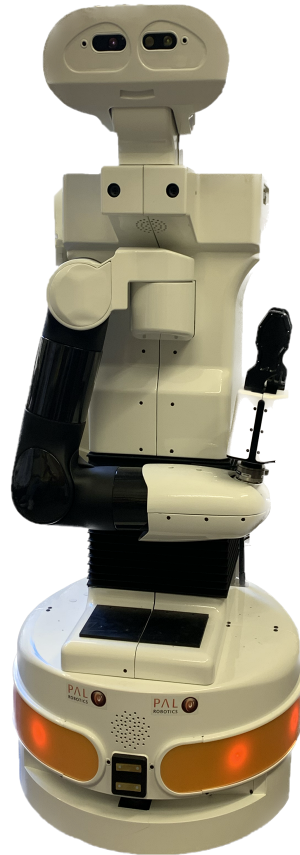
CHALMERS



Enabling Cobots to Automatically Identify and Grasp Household Objects

Cobot Cleaner EENX16-23-21

Bachelor's thesis in EENX16 SysCon

Gustav Vallin, Maja Adler, Markus Nilsson, Oskar Ståhlbom, Saga Losman and Samuel Lee

DEPARTMENT OF ELECTRICAL ENGINEERING

CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2023
www.chalmers.se

Bachelor's thesis 2023

Enabling Cobots to Automatically Identify and Grasp Household Objects

SysCon EENX16-23-21

Gustav Vallin
Maja Adler
Markus Nilsson
Oskar Ståhlbom
Saga Losman
Samuel Lee

Department of Electrical Engineering
Division of Systems and Control
EENX16-23-21
Chalmers University of Technology
Gothenburg, Sweden 2023

Enabling Cobots to Automatically Identify and Grasp Household Objects
SysCon EENX16-23-21

© Gustav Vallin, 2023.
© Maja Adler, 2023.
© Markus Nilsson, 2023.
© Oskar Ståhlbom, 2023.
© Saga Losman, 2023.
© Samuel Lee, 2023.

Supervisor: Karinne Ramirez-Amaro, Department of systems and control
Examiner: Emmanuel Dean, Department of systems and control

Bachelor's Thesis 2023
Department of Electrical Engineering
Division of Systems and Control EENX16-23-21
Chalmers University of Technology
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover: Picture of a TIAGo collaborative robot with a parallel gripper.

Typeset in LaTeX
Printed by Chalmers Reproservice
Gothenburg, Sweden 2023

Enabling Cobots to Automatically Identify and Grasp Household Objects
Gustav Vallin, Maja Adler, Markus Nilsson, Oskar Ståhlbom, Saga Losman & Samuel Lee
Department of Electrical Engineering
Chalmers University of Technology

Abstract

This thesis investigates how ROS: Robot Operating System can be combined with the real-time object detection algorithm YOLO: You Only Look Once, to make a TIAGo: Take It and Go robot clean the floor of a room. More specifically: navigate to, detect, segment, and grasp relatively small household objects on the floor of a room. Thus enabling the robot to remove objects from the floor and place them into their specific storage space, thereby cleaning the room. The project was divided into parts corresponding to the sub-tasks mentioned above, meaning each solution was coded individually to be integrated and simulated before testing was carried out on the real robot. The result of the project was that the task at hand was partially completed in a simulated environment. The robot could successfully navigate around a room, detect an object, navigate to it, grasp it, and then move the object to a specified location. Although a method to sort the object into containers was not accomplished. As for the real world tests, only parts of the grasping and object detection solutions were tested. This project can be seen as an introduction to ROS and YOLO, providing a stepping stone for similar projects.

Keywords: Robotics, Cobot, TIAGo, ROS, YOLO, Computer Vision, Machine Learning

Acknowledgements

We would like to begin by expressing our appreciation to our supervisor Karinne Ramirez-Amaro, who helped us overcome the obstacles we encountered and set up the repository that was being used during this project.

Furthermore, we would like to thank Maximilian Dihelm, who with his knowledge and valuable input has contributed to our ability to test the implementation of our solution on the real robot.

In addition, we would like to thank our examiner Emmanuel Dean for setting up the repository and creating tutorials from which we have been able to learn important concepts for our project.

List of Acronyms

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

API	Application Programming Interface
OGM	Occupancy Grid Map
ROS	Robot Operating System
TIAGo	Take It and Go
YAML	YAML Ain't Markup Language (Recursive acronym)
YOLO	You Only Look Once

Contents

List of Acronyms ix

List of Figures xiii

List of Tables xv

1 Introduction 1

2 Theory 3
- 2.1 TIAGo: Take It And Go . . . 3
  - 2.1.1 Head With a Camera for Object Search and Recognition . . . 4
  - 2.1.2 Arm With a Gripper for Object Grasping and Manipulation . 4
  - 2.1.3 Torso for Vertical Movement Control . . . 6
  - 2.1.4 Mobile Base for Movement and Navigation . . . 7
- 2.2 ROS: Robot Operating System . . . 8
  - 2.2.1 Main ROS Concepts . . . 8
  - 2.2.2 Communication in ROS . . . 9
  - 2.2.3 Tools for Perception, Planning, and Control . . . 12
- 2.3 YOLOv8: You Only Look Once, version 8 . . . 14
  - 2.3.1 Basic Machine Learning Concepts . . . 14
  - 2.3.2 Using CUDA to Accelerate Live Image Processing and Model Training . . . 15
- 2.4 Computer requirements for demanding simulations and image recognition . . . 15

3 Methods and Implementation 17
- 3.1 Enabling the TIAGo to Navigate Through a Room . . . 17
- 3.2 Object Detection and Identification . . . 19
  - 3.2.1 Training the YOLO Model on a Custom Dataset . . . 19
  - 3.2.2 Object Identification with YOLO . . . 20
  - 3.2.3 Centering Objects in the Camera Frame . . . 21
  - 3.2.4 Extracting Object Coordinates . . . 23
- 3.3 Enabling the TIAGo to Grasp Objects . . . 25
  - 3.3.1 Grasping Control Nodes . . . 25
  - 3.3.2 Grasping Action Server . . . 33
- 3.4 Integrating the Individual Solutions . . . 35

4 Results **37**
4.1 Final State and Result of the Navigation Solution 37
4.1.1 Challenges in Navigation and Obstacle Avoidance 37
4.2 Final State and Result of the Object Detection Solution 39
4.2.1 Challenges in Object Detection 42
4.3 Final State and Result of the Grasping Solution 44
4.3.1 Challenges in Grasping 46
4.4 Result of the Integration between the Three Parts 47
4.4.1 Challenges in Integration 49

5 Conclusion **51**
5.1 Performance of the Navigation: Conclusion and Future Considerations 51
5.1.1 What Worked 51
5.1.2 Continuation of the Work 51
5.2 Object Detection: Conclusion and Future Considerations 53
5.2.1 What Worked 53
5.2.2 Continuation of the Work 53
5.3 Grasping: Conclusion and Future Considerations 54
5.3.1 What Worked 54
5.3.2 Continuation of the Work 55
5.4 Integration of the Parts: Conclusion and Future Considerations . . . 55
5.4.1 What Worked 55
5.4.2 Continuation of the Work 55
5.5 Final words 56

Bibliography **57**

A Appendix 1 **I**
A.1 YAML I
A.2 Building a development computer I

List of Figures

2.1 A Scheme Displaying the Communication between the Development Computer and the TIAGo using a ROS API 3
2.2 TIAGo's Head: Includes an RGB-D Camera for Image and Depth Processing and Two Joints for Head Positioning. [[1], p. 23, fig 18. Copyright PAL Robotics] 4
2.3 TIAGo's Arm: Consists of Several Joints for Configuring Position. [[1], p. 18, fig 13. Copyright PAL Robotics] 5
2.4 TIAGo's PAL Gripper: Used for Grasping Objects. [[1], p. 21, fig 16. Copyright PAL Robotics] 6
2.5 TIAGo's Torso: Shown with Minimum and Maximum Extension. [[1], p. 15, fig 9. Copyright PAL Robotics] 6
2.6 TIAGo's Base for Mobility: Equipped with Sensors for Mapping and Navigation. [[1], p. 10, fig 3. Copyright PAL Robotics] 7
2.7 Message File with YAML Syntax 9
2.8 Service File with YAML Syntax 10
2.9 Action File with YAML Syntax 10

3.1 Flowchart for the Navigation Solution 18
3.2 Number of Annotations for Each Class in the Dataset 19
3.3 Object with Bounding Box 20
3.4 Graph of the "camera_yolo" Node Function 21
3.5 Graph of the "center camera" Node Function 22
3.6 Table Segmentation of Objects: Removing the Surroundings and Isolating Objects in the Scene 23
3.7 Segmentation of Objects 24
3.8 Objects Middle Coordinates Visualized as Green Spheres 24
3.9 Octomap Representation vs. "Real Life" View in Simulation 26
3.10 Function Scheme for Head Control Node 27
3.11 Head Positions during Pan Motion for Octomap Creation 28
3.12 Collision Detection with Octomap 28
3.13 Function Scheme for Arm Control Node Utilizing Inverse Kinematics with MoveIt 29
3.14 Function Scheme for Arm Control Node Utilizing Forward Kinematics with MoveIt 30
3.15 Complex Arm Coordination Using MoveIt to Avoid Collision with Objects on Table 31

3.16 Torso Raised After Grasping Object 32
3.17 Function Scheme for Torso Control Node 32
3.18 Gripper Grasping Object 33
3.19 Function Scheme for Grasping Server 34
3.20 Flowchart for the Integration Solution 35

4.1 Simulation Interface Showing the Laser and the Objects It Detects as well as a Planned Navigation Path 38
4.2 Visualization of the Path That the TIAGo Navigates 38
4.3 Loss Values and Metric Results of YOLO Object Detection Training . 39
4.4 Confusion Matrix Showcasing the Accuracy of the Custom Trained Model 40
4.5 Example of Object Detection on the TIAGo RGB-D Camera 41
4.6 Example of Multiple Object Detection on the TIAGo RGB-D Camera 41
4.7 Example of When the YOLO Model Does not Detect an Object . . . 42
4.8 The Green Box Is Only Detected from Some Angles 42
4.9 Example of When the YOLO Model Detects an Object with Low Confidence 43
4.10 Example of When the YOLO Model Detects an Object but Labels It Wrong 43
4.11 Example of When the YOLO Model outputs a False Positive 43
4.12 TIAGo Unfolding Its Arm During Real-Life Testing Using the Arm Control Node Utilizing Inverse Kinematics and MoveIt 45
4.13 TIAGo Picking Up a Bottle During Real-Life Testing 45
4.14 TIAGo Lowering Torso During Real-Life Testing 47
4.15 The Results of the Integration Inside the Simulation 48
4.16 A Centered Object Which Is Not Close Enough for Grasping 49
4.17 An Object in the Lower Part of the Camera Frame is Close Enough for Grasping 50

5.1 Visualization of Distance Reading from Laser 52
5.2 Gap Between Robot and Obstacle During Collision 53

List of Tables

2.1 Topics and Actions Used for Communication with the Robot's Head . 4
2.2 Topics and Actions Used for Communication with the Robot's Arm . 5
2.3 Topics and Actions Used for Communication with the Robot's Gripper 6
2.4 Topics and Actions Used for Navigational Communication with the Robot's Base 7
2.5 Tools for Perception, Planning, and Control 13

3.1 Classes Downloaded from the Google Open Image V7 Dataset 19

A.1 The Components of the Computer I

1 Introduction

Today's technology aims to relieve humans from difficult, strenuous, and repetitive tasks, either in manufacturing or private settings. When it comes to manufacturing, robots can be used to lift heavy loads, repeat demanding movements, or perform work in dangerous environments [2] [3]. In the private sector, the technology aims to help people save time and increase the independence of the elderly or individuals with disabilities [4] [5]. These robots are called Collaborative Robots (Cobots) and are most often used together with simpler tasks that cannot adapt to the dynamics of everyday life. New research in Artificial Intelligence (AI) has further developed the possibilities of Cobots and how they can help people with dynamic tasks [4] [5]. A difficulty with this is to identify and responsively decide how the Cobots should act when interacting with new, previously unseen, objects.

The purpose of this project is to enable a TIAGo robot to automatically clean a room. This means that it should be able to identify, segment, and grasp a relatively small household object such as a fruit, cup, or TV remote in order to place it in its specific storage place. For this sequence to work as intended, the robot has to succeed with the following tasks:

- Navigate toward an object
- Identify the object
- Segment the object
- Grasp the object
- Navigate to the specific storage place
- Place the object in the specific storage place
- Detect obstacles and avoid collision

In the stages of early development for a project like this one, limitations and boundaries need to be established to effectively perform the task at hand. The reason lies in the limitations of knowledge, technology, and time. There are certain limitations motivated by previous work done on the robot and in the TIAGo handbook [6] [7].

The robot was limited to a small room with bright uniform lighting. This was decided to make the identification of the objects as consistent as possible, something previous work discovered was a challenge [6]. The objects were picked up from the floor, meaning the height at which the robot operates remained the same. This limitation was set in order to ensure that the project scope would be reasonable given the time available.

2
Theory

To begin the project, extensive research was conducted and a broad range of information was gathered about the TIAGo: Take It And Go robot, its system, and architecture. Its controllable parts, modular components, and sensors were studied for a better understanding and apprehension of the inner works. The TIAGo is controlled using ROS: Robot Operating System and therefore an effort was also put into researching this system and its integrated tools used for developing robotic solutions. The project also involves the use of the YOLO: You Only Look Once object detection and image segmentation model. Overall, this chapter provides an overview of the research and theoretical foundation of this project, including information about the TIAGo robot, ROS, and YOLO.

2.1 TIAGo: Take It And Go

The TIAGo robot is a complex collaborative robot developed for research, ambient assisted living, light industry, and healthcare [4] [8]. It consists of many different sensors, components, and software that enables human-robot interaction with easy configuration [8]. Figure 2.1 shows the main components utilized in this project. These components are controlled and read using a ROS API constructed for the TIAGo robot. The main components are composed of a head with a camera for object search and recognition, an arm with a gripper for object grasping and manipulation, a mobility base for movement and navigation, and a torso for vertical movement control [1]. These components integrated with code on a development computer and the ROS API constructs a system which can perform the task at hand.

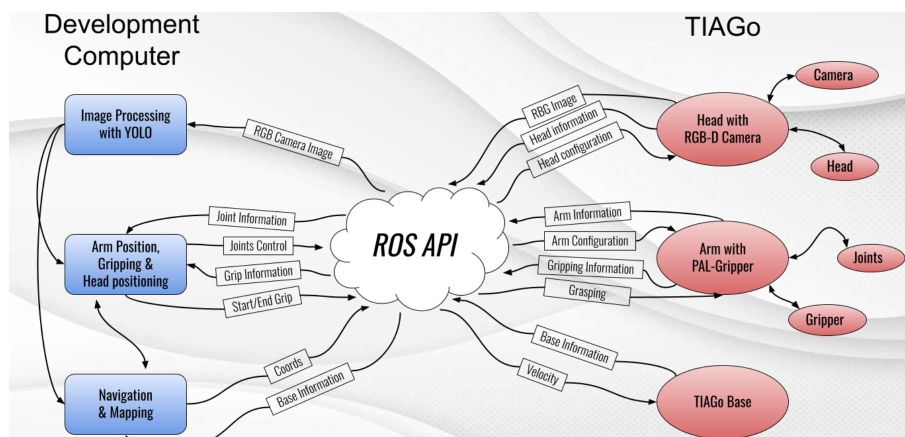


Figure 2.1: A Scheme Displaying the Communication between the Development Computer and the TIAGo using a ROS API

2.1.1 Head With a Camera for Object Search and Recognition

The main function of the head (see Figure 2.2) is moving the RGB-D camera that is mounted on the robot [1]. The head position is set by controlling two motors using joint value configurations to enable tilting and panning of the camera, see Figure 2.2 [1]. These motors are controlled through the ROS API and allow the head to tilt from 45° to -60°, and pan from 75° to -75°. The camera and head were used to send a continuous image stream to the development computer where the images were processed for object detection. Table 2.1 describes which topics and actions are accessed for controlling the head.

Topic(s)/Action(s) used	Description
/xtion/rgb/image_raw	The raw image stream from the RGB-D camera is published here
/head_controller/follow_joint_trajectory	An action where you can send joint names and their required positions for head movement

Table 2.1: Topics and Actions Used for Communication with the Robot's Head

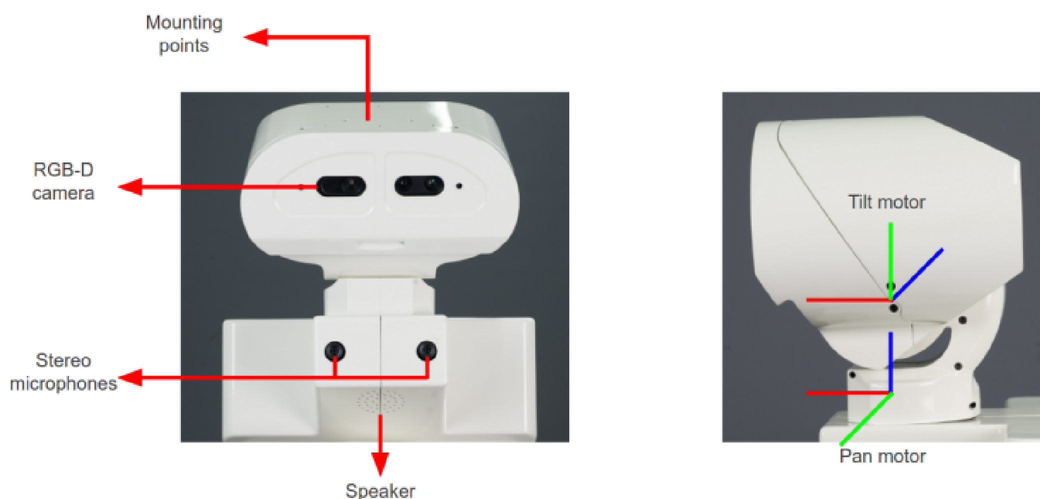


Figure 2.2: TIAGo's Head: Includes an RGB-D Camera for Image and Depth Processing and Two Joints for Head Positioning. [[1], p. 23, fig 18. Copyright PAL Robotics]

2.1.2 Arm With a Gripper for Object Grasping and Manipulation

Figure 2.3 shows the TIAGo's arm which consists of four joints used for moving the gripper [1]. It also consists of an M3D wrist that has three degrees of freedom (3 DoF) similar to the human wrist. These joints are controlled using the ROS API,

although, to prevent overheating and over-current in case of a collision with another object, there are self-protective mechanisms in the motors of the arm.

In this project, the arm is used for positioning the gripper and manipulating the objects. Table 2.2 describes which topics and actions are accessed for controlling the arm.

Topic(s)/Action(s) used	Description
/arm_controller/follow_joint_trajectory	An action where you can send joint names and their required positions for arm movement

Table 2.2: Topics and Actions Used for Communication with the Robot's Arm

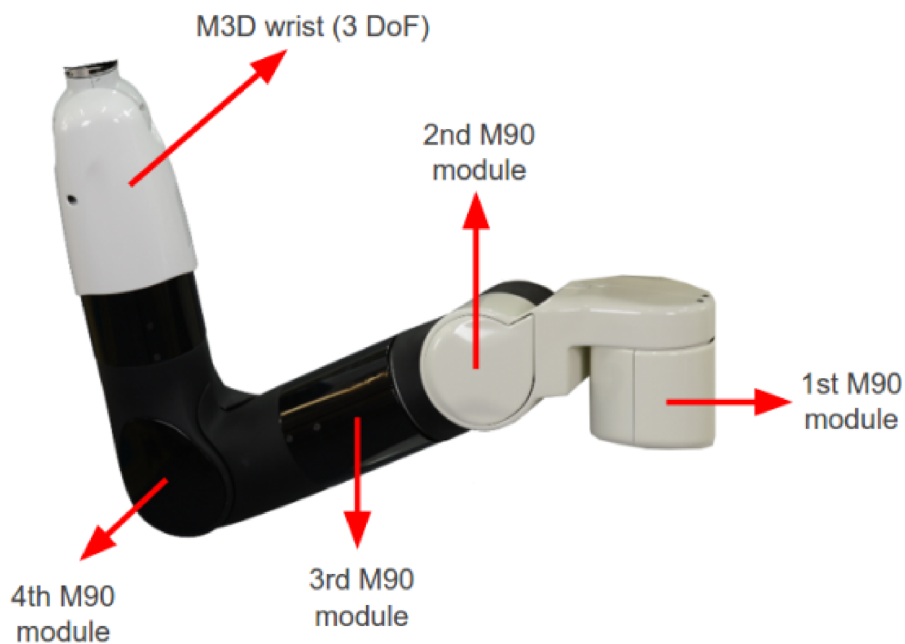


Figure 2.3: TIAGo's Arm: Consists of Several Joints for Configuring Position. [[1], p. 18, fig 13. Copyright PAL Robotics]

Mounted on the wrist of the robot is a parallel gripper (PAL gripper), see Figure 2.4, which lets the robot interact with the environment [1]. This gripper can grab items no larger than 8 centimeters and uses two motors that can be controlled independently. When the robot's arm and gripper are in position, the gripper motors can be activated through ROS to close around the object [1]. In this project, the gripper is used to grasp and hold onto an object while traversing the room before dropping it into the storage place. Table 2.2 describes which topics and actions are accessed for controlling the wrist and gripper.

