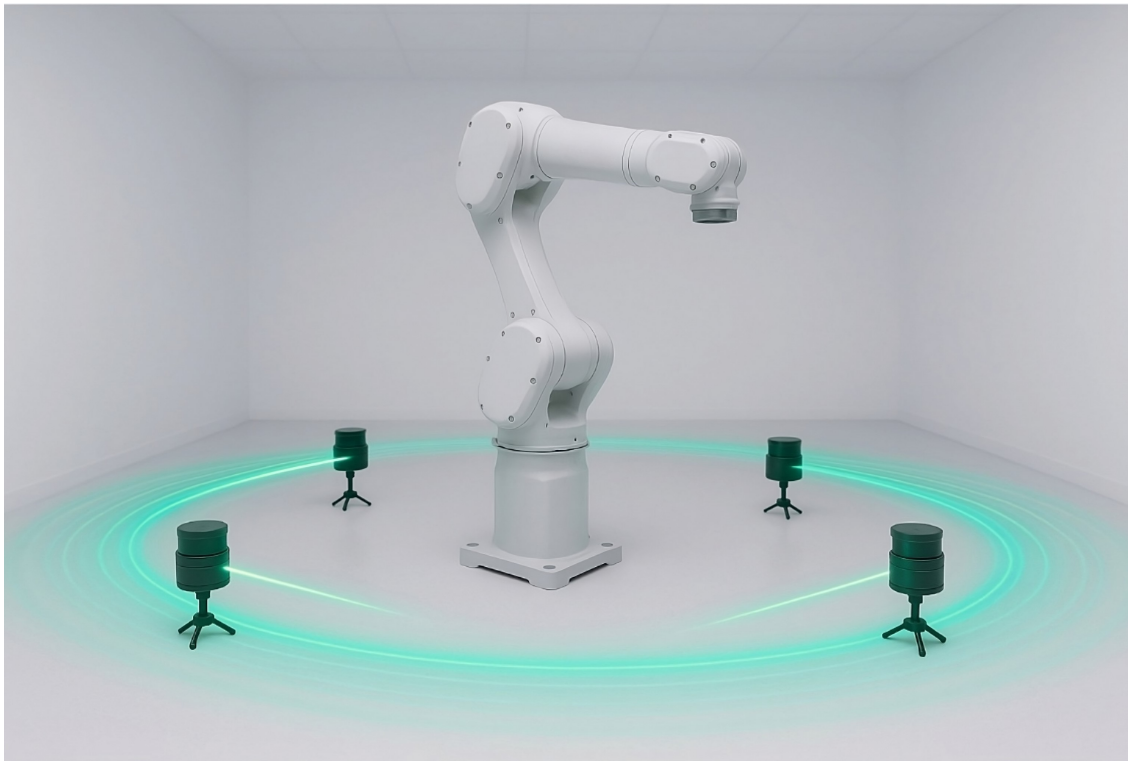

Sensor-Based Virtual Fences for Industrial Robot Safety

Sensorbaserade virtuella staket för industrirobotsäkerhet

Implementing Software-Defined Boundaries for an Industrial Robot

Bachelor's thesis in Electrical Engineering

Nour Abo Saleh, Johan Grahn, Knut Grunewald,
Hanna Petersén, Filip Segerberg

DEPARTMENT OF ELECTRICAL ENGINEERING

CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2025
www.chalmers.se

Bachelor's thesis 2025

Sensor-Based Virtual Fences for Industrial Robot Safety

Implementing Software-Defined Boundaries for an Industrial Robot

Nour Abo Saleh
Johan Grahn
Knut Grunewald
Hanna Petersén
Filip Segerberg

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Electrical Engineering
Chalmers University of Technology
Gothenburg, Sweden 2025

Sensor-Based Virtual Fences for Industrial Robot Safety
Implementing Software-Defined Boundaries for an Industrial Robot

Supervisors: Karinne Ramires-Amaro, Department of Electrical Engineering
Maximilian Diehl, Department of Electrical Engineering
Examiner: Emmanuel Dean, Department of Electrical Engineering

Bachelor's thesis 2025
Department of Electrical Engineering
Chalmers University of Technology
SE-412 96 Gothenburg
Sweden
Telephone +46 31 772 1000

Cover: Visualization of robot and sensor setup constructed in the project. (Generated with ChatGPT, 2025)

Typeset in LaTeXGothenburg, Sweden 2025

Sensor-Based Virtual Fences for Industrial Robot Safety
Implementing Software-Defined Boundaries for an Industrial Robot
Nour Abo Saleh, Johan Grahn, Knut Grunewald,
Hanna Petersén, Filip Segerberg
Department of Electrical Engineering
Chalmers University of Technology

Abstract

In industrial environments, safety has traditionally been ensured using physical barriers such as fences or enclosures to separate humans from robotic systems. While effective, these static solutions limit flexibility and make it difficult to adapt the layout to changing production needs. As factories become more dynamic and collaborative, there is a growing need for smarter, more adaptable safety systems. In this project, a virtual safety system for an industrial robot was developed using LiDAR sensors and real-time data processing. The system was designed to replace traditional physical barriers by creating three safety zones around the robot a safe zone, a warning zone and a restricted zone, depending on the distance of approaching objects. The sensors and control electronics were built and tested in real life, while the behavior and reactions of the robot were evaluated through simulation. The system continuously monitored the area at a height of 15 to 20 cm above the floor and successfully detected objects and classified them into the correct zones.The tests showed that the system detected all intrusions correctly in both warning and stop zones. The average response time was around 10 ms, which is fast enough for real-time feedback. However, the system experienced false intrusions in some cases, especially when using larger zones and more active components—up to 601 false triggers during 20 minutes recorded in one test. The results demonstrate that the system was able to trigger appropriate responses based on risk level, and show that virtual safety zones could be a viable and flexible alternative to traditional physical fences in industrial robot applications.

Keywords: Virtual fence, Industrial robot, LiDAR, SSM, Speed and Separation Monitoring, ROS 2, Safety System, Object Detection.

Acknowledgements

We would like to express our gratitude to our supervisor, Karinne Ramirez-Amaro, for her contributions throughout the project. Her feedback, suggestions and guidance has been valuable in helping us in making progress through the whole project.

In addition, we would like to thank Maximilan Diehl who helped us in the second half of the project. He made it possible for us to continue making progress in the project during the time that Karinne was unavailable. Without him we would not have been able to finish the project in time, so thank you.

Furthermore, we would also like to thank Emmanuel Dean for his guidance and knowledge concerning ROS 2. If not for his crash course in the software we would not have been able to make as much progress as we did.

AI usage

In this study AI has been used for multiple different purposes. No text has been entirely written by AI, but both ChatGPT and Copilot has been used for spellchecking and for improving grammar. Both ChatGPT and Copilot has also been used to refine and debug code.

List of Acronyms

Below is the list of acronyms that have been used throughout this thesis, listed in alphabetical order:

CSV	Comma-seperated values
DToF	Direct Time-of-Flight
FTP	File Transfer Protocol
HRC	Human-robot collaboration
LED	Light Emitting Diode
LiDAR	Light Detection and Ranging
LTS	Long Term Support
OMPL	Open Motion Planning Library
PCB	Printed Circuit Board
PLC	Programmable Logic Controller
ROS 2	Robot Operating System 2
RX	Receive
RXD	Receive Data
SDF	Simulation Description Format
SFTP	Secure File Transfer Protocol
SRDF	Semantic Robot Description Format
SSH	Secure Shell
SSM	Speed and Separation Monitoring
TX	Transmit
TXD	Transmit Data
UART	Universal Asynchronous Receiver/Transmitter
URDF	Unified Robot Description Format
QoS	Quality of Service

Contents

List of Figures

List of Tables

List of Algorithms

1

Introduction

The rapid development of automation and robotics has significantly changed the way modern industries operate. By integrating intelligent systems into production environments, manufacturers have achieved considerable improvements in speed, accuracy, and productivity [1]. Automation has become a key factor in meeting global demand, reducing costs, and increasing the overall quality of manufacturing processes. As these technologies evolve, industrial systems are shifting from rigid and fixed setups to more flexible and adaptive solutions that allow for faster reconfiguration and closer collaboration between machines and human operators.

This increasing integration of automation has also introduced new challenges, particularly in environments where humans and robots work closely. Collaborative systems must ensure that strict safety standards are upheld while supporting dynamic and efficient workflows. Traditionally, safety has been maintained using physical barriers, such as fences or enclosures, that separate people from hazardous areas [2]. Although effective, these static solutions often limit flexibility and require time-consuming changes whenever the production layout is adjusted.

To overcome these limitations, many industries are now moving toward sensor-based systems that can make robots react more flexibly and safely [2]. These systems detect when a human or another object comes too close and can slow down or stop the robot to avoid accidents. This makes it possible to work without fixed safety fences, which also saves space and makes the layout easier to change. Instead of relying only on physical barriers, modern safety solutions are more focused on smart, responsive systems that can adjust to the situation in real time.

This report presents a project that builds on this concept by exploring how a virtual safety system can be developed using cost-effective LiDAR sensors and real-time data processing. The system monitors the area around a robot, detects intrusions, and classifies them into safety zones. Based on this, suitable actions can be triggered to improve safety while maintaining operational flexibility. The system is developed and tested in a simulation environment, to evaluate its potential for use in collaborative industrial applications.

1.1 Purpose

Creating safer and more flexible environments for industrial-robot interaction requires new approaches that go beyond traditional physical barriers. This work presents a system that replaces static fencing with a virtual alternative, using LiDAR sensors to monitor the robot's surroundings and detect nearby movement.

The system defines distance-based safety zones and reacts when an object or person enters them, for example by triggering a slow-down or stop command for the robot. Through this approach, safety becomes part of the robot's environment, managed by sensor logic instead of fixed structures, allowing for a more open, efficient, and adaptable workspace. To evaluate the performance of the system, it is developed and tested in a simulation environment that replicates realistic robot behavior and sensor feedback.

1.2 Goals

The project's main goal was to develop a virtual safety fence that can monitor the area around a non-collaborative robot and detect when a person or object comes too close. The system should be able to define different levels of safety and respond by sending signals that simulate how the robot would behave, such as slowing down or stopping.

Another important part of the work was to test the system in a simulation environment to observe how it performs in different situations. The system was also designed to be modular and relatively inexpensive, so it can be reused or adapted to different robot layouts with minimal changes.

1.3 Limitations / Demarcations

To achieve the project goals, a few limitations were established to keep the work focused and the development process structured. The system was created for one specific robot setup, and all sensor placement and zone logic were adapted to that layout. Although the idea could be applied to other environments, this version was only tested in that setup.

The system was designed to detect objects near ground level, with sensors placed at a fixed height approximately 20 to 30 cm above the floor. This configuration limits the system's ability to detect objects approaching from above.

All testing was performed in a simulated environment. This allowed for controlled evaluations during development, but no physical testing was performed due to safety considerations, so real-world behavior is not confirmed in this version.

The robot used in the study is placed in a room with a smaller area than anticipated for a robot of this size. As a result it was not possible to conduct tests using the

calculated stop-zone radius. Instead, tests were done with a stop-zone radius limited to a maximum of 80 % of the calculated value.

1.4 Background

This project is based on the development of a virtual safety system for the ABB IRB 1200, a compact six-axis industrial robot commonly used in tasks such as machine maintenance, assembly, and material handling [3]. Its small footprint and flexible mounting options make it suitable for production environments where space is limited and safety is a priority. In this study, the robot is already mounted on a platform developed as part of a bachelor's thesis completed in 2024 [4]. Since the IRB 1200 is not equipped with built-in safety functions such as presence detection or automatic stopping when a person enters its working area, external safety systems such as physical fences are required to ensure safe operation around humans. All sensor placements and safety logic in this work were adapted to the geometry and working range of this specific robot model.

A Raspberry Pi 4B was used as the central microcontroller in the system, availability of multiple USB interfaces, and support for general-purpose input/output (GPIO) communication [5]. It handled the coordination between sensor input, data processing, and system output, and served as the platform for executing custom control software developed during the project. To provide visual feedback when objects entered the defined safety zones, a circuit with Light Emitting Diodes (LEDs) was implemented and connected to the GPIO ports of the Raspberry Pi 4B. This setup enabled real-time indication of detections from various directions around the robot during operation and testing.

To support development and testing, the system was built entirely in a simulation environment. Robot Operating System 2 (ROS 2) was used as the main software framework to manage communication between system components. Gazebo, a 3D simulation tool, was used to model the robot and its environment, while RViz2, a visualization tool in the ROS 2 ecosystem, was used to display sensor data, defined zones, and system behavior. These tools are referred to throughout the report and made it possible to evaluate the system during development, even without access to physical hardware. These tools are described in more detail in the theory section.

To ensure that the safety system followed established industry practices, several ISO standards were used as guidance. Although full compliance was beyond the project scope, these standards helped define key requirements for the system architecture and behavior.

ISO 12100:2010 provides a general framework for risk assessment and reduction when designing machinery [6]. It emphasizes that the most effective way to reduce risk is to integrate safety directly into the design, which supports the idea of creating a virtual safety fence. According to the standard, human risk is highest when near the machine, for example, during programming, cleaning, or maintenance scenarios,

which the system is designed to detect and respond to.

ISO 61508 focuses on functional safety in electronic and software-based systems [7]. It introduces principles such as hardware redundancy and continuous monitoring to prevent faults and failures. These ideas influenced the system structure, especially the use of multiple sensors and real-time processing to increase reliability.

For safety specifically related to industrial robots, ISO 10218-1:2011 outlines how robots must behave when people enter their work area [8]. One of these behaviors is defined as Speed and Separation Monitoring (SSM) in which the robot shall keep a predefined minimum distance to the operator att all times. If the operator gets to close to the robot then the robot should slow down or stop completely. It also states that a single failure must not be enough to disable the safety function, and that a protective stop must fully halt the robot and safely manage its power.

The project will use these standards to develop a robust safety system that integrates redundancy and reliable stop functions, ensuring compliance with industry best practices while improving operational safety around the robots. These standards are further explained in Section 4.2.9 of the Method section, where their role in sensor placement and distance calculation is described.

2

Related Works

The goal of this project is to build a virtual fence around an industrial robot, which has been done before in other research projects. The goal is also for the system to be relatively inexpensive, modular and usable for non-collaborative robots. Other systems often lack one or more of these properties. Below, some general solutions to similar problems will be described and four selected studies will be explained in a more detailed manner.

Most of these studies have been done using collaborative robots [9]–[11], which are robots that are designed to work together with humans to some extent [12]. Therefore, these robots often have built in systems to avoid collisions with humans, they are designed for lighter loads and less force, and a safety barrier is not always required. The ABB robot does not react to external forces, e.g. if it were to hit a person, and therefore it must have some type of added safety mechanism to not cause injury to humans. Some studies have used some type of camera systems [11], [13], [14] to identify moving people and objects around industrial robots. In some cases this allows for three dimensional visualization of the room. Studies often used laser scanners in combination with other safety measures (such as a 3D scanner) to detect approaching people [9], [15].

In one study, a system for a human-robot collaboration environment is described [9]. The system is developed for the *KUKA LBR iiwa* which is a collaborative robot, meaning that the robot will automatically reduce speed and power when it comes into contact with something in its environment [16]. This functionality distinguishes the LBR iiwa from the ABB IRB 1200 used in this project, as the latter does not include built in contact detection and thus pose a greater safety risk. Despite the differences, certain methods from the KUKA system are relevant. Notably, the use of two laser scanners positioned around the robot to monitor a 360 degree plane proved effective for detecting the presence of nearby individuals, particularly by identifying legs of people approaching the robot.

While the KUKA LBR iiwa is a collaborative robot designed to operate safely alongside humans, the ABB IRB 1200 is an articulated industrial robot not intended for direct human interaction. However, the use of laser scanners for monitoring the surrounding environment remains applicable to both systems. Also, within the KUKA system the laser scanners were only one of multiple safety mechanisms, the full

setup can be seen in Figure 2.1. When laser scanners are used as the sole input for safety monitoring, as in the system developed for the IRB 1200 within this project, a greater number of sensors is required to ensure reliable environmental awareness.

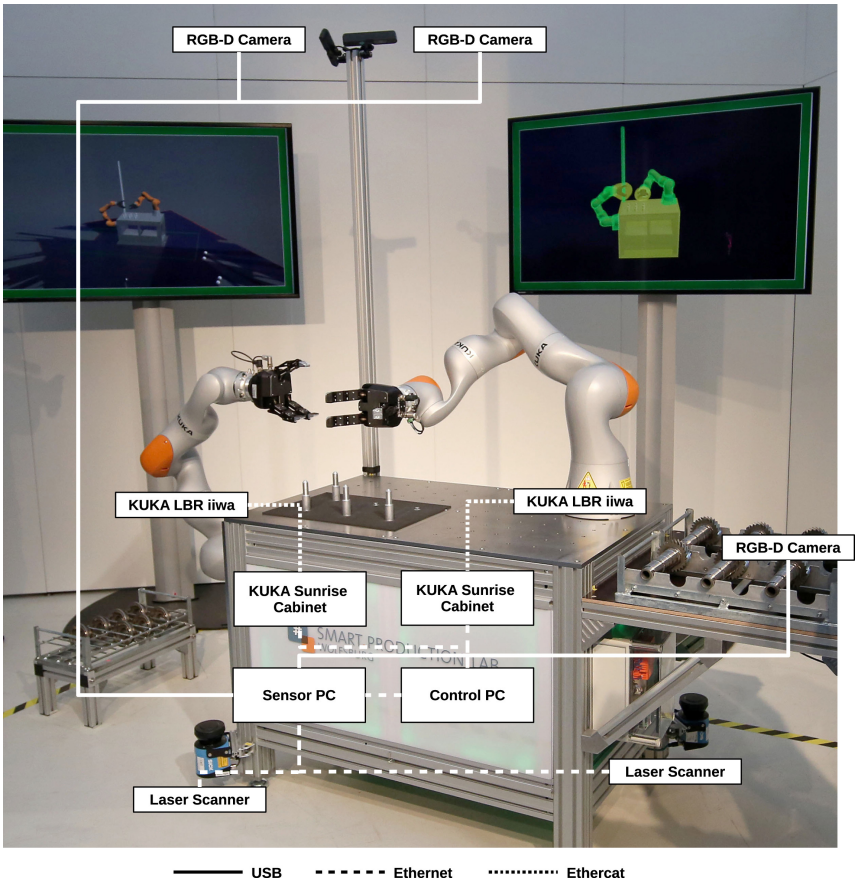

Figure 2.1: Hardware setup of the full safety system around the KUKA LBR iiwa. [9]. Used with permission from Elsevier.

One study compared the times for a set task to be completed in different conditions [17]. They tested the task with a fence around the robot, which is most common today. They tested with two different zones that are monitored by a laser scanner. When someone enter the first zone the robot slows down, and then stops in the next zone. They also tested the task with two different types of SSM where the current position of the robot is taken into account. The study found that the time it took for the robot to complete the task, with some interruptions by someone approaching the robot, decreased both when zones and SSM were used. This indicates that using safety measures other than a physical fence around the robot makes the robot's work more time-efficient. However, monitoring the position of the robot is a more complicated approach to this problem which could be difficult to implement as a modular and inexpensive system. The sensor used in this project costs around 80,000 SEK which is not feasible cost in this project [18]. Although the same methods cannot be used in this project, it shows that the zone-based approach remains time-efficient.

The same or similar problems have also been solved with other methods. One study used an overhead safety camera system which identifies moving objects in different zones [13]. These zones are dynamic, meaning that they follow the robot arm and therefore differ over time depending on where the robot arm is working in each moment. A visualization of these dynamic zones can be seen in Figure 2.2. The robot then receives signals to either slow down or stop completely. The dynamic zones are useful to minimize the time that the robot is still. The study found that this approach allowed for more human-robot collaboration which in turn reduced the time significantly for the tested task, a rear axle assembly. The system increases the time effectiveness but some of the components used are more expensive than reasonable in this project and therefore this is not a good approach to the problem for this project.

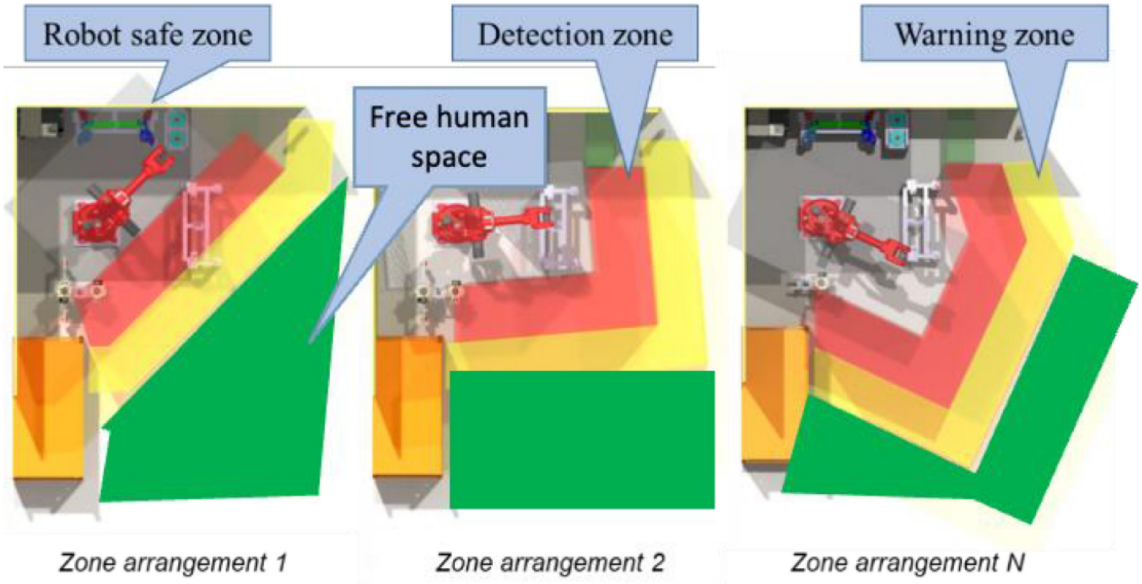

Figure 2.2: Visualization of dynamic zones corresponding to three different positions. [13], CC BY 4.0.

