

# Autonomous Pick-and-Place Procedure with an Industrial Robot Using Multiple 3D Sensors for Object Detection and Obstacle Avoidance

Sindre Bokneberg

Even Falkenberg Langås

## **Supervisors**

Ajit Jha

Martin Marie Hubert Choux

## **University of Agder, 2021**

Faculty of Engineering and Science

Department of Engineering and Sciences

# Acknowledgements

We would like to thank Atle Aalerud for his work leading up to this project, and for his brilliant technical support in the beginning, helping us take good decisions at an early stage. He and his collaborates has created a great learning platform for students as well as a great demonstration platform to inspire the industry to close the gap to research on robotics and computer vision.

Further on, we would like to thank Ajit Jha and Martin Marie Hubert Choux at the University of Agder for advisory and support throughout the entire project. A special thanks to Ajit for his drive for us to think more academically, providing us with more pride and research in this project.

The lab engineers at the university met us with a helpful and positive mindset when enquired about procuring mechanical and electrical components, and we would therefore like to thank them for their help through the production phase of the project.



# Abstract

This thesis proposes a full pipeline autonomous pick-and-place procedure, integrating perception, planning, grasping and control for execution of tasks towards long term industrial automation. Within perception, we demonstrate the detection of a large object (target) including position and orientation (pose) estimation in 3D world. Further on, obstacles in the work area are mapped with proposed filtering prior to motion planning and navigation of an industrial robot to the target's pose. The target is then picked using a custom built motorized 3D printed end gripper, and placed at a desired location in the robot's reachable environment. Point cloud based model-free obstacle avoidance is performed throughout the whole process. The complete pipeline is targeted towards typical tasks in various industries including offshore, logistics and warehouse domain with scanning of the scene, picking and placing of a bulky object from one position to another without or with minimal human intervention.

The proposed methodology was tested upon the point cloud representation of the scene using a network of six RGB-D cameras covering the entire working environment. The empirical results together with the statistical analysis show that the proposed methodology is able to map the environment of volume 10 m x 10 m x 5 m with lesser noise and determine the target position of length 1.2 m with accuracy of 4.8 mm and precision of 3.6 mm from 10000 measurements.

Integrating the proposed object detection and localization, obstacle mapping and gripper with an industrial robot resulted in a consistent, versatile and autonomous pick-and-place procedure. 30 successive tests with multiple obstacles and with the target object placed vertically, horizontally and angled, displayed no collisions and 100% success rate on both gripping and placement of the target.

The entire code developed in the project can be found on [Github](#) including links to CAD-files of the gripper. A video demonstrating the complete pick-and-place procedure can be seen [here](#) or in the URL below. The executable source code can also be found in Appendix E.

Github: [github.com/evenfl/p26\\_master](https://github.com/evenfl/p26_master)

Video: [youtu.be/1QShpxbUy2Q](https://youtu.be/1QShpxbUy2Q)

# Contents

<b>Acknowledgements</b>	<b>i</b>
<b>Abstract</b>	<b>ii</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Background . . . . .	2
1.2 Objective . . . . .	2
1.3 Project overview . . . . .	3
1.3.1 Report structure . . . . .	3
1.3.2 Project management . . . . .	4
<b>2 State-of-the-art</b>	<b>5</b>
2.1 Perception . . . . .	5
2.2 Point cloud processing . . . . .	6
2.2.1 Environment mapping . . . . .	7
2.2.2 Segmentation and model fitting . . . . .	8
2.3 Autonomous pick-and-place . . . . .	8
<b>3 Experimental setup</b>	<b>10</b>
3.1 Hardware setup . . . . .	10
3.2 Software setup . . . . .	12
3.2.1 Point Cloud Library . . . . .	12
3.2.2 Robot Operating System . . . . .	12
3.2.3 MoveIt . . . . .	13
<b>4 Perception</b>	<b>15</b>
4.1 Mapping the environment . . . . .	15
4.2 Object detection and localization . . . . .	16
4.2.1 RANSAC . . . . .	16
4.2.2 Segmentation . . . . .	17
4.2.3 Extracting the object's pose . . . . .	19
4.2.4 Results . . . . .	20
4.2.5 Adding the cylinder to MoveIt . . . . .	30
4.2.6 Real world validation . . . . .	31
4.3 Obstacle mapping . . . . .	31
4.3.1 Denoising . . . . .	32
4.3.2 Results . . . . .	33
4.3.3 Creating an occupancy map with MoveIt . . . . .	34
<b>5 Gripper development</b>	<b>36</b>
5.1 Mechanical design . . . . .	36
5.1.1 Concepts . . . . .	36