Topic(s)/Action(s) used	Description
/hand_controller/follow_joint_trajectory	An action where you can send joint names and their required positions for wrist movement
/parallell_gripper_controller/follow_joint_trajectory	An action where you can send joint names and their required positions for gripper movement

Table 2.3: Topics and Actions Used for Communication with the Robot's Gripper

Figure 2.4: TIAGo's PAL Gripper: Used for Grasping Objects. [[1], p. 21, fig 16. Copyright PAL Robotics]

2.1.3 Torso for Vertical Movement Control

TIAGo's torso (see Figure 2.5) is equipped with a built-in mechanical function that allows the robot to expand from a length of 110cm to 145cm [1]. The head and arms are attached to the torso, making it a central component. In this project, the torso is used to move the robot vertically when picking and placing objects.

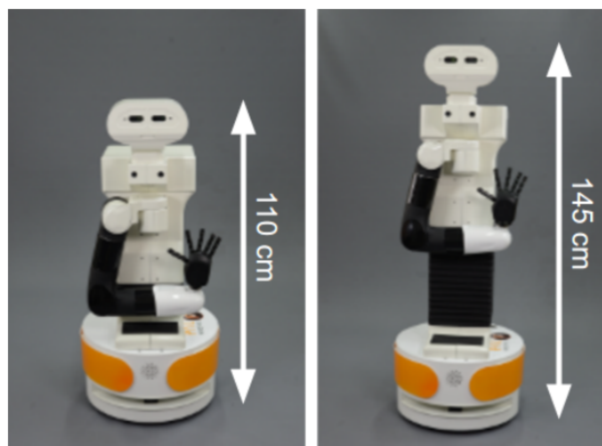


Figure 2.5: TIAGo's Torso: Shown with Minimum and Maximum Extension. [[1], p. 15, fig 9. Copyright PAL Robotics]

2.1.4 Mobile Base for Movement and Navigation

Figure 2.6 shows TIAGo's mobile base which contains a differential drive, an onboard computer, a laser rangefinder, and a sonar sensor [1]. The TIAGo's autonomous navigation framework can be used to map and navigate the room by utilizing the ROS API [9]. This gives TIAGo the ability to plan a path and maneuver to any location on the map whilst avoiding obstacles using laser and sonar sensors. The mapping system uses the 2D laser scanner to create an Occupancy Grid Map (OGM) which then is used by the navigation framework to localize and navigate autonomously [10]. There is also an option for advanced navigation which enables the camera for obstacle avoidance [11]. In this project, the mobile base is used for navigating through the room whilst avoiding obstacles such as furniture and walls. Table 2.4 describes which topics and actions are accessed for controlling the mobile base.

Topic(s)/Action(s) used	**Description**
move_base_controller/cmd_vel	A topic where you can publish an angular or linear speed and direction for moving the base
move_base/goal	Topic used by the move_base action
navigation_stop	Topic created for publishing stop messages

Table 2.4: Topics and Actions Used for Navigational Communication with the Robot's Base

Figure 2.6: TIAGo's Base for Mobility: Equipped with Sensors for Mapping and Navigation. [[1], p. 10, fig 3. Copyright PAL Robotics]

2.2 ROS: Robot Operating System

Robots are complex architectures of hardware. They consist of several sensors, motors, batteries, and other electrical components that are working together to make up a functioning system. These components are complicated and need to work seamlessly with each other. Advanced hardware knowledge and low-level programming are required to tweak and program these components individually for them to work together. This makes the process of controlling a robot time-consuming and convoluted.

To solve this problem ROS can be used. ROS is an open-source software that provides a collection of tools, libraries, and conventions for building and running robot applications [12]. Being an open-source software, the ROS community consists of a large group of developers, both from the private sector and from large corporations. The ROS community offers an abundance of free to use packages and software with state-of-the-art algorithms and tools used in the robotics industry [12] [13]. By working with these tools the robot can efficiently be trained and tested in a safe environment with an easy way to troubleshoot and assure the quality of the system. ROS is built on a modular architecture where most components with a software interface can be integrated and used within a system [14].

ROS comes in several distributions with the latest being Noetic Ninjemys for the Ubuntu 20.04 (Focal Fossa) release Linux distribution, which is the one that is being used in this project [13].

This chapter provides an overview of the main concepts and features of ROS, and how it was used in the project with the TIAGo robot. Specifically, this section covers the main ROS concepts, communication methods and tools that were used in the project.

2.2.1 Main ROS Concepts

ROS packages are fundamental organizational units of ROS code [15]. They contain many resources necessary to perform specific tasks, among these resources are ROS nodes, configuration files, launch files, messages, and service definitions (further explanation in 2.2.2). Another feature of packages is that they are easy to export and share allowing them to be reused across different projects.

The ROS Master is the central component of ROS. The Master makes it possible for nodes to find each other, exchange messages, or call upon services [15]. It manages the communication between different nodes in the ROS system.

The Parameter Server is a ROS component that allows nodes to share and store parameters [16]. It is best used to store static, non-binary data since it is not designed for high performance.

Bag files in ROS are used to store message data [17]. The bag files are created by a tool called rosbag which subscribes to one or more ROS topics. They are used to

record message data which can later be played back. They can be used to capture sensor data or replay recorded data for testing or analysis.

A ROS client library is a collection of code [18]. In this project, both the roscpp and rospy client libraries have been used. Roscpp is written in the programming language C++, while rospy is written in Python. These libraries provide a consistent and easy-to-use API for managing communication, configuration, and other aspects of ROS systems [18].

2.2.2 Communication in ROS

YAML is a markup language and ROS utilizes its syntax for ROS communication files, see A.1 [19]. An example of YAML syntax usage in ROS is the message files. Messages make up the core of communication inside the ROS paradigm [20]. Message files are signified by the ".msg" file ending, and the structure follows a YAML syntax as visualized in Figure 2.7 [20]. As shown, the message is constructed by none (empty messages) or several variable declarations to define the different parts of the message that can later be accessed in code when developing nodes. Messages can be built on other messages and can be seen as recursive data structures [20]. They are sent between nodes to exchange data.

```
message file

# coordinates
float64 x
float64 y
float64 z
```

Figure 2.7: Message File with YAML Syntax

A node is a file within a package which is executable [21]. Nodes communicate via messages over topics, the messages are sent to a topic which then can be accessed by one or several other nodes [21]. By sending a message to one specific topic rather than to each individual node, communication becomes faster and more efficient. The nodes can include a publisher, subscriber, service, action, or a client to the latter two mentioned [15]. A publisher is a node that sends information wrapped in a message via a topic, and a subscriber is a node that receives that information by accessing the message stream of that topic [15].

Services are similar to messages in their structure and like all communication files in ROS, use the YAML format [22]. The core difference between a service and a message is that services consist of two separate messages divided into a request and a response [22]. A service file, as visualized in Figure 2.8, divides the request and response message by the triple hyphen "- - -" [22].

```
# service that performs addition

# request
int64 a
int64 b
---
# response
int64 sum
```

Figure 2.8: Service File with YAML Syntax

Actions are similar to services but provide a more complex structure and instead rely on a goal-oriented request based on a starting point [23]. Actions also provide dynamic feedback during the execution of the request making them suitable for more advanced control operations [23]. An action file has the format ".action" and has the YAML structure as shown in Figure 2.9 [23]. This file is divided into goal, result, and feedback messages separated by the triple hyphen "- - -". The goal message is an instruction for the action to determine what state it should reach for the action to be considered complete [23]. The result message is usually a reached state that is sent back when the action is finished. It does not need to be the final or an expected result, and is useful for supplying information about how well the action was performed and where it failed if something went wrong [23]. For example, if the action reached the expected state or failed along the way in another state. As mentioned, the feedback message is updated and published during the duration of the action. This is useful as it allows for surveillance and evaluation of the ongoing process. Unlike services and messages that are included in the default ROS architecture, actions are part of a package called actionlib (described further under subsection 2.2.3) which implements the action client and action server and applies actions to nodes in the code [23].

```
action that chopps trees

# goal definition
int64 tree_sort
---
# result definition
int64 total_trees_chopped
---
# feedback definition
float64 percent_chopped
```

Figure 2.9: Action File with YAML Syntax

Clients are responsible for sending requests to a server and receiving its response. There are different implementations of clients based on their type, but in general, they follow a process of connecting, sending, and receiving [22] [23]. First, the client checks if the specified server it is supposed to connect to is available and running [23]. If so, it will connect and prepare to send a request. After the request is created

and sent it will await the server's response. Once the server is finished, the client receives the response for further processing and evaluation. Two of the main types of clients used in nodes are service and action clients.

Service clients are users of the service servers, and following the service message protocol the client will send a specified request message for the service server to process [22]. Once the server is done it will return a state of success or failure depending on the server's ability to run and succeed in execution. The client will also receive the specified response message written by the server. It can thereafter evaluate and process this response.

Action clients are users of the action servers, and follow the action message protocol. The client will send a specified goal message for the action server to process [23]. After the server finishes its task, it will return a designated result message to the client who is able to evaluate and process the message. An action client, unlike a service client, has the ability to cancel the goal at any time and receives the current state of the action server when cancellation occurs [23].

Servers are responsible for serving clients and their requests [23]. There are different implementations of servers based on their type, but generally, they follow a process of advertising, callback, and return. A server is initiated by starting and advertising its name on the data flow graph, a client can then connect to the server and send a request [23]. The request is sent to the callback function inside the server and the server begins executing the specified task it is given. After completing the task, the server then returns a response to the client and the communication is completed. Two of the main types of servers used in nodes are service servers and action servers [22] [23].

Service servers distribute a service for the service client users, and following the service message protocol the server will receive a specified request message to process [22]. The request message will be sent to the server's callback function, and the server will then execute this function and write a specified response message to send back to the service client. If the callback function is executed successfully the server will send back a success state to the client along with the written response. If the server does not manage to fully complete the callback function, the client will instead receive a state of failure and no response message will be returned.

Action servers distribute an action for the action client users, following the action message protocol the server will receive a specified goal message to process [23]. The goal message will be sent to the action servers callback function, the server will then begin to execute this function. During the execution, the action server can update and publish the specified feedback message provided in real-time to its feedback topic. The action server will also check for a cancellation request from the action client and shut down accordingly [23]. In that case, the server will send back its current state to the action client. An action server comes with a set of predetermined states that the action client can access [23]. For instance, if the server's execution of the callback function was successful, the specified state of the server will be set to succeeded. If the server failed, the state will be set to aborted.

2.2.3 Tools for Perception, Planning, and Control

In addition to the main ROS concepts and communication methods covered in the previous sections, various tools and packages are available in ROS to simplify and accelerate the development of robotic systems. This section provides an overview of several important ROS tools used in the project.

MoveIt inverse kinematics is an interface controller within the MoveIt framework [24]. When given a desired end-effector position using cartesian coordinates, the module enables the robot to calculate the joint angles and trajectories needed to reach this position [24]. It does this with regard to the given kinematic constraints and with collision avoidance. "Inverse kinematics is the use of kinematic equations to determine the motion of a robot to reach a desired position." [25].

Another interface within the MoveIt framework is the forward kinematics interface. Forward kinematics is the process of determining the position and orientation of the end-effector given the joint angles and the kinematic structure of the robot [26]. It makes it possible for the robot to calculate the pose of the end-effector given the joint angles. In this project, the end-effector consists of a PAL gripper.

Point cloud data is used by the TIAGo for object recognition and localization [27]. It uses the data for the identification of objects in the environment, determining their position and orientation, and planning its motion to manipulate them [28]. The point cloud data is collected by the TIAGo's RGB-D camera [27]. The data is then interpreted and processed using the point cloud library which has multiple algorithms and functions to extract relevant information from the point cloud [29].

An octomap is used to generate a 3D model of an environment, such as a room [30]. The octomap uses point cloud data to create this 3D model. It is constructed in such a way that it fulfills four criteria: it is a full 3D model, it is adaptable, flexible, and compact [31]. You can add new information to the model at any moment and it is not necessary to know the size of the map beforehand - making it flexible. Octomap is used to make it easier to understand the environment and detect obstacles or accepted objects.

Roslaunch is a tool in ROS that makes it possible to start multiple ROS nodes at the same time [32].

Playmotion is an open-source tool that allows one to create and carry out predetermined motions [33].

Tf2 keeps track of the 3D coordinate frames of a robotic system and how these frames change over time [34]. It makes it possible to trace how the frames change over time and how they relate to one another. "Where was the head frame relative to the world frame, 5 seconds ago?" [34]. Tf2 is an improved version of its predecessor tf that offers similar functionalities with enhanced efficiency [35].

Actionlib is a library used in ROS for performing complex actions in a distributed system [23]. It allows clients to send action requests to servers and receive feedback

and results. With actionlib, it is easy to create and manage actions with features like goal cancellation, result feedback, and preemption [23].

Gazebo is an open-source 3D robot simulation software used for, among other things, robotics research, education, and development [36]. It allows users to simulate and test robots in a virtual environment before deploying them in the real world.

RviZ is a robot visualization tool which makes it possible to visualize sensor data, robot models, and environment maps [37]. "It enables you to see the robot's perception of its world (real or simulated)." [38].
Table 2.5 shows a summary of the ROS tools used in this project.

Tool	**Description**
MoveIt	A software for manipulation and movement of the robot
Point Cloud	A set of data points that can represent an object or environment
Octomap	Generates a 3D model of an environment
roslaunch	Enables several ROS nodes to be launched at the same time
Playmotion	Open source. Create and carry out predetermined motions
Tf2	Makes it possible to trace how frames change over time
actionlib	A library used to create and manage actions
Gazebo	A 3D robot simulation program. Enables testing before real world implementation
RviZ	A 3D visualization tool used to display and interact with sensor data from the robot

Table 2.5: Tools for Perception, Planning, and Control

Docker is a tool for controlling the development environment and avoiding clashes with other preinstalled dependencies on different computers. It is a software platform that allows for compartmentalizing different projects into containers [39]. These containers include all code, run-time, system tools, libraries, and settings used to run an application. It allows the user to create an environment that will work the same on different platforms and operating systems. For this project, docker was used to create an environment that kept all tools that were needed, making sure these did not clash with other parts of the computer environment.

2.3 YOLOv8: You Only Look Once, version 8

YOLO is an object detection and image segmentation AI model [40]. The model is useful for object detection in live video because of its speed and accuracy.

YOLO is currently developed by Ultralytics who has continued improving the YOLO framework from YOLOv5 through YOLOv8 [40]. YOLOv8 is the latest version of the model. It is faster and has better accuracy than previous versions. It supports several different AI tasks such as; object detection, segmentation, pose estimation, tracking, and classification.

In this project, the YOLOv8 model is used to identify objects while the robot navigates through the room. Thereby, giving detected objects a confidence score and object type tag. YOLO also provides bounding boxes around the detected objects which allows navigation towards the objects and centering them before grasping.

YOLOv8 includes five pre-trained detection models that can be used, these models come in various sizes. As the size of the YOLO model gets larger it leads to a more accurate model in most cases whilst at the expense of more computer resources. Furthermore, YOLOv8 enables the creation of custom models and has easy commands for training and validating a model with an original dataset.

2.3.1 Basic Machine Learning Concepts

Since YOLO is an AI model [40], the basic concepts around machine learning are key to understanding how the training of the model works.

A feed-forward neural network can be regarded as a non-linear math function that transforms an input set into an output set [41]. This transformation is determined by the neural networks weights. These are decided by a process called "learning" or "training", and computationally takes a lot of resources. After the weight parameters have been set, the network can process new data rapidly [41].

To train a neural network, the model needs to know when its predictions are improving, this is accomplished by measuring the error rate or loss value [42]. The error rate is a measure of how well the model's predictions match the true values or labels in the data, this value reflects the overall performance of the model. Another measure of the performance of the model is its accuracy, which is the proportion of examples where the model produces the correct output [42]. When training a model a training dataset is used, this is often comprised of several pairs containing some sort of data and a correct label for that data [42]. After computing this dataset, the training error is obtained, and continuous training of the model is conducted to reduce it. Following the training phase, the model is evaluated with a test dataset which is used for controlling how well the model works with new unseen data [42]. The trial of the model on the test dataset yields a test error, and it is desirable for this error to be minimized. This would indicate that the model performs well

when confronted with new data. The desired factors for a machine learning model to perform well is:

1. Low training error i.e loss
2. Small gap between the training and test error

These factors connect to the central challenges of machine learning, over-fitting, and under-fitting. Under-fitting occurs when the training error isn't small enough [42], i.e. the model does not perform well on the training data. Over-fitting occurs when the gap between the training and test error is too big [42]. This means that the model works well on the training set but not when presented with new unseen test data [42]. To help avoid over-fitting, a validation test set is also used during training. This dataset is used to approximate the test error during training and allow the weights to be changed accordingly [42].

Another important concept when it comes to training is epochs. One epoch represents a complete training cycle of the whole training dataset [43]. Usually, training is done in many epochs to get a good-performing model.

2.3.2 Using CUDA to Accelerate Live Image Processing and Model Training

NVIDIA developed CUDA as a programming model and parallel computing platform for general computing on Graphical Processing Units (GPUs). It allows developers to use the great computational power of these units to significantly enhance the performance of their applications. To speed up the computing, CUDA allows the compute-intensive part of the application to run in parallel on thousands of GPU cores while the sequential part runs on the CPU [44]. This enables processes such as YOLOv8 to accelerate the live image processing and training of the models.

2.4 Computer requirements for demanding simulations and image recognition

Demanding simulations and image recognition training require a dedicated NVIDIA graphics card. A computer powerful enough to utilize this card is needed. The parts ordered in this project, are presented in Appendix A.2.

3

Methods and Implementation

This project can be divided into four main parts: Navigation, Object Detection, Grasping, and Integration. The navigation, object detection, and grasping parts were coded individually in order to be integrated and tested in the simulated environment. The successful simulation constituted the first demo and after approval from the supervisors, tests were carried out on the real robot. This chapter will explain the general idea behind the code for each of the parts mentioned above.

3.1 Enabling the TIAGo to Navigate Through a Room

The navigation solution utilizes the ROS action interfaces. Its main components are a navigation client and server, which in turn utilizes the predefined “/move_base” action included in the ROS navigation API. The navigation client was created for the sole purpose of initiating the navigation process by sending a custom goal to the navigation server. Figure 3.1 illustrates the overall function of the navigation solution in the form of pseudo code. The goal definition is part of the custom action file “GoToPoint” which receives a "0", "1", or "2" as input to set the precondition for the navigation. Sending a "0" will make the robot navigate a predefined search path consisting of a set of waypoints. A "1" will send the robot to the storage units and a "2" will have the robot return to its docking station. The server will wait for the client to send one of these navigational conditions with the goal message. An important notice is that the navigation client, server, and “GoToPoint” action were written as a part of the solution and were not already included in the ROS navigation API. The “/move_base” action interface is a standard TIAGo function and can be compared to a “black box” whilst the self-written client, server, and action contain the logic for the project’s specific solution.

The navigation server works as follows: The custom action server receives a “GoToPoint” goal which calls the callback function of the server. The callback function starts with creating a client to the server “/move_base” and then initiates a set of waypoints. The waypoints are based on predefined coordinates that correspond to a location and make up a path in the testing environment. Depending on the goal received, the robot starts to navigate. If a “0” was sent, the coordinates of the waypoints are assigned to the “/move_base” action’s goal parameters in order. This is done with a separate function “writeTargetPositions”. When the parameters from a waypoint have been assigned to the “/move_base” goal, the goal is forwarded to

the "/move_base" server. While navigating to the point it constantly checks if a cancellation has been requested, and in that case, it cancels the goal and exits the callback. Upon arrival at the destination, the same process is repeated for the next waypoint in the path. If the current waypoint is the last then it exits the callback function and stops, waiting for a new "GoToPoint" goal. If instead a "1" or "2" is received, the position of the storage units or the docking station respectively is sent to the "/move_base" server. Upon arrival, it exits the callback function and waits for a new "GoToPoint" goal.

To be able to cancel the goal when desired, a publisher and subscriber were created. The publisher sends a stop message to the subscriber. In the navigation server, there is a "Stop" class which has a method that changes the value of a "raise_stop" variable. The variable is changed from zero to one if the published message on the topic is "Stop". This variable is then included as a condition in the navigation loop, which enables the cancellation of the navigation goal.

