School of Engineering
CAI Centre for Artificial Intelligence

RoboDog III: Building a Vision and/or Sound-Based AI Demonstrator on a Robotic Platform

Bachelor's Thesis

Authors	Juri Pfammatter Daniel Schweizer
Supervision	Prof. Dr. Thilo Stadelmann Pascal Sager
Date	09.06.2023

Zürcher Hochschule
für Angewandte Wissenschaften

zh
aw School of Engineering

DECLARATION OF ORIGINALITY

Bachelor’s Thesis at the School of Engineering

DECLARATION OF ORIGINALITY
Bachelor’s Thesis at the School of Engineering

City, Date:	**Name Student:**	
Zürich, 08.06.2023	Juri Pfammatter	[illegible]
Rafz, 08.06.2023	Daniel Schweizer	[illegible]

Acknowledgments

We would like to thank Prof. Thilo Stadelmann and Pascal Sager of the Centre for Artificial Intelligence (CAI) for their guidance during this thesis. We are especially grateful for their trust in our vision and ideas and the support we have received. Additionally, we would like to thank Marcel Wegmann of the Institute of Embedded Systems (INES) for sharing his insight on embedded hardware.

Abstract

With technological advancements in artificial intelligence (AI) and the resulting prevalent position of this topic outside of academia, the demand to present their work to a broad audience has increased in institutions conducting research in AI. For example, students at the Centre of Artificial Intelligence (CAI) at the Zurich University of Applied Sciences (ZHAW) developed an AI demonstrator using a Unitree A1 quadrupedal robot to recognize hand gestures and perform corresponding actions. The Bachelor's thesis at hand integrates this existing gesture recognition into a modular control system based on the Robot Operating System (ROS) and visualizes the main algorithms with a graphical user interface (GUI) and the ROS tool RViz. The robot is equipped with an external ZED2 stereo camera and an Ouster OS1 light detection and ranging (LiDAR) sensor, improving the robot's environmental perception to localize itself and avoid collisions. Moreover, a Kalman filter fuses stereo camera data and inertial measurements, combining relative and absolute state estimations. A LiDAR-based map generated by a simultaneous localization and mapping (SLAM) algorithm corrects this sensor-fusion-based state estimation. The software modules that provide this functionality to achieve the aforementioned research goals were implemented in C++ and Python, employing the corresponding ROS libraries. The developed position controller enables the robot to move on a path with a maximal deviation of 0.2 m when walking on a 2 m straight line. During this motion, the collision avoidance system reliably prevents collisions caused by the robot. Furthermore, the robot is already successfully used for live demonstrations at public events to enhance the understanding of AI. The designed software platform integrates ROS-based position control and collision avoidance and provides a modular base for future projects on the utilized robotic platform.

Keywords: Artificial Intelligence, Robot Operating System ROS, Gesture Recognition, Sensor Fusion, State Estimation, LiDAR

Zusammenfassung

Mit den technologischen Fortschritten im Bereich der künstlichen Intelligenz (KI) und der daraus resultierenden Verbreitung des Themas ausserhalb der Wissenschaft, ist an Institutionen, die im Bereich der KI forschen, der Bedarf gestiegen, ihre Arbeit einem breiten Publikum zu präsentieren. Folglich haben Studierende des Zentrums für Künstliche Intelligenz (CAI) der Zürcher Hochschule für Angewandte Wissenschaften (ZHAW) einen KI-Demonstrator entwickelt, der unter Verwendung eines vierbeinigen Unitree A1-Laufroboters Handgesten erkennt und entsprechende Aktionen ausführt. Die vorliegende Bachelorarbeit integriert diese bestehende Gestenerkennung in ein modulares Steuerungssystem auf der Basis des Robot Operating System (ROS) und visualisiert die wichtigsten Algorithmen mit einer grafischen Benutzeroberfläche (GUI) und dem ROS-Tool RViz. Der Roboter ist mit einer externen ZED2-Stereokamera und einem Ouster OS1 Light Detection and Ranging (LiDAR)-Sensor ausgestattet, der die Umgebungswahrnehmung des Roboters verbessert, um eine Lokalisierung zu ermöglichen und Kollisionen zu vermeiden. Darüber hinaus kombiniert ein Kalman-Filter Stereokameradaten und inertiale Messungen, um relative und absolute Zustandsschätzungen zu generieren. Eine LiDAR-basierte Karte, die durch einen Simultaneous Localization and Mapping (SLAM)-Algorithmus erstellt wird, korrigiert diese auf der Sensorfusion basierende Zustandsschätzung. Die Softwaremodule, die diese Funktionalität bereitstellen, wurden in C++ und Python unter Verwendung der entsprechenden ROS-Bibliotheken implementiert. Der entwickelte Positionsregler ermöglicht es dem Roboter, sich auf einer Bahn mit einer maximalen Abweichung von 0.2 m zu aufzuhalten, wenn eine Bewegung entlang einer 2 m langen Geraden ausgeführt wird. Während dieser Bewegung verhindert das Kollisionsvermeidungssystem zuverlässig durch den Roboter verursachte Kollisionen. Darüber hinaus wird der Roboter bereits erfolgreich für Live-Demonstrationen bei öffentlichen Veranstaltungen eingesetzt, um das Verständnis für KI zu fördern. Die entwickelte Softwareplattform integriert eine ROS-basierte Positionsregelung sowie eine Kollisionsvermeidung und bietet eine modulare Basis für zukünftige Projekte auf der verwendeten Roboterplattform.

List of Symbols

Term	**Definition**	**Unit**
d_b, d_f, d_l, d_r	Size of collision zone on each side of the robot	mm
e_x, e_y	Error in x- and y-axis	m
k_x, k_y	Controller gain for the linear x- and y-axis velocity	$\frac{1}{\mathrm{s}}$
k_α	Controller gain for the angular velocity	$\frac{1}{\mathrm{s}}$
qx, qy, qz, qw	Orientation in quaternion notation	-
res_h	Measurements per horizontal slice	-
x	Position in x-axis	m
$\dot{x}$	Velocity in x-axis	$\frac{\mathrm{m}}{\mathrm{s}}$
y	Position in y-axis	m
$\dot{y}$	Velocity in y-axis	$\frac{\mathrm{m}}{\mathrm{s}}$
z	Position in z-axis	m
$\dot{z}$	Velocity in z-axis	$\frac{\mathrm{m}}{\mathrm{s}}$
α	Z-angle of the trajectory	rad
α_{robot}	Z-angle of the robot	rad
α_{goal}	Z-angle goal of the robot	rad
$\dot{\alpha}$	Z-angle velocity	$\frac{\mathrm{rad}}{\mathrm{s}}$
δ	Z-angle between robot and goal angle	rad
γ	Z-angle between robot and trajectory	rad
$\varphi_1, \varphi_2, \varphi_3, \varphi_4$	Azimuths of corners of the collision zone	rad

List of Abbreviations

Term	Definition
AI (*KI*)	Artificial intelligence (*German*)
API	Application programming interface
CAI	Centre for Artificial Intelligence
(C)CW	(Counter) Clockwise
EKF	Extended Kalman filter
ETH	Swiss Federal Institute of Technology
FSM	Finite State Machine
GPU	Graphics processing unit
GT	Georgia Institute of Technology
GUI	Graphical user interface
IMU	Inertial measurement unit
JSON	JavaScript Object Notation
LCM (server)	Lightweight Communications and Marshalling (server)
LiDAR	Light detection and ranging
MIT	Massachusetts Institute of Technology
ML	Machine learning
MPC	Model predictive control
OS1	Ouster OS1 32 Gen 2 LiDAR device
PC	Personal Computer
qre	Quadruped (software project name)
rbd	Robodog (software project name)
RGB	Red, green and blue
ROS	Robot Operating System
RRT	Rapidly exploring random tree
RSL	Robotic Systems Lab
RTOS	Real-time operating system
SATW	Swiss Academy of Engineering Sciences
SDK	Software development kit
SLAM	Simultaneous localization and mapping
SSH	Secure Shell
UC	University of California
UDP	User Datagram Protocol
USC	University of Southern California
WLAN	Wireless Local Area Network
ZHAW	Zurich University of Applied Sciences

Used ROS Messages and Services

For clarity, these ROS messages are mentioned in the following thesis only by their name. For some of them, fields with no relevance to their understanding have been removed.

Type	Name	Description	Definition	Package
Message	IMU	Inertial measurement unit	Orientation (Quaternion), angular velocity (x, y, z), linear acceleration (x, y, z)	`sensor_msgs`
	PointCloud2	LiDAR measurements	Array of point measurements	`sensor_msgs`
	Pose	Positional state	Point (x, y, z), quaternion (x, y, z, w)	`geometry_msgs`
	String	String	String	`std_msgs`
	Twist	Linear and angular velocity	Linear velocity (x, y, z), angular velocity (x, y, z)	`geometry_msgs`
Service	GeneratePath	Service for path generation	**Request**: Command(string) **Response**: Poses ([pose]), nr. of poses	`rbd_msgs`
	GetLastGesture	Service for receiving the newest gesture	**Request**: None **Response**: Command(string)	`rbd_msgs`
	SetPosition	Service for setting the goal pose	**Request**: Pose **Response**: None	`rbd_msgs`
	SetBodyPose	Service for setting a body pose	**Request**: Roll, pitch, yaw, body height **Response**: None	`qre_msgs`
	SetBool	Service for setting a bool	**Request**: Bool **Response**: Bool	`std_srvs`

Contents

1 Introduction

1.1 Motivation

While concepts of legged robots have been developed as early as the late 1960s, it has taken decades for technology to reach a level where mobile robots capable of executing adaptive autonomous motion could be created. One of the first examples of a robot capable of reacting to its environment is the six-legged robot Genghis, developed in the early 1990s by Rodney Brooks at the Massachusetts Institute of Technology (MIT) [1]. These first-generation legged robots were mostly controlled by heuristic or model-based controllers where the control policy is programmed by human intelligence.

With advancements in technology and the development of artificial intelligence (AI), research in modern quadrupedal robots mostly utilizes some form of machine learning in their control system [2]–[4]. One of the most advanced commercially available examples is the four-legged inspection robot ANYmal, an industrial-grade system developed by the Swiss company ANYbotics based on years of research at the Robotics Systems Lab (RSL) at the Swiss Federal Institute of Technology (ETH) Zurich [5]. Contrary to classical robotics, the research conducted at ETH Zurich focuses on deep reinforcement learning in combination with model-based control policies [6]. These robots are able to learn adaptive gait patterns to improve motion on uneven terrain, enhancing adaptability to complex and dynamic environments like inspection sites or disaster zones [4].

However, the robots available in today's market mainly utilize artificial intelligence to improve robustness and productivity. The underlying algorithms are mostly hidden from consumers since they are part of the proprietary software and not of interest to the user. Nevertheless, it can be insightful to reveal the inner workings to improve the understanding of AI by the broad public.

The Centre for Artificial Intelligence (CAI) of the Zurich University of Applied Sciences (ZHAW) is using a Unitree A1 quadrupedal robot as a communication tool to narrow the gap between humans and machines by developing a robotic system that demonstrates the capabilities of AI. In a prior project thesis, the robot was used to communicate vision-based methods of AI by capturing hand gestures with an integrated camera and classifying them accordingly [7]. Gestures are classified based on detected hand key points, which are visualized on an external screen to increase the comprehensibility of the information generated by the machine learning (ML) algorithm. These key points are then added to a sequence and processed by another ML model. The detected gesture is also visualized on the external screen. Additionally, the robot performs a corresponding action for each identified gesture.

Since this proof of concept executes a sequential C++ script once a gesture is recognized, the motion cannot be altered depending on environmental influences, thus restraining it in terms of safety and responsiveness. Formulated differently, the robot executes a predefined motion without being able to interrupt it, eventually resulting in collisions with obstacles or people.

This thesis aims to build a responsive and expandable robotic software platform serving as a base for future projects conducted at the CAI to demonstrate the concept of AI. While maintaining the current gesture recognition functionality, the robot is extended with safety features and additional sensor hardware. One key step to achieving safe and reactive operations is the parallelization of individual software components. Soft real-time operating systems (RTOS) utilizing Robot Operating System (ROS) allow for the parallel execution of multiple tasks, including emergency stops and collision avoidance protocols. These improvements lower the safety distance required at social events, diversifying possible use cases. Additionally, the responsiveness of the robot enhances the impression of the demonstration.

1.2 Problem Statement and Contribution

The goal of this thesis is to develop a universally usable AI demonstrator to introduce the capabilities and functionality of AI to a broader audience. To showcase this demonstrator at public events like trade fairs, safety is the highest priority. With the existing demonstrator software, the responsiveness is limited due to a sequential C++ script, which does not consider environmental factors like objects or people. Consequently, the underlying software structure has to be reevaluated and fundamentally changed.

These changes should future-proof the software stack using a modular, ROS-based control system, allowing the replacement or expansion of individual software elements. However, the already implemented functionality of gesture recognition and visualization has to remain. Locomotion in space is realized by the development of a localization system as well as a position controller. With additional sensor hardware in the form of a light detection and ranging (LiDAR) sensor and a stereo camera, environmental perception can be enhanced, preventing collisions in advance. The existing visualization of the gesture recognition algorithm is integrated and supplemented by a graphical representation of the collision avoidance system. Furthermore, the extended graphical user interface (GUI) features an emergency stop button and a preview of future movements.

Based on the mentioned goals and requirements, the following research question including three sub-questions is answered in this report:

How can the existing Unitree A1 quadrupedal robotic AI demonstrator be improved to implement a safer and more responsive behavior during interaction with humans?

i. Based on the already implemented gesture recognition and sequential command execution, can ROS be used to create a responsive modular control system?

ii. Can a LiDAR sensor be integrated to create a collision detection system?

iii. Based on existing visualization tools, can the inner workings be further visualized to enhance the comprehensibility of the system?

1.3 Outline

The remainder of this thesis is structured as follows: In Chapter 2, some background of relevant technical topics needed for this work is illuminated, encompassing both, software and hardware components. Chapter 3 outlines the implementation of the control system, the collision avoidance, and the visualizations, followed by the achieved performance results and their discussion in Chapter 4. Chapter 5 interprets the work and gives an outlook for possible future work based on this project.

2 Foundations

2.1 Related Work

2.1.1 Preceding Project Thesis

Langdun et. al. developed an interaction-based AI demonstrator at the ZHAW CAI intending to demystify AI, familiarize people with the topic, and entertainingly convey knowledge about it. To achieve this, they used a Unitree A1 legged robot to recognize four specific hand gestures via the integrated camera and execute a corresponding action for each identified gesture. Additionally, they developed a GUI to efficiently operate the system and to visualize the classification algorithm and finally presented the prototype to a broad audience at a fair.

The system was designed to run the main algorithms on a host PC and use the robot's integrated computers to run hardware drivers, with these two subsystems communicating via a Secure Shell Protocol (SSH) connection and a WLAN hotspot provided by the robot. The authors employed a computer vision-based gesture recognition technology to avoid the use of additional special equipment, such as Marker Gloves. They analyzed three pre-trained, state-of-the-art models for pose estimation, of which they determined the hand-tracking system MediaPipe Hands to be the most appropriate one for the given problem.

To identify gestures in video sequences, the authors evaluated various machine learning classifier algorithms. They utilized 600 samples for each of the 4 hand gestures to be recognized and 1368 random motion patterns, resulting in a total number of 3768 samples for training the models. Finally, a logistic regression classifier has proved itself to be the most appropriate one as it provided the best ratio of accuracy and classification time. The resulting model allowed examining a queue of hand key point coordinates detected in a sequence of video frames for the occurrence of a certain hand gesture to obtain a corresponding output.

Langdun et. al. illustrated the gesture classification process in a video stream showing the recording of the camera integrated at the anterior side of the robot. To visualize the AI approach, all hand key points detected by the MediaPipe model were displayed, which allowed the audience to obtain a better understanding of how the algorithm analyzes their hand gestures. Additionally, in the case of a detected gesture, the classification output as well as the corresponding confidence value were visible in the video stream.

The aforementioned functionality could also be achieved using only a notebook and a camera. However, to achieve a more captivating demonstrator for AI, Langdun et. al. programmed the Unitree A1 robot to execute a corresponding action for each recognized gesture. The authors employed sequential scripts that use the Unitree Legged Software Development Kit (SDK) to apply the required velocities in each axis to the robot.

