detector, Accessed Date: 22.05.2020.

[14] sensor_msgs - LaserScan Documentation. http://docs.ros.org/melodic/api/sensor_msgs/html/msg/LaserScan.html, Accessed Date: 26.05.2020.

[15] URDF Joint - ROS Wiki. http://wiki.ros.org/urdf/XML/joint, Accessed Date: 23.05.2020.

[16] Coordinate Frames for Humanoid Robots. https://www.ros.org/reps/rep-0120.html#base-footprint, Date Accessed: 23.05.2020.

[17] Mateusz Przybyla. Detection and tracking of 2D geometric obstacles from LRF data. In *2017 11th International Workshop on Robot Motion and Control (RoMoCo)*, pages 135–141, July 2017.

[18] OpenCV: Introduction to OpenCV-Python Tutorials. https://docs.opencv.org/master/d0/de3/tutorial_py_intro.html, Date Accessed: 07.06.2020.

[19] Andrea Censi. CSM | Andrea Censi's Website. https://censi.science/software/csm/, Accessed Date: 17.06.2020.

[20] Andrea Censi. An accurate closed-form estimate of ICP's covariance. In *Proceedings 2007 IEEE International Conference on Robotics and Automation*, pages 3167–3172, Rome, Italy, April 2007. IEEE. ISSN: 1050-4729.

[21] Laser Scan Matcher - ROS Wiki. http://wiki.ros.org/laser_scan_matcher, Date Accessed: 17.06.2020.