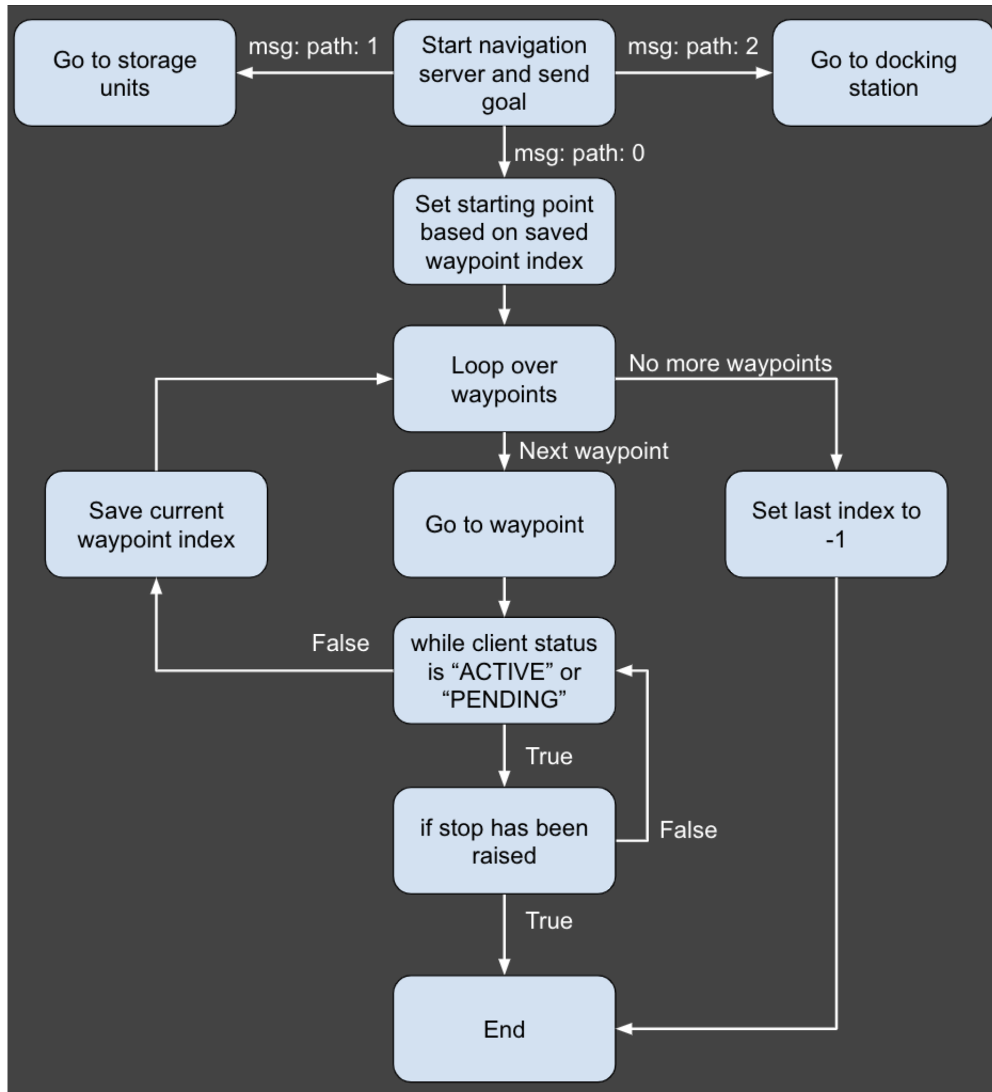


Figure 3.1: Flowchart for the Navigation Solution

3.2 Object Detection and Identification

This section presents the implementation of a custom YOLO model for detecting everyday objects. It covers the training process, utilization of YOLO for object identification, and the techniques used to center objects in the camera frame. The chapter also explains the extraction of precisely estimated object coordinates using the depth camera and the point cloud library.

3.2.1 Training the YOLO Model on a Custom Dataset

To be able to detect relevant everyday objects a custom YOLO model trained on a custom dataset was necessary. The pre-trained YOLO models did not have enough relevant objects for the project's purpose of detecting everyday objects [45].

Therefore, using the YOLO framework, a custom model was trained using classes from the Google Open Image V7 dataset [46] and by utilizing CUDA performance. Relevant classes were downloaded from the dataset using the Open Images Tool Kit V4 [47]. Table 3.1 shows the classes that were downloaded and Figure 3.2 shows the number of annotations for each class.

Ball	Bottle	Box
Coin	Eraser	Glasses
Mug	Pen	Pencil Case
Tennis Ball	Tin-Can	

Table 3.1: Classes Downloaded from the Google Open Image V7 Dataset

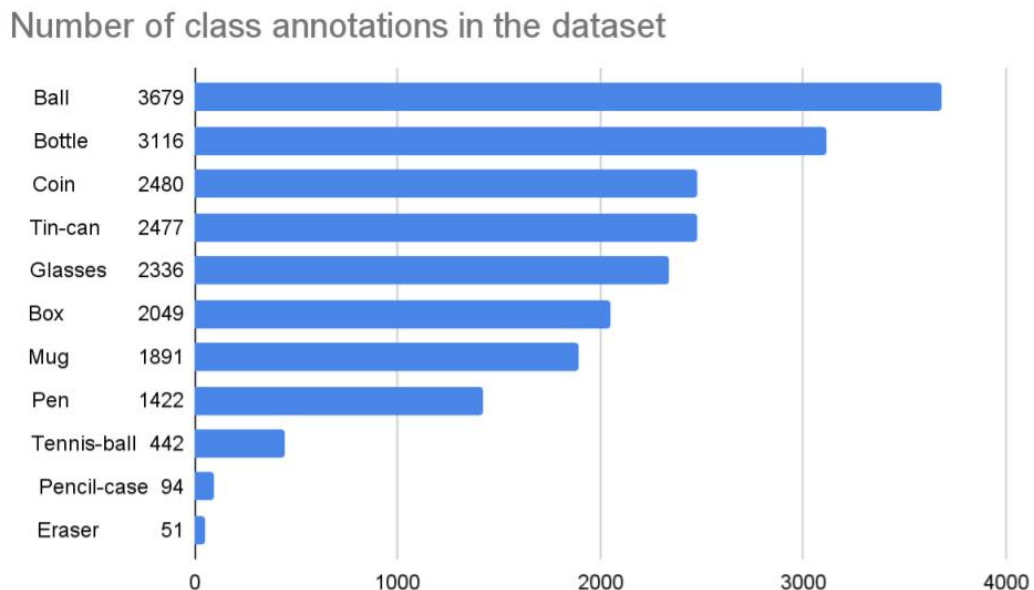


Figure 3.2: Number of Annotations for Each Class in the Dataset

Thereafter the dataset was prepared for training by using the website Roboflow to convert the Open Images CSV format to the YOLO format to be able to train with the YOLO framework [48]. The dataset was also pre-processed by auto-rotating and resizing the images with Roboflow to decrease training time and increase performance [49].

The training was done with the YOLOv8 framework on YOLO's pre-trained models as recommended by the authors [50]. Multiple models were trained on different pre-trained model sizes such as the nano, small and medium models.

3.2.2 Object Identification with YOLO

While navigating around the room, the robot needs to look out for objects to pick up. To do this, YOLOv8 is used. The camera feed from the robot's RGB-D cameras is constantly updated on a topic called "/xtion/rgb/image_raw", to which a node ("camera_yolo") subscribes and accesses the image stream. The image stream is then passed through the YOLO model for object detection. When an object is detected a bounding box is applied around it (see Figure 3.3) which allows for locating the objects in the camera frame.

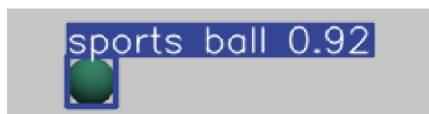


Figure 3.3: Object with Bounding Box

When YOLO detects an object, the "camera_yolo" node applies a filter which only lets accepted objects through. Accepted objects are the ones that the robot is actively looking for, for example: toys, trash, and bottles. Objects such as tables, chairs, and people are thus ignored. If an accepted object is detected and the model's confidence score is above a certain threshold, the node will publish the coordinates and size of the items bounding box to the topic "/coord". The published coordinates are later accessed when centering the object in front of the robot. See Figure 3.4 for the functional scheme.

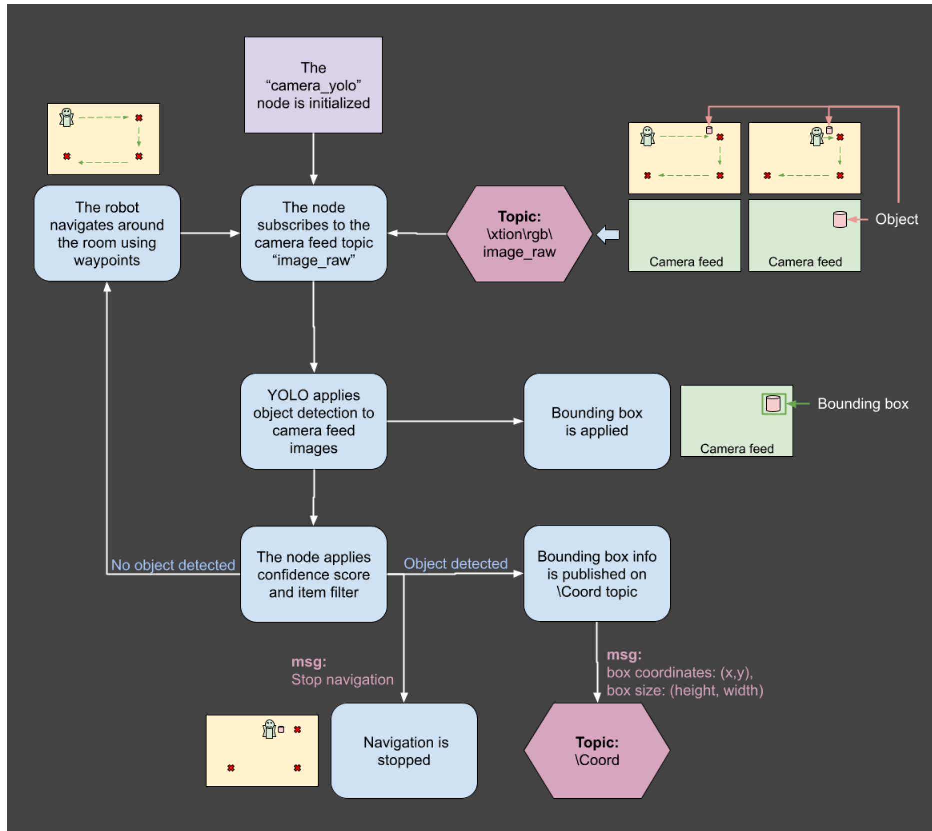


Figure 3.4: Graph of the "camera_yolo" Node Function

3.2.3 Centering Objects in the Camera Frame

To make it easier to find the object's 3D coordinates, and to grab the object, it is important to center the object in the camera frame. This makes sure that the object is straight in front of the robot and close enough for the grasping sequence to be performed. To do this a new node, "center_camera", accesses the bounding box info from the "/coord" topic mentioned above. When an object is detected and coordinates are published, the node turns and moves the robot toward the object depending on its position in the camera frame, see Figure 3.5.

To decide which way the robot should move, the node receives half the width and height of the camera frame. In other words, the center coordinates of the frame. The bounding box center is then retrieved with the same method and later compared to the camera frame center coordinates. Depending on the relative position of the object center to the camera center coordinates, the node decides if it should twist left or right and if it should move forward or backward. To move the robot the node sends a "geometry_msgs/Twist" message to the "/mobile_base_controller/cmd_vel" topic with a linear and angular speed. The linear speed will move the robot forward or backward and the angular speed will twist the robot right or left. The node will finally check if the object is centered, if it is not; the node will loop the previous steps again.

To speed up the process, the node decides the speed of the robot depending on how

far away the object is from the middle. If the object is more than 20 pixels away from the center, the speed is set to 0.15 m/s. Inside the limit, the speed is set to 0.01 m/s for better accuracy. There is also an area of acceptance when the object is centered. If the object is less than 15 pixels from the center, the node accepts this as sufficiently centered and sets the speed of the current velocity vector to "0". When both vectors (angular and linear) are set to "0", the object is centered, the loop is exited and the node returns an empty response.

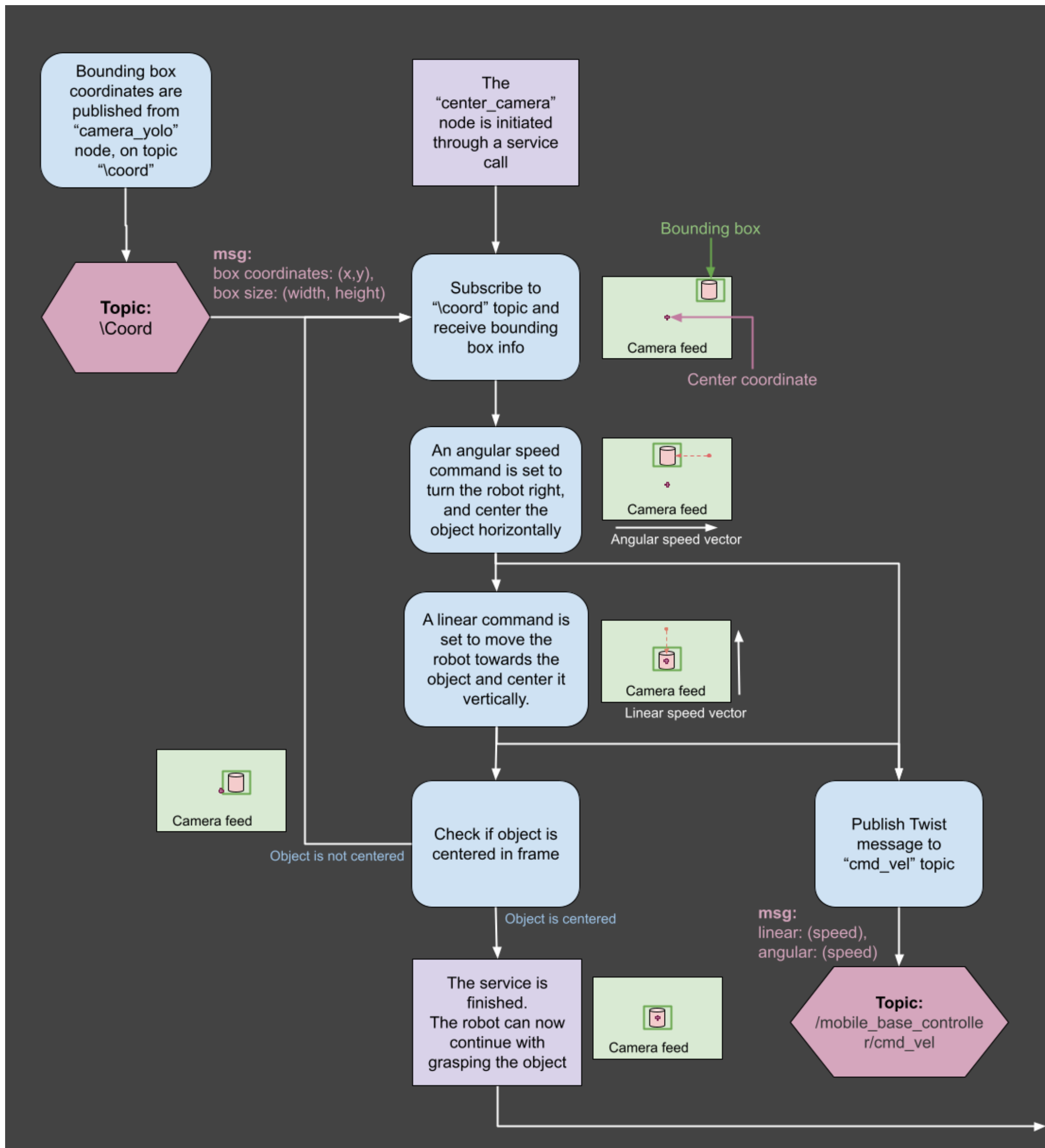


Figure 3.5: Graph of the "center camera" Node Function

3.2.4 Extracting Object Coordinates

To effectively manipulate an object, its precise coordinates must be determined. This information can be extracted using the depth camera of the TIAGo Robot in conjunction with the segmentation of objects through the use of the point cloud library. The depth camera generates a point cloud, a group of points, each containing their x, y, and z coordinates [51]. By utilizing the point cloud library's functions, the objects in the scene can be segmented based on this cloud. Initially, the point cloud library is used to remove the floor from the scene. This step effectively separates the floor from the objects present in the scene. Examples from the simulations are shown in Figure 3.6.

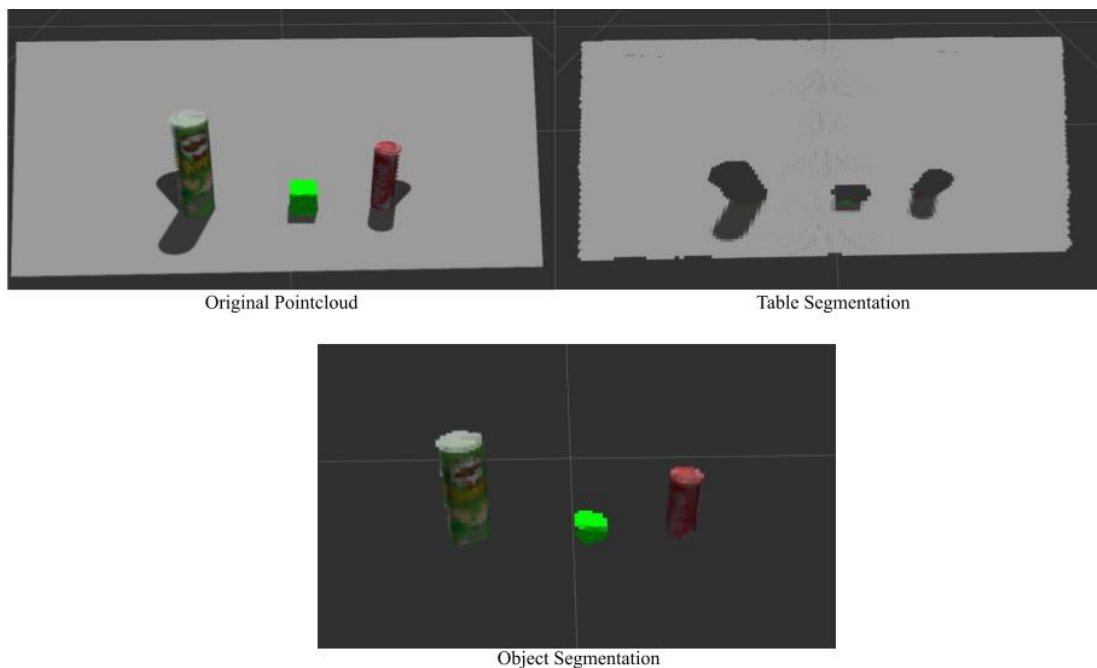


Figure 3.6: Table Segmentation of Objects: Removing the Surroundings and Isolating Objects in the Scene

Following this, the region-growing segmentation algorithm in the point cloud library is applied to the object's point cloud to isolate individual clusters for each object [52].
The region-growing segmentation algorithm in the point cloud library begins by sorting the point clouds based on their curvature value [53]. This is because the points with lower curvature values are in the flattest area, which reduces the number of segmentations [53]. After the cloud is sorted the point with the lowest curvature value is picked and used to start the growth of the region [53]. This works the following way [53]:

- The selected point is added to the set called seeds.
- For every seed point, its neighboring points are found.
    - The normals of the seed and neighboring point are compared and if the angle is less than the angle threshold then the point is added to the current region.

  - The neighboring points curvature values are extracted, if they are under the curvature threshold the point is added to the seeds.
  - Current seed gets removed from the set seed

When the seed set is empty the region has been fully grown and is repeated from the start [53]. In Figure 3.7 the region-based segmentation has been done on the plane segmentation objects from earlier and given a unique color for each cluster.

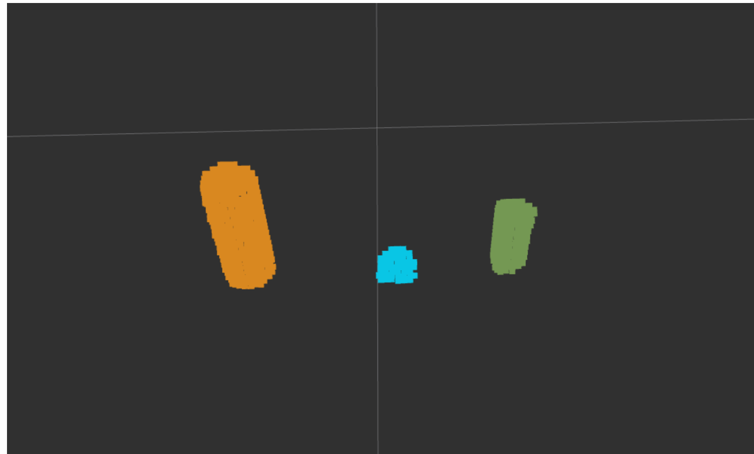


Figure 3.7: Segmentation of Objects

Subsequently, the maximum and minimum points of each cluster corresponding to an object are obtained for the x, y, and z-axis. The middle point coordinates for each object on one axis can then be calculated using the formula:

$$Coordinate_{Middle} = \frac{Coordinate_{max} + Coordinate_{min}}{2}$$

The resulting middle coordinates are visualized as green spheres, shown in Figure 3.8.