5.1.2	Soft robotics gripper . . . . .	40
5.1.3	Concept evaluation . . . . .	41
5.1.4	Gripper overall architecture . . . . .	41
5.1.5	Base . . . . .	43
5.1.6	Arms . . . . .	45
5.1.7	Synchronous gears . . . . .	45
5.1.8	Position sensor . . . . .	46
5.1.9	Complete assembly . . . . .	46
5.2	Hardware setup . . . . .	47
5.2.1	Controller . . . . .	47
5.2.2	Hardware components . . . . .	47
5.2.3	Installation and software setup . . . . .	49
5.3	Control setup . . . . .	53
5.3.1	Control architecture . . . . .	53
5.3.2	Sensor threshold values . . . . .	54
5.3.3	Motor control . . . . .	54
5.4	Complete prototype . . . . .	55
5.4.1	Mounting onto industrial robot . . . . .	55
5.4.2	Gripped target . . . . .	55
<b>6</b>	<b>Autonomous pick-and-place</b>	<b>58</b>
6.1	Building the robot model . . . . .	58
6.1.1	Unified Robot Description Format . . . . .	58
6.1.2	Semantic Robot Description Format . . . . .	59
6.1.3	Base link and end-effector link . . . . .	59
6.1.4	Collision detection . . . . .	59
6.1.5	Inverse kinematics solver . . . . .	60
6.2	Navigation and locomotion of the robot . . . . .	61
6.2.1	Grasping . . . . .	61
6.2.2	Orientation . . . . .	61
6.2.3	Motion planning . . . . .	64
6.2.4	Moving the robot with MoveIt . . . . .	64
6.3	System integration . . . . .	65
6.3.1	Pick-and-place functional description . . . . .	65
6.3.2	ROS nodes and topics . . . . .	66
6.4	Results . . . . .	67
6.4.1	Gripper . . . . .	67
6.4.2	Pick-and-place . . . . .	68
<b>7</b>	<b>Discussions</b>	<b>70</b>
7.1	Object detection and localization . . . . .	70
7.2	Obstacle mapping . . . . .	71
7.3	Gripper . . . . .	73
7.4	Motion Planning . . . . .	73
<b>8</b>	<b>Conclusions</b>	<b>75</b>
	<b>Bibliography</b>	<b>76</b>

<b>A</b>	<b>MoveIt Setup Assistant</b>	<b>80</b>
A.1	Self-collision checking . . . . .	80
<b>B</b>	<b>ABB IRB4400 Datasheet</b>	<b>81</b>
<b>C</b>	<b>DC Motor w/Gearing</b>	<b>84</b>
<b>D</b>	<b>DC Motor Driver VNH2SP30</b>	<b>86</b>
<b>E</b>	<b>Source code</b>	<b>88</b>
E.1	Object detection and localization . . . . .	88
E.1.1	📄 main.h . . . . .	88
E.1.2	📄 addCylinder.h . . . . .	89
E.1.3	📄 addCylinder.cpp . . . . .	90
E.1.4	📄 segment.h . . . . .	91
E.1.5	📄 segment.cpp . . . . .	91
E.1.6	📄 main.cpp . . . . .	95
E.2	Obstacle mapping . . . . .	101
E.2.1	📄 sensor_kinect_pointcloud.yaml . . . . .	101
E.2.2	📄 sensor_manager.launch . . . . .	102
E.2.3	📄 main.h . . . . .	102
E.2.4	📄 main.cpp . . . . .	104
E.3	Pick-and-place . . . . .	107
E.3.1	📄 p26_move.py . . . . .	107
E.4	Gripper . . . . .	117
E.4.1	📄 actuation.py . . . . .	117
E.4.2	📄 gripping.py . . . . .	120