Furthermore, they have chosen a Python multiprocessing approach to implement the interaction of the three encapsulated software components, namely the GUI, the gesture recognition, and the command executor. Consequently, these processes ran in parallel and were prevented from blocking each other. This approach furthermore enhanced the system's modularity, as lean external interfaces were required to communicate with other processes via the WebSocket protocol.

As the execution of the sequential movement scripts could not be interrupted or influenced by an external source or by software, a safety distance of 2 m to the robot was crucial. The authors furthermore delivered a series of additional suggestions to improve the demonstrator, such as enhancing the MediaPipe pose estimation or implementing a 360° gesture recognition via a LiDAR sensor. For all suggestions, the recommended ROS as a development platform to enhance the software's scalability and implement more complex robot behavior [7].

2.1.2 Research Conducted on Unitree A1

Due to its availability at an affordable price, the Unitree A1 quadrupedal robot, as used in this thesis, serves as a base for a broad variety of research projects. A number of them focus on gait and motion control based on reinforcement learning. For example, Bellegarda et. al. from the University of Southern California (USC) have used deep reinforcement learning to improve the robustness of the robot's motion, resulting in the ability to carry up to 6 kg of additional load while moving at 2 $\frac{\mathrm{m}}{\mathrm{s}}$ [8]. At lower speeds, they were even able to bear up to 11 kg of additional load (92% of the robot's weight) [9].

Peng et. al. from Google Research and the University of California (UC), have used a learning-based approach to mimic motion captured from real animals. Through model predictive control (MPC), where a model of the robot is used to predict future states of the robot, they were able to generate motion patterns that allow the Unitree A1 robot to execute this recorded motion, resulting in a more natural motion [10].

Others have focused their research on localization while using the available motion controller of the Unitree A1. Chen and Dellaert of the Georgia Institute of Technology (GT) have developed a ROS-based simultaneous localization and mapping (SLAM) package, allowing localization without an a priori map of the environment [11]. Contrary to the widely used ROS package SLAM Toolbox developed by Macenski et. al. [12], [13], the A1 SLAM package is optimized for the *aggressive motion* experienced by the Unitree A1 [11].

The control systems of the quadrupedal robot ANYmal, developed by Hutter et. al, are based on ROS [14]. In ANYmal, the control system is split up into three subsystems responsible for Locomotion, Navigation, and Inspection, each running ROS on an individual PC. This separation allows for real-time operations on one PC while less critical tasks like navigation and inspection are outsourced to the other two PCs. This ensures that errors in the mentioned two subsystems cannot interfere with critical locomotion operations.

A similar separation of hardware can be found on the Unitree A1, where tasks are split up on two Ubuntu-based computers (a Raspberry Pi and an NVIDIA Jetson board) as well as a controller board. Hard real-time tasks are executed on the controller board, only connected to the Ubuntu-based computers through an Ethernet connection. The Raspberry Pi is responsible for the soft real-time operation of the Unitree SDK while the NVIDIA Jetson board can be used for user programs like ROS and sensor interfaces [15].

2.2 Unitree Legged SDK

Out of the box, the Unitree A1 quadrupedal robot is equipped with an SDK which offers both low-level and high-level control. The SDK incorporates C++ libraries to enable direct real-time control of the robot [16].

The high-level interface offers the ability to switch between three modes: *standing*, *walking*, and *sport* mode. For both the walking and sport mode, a set of linear and angular velocity specifications can be programmed, which are then executed by the robot. In addition to the linear and angular velocities, the orientation of the robot body can be chosen by its roll, pitch, and yaw angles. This allows for control of the robot with minimal interference to the gait patterns and locomotion policies.

Should a more fundamental approach be chosen, the low-level control can be used to control the velocity, position, and torque of each of the 12 joints individually. Therefore the gait movement can be optimized for special use cases. Additionally, the SDK offers measurements of the four pressure sensors, one in each foot, which can help to determine their contact with the ground.

This SDK is also utilized by the Unitree joystick controller, which allows for the manual movement of the robot. It has to be mentioned that the controller itself simply transmits the input velocity to the SDK, which then controls the robot. Since the SDK provides no implementation of collision detection or stability control, the robot often performs movements exceeding its dynamic abilities, possibly leading to crashes and damage to the robot itself. Even though the robot is equipped with both an RGB and a depth camera, it cannot detect obstacles, raising the risk of collision with both static and dynamic obstacles [15].

Moreover, the C++ interface of the SDK is achieved over structs - a custom-named group of several related variables - instead of function calls. The forward velocity for example can be set by setting the `forwardSpeed` variable to a value between -1 and 1.

In the previous project thesis, the Unitree SDK has been used to program pre-defined movement sequences which are executed depending on the given command [7]. This approach has the drawback of not being able to react to changing circumstances. For example, if an obstacle is placed in front of the robot, it is not detected and therefore leads to a collision. Additionally, functions like gesture recognition and emergency stops are blocked by the running script. To overcome this, a responsive real-time system consisting of path planning, position control, and collision avoidance has to be implemented.

2.3 ROS

There are several ways to implement the aforementioned functions needed for a responsive real-time system. ROS, a set of open-source software frameworks and libraries, has established itself as the industry standard for scientific robotics prototyping [17]. It can even be found in the ANYmal robot - a commercial-grade autonomous inspection robot for rugged terrains [5].

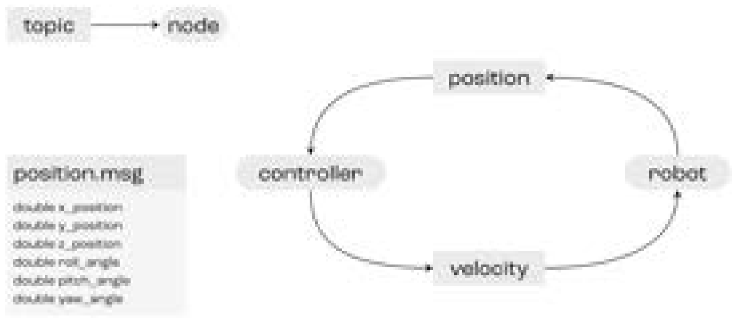
2.3.1 Basic Principle of Communication

ROS is a message-based communication network where individual nodes can subscribe to and publish messages on a topic to communicate with each other. Each node - often also called a package - is a self-contained C++ or Python program which offers functionality in one specific area. Through messages and services, they can access information shared with other nodes. For a broad variety of use cases, ranging from path-planning algorithms to motor control, the ROS libraries offer already existing functionality. Additionally, robots can be simulated and visualized with tools like RViz and Gazebo which can speed up the development process significantly. For real-world applications, communication between robot and host over a network is made possible by setting up a remote single master network. This network then allows communication as if all components were local to the host [17].

At the core of each ROS network lies the ROS master which coordinates the location of nodes, subscribing, publishing, and the transmission and timing of messages and services. One fundamental principle of ROS is, that every network, even when multiple robots are connected, features only one master. This allows for a clear hierarchy and permits peer-to-peer communication between all nodes [18].

2.3.2 Example of Nodes, Messages, and Services

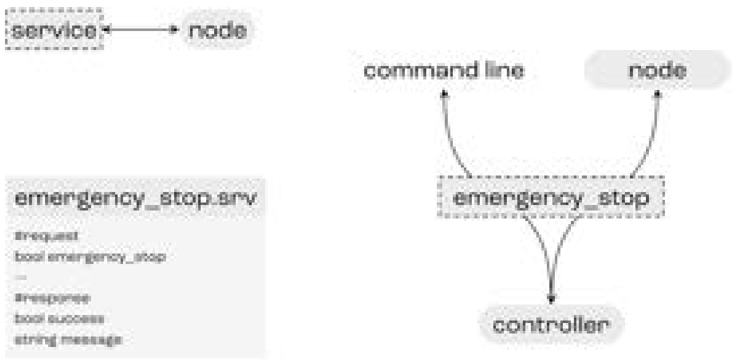
Although ROS already offers a vast collection of message types, it can be necessary to define custom messages which can be achieved through a `.msg` file. An example of such a file for a custom position message can be seen in Figure 1. A simple ROS network consisting of two nodes can be seen in Figure 1. These two nodes communicate with each other with two messages published to the `position` and `velocity` topic. The `position` message, published to the `position` topic offers information about the actual position of the robot with respect to a map coordinate system and the velocity message states the desired velocity of the robot. The `controller` node subscribes to the `position` message, calculates the error in position, and publishes the resulting control velocity to the `velocity` message. The `robot node`, which subscribes to this topic, then controls the robot's movement based on the received desired velocity. The resulting position is then published in the `position` message. Thus a highly simplified feedback control loop with minimal dependency is made possible.



```
position.msg
double x_position
double y_position
double z_position
double roll_angle
double pitch_angle
double yaw_angle
```

Figure 1: Graphical representation of a two-node feedback controller with matching message definition. The `controller` node publishes a desired velocity to the `velocity` topic, which is then executed by the `robot`. The resulting change in position is fed back via the `position` topic.

In addition to periodically published messages, ROS offers discontinuous communication through services. These services consist of an array of messages, split up into a request and a subsequent response. Although many predefined service structures exist, custom service routines can be defined in a `.srv` file. The structure of a `.srv` file and the act of performing a service call is best explained in the following example. The `emergency_stop` service which can be seen in Figure 2 consists of a request of boolean type, i.e. true or false. This request can be generated by a service client, either in a separate `node` or over the `command line` to perform an emergency stop. The `controller` node, which also acts as the service server, then processes this request, performs the emergency stop, and replies with a message depending on the outcome of the emergency stop. This response can then again be evaluated by the service client.



```
emergency_stop.srv
#request
bool emergency_stop
---
#response
bool success
string message
```

Figure 2: Graphical representation of a service call and matching service definition. The `emergency_stop` service can be requested both from a `node` and from the `command line`. After finishing the emergency protocol, the service provides a response about the outcome.

2.3.3 ROS Versions and Rosbridge

With the development of ROS, various versions of the system were published. A specific version of ROS can cause compatibility restrictions when using it together with other software. The version needed for compatibility with the Unitree SDK, ROS Melodic supports only Python 2.7 and lower. To nevertheless employ Python 3 programs along with older ROS versions, rosbridge can be used. Rosbridge is a ROS community project, aiming to provide communication with ROS for non-ROS applications through various front ends and using a JavaScript Object Notation (JSON)-based application programming interface (API). The rosbridge protocol enables the use of various fundamental ROS functionalities such as topics and services [19]. To communicate with ROS, for example, from a non-ROS Python program, the `roslibpy` package can be used. It allows to establish a connection to the rosbridge server via the WebSocket protocol and provides the use of the aforementioned fundamental ROS functionalities [20].

2.3.4 Existing Software for the Unitree A1

For the Unitree A1 used in this thesis, some basic ROS packages exist. The manufacturer provides packages consisting of the robot model, control modes, and motor drivers. These can then be used to spawn the robot in simulation tools like Gazebo and RViz or to control it in real life through communication with the SDK [21], [22]. On their own, these packages provide no way of controlling the robot's velocity through a ROS message. Neither does the simulation reflect the real configuration of the robot's position. This is where the `qre` packages written by MYBOTSHOP (distributor of robotics and automation technology products) come into place. They offer a bridge between ROS and the Unitree SDK by translating ROS messages into SDK commands [23]. Both the linear and angular velocity can be set via a ROS message and the pose of the robot can be altered through a ROS service. With these two simple commands, a control system for the robot can be designed to fit the requirements of this thesis.

Since ROS itself is not real-time capable, the Unitree SDK utilizes a Lightweight Communications and Marshalling (LCM) server which allows for high bandwidth and low latency communication via the User Datagram Protocol (UDP) [24]. As seen in Figure 3, a separate node is responsible for receiving ROS messages and converting them into LCM commands to be sent in real-time to the Unitree SDK and vice-versa. As a result, a maximum real-time communication bandwidth of 1 kHz can be achieved.

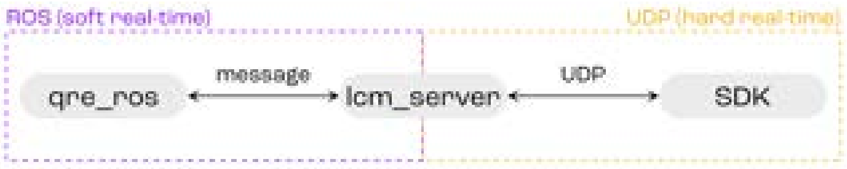


Figure 3: The LCM server enables real-time communication between the ROS node and the Unitree SDK by converting ROS messages into UDP commands in real-time.

2.4 LiDAR

The abbreviation LiDAR stands for "light detection and ranging" and describes a widely used type of laser measurement device. LiDARs are active sensors sending and receiving electromagnetic waves with wavelengths of typically around 1.5 μm, which differentiates them from radar sensors using longer wavelengths. Measurements typically contain angular and distance information, with the latter being obtained for instance by measuring the time of flight of the laser beam employed by the sensor. Depending on the type of sensor, additional information such as the target's reflectivity or even the velocity of an observed object can be derived by using physical laws such as the Doppler shift [25].

LiDAR sensors are utilized in a wide range of applications. One of the most important ones is generating 3D imagery of the environments of autonomously-driving vehicles and mobile robots. Among various other areas, LiDAR is also widely used in engineering and survey mapping, vegetation measurement, mapping in civil and military aerospace applications as well as monitoring in meteorology [25], [26].

3 Methods

To fulfill the goal of a responsive and safe operating software package, the open-source platform ROS has been chosen, which enables running multiple programs (nodes) in parallel while enabling communication between them. The developed ROS packages can be split up into four parts, encompassing position control, localization, a collision avoidance system, and an auxiliary node for gesture recognition, each consisting of one or more ROS nodes. As non-ROS components communicate with ROS via rosbridge, the existing gesture recognition algorithm and the GUI developed in this thesis run on Python 3.10.

Figure 4 shows how these nodes are connected and interact with each other. Nodes with the prefix `rbd` were developed during this thesis, while the nodes marked with a green dot are open-source nodes available as ROS packages. The `ekf_localization` and `slam_toolbox` nodes are used in combination with the `rbd_localization` node to determine the global position of the robot. Finally, the `qre_ros` driver is used to control the robot itself by velocity and pose commands.

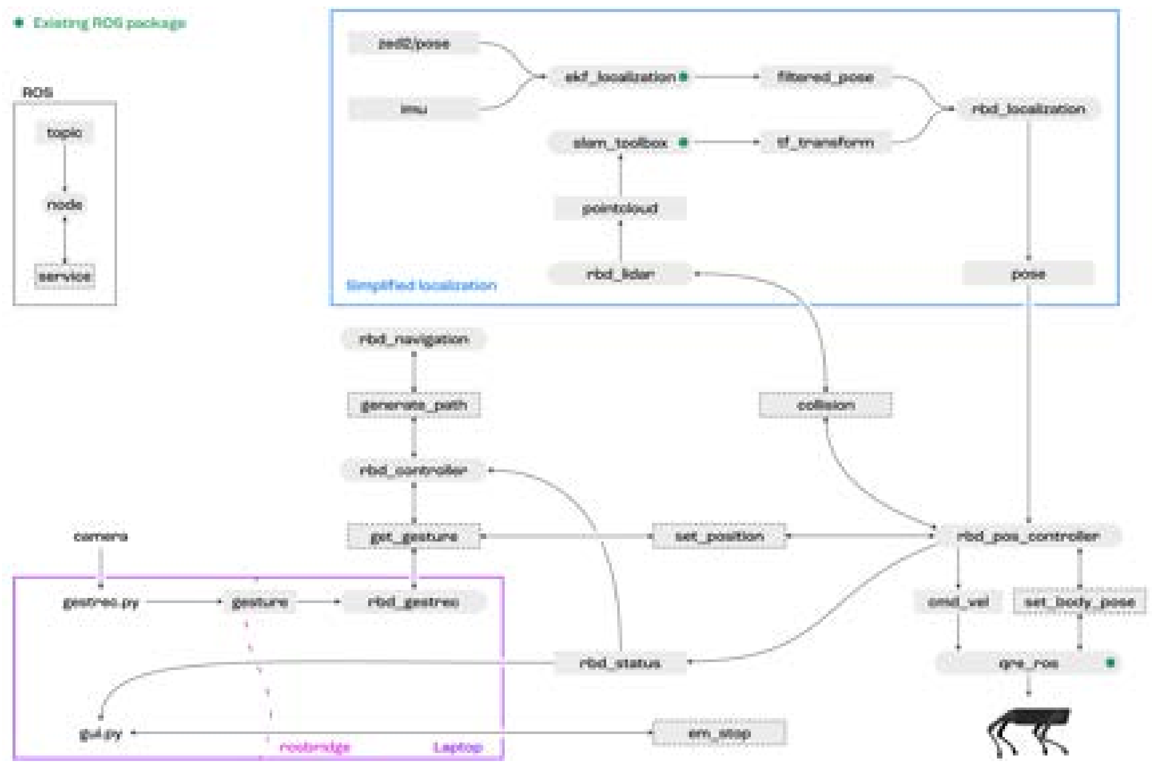


Figure 4: Graphical representation of the ROS system, which shows the used nodes and their connection by topics and services.