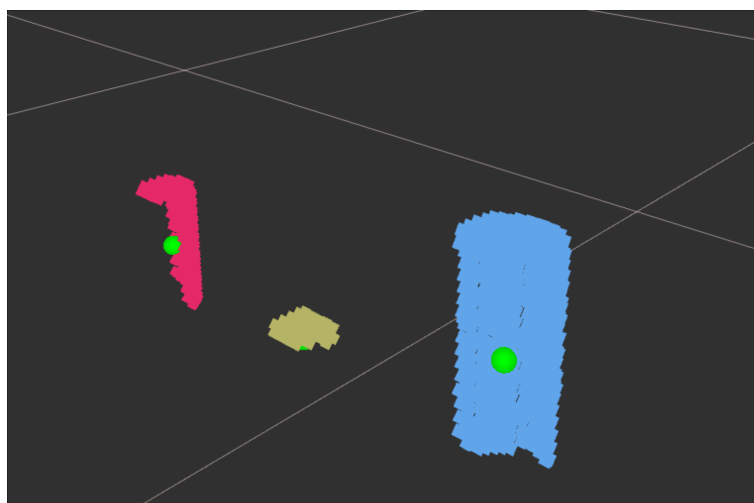


Figure 3.8: Objects Middle Coordinates Visualized as Green Spheres

Finally, the object straight in front of the robot is selected. This object should be the object that YOLO identified and centered. The object's coordinates are then published for the grasping nodes to use.

Furthermore, the coordinates are translated from the tf frame of the depth camera to the base of the robot, which means that the coordinates published are relative to the coordinate system at the base of the robot for ease of use by the grasping nodes.

3.3 Enabling the TIAGo to Grasp Objects

Grasping an object in a safe and successful manner requires coordination between several components of the robot. To move the gripper to a desired position, where it is able to successfully grasp an object, both the head, arm, and torso joints need to be configured and moved. The arm and torso movements utilize either inverse or forward kinematics, both having their perks depending on the situation. The head control is necessary for the perception of the close environment to filter off dangerous movement solutions where collisions occur.

The grasping consists of several nodes that are used for control of the specified parts mentioned above. The nodes are constructed in the form of either an action server or a service server. This structure allows for easy control of the head, arm, and torso movements using plain service or action calls with the specified request or goal parameters needed for the movement. For the grasping sequence to work the control nodes and their functionality needs to be called in specific orders with different parameters provided depending on which state the sequence is in. To achieve this workflow a main action server for grasping is created and provided with the clients to the corresponding control nodes. This main action server can thereafter make the correct calls in the right order to the specified service or action. The use of an action server also makes it easy to call the grasping sequence from another workflow and provide it with relevant information used in the sequence, such as object position and height of an object.

3.3.1 Grasping Control Nodes

The control nodes utilize an abundance of tools to provide a desired robot behavior. MoveIt, Tf2, and provided action interfaces from the TIAGo handbook such as the arm, head, and torso joint trajectory controllers, are employed to perform movement using inverse and forward kinematics. The point cloud generated allows for the construction of an octomap which is used by MoveIt to avoid collision with the nearby environment. Services that allow for the gripper to grip and hold as well as release the object are also used and are provided by the TIAGo handbook.

The first step of the grasping sequence consists of building the octomap (see Figure 3.9) to provide MoveIt with the necessary information needed for collision avoidance. The octomap is generated from the point cloud, mentioned in chapter 3.2.4, which in itself is gathered from the RGB-D camera.

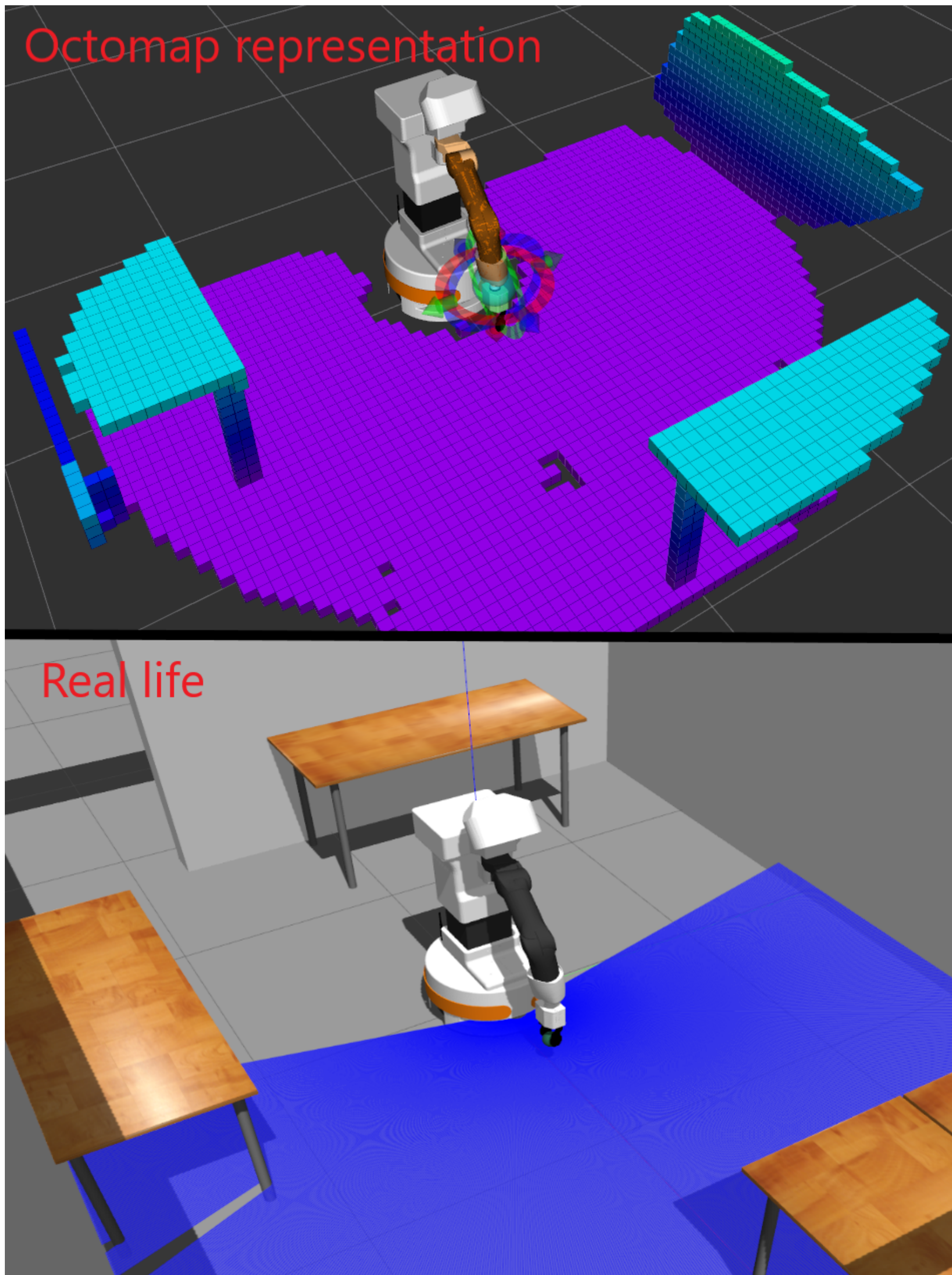


Figure 3.9: Octomap Representation vs. "Real Life" View in Simulation

The generation of the octomap is provided by the TIAGo system and the data used to generate it is accessed from the RGB-D cameras which record the view from the current orientation of the head. This means that to properly build an octomap of the surrounding environment, head movements that allow for a broad camera panning are required. Therefore, a corresponding head movement node was created, represented in Figure 3.10.

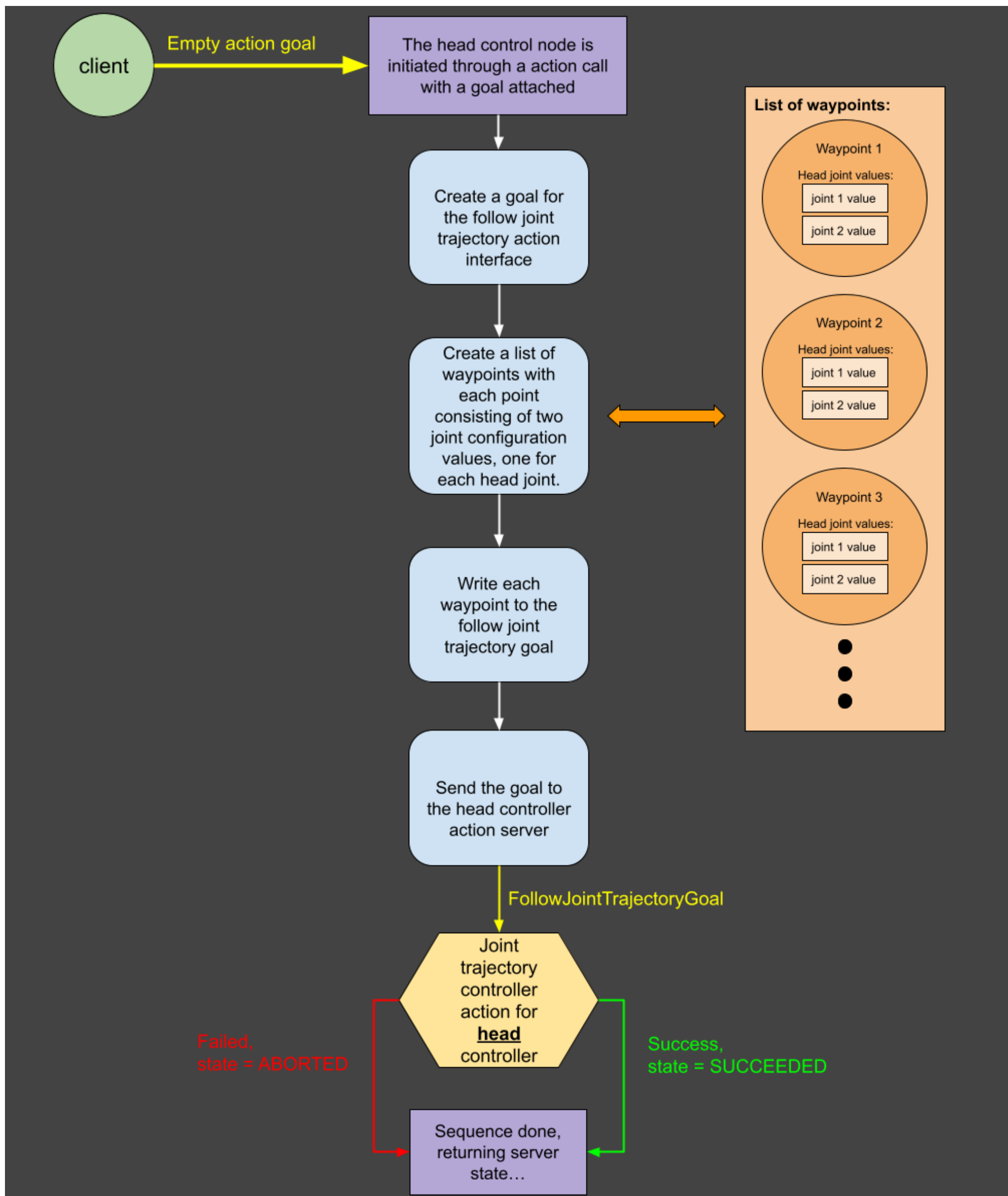


Figure 3.10: Function Scheme for Head Control Node

The node is implemented as an action server and makes use of the joint trajectory controller action interface provided by the TIAGo system to move joints. To get a proper scan of the room the head needs to acquire several different joint configurations. In order to easily define these configurations and send them to the trajectory controller, a simple vector consisting of joint configurations is used. When the node receives a start command the server will make a call to the joint trajectory controller responsible for moving the head joint and send the joint configuration vector. The controller will attempt to reach these points resulting in pointing the camera in different directions , see Figure 3.11. During this process, the octomap generation will automatically create a 3D model of the nearby environment which then can be used by MoveIt.

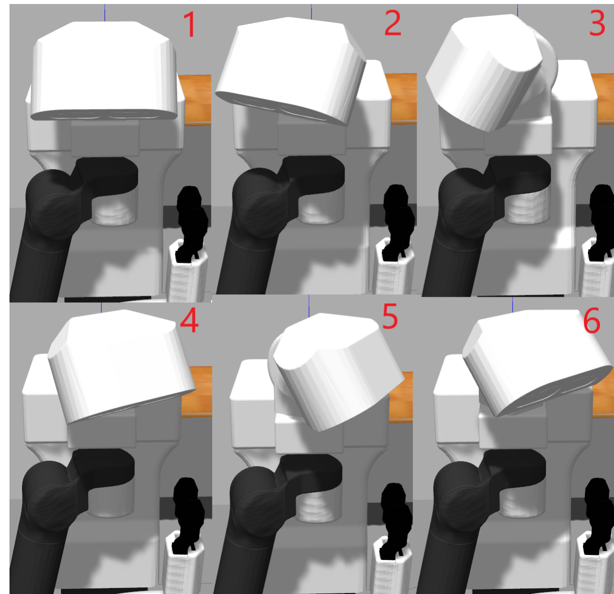


Figure 3.11: Head Positions during Pan Motion for Octomap Creation

MoveIt provides an easy and efficient way to integrate inverse kinematics and provides collision-aware movement for both inverse and forward kinematics, see Figure 3.12. MoveIt works by receiving a set of instructions for a specific movement. Target values are then written and sent to the planning interface. The interface can add additional information such as the octomap which allows for planning and executing the motion.

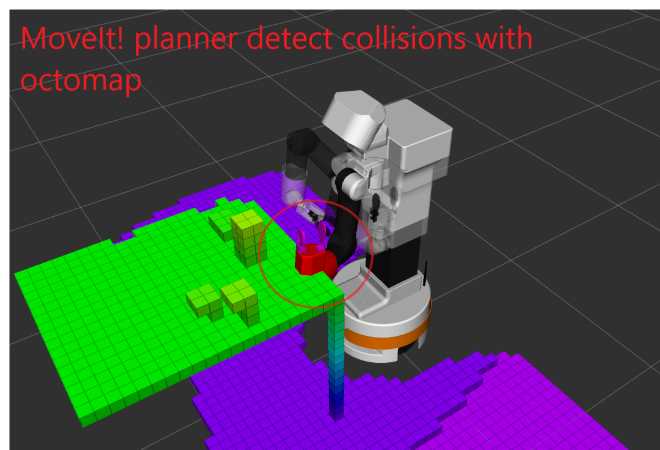


Figure 3.12: Collision Detection with Octomap

From the control node's perspective, a movement using MoveIt mainly consists of four steps: selection, configuration, planning, and movement. Selection is the step in which a specified joint group or individually chosen joints that are to be moved are specified and fed to the MoveIt interface. The choice depends on the case of using either forward or inverse kinematics. A joint group, depending on the desired behavior, can for example consist of all the arm joints or both the torso and the arm joints combined. After selecting a group or individual joints, configurations need to

be made to provide MoveIt with the necessary information of how it should move the group. This also depends on the kinematics that are currently utilized. In the case of inverse kinematics, the MoveIt interface first needs to be provided with an origin for its cartesian coordinate system. In this case, the base of the robot will be the referenced system for the robot since the point cloud coordinates also uses the base as its starting point. After selecting an origin the received cartesian coordinates and quaternion orientation can be written to the interface and the system is ready to move onto planning. An important clarification for the inverse kinematics is that the selected group decides which frame of the robot the positional coordinates and orientation are meant for, in this case, it is the end-effector i.e. the grippers target position being written to the interface. The specifics of the inverse kinematics control node can be seen in Figure 3.13.

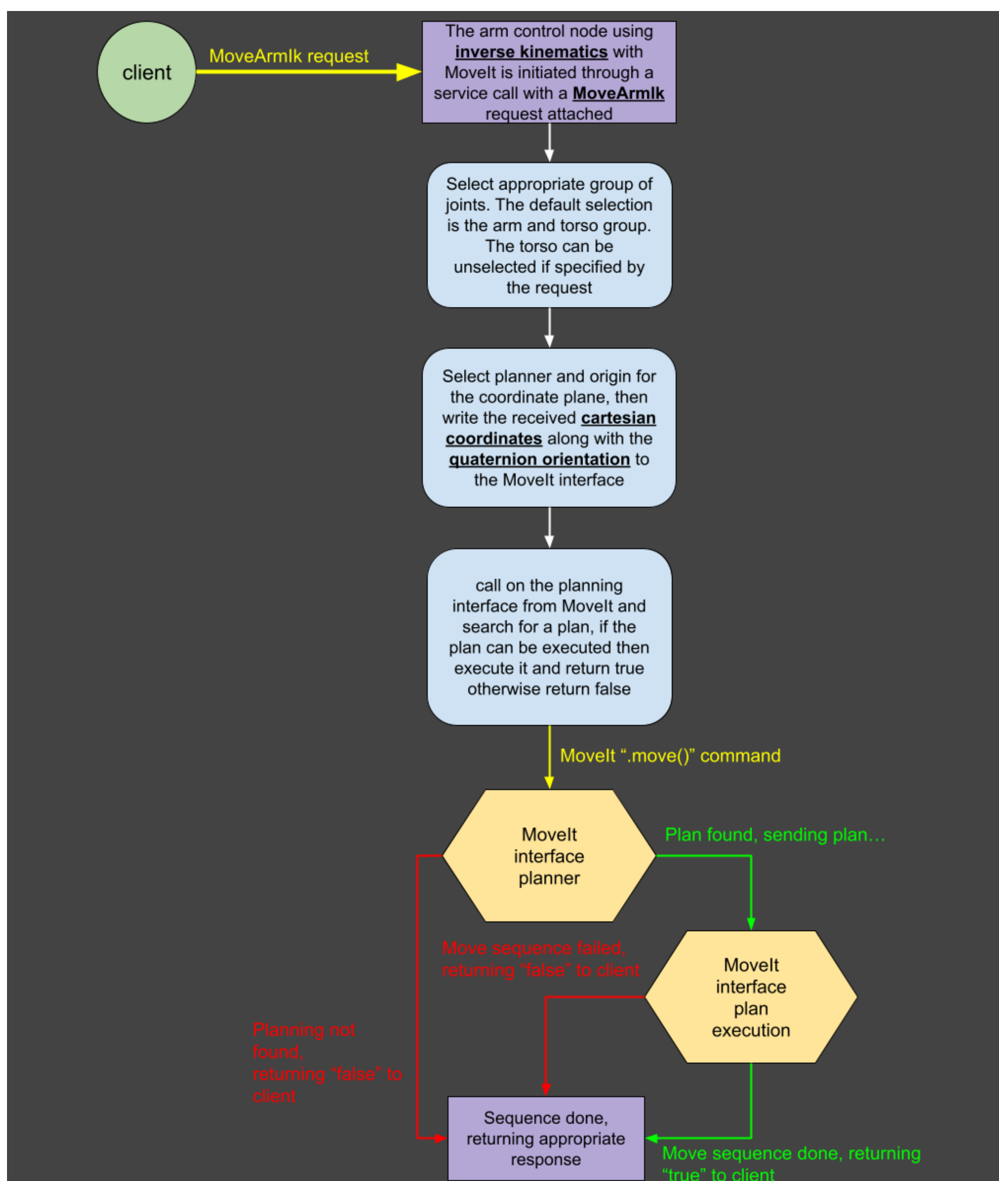


Figure 3.13: Function Scheme for Arm Control Node Utilizing Inverse Kinematics with MoveIt

For forward kinematics the configuration process does not require the selection of an origin since the interface is not moving the joints to a point in a referenced space but instead to targeted joints values. The only configuration needed for specifying a goal pose using forward kinematics in MoveIt is writing the targeted joint values received to their corresponding joint. The specifics of the forward kinematics control node can be seen in Figure 3.14.