# Chapter 1

## Introduction

Autonomous robotics are a growing industry with a wide range of applications. They are becoming a necessity for companies to assert themselves in a competitive market. Many industries are applying industrial robots to replace manual work or to solve new challenges inducing lower costs due to improved efficiency and reduced lead time. However, automating industrial applications can come at the cost of safety for humans and expensive equipment if used without caution. Through the fourth industrial revolution (industry 4.0), research have introduced new methods for safer use of robotics with collision handling and human-machine interaction.

Health, safety and the environment (HSE) is a hot topic now that the technology has come far enough to be able to automate processes where machines can replace humans working in harsh conditions. In the oil and gas industry, the easily accessible oil reservoirs are already explored, and the industry is exploring reservoirs in less accessible areas such as in the Barents Sea. Here, winter temperatures can range between  $-20^{\circ}\text{C}$  and  $-30^{\circ}\text{C}$  [1]. This makes it desirable to be able to use industrial robots and being able to control and monitor the operations from onshore control rooms. The same concept applies to smelters where humans work in extremely high temperatures. These examples are just a few of many cases where humans work in harsh environments, and where it would be desirable to use industrial robots.

Equinor is a company that owns and operates several oil rigs, where some of them are able to operate unmanned due to the high amount of automation. One of these rigs is Valemon [2], which initially had a crew of 40. Now, the rig has only a few operators controlling the machinery from an onshore control room where they can monitor the rig using 147 cameras. When Valemon started operating with only onshore operators in 2018, the offshore crew of 40 was moved to onshore positions within the company, according to Nina Koch, the production director at the time [3]. This is an example where automation led to cost reduction at the platform as well as new opportunities for the crew and the company.

In manned environments, such as in the logistics and warehouse industry, there are many repetitive and heavy operations that can be burdensome for humans to perform over a long period of time. Picking and placing heavy objects can in many cases cause deepening of the worker's spine load problem, which can be a burden for life [4]. By using industrial robots to carry out these repetitive tasks, the workers are spared for strain, and companies can save cost through improved efficiency and reduced lead time.

To increase the safety of robotic applications in a manned environment, many robots are caged to keep a fixed, predictable environment without putting human workers in danger. This is a very inefficient approach in tight spaces and it limits the potential of multiple machines and humans collaborating. HSE is crucial when placing a large machine outside a cage in a harsh environment alongside humans and unpredictable obstacles. A focus area in the industry today is to use computer

vision to detect obstacles such as workers and equipment. State-of-the-art 3D sensors, such as the Microsoft Kinect V2 used in this project, generates large datasets of the environment in 3D. It is desirable to use these datasets to create position constraints for the robot to avoid colliding with obstacles [5]. Obstacle avoidance can increase efficiency because the robots can be able to operate simultaneously and alongside humans and other machinery without colliding.

This thesis demonstrates an autonomous pick-and-place procedure which can be applicable for multiple industries such as offshore, warehouse and logistics domain. Making the procedure autonomous required both object detection and localization, and obstacle mapping through the use of 3D perception algorithms. This is an important part of Industry 4.0 making machinery autonomous. Detection algorithms can use 2D- or 3D sensors which are getting cheaper and cheaper as years go by. Research and technology regarding computer vision is evolving at a rapid pace and it is used for a wide range of applications, not only within robotics.

## 1.1 Background

This project was proposed by National Oilwell Varco (NOV). The department located in Kristiansand, Norway, is currently working on research concerning robotics and computer vision.

The thesis builds on previous work done at a robotics lab at the University of Agder, see [5], [6] and [7]. These papers describes how the robotics lab is 3D-mapped using 6 3D sensor nodes as well as how these sensors are calibrated. They also propose a method for compressing and filtering the data locally with one embedded system for each sensor node. This thesis will use the experimental setup, calibration and filtration method proposed in these three papers as a base. From this, the thesis will propose the software and gripper needed to lift a large object and place it at a goal position with an industrial ABB robot without colliding with any obstacles surrounding it.