Another study also solves a similar problem [19]. They used multiple sensors mounted around a robot arm to detect if something gets close to the arm, with the goal to be able to detect humans without contact. The stop zone is relative to the robot arm as seen in Figure 2.3 and in some cases the robot did not stop moving until an object was within 200 mm of the robot arm. This system is placed on an *UR10* robot, which is a collaborative robot in contrast to the ABB IRB 1200 [10]. This entails that there is always an internal fallback safety measure if the external safety system fails. With the ABB-robot the safety is entirely dependent on the external safety measures. The ABB robot is also able to move faster than the UR10, at some axes the difference is more than 400° per second [10], [20]. These differences may indicate that humans are able to work closer to the UR10 robot than the ABB robot. Therefore, the safety zone should be bigger around the ABB robot than the 200 mm used in this study.

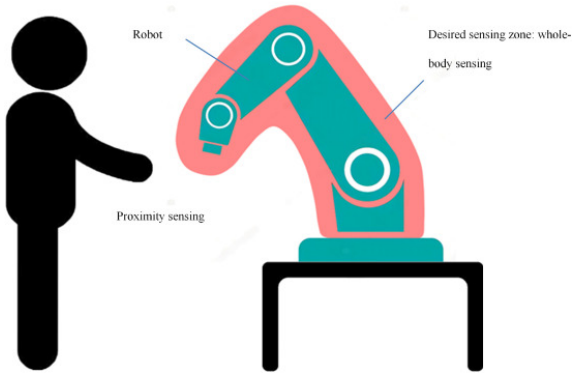

Figure 2.3: Representation of the stop zone around the UR10 robot arm. [19]. Used with permission from Elsevier.

3

Theory

This section introduces the reader to the necessary concepts and theory for this study. The chapter is divided into ten parts which offers greater insight into key concepts and components used in the project.

3.1 Industrial Robot

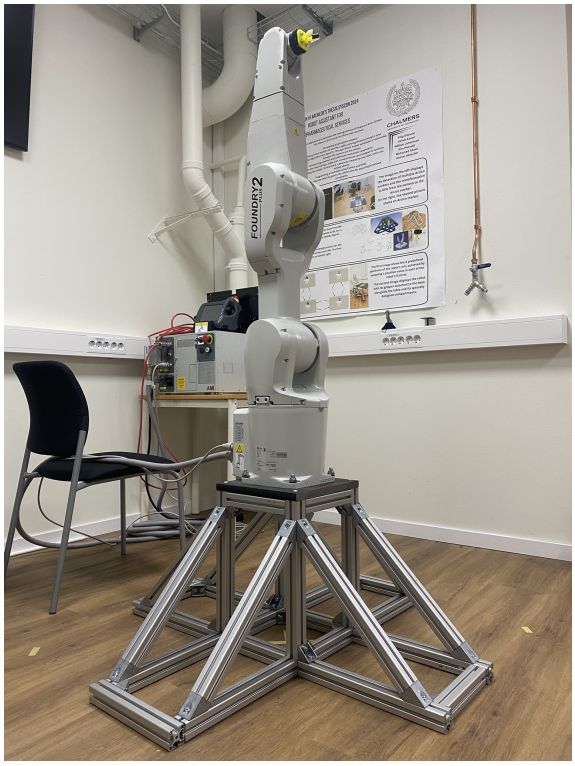

Figure 3.1: The ABB IRB 1200 robot.

The ABB IRB 1200 is a compact and versatile six-axis industrial robot that is commonly used in assembly, machine maintenance, and material handling applications [3]. Its compact footprint and high precision makes it suitable for environments where space is limited and fast, repeatable motion is required. The robot offers a reach of up to 900 mm and a payload capacity of up to 7 kg, depending on the

configuration. It supports multiple mounting options and can be integrated into horizontal and vertical production layouts. The robot is displayed in Figure 3.1.

The IRB 1200 is typically controlled via ABB's IRC5 or OmniCore controller, which enables advanced motion programming and supports external safety and communication interfaces [3]. This allows the robot to be connected to external devices such as sensors, emergency stop circuits, and programmable logic controllers (PLCs). Safety functions can be triggered through digital inputs, and communication can be handled through industrial protocols such as Ethernet/IP or PROFINET, depending on the setup. These integration options make it possible to build systems where the robot reacts to sensor input in real time.

3.2 Raspberry Pi 4B

The Raspberry Pi 4B is a small microcomputer designed for embedded applications, optimized for size, power consumption, and flexibility [5] . The MC can be seen in Figure 3.2. It includes a 64-bit quad-core processor, up to 8 GB of RAM, USB 3.0 ports, Bluetooth 5.0, dual-band Wi-Fi, and gigabit Ethernet. It also features General Purpose Input/Output (GPIO) pins for controlling external components such as LEDs or sensors. Compared to earlier models, the 4B offers higher performance while maintaining the same low power requirements and compatibility with existing Raspberry Pi software.

Figure 3.2: The Raspberry Pi 4B.

A Raspberry Pi can be used to handle data from sensors, apply logic for zone detection, and send signals to LEDs based on security threat level. Its combination of computing power, connectivity, and real-time GPIO control made it suitable for running the safety system without additional hardware.

3.3 LiDAR Sensors

A 2D LiDAR sensor functions by sending out short laser pulses and measuring the time taken before they bounce back from nearby objects [21]. The sensor switches the laser on and off very quickly, and the reflected pulses are picked up by an optical receiver. Since the speed of light is known, the distance can be calculated based on the time between when the pulse was sent and when it was received. The sensor rotates and scans a flat surface around itself, which gives a 360-degree view close to the ground. One sensor is displayed in Figure 3.3

Figure 3.3: The LD06 LiDAR Sensor.

The LiDAR sensors used in this project are the LD06 LiDAR. These sensor sends out data packets continuously while it scans [22]. Each data packet includes 12 angles and 12 distance, which tells how far away an object is and in which direction. This data can be read through a serial connection and is updated multiple times per second. The system will use this information to decide which safety zone the object is in. The sensors are the foundation of making it possible to create a virtual safety fence where the robot can react if something gets too close.

3.4 KiCad

KiCad is an open-source software suite for electronic design automation (EDA), used to design and document electronic circuits. In this project, KiCad was used to develop the schematic and printed circuit board (PCB) layout for the LED circuit that provided visual feedback in the safety system. The software enabled precise placement of components, such as resistors and transistors, and ensured correct routing of signals between the Raspberry Pi and the LEDs. By using KiCad, it was possible to generate manufacturing files including Gerber files and a bill of

materials, which were necessary for producing a functional and reliable PCB version of the LED driver circuit.

3.5 Robot Operating System 2

Robot operating system 2 (ROS 2) is an open source framework that helps users quickly build components for actuators, sensors and control system [23]. The components are then easily connected using built-in ROS 2 tools that enables quick testing, training and quality assurance. These built in communication tools makes sending messages to a variety of visualization tool easy, offering quick and intuitive visualization of data. ROS 2 also support working with simulated robots instead of real robots making testing easier and faster. This in combination with that ROS 2 supports hardware interfaces for a broad variety of robots components have made ROS 2 an globally used tool for robot development.

3.5.1 ROS Distribution

The latest distribution for ROS is Jazzy jalisco and was released May 23, 2024 and is a group of ROS packages configured to work reliably together [24]. Once a distribution is released, changes and updates are limited in order to maintain a stable code base until the next distribution.

3.5.2 Workspaces

A workspace is a directory containing ROS 2 packages, and a project consists of workspaces on different levels (called underlays and overlays) [25]. This structure makes it possible to create new packages in the overlay without interfering with the existing packages in the underlay.

3.5.3 Nodes and communication

ROS 2 functions primarily using nodes, where each node works independently on a specific task [26]. Nodes can communicate with each other using topics, services, actions, or parameters which together creates a ROS 2 network. The topics act as buses for sending and receiving messages, to which different nodes can either subscribe to (read data) or publish to (send data). It should be noted that communication to and from a topic can be any combination from one-to-one communication to many-to-many communication, furthermore displaying the power of sending and receiving messages from topics.

Unlike topics, the services do not work with open data streams, but rather on a call-and-response basis [26]. A service client located in a node sends a request to which the service server (also located in a node) returns a response. This server can be configured to run in a single or multi-threaded environment, allowing for parallel request handling if necessary. Typical usages of services include triggering actions or fetching current statuses.

Actions build on topics and services for node to node communication [26]. They are typically intended for long-running tasks, and consist of goal, feedback, and result stages. The action client node sends the goal request to the action server and the action server sends a goal response and starts executing the goal. During the time the action server works on the request from the action client it publishes feedback on a topic that action client subscribes to. When the action server completes the goal, the action client sends a result request to the action server. The action server then sends a response and the process is complete.

3.5.4 Launch files

A launch file in ROS 2 is a Python script that automates the launching process of nodes and their parameters, configurations and other runtime settings [27].

3.5.5 Transforms

In ROS 2, tf2 (also called transforms) is a library responsible for keeping track of coordinate frames [28]. The coordinate frames are then used for tracking the positions and orientation of modules within a virtual environment. All coordinate frames are defined with relation to a parent coordinate frame, with the world coordinate frame being the center referees to all other frames. This creates a tree structure that ensures that all frames are consistently related and positioned in visualization or simulation.

3.5.6 Joint Limits

The Joint Limits configuration file specifies physical constraints for each joint of the robot, such as maximum velocity, maximum acceleration, and range of motion [29]. For the ABB IRB 1200, this includes all six rotational joints of the articulated arm. These limits are essential for ensuring that planned motion stays within the robot's mechanical capabilities.

3.5.7 Controllers

A controller is a software component that implements control theory to automatically adjust a system's input to achieve a desired output [30]. Open-loop control operates on fixed input and therefore does not require sensors or feedback mechanisms, but is also less accurate when dealing with disturbances [31]. Closed-loop control implements feedback to adjust control signals based on the error, which is the difference between actual and desired output. PD (Proportional-Derivative) and PID (Proportional-Integral-Derivative) controllers are common in closed-loop systems and are responsible for driving the error towards zero [32]. PD improves response and reduces overshoot by adjusting the control signal based on the current error and its rate of change. PID adds an integral term to eliminate steady-state error (difference between the desired output and the actual output of a control system once it has stabilized over time) making it more robust for systems needing long-term accuracy. For example, PD control can be used for altitude stabilization

(pitch, roll, yaw) in drones as it provides fast and smooth responses without too much overshoot. PID control could be used for balancing systems (e.g. Segways or inverted pendulum) due to its fast response and minimal steady-state error. PID controllers are typically the most widely used type of controllers in industry.

In ROS 2, controllers are designed to separate control logic from the hardware, making the system more modular [33]. Our system uses a ”JointTrajectoryController” from the ros2_controllers package, which is meant to execute a trajectory for one or more joints. This controller is defined in irb1200_controllers.yaml from the irb1200_ros2_gazebo package and is named ”irb1200_controller”. It receives a trajectory generated by MoveIt2 through the "FollowJointTrajectory" action interface as a control_msgs/action/FollowJointTrajectory message (a list of joint positions/velocities/accelerations/timestamps). Each joint state in this trajectory is used as the next reference point (input), and the difference between the current state (feedback from /joint_state_broadcaster that continuously publishes the current joint state) and the current reference point is the error. The controller aims to minimize this error, or in other words, it aims to bring the current state as close as possible to the desired state. If the ”AddTimeParametrization” response adapter is used in the motion planning pipeline, then time-based vectors such as velocity and acceleration will be included in the states. In this case, the controller will generally run a closed loop with PD or PID control to achieve the desired profile. Without the response adapter, only position data is considered and the controller can simply run an open-loop with feedforward control, where input is anticipated.

The controller manager node is responsible for the connection between the controllers and the hardware-abstractions in the ros2_control framework [33]. It manages the controller's interface requirements and has access to the hardware components provided interfaces. The control manager uses this to connect the controllers needs with the hardware capabilities. It also reports errors if multiple controllers want access to the same hardware, avoiding access conflicts. It is executed in a control loop managed by the an update method that reads data from hardware components and outputs it to all active controllers.

3.6 Simulation and Visualization

RViz2 is a 3D visualization tool for the ROS 2 framework which, among other things, allows users to visualize sensor data and robot models. While RViz2 focuses on visualization, Gazebo Harmonic complements it by offering a dynamic simulation environment [34] [35]. Gazebo incorporates physics, lighting, and sensor feedback. This allows developers to validate robot behavior without physical hardware. It also integrates easily with the ROS 2 framework, providing synchronized testing between simulated environments (robot) and real-time systems (sensor data).

3.7 Motion Planning

Motion planners are a fundamental component in robotics to enable robotic movement from start positions to goal positions while taking consideration to physical restraint and kinematics. This section introduces the tools and frameworks used for motion planning and their integration with MoveIt2.

3.7.1 MoveIt2

Moveit2 is a framework that is well supported for ROS 2 [36]. It provides motion planning, manipulation, 3D perception and kinematics for both simulated and real world systems. The interface is through a ROS 2 action or service that can be configured through the move_group node. To plan a trajectory, MoveIt2 uses one of the defined motion planning libraries such as Open Motion Planning Library (3.7.2), passing the current and goal joint states as well as any constraints such as joint limits or obstacles as input. OMPL will then generate random samples in joint space and connect them using a motion planning algorithm, such as RRTConnect (Rapidly-exploring Random Tree Connect) [37]. MoveIt2 uses a ”planning scene” to check whether a state is valid. The planning scene is a snapshot of the current state of the simulated environment and is updated through the planning_scene_monitor node. This node subscribes to /joint_states for the robot state, /tf for transforms and /planning_scene or /collision_object for world updates. The planner will stop when a valid path is found or if it runs out of time. The output, a discrete sequence of joint states (of the ROS 2 format trajectory_msgs/msg/JointTrajectory), can be sent to a controller to move the robot. It is also important to note that motion plans used by MoveIt2 are typically pre- and/or post-processed using modular plugins called adapters.

3.7.2 OMPL

An OMPL planner is a set of motion planning algorithms provided by the MoveIt2 integrated Open Motion Planning Library [38]. The OMPL Planner itself is not a single motion planning algorithm, but rather a plugin containing a collection of many different motion planning algorithms. In this project, the IRB 1200 library provides a selection of configurations for some of these algorithms.

3.7.3 Kinematics Configuration

To enable the robot to move with MoveIt2, a kinematics file is used [39]. These files configures how MoveIt2 calculates the robots joints movements to reaching the desired positions or orientation.

- kinematics_solver: Defines which plugin MoveIt2 should use to calculate joint angles.
- kinematics_solver_search_resolution: Sets how precisely the solver searches

for solutions. Lower values result in more accurate results but increase computation time.

- kinematics_solver_timeout: Specifies the maximum time the solver is allowed to spend finding a valid solution.

3.7.4 Adapters

There are two types of adapters- request adapters which apply pre-processing techniques- and response adapters that apply post-processing techniques to a motion plan [40]. Request adapters can modify or validate a plan before it reaches the planner. Some common examples include "FixWorkSpaceBounds", which ensures that the goal pose is within the workspace bounds defined by the planner configuration (this prevents invalid goals that are physically unreachable), and "FixStartStateBounds" for correcting small violations on the robot's joint limits in its start state. For the response adapters, "AddTimeParametrization" is typically used for enabling velocity and acceleration to the trajectory, allowing it to be interpreted by controllers and real-world robots.

3.7.5 Abstract Time ROS 2

Running logged data against a simulated robot requires the use of abstract time to ensure accurate synchronization between components, as the simulation itself becomes a limiting factor in timing [41]. To address this a Gazebo clock bridge can be implemented. The clock bridge publishes the system time allowing the entire system to operate based on the simulation time.

3.7.6 Unified Robot Description Format

URDF or Unified Robot Description Format is a commonly used XML file format in ROS 2 for both the description of the robot model and the collision boundaries [42]. A URDF file consists of joints and links, where joints represent physical parts of the robot and links connect two joints. Some common types of joints are revolute (rotates around an axis), prismatic (slides along an axis), fixed (no motion) or continuous (rotates without limit). The file can also contain so called "transmissions", used in combination with controllers, that map actuator effort to joint movements for simulation/control. Colors and textures can also be defined as "materials".

3.7.7 Semantic Robot Description Format

SRDF or Semantic Robot Description Format is also an XML file format that complements URDF by specifying joint groups, additional collision information, default robot configurations, and additional transforms needed for defining a robot's position [42]. The URDF file only defines the physical joints on the robot, whereas the SRDF adds virtual joints to the visualization and simulation world's coordinate system. The SRDF also contains the joint groups used by MoveIt2's motion planning, such as the "arm" group or "gripper" group.

3.8 Programming Languages

Programming languages are used to instruct computers to perform specific tasks [43]. They serve as the primary medium through which humans communicate with machines, translating algorithms and tasks into executable actions. At their core, programming languages are governed by syntax, structure, and semantics, meaning, enabling precise manipulation of data and execution of logic. These languages vary in abstraction, from low-level machine-oriented languages to high-level, human-readable ones like Python or Java. In the context of safety-critical systems, such as industrial automation or robotics, the choice of programming language influences not only the system's functionality and efficiency but also its reliability, maintainability, and safety compliance.

3.8.1 Python

Python is a high-level programming language with a simple and easy-to-learn syntax, making it well suited for rapid prototyping [44]. Its straightforward structure also helps reduce the difficulty and cost of maintaining programs. Unlike compiled languages, Python is interpreted and runs code line by line. When the interpreter encounters an error, it raises an exception. If the program does not handle the exception, the interpreter prints a stack trace and then exits the program.

Python is commonly used to build websites, handle data, and develop software efficiently [45]. It integrates well with databases, supports complex mathematical operations, and simplifies the automation of repetitive tasks. Thanks to its ease of testing and modification, developers can quickly experiment with ideas and iteratively improve their programs. This adaptability is a major reason why Python is applied in a wide range of fields.

3.8.2 C++

C++ is a powerful, general-purpose programming language that was developed as an extension of the C programming language [46]. It combines the performance and efficiency of low-level programming with the flexibility and structure of high-level, object-oriented programming.

C++ builds on the foundations of C but introduces features such as classes, objects, inheritance, and polymorphism - key concepts within object oriented programming [47]. These features allow developers to create complex programs that are modular, maintainable, and easier to extend. At the same time, C++ retains close control over hardware through manual memory management, which are critical for applications that demand high efficiency and precise resource handling.

One of the key advantages of C++ is its execution speed [46]. Unlike interpreted languages, C++ code is compiled directly into machine code, allowing it to run extremely fast and with minimal overhead. This makes it especially suitable for developing real-time applications, operating systems, robotics and safety systems

where performance is non-negotiable.

4

Methods

This chapter presents the methodology used to develop the safety monitoring system. The project was divided into four key areas: Hardware, Software, Visualization, and Simulation. Hardware covers the physical setup, including LiDAR sensors and the Raspberry Pi. Software describes how data was collected and processed in real time. Visualization involves tools for displaying sensor data, and Simulation was used to test the system in virtual scenarios. Each section outlines the key steps and decisions made during development.

4.1 Hardware

This section outlines the hardware components used to implement the safety monitoring system. The core of the setup consisted of four LiDAR sensors. These sensors were mounted at custom made stands and placed around the industrial robot to ensure full 360-degree coverage of the surrounding area. Each LiDAR unit was connected to a central processing unit via an USB-bridge to a Raspberry Pi 4B, which served as the main processing unit for the system. The Raspberry Pi was chosen due to its multiple USB ports, GPIO capabilities, and support for Python and native ROS 2. Additional hardware included LED indicators for visual feedback, a physical emergency stop switch, and custom mounts to securely position each sensor at the correct angle. The design emphasized both robustness and modularity to facilitate easy testing, repositioning, and maintenance during the development process.

4.1.1 Sensors - LD06 LiDAR

A core requirement of the project was the reliable detection of a person in close proximity to the industrial robot, making sensor quality, consistency, and redundancy critical for ensuring system safety. To achieve complete 360-degree coverage around the robot, it was necessary to deploy multiple strategically positioned LiDAR sensors to eliminate blind spots and provide continuous monitoring from all directions.

The sensors chosen for the project were four of the LD06 LiDAR sensor primarily due to its high reliability of 12 meters measuring distance and 360.00 degrees of coverage [22]. The LD06 uses Direct Time-of-Flight (DTOF) technology, enabling

it to perform up to 4,500 distance measurements per second, ensuring accurate and real-time spatial awareness necessary for defining responsive virtual boundaries around industrial robots. It emits infrared laser pulses that travel toward nearby objects, reflect off their surfaces, and are then received back by the sensor. The sensor then calculates the distance to obstacles based on the time delay between emission and reception. The LD06 is powered using a 5 V DC supply.

For data communication, the sensor transmits measurement to a Raspberry Pi via UART, a serial communication interface, at a 230,400 baud rate [22]. Each measurement package starts with a 0x54 header byte, followed by metadata, such as package length, start angle, and stop angle, which then is followed by 12 measurement points, each encoded in three bytes: two bytes for distance: LSB - Least Significant Byte, and MSB - Most Significant Byte [48]. One byte for confidence was also included. A representation of the packet structure can be found in Table 4.1 and the structure for one measurement point can be found in Table 4.2.

Table 4.1: Sensor data packet structure.

Field	**Content**
Start character	54H
Data length	2CH
Radar speed	LSB, MSB
Start angle	LSB, MSB
Data (12 measurement points)	Table 4.2
End angle	LSB, MSB
Timestamp	LSB, MSB
CRC Check	1 Byte

Table 4.2: Structure of one sensor distance measurement.

Component	**Description**
Distance (LSB)	Lower byte of distance value
Distance (MSB)	Upper byte of distance value
Confidence	1 Byte (0–255) confidence value

To interpret the distance, the two bytes are combined: LSB and MSB must be shifted and added accordingly to reconstruct the full 16-bit distance value in millimeters. This bit-wise handling is critical for accurate distance computation and often needs to be done manually when parsing the raw serial data. Additionally, each data frame includes a CRC-8 checksum to ensure communication integrity [22]. The angular resolution is achieved by linearly interpolating between the start and end angles provided in each frame, enabling precise 360.00 degree environmental mapping.

Each distance measurement from the sensor is accompanied by a confidence value that indicates the reliability or quality of the reading [22]. This value reflects the intensity of the reflected infrared signal - how strong the sensor's return signal is after the laser pulse bounces off an object. A higher confidence value means that the reflection is strong and the measured distance is likely accurate, while a lower confidence suggests a weak signal, which could be due to factors like long distance, dark or absorbent surfaces, or oblique angles. For example, white objects within 6 meters typically yield a confidence value of around 200, which is considered strong. These confidence data would be used to filter out unreliable readings and improve the robustness of obstacle detection in the virtual safety fence system.