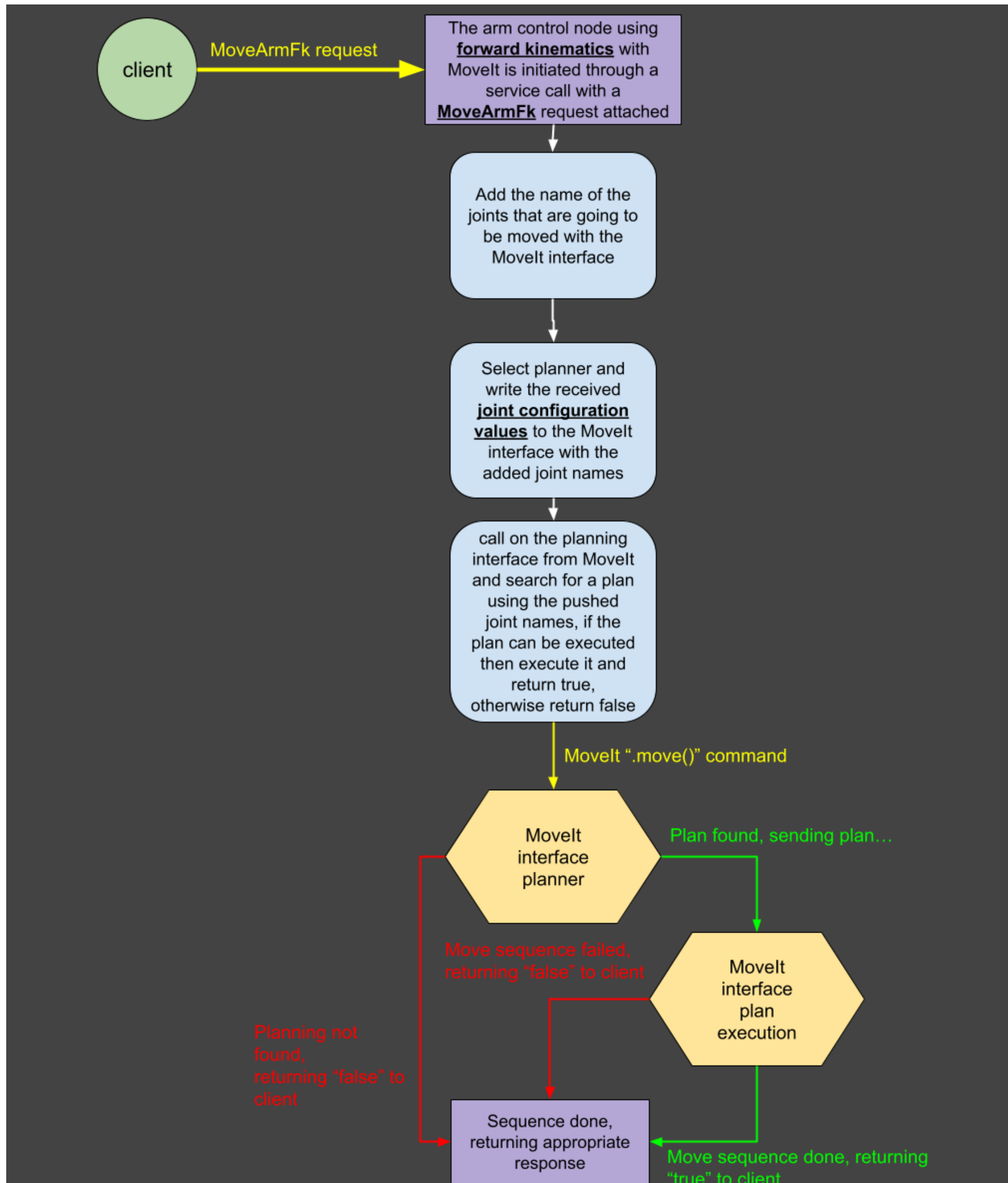


Figure 3.14: Function Scheme for Arm Control Node Utilizing Forward Kinematics with MoveIt

After the appropriate configuration has been made the targeted values can be sent to the planning interface. Planning for the motion involves selecting a planner, which can be summarized as a specific algorithm used for finding the necessary path for performing a movement. This planner should make a plan for the joints to reach their target, avoiding collisions and dangerous movements which might cause damage. After selecting the planner, MoveIt will create a plan using the configured values and additional information like the octomap to create a movement path. The final step is to call the move command and MoveIt will automatically communicate with the required interfaces using the created plan to perform the correct movements. The MoveIt library is therefore a powerful tool in terms of joint manipulation and can achieve complex movement sequences if the data received is sufficient enough. A simulated example of these movements can be seen in Figure 3.15.

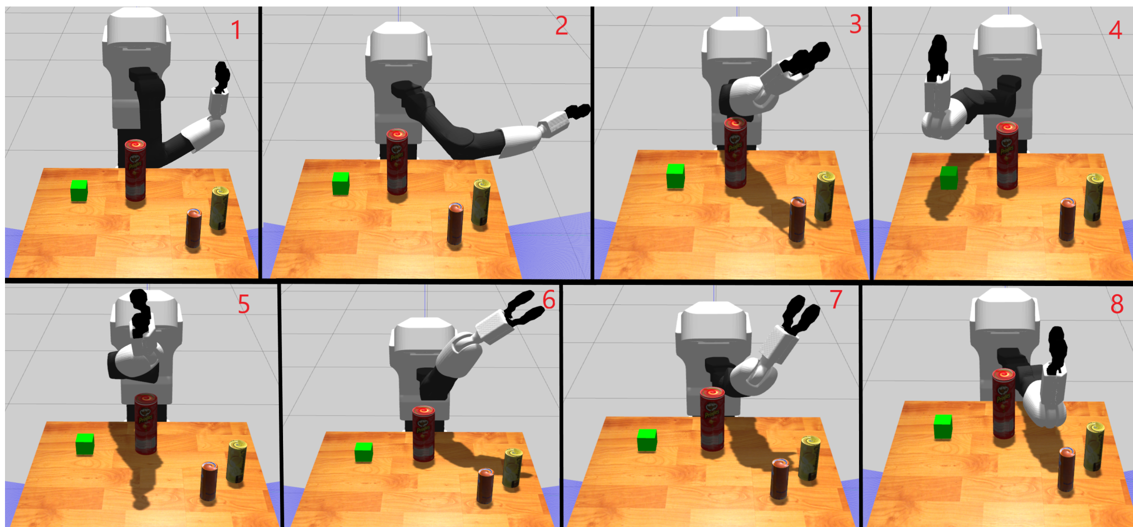


Figure 3.15: Complex Arm Coordination Using MoveIt to Avoid Collision with Objects on Table

The torso control provides a simple service for controlling movements of the torso. The service receives a joint configuration value for the torso and moves the torso to the specified joint configuration. This is done using the joint trajectory controller action with the torso controller selected. The service allows for precise movements which are needed when moving the gripper over the object, mainly avoiding collision between the gripper and the floor. The torso movements are used as a complement to the MoveIt interface since precise movements when moving a group of joints near the floor are risky and can easily go wrong, especially compared to solely moving the torso joint up and down, see Figure 3.16. Figure 3.17 below illustrates the control node for controlling the torso.

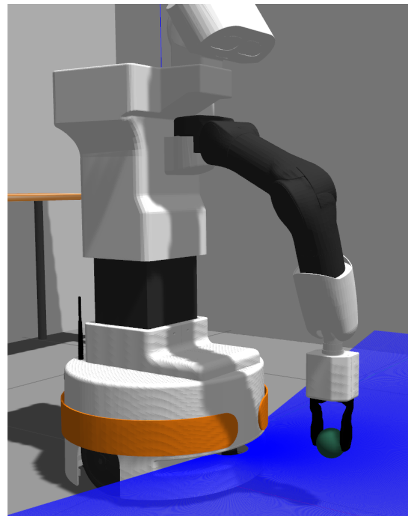


Figure 3.16: Torso Raised After Grasping Object

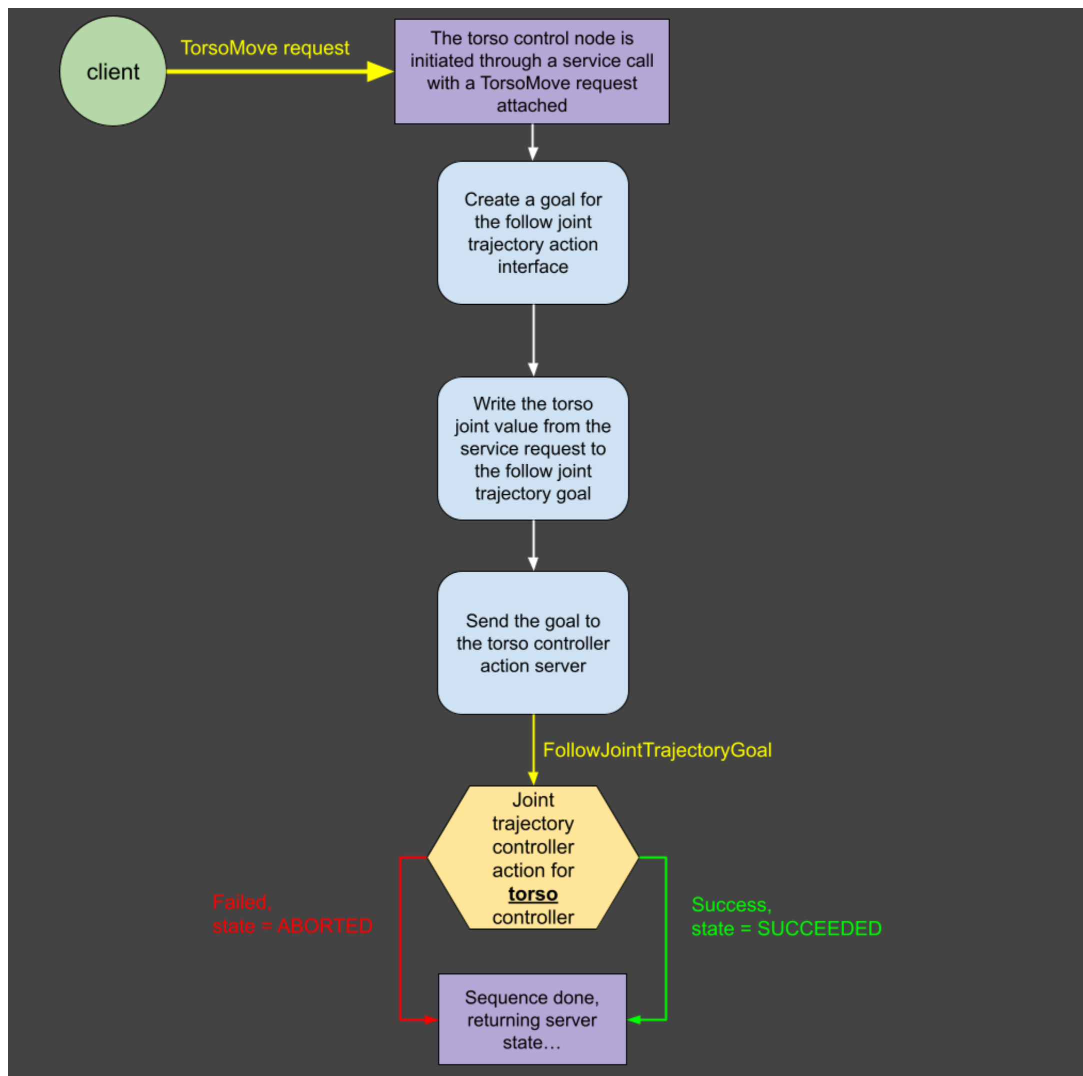


Figure 3.17: Function Scheme for Torso Control Node

To be able to grip and release the object (Figure 3.18) the TIAGo system provides a set of services. These services are called to start grasping or releasing the object. Other than saving time, the advantage of using these services is that they ensure that the motors of the gripper are not overheating, providing a safe way to grasp objects [54].

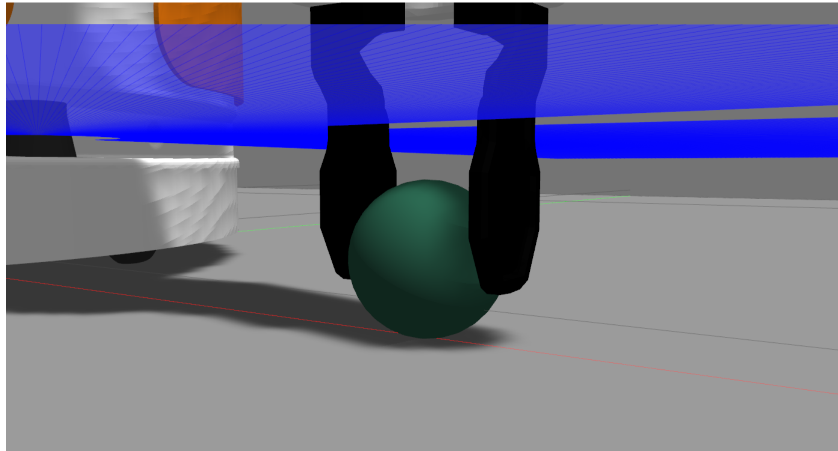


Figure 3.18: Gripper Grasping Object

3.3.2 Grasping Action Server

The grasping action server itself can be seen as an assembly of the control nodes with added workflow and coordination, see Figure 3.19. The server is provided with all the clients that communicates with the control node servers. It also contains the necessary service and action definitions for requests and goals that are written and sent by the clients. The grasping action server has its own action definition to provide the necessary information needed to grasp a located object. The goal definition for the server provides the cartesian coordinates x, y, and z of the object relative to the base of the robot along with the object height. The goal also consists of the option to either pick up or drop off an object. If told to drop, it can receive the choice of what box to drop the object into. The two options each have different workflows from each other, they differ in some movement calls but in general follow the same structure of: building an octomap, moving the arm, and then either grasping or releasing an object. Another feature of the grasping action server is that it checks for failures before moving on to the next step of the grasping sequence. This means that the server will abort the sequence if a service or action call fails, making the solution safer when called from other contexts.

The grasping sequence option begins with a call to the control node that builds the octomap. After the head movement action has succeeded and the octomap has been built, the server makes a call to the arm control node using inverse kinematics with MoveIt. The coordinates and height provided by the goal are used by the inverse kinematics control node to place the end-effector at an offset position over the object. The server then continues by calling the torso control node. From the previously acquired position, the torso control node can take over and precisely lower

the gripper over the object. From here on the provided TIAGo grasp service will be called and the object will be gripped. To complete the grasp, the server calls the torso control service once again to raise the torso. The grasping sequence for picking up an object is now complete and the server state is set to succeeded.

The release sequence consists of predefined movements with the only varying option being the box in which the object should be dropped into. The sequence begins with a call to the control node that builds the octomap. After the octomap is built it moves on to arm movement using forward kinematics and MoveIt. Here the joint configuration values are preset and depend on the box selection provided with the goal sent to the grasping action server. After the arm is moved and positioned above the selected box the server will make a call to the release service provided by the TIAGo system. This will make the gripper open and the object is dropped into the appropriate container. After this, a final call will be made to the arm control node using forward kinematics and MoveIt. The joint configurations provided in this call are also predetermined and represent a tucked arm position for the robot, the release sequence is now complete and the server state is set to succeeded.

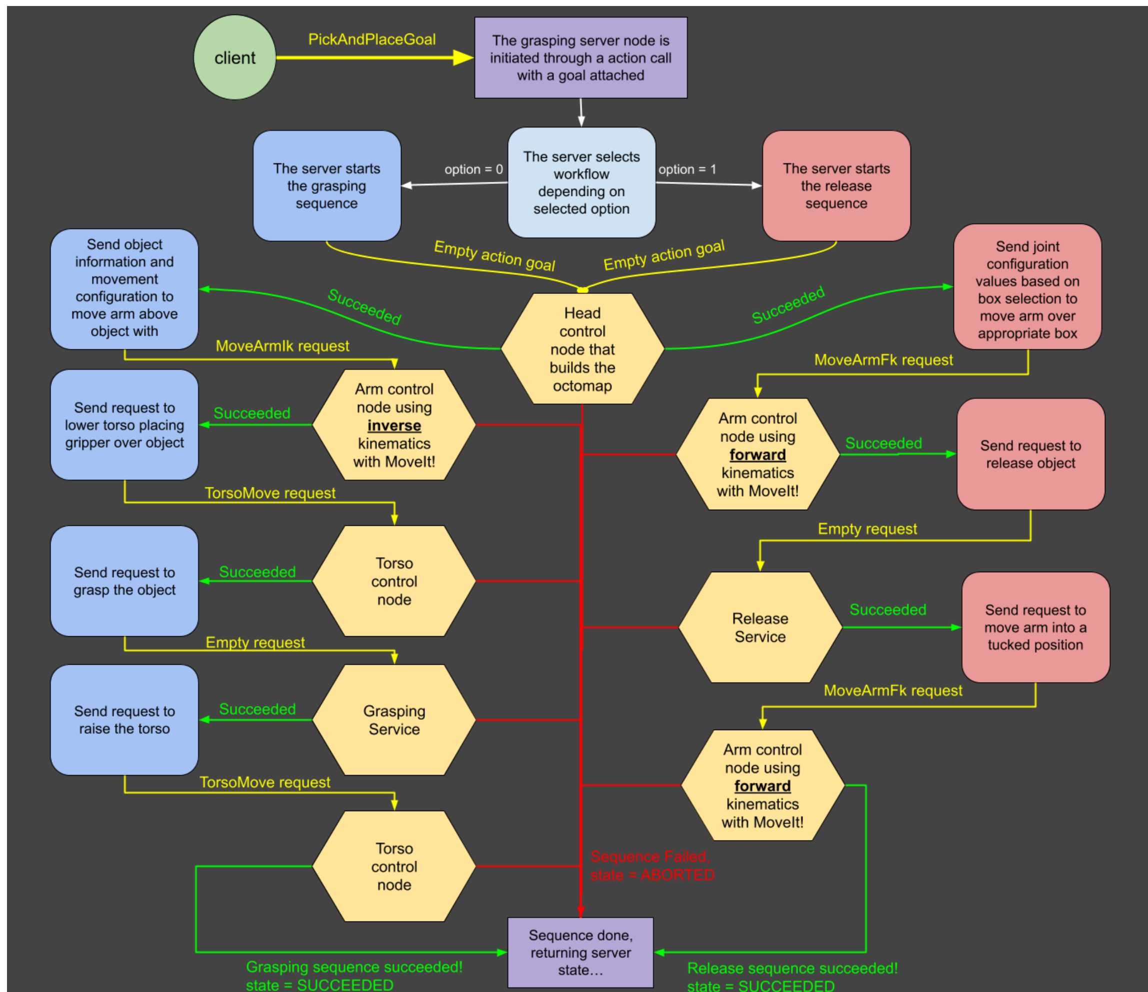


Figure 3.19: Function Scheme for Grasping Server

3.4 Integrating the Individual Solutions

The final step of the code-related work was to combine and integrate the navigation, object detection, and grasping solutions. This was done by creating a server called "clean_room". An illustration of the "clean_room" server and how it operates is shown in the flowchart in Figure 3.20. The server is initiated when it receives an empty goal from a client, which in turn executes its callback function. The callback executes procedures from each individual part asynchronously, as follows: It lowers the head to enable object detection, starts the navigation, and scans for accepted objects. When an accepted object is detected, it stops the navigation and allows for the node "center_camera" to position the robot and enable the grasping sequence. After the object is picked up the robot navigates to a drop-off area and calls the release sequence to place the object in an appropriate container.

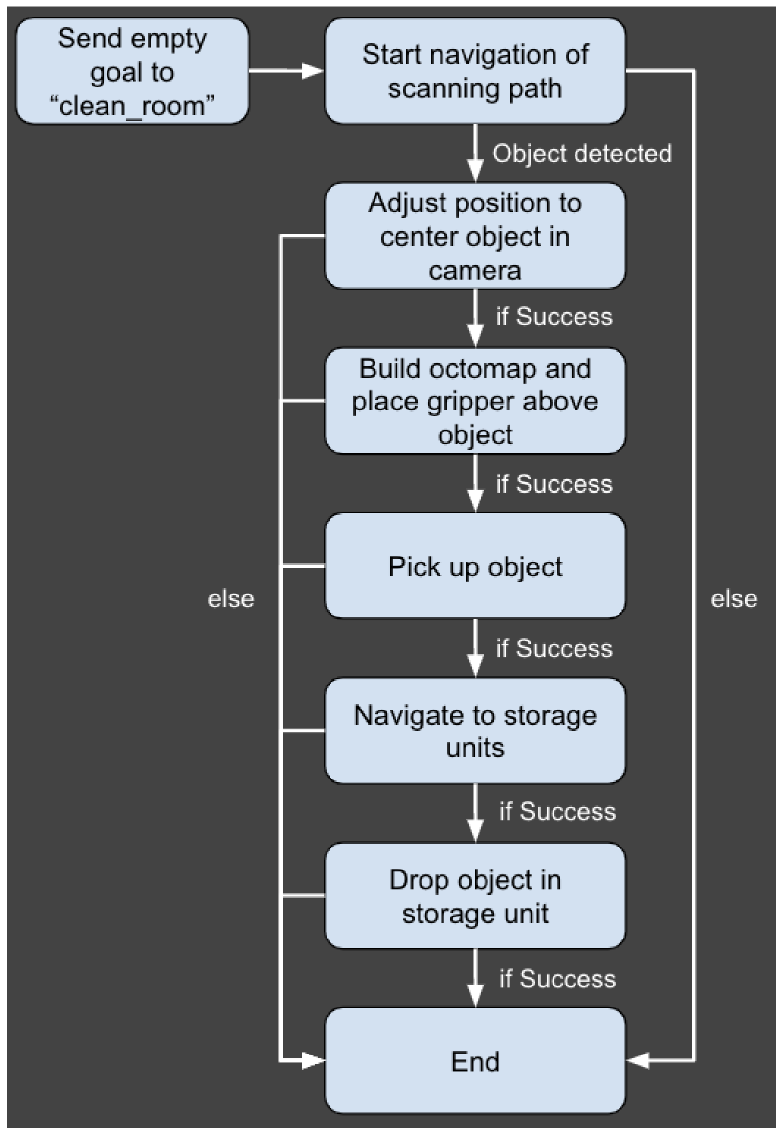


Figure 3.20: Flowchart for the Integration Solution

When an object has been detected and centered, the object coordinates are extracted using point cloud and sent to the grasping action server. The server proceeds to build an octomap in order to plan the arms movements with respect to the robot's surroundings. This enables the initiation of the grasping sequence. Once it is complete, the navigation is resumed with the storage units as the destination.