The resulting control flow can be described as follows: First, a gesture is recognized by the camera, which then triggers a command to be executed by the control system. This control system generates a path to follow and subsequently commands the robot to move to the intermediate waypoints. During this, a LiDAR-based collision avoidance algorithm prevents collisions with both static and dynamic objects.

At the center of the system lies the main controller `rbd_controller` which coordinates the control flow. After launching the system, this controller is in emergency stop mode by default. As soon as the emergency stop is disabled, a gesture is requested from the gesture recognition node `rbd_gestrec`, which then waits until a gesture is performed in front of the camera. Based on the received gesture, a command addressing the `rbd_navigation` node is generated to receive a movement path from this node. The navigation node is equipped with predefined movement patterns for each known command, which are returned upon request as an array of poses.

These poses are then transferred individually to the position controller `rbd_pos_controller` which is responsible for the execution of movement. The position control algorithm first rotates the robot in the direction of the linear trajectory to ensure minimal side forces while moving. After orienting, a feedback loop generates velocities in local x- and y-directions, resulting in a movement toward the desired endpoint. Upon reaching this point, the robot is finally oriented to the desired orientation in global coordinates. This sequence is then repeated for all remaining poses of the pose array, during which collisions are monitored by the collision avoidance node `rbd_lidar`, resulting in an executed choreography of movements.

3.1 Integration of Existing Gesture Recognition

As mentioned earlier, this thesis uses the existing gesture recognition software developed in the preceding project thesis. In this algorithm, the gesture recognition runs in a parallel process provided by Python's multiprocessing library to increase performance. Initially, an image is obtained from the robot's integrated camera and subsequently analyzed for the occurrence of characteristic hand key points using the MediaPipe library. The detected key point coordinates from a sequence of video frames are first preprocessed and then added to a unidimensional queue. This queue is then forwarded into a logistic regression model and classified in terms of the contained gesture [7].

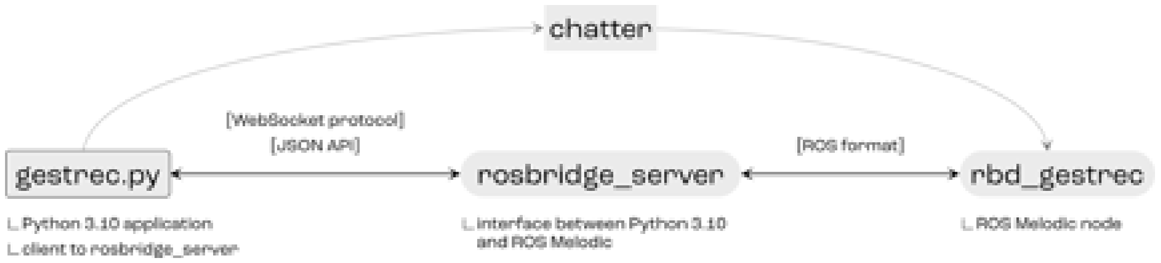


Figure 5: Rosbridge offers an API identical to ROS messages.

The ROS packages required for controlling the Unitree A1 solely operate on ROS Melodic, which only runs on Python versions up to 2.7. Hence, custom ROS packages required in this thesis were restricted to Python 2.7. Since the existing gesture recognition software uses recent Python 3 libraries and frameworks such as MediaPipe, the application could not be run directly using ROS Melodic. To employ the existing gesture recognition application nevertheless, rosbridge, and the corresponding Python package `roslibpy` were used as an interface from Python 3 to ROS Melodic. The rosbridge configuration utilized in this thesis mainly consists of three components: the Python 3 gesture recognition application using the `roslibpy` library, the rosbridge itself, and the `rbd_gestrec` ROS node that receives information about gestures recognized by the Python 3 program. A typical communication process from Python 3 to ROS as employed in this thesis is conceptually visualized in Figure 5 and can be summarized as follows [19], [20]:

i. In the Python 3 application, a client for the rosbridge server is created. Publishing a ROS message can be achieved by calling the corresponding method of `roslibpy` and specifying the client as an argument.

ii. The message is sent to the rosbridge server via the WebSocket protocol using a JSON API.

iii. After the rosbridge server has received the message, the `rosbridge_library` package, which is part of rosbridge, converts it from the universal JSON format to the ROS format.

iv. The message is forwarded to the `rbd_gestrec` ROS node by the rosbridge server.

3.2 Control System Design

The main goal of the control system is to execute a series of movements based on the gesture received from a user while staying responsive to emergencies. To reach this goal, the main controller node `rbd_controller` coordinates the communication between the task-specific nodes by requesting information and distributing it to the corresponding nodes. To ensure a clearly defined control flow, the controller is designed as a finite state machine (FSM), which can only be in one state at a time [27]. As seen in Figure 6, all states, except for the emergency stop, can be preempted by both the emergency stop and the collision prevention.

For safety purposes, the `Emergency Stop` state has been chosen as the first active state after starting the program. Only if this emergency stop is disabled manually via the command line by the user, the state is changed to `Waiting for Gesture`. This safety feature ensures no unexpected movements after starting up the robot at a live demonstration. If the emergency stop is enabled during any of the other states, the state changes to the `Emergency Stop` state, therefore requiring user input to continue the control flow.

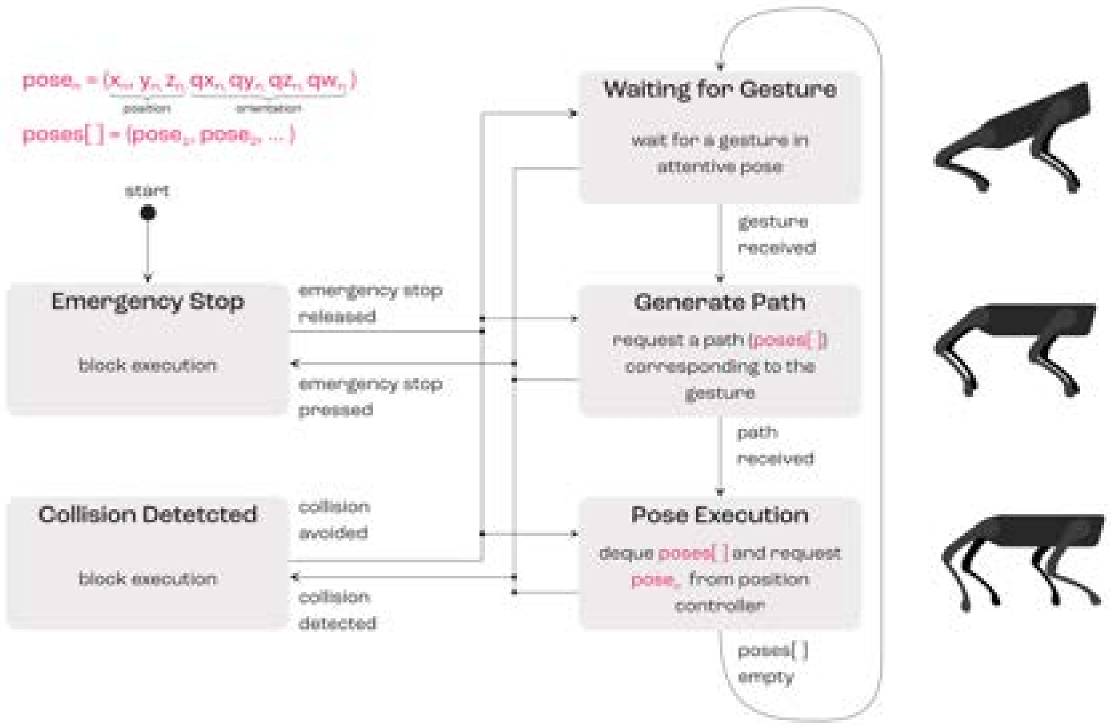

pose$_n$ = ($\underbrace{x_n, y_n, z_n}_{position}$, $\underbrace{qx_n, qy_n, qz_n, qw_n}_{orientation}$)
poses[ ] = (pose$_1$, pose$_2$, ...)
start
Waiting for Gesture
wait for a gesture in attentive pose
gesture received
Emergency Stop
block execution
emergency stop released
emergency stop pressed
Generate Path
request a path (poses[ ]) corresponding to the gesture
path received
Collision Detetcted
block execution
collision avoided
collision detected
Pose Execution
deque poses[ ] and request pose$_n$ from position controller
poses[ ] empty

Figure 6: The state machine diagram shows all possible states and their transition conditions. Based on the command from the gesture recognition node, an array of poses is generated, whose elements are then passed individually to the position controller to create a sequence of movements.

The same applies to the `Collision Detected` state with only one difference: The state change is initiated by the `rbd_lidar` node based on the measured distances to the robot's environment. If a certain number of measurements surpass the distance threshold, the `rbd_lidar` node automatically stops the movement to prevent collisions. Contrary to the emergency stop, the state machine can return from the `Collision Detected` state without user input if the environment is safe, i.e., no measurements surpass the distance threshold.

Control Flow

After the emergency stop has been released and if the environment is free of obstacles, the state machine transitions to the `Waiting for Gesture` state. On entry, the robot is oriented around the y-axis to the attentive *sitting* pose. On one hand, this signals the user that the robot is ready to receive a gesture. On the other hand, the camera's field of view is also oriented toward the user.

As soon as a gesture is recognized with enough confidence, the robot is moved back to its initial orientation and a corresponding path is requested from the `rbd_gestrec` node. The gesture recognition node then responds with an array of poses representing a sequence corresponding to the requested command. These pose elements can be described as a position with a corresponding orientation.

After receiving a path, each element of the pose array is dequeued and a request for the pose element is sent to the position controller. As soon as the robot reaches this pose, a signal is sent back to the main controller and the next element is dequeued, resulting in a continuous motion to the desired goal points. If either the emergency stop is pressed or a collision is detected during this motion and subsequently released, the state machine returns to the previous state to continue the desired motion.

Finally, after all elements have been dequeued and the pose array is empty, the state changes back to `Waiting for Gesture` and a new gesture can be performed.

3.2.1 Localization and Path Generation

There are many ways to determine the position of an object, each with its advantages and drawbacks. For this robotic platform specifically, there is a redundancy of sensors that can be used. The stereo camera, the LiDAR sensor, and the robot itself feature an inertial measurement unit (IMU) including an accelerometer, which can measure linear accelerations, and a gyroscope measuring rotational velocities. These sensors can be used to determine a relative position. Additionally, the stereo camera and the LiDAR sensor can be used to determine absolute positions through the identification of features in the environment. The mentioned sensors can be categorized as follows:

Name	Bandwidth	Localization Type	Error over time
IMU sensor	500 Hz	relative	large
stereo camera	15 Hz	absolute	small
LiDAR sensor	0.2 Hz	absolute	none

Table 1: Categorization of the available sensors through which the position of the robot can be determined.

IMU sensors are typically high in noise and feature minimal offsets in the measurements, which lead to a drift in position when integrated over time. With this approach, only a relative position can be determined, which can lead to noticeable displacements over time.

In contrast, a stereo camera or a LiDAR sensor can be utilized to determine a position based on environmental features, thus providing an absolute position. Opposite to the relative positions, absolute positions are free of positional drift while sudden changes in positions can occur when new landmarks are detected.

Sensor Fusion
To combine the strengths of the mentioned sensors it can be useful to combine the advantages of both approaches by using a sensor fusion algorithm which is shown symbolically in Figure 7. The sensor fusion algorithm used in this thesis is the extended version of a Kalman filter. This non-linear model-based filter can be fed with an arbitrary number of sensor signals in the form of 15 states $(x, y, z, roll, pitch, yaw, \dot{x}, \dot{y}, \dot{z}, \dot{roll}, \dot{pitch}, \dot{yaw}, \ddot{x}, \ddot{y}, \ddot{z})$, which are then combined by the Kalman algorithm to predict and correct the position measurement. The algorithm used in this thesis is available as a ROS package under the name `robot_localization` [28]. This ROS node can be configured to subscribe to an arbitrary number of ROS messages consisting either of IMU, odometry, twist, or pose data which are combined into an accurate state estimation.

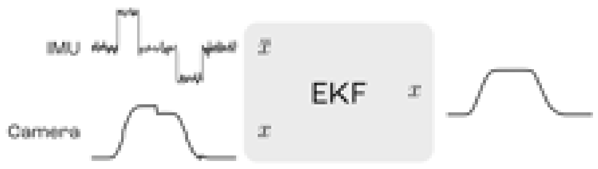


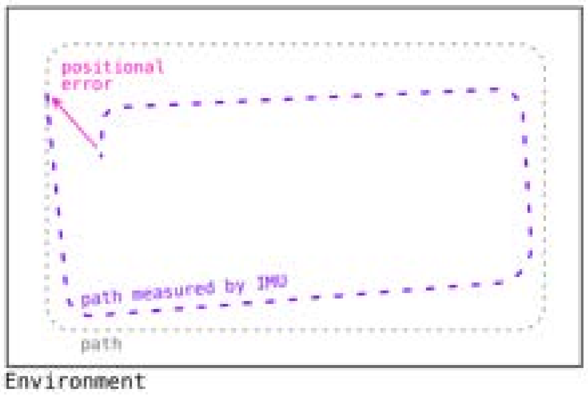
Figure 7: Graphical representation of the working principle of an extended Kalman filter (EKF). Through sensor fusion, a noisy but consistent IMU signal can be combined with a low-noise but inconsistent camera position signal to form a consistent low-noise position measurement.

Furthermore, the accompanying paper by Moore and Stouch [29] shows that combining two different sensor types results in better estimations than the combination of two identical sensor types. Therefore, there is no advantage in using more than one of the three IMUs available. Subsequently, the IMU of the LiDAR sensor has been chosen due to its low noise level and close positioning to the center of rotation. The IMU measurements are combined with the pose estimated by the stereo camera's internal 3D localization, resulting in drift-free state estimation. In ROS, this estimated position is described as the relationship between the robot `base_link` and the `odom` coordinate system.

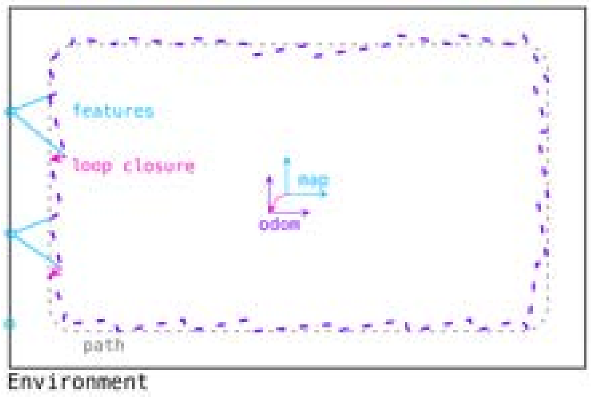
SLAM
Since the absolute position provided by the stereo camera is not completely accurate and only uses the camera's field of view, the LiDAR data are also used as a reference to correct the pose periodically. LiDAR-based state estimation can be achieved through SLAM, where a two-dimensional map is created based on two-dimensional laser scans. This map is then used as a reference for further measurements to determine the absolute position based on recognized features. This algorithm, also known as loop closure, leads to accurate and reliable measurements. Its only drawback is the high computational cost, leading to low bandwidth.

In ROS, this correction can be implemented elegantly by adding a `map` coordinate system. The `odom` coordinate system is then transformed to correct the error, resulting in a correct `base_link` position with respect to the `map` coordinate system. The ROS package `slam_toolbox` [12] creates a map from the received LiDAR data and calculates the coordinate transformation based on SLAM.

Figure 8 shows a concept of correcting a state estimation featuring drift by periodically performing loop closure (magenta arrows). In Subfigure 8a, the trajectory estimated from relative measurements shows a large absolute error after execution of the motion path. By adding SLAM, the state estimation drifts only between corrections, as seen in Subfigure 8b.



(a) Localization without SLAM.



(b) Localization with SLAM.

Figure 8: The conceptual comparison of state estimation with and without SLAM shows how relative measurements featuring drift can be corrected periodically with an absolute measurement.