4.1.2 Microcomputer (MC) - Raspberry Pi 4B

The Raspberry Pi 4B served as the central processing unit in the project, offering a compact and powerful platform capable of running a full operating system and handling external connections, such as buttons, sensors and LEDs [5]. In this project, the Raspberry Pi was configured to run Ubuntu 24.04 LTS, a lightweight Linux distribution that supports native ROS 2 [49].

The system was operated remotely via SSH (Secure Shell) using PuTTY, a Windows-based terminal emulator [50]. This allowed for full command-line access to the Raspberry Pi from a separate development computer, enabling headless operation (i.e., without a monitor or keyboard connected to the Pi). File management and code deployment were handled using FileZilla, a FTP-client (File Transfer Protocol), also running via SSH, providing a graphical interface to navigate the Raspberry Pi's file system, upload and download files, and manage project directories [51]. This setup allowed for a smooth development workflow and efficient remote debugging and updating of the system throughout the project.

4.1.3 Communication Between Sensors and MC - CP2102 UART Bridge

To power and communicate with the four LD06 sensors, each sensor was connected to a CP2102 USB to UART bridge module. This bridge provides a simple and reliable way to establish serial communication with the Raspberry Pi while also providing the necessary power to the sensors. Each CP2102 module includes a 5 V output pin and a GND pin, which were used to power the sensors directly. For data transmission, the RX (receive) pin on the CP2102 was connected to the sensor's TX (transmit) pin, allowing the Raspberry Pi to receive measurement data from each LiDAR unit.

Figure 4.1: CP2102 UART USB bridge.

4.1.4 LEDs and circuitry

In the system, Light Emitting Diodes (LEDs) were implemented as visual warning indicators to signal when an object entered one of the defined safety zones. Each LiDAR sensor was connected to an individual LED, allowing the system to show detection events from different directions around the robot during operation and testing.

The LEDs were controlled by the Raspberry Pi's GPIO pins connected to a transistor-based circuit. In each LED circuit, two NPN transistors (a Negative-Positive-Negative layer structure) were used. The first transistor amplified the signal from the Raspberry Pi to provide enough control current, while the second transistor handled the switching of the LED. The LEDs were powered using the 3.3 V supply from the Raspberry Pi's power pin, while the GPIO pins acted only as control inputs to the transistors. This setup reduced the electrical load on the GPIO pins and ensured stable and safe operation. Current-limiting resistors were also a part of the circuit to prevent overcurrent and protect the components and the Raspberry Pi. The circuit was first tested on a breadboard to verify correct functionality.

After successful testing, the circuit was transferred to a PCB designed in KiCad. The schematic and layout were created based on the breadboard version, and the final design was exported as Gerber files for manufacturing. The schematic of the LED circuit can be seen in Figure 4.2

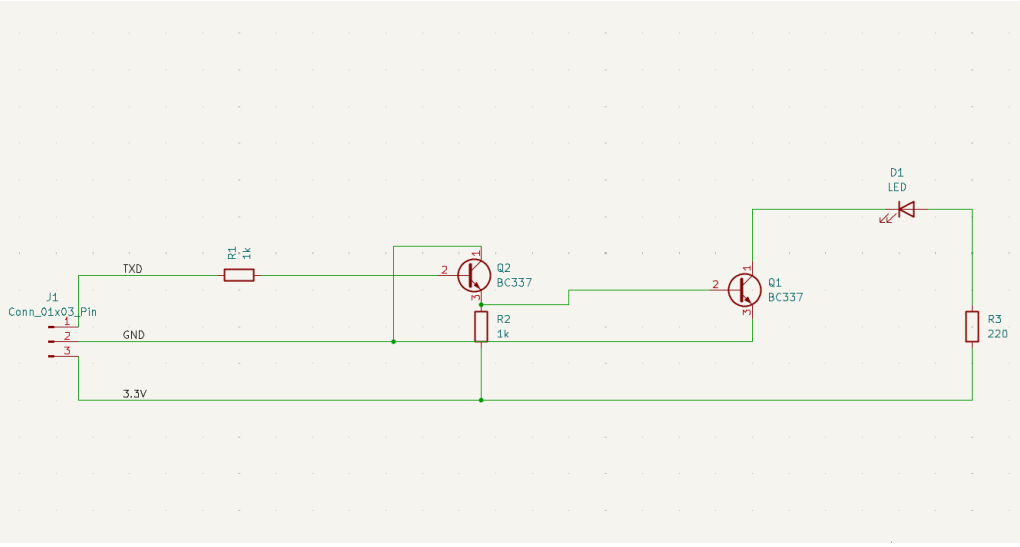

Figure 4.2: Schematic of the LED circuit used for visual zone indication.

4.1.5 3D-printed Sensor Stands

Due to the modular nature of the safety system, the sensors were intentionally not permanently mounted to the base of the robot. Instead, they were placed on custom-designed 3D printed stands, allowing flexible placement around the robot for optimal coverage. The stands were designed to elevate the sensors off the ground and had a can-like structure with outward-extending legs for stability. The hollow interior of each stand was filled with added weight to make them resistant to tipping if accidentally bumped. To further enhance visibility and reduce the risk of someone unintentionally knocking them over, the stands were printed in bright yellow, ensuring they would be easily noticeable in a typical industrial setting.

Because the sensors each scan a single plane, placing them at the same height could result in them detecting each other's laser pulses, which would interfere with accurate distance measurements. To avoid this, the sensor stands were designed with height differences of 1 cm, ensuring that each sensor scans a separate, non-overlapping plane. A 3D model of the sensor stand can be seen in Figure 4.3.

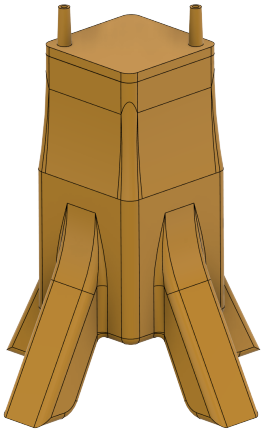

Figure 4.3: 3D model of one of the sensor stands. All sensor stands differ slightly in height.

4.2 Software

This section will describe the processes and methods of developing the software for the project. The software developed was a safety system based on interpreting data from LiDAR sensors in real time. Development was carried out using Python, chosen for the ability to rapidly develop systems, and support for many necessary features of the safety system. The software was implemented in stages, starting with communication with a single sensor, then scaling up to handle multiple sensors simultaneously, and then iteratively adding more features.

4.2.1 Reading Data from One Sensor

To collect and interpret distance data from the LD06 LiDAR sensors, a Python program was developed. The goal was to establish communication with the sensors via USB and extract useful measurement data, angles and distances, that could be used to monitor the area around the robot.

The script was designed to handle the sensor's communication protocol, which involves continuously receiving small data packets containing angle and distance values. To ensure that the data were read correctly, a step-by-step process was used in which the program first searched for the specific header byte 54H, and then saw if the correct header was followed by the correct data length 2C, both values specified in the sensor data [48]. If this was satisfied, then the program would continuously read the sensor data. A pseudo algorithm can be seen in Algorithm 1.

A representation of how a sensor sees its environment can be seen in Figure 4.4. Once the program was working reliably, it was used as a core part of further development of the system to continuously gather measurement data from the LiDAR sensors.

```
   Data: Continuous byte stream from sensor
   Result: Parsed sensor data packets
 1 Initialize:
 2 Set state to READ_FIRST_BYTE
 3 while system is active do
 4 |   if data stream is misaligned for extended time then
 5 |   |   Raise warning
 6 |   end
 7 |   if state is READ_FIRST_BYTE then
 8 |   |   Wait for start of packet byte
 9 |   |   When correct header received, set state to
   |   |    READ_SECOND_BYTE
10 |   else if state is READ_SECOND_BYTE then
11 |   |   Confirm second header byte
12 |   |   Set state to READ_REST_OF_PACKET
13 |   else if state is READ_REST_OF_PACKET then
14 |   |   Read remaining bytes and validate complete packet
15 |   |   Set state to STREAMING
16 |   else if state is STREAMING then
17 |   |   Continuously read and process full packets
18 |   else
19 |   |   Corrupt reading
20 |   |   Set state to READ_FIRST_BYTE
21 |   end
22 end
```

Algorithm 1: Generalized sensor data parsing loop

4.2.2 Reading Data from Multiple Sensors

The project is as stated about creating a safety system around one industrial robot, and one sensor alone could not cover the entire surrounding area. To effectively monitor the entire environment and create a 360-degree safety perimeter, a total of four LD06 sensors was used, positioned around the robot. To handle this setup in software, the original data reading program - initially designed to work with a single sensor - was restructured into a function, making it easier to reuse and scale.

To allow the system to read data from all four sensors at the same time, the Python multiprocessing library was implemented. Multiprocessing in Python allows multiple functions to run in parallel within a single program [52]. Each process can run independently, which is particularly useful when performing I/O-bound operations like reading serial data from sensors.

The four sensors were connected to the Raspberry Pi using the CP2102 UART USB bridge, and each sensor's respective port was referenced in the software. For each sensor, a process was created, allowing the data reading function, Algorithm 1, to run for each sensor. This approach allowed the program to simultaneously collect

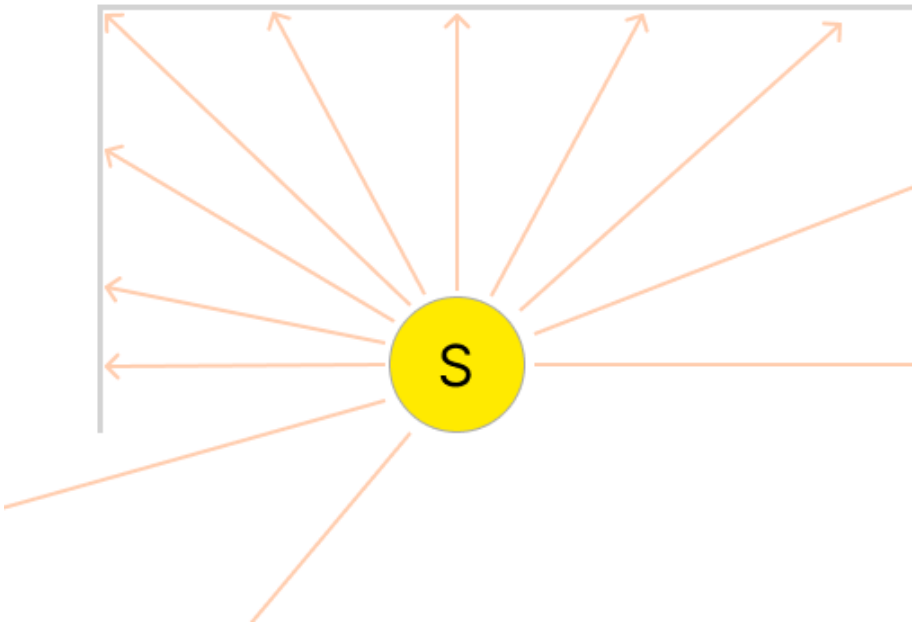

Figure 4.4: Representation of the sensor measuring its environment.

data from all sensors. An overview of the program architecture can be seen in Figure 4.5 As a result, the system could process inputs from all directions around the robot in real-time.

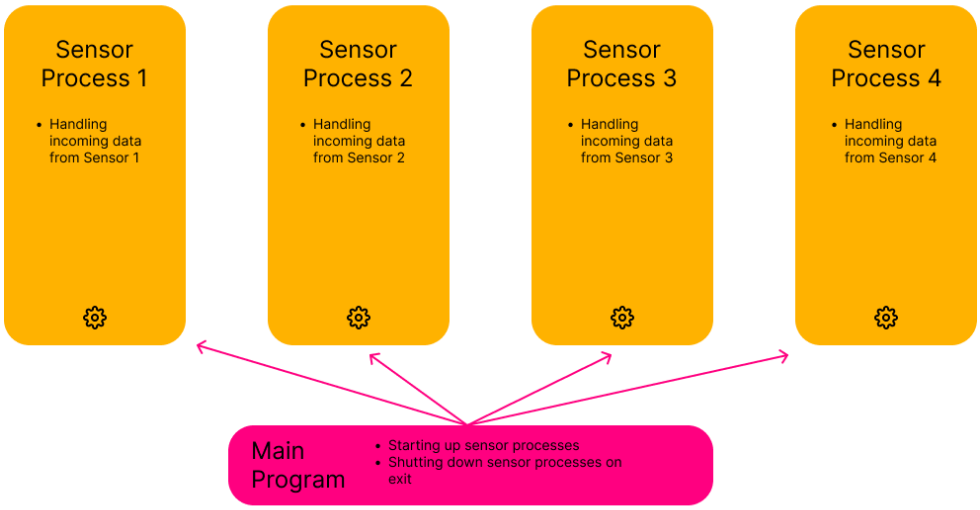

Figure 4.5: Representation of the program architecture and the parallel multi processes for handling incoming data from the different sensors.

4.2.3 Calibration

The sensors in this system were designed to be modular and easily repositioned, rather than fixed in predetermined locations. As a result, it was crucial to develop a calibration program. This program needed to determine the position of each sensor in relation to the robot and map the static environment, such as walls and obstacles, around the robot.

The safety detection mechanism would be based on monitoring boundary crossings relative to the robot, not the sensors themselves. Therefore, it was necessary to know the precise x- and y-coordinates of each sensor in relation to the robot's position, so that a distance conversion for measurements made by the sensors could be translated into robot perspective later on. The robot was defined as the origin point (0,0) in a cartesian coordinate system. Each sensor's x- and y-coordinates were treated as positional offsets relative to this origin. The global coordinate system with the robot as the origin is visualized in Figure 4.6.

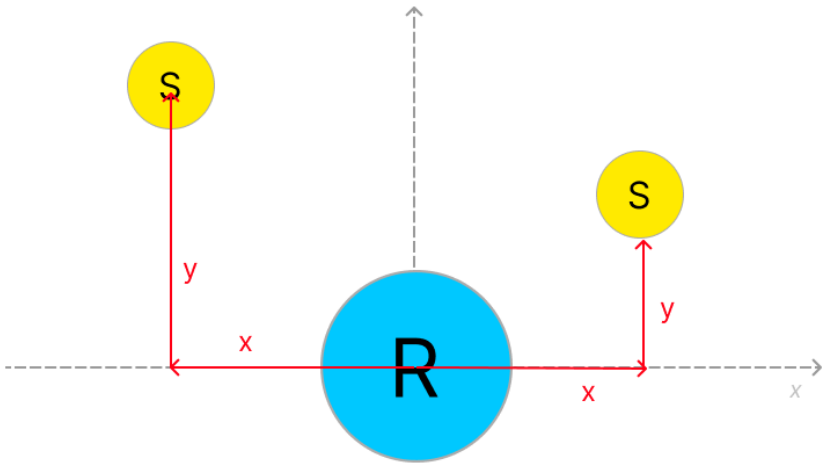

Figure 4.6: Representation of the robot coordinate system and different sensors x- and y- values in relation to robot.

With the sensors capable of making 4,500 measurements per second and scanning with a 0.01 degree angular resolution over a 360 degree range, it was important to strike a balance between spatial accuracy and computational efficiency. It was determined that a 0.1 degree resolution was sufficient for environmental mapping. At the sensor's maximum range of 12 meters, a 0.1 degree step corresponds to a spatial resolution of approximately 21 mm, which is well within the sensor's stated distance accuracy of $\pm$ 45 mm. This validated the decision to use 0.1 degree steps for calibration.

To set up the system, each sensor had to be manually placed and its distance to the robot measured along both the x- and y-axes. It was then constructed so at the start of the calibration process, the user was prompted to input the x- and y-coordinates of the sensor (in millimeters) relative to the robot's base. These coordinates were

essential for later calculating the boundaries in robot-centric space. If the sensors were moved then the system had to be calibrated again.

The calibration program was structured around creating an array with 3,600 elements for each sensor, where each element represented one 0.1 degree increment around the sensor. Initially, all array values were set to zero. During calibration, the sensor would continuously scan the room, and for each reading, it would insert the measured distance into the corresponding array index based on the angle of the reading. If the element at that index was still zero, the new reading was directly stored. If a value was already present, the program would take the average of the new and existing values, allowing it to smooth out noise or minor fluctuations in the sensor data over multiple scans.

To properly average the distance for a given angle, a second 'instances' array was introduced, also with 3,600 elements, corresponding to the 0.1 degree resolution, for each sensor. This array kept track of how many times each angle had been scanned. When a new reading was received, the program would now calculate a weighted average, multiplying the existing value by the number of times that angle had already been scanned, adding the new reading, and dividing the sum by the updated scan count. This greatly improved the stability and accuracy of the calibration, ensuring that outlier readings had a minimal impact over time as more data accumulated.

To further enhance the reliability of the data, the program also used the confidence value provided by the LD06 sensor for each distance reading. Readings with low confidence, indicative of weak reflections or potential errors, were discarded entirely, preventing poor-quality data from influencing the calibration.

The calibration process would run in 30 second increments until a distance had been read for all 3,600 angles for each sensor. After the calibration was completed, the finalized 360-degree distance profile was saved to a .csv file, along with each sensors position. This file could then be loaded and referenced in subsequent programs, enabling the system to interpret live sensor data in the context of the robot's surroundings. If an object is placed in the stop-zone, which should not be classified as an intrusion, then the system must be calibrated again to take the new object into account. The entire pseudo algorithm for calibrating the sensors can be seen in Algorithm 2.

4.2.4 Testing the Calibration Program

To evaluate the calibration program, a test script was written to extract the distance measurements for each sensor at selected angles in the calibration .csv file. In the test the angles 0°, 90°, 180° and 270° were chosen for each sensor. At the same angles, the actual distances was then measured manually using a measuring tape, from the sensor to the nearest obstruction. These measured distances were compared against the corresponding calibrated values. The average difference, the absolute average difference and the maximum absolute difference were then found and evaluated.

Data: Incoming data from four sensors
Result: A CSV file with calibrated distances at 3,600 angles per sensor
Initialization:
Initialize arrays `calibrated_values` and `instances` with 3,600 zeros
Set minimum confidence threshold
Ask user for (x, y) positions of each sensor
while *not all 3,600 angles are calibrated* **do**
    Read incoming distances and angles from sensor
    Round angles to nearest 0.1 degree
    for $i \leftarrow 1$ **to** 12 **do**
        `angle_index` $\leftarrow$ rounded angle at index i
        if *`calibrated_values[angle_index]` is 0* **then**
            `calibrated_values[angle_index]` $\leftarrow$ current distance
            `instances[angle_index]` $\leftarrow 1$
        end
        else
            old $\leftarrow$ `calibrated_values[angle_index]`
            new $\leftarrow$ current distance
            count $\leftarrow$ `instances[angle_index]`
            average $\leftarrow \dfrac{old \cdot count + new}{count + 1}$
            `calibrated_values[angle_index]` $\leftarrow$ average
            `instances[angle_index]` $\leftarrow$ count + 1
        end
    end
end
Save `calibrated_values` and (x, y) positions to a CSV file

Algorithm 2: Calibration loop for LiDAR sensors

4.2.5 Object Detection in Relation to Sensors

In order to begin constructing the actual virtual safety system, a program was required to detect intrusions, specifically whether an object or person crossed a defined boundary relative to one of the sensors, which would later be translated to a robot-relative coordinate system. The goal of this program was to issue a warning whenever a measured distance was both within a defined proximity of the sensor (e.g. 300 mm) and had a high confidence level. Importantly, it also needed to account for stationary obstacles like walls that were already present in the calibrated environment - meaning the system should not trigger false positives simply because an object was close, but only when it was unexpectedly closer than during calibration.

To implement this, the first step was to load the calibration data from a .csv file, which contained the average distances around each sensor collected during the initial scanning phase. These values represented the expected distances to surrounding surfaces at 0.1 degree angular resolution, 3,600 entries total. The values were loaded from the file into an array.

The program for handling potential intrusions were built on top of the data reading program for one sensor, Algorithm 1, which processes incoming measurement data packets in batches of 12 readings. Each data batch included 12 angles, 12 distances, and 12 confidence values. The angles were rounded to 0.1 degree resolutions, and these angles were used to reference the measured distances at those angles in the calibration array.

The core logic for intrusion detection was included within a function, where the 12 readings for angles, distances and confidences are looped. For each packet, low-confidence values readings are filtered out by checking if the confidence is above a user-defined threshold. For high-confidence readings, the measured distance at a measured angle gets checked for intrusion in an if statement:

- If the calibrated value is already below the threshold (e.g. due to a wall), it only triggers if the new reading is even closer than the calibration by a set accuracy margin (e.g. 60 mm).
- If the calibrated value is above the threshold, the system will alert if the new distance drops below the threshold, indicating something has entered a previously clear area.

Each valid reading that satisfies either of these alert conditions increments an alert index. If more than three such readings are found in a single packet, the program prints an intrusion message to signal that a person or object may have entered the threshold defined around the sensor. Although the sensor values were filtered by confidence, some nonsense data slipped through. The alert index filtering limited the amount of false alerts tolerable, reducing the number of false intrusions. The pseudo algorithm for finding intrusions can be seen in Algorithm 3.

This structure proved successful, as it provided a robust and flexible foundation for real-time intrusion detection in relation to reach sensor. It filtered out irrelevant or noisy readings, and was comparing incoming data against a known environment. This is demonstrated in Figure 4.7

While this program did not yet calculate the object's position in relation to the robot itself, it was still a valuable tool during development. It helped verify that each individual sensor was functioning correctly and provided immediate feedback when something entered its detection zone. This became especially useful when restarting the Raspberry Pi, as the order in which the sensors were connected affected how they were indexed by the system. The sensors were labeled from 0 to 3. As an example, if Sensor 0 was plugged in first, it would be referenced as Sensor 0 in the system. But if Sensor 1 was connected first, it would be referenced as Sensor 0 in the system. As a result, the same physical sensor could be assigned a different ID after a restart or reconnection. By using this program and observing which sensor triggered intrusion alerts in response to entering within a 10 cm threshold, it was possible to quickly identify and label each sensor correctly before running calibration.