As the robot arrives at the storage units it performs the release sequence, and following the segmentation it does so above the corresponding storage unit. Once the item has been dropped off, the robot navigates back to the docking station, and the state variables are reset.

4
Results

This chapter presents the final state of the navigation, object detection, and grasping as individual parts. Then it proceeds to present the result of integrating them and implementing the pipeline on the real robot. In addition to this, challenges that were encountered and their impact on the result are also presented.

4.1 Final State and Result of the Navigation Solution

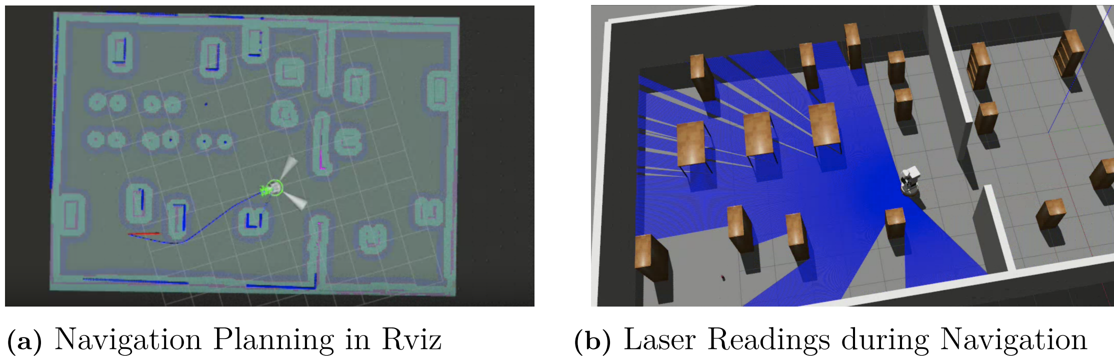
The navigation solution enables the robot to perform three predefined maneuvers. First and foremost, it is able to navigate to a set of waypoints along a predefined path whilst avoiding obstacles in its way. The waypoints are visualized in Figure 4.2 and the path planning and scanning of obstacles is visualized in Figure 4.1. Secondly, the navigation is pausable which is needed when an accepted object is detected and the grasping should start. When the navigation of the scanning path has been paused it is possible to go to the storage units, which are needed to store away an item after it has been picked up. When the destination of the storage units has been reached, the robot is able to resume the navigation of the scanning path from the latest visited waypoint. This means that the navigation can be interrupted at any time between the waypoints in the scanning path, navigate to the storage units, and then continue the search for more accept objects. Finally, if desired, it can also return to its initial position. Unfortunately, the results presented here are only true for the simulations. The navigation works as intended in the simulations except for problems when rounding outside corners when being close to the wall and detecting obstacles elevated from the floor, for example, a table. No tests involving the navigation were carried out on the real robot and are left for future studies. This is due to slip between the floor in the testing environment and the wheels of the robot, and uncertainties concerning the actuators that are operating the wheels. These problems appeared late in the project and could not be avoided in the remaining time, thus the navigation was not implemented or tested on the real robot.

4.1.1 Challenges in Navigation and Obstacle Avoidance

As mentioned, the robot avoided obstacles when it navigated along the path. However, problems occurred when it encountered a table or drove too close to the wall when approaching an outside corner in the simulation environment. The problem

with tables is illustrated in Figure 4.1b. The scanning laser is at ground level and only recognizes the legs of the table. This could easily be fixed with advanced navigation where the camera helps detect obstacles. The problem with the corners will be discussed in more detail in section 5.1.

Although this problem was avoided by placing the waypoints in a way that excludes these problematic obstacles. This means it had no problems navigating through the path, in the sense that the path avoids less problematic obstacles such as bookshelves or similar. The laser readings and the planned path are visualized in Figure 4.1 and in Figure 4.2.



(a) Navigation Planning in Rviz

(b) Laser Readings during Navigation

Figure 4.1: Simulation Interface Showing the Laser and the Objects It Detects as well as a Planned Navigation Path

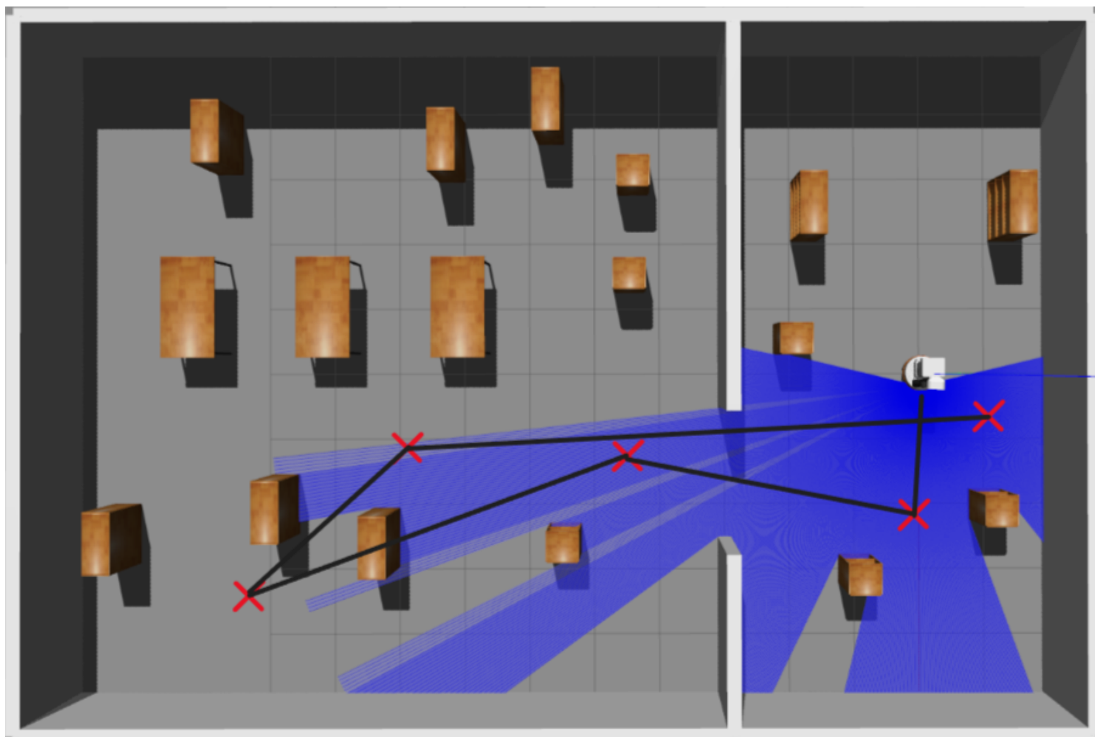


Figure 4.2: Visualization of the Path That the TIAGo Navigates

Besides challenges linked to the solution of the task, there were also issues when running the simulations. For the computers available for the project the simulation did not work without a dedicated NVIDIA graphics card. When using a computer without a NVIDIA GPU the laser sensors did not work and so the robot was not able to navigate at all. Fortunately, it worked on a PC with a dedicated NVIDIA graphics card which solved the problem of being unable to test the solutions.

4.2 Final State and Result of the Object Detection Solution

Currently, the object detection part of the project works in some instances. The custom-trained YOLO model is applied to the robot's image stream, and objects are tagged with an object category and confidence score. When an accepted object of sufficient confidence score is found, the robot navigates towards it with the help of the YOLO bounding boxes and uses pointcloud to find the object's coordinates in relation to the robot.

The YOLO object detection model was trained for 300 epochs and the best model was picked out by the YOLO training algorithm. After training, the validation and training loss, per epoch were acquired as shown in Figure 4.3. There is a clear improvement in training and validation loss during the first 100 epochs although after around the 100th epoch, the validation loss starts to increase, indicating that the model was starting to overfit the data. At around the same time, epoch-wise, the precision, recall, and mAP metrics start to plateau, indicating the model is nearing its performance limit with the data set.

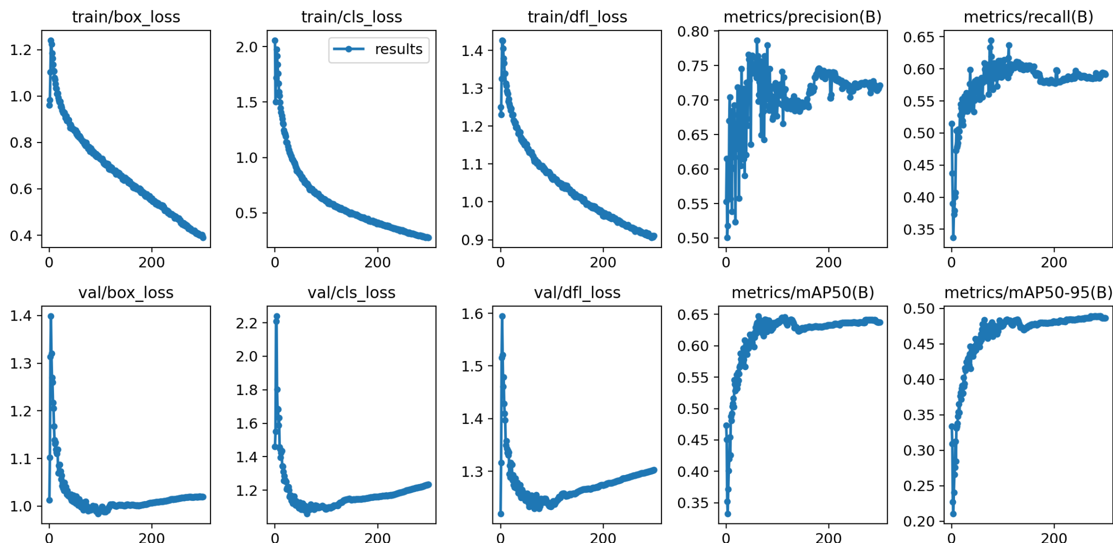


Figure 4.3: Loss Values and Metric Results of YOLO Object Detection Training

The confusion matrix presented in Figure 4.4, was also produced showcasing the accuracy of the model on the test dataset. As shown, the detection model is not accurate with either the eraser or empty backgrounds, which should've been expected considering that they had none or very few annotations in the dataset.

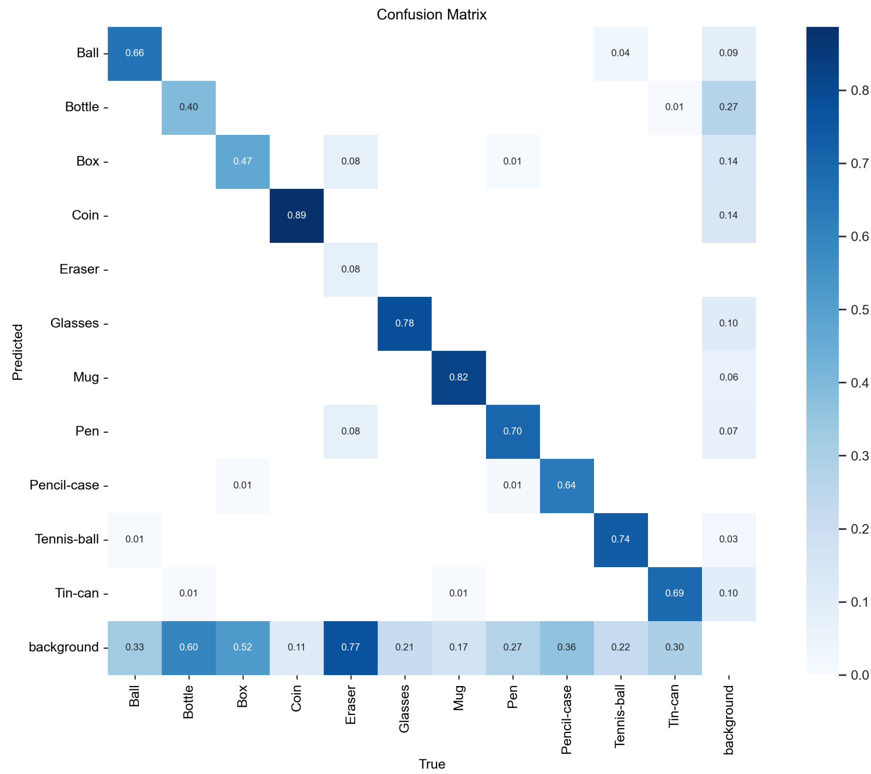


Figure 4.4: Confusion Matrix Showcasing the Accuracy of the Custom Trained Model

The model was tested on the real TIAGo RGB-D camera through recording with BAG files. It could successfully recognize several different types of objects and even identify grouped objects, see Figure 4.5 and 4.6. As seen in the picture, the BAG files have been recorded with a changed RGB value and all the images have swapped red and blue values. As this was the case it might have had an effect on the image recognition.

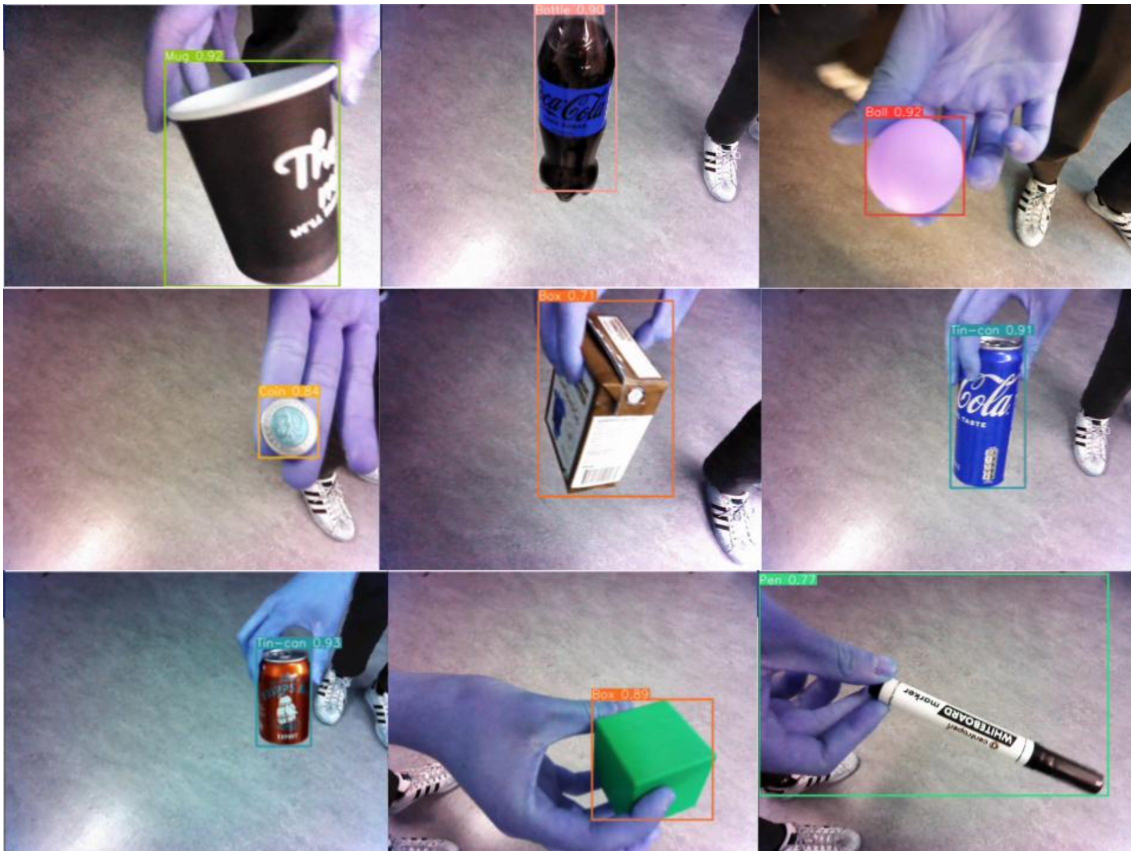


Figure 4.5: Example of Object Detection on the TIAGo RGB-D Camera

Figure 4.6: Example of Multiple Object Detection on the TIAGo RGB-D Camera

The center camera node worked well in the simulation and could even pick one object from a group of objects to navigate towards. This was not tested on the real robot, however, much for the same reasons as the navigation, see section 4.1.

The extraction of object coordinates could not be tested on the real TIAGO robot within the time constraints either, therefore the results are the same as presented in section 3.2.4 and section 4.4 in the gazebo simulation. At the end of the simulation, the coordinate extraction worked every time with the few scenarios that were run.

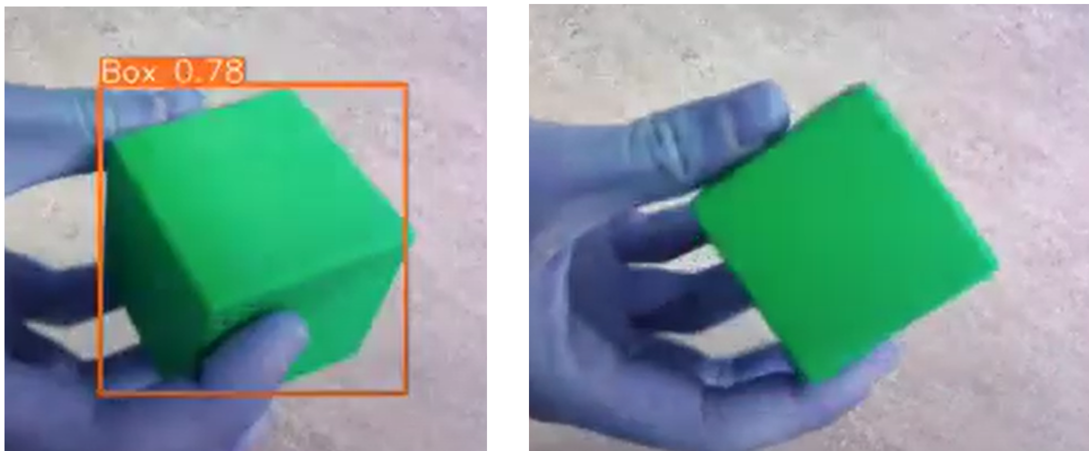
4.2.1 Challenges in Object Detection

The current simulations make use of green balls as accepted objects, these were easy for the model to detect, but the problems arose when other items were present. The custom-trained model is not very accurate with items of more complex shapes, values, and patterns which was revealed when replaying the BAG files. When testing the model, five distinctive problems were revealed:

1. The object was not detected at all (Figure 4.7).

Figure 4.7: Example of When the YOLO Model Does not Detect an Object

2. The object could only be detected from some angles or in close proximity to the camera (Figure 4.8).



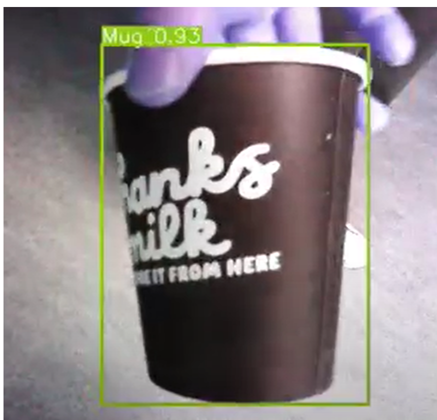
(a) Green Box Shown from the Side **(b)** Green Box Shown from the Front

Figure 4.8: The Green Box Is Only Detected from Some Angles

3. The objects confidence score was too low for the node to recognize it as an accepted item (Figure 4.9).

Figure 4.9: Example of When the YOLO Model Detects an Object with Low Confidence

4. The object was detected as another object (Figure 4.10).



(a) Coffee Mug Placed Close to the Camera Detected as "Mug"

(b) Coffee Mug Placed Far from the Camera Detected as "Ball"

Figure 4.10: Example of When the YOLO Model Detects an Object but Labels It Wrong

5. An object is detected where there is no object, the model gives a false positive (Figure 4.11).

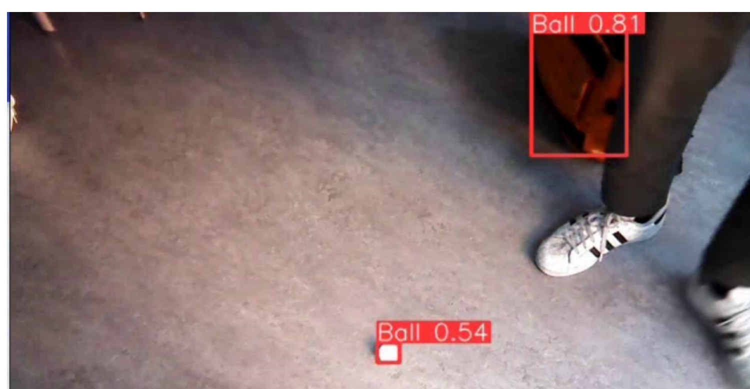


Figure 4.11: Example of When the YOLO Model outputs a False Positive

Furthermore, the solution does not currently sort the objects into different categories and containers, this was one of the aims that was not completed because of time constraints. The focus was put on making a full integration of all parts, managing one item at a time, and continuing from there. This part was not completed since the recognition model, as stated above, was not good enough at identifying object types accurately, thus preventing a solution for sorting.