In conclusion, the combination of both fast relative and slow absolute measurements shown in Subfigure 8b leads to a sufficiently accurate state estimation with high bandwidth and minimal positional drift over time.

3.2.2 Position Control

Similar to the main controller, the position controller is realized with an FSM, which is depicted in Figure 9. A request to the `set_position` service initiates the first state change. The FSM features not only states but also a status variable, required to display information about the algorithm in the GUI. In addition to the emergency and collision policy mentioned in Chapter 3.2, the two states `Emergency Stop` and `Collision Detected` change all velocities to zero during the entry of the state. This ensures that the robot stops before colliding with a person or an object. If either the emergency stop is released or the environment is clear of any objects, the state is changed to `Waiting for Pose` and the status to `idle`, signaling to the main controller, that a new pose can be requested. Since the position controller node runs in parallel to the main controller node, this is necessary to ensure a linear control flow. As soon as the main controller requests a pose from the `set_position` service, provided by the position controller, the waiting state is left and the status is changed to `running`.

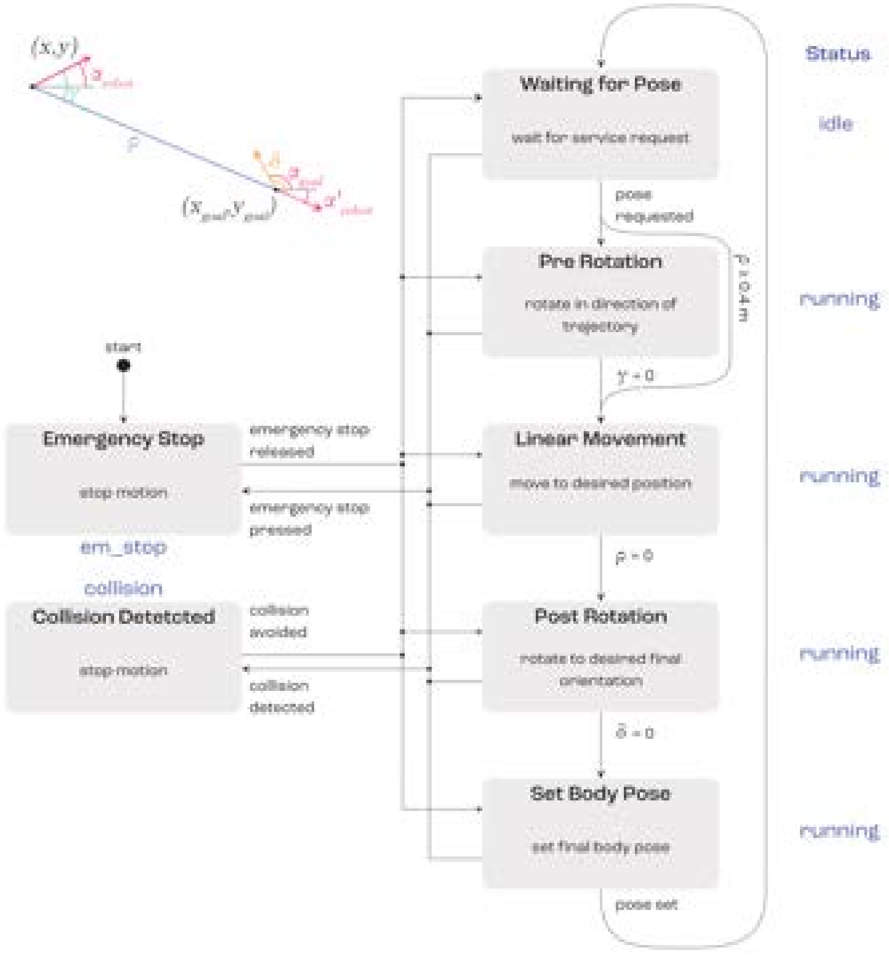


Figure 9: The state machine diagram shows all possible states of the position controller and their transition conditions. Based on the requested goal pose the robot is first rotated in the direction of the trajectory. Subsequently, the linear velocities are controlled to move the robot to the goal position, where it is oriented to the requested orientation.

The movement control is split up into four parts, each with its own controller policy described in Equation 1. First, the robot is rotated in the direction of the remaining trajectory to ensure minimal side forces during movement, which could lead to a vulnerability to instability. In other words, the angle between the trajectory and the actual angle of the robot (γ) is minimized each time a new pose measurement is available (in this case 10 Hz). This step is skipped if the remaining distance is shorter than 0.4 m, which has been determined empirically. After the robot is oriented, the forward and side velocities of the robot are controlled relative to the remaining distance and angle, resulting in smooth movements towards the desired endpoint and minimizing the Euler distance ρ between the robot and its goal point. If the endpoint is reached, the robot is rotated towards the desired endpoint angle by minimizing the angle between the desired angle and the actual angle of the robot (δ). After this step, the robot is at the desired position and yaw angle. Therefore only the roll and pitch angle needs to be set. This can be done by requesting the `set_body_pose` service of the `qre_ros` node. Finally, the status is set back to `idle`, which signals to the main controller, that a new pose can be requested, and the state is changed back to `Waiting for Pose`.

The following equations describe how the velocities of the robot have to be controlled based on the difference in position and orientation:

$$
\begin{array}{lc}
 & \textbf{Control Policy} \\
\textbf{Pre Rotation } \min(\gamma)\textbf{:} & \dot{\alpha} = \gamma k_\alpha \\
 & \text{(optional pos. control)} \\
\textbf{Linear Movement } \min(\rho)\textbf{:} & \dot{x} = \rho\ k_x \cos(\gamma) \\
 & \dot{y} = \rho\ k_y \sin(\gamma) \\
\textbf{Post Rotation } \min(\delta)\textbf{:} & \dot{\alpha} = \delta k_\alpha \\
 & \text{(optional pos. control)}
\end{array}
\qquad
\begin{array}{c}
\textbf{Geometric Variables} \\
e_x = x_{goal} - x \\
e_y = y_{goal} - y \\
\rho = \sqrt{e_x^2 + e_y^2} \\
\alpha = \text{atan2}(e_y, e_x) \\
\gamma = \alpha - \alpha_{robot} \\
\delta = \alpha_{goal} - \alpha_{robot}
\end{array}
\tag{1}
$$

If the robot does not rotate perfectly around its central axis, the position can be controlled additionally during rotation, leading to a smaller positional error after the rotation. Although some imperfections are expected, their amount cannot be predicted and will have to be measured experimentally.

Due to its nature, the high-level robot control by velocities shows integrating behavior on its own. Hence, even a simple proportional controller will be sufficient in reaching a position with no remaining static error [30].

3.3 Collision Detection

As an overarching aim, the collision avoidance system implemented in this thesis must work reliably and safely, even if ambient conditions like illumination vary. As it is intended to use the A1 in crowded, unsteady, and indeterministic environments such as trade fairs, 360-degree responsivity is required. LiDAR sensors fulfill these requirements and are state-of-the-art technology for various collision avoidance applications. Since a two-dimensional LiDAR would have constrained the possibilities for more sophisticated analysis of the measurement data in future projects, such as environment segmentation or object classification, a LiDAR generating three-dimensional data is deployed.

3.3.1 LiDAR Sensor

In this thesis, an Ouster OS1 32 Gen 2 LiDAR device (OS1) is utilized, delivering three-dimensional spatial data in a range from 0.3 m to approximately 50 m or 100 m, depending on the target's reflectivity. This LiDAR model has already been implemented in various projects using Unitree A1 robots. A range accuracy of ± 30 mm for Lambertian targets (reflecting irradiating light in all directions) respectively ± 100 mm for retroreflectors (e.g. mirrors) and a precision in the low centimeter range can be achieved, depending on the specific conditions. The amount of data obtained can be adjusted by configurable parameters such as angular resolution or the sensor's rotation rate. A 3D-printed fixture enables the installation of the OS1 on the back of the A1, which also powers the sensor with its 19 V output [31].

The OS1 delivers complexly structured data packages at a frequency of 10 Hz or 20 Hz (configurable) that need to be reliably analyzed in a safety-critical collision avoidance system, generating the demand for an appropriate method to process the data. Per default, the measurement information is parsed into measurement blocks consisting of a header, a status sequence, and the measurement data themselves. Instead of developing a novel algorithm to explore the measurement blocks, the Ouster SDK can be used. However, as a feature for robotics applications, Ouster provides the ROS package `ouster-ros` in addition to the SDK to process the LiDAR point clouds. The package features three main functionalities with a corresponding ROS launch file for each of them: In the sensor mode, data from a running LiDAR sensor can be accessed. This information can be captured and stored in a ROS bag file by using the recording mode and are subsequently accessible via the replay mode [32], [33].

The `ouster-ros` package provides various ROS messages that can be processed, either in real-time by utilizing the sensor mode but also when replaying bag files. A range image is available on a topic named `ouster/range_image`, data from the IMU are published on the topic `ouster/imu` whereas the point clouds are accessible via `ouster/points` under the message type `PointCloud2`. For each 360-degree measurement, one `PointCloud2` message is published as a vector containing different metadata sections with information about the message format and the measurement itself, as well as a data section containing, among others, spatial point coordinates, an object reflectivity value, and range information [33].

In this thesis, the `range` data field is accessed and analyzed to detect objects encountering near the A1. As an already integrated feature, the `ouster-ros` package contains a launch file for RViz, which is automatically called when launching `Sensor` or `Replay`. RViz allows for a broad range of modification actions on the visual aspects of the point cloud, which is utilized in this application for the visualization of collision-prone objects near the A1.

3.3.2 Collision Avoidance Algorithm

For the collision avoidance system, the package `rbd_lidar` was implemented. The collision avoidance node aims to detect and visualize objects within a critical zone (illustrated in Figure 10) of the A1, where the critical zone describes a cuboid-shaped volume in which no person or object should be located while the robot is performing actions.

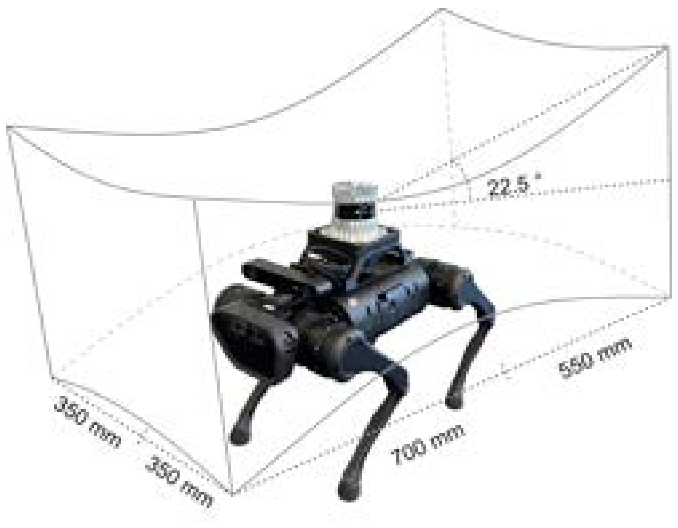


Figure 10: The critical zone around the Unitree A1 is rectangular from a top view and vertically constrained by curved edges due to the polar measurement of the LiDAR sensor. The cuboid's center of mass is offset from the LiDAR's center since per default, the robot's front critical zone (700 mm) is larger than the rear zone (550 mm).

The `rbd_lidar` node receives a `PointCloud2` message containing the LiDAR measurements published by the Ouster ROS node. The algorithm generates both a new `PointCloud2` message containing only the points within the critical zone and a `PointCloud2` message with the central LiDAR scan slice to be converted into a `LaserScan` message for subsequent SLAM localization. Additionally, the algorithm executes a service call if too many points are detected inside the critical zone.

Conceptional Description of the Algorithm

The following paragraphs are a conceptional description of how the algorithm functions and Figure 11 visualizes its main subtasks.

To facilitate manipulating the point cloud, the `PointCloud2` data are converted into a `pcl::PointXYZI` data type provided by the `pcl` C++ library. Using this data type, spatial coordinates, and an intensity value can be specified for each measurement point. For the subsequent steps, a two-dimensional array containing the range information for each point in the point cloud is created. The array possesses the dimension $32 \times res_h$, where each of the 32 rows describes one of the horizontal slices in the point cloud and with res_h describing the horizontal resolution configured for the OS1, which can be either 512, 1024, or 2048 measurement points per slice. This two-dimensional array, representing polar coordinates, increases the utility of the measurements for programming purposes since the raw `PointCloud2` data are simply stored inside a unidimensional vector, which does not allow row- and angle-based access to single points inside the point cloud.

Based on the two-dimensional range array, the actual classification of the measurements is executed. The critical zone can be chosen asymmetrically by specifying a perpendicular distance to each side of the robot. These values can be defined in the `default.yaml` file of the `rbd_lidar` package and are per default set to 350 mm for the left and right side of the robot. Since many of the robot's movements are forward with a higher velocity, the critical distance in front of the robot is set to 700 mm whereas the zone rear to the robot is set to 550 mm as a standard. The aforementioned four values have been determined empirically through experimental testing and have proven to be suitable for enabling unimpeded interaction with the robot while effectively avoiding collisions. Based on these values (d_b, d_f, d_l, d_r), the collision avoidance algorithm first calculates four azimuths $\varphi_1 ... \varphi_4$ (depicted in Figure 12) for the angular sections around the robot, each standing for one side of it.

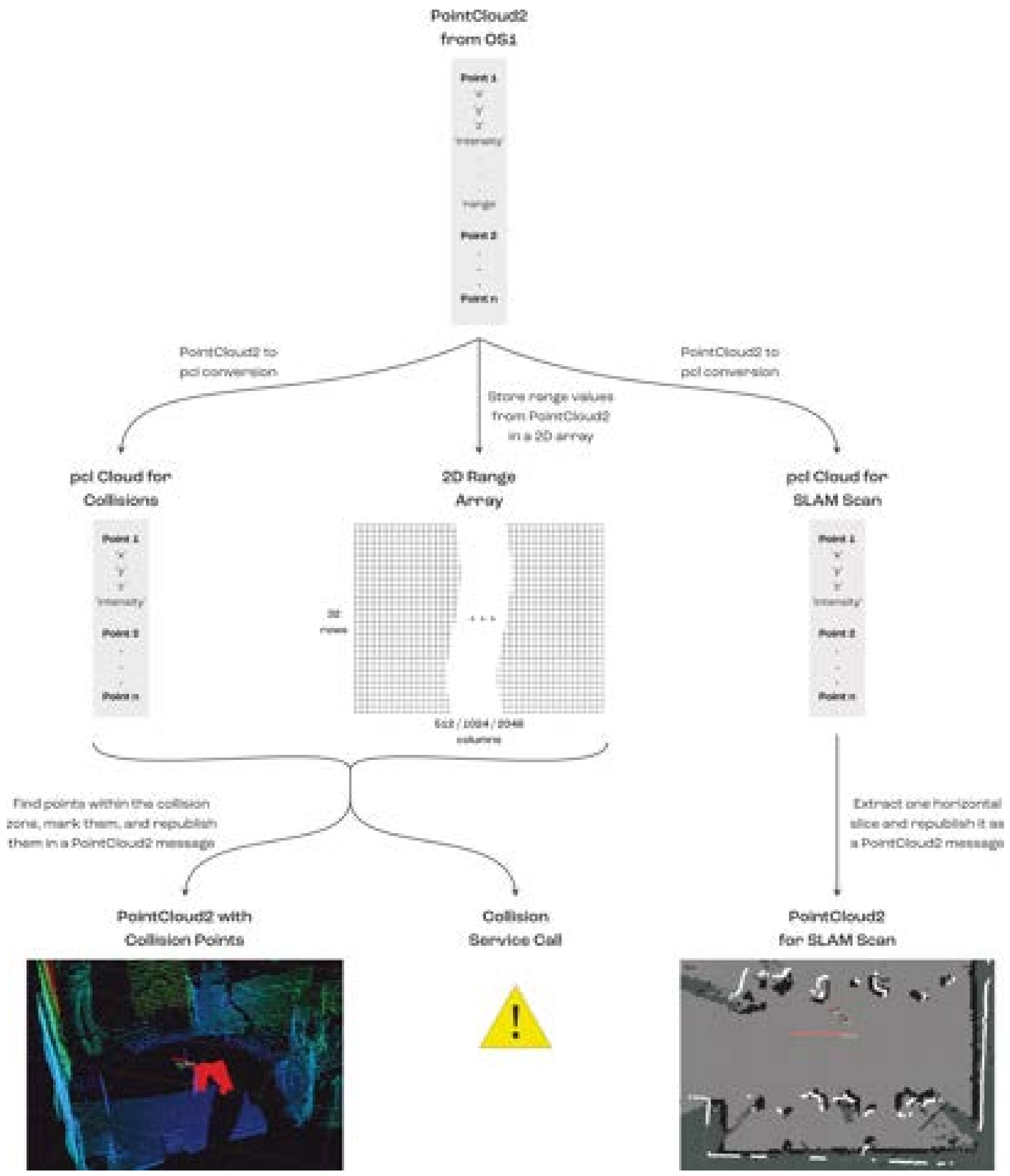

PointCloud2 from OS1
Point 1
x
y
z
intensity
range
Point 2
Point n
PointCloud2 to pcl conversion
Store range values from PointCloud2 in a 2D array
PointCloud2 to pcl conversion
pcl Cloud for Collisions
2D Range Array
pcl Cloud for SLAM Scan
32 rows
512 / 1024 / 2048 columns
Find points within the collision zone, mark them, and republish them in a PointCloud2 message
Extract one horizontal slice and republish it as a PointCloud2 message
PointCloud2 with Collision Points
Collision Service Call
PointCloud2 for SLAM Scan

Figure 11: The collision detection algorithm generates three main outputs, all of which are derived from the `PointCloud2` message obtained from the OS1 measurements.