Data: Angles, distances, and confidence values from one LiDAR sensor.
Result: Intrusions found within a set distance threshold.
Filter measurements:
Remove all measurements with confidence below a defined threshold
Start intrusion counter
foreach *remaining measurement* **do**
    Compare measured distance to reference value from calibration
    if *calibrated values are closer than threshold* ***and*** *incoming measurements are closer than calibrated values* **then**
        Add +1 to intrusion counter
    else if *calibrated values are not closer than threshold* ***and*** *incoming measurements are closer than threshold* **then**
        Add +1 to intrusion counter
end
Evaluate intrusion level:
Count number of intrusions
Determine current state:
if *Intrusions violations > 3* **then**
    Prompt INTRUSION message
end

Algorithm 3: Intrusion detection based on LiDAR measurements and calibration

4.2.6 Converting Calibrated Sensor Data to the Robot Coordinate System

Following the calibration procedure, the recorded distances were still relative to each sensor's local coordinate system rather than the robot's global coordinate system. To enable spatial interpretation of the sensor data from the robot's perspective, each reading needed to be transformed using vector addition.

As described earlier, the user is prompted at the beginning of the calibration routine to input the x- and y-coordinates of each sensor's position relative to the robot. These coordinates are then stored in the calibration file. To facilitate the transformation, these cartesian coordinates are first converted to polar coordinates using the following relationships:

$$r = \sqrt{x^2 + y^2} \tag{4.1}$$

$$\phi = \arctan\left(\frac{y}{x}\right) \tag{4.2}$$

Where r is the distance from the origin, x and y are the x- and y-coordinates and ϕ is the counter-clockwise angle in the polar coordinate system.

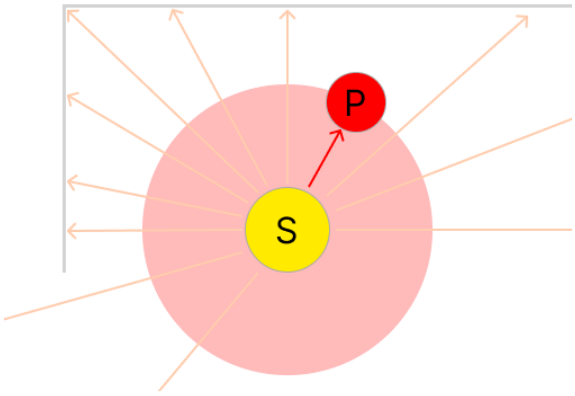

Figure 4.7: Representation of the detection zone around a sensor, with a person breaking the set threshold.

It is important to note that the sensors report angles in a clockwise direction, whereas the standard trigonometric convention assumes angles to be positive in the counter-clockwise direction. Therefore, all angle values obtained from the sensors were adjusted by subtracting them from 360 degrees, e.g. a sensor reading of 90 degrees (clockwise) was interpreted as 270 degrees in the robot's coordinate system. This is demonstrated in Figure 4.8.

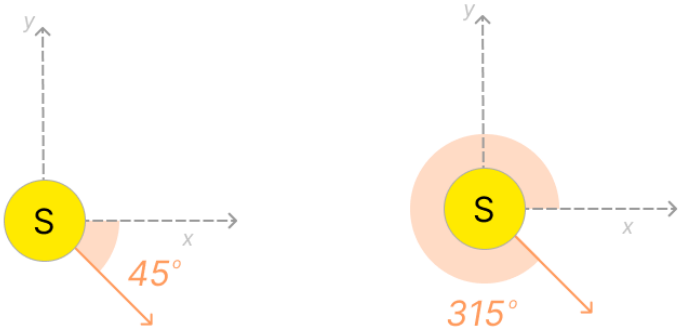

Figure 4.8: How sensor angles were recalculated from an counter clockwise direction to a clockwise direction.

These r and ϕ values represent the radial distance and angle from the robot's center to the respective sensor. Using this offset, each calibrated distance measurement can be recalculated into the robot's coordinate system. This is done using the following formulas shown below [53]:

$$r = \sqrt{r_1^2 + r_2^2 + 2r_1 r_2 \cos(\phi_2 - \phi_1)} \tag{4.3}$$

$$\phi = \phi_1 + \arctan 2\left(r_2 \sin(\phi_2 - \phi_1), r_1 + r_2 \cos(\phi_2 - \phi_1)\right) \tag{4.4}$$

Here, r_1 is the distance from the robot to the sensor, and ϕ_1 is the angular offset of the sensor in the robot's coordinate system. r_2 is the measured (calibrated) distance at angle ϕ_2 relative to the sensor's local orientation. r is the distance from the robot to the detected object and ϕ is the angle from the robot to the object. The parameters for the calculation is visualized in Figure 4.9. The procedure for how the calculation is made can be seen in Algorithm 4.

Data: csv file with 3,600 calibrated measurements per sensor: (r_2, ϕ_2)
Sensor position (x, y) and orientation ϕ_1 (relative to robot)
Result: Transformed object positions (r, ϕ) in robot's coordinate system
Convert sensor position to polar coordinates:
    $r_1 \leftarrow \sqrt{x^2 + y^2}$
    $\phi_1 \leftarrow \text{atan2}(y, x)$
Load 3,600 measurements from csv file into array `measurements`
foreach (r_2, ϕ_2) *in* `measurements` **do**
    Adjust angle from sensor:
        $\phi_2 \leftarrow 360° - \phi_2$
    Compute angle difference:
        $\Delta\phi \leftarrow \phi_2 - \phi_1$
    Apply vector addition to compute global object position:
        $r \leftarrow \sqrt{r_1^2 + r_2^2 + 2 r_1 r_2 \cos(\Delta\phi)}$
        $\phi \leftarrow \phi_1 + \text{atan2}(r_2 \sin(\Delta\phi), r_1 + r_2 \cos(\Delta\phi))$
    Store (r, ϕ) in array;
end

Algorithm 4: Transform 3,600 sensor measurements to robot's coordinate system

Applying these transformations for all four sensors provided a unified and accurate representation of the robot's surroundings, allowing the system to detect obstacles and assess proximity from a common spatial reference as seen in Figure 4.10

4.2.7 Object Detection in Relation to Robot

To enable a functional safety system, the incoming sensor data stream needed to be continuously converted into the robot's coordinate system. This would allow for a unified spatial understanding of the environment, enabling decisions about robot behavior based on object proximity regardless of which sensor detected it.

As with the angle conversion during the calibration process, all real-time sensor angles, measured clockwise, were adjusted to fit the counter-clockwise convention of the robot's coordinate system. For instance, a sensor reading of 90 degrees was interpreted as 270 degrees, 45 degrees became 315 degrees, etc (see Figure 4.8). This conversion ensured consistency in spatial interpretation.

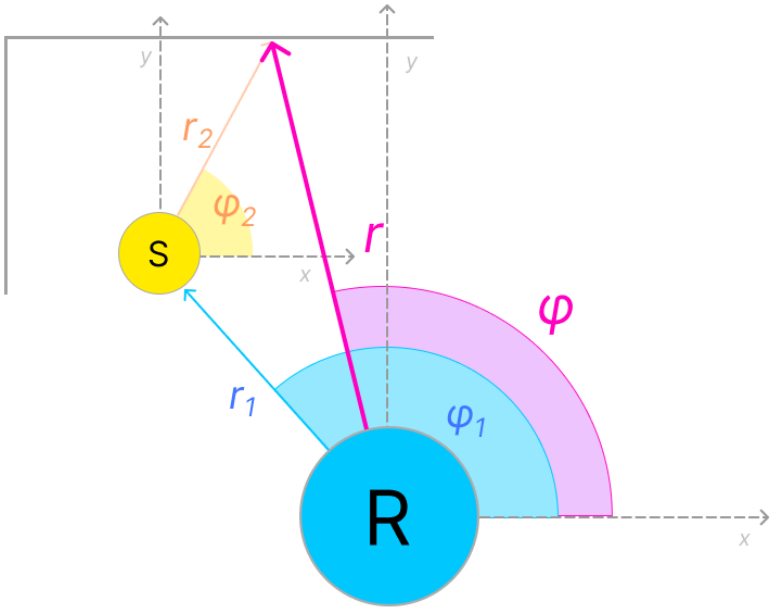

Figure 4.9: The different parameters for calculating the distance and angle from a sensor perspective to a robot perspective.

All sensor data was then continuously transformed using the same vector addition method applied during calibration. Each sensor's angle of perspective was converted into the robot's frame of reference using Equation 4.2. The measured distance was also recalculated relative to the robot using Equation 4.1. This transformation used the x- and y-values for each sensor put in during the calibration process.

Two critical distance thresholds, seen in Figure 4.11, were implemented to govern the robot's response to detected objects:

- **Slow-zone threshold:** If an object was detected within this range, the robot would continue operating but at reduced speed. This boundary was set 500 mm beyond the stop-zone for added safety margin, during testing.
- **Stop-zone threshold:** If an object was detected within this range, the robot would halt immediately.

The logic used to trigger alerts was an extension of the method developed for sensor-based detection. Each sensor sends data in packets of 12 readings. For each reading, after converting the measured angle and recalculating the position relative to the robot, the program evaluated whether the detected point fell within either threshold. Two alert counters were used:

- One alert counter for Slow-zone intrusions
- One alert counter for Stop-zone intrusions

If more than three in a single data packet triggered the same type of alert, or if

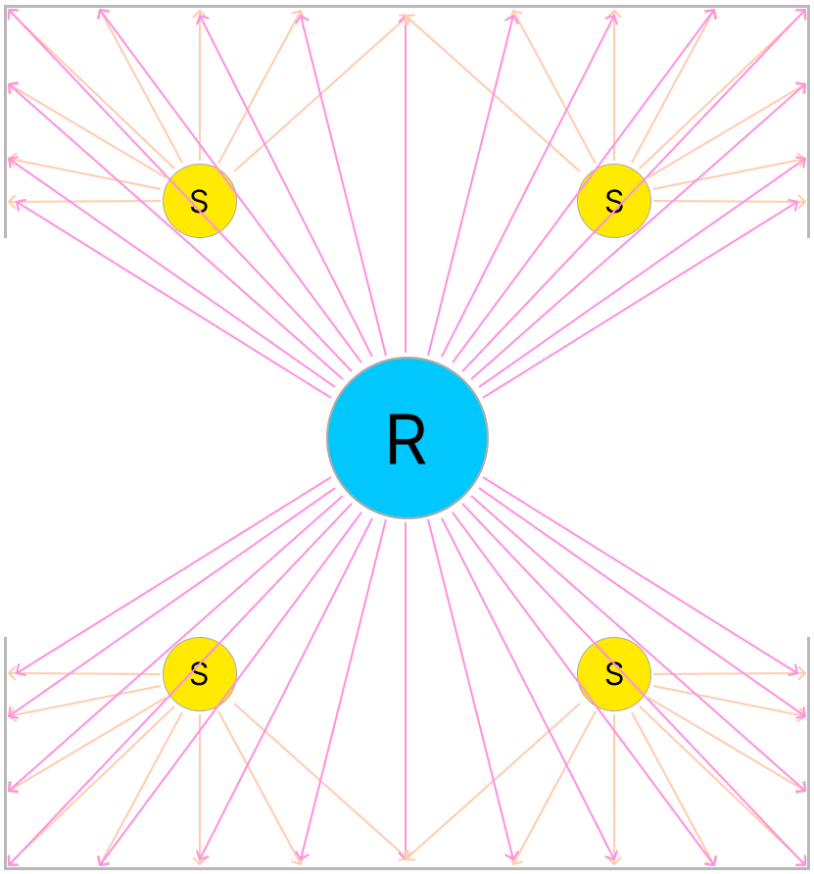

Figure 4.10: How the robot sees the environment after vector addition done to the sensors calibrated values.

the combined number of slow- and stop-alert counters in that packet was more than three, a system-wide ”SLOW” or ”STOP” warning was issued. The alert also included information about which sensor triggered it. The entire pseudo algorithm for handling intrusions can be seen in Algorithm 5. The safety system architecture can be seen in Figure 4.12.

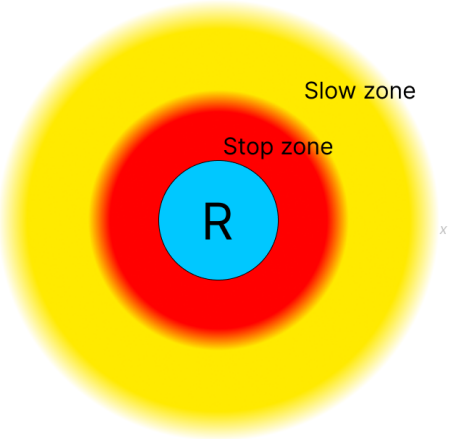

Figure 4.11: Representation of safety zones.

Data: Angles, distances, and confidence values from one LiDAR sensor.
Result: Intrusions found within a set distance threshold.

Filter measurements:
Remove all measurements with confidence below a defined threshold
Recalculate incoming sensor measurements and angles to robot perspective
Start intrusion zone counters:
stop_violation_counter
slow_violation_counter
foreach *remaining measurement* **do**
    Compare measured distance to reference value from calibration
    if *calibrated values are closer than* ***stop-****threshold* ***and*** *incoming measurements are closer than calibrated values* **then**
        Add +1 to *stop_violation_counter*
    else if *calibrated values are not closer than* ***stop-****threshold* ***and*** *incoming measurements are closer than* ***stop-****threshold* **then**
        Add +1 to *stop_violation_counter*
    else if *calibrated values are closer than* ***slow-****threshold* ***and*** *incoming measurements are closer than calibrated values* **then**
        Add +1 to *slow_violation_counter*
    else if *calibrated values are not closer than* ***slow-****threshold* ***and*** *incoming measurements are closer than* ***slow-****threshold* **then**
        Add +1 to *slow_violation_counter*
end
Evaluate intrusion level:
if *stop_violation_counter* > 3 **then**
    Prompt STOP message and Sensor ID
else if *slow_violation_counter* > 3 **then**
    Prompt SLOW message and Sensor ID
else if *slow_violation_counter + stop_violation_counter* > 3 **then**
    Prompt STOP message and Sensor ID

Algorithm 5: Intrusion detection in relation to robot

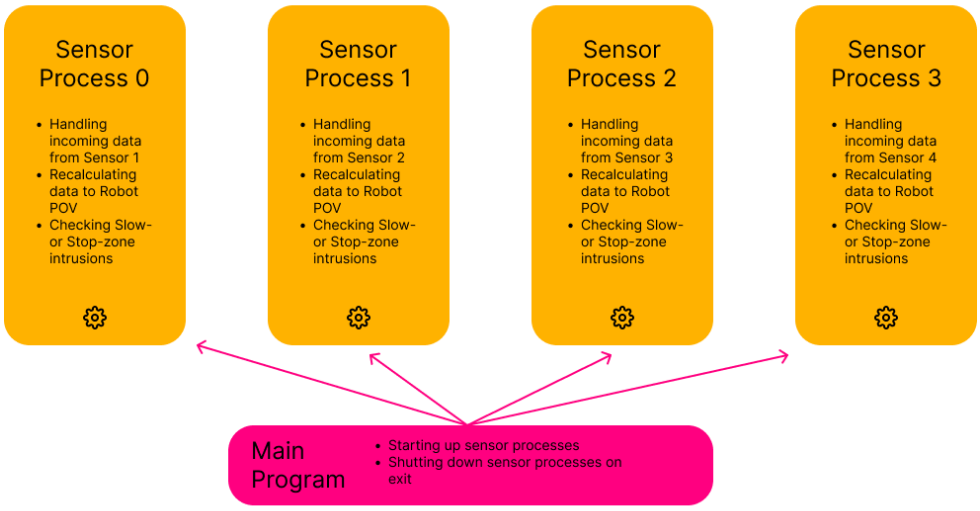

Figure 4.12: Representation of program architecture at this stage

4.2.8 Testing Object Detection in Relation to Robot

To evaluate the object detection system in relation to the robot's safety zones, two types of test cases were conducted: one for slow-zone intrusions and one for stop-zone intrusions. The stop-zone was defined as a radius of 1,000 mm from the robot, while the slow-zone extended to 1,500 mm. These distances were clearly marked on the ground to ensure consistent positioning during testing.

Each test involved a person starting from a position well outside the defined safety zones and then walking into the area, exceeding the respective threshold by more than 100 mm. The system was expected to issue a "SLOW" alert when entering the slow-zone and a "STOP" alert upon entering the stop-zone. In each case, the system also displayed the angle and distance of the detected object relative to the robot, along with the corresponding safety message.

The outcome of each test was recorded using a binary scoring method: a score of 1 for a successful alert and 0 for an unsuccessful one. A total of 10 tests were performed for each safety zone, and the results were documented in a test protocol.

4.2.9 Determining the Size of the Safety Zones

According to ISO 13855 [54], the minimum distance to a hazardous point can be determined using the following formula:

$$S = K \cdot (t_1 + t_2) + C + Z_t \tag{4.5}$$

This formula in turn creates a defined critical safety zone around a machine, in this

case an industrial robot. The value of K, representing the human approach speed in mm/s, is often set according to recommended values in ISO 13855. For full-body movement, the standard specifies $K = 1600$ mm/s which corresponds to a fast walking speed. t_1 is the response time of the system [s], t_2 is the machine stopping time [s], Z_t is a general safety allowance and C is an additional distance depending on the height of the scanning plane and the scanner resolution: $C = 1200 - 0.4H$, where H is the height of the scanning plane.

One key parameter in this equation is the variable K, which represents the intruding object's approach speed. Although it was theoretically possible to calculate this speed dynamically within the system, the associated complexity of the calculations led to the decision to use a fixed value.

While the ISO standard assumes a typical human walking speed of 1,600 mm/s, this value was deemed insufficient for the intended safety requirements. The system is designed not only for normal operations but also to account for unexpected or accidental intrusions. Therefore, a more conservative speed of 4,000 mm/s was selected to reflect the velocity of a person unintentionally entering the safety zone, for example by stumbling or tripping.

To determine the system's reaction time, several tests were conducted. These measured the interval between the detection of an object and the transmission of a stop signal to the robot. The results showed an average reaction time of 10 ms, which was used as the t_1 value in the equation. The robot's actual stopping time (t_2) could not be measured directly during testing. Instead, it was estimated based on the robot's datasheet specifications. The robot has a maximum speed of 3 m/s and a peak deceleration of 36 m/s^2, yielding a theoretical stopping time of approximately 83 ms [20]. To ensure a conservative safety margin, this value was rounded up to 100 ms.

The approach distance C was calculated using a detection height H of 150 mm, resulting in a C value of 1140 mm. Additionally, a general safety allowance Z_t of 900 mm was included. This value corresponds to the robot's maximum operational reach. Combining these values, the minimum safe stopping distance was calculated as follows:

$$S = K \cdot (t_1 + t_2) + C + Z_t = 4000 \cdot (0.01 + 0.1) + 1140 + 900 = 2480 \text{ mm} \quad (4.6)$$

For simplicity and safety, this value was rounded up to 2,500 mm. Due to spatial constraints in the development environment, testing with the full 2,500 mm stop-zone was not feasible. Instead, tests were conducted using only reduced stop zones of 1,000 mm and 2,000 mm. The slow-zone was also implemented, extending always 500 mm beyond the boundary of the stop-zone.

4.2.10 Creating a ROS 2 Workspace on the Raspberry Pi

To effectively stop the robot's movement and enable real-time communication of sensor data and safety alerts, the system needed to broadcast information such as proximity readings and stop- or slow-zone warnings. This logic is implemented in the Raspberry Pi using a ROS 2 package named `virtual_fences`, containing three Python scripts, each serving a distinct role in the safety system.

1. **Identification** – This script was used to determine which physical sensor corresponds to which ID in the system. By waving a hand within 10 cm of one sensor, the ID of that sensor would be displayed, which then could be used when entering x- and y-position for that sensor in the calibration program.

2. **Calibration** – This script guided the user through entering the physical x- and y-positions of each sensor. It then performed a 360.0 degree scan of the environment and saved the average measured distances at each angle into a `.csv` file. These calibration files were later loaded into the safety system to define expected environmental boundaries.

3. **Safety System** – This was the main script responsible for evaluating the environment in real time. It continuously processed sensor readings and determined whether any object or person had entered the robot's predefined safety zones (slow or stop).

Each of these scripts could be executed using the command:

```
ros2 run virtual_fences <script_name>
```

Where `<script_name>` would be replaced with either `identify`, `calibrate`, or `activate`, depending on which functionality was required. With the ROS 2 project in place, development towards a working simulation could take place.

4.2.11 Publishing Sensor Data

The safety system script uses Python's `multiprocessing` library so that each sensor can be handled by a separate process. When a sensor process receives a data packet from the LiDAR, it places the data in a queue, which serves as a shared memory between the processes. A ROS 2 node then periodically publishes messages from the shared queue at a rate of 100 ms as `LaserScan` messages on the /laser_scan topic. This makes the data available to all other nodes in the network, including visualization tools such as RViz2.

This approach allowed all sensor data to be published on a single topic, while also ensuring that each sensor process could operate independently. Furthermore, computational overhead was reduced as sensor processes were not burdened with publishing tasks and could instead focus solely on reading data. The architecture with the sensor data publisher can be seen in Figure 4.13

As the purpose of publishing sensor data was solely for visualization, and not part of the real-time safety mechanism, a 100 ms interval was deemed sufficient.