To accelerate the real-time performance of the created object detection model, CUDA was supposed to be implemented. Pytorch did not recognize CUDA in the docker container which prevented the use of CUDA both for training and real-time detection. Training of the model could be done outside of the docker, however, whilst the real-time detection would not be able to access CUDA in the final solution. Although considering the speed of YOLO and the computing power of the PC that was built it did not have a major effect on the end result.

YOLO and CUDA require more computing power than was available in the laptops that were used throughout this project. Therefore the computer built for the project was used. The specs of the computer can be seen in Table A.1 under section **??**. Since CUDA did not work in the docker container, the computer ended up being used primarily for training the object detection algorithm and using YOLO live when running the simulation.

4.3 Final State and Result of the Grasping Solution

The solution for the grasping and release sequence, as presented under subsection 3.3, was implemented and tested in simulations accordingly, both as an individual solution as well as part of the combined solution involving the other modules of the project. The grasping solution required detailed evaluation to make sure the provided sequence was sufficient enough to be used in integration with the other modules. Criteria like functionality, coordination, and consistency are relevant to explore when analyzing the performance of the different parts controlled by the sequence. Furthermore, real-life testing was also done with some of the grasping control nodes to explore how well the solution works in a real-life setting, such testing is also valuable to examine the correspondence with the simulations done.

Functionality is a simple measure since it is only a matter of whether or not the robot performs as intended by design. The grasping server represents the process of either picking up or releasing an object in a safe manner i.e. avoiding collisions with the ground and other surroundings. From this perspective, the solution performed well in the simulated environment but faced some difficulties during the real-life testing. Tests in the simulated environment also revealed some issues that required tweaking of certain parameters, these problems were connected to the coordination of the grasping sequence and occurred when trying to grasp objects with different heights. There were some additional problems concerning the gripper and its ser-

vices, these only occurred when simulating in the docker container and revolved around predefined nodes not launching correctly.

The tests performed in the simulation and real life were first and foremost ideal scenarios where the object location and height were known variables. The first tests performed to initially get started with a working solution were based on checking the individual control nodes by calling them separately. An example of the arm being unfolded from manual commands can be seen in Figure 4.12. Manual testing was especially important during real-life testing because, unlike the simulation, wrong movements and collisions can actually cause damage to the robot. This also meant that all the parts needed to work in the simulation beforehand which they did. Problems instead appeared when trying to call the commands on the real robot, some of the interfaces working in the simulation did not work on the real robot, this combined with the limit of time meant that fast adaptations were required affecting parts of the end result. With the individual control nodes working in the simulation, the grasping server was easy to set up virtually and yielded adequate results when performing the whole automated sequence, both for the grasping and releasing. The same could not be said for the real-life testing. Here the testing posed issues with the arm, torso, and head movements, meaning no automated sequence could successfully be performed. The best result gathered under these circumstances was a monitored demo of the movements with manual calls to the control nodes. The nodes first required some quick maintenance to deal with the issues mentioned above but finally yielded a result in which the robot successfully picked up a bottle as seen in Figure 4.13.

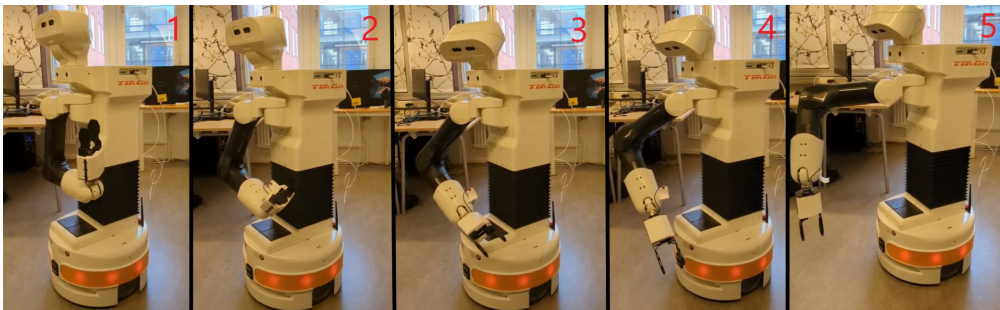


Figure 4.12: TIAGo Unfolding Its Arm During Real-Life Testing Using the Arm Control Node Utilizing Inverse Kinematics and MoveIt

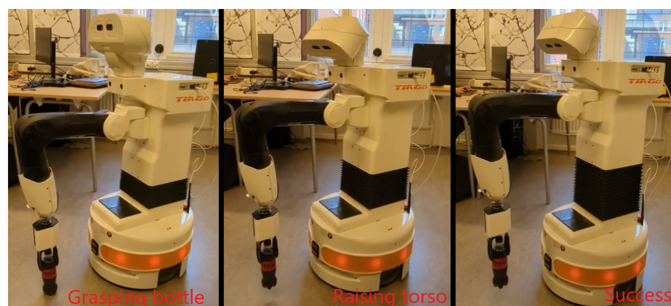


Figure 4.13: TIAGo Picking Up a Bottle During Real-Life Testing

4.3.1 Challenges in Grasping

The majority of the problems mentioned above are correlated to the real-life testing, but the simulation test also revealed some issues with coordination and consistency as well as some problems with the gripper. These problems will be discussed in detail here along with the proposed and implemented fixes.

The majority of nodes were developed and tested on a native version of ROS and TIAGo system meaning they were not placed into the docker container until later on in the project. After migrating the files to the docker container the grasping and release sequence was tested in the same simulations that were previously working on the native versions but suddenly the grasping and release service were no longer running meaning the gripper could neither grasp nor drop an object anymore. After investigating the issue a proposed solution of manually trying to launch the nodes responsible for the services was suggested. This worked for the service responsible for grasping since the launch files for the control node were easy to track down. The same could not be said for the launch file of the control node responsible for the release service. This file could not be found and therefore the release service could never be started in the docker container leading to the release sequence breaking.

During real-life testing there were several issues concerning the control of the different joints, both when calling and executing them. The control nodes were tested by doing monitored manual calls. During these tests all of the control node services and actions utilizing the follow joint trajectory controller action interface could not be called. This meant that both the head control node, responsible for building the octomap, and the torso control node, responsible for raising and lowering the torso, failed as the joint configuration values could not be sent to the joint controller action. Since the octomap could not be built, the sequence lost collision detection with the environment. However, in this controlled environment, it was acceptable as surrounding obstacles were moved aside, ensuring no collisions could occur. The bigger part of this issue lay in the fact that the torso could not be moved which is a crucial part of the grasping sequence.

The only control node that worked was the one utilizing MoveIt. Since MoveIt also has access to the torso control joint it became evident that to successfully complete the intended movements a MoveIt node for controlling the torso needed to be created. Because of the time limit of the testing, the node was recreated from the already existing arm control node utilizing forward kinematics with MoveIt. This control node manipulates the torso joint as well as all of the arm joints, this meant that to only move the torso all of the arm joints needed to maintain their current values. At the moment there was not any good way to fetch these values so a compensation was made to read the values and manually put them into the code. After creating this control node the torso could now be moved utilizing MoveIt instead, as seen in Figure 4.14, although with some manual configurations. Another issue faced was the fact that MoveIt did not seem to properly recognize where the robot's body was when moving the arm, in one instance it accidentally planned a path that would result in a clear collision with its own base. This movement was stopped before it

could cause any damage to the robot but this meant that the movement was not reliable on the real robot compared to how well it worked in the simulations.

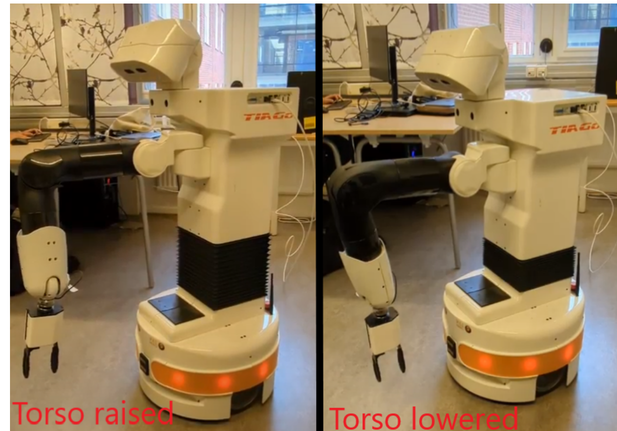


Figure 4.14: TIAGo Lowering Torso During Real-Life Testing

4.4 Result of the Integration between the Three Parts

The integration worked as expected in the simulation. When launching the simulation all the desired visualization tools started correctly, these tools included the RViz navigation, gazebo simulation, and YOLO window. Then when sending an empty goal to the clean_room the navigation started and the robot navigated to one waypoint at a time. An object of interest was detected, the navigation was canceled, and the camera centering took over. It centered the object in the camera and the object coordinates were extracted using the depth camera. The TIAGo then started building the octomap for the grasping and, when finished, grabbed the object and picked it up. Thereafter the robot navigated back to the intended location of the storage units, as expected.

However, as mentioned in section 4.3.1 the release service was not able to start. This meant that the part of the simulation where the items are placed in the storage units was not performed. A consequence of this was that the robot started tucking its arm and opening its gripper which resulted in an inaccurate drop-off. Once this erroneous maneuver was performed, a new empty goal was sent to the clean_room node which makes it resume the navigation. This means it is repeating the process described above until it finishes navigating the path without detecting another accepted object. The resulting integration is illustrated in Figure 4.15.

In the simulation a limited amount of different graspable objects were available and the only object recognized by the image recognition was a ball. This made the sorting for the storage units hard to implement. The fact that the grasping service did not work was not helpful either. So there was no sorting tested when the intended drop-off would happen.

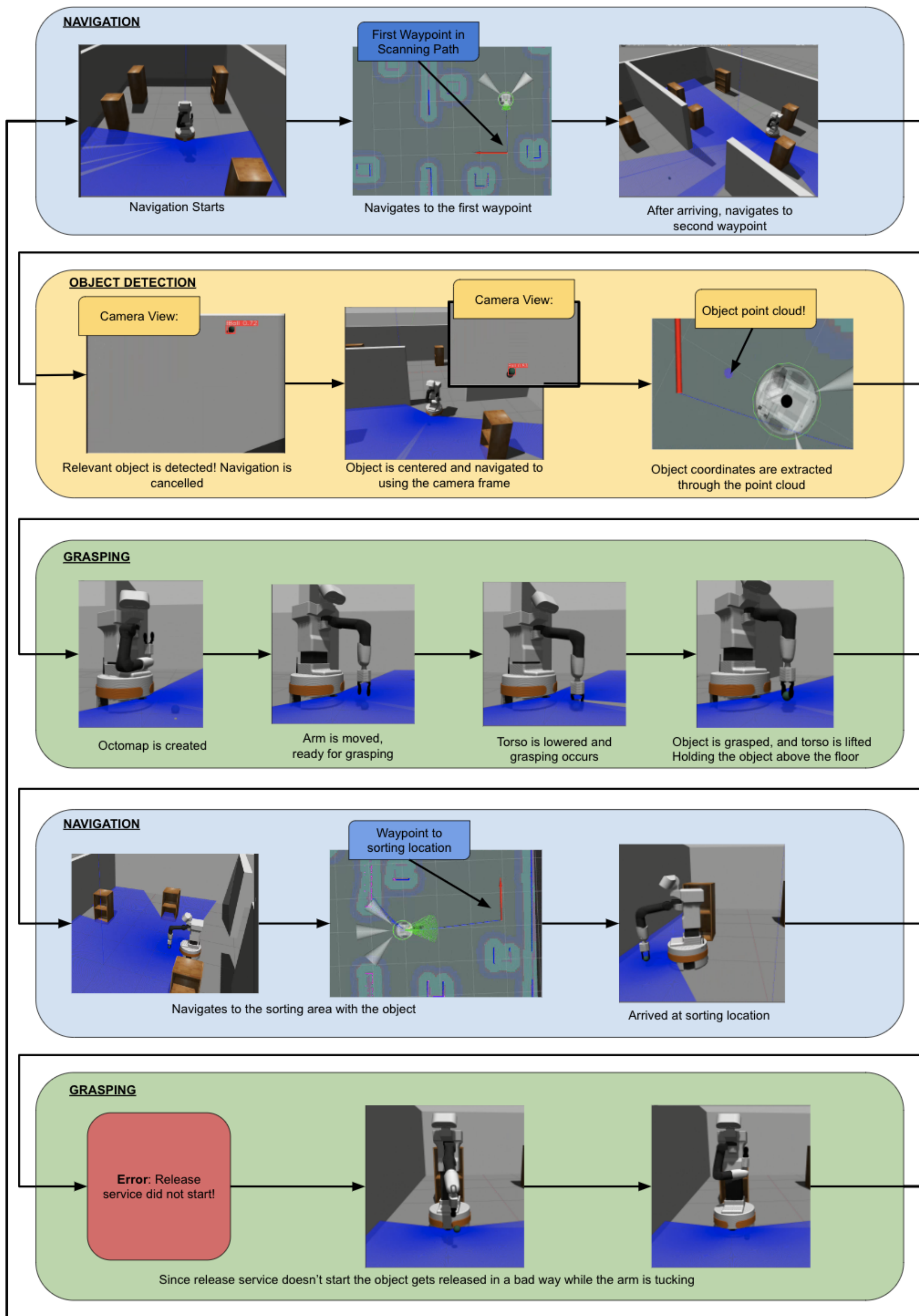
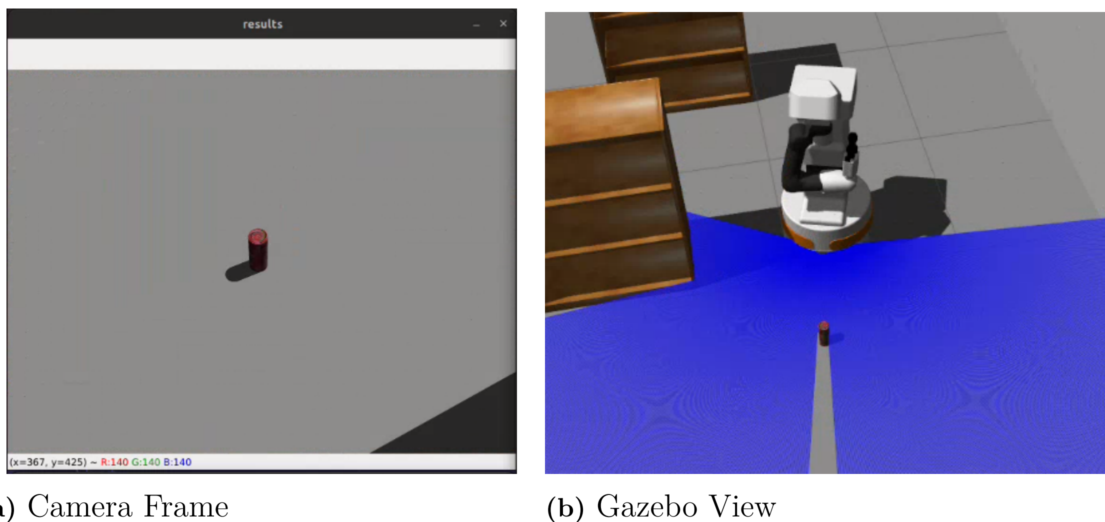


Figure 4.15: The Results of the Integration Inside the Simulation

4.4.1 Challenges in Integration

A challenge yet to be overcome regarding the navigation is to figure out how the advanced navigation can be used without interfering with the movement of the head. When integrating the navigation with the other parts of the project, the advanced navigation had to be turned off. As a consequence of this, the robot had problems avoiding obstacles elevated from the floor. The reason for this is that the robot now only uses the laser, which is located at the base of the robot, for obstacle detection. This means that it is unable to detect a surface above ground level.

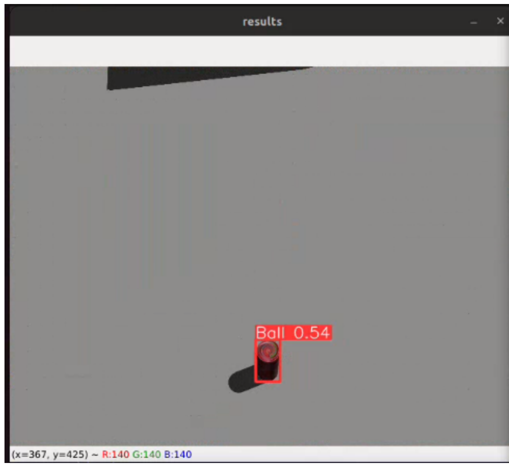
The advanced navigation was turned off because the head needs to remain in a downward-facing position for both the centering of the object within the camera frame and the coordinate extraction. Furthermore, the advanced navigation overwrites the head control of the grabbing nodes. This interferes with the head movement that the grasping action node uses to build the octomap needed for grasping and releasing. Another problem encountered was that the robot was not close enough to grasp the item after centering it, see Figure 4.16.



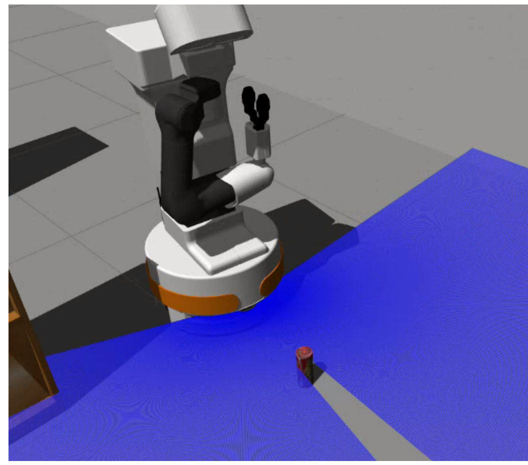
(a) Camera Frame **(b)** Gazebo View

Figure 4.16: A Centered Object Which Is Not Close Enough for Grasping

This was solved by changing the parameters in the node so that the robot navigated further towards the object, positioning it in the bottom center of the camera frame instead. This closed the gap between the robot and the item (Figure 4.17).



(a) Camera Frame



(b) Gazebo View

Figure 4.17: An Object in the Lower Part of the Camera Frame is Close Enough for Grasping

5
Conclusion

This chapter concludes what worked and suggests the continuation of the work related to the navigation, object detection, and grasping on an individual level. Thereafter, the same conclusions are drawn for the integration of all parts and the project as a whole.

5.1 Performance of the Navigation: Conclusion and Future Considerations

This section concludes the work related to the navigation. In other words, what worked and the suggested continuation to this part of the project.

5.1.1 What Worked

When testing the navigation in simulation it worked almost as intended. More specifically it was able to navigate through the room but had some problems making sharp turns around outside corners. It was also able to avoid elevated objects if the camera was faced in a downward position. It was able to navigate the predetermined path without any problems as the waypoints were set to avoid sharp turns around any outside corners. The navigation could also be interrupted to allow for grasping and continued when the item should be stored away or had been stored away. In other words: interrupted when grasping should be performed and continued after it had been performed.

5.1.2 Continuation of the Work

The obvious continuation of the work regarding the navigation, on an individual level, is to test it on the real robot. Since it worked mostly as intended in the simulation is it yet to be seen whether adjustments need to be made in order to make it work in the real testing environment. As mentioned in section 4.1, this is a problem related to the wheels of the robot and the floor in the testing environment. A suggestion for solving the slip problem would be cleaning the floor and the wheels of the robot. It was brought to attention that the room had not been cleaned as often as it should due to limitations of access because of the expensive and valuable equipment stored there.

Before testing on the real robot, a solution to the problems regarding turns around

corners should be found. During testing in the simulations, the robot continuously drove into corners and got stuck. Two plausible reasons for this were devised. The first reason might be that the distance-measuring laser got faulty measurements. As the robot approaches a corner, its laser sensor extends beyond the corner before the physical body of the robot, resulting in a misleading perception of being farther away from the wall than it actually is, see Figure 5.1. This might prompt the robot to turn preemptively and drive into the wall.