For each section, the aforementioned range array is investigated in terms of points within a perpendicular distance lower than the critical value for the corresponding side of the A1. Points within the critical zone are counted and marked via the point's intensity attribute, which enables the subsequent visualization in RViz. Since the OS1 has a minimum range of 0.3 m, a corresponding spherical blind zone is defined. Measurements within this zone are ignored as they fall below the specified minimum range and are therefore considered unreliable. Points beyond the collision zone are discarded to finally obtain a point cloud that contains the critical points only. As this point cloud is not yet available in a ROS-compatible format, it must be reconverted to the `PointCloud2` format and is subsequently published as a ROS message, which can then be accessed for visualization purposes.

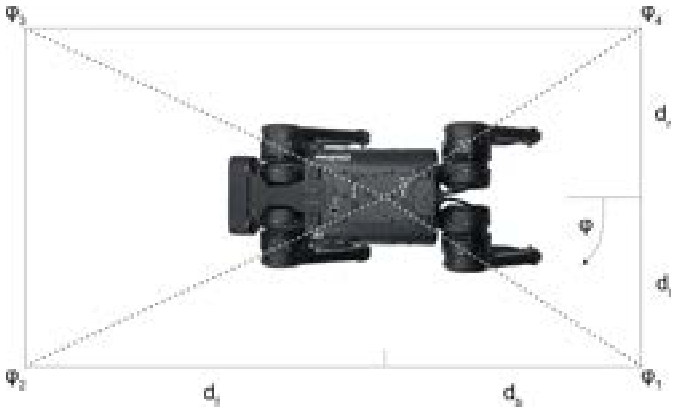


Figure 12: The four angular sections of the critical zone are defined by the four distance values specified in the `default.yaml` file [34].

Since the OS1 is not only used for collision detection but also for SLAM, the horizontal LiDAR scan slice at the height of the OS1 itself is required as input for the SLAM algorithm. To achieve this, the mentioned slice is, in addition to the `PointCloud2` containing collision points, stored in an additional `PointCloud2` data structure. The corresponding message is then converted by the `PointCloudToLaserScanNode` node of the `pointcloud_to_laserscan` package and republished as a `LaserScan` message, enabling the SLAM algorithm to construct a map from it [35].

Finally, the total number of points within the collision zone is checked, and the collision status is refreshed if necessary. If the collision status has changed, the corresponding service of the controller node is called, enabling collision stop reactions.

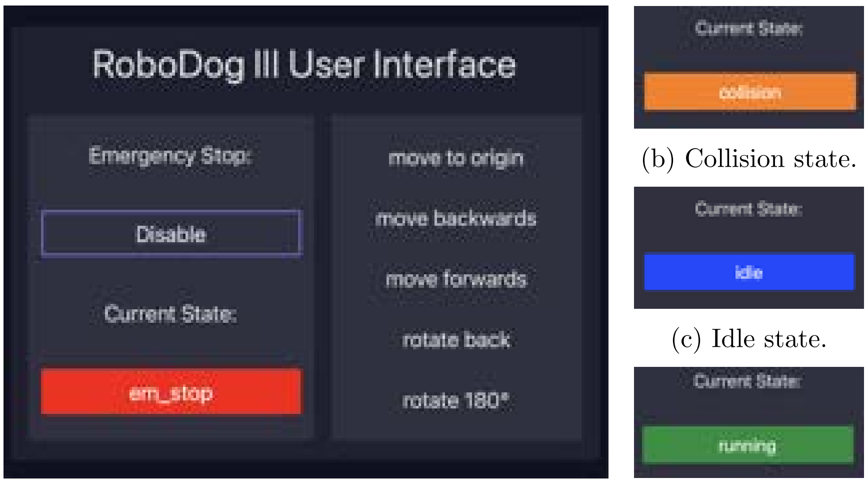
3.4 Visualization

As an interface between software logic and the human audience, the visualization has the important task of intuitively depicting the system's process logic. Since the AI demonstrator developed in this thesis is meant to not only be presented to trained professionals but to a broader public, it must hide complex program logic and illustrate the main ideas behind the employed algorithms. The visualization of the gesture recognition, which displays detected hand key points and the gesture classification with a probability value, serves as the central visualization element. It is adapted from the preceding project thesis [7] and incorporated into this work. In addition to this visualization of the main AI component, further illustrations are necessary to make the inner workings of the system graspable. The following sections outline how a GUI and a point cloud visualization have been implemented to reach this goal.

3.4.1 GUI

Figure 13 illustrates the GUI, which features a software emergency stop button and feedback on the current and future states, enhancing the transparency of the decision process inside the robot. While the LiDAR point cloud and the camera image offer information on how the robot perceives its environment, the GUI displays the decisions the robot's control system derives from the sensor data.

The GUI utilizes the Python framework CustomTkinter, allowing fast development of modular graphical user interfaces with a modern appearance [36]. This change of frameworks is necessary because the GUI developed in the preceding project thesis is programmed using the framework PyQt5 [7], [37]. Unfortunately, compatibility issues with PyQt5, Linux Ubuntu, and the environment used for the gesture recognition software exist and could only be fixed by choosing another framework.



(a) User interface with enabled emergency stop. (b) Collision state. (c) Idle state. (d) Running state.

Figure 13: The graphical user interface displays both the status of the robot as well as the next five planned movements. For safety purposes, a software emergency button is available to stop the robot in emergencies.

Similar to the gesture recognition software, the GUI utilizes rosbridge to communicate with ROS. If the emergency stop, located on the upper left, is pressed, the `em_stop` service is called via rosbridge, interrupting the robot's motion. The resulting change of states updates the displayed state, informing the user of a successful emergency stop. As seen in Subfigures 13a to

13d, the GUI displays all four possible states of the position controller. To enhance its visibility, the background color of the state label changes corresponding to the current state. Red signals an `emergency stop`, orange a `collision`, blue the `idle` state, and green that the position controller is `running`.

Additionally, the next five movements are displayed, on the right half of the GUI seen in Figure 13, to communicate how the robot plans its motion. Path elements are dequeued from the bottom once they are reached successfully, similar to how the main controller processes the pose array. This visualization provides additional information on how the robot plans and navigates through the environment. In some cases, the robot's motion is not directly comprehensible at first glance, and additional information is required.

3.4.2 Collision Detection

The collision avoidance algorithm is illustrated using the ROS tool RViz. This application enables, among others, the efficient visualization of `PointCloud2` messages. To make the analysis of the point cloud accessible to an audience, two main requirements are to be fulfilled:

i. The point cloud delivered by the OS1 must be illustrated in a way that allows a direct understanding of the depicted spatial information and a quick orientation inside the environment.

ii. Points inside the critical zone around the A1 must be visually accentuated unambiguously.

To fulfill these goals, the original data provided by the OS1 are represented by a rainbow color-mapped point cloud, with colors changing depending on the distance of a point to the sensor, to provide depth information. Furthermore, the collision avoidance algorithm of the `rbd_lidar` package delivers a point cloud containing only the points within the critical zone around the robot. Figure 14a illustrates a typical point cloud as it is obtained from the OS1 with the aforementioned RViz settings in a crowded environment. The manipulated point cloud containing the measurements within the critical zone is overlaid over the aforementioned data and colored in red for differentiation and accentuation. Figure 14b displays how a person surpassing the critical distance thresholds is highlighted by red color.

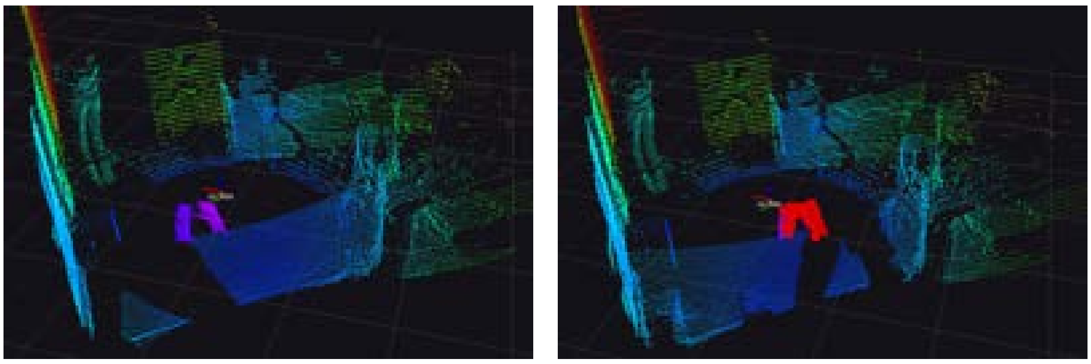
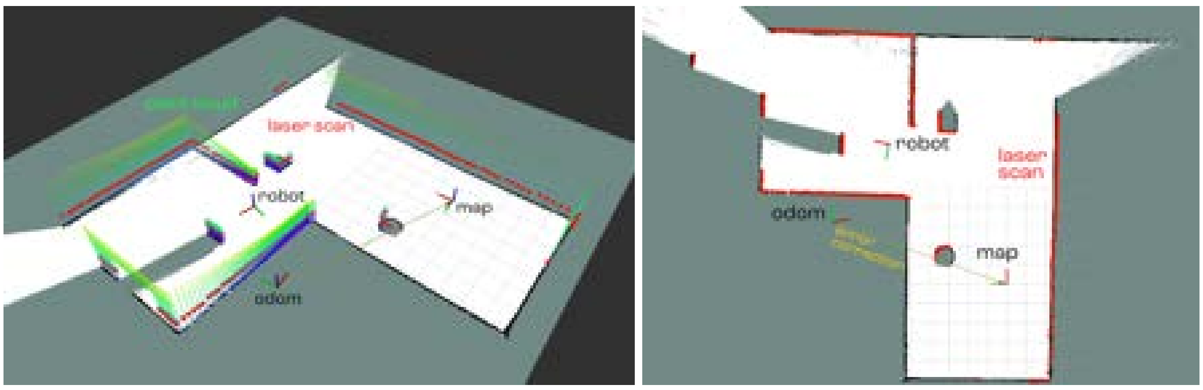

(a) Point cloud without collisions. (b) Point cloud with a detected collision.

Figure 14: The raw point cloud obtained from the OS1 in a crowded environment is visualized in RViz using a color mapping. As a person surpasses the thresholds of the critical zone, the corresponding points are visually accentuated by applying red coloring.

3.4.3 SLAM

Additionally, RViz is utilized to visualize the map created by the `slam_toolbox` to offer insight into the SLAM algorithm. The center slice of the three-dimensional point cloud is converted to a two-dimensional `LaserScan` message. The `slam_toolbox` then creates a map based on this message, representing the environment in two-dimensional space [12]. As seen in Figure 15, the `odom` coordinate frame is transformed with respect to the `map` frame by its error each time features of the new laser scan measurements fit the available map. The total distance between the `odom` and `map` frame is representative of the total long-time error of the state estimation without SLAM. By choosing the robot's pose with respect to the `map` frame instead of the `odom` frame, state estimation without drastic long-time error is possible. In conclusion, continuous visualization of the map generation and frame transformation offers insight into an otherwise complex algorithm.



(a) Orbit view. (b) Top view.

Figure 15: The three-dimensional point cloud is first converted to a two-dimensional `LaserScan` topic. The `slam_toolbox` then utilizes this laser scan to create a map of the environment. New laser scan messages are continuously compared to the existing features to determine the correct position of the robot.

4 Results and Discussion

This Chapter describes the experimental testing of the implemented software features. First, the experimental setup is explained to give information about how these experiments were carried out and the type of results that can be expected. Subsequently, the results are described in detail and discussed in terms of reliability and importance to this thesis.

4.1 Control System

4.1.1 Localization

To measure the performance of the localization algorithm, it is tested for positional accuracy and long-term positional error (drift). The mentioned results refer to a motion combining linear and angular motion. From the starting position (0,0) the robot walks forward to (1,0), rotates 180° to move to (-1,0), where it rotates 180° to finally return to the starting point. During this motion, the positional accuracy and performance of the localization are supervised qualitatively. The measured difference in position between the final and the starting position is used to derive the long-term positional error.

This setup has been chosen because the linear movement allows for better supervision while the angular rotation brings the localization to its limits. These experiments were conducted in a static environment with multiple types of features of different sizes and complexities like walls, windows, tables, and chairs.

Localization through Sensor Fusion
At first, localization relying solely on the data from the LiDAR's IMU and the position provided by the stereo camera was tested. In theory, the combination of relative IMU measurements and the absolute pose calculated from environmental features should be sufficient for reliable state estimations. Therefore, the Kalman filter has been configured to receive the LiDAR's IMU data as relative measurements to be corrected by the absolute pose from the stereo camera.

The bias of the IMU led to noticeable drifts in the pose prediction, which were then corrected by the pose of the stereo camera. This problem can be resolved by either filtering the IMU measurement prior to the state estimation or entirely removing the IMU from the Kalman filter setup. Since the bias of the IMU is arbitrary, removing it can be challenging and successes are not guaranteed. However, the ZED2 stereo camera already offers sensor fusion with the internal precalibrated IMU of the camera resulting in almost drift-free state estimations.

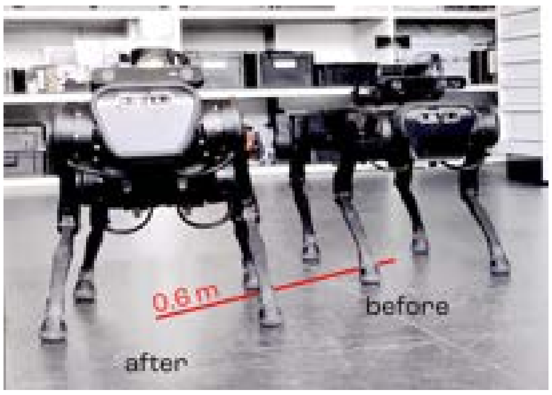


Figure 16: After a linear motion with two 180° rotations and only using the sensor fusion via EKF, the robots home position drifted to the left by 0.6 m.

Even if the stereo camera is fused with its IMU, the resulting pose is far from perfect. For small movements in high-contrast environments, the stereo camera is able to detect enough features to match and extract a position from. Contrary, if the environment consists of low-contrast objects like plain white walls or if the robot rotates rapidly or explores new territory, the camera simply does not possess enough features to perform loop closure. Another disadvantage is the limited field of view, resulting in the inability to detect features on the sides or back of the robot. In practice, this has led to the positional error of up to 0.6 m for a simple linear motion with two 180° rotations which can be seen in Figure 16.

Localization with Added Mapping
In contrast to the stereo camera, the OS1 delivers spatial information 360° around the robot, resulting in more reliable state estimation. To extract features from the three-dimensional point cloud, the central horizontal slice is converted into a two-dimensional laser scan. Based on the laser scan the SLAM package `slam_toolbox` constructs a map of the environment. New laser scans are then compared to this map and if reoccurring features are recognized the current position is calculated.