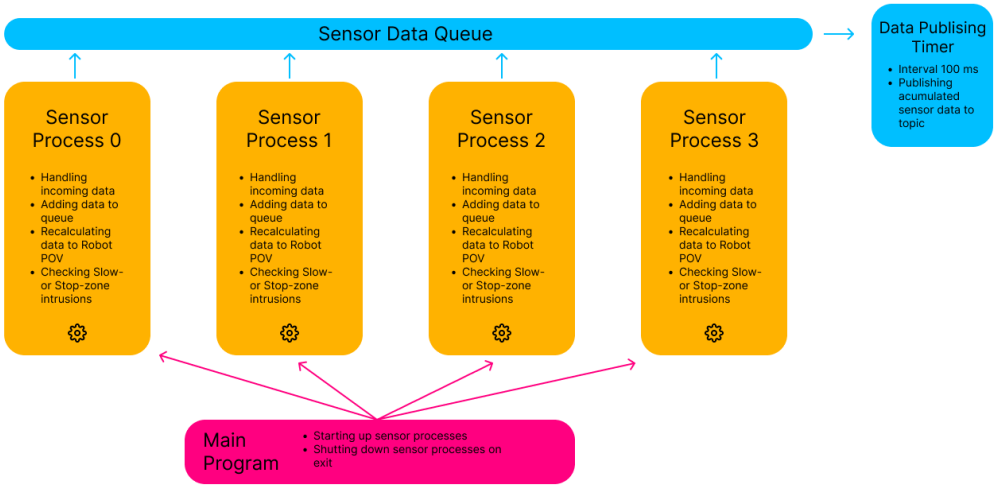

Figure 4.13: Program architecture with data queue and timed data publishing

4.2.12 Publishing Robot Control Messages

To indicate a safety zone breach, an independent timer node publishes control messages on the /robot_control_topic at a high frequency (every ten milliseconds). To prevent the system from being overwhelmed with simultaneous messages from multiple sensors, this timer node also uses a dedicated queue to read control messages from each sensor process, indicating whether an object entered either the slow-zone or stop-zone. If a sensor process detects an object in the slow-zone, a slow message is placed in the queue. Similarly, if an object enters the stop-zone, a stop message is inserted. When one or more stop messages are detected, the node publishes a single STOP-signal to the robot. Once a slow message is detected, a SLOW-signal is sent. These control signals were represented by integer values, with 2 indicating a slow-zone breach and 1 indicating a stop-zone breach. Only one message is sent during each zone breach. This architecture can be seen in Figure 4.14.

The architecture offered several advantages:

- Only one control message was published per event, avoiding message congestion or potential conflicts caused by multiple sensors triggering simultaneously.
- The high-frequency timer ensured fast response times to safety zone violations.
- Communication with the robot was simplified, as only a single ROS 2 topic was needed to convey control messages.

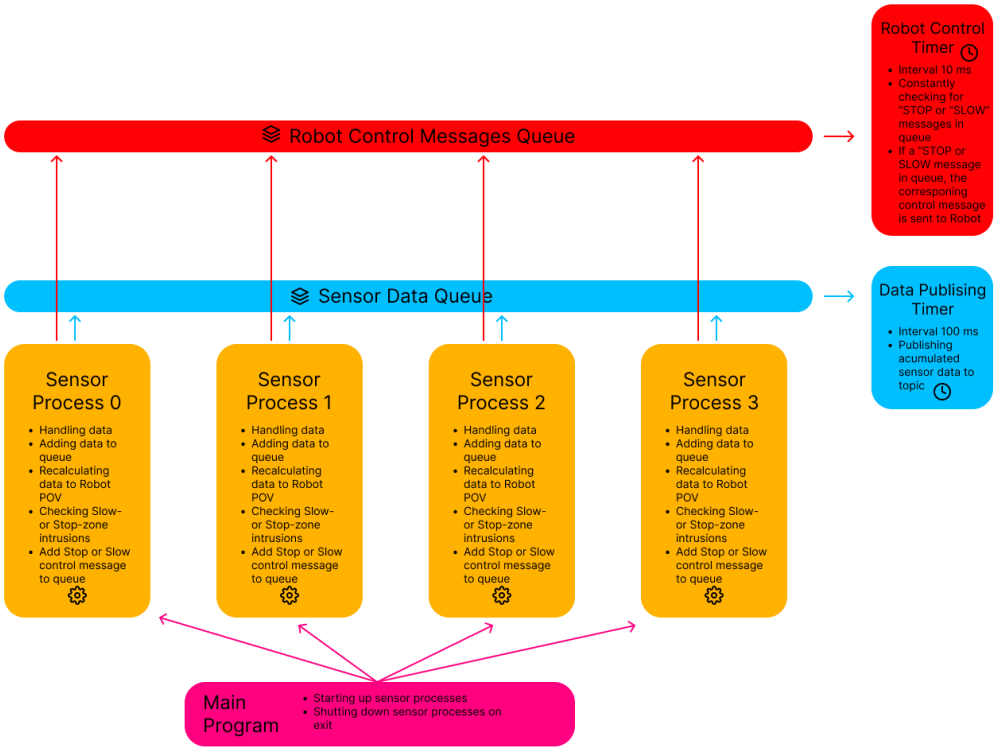

Figure 4.14: Program architecture with control messages queue and timed messages publishing

4.2.13 Visually Displaying Zone Breaches using LEDs

To effectively communicate proximity breaches to a user, visual indicators were implemented using LEDs connected to the Raspberry Pi's GPIO pins. Four LEDs were used in total, with each one assigned to a specific sensor.

In the software, each sensor was linked to a corresponding GPIO pin, and its LED behavior was controlled based on the type of safety zone breach detected:

- If an object entered the slow-zone, the assigned LED began blinking slowly to signal caution.
- If the object continued into the stop-zone, the LED switched to a rapid blinking, indicating that the object had reached the critical area closest to the robot.

Figure 4.15: Demonstration of LEDs lighting up when a person breaches a security zone. The safety zones are here smaller than allowed due to demonstration purposes.

Since the LEDs served solely as a visual indicator to show when an intrusion had occurred, the real-time control of the LEDs was assigned a lower priority. During testing, the LEDs accurately reflected intrusion events, but with a slight delay that did not affect the speed and performance of the overall safety system. A demonstration of the LEDs can be seen in Figure 4.15.

This progressive LED signaling provided an intuitive and immediate way to assess the system's status during testing and its architecture can be sen in Figure 4.16. It would allow for users to quickly understand both the presence of intrusions and their severity, contributing to a more transparent and user-friendly interface.

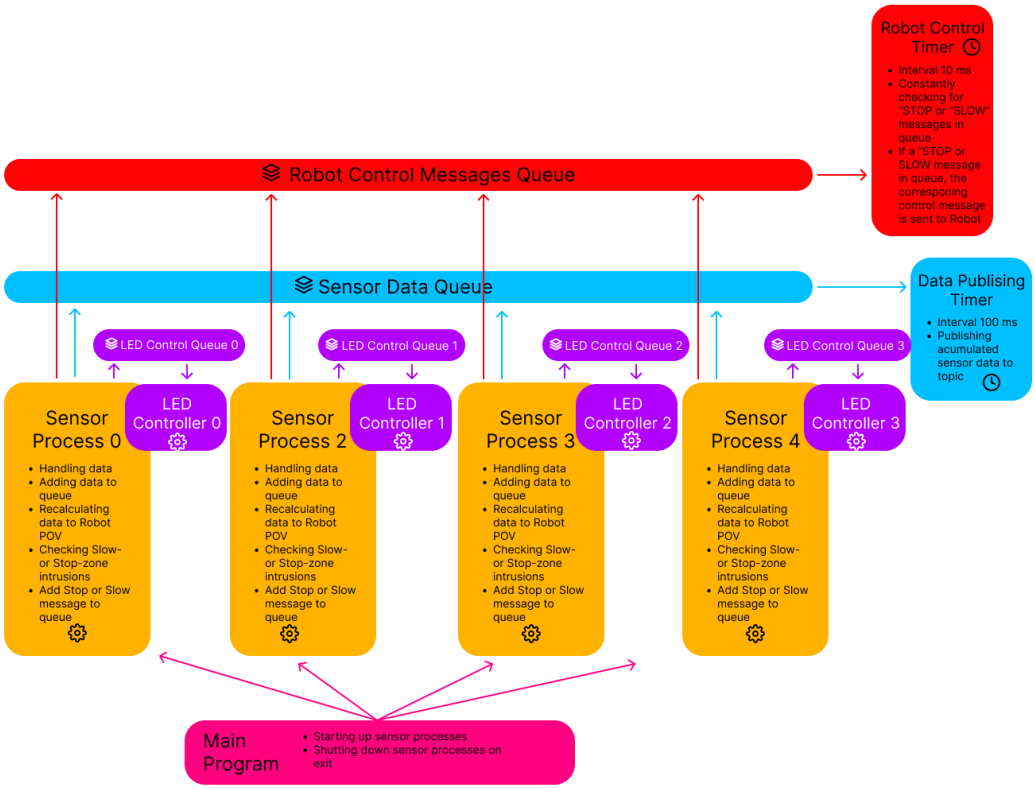

Figure 4.16: Program architecture with added LED functionality

4.2.14 Testing the Safety System

The following tests were carried out to test the safety system:

- **Response Times**
  - Incoming data → "STOP" signal sent to robot, **without** sensor data publishing, LED control, or emergency stop button
  - Incoming data → "STOP" signal sent to robot, **without** sensor data publishing
  - Incoming data → "STOP" signal sent to robot, **with** sensor data publishing
  - Emergency button pressed → "STOP" signal sent to robot, **without**

sensor data publishing

  - Emergency button pressed → "STOP" signal sent to robot, **with** sensor data publishing

- **Data Filtering**
  - Number of false STOP intrusions, 1 m stop-zone, **without** sensor data publishing
  - Number of false STOP intrusions, 1 m stop-zone, **with** sensor data publishing
  - Number of false STOP intrusions, 2 m stop-zone, **without** sensor data publishing
  - Number of false STOP intrusions, 2 m stop-zone, **with** sensor data publishing
  - Number of false STOP intrusions, 1 m stop-zone, **without** sensor data publishing, LED control, or emergency stop button
  - Number of false STOP intrusions, 2 m stop-zone, **without** sensor data publishing, LED control, or emergency stop button

The response time was measured in the software by time stamping key events in the system. When a data packet was received from the LiDAR sensors, the current system time was recorded. If the packet indicated an object within the defined stop-zone (set to 1,000 mm), this timestamp was sent along with the `STOP` message to the Robot Control Timer node. A second timestamp was taken in the Robot Control Timer node immediately after the `STOP` signal was published to the robot. The time difference between these two timestamps was calculated and printed to the terminal, providing a direct measurement of system response time. This value was noted in a test protocol. The tests were conducted by having a test person walk into the stop-zone to trigger an intrusion, and 10 measurements were collected for each test case.

The stop button test was measured using the same timing method as the intrusion response time tests. When the stop button was pressed, the press detection time was recorded and sent to the Robot Control Timer. There, a second timestamp was taken at the moment the stop signal was published. The response time was calculated as the difference between these two timestamps and displayed in the terminal. To perform the tests, a test person manually pressed the stop button while positioned well outside the defined stop-zone, ensuring that the intrusion detection system was not triggered simultaneously. The response times were shown in the terminal and documented in a test protocol.

To evaluate the performance of the data filtering algorithm, a `STOP`-intrusion counter

was implemented in the software. This counter incremented whenever an intrusion, real or false, was detected and printed the running count to the terminal. The tests were conducted by monitoring the stop-zone over a 20-minute period while ensuring that no real intrusions occurred. The number of false intrusion detections during this time was recorded and noted in the test protocol. One measurement was performed per test case. The reason for the different test cases was that it was observed that the system behaved differently depending on which functions were active while the safety system was active.

4.3 Simulation and Visualization of the ABB Robot

This section describes how simulation and visualization were implemented to test and validate the robot system in a virtual environment. The simulation was carried out using Gazebo, which provided a realistic physics engine and 3D environment for evaluating robot behavior. Visualization was handled using RViz2, which enabled monitoring of the system in real time. The setup included migrating existing simulation packages to be compatible with ROS 2 Jazzy, configuring controllers and kinematics, and developing custom nodes for goal state handling and sensor input.

4.3.1 Versions and Compatibility

The simulation tool used for this project was Gazebo with the version GZ Harmonic. Visualization was implemented with RViz2 and the workspace was set up on an external pc with Ubuntu 24.04.2 LTS and ROS 2 Jazzy LTS.

The code for the IRB 1200 RViz2 and Gazebo packages were originally developed for the Foxy and Humble ROS distributions and were not compatible with Jazzy. Migration resulted in updating the Gazebo integration from the use of the gazebo_ros packages to ros_gz_sim which is a newer bridge designed to handle launch files and ROS 2 enabled executables. Additionally, Joint_state_controller was updated to joint_state_broadcaster due to a need to differentiate between simple data broadcasters and controller.

In the migration to Jazzy the processes of spawning controllers changes from using a command format of a single string to instead use a list of arguments resulting in a less error prone creation of the controllers. Furthermore, Foxy did not have defined joint_limits which is a standard check for configuration files in Jazzy. Therefore a joint_limits file was implemented with both max acceleration and max velocity. There were also some changes made to the planner where in Jazzy the planning face is separated into a preprocessing face and postprocessing face. This differs from Foxy where such separation is not required.

4.3.2 Simulation and Visualization Project Structure

The source code for the robot IRB 1200 is fetched from the IFRA (2022) ROS2.0 ROBOT SIMULATION [55], which is an open-source platform for robots created by Cranfield University.

The code used in this project is from the `IRB1200` folder, which includes the packages `irb1200_ros2_gazebo` and `irb1200_ros2_moveit`. The `irb1200_ros2_gazebo` package includes everything required to run the robot simulation in Gazebo, while `irb1200_ros2_moveit` provides the necessary components for the implementation of motion planning. The folder structure of `IRB1200` is displayed in Figure 4.17

```
IRB1200/
├── irb1200_ros2_gazebo/
│   ├── config/
│   ├── launch/
│   ├── meshes/
│   ├── urdf/
│   ├── worlds/
│   ├── CMakeLists.txt
│   └── package.xml
└── irb1200_ros2_moveit2/
    ├── config/
    ├── launch/
    ├── CMakeLists.txt
    └── package.xml
```

Figure 4.17: Folder structure for the IRB1200

4.3.3 Launch Configurations

The launch file is used to launch Gazebo, MoveIt2, RViz2, transform nodes, controllers, sensor models and the action node XYZW. This enables a streamlined way to initialize all the necessary nodes for running the simulation, visualization and other important nodes.

Furthermore when initializing the launch file the necessary URDF and mesh files for visualizing the robot will loaded. The robot will both visualized in RViz2 and gazebo. In Gazebo the robot will be displayed in and a world file which defines a minimal environment containing only a ground plane and a light source.

4.3.4 Transforms and Placement of Objects

The placement of objects in RViz2 is determined through transforms using the tf2_ros packages which builds a TF tree representing the relationships between coordinate frames [56], shown in Figure 4.18. For moving frames such as the robot itself, the packages robot_state_publisher is used to dynamically publish updated transforms based on the robots current joint states that are published

by the joint_state_broadcaster [57]. Stationary frames such as the sensor models and sensor readings were instead transformed with the static_transform_publisher packages.

4.3.5 Visualization of Sensor Data

To visualize sensor data, a node in RViz2 (Figure 4.19) subscribes to the /laser_scan topic published by the raspberry-pi node. The ROS 2 sensor_msgs package is used for the data format of the messages published on the topic, which provides a simple and robust message standard for 2D-LiDAR sensors.

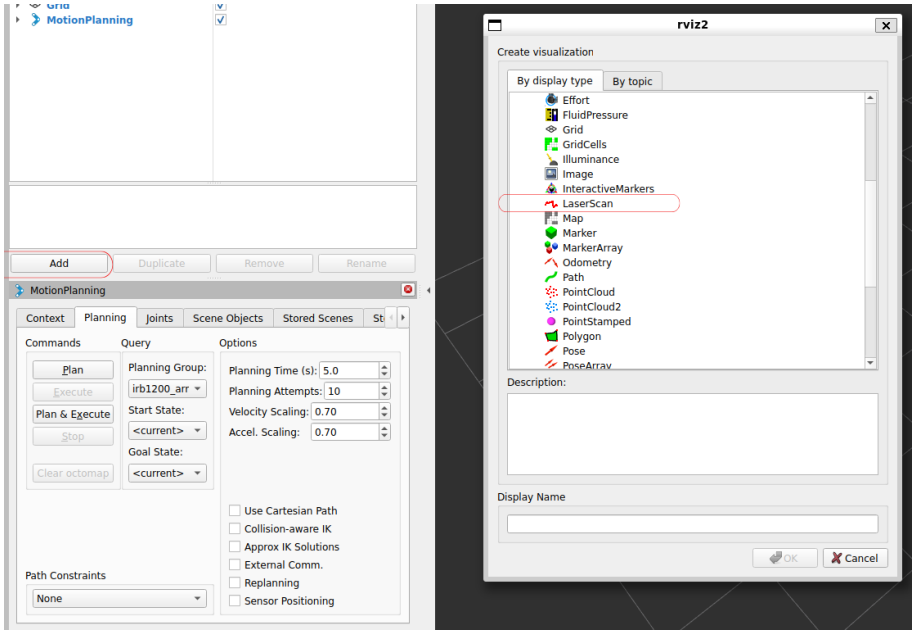

Figure 4.19: Implementation of the Laserscan visualization.

4.3.6 Motion Planning and Adapters

Goal states can be sent to the /move_group either by the custom-built action node moveXYZW_action.cpp or through the RViz2 MotionPlanning interface. When a goal state is set, MoveIt2 generates a joint trajectory using the OMPL motion planning library. Four request adapters are then used to pre-process the trajectory before it is sent to the controller. "ValidateWorkspaceBounds" checks that the requested start and goal states lie within the defined workspace bounds of the robot [58]. This prevents invalid plans caused by user errors such as requesting a goal state that is physically unreachable. "CheckStartStateBounds" validates that the starting position lies within the robot's joint limits. "CheckStartStateCollision" checks if the robot is in self-collision or colliding with the environment. If a collision or joint limit violation is detected, the request will either be rejected or the start state will be nudged slightly and a second attempt will be made.

After the plan is completed and before it is executed, three response adapters are

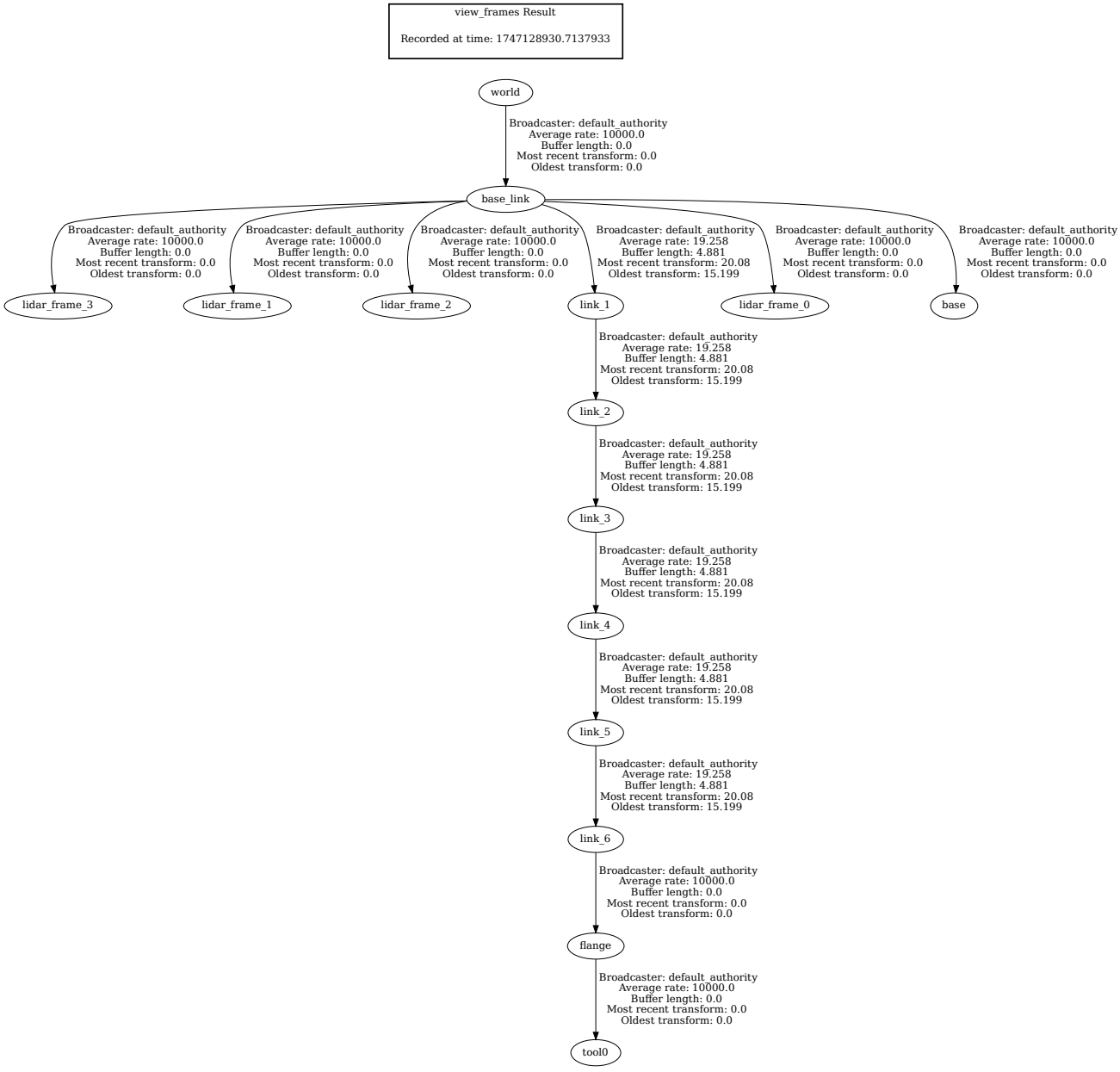

Figure 4.18: TF2 frame tree generated using the command ros2 run tf2_tools view_frames, illustrating the hierarchical relationship and transformation connectivity between frames in the robot system.

also used to post-process the trajectory. ”AddTimeOptimalParametrization” adds time stamps to the joint states in the trajectory based on velocity and acceleration limits, using the Time Optimal Trajectory Generation (TOTG) algorithm- allowing controllers and real-world robots to interpret the plan. TOTG works by fitting path segments to the original trajectory and then sampling new waypoints from the optimized path [59]. ValidateSolution ensures that the trajectory hasn't become invalid due to dynamic environments or rounding errors. Lastly, the ”DisplayMotionPath” adapter publishes the plan to RViz2 for a visualization of the path. This is useful for debugging and confirming reasonable motion paths.

4.3.7 Kinematics Solver for Simulation

The kinematics solver implemented in this project is a KDL kinematics plugin which is a commonly used kinematics solver for MoveIt2. A kinematic solver is responsible for finding joint values that achieves a desired position for the robot [39]. Both the values for the kinematics_solver_search_resolution and the kinematics_solver_timeout hade already been optimized from the IFRA (2022) ROS2.0 ROBOT SIMULATION developers [55]. This plugin is loaded in the launch file and passed as a parameter to move_group node.