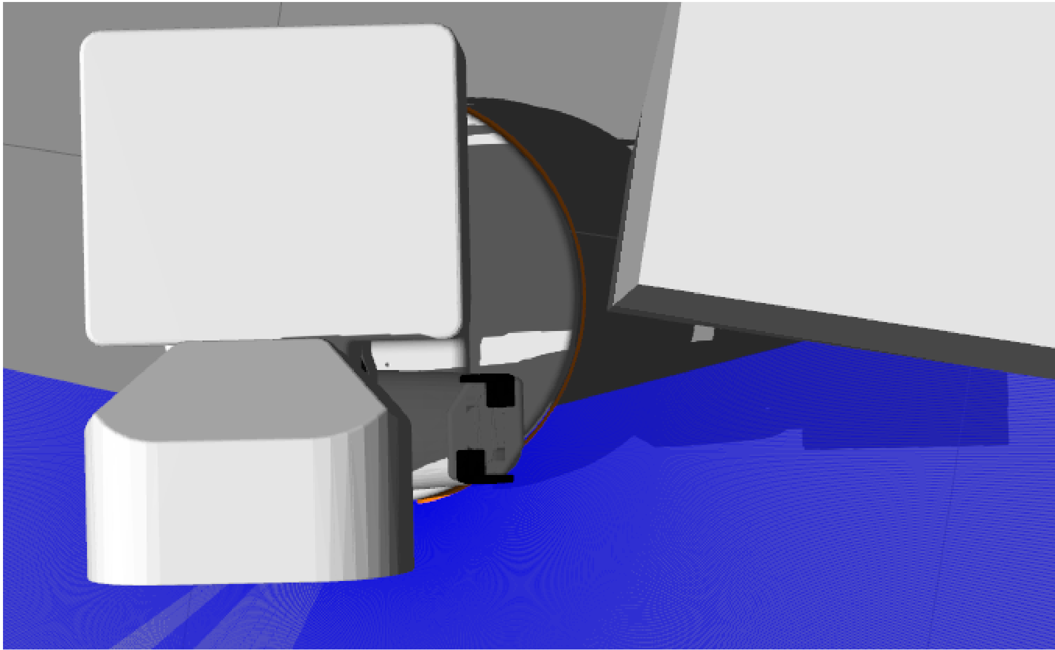


Figure 5.1: Visualization of Distance Reading from Laser

The second reason could be that the robot believes that it has collided with an obstacle when it really has not. When the robot collided with an obstacle, the robot's camera was used to inspect the collision area. It could be seen that the robot actually was very close to an obstacle (see Figure 5.2), but had in fact not hit anything, it only acted as if it had. This could easily be translated to the corner problem where the robot thinks it has collided with the corner after approaching.

Even though this is a reasonably big issue it could temporarily be solved by putting the waypoints away from corners to avoid these critical situations. This navigational problem appeared both in the simulation environment for the project and the environment provided in the practice repository "TIAGo tutorials".

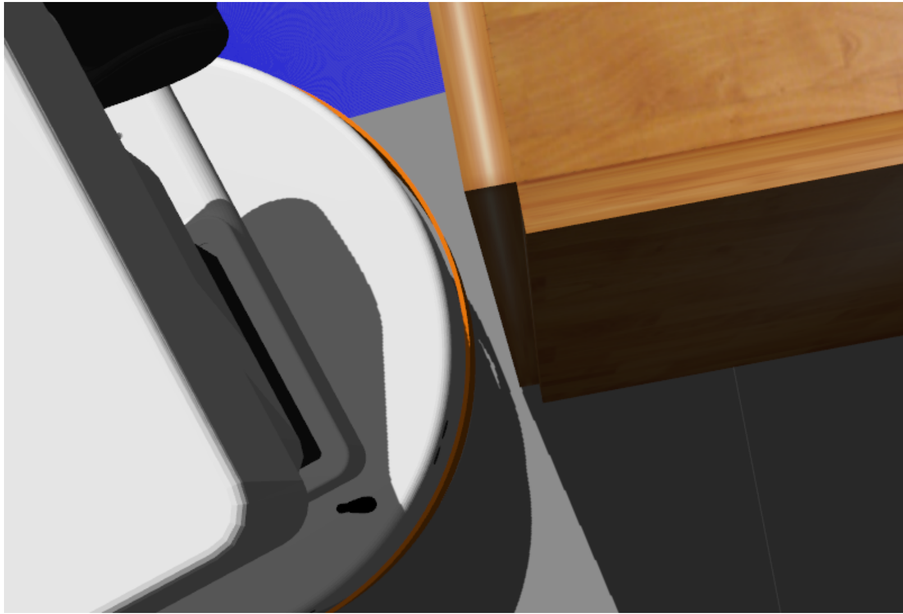


Figure 5.2: Gap Between Robot and Obstacle During Collision

5.2 Object Detection: Conclusion and Future Considerations

This section concludes the work related to the object detection.

5.2.1 What Worked

The object detection with YOLO was able to be integrated into ROS although the object detection had issues. The model could identify some objects quite accurately, especially balls. Adding to that, it could differentiate the objects and apply a bounding box around them. This was used for tracking the center of the object in real-time. This simplified the usage of the point cloud and gripping. The coordinate extraction worked well in the simulated environment but was not able to be tested.

5.2.2 Continuation of the Work

To further develop the object detection, a new YOLO model should be trained to identify more objects from different angles and distances. This would significantly improve the performance of the solution as it would enable more reliable detection and identification of different objects. This would be done by training the model on a different data set with more angles of the objects in different light settings. The reason for this is that the object detection task requires the object detection to work with objects seen from the head of the TIAGo robot, at a high angle. Therefore it would most likely increase the performance of the model if the data set included, and trained on, more images from a high angle. Furthermore, the data set should be improved by having more balanced data. For instance, the classes eraser and tennis ball had very few instances in the data set which made the model's performance at

detecting those unsatisfactory. More classes should also be added to the data set so that the model can detect more everyday objects and be more versatile.

Another issue regarding the model was that it frequently gave false positives, i.e. it detected an object when there were none, see Figure 4.11. Adding images of the background, without any objects that the model is being trained on the present, could improve this. Since the model gets to train on scenarios when no objects should be detected, false positive rates are decreased and the model becomes more accurate.

The current tests are carried out on gray vinyl flooring which did not vary in height or texture. The model's reaction to different backgrounds and materials like floor types and carpets could be explored to prepare it for all types of homes and environments. A solution for the conflict between PyTorch and CUDA would be a great addition since faster real-time object detection would speed up computation, thus enabling the use of bigger YOLO models. Bigger models could create higher confidence scores and improved predictions overall.

A final improvement would be to create a solution for when false positives are detected and then stops being detected. The robot will currently stop navigation and start the "center_camera" node, which will control the robot until the object is no longer detected. The system will then stop completely as no object is present but the "center_camera" node is running and navigation is canceled. This could be solved by implementing a timeout for the "center_camera" service where the navigation will continue if the service response is not received in a specific time. For the coordinate extraction parts, future work would be to implement this on the real robot, instead of inside the gazebo simulations. As for the object coordinate extraction using the point cloud further development to implement this with the real TIAGO robot.

5.3 Grasping: Conclusion and Future Considerations

This section concludes the work related to the grasping.

5.3.1 What Worked

The grasping solution worked well in the simulation and most of the issues only occurred when trying to put the solution on the real robot. The grasping solution integrated in this project relies much on the function of the libraries it utilizes such as MoveIt and the action interfaces for controlling the joints that are provided by the TIAGo system. The development of the grasping and release sequence is a combination of structured calls to these systems along with desired values attached. If the values are adequate and the calls are successful the grasping solution works provided that the integrated systems work.

5.3.2 Continuation of the Work

For further work on the grasping solution, the next step would revolve around investigating the issues faced during the real-life implementation. Finding out what caused the issue with MoveIt and the joint controller interface and getting the control nodes to work as intended on the real robot would be a first step to a real working solution for the whole project. For future development of the grasping solution, one could also explore more advanced grasping sequences. This could involve testing the limits of collision detection by trying to grasp objects in hard-to-reach locations. New research could also involve grasping objects that are very close to the floor and hard to grasp like clothing. Future projects could also investigate the use of other grippers like a humanoid hand for more precise gripping.

5.4 Integration of the Parts: Conclusion and Future Considerations

This section concludes the work related to the integration of all parts. In other words, what worked and the suggested continuation to this part of the project.

5.4.1 What Worked

Integration of the different parts worked in the simulations except for the fact that the grasping services struggled. This caused another problem in itself which were that the objects were not placed in any storage units. Due to struggles with the navigation, covered in section 5.1, no tests of the integration were made in the real world on the robot.

5.4.2 Continuation of the Work

One issue to be considered in the continuation of the work is the advanced navigation. It would be good to have it included but that means figuring out how to turn off the panning on of the head when it is not navigating. Solving this problem would eliminate the problem of detecting obstacles elevated from the floor and improve the navigation.

Testing the integration in the real world would be another thing that would be important for future studies. Though to do this the navigation need to be tested and working outside of the simulation.

When integrating the object detection nodes, there was a long deliberation of how these should work together with the solution as a whole. The first idea was to only interact through publishers and subscribers. This was partly used in the final solution, the "camera" node subscribes to the image stream and then publishes bounding box coordinates. The "center camera" node, however, worked better when it could be started as a service and return an empty message when finished centering.

This gives the main node better control over when the node should pilot the robot and not.

Another thing is the storage units which do not exist at the moment. This is mainly because of the model used for object detection. As covered in section 4.2 the model has problems distinguishing what object it looks at and in the simulation the amount of objects to test on is limited. As a result of this, balls were the only detectable objects in the simulation and beyond that the grasping services did not launch which hindered the dropping of objects. So, to successfully implement this a new better model need to be used and the grasping services need to be started properly. If that could be done it would simply require different arm joint configurations for different types of objects which would not be hard to implement.

5.5 Final words

To summarize the thesis about enabling a TIAGo robot to automatically clean a room, more time would be needed to get it fully operational in the real world. Most of the time spent on the project has been put into learning how ROS works including its concepts, system and the specific APIs for the TIAGo robot. This learning took a lot of time from the implementation and creating the different packages for the navigation, object detection, and grasping. At the current state the navigation works partly in the simulations with problems when rounding outside corners. The image recognition has room for improvement which would be possible by training new and bigger models. New better models would mean that the objects can be distinguished from each other which results in a possibility to get the sorting working as well. The grasping could use some fine tuning but works fine in the simulations except for the drop-off action. We as a group see great potential to finish the task and see results given from tests in the real world if more time was available.

Bibliography

[1] “TIAGo Handbook,” 1.29 ed., PAL Robotics, Barcelona, Spain, pp. 9–25, 2020.

[2] "Så funkar det: cobot". ABB. (n.d.). Accessed on: May 11, 2023. [Online]. Available: https://new.abb.com/se/om-abb/teknik/sa-funkar-det/cobot

[3] “COBOT APPLICATIONS,” Universal Robots, Nov. 10, 2020, Accessed on: May 12, 2023. [Online]. Available: https://www.universal-robots.com/blog/cobot-applications/

[4] S. Fran, Speaker, C. Collados, Dr. R. R Murphy and Dr. T. Luby, Interviewees, “Cobots in Healthcare - Transforming Medical Services,” *The Robot Podcast*, Mar. 21, 2021. [Podcast]. [Online]. Available: https://global.abb/group/en/media/podcasts/the-robot-podcast/s01/e04

[5] J. Wright, “Inside Japan’s long experiment in automating elder care,” Jan. 9, 2023. [Online]. Available: https://www.technologyreview.com/2023/01/09/1065135/japan-automating-eldercare-robots/, Accessed: May 11, 2023.

[6] P. Johansson, S. Pei, and J. Åkesson, “Enabling cobots to automatically identify and grasp household objects,” Chalmers University of Technology, Gothenburg, unpublished.

[7] “TIAGo Handbook,” 1.29 ed., PAL Robotics, Barcelona, Spain, 2020.

[8] “TIAGo: The mobile manipulator robot that fits and adapts to your research, not the other way around,” PAL Robotics, (n.d.), Accessed on: May 9, 2023. [Online]. Available: https://pal-robotics.com/robots/tiago/

[9] “TIAGo Handbook,” 1.29 ed., PAL Robotics, Barcelona, Spain, pp. 179–197, 2020.

[10] “TIAGo Handbook,” 1.29 ed., PAL Robotics, Barcelona, Spain, p. 182, 2020.

[11] “TIAGo Mobile Manipulator Robot: a platform for navigation,” PAL Robotis, Oct. 30, 2020, Accessed on: May 15, 2023. [Online]. Available: https://blog.pal-robotics.com/versatility-tiago-research-platform-navigation/

[12] “ROS/Introduction,” ROS Wiki, Aug. 8, 2018, Accessed: May 9, 2023. [Online]. Available: http://wiki.ros.org/ROS/Introduction

[13] “ROS - Robotic Operating System,” ROS, (n.d.), Accessed: May 9, 2023. [Online]. Available: https://www.ros.org/

[14] “Why ROS?” ROS, (n.d.), Accessed: May 9, 2023. [Online]. Available: https://www.ros.org/blog/why-ros/

[15] “ROS/Concepts,” ROS Wiki, Sep. 20, 2022, Accessed: May 9, 2023. [Online]. Available: http://wiki.ros.org/ROS/Concepts

[16] “Parameter Server,” ROS Wiki, Nov. 8, 2018, Accessed: May 9, 2023. [Online]. Available: http://wiki.ros.org/Parameter%20Server

[17] "Bags," ROS Wiki, Feb. 15, 2022, Accessed: May 9, 2023. [Online]. Available: http://wiki.ros.org/Bags

[18] ROS Wiki, "Client Libraries," Dec. 17, 2020, Accessed: May 9, 2023. [Online]. Available: http://wiki.ros.org/Client%20Libraries

[19] "YAML Ain't Markup Language (YAML^{TM}) version 1.2," YAML, Oct. 2021, Accessed: May 9, 2023. [Online]. Available: https://yaml.org/spec/1.2.2/

[20] "msg," ROS Wiki, Jan. 31, 2023, Accessed: May 9, 2023. [Online]. Available: http://wiki.ros.org/msg

[21] "Nodes," ROS Wiki, Dec. 4, 2018, Accessed: May 9, 2023. [Online]. Available: http://wiki.ros.org/Nodes

[22] "Services," ROS Wiki, Jul. 18, 2019, Accessed: May 9, 2023. [Online]. Available: http://wiki.ros.org/Services

[23] "actionlib," ROS Wiki, Oct. 30, 2018, Accessed: May 9, 2023. [Online]. Available: http://wiki.ros.org/actionlib

[24] "Create a kinematics solution using IK Fast," ROS Wiki, Oct. 21, 2015, Accessed: May 9, 2023. [Online]. Available: http://wiki.ros.org/Industrial/Tutorials/Create_a_Fast_IK_Solution

[25] "Inverse kinematics (IK) algorithm design with MATLAB and Simulink," MathWorks, (n.d.), Accessed: May 9, 2023. [Online]. Available: https://se.mathworks.com/discovery/inverse-kinematics.html

[26] "Kinematics — MoveIt Documentation: Humble documentation," MoveIt, 2023, Accessed: May 9, 2023. [Online]. Available: https://moveit.picknik.ai/humble/doc/concepts/kinematics.html

[27] "Table segmentation (C++)," ROS Wiki, Feb. 24, 2023, Accessed: May 9, 2023. [Online]. Available: http://wiki.ros.org/Robots/TIAGo/Tutorials/PointCloud

[28] "PCL Overview," ROS Wiki, Feb. 7, 2022, Accessed: May 9, 2023. [Online]. Available: http://wiki.ros.org/pcl/Overview

[29] R. B. Rusu and S. Cousins, "3D is here: Point Cloud Library (PCL)," in *IEEE International Conference on Robotics and Automation (ICRA)*. Shanghai, China: IEEE, May 9-13 2011.

[30] "Planning with Octomap demo," ROS Wiki, Feb. 24, 2023, Accessed: May 9, 2023. [Online]. Available: http://wiki.ros.org/Robots/TIAGo/Tutorials/MoveIt/Planning_Octomap

[31] A. Hornung, K. Wurm, M. Bennewitz, C. Stachniss, and W. Burgard, "OctoMap: An Efficient Probabilistic 3D Mapping Framework Based on Octrees," *in Autonomous Robots*, 2013, Accessed: May 9, 2023. [Online]. Available: http://octomap.github.io/

[32] "roslaunch," ROS Wiki, Oct. 23, 2019, Accessed: May 9, 2023. [Online]. Available: http://wiki.ros.org/roslaunch

[33] "play_motion_builder," ROS Wiki, Oct. 21, 2020, Accessed: May 9, 2023. [Online]. Available: http://wiki.ros.org/play_motion_builder

[34] "tf2," ROS Wiki, Mar. 5, 2019, Accessed: May 9, 2023. [Online]. Available: http://wiki.ros.org/tf2

[35] "tf," ROS Wiki, Oct. 2, 2017, Accessed: May 12, 2023. [Online]. Available: http://wiki.ros.org/tf

[36] "About Gazebo," Open Robotis, (n.d.), Accessed on: May 14, 2023. [Online]. Available: https://gazebosim.org/about
[37] "Visualization and logging," Open Robotis, (n.d.), Accessed on: May 14, 2023. [Online]. Available: https://classic.gazebosim.org/tutorials?tut=drcsim_visualization&cat=drcsim
[38] Automatic Addison, "What is the Difference Between rviz and Gazebo?" *Automatic Addison*, Jun. 24, 2020, Accessed on: May 14, 2023. [Online]. Available: https://automaticaddison.com/what-is-the-difference-between-rviz-and-gazebo/
[39] "What is a Container?" Docker, (n.d.), Accessed: May 9, 2023. [Online]. Available: https://www.docker.com/resources/what-container/
[40] "YOLO: A Brief History," Ultralytics, Dec. 5, 2022, Accessed: Apr. 6, 2023. [Online]. Available: https://docs.ultralytics.com/#yolo-a-brief-history
[41] C. M. Bishop, "Neural networks and their applications," *Review of Scientific Instruments*, vol. 65, no. 6, pp. 1803–1832, June 1994. [Online]. Available: https://doi.org/10.1063/1.1144830
[42] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning.* MIT Press, 2016, ch. Chapter 5: Machine Learning Basics, http://www.deeplearningbook.org.
[43] J. Brownlee, "What is the difference between a batch and an epoch in a neural network," *Machine Learning Mastery*, vol. 20, 2018, https://deeplearning.lipingyang.org/wp-content/uploads/2018/07/What-is-the-Difference-Between-a-Batch-and-an-Epoch-in-a-Neural-Network_.pdf.
[44] "CUDA Zone," NVIDIA, (n.d.), Accessed: Apr. 11, 2023. [Online]. Available: https://developer.nvidia.com/cuda-zone
[45] "Detect in Ultralytics YOLOv8 Docs," Ultralytics, Mar. 12, 2023, Accessed: May. 9, 2023. [Online]. Available: https://docs.ultralytics.com/tasks/detect/
[46] "Open Images V7," Google, Oct. 2022, Accessed: May. 9, 2023. [Online]. Available: https://storage.googleapis.com/openimages/web/index.html
[47] V. Angelo, "OIDv4_ToolKit 2018," Jun. 25, 2019, Accessed: May. 9, 2023. [Online]. Available: https://github.com/EscVM/OIDv4_ToolKit
[48] J. Solawetz, "How to convert openimages csv to yolo darknet txt," (n.d.), Accessed: May. 9, 2023. [Online]. Available: https://roboflow.com/convert/openimages-csv-to-yolo-darknet-txt
[49] "Image Preprocessing," Roboflow, May 4, 2023, Accessed: May. 9, 2023. [Online]. Available: https://docs.roboflow.com/image-transformations/image-preprocessing
[50] "Train - Ultralytics YOLOv8 Docs," Ultralytics, Mar. 12, 2023, Accessed: May 9, 2023. [Online]. Available: https://docs.ultralytics.com/modes/train/
[51] "TIAGo Handbook," 1.29 ed., PAL Robotics, Barcelona, Spain, p. 112, 2020.
[52] "Region Based Segmentation (C++)," ROS Wiki, Feb. 24, 2023, Accessed: May 10, 2023. [Online]. Available: http://wiki.ros.org/Robots/TIAGo/Tutorials/RegionSegmentation
[53] "Region Growing Segmentation Tutorial," Point Cloud Library (PCL), (n.d.), Accessed: May 11, 2023. [Online]. Available: https://pcl.readthedocs.io/projects/tutorials/en/latest/region_growing_segmentation.html

[54] "TIAGo Handbook," 1.29 ed., PAL Robotics, Barcelona, Spain, p. 151, 2020.

A

Appendix 1

A.1 YAML

YAML is the name of the language syntax used for ROS communication files. As stated in the YAML specification v1.2.2: "YAML (a recursive acronym for “YAML Ain’t Markup Language”) is a data serialization language designed to be human-friendly and work well with modern programming languages for common everyday tasks.". An example of one of the use cases for the YAML syntax in the ROS architecture is the easy data type specifications in messages. The YAML format supports integers, floating point numbers, strings as well as lists and dictionaries among other data types, this allows for easy integration and use with languages like Python and C++. YAML is a versatile language and consists of a wide range of structures, collections and other tools useful for, amongst other things, communication file syntax as seen in ROS. An example of where it is used is in message files.

A.2 Building a development computer

Part	**Part Name**
Graphics card	ASUS Geforce RTX 3070 Ti TUF 8GB Gaming V2
CPU	i7-13700KF
RAM	Fury Beast 16GB DDR5 4800Mhz
Power Supply	Corsair RM1000e 1000W
SSD	WD Green SN350 480Gb
Tower	Kolink Observatory Y
CPU cooler	MasterLiquid ML240L V2
Motherboard	ASUS Prime Z790-P Wi-Fi DDR5 ATX

Table A.1: The Components of the Computer

When the computer was built Windows 10 was installed and a dual boot was made by partitioning the SSD and installing Ubuntu 22.04 on the new partition.

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

CHALMERS