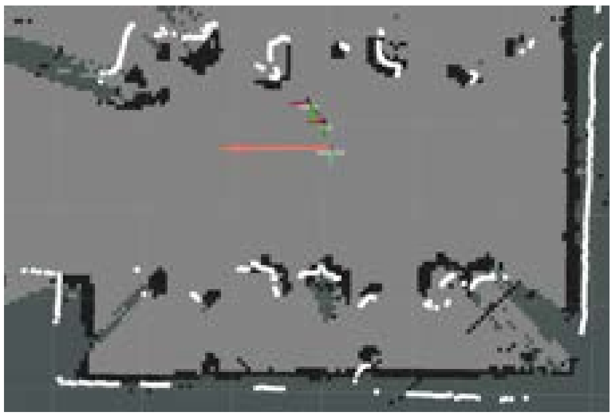


Figure 17: The SLAM algorithm creates and updates a two-dimensional map (gray/black) based on laser scans (white). The correction of the state estimation (red arrow) is performed by updating the transformation of the `odom` frame with respect to the `map` frame. A video of the SLAM visualization can be found here.

An example of the mapping and the resulting state estimation can be seen in Figure 17. The white pixels represent the measurements of the latest laser scan with respect to the `map` coordinate frame. These pixels are then converted by the SLAM algorithm, resulting in the black and gray map. Features are marked black on the map, while the space in between is filled with gray pixels. If enough features of the laser scan match the environmental map, the pose of the robot (red arrow) is corrected by transforming the `odom` frame with respect to the `map` frame. Consequently, new pose estimations of the stereo camera, which are calculated with respect to the `odom` frame, are corrected by the aforementioned transformation.

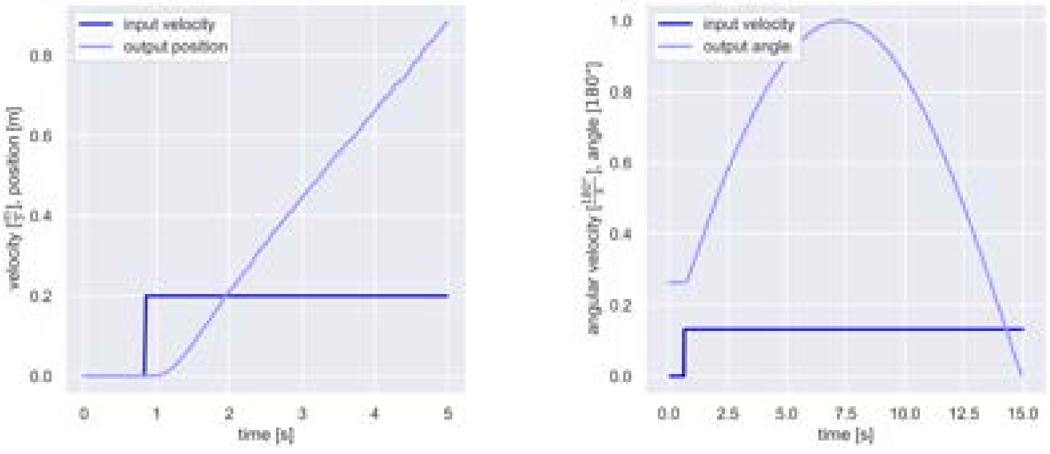
In most cases, the mapping resulted in a successfully corrected pose while producing a correct map of the environment. The quality of the map and the corresponding state estimation is highly dependent on the complexity of the environment. Clearly defined objects like walls or large boxes result in larger and less complex features, resulting in better performance of the state correction. Environments with more diverse objects, like the example in the last paragraph, propose a larger challenge to the SLAM algorithm. Smaller reoccurring features are more likely to be mismatched, leading to incorrect corrections, subsequently propagating into the map. In consequence, the positional drift measured less than 0.2 m as long as the map was corrected successfully. In cases where the robot's velocity was too high or too many features changed during the motion, the resulting error was larger than without SLAM.

4.1.2 Position Control

In the following Subchapter, the mentioned velocities refer to the local coordinate system of the robot while desired positions and orientations are described in the global coordinate system. Specifically, velocities in the x-axis refer to forward or backward motion, and velocities in the y-axis to sideways motion.

Since the absolute position of the robot in space cannot be measured without additional measurement equipment, the position generated by the localization algorithm without SLAM, mentioned in the last Subchapter is used. An environment with optimal features has been chosen to reduce localization errors, which would propagate into the position control measurements.

Step Responses
To gain information about the latency of the velocity control system, a step response measurement can be performed. From these measurements, valuable information about the system characteristics can be determined which later influence the controller design. Since only the x- and y-axis linear velocity and the z-axis angular velocity can be controlled, three-step measurements have been performed. As seen in Figure 18, the latency of the control system is lower than 0.1 seconds. Since this latency already includes the latency of the LCM server in combination with the Unitree SDK, it will be sufficient for this application. Apart from the latency, the system response shows integrating behavior - in other words, for a constant input velocity the output position or angle increases constantly. As a result, a controller which only controls the input velocity proportional to the positional error is sufficient to reach the goal position without any static error.



(a) Step response in the x-axis. (b) Step response of z-axis rotation.

Figure 18: The step response compares the input velocity with the robot's output position and angle. Both linear and angular step responses show integrating behavior with minor latency.

If applied to the real system, the saturated proportional controller mentioned in Equation 1 performs as seen in Figure 19. The controller was programmed to rotate the robot to 180° and back. First, the error between the desired and actual orientation is larger than the saturation threshold. Therefore the controller output is set to its maximum of 36 $\frac{\circ}{s}$. After around 2.5 seconds, the controller starts lowering the output velocity relative to the remaining angular distance, ensuring a smooth approach to the desired orientation. To reach the initial position of 0°, the negative error in orientation leads to a negative angular speed, again resulting in the movement towards the goal orientation. It has to be noted that the control parameters for this example have been chosen lower than for the final controller configuration to improve this visualization.

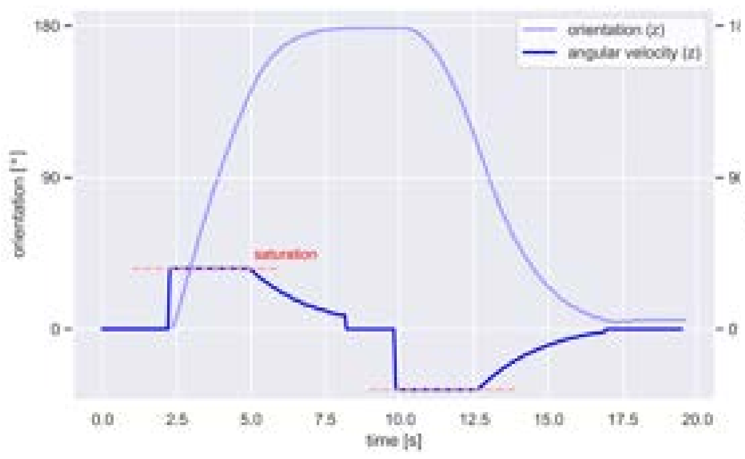


Figure 19: Angular velocity and resulting orientation during a 180° rotation with a saturated P-controller (tuned to less dynamic motion for visualization purposes).

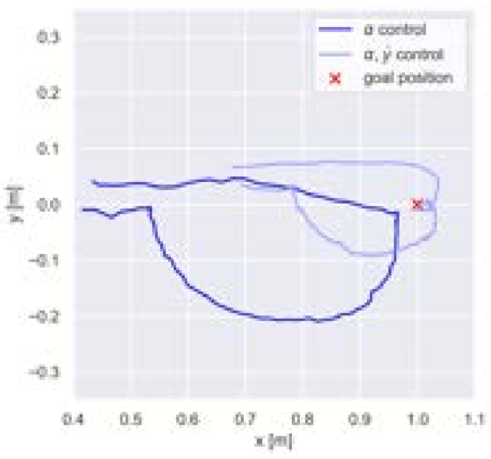


Figure 20: The deviation in the end position after a 180° rotation can be reduced by more than a factor of two with additional y-axis control.

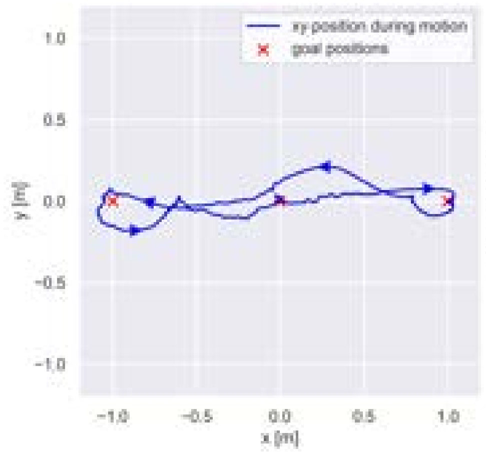
Positional Drift During Rotation
Due to variations between the model used by the Unitree SDK and the real robot, the robotic kinematics and the resulting movements contain significant deviations. Assuming a perfect model, if only the angular velocity is controlled to rotate the robot by 180°, there should be no error in position afterward. In the real system, however, this error is as large as 0.4 m for a rotation of 180°, leading to unwanted deviations from the path. To restrain this behavior the linear velocity of the robot has to be controlled during the rotation. As seen in Figure 20, this lowers the positional error significantly by more than a factor of two. Experimental testing has shown that additionally manipulating the x-axis linear velocity does not improve the positional error. This could be due to the fact, that the robot generally drifts only in y-direction.

Walking on a Straight Line
Based on the findings mentioned in the previous paragraphs, the controller has been tuned empirically to perform steady motion with minimal deviations from the desired path while still moving dynamically. This has been achieved with the following parameters:

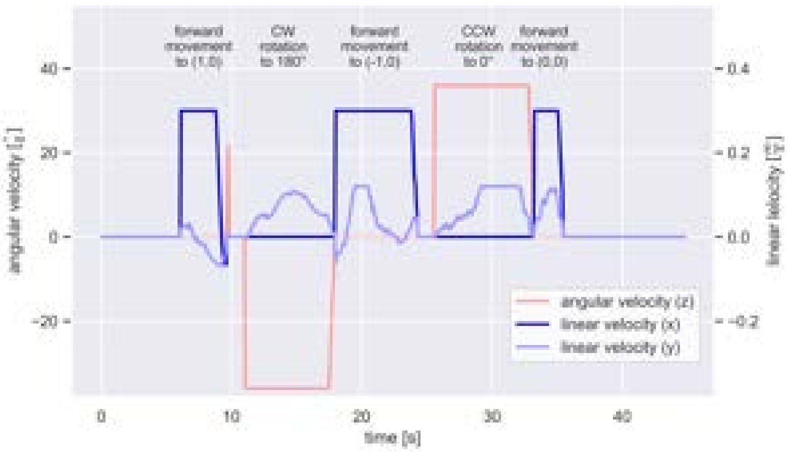
Parameter	Variable	Gain	Saturation Value	Unit
Linear velocity gain for movement in x-axis	k_x	2.5	±0.3	$\frac{m}{s}$
Linear velocity gain for movement in y-axis	k_y	2.0	±0.12	$\frac{m}{s}$
Angular velocity gain for rotation around z-axis	k_α	1.0	±36	$\frac{\circ}{s}$

Table 2: Empirically determined controller parameters and their saturation limits for the controller described in Equation 1.

The resulting behavior can be seen in Figure 21a, where the motion to three desired end-points ((1,0), (-1,0), and (0,0)) is performed. With a maximum deviation of around 0.2 m the motion pattern of the controller can be considered sufficient. When observed by eye, the deviation from the desired path is almost negligible. More importantly, the robot reaches the desired position quickly and efficiently without unnecessary movements. As seen in Figure 21b this maneuver takes no longer than 30 seconds.



(a) Position on xy-plane.



(b) Linear and angular velocities.

Figure 21: The executed path on the xy-plane shows a maximal deviation of 0.2 m during the desired motion (0,0) →(1,0)→(-1,0)→(0,0). The corresponding control velocities show how this motion has been achieved efficiently with a saturated p-controller.

Both linear (x) and angular output (z) velocities of the position controller are set to their maximum for most of the time, resulting in fast and dynamic movement. These thresholds have been chosen to ensure safe movements and prevent overshooting the target position.

While the linear velocity in the x-direction is responsible for forward motion and the angular velocity around the z-axis controls the orientation, the linear motion in the y-direction minimizes positional errors due to either drift or misalignment. Without this correction, the controller would never be able to reach the desired position if the initial orientation or the linear movement shows imperfections.

4.1.3 Discussion

Both the localization and position control satisfy the requirements of dynamically conducting motion from one point to another while compensating for errors. Through feedback control, the robot is able to dynamically adapt to mechanical imperfections and reach the desired position nevertheless. Although deviations of up to 0.2 m occur during linear motion over 2 m, they are barely visible. Long-term positional drift however is critical during prolonged periods of live demonstrations. If the robot has to be manually reset after performing a few tricks, the overall impression of the robot is compromised. The added SLAM algorithm successfully solves this problem by contributing absolute measurements and resetting the positional error periodically. In static environments, these corrections led to less than 0.2 m of drift after performing multiple motion sequences, thus satisfying the requirements for a stable demonstration. However, dynamic environments like crowds pose potential difficulties to both the SLAM algorithm and the stability of the ZED2 camera. These rapid changes in features have to be tested further, ideally during a live demonstration.

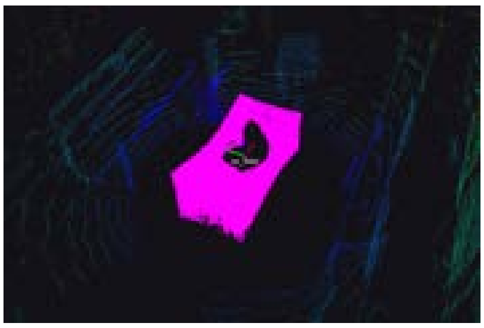
4.2 Collision Detection

To prevent the robot from accidentally coming into conflict with objects or people when performing actions, the `rbd_lidar` package implements a collision detection algorithm to recognize points within a critical zone around the A1. This zone is cuboid-shaped and defined by four perpendicular distances in the x and y direction to the OS1, which can be specified via the `default.yaml` file. Measurements within the critical zone are counted and cause the robot to halt if a certain numerical threshold of points is exceeded. As default values for the collision zone, the front critical distance is set to 700 mm, the back distance to 550 mm, and the lateral spacing amounts to 350 mm on each side. These values were determined empirically and proved to allow unobstructed interaction with the robot while still effectively avoiding collisions. However, if the robot is used in more uncommon environments, for example with numerous reflective surfaces, the distance thresholds might need to be readjusted, as these surfaces cause an accuracy reduction in the LiDAR measurements.

The following paragraphs outline and discuss the experimental validation of the collision detection algorithm. As measures of its quality, these experiments fathom the shape of the critical zone and the algorithm's reliability. Video proof of the collision detection and its visualization can be found here

4.2.1 Shape of Critical Zone

To consider the location of the OS1 and the robot's dimensions, a cuboid-shaped collision zone was defined. Since the robot moves the fastest in the forward direction while performing its actions, the critical distance is chosen as the longest in this direction. Due to the polar measurement mode of the OS1, the zone is vertically bounded by curved edges. The correctness of the expected geometry of the collision zone was verified experimentally by measurements. The goal of these measurements was to obtain a spatial representation of the zone, within which measurement points are effectively detected as collisions. For this purpose, the robot was placed on a free surface and the collision detection was launched. In RViz, the measured LiDAR point cloud was displayed and points within the critical zone were accentuated. The illustration was configured to preserve these points for one minute while displaying the remaining point cloud in real-time.


(a) Shape of the collision zone.

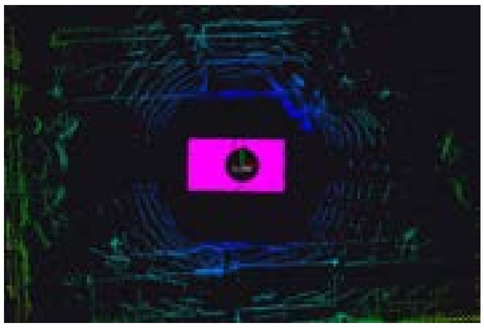
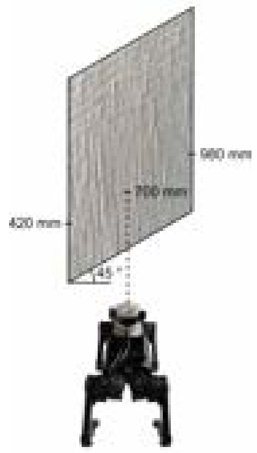

(b) Spherical blind zone.

Figure 22: The verification of the critical zone displays all the expected characteristics. These measurements illustrate the cuboid-shaped geometry vertically constrained by curves and a spherical blind zone in the center of the point cloud.