4.3.8 Computing Motion Commands using Controller

The controller setup integrates MoveIt2 with the ros2-control package. In this project the real-time controller ”irb1200_controller”, defined in the irb1200_ros2_gazebo package under irb1200_controller.yaml. This controller is a ”JointTrajectoryController” that interprets joint space trajectories sent by MoveIt2 through the ”FollowJointTrajectory” action interface and computes new joint states (see theory for explanation). The controller is managed by the ”controller_manager” node, which defines controller launch and hardware parameters and runs a control loop at 100 Hz. A ”joint_state_broadcaster” is used to publish the robot's joint states, allowing tools like RViz2 and Moveit2 access to this information. The controller itself is also subscribed to this topic for feedback data.

4.3.9 Sending Goal States

To enable the robot to receive and execute goal states an action node was implemented. The action node handles receiving a goal state in form of position (x, y, z), euler angles (yaw, pitch, roll) and forwarding it to the motion planner. The structure of the action nodes are displayed in figure 4.20.

```
ros2_actions/
├── scripts/
│   ├── move_xyzw_client.py
│   └── moveXYZW_action.cpp
├── CMakeLists.txt
└── package.xml
ros2_data/
├── action/
│   └── MoveXYZW.action
├── CMakeLists.txt
└── package.xml
```

Figure 4.20: Folder structure for the action nodes

In ros2_actions the action server is created in moveXYZW_action.cpp. The action server handles incoming goals, cancellation and requests. When a goal is received it is convert to a MoveTt2 compatible target position and sends the data to the move group interface that creates a motion plan. If the creation of the motion plane is successful the plan will be executed.

move_xyzw_client.py creates a action client responsible for sending new goal states to the action server. It waits for the action server to become available and then sends a goal to moveXYZW_action.cpp. This process repeats over four sequences simulating a working robot.

Both the move_xyzw_client.py and MoveXYZW.action is based on the IFRA (2022) ROS2.0 ROBOT SIMULATION [55].

In the ros2_data folder the action interface for the ros2_actions is defined. This includes what data the action node should send and what type of results that are expected.

4.3.10 Stopping the Robot

To stop the robot when the stop-zone is breached, stop_robot.py runs a node listening to the /robot_ control_topic. When a ”1” is published, indicating a stop-zone breach, the node will cancel the robot’s current action and publish a joint trajectory with the current state and zero velocity.

4.3.11 Time Implementation

To enable the execution of the simulation, visualization and MoveIt2 a clock bridge was created between Gazebo and ROS 2. The clock bridge needs to be manually started in a separate terminal using the command `ros2 run ros_gz_bridge parameter_bridge /clock@rosgraph_msgs/msg/Clock[gz.msgs.Clock` where the time will get published on the topic /clock. This allows the robot_state_publisher, move_group, RViz2 and the moveXYZW_action.cpp node to get access to the

simulation time provided by Gazebo. This synchronizes all components to the same time and ensures that they run consistently.

4.3.12 Sensor Model

The sensor model is described in the folder my_robot_description with the following folder structure displayed in Figure 4.21.

```
my_robot_description/
├── models/
│   └── sensor_model/
│       ├── meshes/
│       │   └── sensor_model.stl
│       ├── model.config
│       └── sensor_model.urdf
├── CMakeLists.txt
└── package.xml
```

Figure 4.21: Folder structure for the sensor model.

In the meshes folder a sensor.STL file is defined which defines the sensor geometry. The STL file is then referenced in the sensor_model.urdf for RViz2 to load for visualization.

5

Results

This chapter presents the outcomes of the project, based on the implementation and testing of the developed safety system. It includes results from software functionality and the integrated system as a whole. Key findings from development and tests are summarized to evaluate how well the system meets its intended goals.

5.1 Hardware

This section includes all of the hardware that were used in the project to construct the safety system. This chapter is divided into the following parts: the LD06 LiDAR sensors, the LED circuit, the 3D printed sensor stands, the CP2102 UART bridge and the Rapberry Pi 4. An image of the system setup can be seen in Figure 5.1.

5.1.1 3D Printed Sensor Stands

The final 3D-printed sensor stands can be seen in Figure 5.2. Although the stands were designed with good intentions, they had several shortcomings. The elevation difference between sensors was too small, leading to false intrusion detections from nearby sensors. For example, when sensor 1 scanned the same plane as sensor 2, sensor 2 sometimes interpreted laser light from sensor 1 as its own reflected signal, triggering false intrusions.

Furthermore, the four-legged stand design proved inadequate for ensuring sensor stability. In order for the sensors to be stabilized, all four legs needed to be perfectly level, which was not always the case as the floor was uneven in some places. On uneven surfaces, the sensor would wobble slightly due to the rotational movement of the LiDAR, leading to reduced confidence in the incoming data and offsets from the calibration values - ultimately causing false intrusion detections. A more stable and reliable alternative would be a three-legged stand in a triangular configuration, which offers better stability on uneven surfaces.

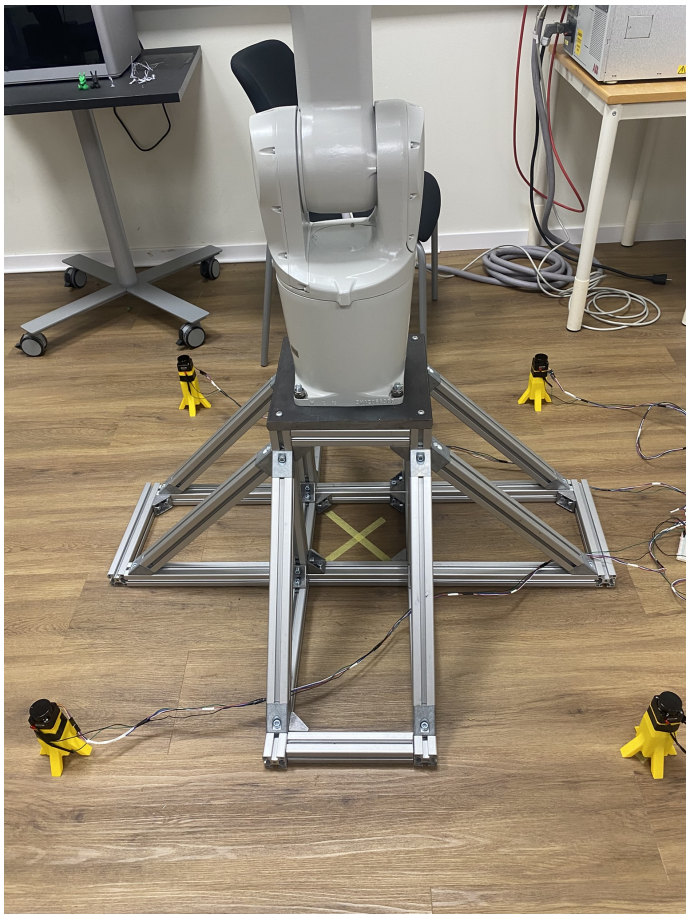

Figure 5.1: Image of system setup.

Figure 5.2: 3D printed stand with a sensor.

5.1.2 LED Circuit Functionality

The LED circuit showed stable and consistent behavior that matched the intended zone logic. Each LiDAR sensor was connected to a separate LED, which made it possible to clearly observe from which direction an object was detected. The LED response reflected the zone status LEDs blinked when an object entered the slow-zone and blinked rapidly when the stop-zone was breached.

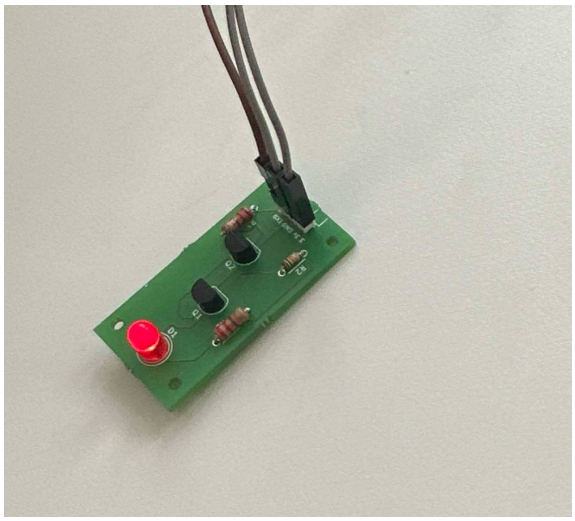

Figure 5.3: LED circuit used for zone indication.

5.1.3 Hardware Setup

The complete hardware setup is illustrated in the schematic in Figure 5.4. Each LiDAR sensor is connected to the Raspberry Pi via a CP2102 UART-to-USB bridge, enabling serial communication as well as providing power and ground, represented by the red lines in the figure. The LEDs are connected directly to the Raspberry Pi's GPIO pins. They share a common 3.3 V power source and ground, as indicated by the blue lines. Each LED also receives a separate UART signal from an individual GPIO pin on the Pi.

5.2 ROS 2 Workspace on the Raspberry Pi

A custom ROS 2 workspace was successfully created on the Raspberry Pi to manage the different components of the safety system. Within this workspace, a dedicated package named `virtual_fences` was developed, containing all core functionality of the system. This package included separate scripts for sensor identification, environment calibration, and real-time safety monitoring.

Each ROS 2 program developed in the project was executed using the `ros2 run` command within the ROS 2 workspace. The three main scripts, sensor identification, calibration, and safety system activation, were organized under the `virtual_fences` package and launched individually as follows:

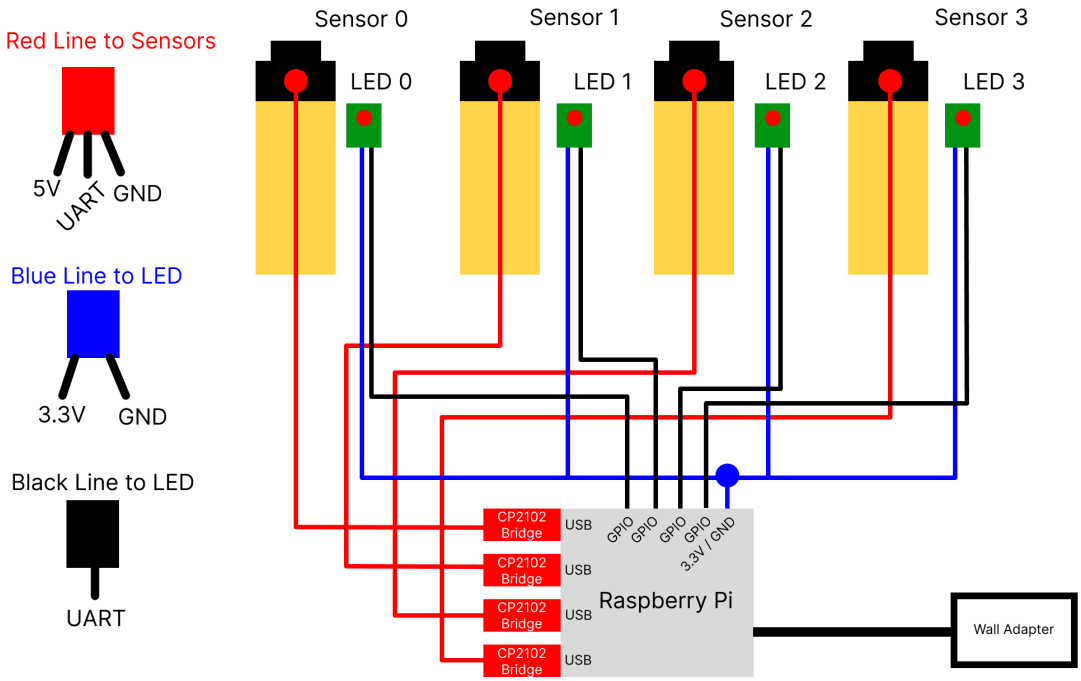

Figure 5.4: The final hardware setup schematic.

- **Sensor identification:** `ros2 run virtual_fences identify`
- **Calibration:** `ros2 run virtual_fences calibrate`
- **Safety system activation:** `ros2 run virtual_fences activate`

5.3 Start Up Sequence of the Safety System

To set up the system the sensors are manually placed and plugged into the Raspberry Pi. The sensors' IDs are then verified using the identification program, and the sensors' x- and y-position are measured from the origin (X under robot), as shown in Figure 5.1. When the setup is completed, the calibration can be initiated. With the calibration finished, the safety system can be activated.

5.3.1 Identification

The identification program is able to detect when an object is placed within 10 cm of a sensor, by for example waving a hand. When triggered, the program prints the corresponding sensor ID to the terminal as seen in Figure 5.5. This allows the user to match each physical sensor with its software reference, ensuring accurate sensor placement during the calibration process.

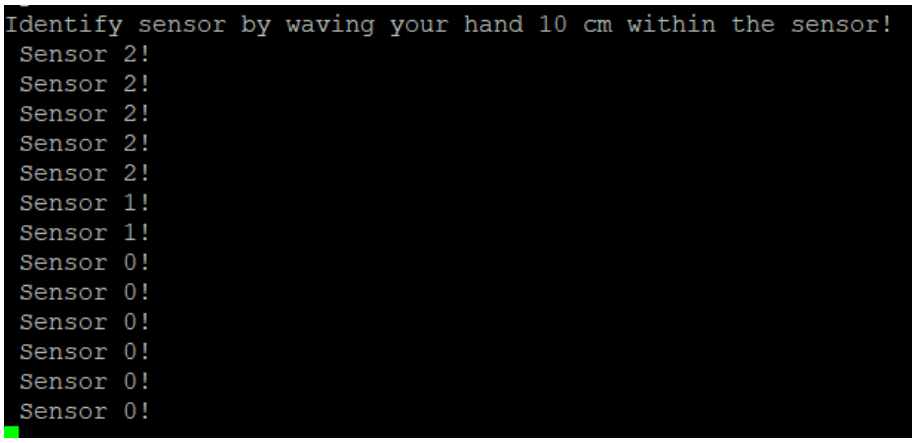

Figure 5.5: What is displayed in the terminal during identification. When the user waves it's hand in front of the sensor, it is displayed which sensor this is in the software.

5.3.2 Calibration

The calibration program generates a baseline 360.0 degree scan of the environment around each sensor relative to the robot's position. As the sensors were designed to be movable, it was necessary to manually define the position of each sensor in the robot's coordinate system. When the calibration script is started, the user is prompted to input the x- and y-coordinates of each sensor relative to the robot's base, which is defined as the origin (0,0) of a cartesian coordinate system.

After the positions are entered, the program collects distance measurements at a 0.1 degree resolution, resulting in 3,600 data points per sensor. The program runs the calibration in 30 second increments, until all 3,600 distance values are filled. This process is seen in the terminal in Figure 5.6.

The final result is a 360.0 degree distance profile for each sensor. This data is saved in a .csv file along with the sensor's position, and is then used as a reference during the real-time intrusion detection to identify changes in the environment.

5.3.3 Calibration test

The results of the calibration program test are summarized in Table 5.1 and the raw data can be seen in Appendix A. The maximal absolute difference between a calibrated value and a manually measured one is 36 mm, which is within the ranging accuracy of the sensors. The ranging accuracy for the LD06 sensor is typically $\pm$ 30 mm, with a maximum deviation of up to $\pm$ 45 mm [48]. The difference of 36 mm at sensor 2 is the only measured difference greater than 30 mm, the typical ranging accuracy. The average absolute difference is 14 mm which is less than 50 % of the typical ranging accuracy.

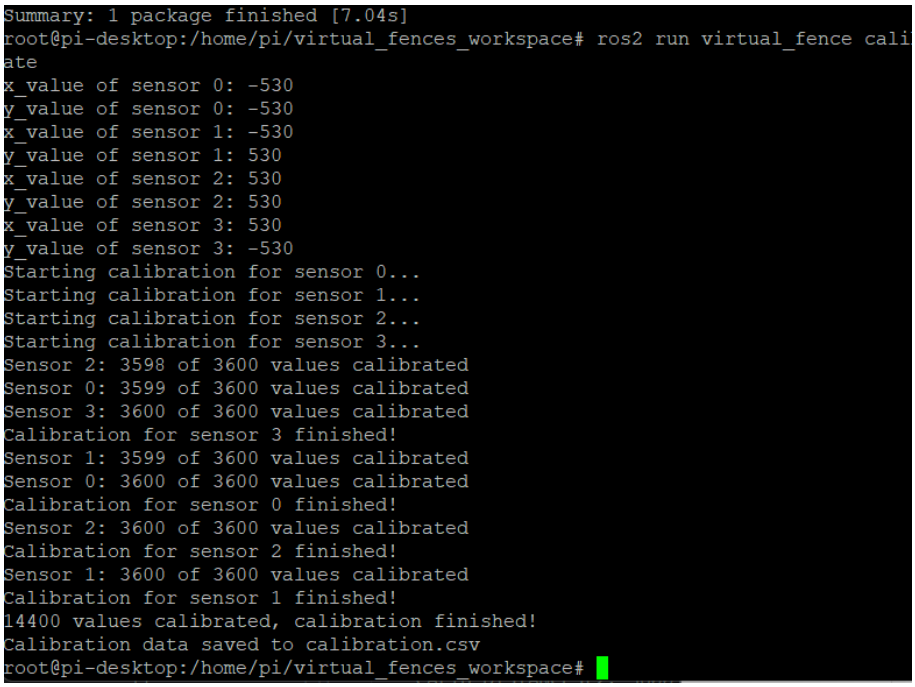

Figure 5.6: What is displayed in the terminal during calibration. The calibrated values are saved to a .csv file.

Table 5.1: Calculated average differences between the calibrated values and the measured distances at four angles for each sensors.

Sensor no.	Avg. diff. [mm]	Avg. abs. diff. [mm]	Max. abs. diff. [mm]
0	-3	8	17
1	-13	13	23
2	-22	22	36
3	-11	11	17
All sensors	-12	14	36

The results show that the only difference between the calibrated values and the manually measured values is within the accuracy of the sensors. Therefore, it cannot be shown that the differences come from anything other than the sensors accuracy. Thus, the calibration program functioned as expected.

5.3.4 Starting the Safety System

Once the system has been fully configured and calibrated, the safety system is activated by executing the following command within the ROS 2 workspace:

```
ros2 run virtual_fences activate
```

This command launches the main monitoring script, which begins real-time pro-

cessing of sensor data, intrusion detection, and communication with the robot and visualization tools.

5.4 Safety System

The safety system program serves as the core runtime component of the project, responsible for real-time monitoring of the robot's surroundings and detecting potential intrusions into predefined safety zones. Once initiated, the program loads calibrated measurements, then continuously recalculates incoming measurements to the robots perspective from all four sensors, and detects intrusions relative to the defined safety zones. The system visualizes intrusions using LEDs. When an intrusion was found within the stop-zone the system sends a stop-command to the robot and prompts the user that a stop intrusion was found. The system could also be stopped by using the emergency stop button. What was displayed during runtime can be seen in Figure 5.7.

```
Sensor 1: Slow zone breached, distance to object: 650 mm at angle 94 degrees, ti
me:09:25:47
Sensor 2: Slow zone breached, distance to object: 651 mm at angle 86 degrees, ti
me:09:25:48
Sensor 1: Slow zone breached, distance to object: 630 mm at angle 93 degrees, ti
me:09:25:48
Sensor 2: Slow zone breached, distance to object: 641 mm at angle 87 degrees, ti
me:09:25:48
Sensor 1: Slow zone breached, distance to object: 688 mm at angle 94 degrees, ti
me:09:25:48
Sensor 2: Slow zone breached, distance to object: 511 mm at angle 75 degrees, ti
me:09:25:48
Sensor 1: Slow zone breached, distance to object: 671 mm at angle 94 degrees, ti
me:09:25:48
Sensor 1: STOP ZONE BREACHED, distance to object: 485 mm at angle 82 degrees, ti
me:09:25:48
Sensor 3: STOP ZONE BREACHED, distance to object: 314 mm at angle 72 degrees, ti
me:09:25:48
Sensor 2: STOP ZONE BREACHED, distance to object: 434 mm at angle 71 degrees, ti
```

Figure 5.7: What is displayed in the terminal when the safety system is activated. The safety-zones have been adjusted to be smaller than calculated.

5.4.1 Testing Object Detection in Relation to Robot

The results of these tests are summarized in Table 5.2. In all cases, the system successfully identified the presence of an object in both the slow and stop zones. The system achieved a 100 % detection rate for both thresholds, indicating that the implemented logic for zone classification and object recognition operated reliably under the tested conditions.

The size of the stop-zone was determined to have a size of 2,500 mm, and the slow-zone extended 500 mm beyond that. During testing the stop zone was reduced to 80 % of its calculated value.

Table 5.2: Test for object detection within safety zones.

Test Number	Slow Zone Result (0/1)	Stop Zone Result (0/1)
1	1	1
2	1	1
3	1	1
4	1	1
5	1	1
6	1	1
7	1	1
8	1	1
9	1	1
10	1	1
Total Successes	**10/10**	**10/10**

5.4.2 Response Time Tests

The results of these tests are summarized in Table 5.3. When no sensor data was being published, the system demonstrated the fastest response, with an average time of 9.3 ms and low variability. Introducing sensor data publishing caused a slight increase in the average response time to 10.7 ms, while the median remained at 9.0 ms indicating occasional delays and increased variability in performance. Interestingly, removing both LED control and the emergency button resulted in a higher response time, averaging 23.2 ms.

The response time associated with the emergency button remained consistent across tests, averaging 14 ms both with and without sensor data publishing. This suggests that the emergency stop mechanism operates slightly slower than the rest of the safety system.

Table 5.3: Summary of response time tests.

Test Case	n	Avg (ms)	Med (ms)	Std Dev (ms)
wo. Sens. Data Publish.	10	9.3	9.0	3.3
w. Data Publish.	10	10.7	9.0	4.9
wo. Data Publish, LED, Emrg. Btn.	10	23.2	24.5	6.5
Emrg. Btn. wo. Data Publish.	10	14.0	14.5	2.4
Emrg. Btn. w. Data Publish.	10	14.0	11.0	6.6

5.4.3 Data Filtering Tests

The results of these tests are summarized in Table 5.4. When the stop zone was set to 1 meter and data publishing was enabled (Test Case 1), the system detected 270 false intrusions. In contrast, disabling data publishing under the same conditions (Test Case 2) completely eliminated false positives, highlighting the effect of more

concurrent processes impacted the system accuracy. A stripped-down configuration without data publishing, LED control, or emergency button (Test Case 5) also performed better than the full setup, though 12 false detections were still recorded.