## 1.2 Objective

That leads us to the main objective of this thesis, which is to *develop an autonomous pick-and-place procedure*. This should be done by using an industrial robot to carry out the pick-and-place procedure of an object while avoiding static obstacles in the reachable environment. Both the object and the obstacles are to be detected and localized by 6 sensor nodes covering the entire operational environment. To solve this problem, the following tasks in prioritized order has to be performed:

1. Mapping the environment in 3D
2. Object detection and localization
3. Obstacle mapping
4. Design, production and implementation of automatic low-cost gripper
5. Navigation and locomotion of robot
6. Use point 1.-5. to perform the autonomous procedure in Figure 1.1 with obstacle avoidance

### Limitations

The object to be detected, localized, picked and placed is limited to a large but light object with a length of more than 1 m. Obstacle mapping should be model-free, meaning that the robot, also with the cylindrical target object attached, should be able to avoid obstacles of any size or shape in

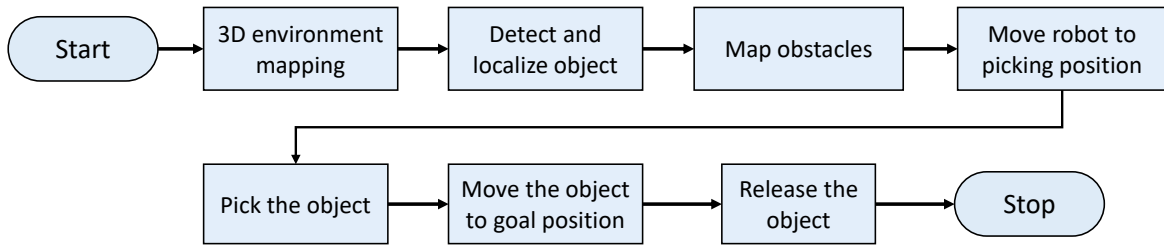


Figure 1.1: *Objective*

the work area. The gripper is not required to be model-free, and can therefore be developed based on the target object chosen. This implies that the gripper itself will be automatic, and it should work in the sequence of a complete pick-and-place procedure.

### 1.3 Project overview

Figure 1.2 shows the *Work Breakdown Structure* (WBS) of the project and how the project is split up in different tasks which are reflected in the report structure.

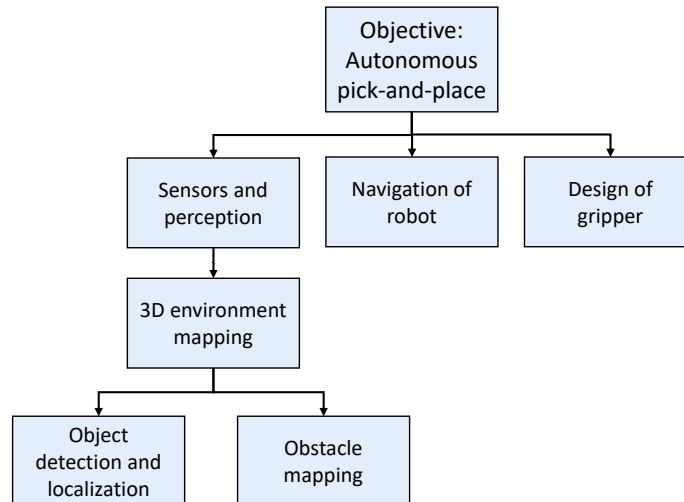


Figure 1.2: *Work breakdown structure of the project*

#### 1.3.1 Report structure

This project is built up by a wide range of subjects, and the report is built up to separate them in a tidy way. This section will explain the chapters shortly to give an understanding on how the report is structured.