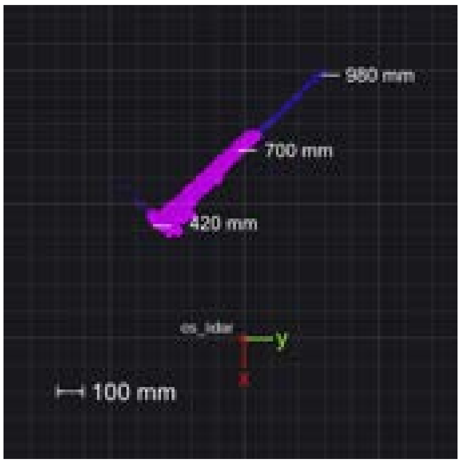
To verify the geometry of the collision zone, a diffuse reflective, planar surface was moved around the robot to record collision points. The time-delayed fading of the measurements resulted in a three-dimensional image of all points detected as collisions. The corresponding result is shown in Figure 22a. The measurements illustrate, that the collision zone perceived by the system is cuboid-shaped and vertically constrained by curved edges. The blind zone defined by the minimum measurement distance of the OS1 is visible in Figure 22b and can be recognized in the center of the point cloud as a spherical void. The verification of the zone's dimensions is addressed in the subsequent Subchapter.

4.2.2 Reliability

Accuracy
To verify the accuracy of the collision detection, measurements on a planar test target angled by 45° were conducted with the experimental setting illustrated in Figure 23a. The test plane was clothed in fabric (non-ideal Lambertian target) to simulate a target surface, such as a person's pants, when demonstrating the robot to an audience. The OS1 reaches a range accuracy of ±30 mm for Lambertian targets and ±100 mm for retroreflectors, with a precision of ±7 mm for targets closer than 1000 mm. To account for non-idealities, a tolerance of ±80 mm was defined within which the boundary found in the experiment between points detected as collisions and those not detected must lie. For a distance of 700 mm (default critical distance at the front of the A1), the corresponding output point cloud of the collision avoidance algorithm is shown in Figure 23b. In accordance with the aforementioned tolerance, the effective threshold must lay in a range of 620 mm to 780 mm. The illustration indicates that the effective critical threshold perceived from the LiDAR measurements is located at around 750 mm instead of 700 mm, which fulfills the requirement. For all other tested distances (350 mm and 550 mm), the algorithm showed similar or better results.



(a) Experimental setting for distance verification.



(b) Resulting point cloud from a top view.

Figure 23: With the experimental setting displayed in Figure 23a, the point cloud from Figure 23b is obtained. This result illustrates that the effective critical threshold perceived by the system is offset from the target distance by approximately 50 mm.

Reaction Time
The reaction time of the LiDAR-based collision stop is a quantity for the reliability of the collision detection since the robot needs to be stopped from moving into people or objects. This parameter was analyzed by rapidly moving a fabric surface into the A1's critical zone and measuring the time for the robot to not displace itself anymore. As not only the reaction time of the software is relevant, but also the delay caused by mechanical inertia, the system's overall

reaction time was evaluated by hand using a stopwatch. In twenty measurements, a reaction time of 0.44 s $\pm$ 0.09 s has been found. This standard deviation of 0.09 s contains the deviations caused by the mechanical system as well as those of the time measurements. Statistically, more than 99.7% of the measurements lie within a three-times standard deviation from the mean value. Therefore, the maximum reaction time of 0.44 s plus three times the measured standard deviation of 0.09 s (0.70 s) must lie within the robot's maximum allowed reaction time.

As mentioned in the previous Subchapter, for fabric surfaces, a maximum deviation of the LiDAR measurements of approximately 50 mm was measured. Considering this deviation, the robot's dimensions, and the currently implemented robot movements, a sideways motion with the maximum lateral controller output speed of 120 $\frac{\text{mm}}{\text{s}}$ is the most critical movement. With the robot's width of 400 mm and the lateral critical distances of 350 mm on each side, a critical space of $350\,\text{mm} - \frac{400\,\text{mm}}{2} = 150\,\text{mm}$ to the left and right of the A1 can be obtained. Therefore, a maximum reaction time of $\frac{150\,\text{mm} - 50\,\text{mm}}{120\,\frac{\text{mm}}{\text{s}}} = 0.83\,\text{s}$ is allowed (considering the maximum LiDAR measurement deviation of 50 mm). Since this is higher than the achieved 0.70 s, the achieved reaction time can be considered appropriate, which is consistent with the circumstance that no collisions occurred during the laboratory tests.

4.2.3 Discussion

To achieve safe and reactive behavior during the interaction with humans, a collision detection algorithm was implemented in the ROS package `rbd_lidar`. For each side of the robot, the distance below which a person or object is detected as being at risk of collision can be specified dynamically at each restart. These critical distances result in a cuboid-shaped collision zone with a blind zone given through the minimum range of the OS1.

While the OS1 sensor is capable of providing a large amount of information at a high rate, it is subject to certain limitations in terms of accuracy, which is why range measurement deviations of up to 100 mm can be expected depending on the target material. For fabric surfaces, accuracy was found to be in the range of approximately 50 mm, which was sufficient for collision prevention in all laboratory tests. However, if safety requirements have to be increased, the critical distances should be chosen correspondingly higher to compensate for measurement inaccuracies. Due to latencies in various parts of the system, the reaction time is limited to around 0.44 s $\pm$ 0.09 s, during which the robot continues to move at its current velocity. In laboratory experiments, the default values set for the critical distances were sufficient to prevent all possible collisions. However, if the robot is moved more dynamically in an application or if highly reflective objects are located nearby, it may be appropriate to enlarge the collision zone to guarantee a high degree of safety.

Through the collision detection algorithm implemented in the `rbd_lidar` package, a LiDAR-based collision avoidance capable of detecting points within a critical zone around the A1 is realized. This allows the safe execution of movements without accidentally coming into conflict with people or objects near the robot. Depending on parameters such as movement velocity and security requirements, the dimensions of the critical zone can be enlarged to enhance the security level.

4.3 Visualization and Live Demonstrations

4.3.1 Visualization

The overall goal of this work is to provide a demonstrator that conveys AI to a broad public. To make the system intuitively understandable, visualization measures are required to illustrate the main AI logic as well as the most important auxiliary processes, which is achieved by using existing graphical visualization tools.

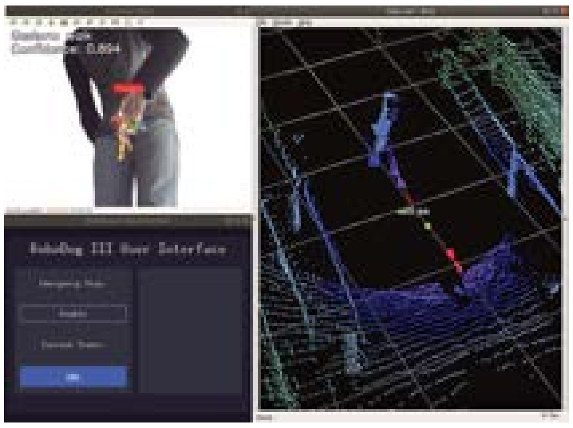
The final visualization used when demonstrating the robot to an audience consists of three main elements interacting with each other:

i. Detecting and classifying hand gestures was implemented in the previous project thesis and the corresponding visualization is integrated in this work.

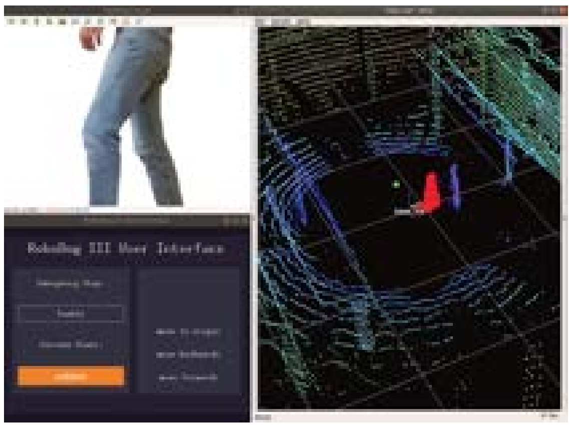
ii. The GUI implemented in this thesis allows the audience to grasp which state the robot is currently in and what its next planned actions are. For example, the state changes to `collision` if a person approaches the robot too close and a collision is detected. The comprehensibility of the system's inner workings is further enhanced by a visualized queue of the next tasks the robot intends to perform. Thus, the robot's reaction to a gesture does not simply remain a physical movement, but it becomes a logical sequence of control actions.

iii. Finally, the point cloud obtained by the LiDAR measurements is displayed in RViz using *rainbow* color mapping to provide depth information. Objects or people entering the critical zone around the robot are highlighted in red color.

Figure 24 illustrates the interaction and individual tasks of the three subcomponents that build the final visualization. Figure 24a displays the gesture recognition procedure: To register a gesture, the robot lowers its hips and thereby enters the *attentive* pose. The controller is now in the `idle` state, which is acknowledged with a corresponding blue label in the GUI. The camera stream for gesture recognition is running and ready to register a video sequence.



(a) Gesture recognition procedure.



(b) Reaction to a prevented collision.

Figure 24: While in the `idle` state (blue label in the GUI), the key points used to determine a gesture are visualized live in the video window. If a person or an object enters the collision zone, the state changes to `collision` (orange label), and the colliding points in the point cloud are marked red.

If a person moves in front of the robot to perform a gesture, the gesture recognition implemented in Python 3.10 detects hand key points, classifies the gesture presented, and displays the classification output in the same window, as shown in the upper left of Figure 24a [7]. Using rosbridge, the detected gesture is forwarded from Python 3.10 to ROS running on the A1. The waypoints calculated based on the recognized gesture are added to the GUI's action queue, which is visible in the lower left of Figure 24b. During the execution of the corresponding action, the robot is in `running` state. As soon as a person approaches the robot, or vice versa, the `rbd_lidar` node detects a possible collision and the `rbd_controller` node executes a stop reaction. The collision is visualized in the corresponding point cloud by red color and causes a state change to `collision`, visible as an orange label in the GUI display. An example of the three visualization components in the case of a collision is displayed in Figure 24b.

4.3.2 Live Demonstrations

In order to verify the control system, the comprehensibility of the created visualizations, and the collision detection as well as to get feedback for the optimization of the system, the developed robotic AI demonstrator was presented at the Swiss Academy of Engineering Sciences (SATW) annual congress in the context of a trade fair. Gestures were performed in front of the robot, causing it to carry out the corresponding actions. The illustration of the system's workings was done both by the robot's physical movements and the visualizations of the main algorithms on an external monitor.

System Performance
The large number of people at the trade fair resulted in an unsteady environment in which the robot had to localize itself. However, the fusion of the stereo camera data and the LiDAR scan allowed the A1 to successfully determine its position, even among a significant number of moving spectators. Highly dynamic environmental conditions were found to disturb the stereo camera localization and to sporadically cause noticeable repositioning due to the SLAM algorithm of more than 0.3 m, which is enabled by the LiDAR scan data.

Throughout the entire period of the demonstration, moving the robot into objects or people was prevented through the collision detection algorithm. The chosen default values for the collision zone dimensions proved to be appropriate to detect collisions, considering measurement inaccuracies of the OS1 and reaction time of the collision stop.

Overall, the robot was able to arouse a significant amount of interest from a broad audience with solid basic technical knowledge but varying levels of expertise in the field of robotics. Throughout the entire audience, the visualizations were perceived as intuitively understandable. The hand key points displayed in the illustration of the gesture recognition algorithm were intuitively linked with the topic of AI. Associating the robot's physical reaction with the processes in the software was supported by the task queue corresponding to the robot's next actions. Using red as an alarm color to visualize points causing the detection of a collision, proved to be intuitively graspable for the majority of people.

Feedback and Observations
In the course of the presentation at the SATW annual congress, feedback of various kinds was obtained, reflecting the interests of the audience. It became apparent that additional status displays can prove useful to enhance the comprehensibility of the processes in the robot. Therefore, additional label colors for the states (`collision`, `idle`, and `running`) have been integrated into the GUI after the presentation.

With the current implementation, the robot is prevented from actively moving into objects or people. However, a frequently asked question was whether the robot was able to dodge approaching individuals, which should be considered in future projects based on this thesis.

As mentioned in the preceding project thesis, illumination changes can affect the reliability of the gesture detection [7]. Additionally, exactly reproducing the movement patterns that had been used to train the logistic regression model that classifies the gestures caused issues in some cases. Furthermore, various visitors attempted to perform gestures exceeding the pre-trained four gestures.

4.3.3 Discussion

To guarantee secure interaction with humans, a control system using LiDAR-based collision detection was implemented in this thesis. Additionally, visualization measures have the purpose to provide diverse information about the system's functionality and, therefore, act as a human-machine interface. Both could be presented at the SATW annual congress in the context of a trade fair.

The system proved its stability, also in a dynamic environment. Even in dynamic surroundings, the combination of stereo camera and OS1 enabled localization, while the collision detection algorithm enhanced the system's security level significantly.

The audience is notified about details of the gesture recognition process, which supported users in understanding how and when their hand movements were observed by the robot. As the robot performs an action, the understanding of why and how it behaves in a certain way is enhanced by a state display and a task queue. The visualization of the point cloud obtained from LiDAR measurements displays how the robot perceives its environment and allows a direct link between a person's presence near the robot and its reaction, which was well-comprehensible for the audience.

New and helpful feedback was obtained at the aforementioned trade fair, providing various system optimization proposals, one of which has already been integrated.

5 Conclusions and Outlook

5.1 Main Achievements and Interpretation of the Results

First and foremost, the sequential motion control script was successfully replaced by a modular soft real-time system able to tackle more complex tasks efficiently. Through parallel execution of multiple programs, i.e. ROS nodes, the control system is able to handle multiple tasks at once. Multitasking is crucial for the safe execution of motion in a public space where collisions cannot be tolerated. The collision avoidance algorithm prevents all collisions caused by the robot's movement by stopping if an object is detected within a safety bounding box. During all of the conducted testing and live demonstrations, the developed safety system has proven its reliability and no collision has occurred.

During the live demonstrations, the robot impressed people with different levels of expertise. Due to its resemblance to a real dog, the robot dog caught the attention of most of the spectators quickly. People outside of the field of robotics understood the goal of the robot - executing a trick based on a gesture - intuitively. Nevertheless, the chosen approach of a ROS-based control system also caught the interest of experts in robotics and AI.

With the addition of a stereo camera and a LiDAR sensor, environmental perception allows the robot to not only prevent collisions but also to estimate its position and orientation. A Kalman filter fuses the stereo camera data with the inertial measurements to generate a pose with higher bandwidth and lower long-time error than a pose generated by one of them individually. Additionally, a SLAM algorithm creates a map based on the horizontal slice of the LiDAR point cloud. If features of the newest LiDAR measurements match existing map features, the localization node corrects the pose estimation accordingly. This correction with absolute measurements allows the robot to perform longer without having to be restarted. Furthermore, the overall impression is improved since the robot accurately returns to its initial position.

Extending the existing visualization of the key points generated by MediaPipe, the ROS-based visualization tool RViz allows visualizations of all ROS topics. Namely, the visualization of the three-dimensional raw point cloud as well as the illustration of the detected collisions provides insight into how the collision algorithm decides when to stop. Furthermore, RViz visualizes the two-dimensional laser scan based on the identical raw point cloud data and the resulting map. This map offers information on how the robot perceives its environment and how the current location is determined. Not only do they offer insight during live demonstrations, but these visualizations are also valuable during the development phase. Complex data structures like point clouds or coordinate frame transformations can be visualized intuitively without the need for additional code development.

In conclusion, the change to a modular ROS-based control system proved to be the right choice. Although this change required additional work to achieve the same high-level goal that the project thesis already had achieved, the resulting software stack allows for faster development in the future. This has also been the case for the features added in this thesis. The collision avoidance system was developed segregated from the control system and tested on its own. After these tests were successfully completed, the integration was possible with minimal effort due to clearly defined communication interfaces i.e. ROS messages and services.

5.2 Limitations

The limiting factor during live demonstrations is the runtime of the two available batteries. Since the WLAN hotspot turns off after a discharge of around 60%, and the communication between the gesture recognition on the host laptop and the robot depends on this WLAN connection, the battery cannot be utilized to its full potential. In combination with the added LiDAR sensor and stereo camera, the demonstration can be conducted for a maximum of 20 minutes until the battery has to be changed. As a partial solution to this problem, a third battery has been ordered.