Increasing the stop distance to 2 meters resulted in a notable rise in false detections across all configurations. With data publishing enabled (Test Case 3), 601 false intrusions were detected - more than double the number in the 1 meter case. Disabling data publishing (Test Case 4) reduced this number to 187, and removing all additional components (Test Case 6) brought it further down to 161.

These results suggest that extending the stop zone increases the likelihood of false detections. Additionally, system load, particularly sensor data publishing, appears to contribute significantly to false positives.

Table 5.4: False intrusions detected during filtering tests.

Test Case	Configuration	n, False Intrusions
1	Stop dist. 1 m, with data publishing	270
2	Stop dist. 1 m, without data publishing	0
5	Stop dist. 1 m, no publishing, no LED, no Emrg. Btn.	12
3	Stop dist. 2 m, with data publishing	601
4	Stop dist. 2 m, without data publishing	187
6	Stop dist. 2 m, no publishing, no LED, no Emrg. Btn	161

5.5 Simulation and Visualization

In this section the result of the simulation and visualization are presented. The processes of launching the robots environments, planning and executing motions, visualizing sensor data and handling safety breaches are all demonstrated with supporting figures and explained in the following subsections.

5.5.1 Launching Environments

Running the launch file will open the two robot environments in Gazebo and RViz2, displayed in Figure 5.8 and 5.9. The models, MoveIt, and the controllers will all load and be ready for planning.

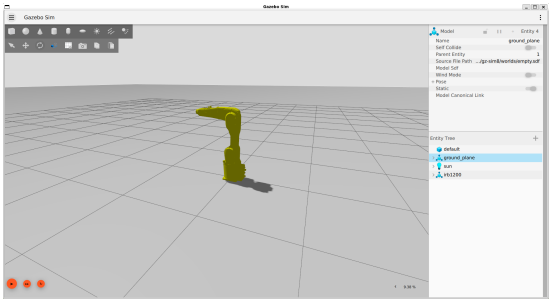

Figure 5.8: Simulated robot model in Gazebo

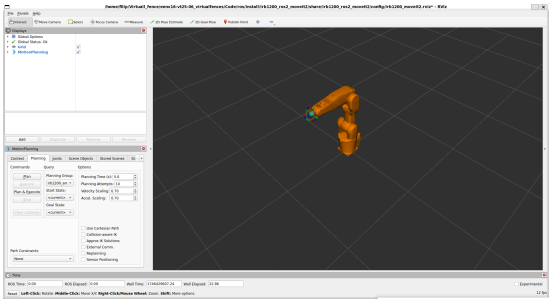

Figure 5.9: Visualized robot model in Rviz

5.5.2 Planning and Execution in RViz2

In the RViz2 motion planning interface a goal state is successfully set, planned and published which can be seen in Figure 5.10. The simulated robot in Gazebo successfully subscribes to the /robot_state_publisher and execute the motion, while RViz2 continuously updates the visualization of the robot's position displayed in Figure 5.11, 5.12 and 5.13.

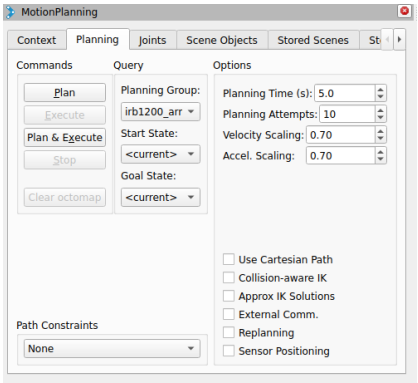

Figure 5.10: Motion planner interface in RViz2.

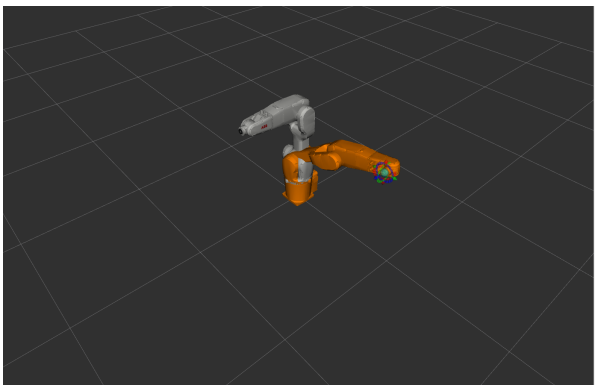

Figure 5.11: Displays robot before execution with the grey model being the current robot position and orange the selected goal state

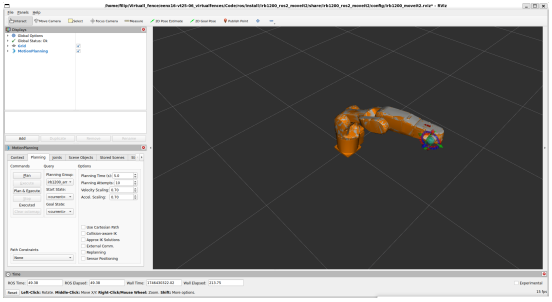

Figure 5.12: Visulized robot position in RViz2 after execution.

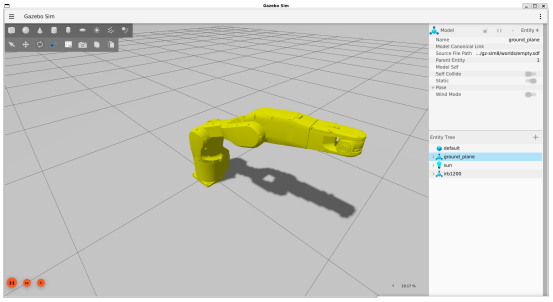

Figure 5.13: Simulated robot position in Gazebo after execution

5.5.3 Visualization of Laserscan Data

The sensor data was successfully visualized in RViz2 by adding the LaserScan display and subscribing it to the lidar_scan_all topic. The display setting ”increase decay time” was raised to ten seconds in order to create a mapping of the environment, depicted in Figure 5.14.

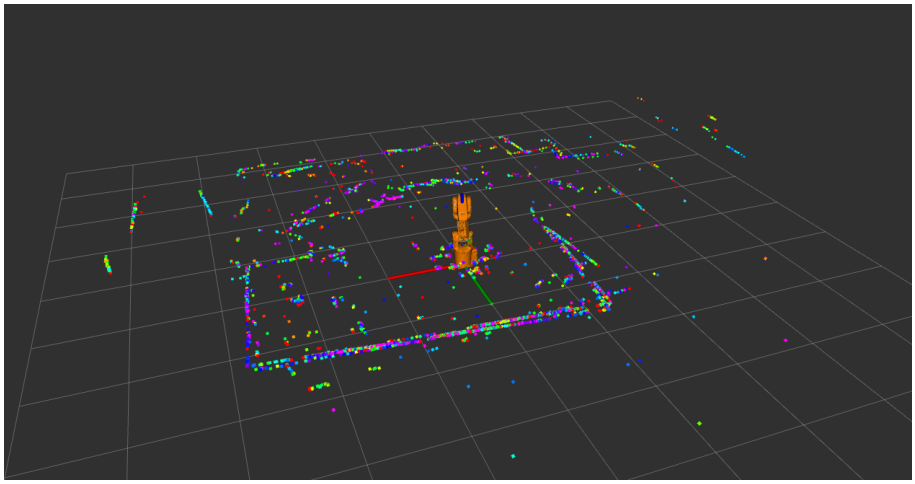

Figure 5.14: LiDAR sensor data visulized in RViz2 with ten seconds decay time.

5.5.4 Visualizing the Sensor Model

Only one sensor model was able to be displayed in RVivz, seen in Figure 5.15. Attempts to configure the tf2 tree for all four sensor were unsuccessful, resulting the visualization to a single sensor.

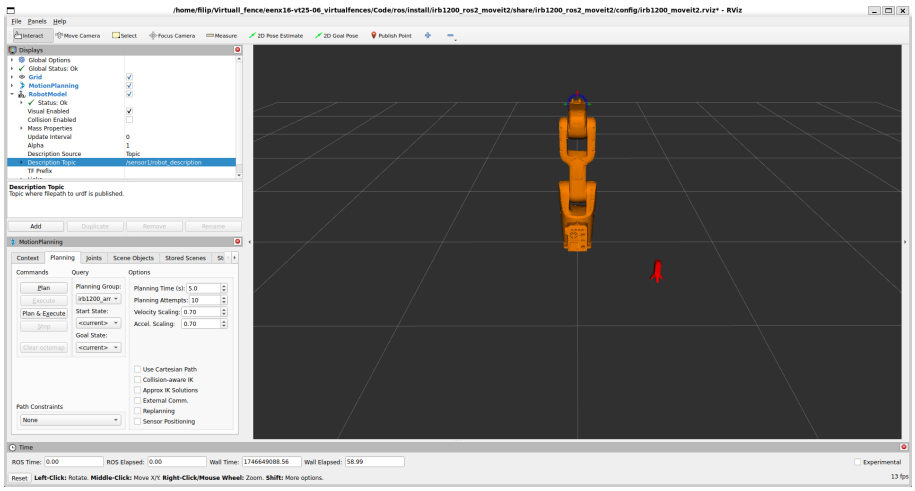

Figure 5.15: Sensor model visualized in Rviz2.

5.5.5 Sending Goal States to the Robot

Running the script ros2_actions and move_xyzw_client.py results in an action client being created and sending position data to the action server as seen in Figure 5.16.

```
[INFO] [1746634388.842231461] [move_xyzw_client]: Waiting for
action server…
[INFO] [1746634388.845242561] [move_xyzw_client]: Sending goal
: x=1.0, y=0.0, z=0.4
[INFO] [1746634402.402460619] [move_xyzw_client]: Result: Move
XYZW:SUCCESS
```

Figure 5.16: Action client output in terminal.

The action client successfully sends four repeating goal states resulting in the robot doing four distinct motions. The corresponding robot positions are shown in Figure 5.17, 5.18, 5.19 and 5.20.

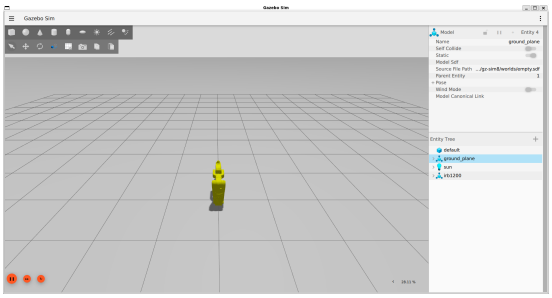

Figure 5.17: Robot position 1.

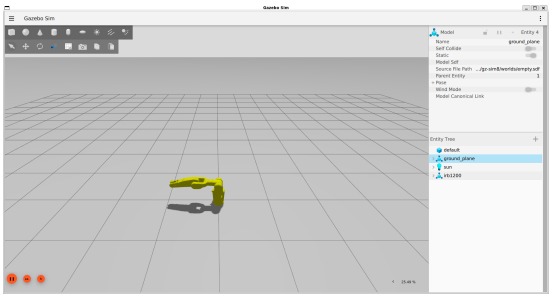

Figure 5.18: Robot position 2.

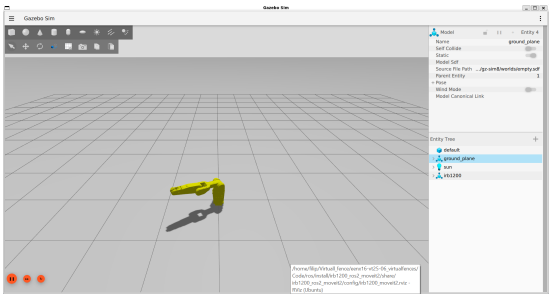

Figure 5.19: Robot position 3.

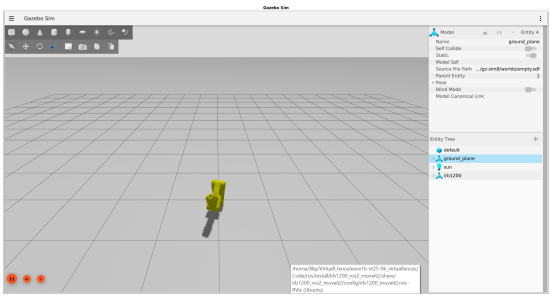

Figure 5.20: Robot position 4.

5.5.6 Stopping the Robot Execution During Zone Breach

During the execution of any plan, a /stop node subscribes to the /breach topic. If a ”1” is detected, indicating a stop zone breach, the joint states from the current trajectory are published again with zero velocity- effectively stopping the robot.

6

Discussion

This chapter reflects on the results of the project and evaluates the performance, limitations, and potential improvements of the developed safety system. The main objectives were to detect objects entering predefined safety zones around an industrial robot and to trigger appropriate responses based on their location. While the core functionality was achieved, several challenges related to system stability, sensor reliability, robot movements, clock configuration and hardware limitations emerged during testing. In the following subsections, key aspects of the system are analyzed in detail, including object detection logic, connection monitoring, performance bottlenecks, simulated robot movements, sensor models placement, clock implementation and suggestions for future improvements.

6.1 Hardware

The hardware used in the system played a significant role in its overall performance and reliability. This section discusses the capabilities and limitations of the Raspberry Pi and the LD06 LiDAR sensors, and how these components influenced system stability, responsiveness, and other features.

6.1.1 2D LiDAR Sensors

The sensors in this study are only able to scan their surroundings in one plane each. This entails that intrusions detected by the safety system are only detected in a 2D-plane at around 15 to 20 cm above the floor. The 2D-sensors were chosen since they were relatively cheap LiDARS which made it possible to make a relatively inexpensive safety system. This, however, also comes with some risks. If someone were to approach the robot, they would only be detected when their legs are in the stop-zone while they could be much closer with their arms or other parts of their body. If someone were to stand outside the stop-zone and then fall into it, it is possible that they would not be detected until their upper body is within the 2D-plane of the sensors. To mitigate this risk, when the radius of the stop-zone was calculated with Equation 4.5, a higher speed of the person approaching was used than what is typical under normal circumstances.

To increase the safety of the system it would be possible to place some additional

sensors in an another plane above the existing one. In that way, the system would still be relatively inexpensive with some added safety. However, this would not solve all of the problems associated with the sensors only measuring in one plane. Something or someone could pass between the two planes and not be detected, but if someone were to fall into the stop-zone they would probably be detected earlier. There might be some problem with the placement of these additional sensors. If they are placed to high the robot arm might move within the scanned plane which would produce an intrusion message with the way the system is built now. If they are placed to low the security would not be increased significantly.

To increase the safety even further, all the 2D-sensors could be replaced with a few cameras placed at different positions around the robot, creating a 3D safety zone. Then intrusions would be detected in all planes and someone stretching their arm towards the robot or someone falling into the stop-zone, would be detected. Because of this, it would probably be possible to decrease the radius for the stop-zone. This would in turn decrease the amount of stops of the robot since the operator would have to walk closer to the robot to stop it. Using cameras would however be more expensive than the 2D-senors used in this study and the safety system would be of higher cost.

6.1.2 Sensor Stands and Interference

It was found during testing that recurrent low-confidence readings were often caused by one of two reasons. Either the sensors were interfering with each other, which is described more in the next paragraph, or the sensor stand was unsteady. This unsteadiness, in most cases, was because of uneven flooring. When the sensor stand were placed in some areas of the floor, it would vibrate slightly since only three legs were in contact with the floor and the sensor is spinning inside the casing. This would give low confidence readings. In turn the calibration program would not finish, since there were not enough high confidence measurements to fill the .csv file. This problem could be solved by making sensor stands with three legs instead of four. Then it would always be stable since all legs would be in contact with the floor at all times. However, this problem of vibrations opened up for the potential problem that the floor might shake slightly when someone is moving around in the room or something heavy is placed down. This has not been noticed as a problem during testing but could become a problem when in use.

The sensors also returned faulty readings if the sensors interfered with each other. During testing, it was noticed that the LiDAR sensors occasionally interfered with each other when placed at the same height. To prevent this issue, the LiDAR sensors were positioned at varying heights by using objects to elevate them further, in addition to the height differences provided by the 3D-printed sensor stands. Since the sensors scan in a horizontal plane, placing them at slightly different vertical positions helped reduce the chance of overlapping scan fields. This arrangement appeared to improve stability and reduce the risk of unexpected readings. However, the height difference should have been larger than 1 cm for each sensors since they still interfered with each other on a few occasions.

6.1.3 LED Circuitry

Another important part of the development was the LED circuit, which went through some changes during the project. The original idea was to control the LED using the TXD (transmit data) pin from the USB interface. Since TXD normally stays at 3.3 V when idle (logic HIGH) and drops to 0 V during transmission (logic LOW), a two-transistor switching circuit was designed to invert the signal and control the LED based on data transmission activity. The goal was to make the LED turn on when data was being sent and off when nothing is happening.

However, during testing, this setup caused problems. The LED circuit was connected to the TXD pin of the CP2102 USB-to-serial converter, and the sensor was connected to the RXD (receive data) pin to receive data. This meant that the system was sending and receiving data at the same time, which caused problems. It affected the voltage levels and timing of the signal. This led to unstable sensor behavior, including corrupted data, missing packets. The issue became more noticeable when multiple sensors were connected at once, which made timing even more critical. After troubleshooting, it was discovered that the TXD pin of the UART bridge was not a reliable for controlling the LED in this type of system.

To fix the issue, the setup was modified so that the LEDs were controlled through a serial connection with the Pi's GPIO pins instead of the TXD pin from the USB adapter. GPIO is designed for digital control and is completely separate from the USB communication, so it worked without causing interference. The 3.3 V power came from the Raspberry Pi's power pin, and the GPIO pins were only used to switch the transistors. This reduced the load on the GPIO and made the circuit stable.

Technically, the two-transistor circuit was no longer needed when switching to GPIO. While it is possible to power a standard LED directly from a Raspberry Pi GPIO pin, this is not always the best solution. The GPIO pins can only supply up to 16 mA, which is just enough for basic LED functionality. But having the constructed PCB boards with transistors in place would however give better protection for the Raspberry Pi by reducing the risk of overcurrent and burn the GPIO Pins.

6.1.4 Housing for components

As of now, none of the components have any casing or housing which is not ideal during use or if developed further. The Raspberry Pi should be encased in some type of housing to protect the computer and its connections. Ideally this would be placed where it will not be stepped on or be in the way of the robot. This could, for example, be directly under the robot. The PCBs should also be placed in some type of housing to make sure that none of the components gets damaged. The cables should ideally be placed in some type of tubing for protection, and preferably be placed in the slots of the aluminum profiles to ensure that no one walking close risks falling because of them. Unfortunately none of these things were possible to complete within the given time frame of the project.

6.2 Software

The software for this project was developed in Python and ran on a Raspberry Pi using ROS 2 and this section discusses its capabilities and limitations. The section also discusses the simulation and visualization of the robot and areas of improvement regarding the software, such as changing the coding language and updating the calibration resolution.

6.2.1 Object Detection Logic

The results from the object detection test indicate that the system successfully detected objects and responded appropriately based on the predefined safety zones. During testing, the logic governing the slow- and stop- zones operated as intended in the majority of cases, with the system correctly transitioning between states when objects entered the monitored area. In all tests, an approaching person was consistently detected, confirming that the zone-based safety logic functioned as designed in principle.

However with the logic in place, a substantial amount of false intrusions were recorded as seen in Table 5.4. This suggests that more filtering needs to be done in order to avoid false positives and improve the system robustness.

6.2.2 Modularity of the Safety System

The system showed a high degree of flexibility when it came to repositioning the LiDAR sensors. Since each sensor is mounted on a separate 3D-printed stand and calibrated individually, it was easy to move a sensor to a new position and simply rerun the calibration process without needing to change anything in the code. This made it possible to adapt the system to different robot cells or layout changes with minimal effort, which is especially useful in industrial environments that may change over time.

Compared to the KUKA LBR iiwa robot study [9], which used two fixed laser scanners to monitor the area around the robot, the system built for this project offers a more flexible and adaptable solution for changing industrial environments. In that project, the scanners were mounted in predetermined positions to cover a 360 degree area and were not designed to be moved. Any change in the robot cell layout would require mechanical adjustments and recalibration of the scanning zones. In contrast, the LiDAR-based system built in this project allows each sensor to be placed freely using separate stands. If the layout changes, the sensors can be repositioned, and only the calibration needs to be updated—no code changes are required. This makes the solution easier to maintain, reuse, and adapt when the workspace changes.

6.2.3 Sensor Data Confidence and Filtering

Sensor readings were filtered by confidence, removing low confidence measurements. This step was necessary to reduce unreliable measurements, especially in cases where reflections or distance affected the signal quality. Filtering out these low-confidence values reduced intrusion messages that did not reflect reality. During testing, no loss of sensitivity was observed, although this had initially been thought to be a concern.

During most of the testing, the radius of the stop-zone was smaller than intended in the final version because of the limited size of the room where the robot was placed. The intended radius was 2.5 m from the center of the robot but during testing, a radius of maximally 2 m was used. When the radius is larger, more long-distance measurements will be used for intrusion warnings. Measurements at longer distances typically have lower confidences, and then there is a slight risk that valid detections will be filtered out. This, however, was assumed not to have a significant impact on the results since the testing done at 2 m was always able to detect an approaching human. There were more faulty detections though, which could infer that a more thorough filtering method is needed when the radius is increased.

6.2.4 System Performance and Instability

During testing, instability issues arose due to the use of multiple threads combined with the limited processing capacity of the Raspberry Pi. The system struggled to handle concurrent operations, resulting in increased sensor misreadings and incorrect interpretations, as shown in Table 5.4. These issues became more apparent as additional features were integrated into the program.

One of the most resource-intensive features was the publishing of all sensor data to ROS 2 topics, which ran in parallel with the core safety system. This functionality, intended solely for visualizing sensor data in RViz2, was not essential for the safety system itself. However, the visualization proved valuable for identifying faulty sensor readings. As shown in Figure 5.14, even high-confidence data points appeared outside of walls or in empty spaces, indicating errors. This highlights the need for improved filtering methods in future work to reduce false intrusions like those observed in the test.

Some of the system load issues can also be attributed to the use of Python. While Python offers rapid development and is well-suited for prototyping, it is not ideal for time-critical applications like a safety system. When multiple computationally intensive processes, such as transforming sensor data into the robot's coordinate frame, ran in parallel, the system's stability suffered. A more robust solution would have been to implement performance-critical components in C or C++. These languages offer better control over memory, timing, and thread management, and are more compatible with ROS 2, which is primarily built in C++. However, switching to a lower-level language would likely have slowed down development due to increased debugging and implementation complexity.