Chapter 2: *State-of-the-art* summarizes state-of-the-art research on the topics relevant for the methods used in this project. Next, Chapter 3: *Experimental setup* explains how the system is set up with regards to both hardware and software. This chapter will also give an explanation on the most important software tools used. Chapter 4: *Perception* explains the methods and results for the subjects related to perception. The main topics here are object detection and localization and obstacle mapping. Further on, Chapter 5: *Gripper development* will explain the design and production of the gripper from the conceptual phase to a complete prototype. The integration of Chapter 4 and 5 into one autonomous pick-and-place procedure will then be explained in Chapter 6: *Autonomous pick-and-place*. This chapter includes the navigation and locomotion of the robot as well as system

integration to make everything work together as one system. At last, it presents the final results from the pick-and-place procedure. Chapter 7: *Discussions* presents discussions on relevant subjects and on the final result. The final chapter, Chapter 8: *Conclusions* presents the conclusions on the project in total.

### 1.3.2 Project management

A Gantt chart was made to get an overview of the expected progress in the project, the chart were then updated continuously to keep track on the progress, see Figure 1.3 for the Gantt chart of the final progress. Such a way of planning the project made it easier to keep track of the progress and staying within schedule.

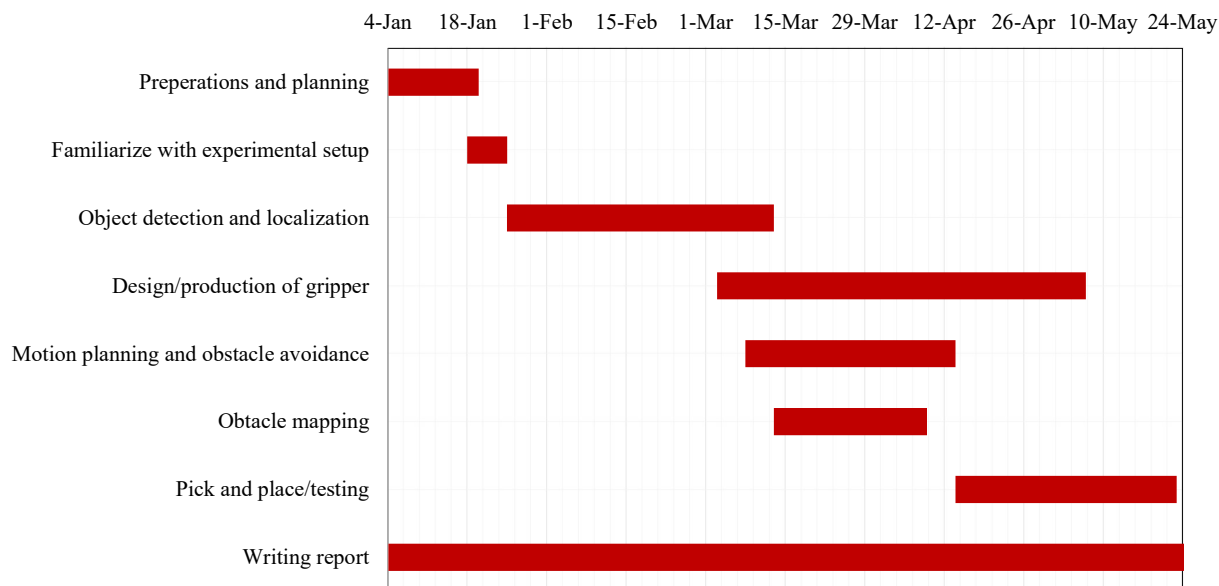


Figure 1.3: *Gantt chart showing final progress*

In addition to the Gantt chart, a Trello board was used to implement the Scrum method. Scrum is a method used in project planning to keep track of the progress in detail, while the Trello board is a tool to implement this method. It is often used in software development because it is an effective way of collaborating with other team members. In a large team, where a project manager manages the progress, it can be beneficial to have lists for suggested tasks, approved tasks, tasks in review, etc. Since there were only two students working on this project it was found satisfactory to use three lists:

- To do
- In progress
- Completed

In addition to the lists, the following labels where used to categorize the tasks:

- On hold
- Priority
- Urgent
- Report
- Code

The software code developed in this project utilized git to enable the group to work simultaneously and with backup and version control. Git allowed each team member to edit their own version of the source code locally before merging it together. A repository was made by using a DevOps platform (GitHub). This repository could then be cloned to any machine by users with access to the Git repository. One of the members was made merge master and had the responsibility to merge the different versions on the development branches without damaging the master branch. The GitHub repository can be found [here](#).