A less common problem during live demonstrations is that the robot has to be restarted due to the fact that it cannot actively walk away from obstacles. If the robot comes too close to a static object e.g. a wall, it will stop to prevent a collision. With the current implementation of the collision avoidance system, the robot will remain stationary forever. However, thanks to the LiDAR measurements and the map created by the SLAM algorithm, information about where the robot could move is already available.

5.3 Ideas for Future Projects

Feedback Hardware

Live demonstrations have shown that the audience focuses more on the robot than the visualization when interacting with it. Therefore, feedback could be integrated into the robot hardware via a status LED. Through different colors, this LED could signal the same information as the status in the GUI. For example, if a gesture is recognized but the hand of the user is in the collision area, the robot does not move until the hand is removed from the collision area. This lack of motion can be interpreted as an unrecognized gesture. However, the GUI shows that the state has changed to `collision` instead of `idle`. A LED could signal that the gesture is recognized while also recognizing a collision through a change of color.

In addition to the software emergency stop in the GUI, a physical emergency stop button would add haptic feedback. Even though there is no reliability issue with the software emergency stop, a physical button could improve the perception of safety and ease of use.

Gesture Recognition

The conducted live demonstrations have shown that the margin of error for the performed gestures is very low, i.e. if the gesture does not match the training data, it is not recognized. To improve this, more diverse training data needs to be collected to train the existing classifier. Furthermore, additional gestures corresponding to different motion patterns - tricks - could be trained. During the live demonstrations, people tried a wide variety of gestures, some of them not among the trained ones. Alternatively, a different machine-learning approach to processing the key points could be chosen.

However, the internal camera image will always be limited by its performance in high-contrast light situations. On the contrary, the depth image generated by the LiDAR scanner does not rely on available light. This depth image could replace or be combined with the RGB image.

Active Collision Avoidance

Although the implemented collision avoidance system reliably prevents collisions caused by the robot's motion by stopping it, the robot's reaction is limited if the obstacle remains. Active collision avoidance by moving away from the colliding object could improve this. As a result, the robot would not have to be reset, if it comes too close to a static object e.g. a wall.

This could be either achieved with a low-level algorithm, simply moving the robot away from the colliding object, or with a more sophisticated, map-based algorithm. The map created by the SLAM algorithm could help to achieve this by changing the motion path depending on the environment. If this environment changes, the path can easily be recalculated and the object evaded.

The needed path between two points on a map can be computed with conventional algorithms like rapidly exploring random trees (RRT) or probabilistic roadmaps. This path can then be changed dynamically if the environment changes. However, a more modern approach relying on machine learning would fit this AI demonstrator even better.

Reinforcement Learning

In terms of new machine learning algorithms, reinforcement learning poses fascinating opportunities for improving the gait motion of the robot. Instead of relying on fixed, predefined gait patterns, an agent can be trained to control the robot based on a learned policy. This training can be done in existing simulation tools like the NVIDIA Omniverse Isaac Sim$^{\text{TM}}$ [38] or the MuJoCo advanced physics simulations [39].

While MuJoCo offers cross-platform compatibility through C++ and Python and a simple API, the NVIDIA Isaac Sim$^{\text{TM}}$ is more advanced but requires proprietary hardware. However, the Unitree A1 is equipped with a compatible NVIDIA Jetson board with a powerful graphics processing unit (GPU), currently not used to its full potential. The NVIDIA Isaac Sim$^{\text{TM}}$ also offers a ROS interface and can simulate measurements from the ZED2 camera. Robot models of the Unitree A1 are available for both options, reducing the amount of work required to get started.

6 Bibliography

References

[1] C. Angle, "Genghis, a six legged autonomous walking robot," Doctoral Thesis, Massachusetts Institute of Technology, 1989. [Online]. Available: https://dspace.mit.edu/bitstream/handle/1721.1/14531/20978065-MIT.pdf?sequence=2 (visited on 05/18/2023).

[2] A. Winkler, "Optimization-based motion planning for legged robots," Doctoral Thesis, ETH Zurich, Zurich, Switerland, May 2018. [Online]. Available: https://www.research-collection.ethz.ch/handle/20.500.11850/272432?show=full (visited on 05/15/2023).

[3] S. Jeon and S. Kim, "Real-time Optimal Landing Control of the MIT Mini Cheetah," Oct. 2021. [Online]. Available: https://arxiv.org/abs/2110.02799 (visited on 05/18/2023).

[4] J. Lee, J. Hwangbo, and M. Hutter, "Robust recovery controller for a quadrupedal robot using deep reinforcement learning," Jan. 2019. [Online]. Available: https://arxiv.org/abs/1901.07517 (visited on 05/15/2023).

[5] ANYbotics. "Meet ANYmal, your new inspector." (2023), [Online]. Available: https://shorturl.at/bzIPR (visited on 03/18/2023).

[6] M. Patel, G. Waibel, S. Khattak, and M. Hutter, "Lidar-guided object search and detection in subterranean environments," in *2022 IEEE International Symposium on Safety, Security, and Rescue Robotics (SSRR)*, IEEE, Nov. 2022, pp. 41–46. DOI: 10.1109/SSRR56537.2022.10018684.

[7] T. S. Langdun, M. Oswald, T. Stadelmann, and P. Sager, "Building a vision-based ai demonstrator with unitree a1 quadruped robot," Dec. 2022. [Online]. Available: https://www.zhaw.ch/storage/engineering/institute-zentren/cai/studentische_arbeiten/Herbst_2022-CVPC/PA-RoboDog.pdf (visited on 03/01/2023).

[8] G. Bellegarda, Y. Chen, Z. Liu, and Q. Nguyen, "Robust high-speed running for quadruped robots via deep reinforcement learning," in *2022 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, IEEE, 2022, pp. 10 364–10 370.

[9] M. Sombolestan, Y. Chen, and Q. Nguyen, "Adaptive force-based control for legged robots," in *2021 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, IEEE, 2021, pp. 7440–7447.

[10] X. B. Peng, E. Coumans, T. Zhang, T. Lee, J. Tan, and S. Levine, "Learning agile robotic locomotion skills by imitating animals," *arXiv preprint arXiv:2004.00784*, Apr. 2020. [Online]. Available: https://arxiv.org/abs/2004.00784 (visited on 04/07/2023).

[11] J. Chen and F. Dellaert, "A1 slam: Quadruped slam using the a1's onboard sensors," *arXiv preprint arXiv:2211.14432*, Nov. 2022. [Online]. Available: https://arxiv.org/abs/2211.14432 (visited on 04/08/2023).

[12] S. Macenski, *SLAM toolbox*, version 1.1.6, Jun. 9, 2020. [Online]. Available: https://github.com/SteveMacenski/slam_toolbox/releases/tag/1.1.6 (visited on 05/02/2023).

[13] S. Macenski and I. Jambrecic, "Slam toolbox: Slam for the dynamic world," *Journal of Open Source Software*, vol. 6, no. 61, p. 2783, May 2021. [Online]. Available: https://www.researchgate.net/publication/351568967_SLAM_Toolbox_SLAM_for_the_dynamic_world (visited on 04/27/2023).

[14] M. Hutter, C. Gehrig, D. Jud, A. Lauber, and D. C. Belliosco, "ANYmal - a highly mobile and dynamic quadrupedal robot," ETH Zurich, Zurich, Switzerland, Oct. 2016. [Online]. Available: https://doi.org/10.3929/ethz-a-010686165 (visited on 05/22/2023).

[15] MYBOTSHOP, *A1 Software Developer Guide*, 2020. [Online]. Available: https://www.trossenrobotics.com/Shared/XSeries/A1SoftwareGuidev2.0-en.pdf (visited on 03/18/2023).

[16] T. Zhang, B. Zk, K. Okada, and Z. Weiwei, *Unitree_legged_sdk*, 2021. [Online]. Available: https://github.com/unitreerobotics/unitree_legged_sdk (visited on 03/18/2023).

[17] Open Robotics. “The ROS Ecosystem.” (2021), [Online]. Available: https://www.ros.org/blog/ecosystem/ (visited on 03/18/2023).

[18] Open Source Robotics Foundation, Inc. “ROS Graph Concepts: Master.” (2018), [Online]. Available: http://wiki.ros.org/Master (visited on 03/18/2023).

[19] J. Mace. “Rosbridge_suite.” (2022), [Online]. Available: http://wiki.ros.org/rosbridge_suite (visited on 04/01/2023).

[20] Gramazio Kohler Research. “Roslibpy: ROS Bridge library.” (2019), [Online]. Available: https://roslibpy.readthedocs.io/en/latest/index.html# (visited on 04/01/2023).

[21] T. Zhang, L. Xiao, A. Whang, D. Baldwin, G. Affonso, and B. Zk, *Unitree_ros*, 2023. [Online]. Available: https://github.com/unitreerobotics/unitree_ros (visited on 03/18/2023).

[22] T. Zhang and B. Zk, *Unitree_ros_to_real*, 2021. [Online]. Available: https://github.com/unitreerobotics/unitree_ros_to_real/tree/master/unitree_legged_real/src/exe (visited on 03/18/2023).

[23] T. Mehmood, *Qre*, 2020. [Online]. Available: https://my.hidrive.com/share/m5erkbhd12#%5C$/ (visited on 03/18/2023).

[24] A. Huang, E. Olson, and D. Moore, “Lightweight Communications and Marshalling for Low-Latency Interprocess Communication,” Cambridge, Massachusets 02139 USA, Tech. Rep. MIT-CSAIL-TR-2009-041, Sep. 2009. [Online]. Available: http://dspace.mit.edu/bitstream/handle/1721.1/46708/MIT-CSAIL-TR-2009-041.pdf (visited on 03/18/2023).

[25] P. F. McManamon, *Lidar technologies and systems* (SPIE Press monograph ; PM300). Bellingham, Washington State: Society of Photo-Optical Instrumentation Engineers, 2019 - 2019, ISBN: 1-5231-3393-7. [Online]. Available: https://www.spiedigitallibrary.org/ebooks/PM/LiDAR-Technologies-and-Systems/eISBN-9781510625402/10.1117/3.2518254?SSO=1 (visited on 04/04/2023).

[26] J. Li, H. Qin, J. Wang, and J. Li, “Openstreetmap-based autonomous navigation for the four wheel-legged robot via 3d-lidar and ccd camera,” 3, vol. 69, 2022, pp. 2708–2717. DOI: 10.1109/TIE.2021.3070508.

[27] Embedded Staff. “A crash course in UML state machines: Part 1.” (2009), [Online]. Available: https://www.embedded.com/a-crash-course-in-uml-state-machines-part-1/ (visited on 05/18/2023).

[28] T. Moore, S. Macneski, A. Klintberg, and M. Zhang, *Robot_localization*, version 2.6.12, 2022. [Online]. Available: https://github.com/cra-ros-pkg/robot_localization/tree/melodic-devel (visited on 03/28/2023).

[29] T. Moore and D. Stouch, *A Generalized Extended Kalman Filter Implementation for the Robot Operating System*. Springer, Jul. 2014, pp. 335–348.

[30] S. Zacher and M. Reuter, *Regelungstechnik für Ingenieure*, 13th ed. Wiesbaden, Germany: Vieweg+Teubner Verlag Wiesbaden, Dec. 2023, ISBN: 978-3-8348-9837-1. [Online]. Available: https://doi.org/10.1007/978-3-8348-9837-1 (visited on 04/07/2023).

[31] Ouster, Inc. “Datasheet-rev06-v2p4-os1.” (2022), [Online]. Available: https://ouster.com/downloads/ (visited on 03/03/2023).

[32] Ouster, Inc. “Software User Manual.” (2021), [Online]. Available: https://data.ouster.io/downloads/software-user-manual/software-user-manual-v2p0.pdf (visited on 03/03/2023).

[33] Ouster, Inc., *Official ROS1/ROS2 drivers for Ouster sensors*, 2023. [Online]. Available: https://github.com/ouster-lidar/ouster-ros (visited on 03/03/2023).

[34] Unitree. “Unitree A1.” (2022), [Online]. Available: https://m.unitree.com/a1/ (visited on 05/06/2023).

[35] P. Bovbel and T. Foote, *Pointcloud_to_laserscan*, version 1.4.1, 2019. [Online]. Available: https://github.com/ros-perception/pointcloud_to_laserscan/releases/tag/1.4.1 (visited on 05/12/2023).

[36] T. Schimansky. “CustomTkinter.” (2023), [Online]. Available: https://github.com/TomSchimansky/CustomTkinter (visited on 05/03/2023).

[37] Riverbank Computing Limited and The Qt Company. “PyQt.” (2022), [Online]. Available: https://www.riverbankcomputing.com/software/pyqt/ (visited on 05/03/2023).

[38] NVIDIA Corporation. “NVIDIA Isaac Sim.” (2023), [Online]. Available: https://developer.nvidia.com/isaac-sim (visited on 05/27/2023).

[39] DeepMind Technologies Limited. “MuJoCo Advanced physics simulation.” (2021), [Online]. Available: https://mujoco.org (visited on 05/27/2023).

List of Figures

List of Tables

List of Videos

7 Appendix

To lower the amount of work needed to understand the developed code base in subsequent projects, the documentation of the code itself, as well as high-level concepts are available.

7.1 Code Base

The code base itself can be found on GitHub. The code offers comments and additional README files, informing about the main purpose of the corresponding package. The code is split up into three parts, each corresponding to a different environment:

Environment	Packages	Description

`rbd_local`	`GUI`
	`utilities`
	`zhaw_pa_robodog`
		These files require Python 3.10 and have therefore to be executed in a **conda environment** with the required dependencies. They communicate via the rosbridge with the ROS packages. `utils` contains files to start all of the software by running `./rbd_startup.sh`.

`rbd_ros_local`	`rbd_gestrec`	ROS packages running on the **local Ubuntu VM** on a Laptop connected via the WLAN Hotspot. They feature the auxiliary ROS node for gesture recognition and the RViz configurations.

`rbd_ros`	`pointcloud_to_laserscan`
	`rbd_controller`
	`rbd_template`
	`rbd_lidar`
	`rbd_localization`
	`rbd_msgs`
	`rbd_navigation`
	`rbd_pos_controller`
	`utils`
		ROS packages running on the robot. They can simply be copied into the `catkin_ws` folder of the **NVIDIA Jetson** to be built on target.

7.2 Documentation

The high-level concepts of the ROS-based system can be found on a dedicated website. Additionally, documentation on how to install the available virtual machine (VM), the steps needed to get the robot up and running, the needed dependencies for developing, specifications and datasheets, a guide on how to replace the NVIDIA Jetson board and information useful for the software development can be found there. The following chapters are available:

- Introduction
- Quickstart Guide
- Introduction to ROS
- Introduction to the System
  - Available Nodes
  - Style-Guide & Clean Code
- Hardware
  - Datasheets and Specifications
  - Disassembling the Unitree A1
- Software
  - Creating a Node
  - Working with the Ouster OS1
  - Working with the ZED2
  - A Remark on Time Zones
  - Connecting the A1 to the Internet

7.3 Project Task

Deep Learning has revolutionized the way pattern recognition problems like image analysis can be solved today in practice, and is thus permeating industry and society. Yet, the technology behind the success stories remains vague and mysterious to most, to a large degree because it all happens ”in the virtual world” (inside a computer) and is less tangible than other technological artifacts involving customer-facing hardware like cars, robots, etc ”in the real world”.

Based on our ”Unitree A1” 4-legged robotic platform (equipped with cameras, microphones, and powerful onboard computers) and foundational work of a prior thesis, the goal of this BA is to identify and implement tangible demonstrators of the kind of work (in computer vision, pattern recognition and/or natural language processing) that the ZHAW Centre for AI (CAI) is carrying out in research. Possible examples include

- the robot to follow a specified person indoors
- the dog listening to spoken commands and performing respective actions
- visualizing the inner workings of the algorithms running on the robot on an attached screen (explainable AI)
- learning the locomotion of certain animals by imitating videos (reinforcement learning)

The result shall be used by the CAI at exhibitions like the ”Nacht der Technik”.

Work packages:

- Review of related work (prior thesis, blogs, scientific literature, public GitHub repos, other computer vision demonstrators like e.g. link)
- Define the final use case and scope together with supervisors
- Set up the development environment (locally, on the robot, and on our GPU cluster)
- Build an initial prototype, iterate (focus on functionality, ”wow”-factor, usability)
- Write a scientific report with a focus on motivation, argumentation, methods, evaluation & results (the BA thesis)