6.2.5 System Reaction Time

The response time tests show varying delays depending on system configuration and signal origin. The shortest average response times were observed in the test cases without sensor data publishing and without external signal triggers, with a mean response time of **9.3 ms** and a relatively low standard deviation of **3.3 ms**. This suggests a highly consistent and efficient response when the system operates in this configuration.

When data publishing was enabled, the average response time increased slightly to **10.7 ms**, accompanied by a larger standard deviation (**4.9 ms**). This indicates a modest performance cost associated with publishing data to the ROS 2 network, likely due to the additional processing or inter-process communication overhead.

The test case simulating a stop signal without data publishing, LED activation, or emergency button use showed a significantly higher mean response time of **23.2 ms**. Although still within acceptable limits, the increased delay further enforces the unpredictability of the system behavior. Even removing safety features interestingly increases the system latency, also suggesting that the use of a C++ complied program would increase the control of memory management and free up hardware resources.

Emergency stop button scenarios displayed consistent response times around **14 ms**. Both configurations with and without data publishing yielded the same average response, but the test *with publishing* showed a higher variability, **std dev 6.6 ms**, compared to **2.4 ms** when publishing was disabled. This reinforces the observation that system load from data publishing introduces latency variability, even if the average delay remains comparable.

6.2.6 Resolution of Sensor Data

During testing it was sometimes found that small objects that were present in the detection zones during calibration would still be detected as an intrusion during testing. This indicates that the calibration does not agree with the actual distances. These objects were things such as the cables connecting the sensors, the LEDs and the button with the Raspberry Pi. When the confidence threshold was increased for the calibration program it would take a longer time for the calibration.csv file to be filled. This suggests that these small objects resulted in low confidence readings and therefore had a higher risk of an inaccurate distance reading. This could also be a problem with the rounding of angles during the calibration process, but then probably when the object is placed further from the sensors as described below.

A similar problem would occur with some slightly larger objects, such as the thinner of the two cables connecting the robot with the robot controller. This occurred more often the longer the distance from the sensor, therefore it is more likely to be a problem with the rounding of the angles. The sensors measure every 0.01 degrees which is then rounded to 0.1 degrees during the calibration. At short distances this

is does not affect the spatial resolution notably. However, at larger distances this could have an impact at the results. Two meters from the sensors the resolution is 3.5 mm in the calibration instead of the 0.4 mm resolution that the sensors measure. At 3 meters this difference is 5.2 mm and 0.5 mm. In practice this means that an object that is 3 mm wide, placed 2 meters from a sensor could avoid detection during the calibration and then be detected as an intrusion when the virtual fence is activated.

To mitigate this issue the angles should not be rounded during the calibration. However, this did not seem feasible during this study due to existing timing and process issues within the program described above. During the planning stages of the project it was not thought that the rounding of angles would have an impact on the results and it proved time consuming to rewrite the program to accommodate additional calculations. This issue should be taken into account if the system is developed further in the future.

6.2.7 Connecting the Raspberry Pi to the Robot Simulation

A significant amount of time was spent trying to establish communication between the virtual machine and the Windows operating system. These efforts were ultimately unsuccessful due to the Windows firewall restricting most communication to the virtual machine. After repeated attempts to bridge the virtual machine with Windows, a dual-boot configuration with native Ubuntu 24.04 was implemented.

6.2.8 Connection Monitoring

In the current project, there is no built-in mechanism to verify whether the Raspberry Pi and the robot are connected. To ensure safe operation, a connection monitoring system must therefore be implemented. This can be achieved using some of ROS 2's Quality of Service (QoS) policies [60]. For instance, the `Deadline` policy can be used to detect if a publisher fails to send data within a specified time interval, providing a way to monitor the health of the communication link.

6.2.9 The Robot Fails to Iterate Over the Same Four Movements

As opposed to earlier versions, Jazzy requires you to define acceleration limits for each joint in a file called joint_limits.yaml. These limits were all given a default value of "1.0" m/s, which generally works, but sometimes results in a robot movement failure. It was decided not to spend time resolving this issue, as it occurs on the simulation's end and is not related to the functionality of the virtual fence. Simply running a plan that doesn't violate the joint limits is sufficient for our purpose.

6.2.10 Different Speeds Depending on Zones

For a full implementation of the safety system, the robot should change its movement speed depending on which zone is being breached. Currently, the robot only reacts to

a breach with stopping and publishing a joint trajectory with the current state zero velocity. Instead the robot should depending on the zone either publisher a lower joint trajectory speed or a zero trajectory speed. At the moment, values for different zone breaches are being published from the control node on the Raspberry Pi, but the simulated robot only reacts to the stop-zone breach. Further implementation is therefore needed.

6.2.11 Visualization of Sensor Models

After successfully implementing one sensor model in RViz2, an attempt to implement all four models was made. This was ultimately unsuccessful duo to conflicts when trying to create unique transform for every single model. This error was not investigated further duo to it not effecting the functionality of the overall project. Duo to only being able to display one sensor model the decision was made to not visualize any sensors in the final implementation.

7

Conclusion

The aim of this project was to design and evaluate a virtual safety system for an industrial robot using LiDAR sensors, zone logic, and modular software-hardware integration. The concept was based on replacing traditional physical safety fences with sensor-defined zones capable of identifying object intrusions in real time. The system was implemented using ROS 2 and a Raspberry Pi, and tested both in a simulation environment and through physical validation of output signals.

The project achieved its core objectives. The system successfully detected intrusions into both the slow- and stop-zone, and the responses followed the intended zone logic. The combination of LiDAR data processing, real-time evaluation, and visual feedback proved effective in creating a clear and responsive safety mechanism. The results confirmed that a system of this kind can contribute to safer human-robot interaction, especially in environments where flexibility and adaptability are required.

A major part of the work involved combining hardware and software into a functioning and synchronized system. On the hardware side, four 2D LiDAR sensors were used to scan the robot's surroundings in different directions, enabling detection coverage across multiple zones. The layout was modular and could be adjusted to fit different industrial scenarios. On the software side, the system was structured using a ROS 2-based architecture with dedicated nodes for data input, zone classification, and output. Simulation tools such as Gazebo and RViz2 played a key role in verifying the system's behavior and contributed to faster iteration during development.

Although the system performed reliably under test conditions, some limitations were observed. Most notably, the sensors only scanned in a single horizontal plane, making it difficult to detect objects approaching from above or below. Additionally, occasional false positives were observed due to sensor noise and angle limitations. Some issues were also observed with the robot's ability to consistently repeat its movements. These challenges highlight the need for further development in future versions, especially in the areas of filtering, vertical coverage, and real-time robustness.

In summary, the project demonstrates that virtual safety zones using LiDAR and

ROS 2 can provide a flexible and effective solution for enhancing industrial robot safety. The system's modular structure and adaptable software make it suitable for future improvements and integration in more complex collaborative environments. With additional refinement, this approach could serve as a viable alternative or complement to traditional safety measures in modern automation.

8

References

[1] B. Siciliano and O. Khatib, *Springer Handbook of Robotics*, 2nd ed. Cham, Switzerland: Springer, 2016.

[2] S. Haddadin, A. De Luca, and A. Albu-Schäffer, "Robot collisions: A survey on detection, isolation, and identification," *IEEE Transactions on Robotics*, vol. 33, no. 6, pp. 1292–1312, Dec. 2017. DOI: 10.1109/TRO.2017.2723903.

[3] ABB. "IRB 1200 – Industrial robot." ABB Robotics. (2023), [Online]. Available: https://new.abb.com/products/robotics/industrial-robots/irb-1200.

[4] F. Kitevski, W. Johansson, M. Adaan, K. Hotvedt, S. Kanon, and A. Abdullah, "Robot assistant for biopharmaceutical service: Integration of perception algorithms with robot controller," Bachelor's thesis, Department of Electrical Engineering, Chalmers University of Technology, 2024.

[5] Raspberry Pi Ltd. "Raspberry Pi 4 Model B Datasheet." (2025), [Online]. Available: https://datasheets.raspberrypi.com/rpi4/raspberry-pi-4-datasheet.pdf (visited on 2025-05-12).

[6] *ISO 12100:2010 – Safety of machinery – General principles for design – Risk assessment and risk reduction*, Standard, Geneva, Switzerland: International Organization for Standardization, 2010. [Online]. Available: https://www.iso.org/standard/51528.html (visited on 2025-05-04).

[7] *ISO 61508:2010 – Functional safety of electrical/electronic/programmable electronic safety-related systems*, Standard, Geneva, Switzerland: International Organization for Standardization, 2010. [Online]. Available: https://www.iso.org/standard/68387.html (visited on 2025-05-04).

[8] *ISO 10218-1:2011 – Robots and robotic devices – Safety requirements for industrial robots – Part 1: Robots*, Standard, Geneva, Switzerland: International Organization for Standardization, 2011. [Online]. Available: https://www.iso.org/standard/51330.html (visited on 2025-05-04).

[9] J. de Gea Fernández, D. Mronga, M. Günther, *et al.*, "Multimodal sensor-based whole-body control for human–robot collaboration in industrial settings," *Robotics and Autonomous Systems*, vol. 94, pp. 102–119, 2017, ISSN: 0921-8890. DOI: https://doi.org/10.1016/j.robot.2017.04.007.

[10] Universal Robots, *UR10e Collaborative Robot Arm*, 2025. [Online]. Available: https://www.universal-robots.com/products/ur10e/ (visited on 2025-04-29).

[11] T. Wan and B. Shi, "Virtual fence environment perception algorithm for industrial robots based on 3d vision," in *2023 42nd Chinese Control Conference (CCC)*, 2023, pp. 7412–7416. DOI: 10.23919/CCC58697.2023.10240273.

[12] Association for advancing automation robotics, *What are collaborative robots?* [Online]. Available: https://www.automate.org/robotics/cobots/what-are-collaborative-robots (visited on 2025-04-29).

[13] P. Karagiannis, N. Kousi, G. Michalos, *et al.*, "Adaptive speed and separation monitoring based on switching of safety zones for effective human robot collaboration," *Robotics and Computer-Integrated Manufacturing*, vol. 77, p. 102 361, 2022, ISSN: 0736-5845. DOI: https://doi.org/10.1016/j.rcim.2022.102361.

[14] A. Mohammed, B. Schmidt, and L. Wang, "Active collision avoidance for human–robot collaboration driven by vision sensors," *International Journal of Computer Integrated Manufacturing*, vol. 30, no. 9, pp. 970–980, 2017. DOI: 10.1080/0951192X.2016.1268269.

[15] D. Podgorelec, S. Uran, A. Nerat, *et al.*, "Lidar-based maintenance of a safe distance between a human and a robot arm," *Sensors*, vol. 23, no. 9, 2023, ISSN: 1424-8220. DOI: 10.3390/s23094305.

[16] KUKA Robotics. "Lbr iiwa." (2025), [Online]. Available: https://www.kuka.com/sv-se/produkter-tj%C3%A4nster/robotsystem/industrirobotar/lbr-iiwa (visited on 2025-04-21).

[17] C. Byner, B. Matthias, and H. Ding, "Dynamic speed and separation monitoring for collaborative robot applications – concepts and performance," *Robotics and Computer-Integrated Manufacturing*, vol. 58, pp. 239–252, 2019, ISSN: 0736-5845. DOI: https://doi.org/10.1016/j.rcim.2018.11.002.

[18] Leuze Electronic, *Rsl440-m/cu429-10 safety laser scanner*, 2024. [Online]. Available: https://www.leuze.com/en-se/rsl440-m-cu429-10/53800238 (visited on 2025-05-07).

[19] K. Qi, Z. Song, and J. S. Dai, "Safe physical human-robot interaction: A quasi whole-body sensing method based on novel laser-ranging sensor ring pairs," *Robotics and Computer-Integrated Manufacturing*, vol. 75, p. 102 280, 2022, ISSN: 0736-5845. DOI: https://doi.org/10.1016/j.rcim.2021.102280.

[20] ABB, *IRB 6700 Product Specification*, 2023. [Online]. Available: https://search.abb.com/library/Download.aspx?DocumentID=9AKK106103A6066&LanguageCode=en&DocumentPartId=&Action=Launch (visited on 2025-04-29).

[21] Nationalencyklopedin. "Lidar." (2025), [Online]. Available: https://www.ne.se/uppslagsverk/encyklopedi/l%C3%A5ng/lidar (visited on 2025-05-01).

[22] LDROBOT. “LD06 2D LiDAR Sensor Development Manual v1.0.” Online manual. (2020), [Online]. Available: https://storage.googleapis.com/mauser-public-images/prod_description_document/2021/315/8fcea7f5d479f4f4b71316d80b77ff45_096-6212_a.pdf.

[23] Open Robotics, *Ros - robot operating system*, 2021. [Online]. Available: https://www.ros.org/ (visited on 2025-05-11).

[24] Open Source Robotics Foundation. “Distributions.” ROS 2 Documentation. (2025), [Online]. Available: https://docs.ros.org/en/rolling/Releases.html (visited on 2025-05-06).

[25] O. S. R. Foundation, *Creating a workspace — ros 2 documentation: Jazzy*, 2025. [Online]. Available: https://docs.ros.org/en/jazzy/Tutorials/Beginner-Client-Libraries/Creating-A-Workspace/Creating-A-Workspace.html (visited on 2025-05-12).

[26] Open Source Robotics Foundation. “Understanding ROS 2 Nodes.” ROS 2 Documentation. (2025), [Online]. Available: https://docs.ros.org/en/foxy/Tutorials/Beginner-CLI-Tools/Understanding-ROS2-Nodes/Understanding-ROS2-Nodes.html (visited on 2025-05-06).

[27] Open Robotics. “Launch.” (2025), [Online]. Available: https://docs.ros.org/en/jazzy/Tutorials/Intermediate/Launch/Launch-Main.html (visited on 2025-05-09).

[28] Open Robotics. “Introduction to tf2.” (2025), [Online]. Available: https://docs.ros.org/en/jazzy/Tutorials/Intermediate/Tf2/Introduction-To-Tf2.html (visited on 2025-05-09).

[29] PickNik Robotics. “Moveit configuration.” (2025), [Online]. Available: https://moveit.picknik.ai/main/doc/how_to_guides/moveit_configuration/moveit_configuration_tutorial.html (visited on 2025-05-09).

[30] ros2_control Development Team, *Getting started*, 2025. [Online]. Available: https://control.ros.org/rolling/doc/getting_started/getting_started.html (visited on 2025-05-12).

[31] S. Simrock, *Control theory*, 2008. [Online]. Available: https://cds.cern.ch/record/1100534/files/p73.pdf (visited on 2025-05-12).

[32] P. J. Woolf, *Chemical Process Dynamics and Controls*, 1st ed. openmichigan, 2009. (visited on 2025-05-12).

[33] ros2_control Development Team, *Ros2_control documentation — jazzy*, 2025. [Online]. Available: https://control.ros.org/jazzy/index.html (visited on 2025-05-09).

[34] Open Robotics, *Rviz user guide*, 2025. [Online]. Available: https://docs.ros.org/en/jazzy/Tutorials/Intermediate/RViz/RViz-User-Guide/RViz-User-Guide.html (visited on 2025-05-09).

[35] Open Source Robotics Foundation. “Gazebo – Open Source Robotics Simulator.” (2025), [Online]. Available: https://gazebosim.org/home (visited on 2025-05-06).

[36] PickNik Robotics, *Moveit 2 documentation*, 2025. [Online]. Available: https://moveit.picknik.ai/main/index.html (visited on 2025-05-09).

[37] M. Contributors, *Ompl planner*, 2025. [Online]. Available: https://moveit.picknik.ai/main/doc/examples/ompl_interface/ompl_interface_tutorial.html (visited on 2025-05-12).

[38] Kavraki Lab, Rice University. "OMPL – Open Motion Planning Library." (2025), [Online]. Available: https://ompl.kavrakilab.org/ (visited on 2025-05-06).

[39] M. Contributors, *Kinematics configuration — moveit_tutorials kinetic documentation*, https://docs.ros.org/en/kinetic/api/moveit_tutorials/html/doc/kinematics_configuration/kinematics_configuration_tutorial.html, 2017. (visited on 2025-05-13).

[40] PickNik Robotics, *Motion planning — moveit documentation*, 2025. [Online]. Available: https://moveit.picknik.ai/main/doc/concepts/motion_planning.html (visited on 2025-05-09).

[41] T. Foote, *Clock and time — ros 2 design*, 2018. [Online]. Available: https://design.ros2.org/articles/clock_and_time.html (visited on 2025-05-13).

[42] PickNik Robotics, *Urdf and srdf tutorial*, 2025. [Online]. Available: https://moveit.picknik.ai/main/doc/examples/urdf_srdf/urdf_srdf_tutorial.html (visited on 2025-05-09).

[43] GitHub, *What is a programming language?* 2024. [Online]. Available: https://github.com/resources/articles/software-development/what-is-a-programming-language (visited on 2025-05-09).

[44] T. P. S. Foundation. "What is python? executive summary." (2025), [Online]. Available: https://www.python.org/doc/essays/blurb/ (visited on 2025-05-09).

[45] W3Schools. "Python introduction." (2025), [Online]. Available: https://www.w3schools.com/python/python_intro.asp (visited on 2025-05-09).

[46] GeeksforGeeks, *Introduction to c++ programming language*, 2025. [Online]. Available: https://www.geeksforgeeks.org/cpp-programming-intro/ (visited on 2025-05-09).

[47] GeeksforGeeks, *C++ programming language*, 2025. [Online]. Available: https://www.geeksforgeeks.org/c-plus-plus/ (visited on 2025-05-09).

[48] Inno-maker, *Ld06 lidar sensor datasheet*, Online, 2020. (visited on 2025-05-09).

[49] Canonical Ltd., *What is ros*. [Online]. Available: https://ubuntu.com/robotics/what-is-ros (visited on 2025-05-11).

[50] S. Tatham, *Putty: A free ssh and telnet client*, 2025. [Online]. Available: https://www.putty.org/ (visited on 2025-05-12).

[51] T. Kosse and contributors, *Filezilla: The free ftp solution*, 2025. [Online]. Available: https://filezilla-project.org/ (visited on 2025-05-12).

[52] Python Software Foundation, *Multiprocessing — process-based parallelism*, 2025. [Online]. Available: https://docs.python.org/3/library/multiprocessing.html (visited on 2025-05-12).

[53] M. Viola (https://math.stackexchange.com/users/218419/mark-viola), *Adding two polar vectors*, Mathematics Stack Exchange. [Online]. Available: https://math.stackexchange.com/q/1365938 (visited on 2025-04-04).

[54] *ISO 13855:2010 – Safety of machinery – Positioning of safeguards with respect to the approach speeds of parts of the human body*, Standard, Accessed: 2025-05-04, Geneva, Switzerland: International Organization for Standardization, 2010. [Online]. Available: https://www.iso.org/standard/53971.html.

[55] IFRA-Cranfield, *Ros2.0 robot simulation - ros2.0 foxy*, 2023. [Online]. Available: https://github.com/IFRA-Cranfield/ros2_RobotSimulation/tree/foxy/ (visited on 2025-05-12).

[56] W. M. Tully Foote Eitan Marder-Eppstein, *Tf2::transform class reference — ros documentation: Jade*, 2017. [Online]. Available: https://docs.ros.org/en/jade/api/tf2/html/classtf2_1_1Transform.html (visited on 2025-05-12).

[57] O. Robotics, *Robot_state_publisher*, 2020. [Online]. Available: https://wiki.ros.org/robot_state_publisher (visited on 2025-05-12).

[58] M. Contributors, *Planning adapter tutorials*, 2025. [Online]. Available: https://moveit.picknik.ai/main/doc/examples/planning_adapters/planning_adapters_tutorial.html (visited on 2025-05-12).

[59] P. Robotics, *Time parameterization — moveit documentation: Humble*, https://moveit.picknik.ai/humble/doc/examples/time_parameterization/time_parameterization_tutorial.html, 2025. (visited on 2025-05-13).

[60] Open Robotics, *About quality of service (qos) settings*, 2025. [Online]. Available: https://docs.ros.org/en/jazzy/Concepts/Intermediate/About-Quality-of-Service-Settings.html (visited on 2025-05-09).

A

Appendix A

The raw data from the testing of the calibration program can be seen in table A.1 together with some calculations of differences.

Table A.1: Testing of the calibration program. The difference is the calibrated distance subtracted by the measured distance.

	Sensor number			
	0	1	2	3
	(values in mm)			
0° calibrated	3196	3222	2173	2264
0° measured	3200	3215	2190	2260
Diff.	-4	7	-17	4
Abs. diff.	4	7	17	4
90° calibrated	1022	2111	1796	845
90° measured	1045	2120	1810	850
Diff.	-23	-9	-14	-5
Abs. diff.	23	9	14	5
180° calibrated	770	576	1796	1709
180° measured	800	585	1810	1745
Diff.	-30	-9	-14	-36
Abs. diff.	30	9	14	36
270° calibrated	1023	1025	1550	4481
270° measured	1035	1025	1565	4498
Diff.	-12	0	-15	-17
Abs. diff.	12	0	15	17

The raw data and some calculations for measured response times can be seen in Table A.2. Test case 1, 2 and 3 is the time from a detected intrusion by the sensors to a sent stop message. Test case 1 is with data publishing for visualization. Test case 2 is without data publishing and test case 3 is without data publishing and also without the LED control and the emergency button. Test case 4 and 5 is measuring the time from a signal from the emergency button to a sent stop message. Test case 4 is with data publishing and test case 5 is without.

Table A.2: Measured response times from data reception to sent stop message for test cases 1-5. Description of the test cases are provided in the text.

	Test Case				
	1	2	3	4	5
Test no.	(response times in ms)				
1	9	8	25	10	17
2	21	4	22	27	11
3	8	8	21	17	16
4	8	16	23	11	16
5	12	10	25	9	14
6	14	12	25	11	13
7	15	7	24	24	12
8	5	11	33	8	15
9	9	10	29	10	16
10	6	7	22	13	10
Average	10.7	9.3	24.9	14	14
Median	9	9	24.5	11	14.5
Std. Dev.	4.9	3.3	3.6	6.6	2.4

DEPARTMENT OF ELECTRICAL ENGINEERING
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden
www.chalmers.se

CHALMERS
UNIVERSITY OF TECHNOLOGY