# Chapter 2

## State-of-the-art

Making a pick-and-place procedure autonomous is a comprehensive task. Gathering and using previous research on the topic is crucial to make it work within the frames of a master thesis. First and foremost, it requires advanced perception algorithms based on years of rapid research and development. The goal of this chapter is to present relevant research on which this thesis builds upon. It is divided into three sections, a hardware related section containing 3D sensors and perception, a software related covering point cloud processing and a section covering autonomous pick-and-place.

### 2.1 Perception

Perception is a crucial part of robotics for making processes autonomous. Its definition from [8]:

*Perception is the organization, identification, and interpretation of sensory information in order to represent and understand the presented information or environment.*

For computational systems, perception can be obtained through various sensors. Within the field of intelligent robotics, cameras are becoming the preferred choice for most applications due to the rapid development of computer vision algorithms as the cost of computational power is reduced. Traditional 2D cameras are used frequently for object detection and automatic incident detection (AID) [9]. However, 2D cameras will in many application fall behind because of the lack of depth information. Depth information can be estimated in 2D if the size of the perceived objects is known [10]. This is not always the case, and for collision avoidance application with unknown obstacles, the raw depth information from the sensors can be crucial [11]. This is where the advantage of 3D sensors, such as Light Detection And Ranging (Lidar) sensors and RGB-D cameras, plays an important part.

Lidars use light pulses to estimate the distance to a point in 3D space using the time-of-flight principle. They are often built up by several lasers placed on a rotating gimbal or by the emerging solid state lidar technology with many static lasers. They are a popular choice for many robotic applications such as UAV and automotive [12]. However, their small field of view (FOV) limits their use [13], e.g. in a warehouses where the sensors often must be able to work at a short range, thus requiring a larger FOV.

The development of RGB-D cameras is gaining lots of interest because of their information richness [14] and scalability [7]. Compared to lidars, they generally have a shorter range but a larger FOV. This makes them suitable for applications where a large FOV is desired and range is unimportant. RGB-D cameras can be used for many purposes, and even cheap RGB-D cameras, such as the Kinect V2 can accomplish great results [6]. However, when using cheap equipment, [14] faced issues with noise affection in motion fusion due to precision issues when using Kinect cameras in

dynamic environments. RGB-D cameras such as the Kinect V2 utilizes active infra-red (IR) sensors to measure the depth using the time-of-flight principle [5]. This means that the sensor has one IR emitter and one IR sensor. It emits an IR pulse from the emitter, and calculates the distance to an object by measuring the time it takes before the IR sensor receives the pulse.

Perception systems together with industrial robot cells could be implemented as eye-in-hand [15], eye-to-hand [5] and [16] also covered in this project, and a hybrid-combination [17] with both eye-in-hand and eye-to-hand. The eye-in-hand method often establishes a narrow sight from the robots perspective, providing a closer look at the point of interest, thus providing a higher resolution and accuracy. And eye-to-hand providing wide coverage, enabling features like obstacle mapping and hazardous detection around the robot. [16] shows the use of eye-to-hand perception, reducing system maintenance on sensors inside a hot environment with nuclear fuel pellets, moving the perception system outside the hot cell.

The projection of a single camera eye-in-hand sensor implies that it cannot look behind objects, inducing a shadowed area. This shadow can be removed by utilizing multiple sensors spread around the environment, covering all sides of the objects. In addition, several sensors nodes leads to better resolution [5]. This makes it very beneficial for any industrial application with a variable environment in a fixed area like robots moving in an environment with intervention from humans and other equipment. A 3D sensor based virtual environment can be used for object detection [18], obstacle mapping [19], human detection [20] and more, with loads of new research coming each year.

## 2.2 Point cloud processing

Point clouds are a product of the information obtained from 3D sensors. A point cloud is a set of data points in 3D space. Each point has x-, y-, and z-coordinates, but they can also hold information about color and intensity. Intensity represents the energy reflected by an object. For example, a retroflective surface will have a very high intensity [7]. For points caused by dust in the air or objects with either a dark colour or a large distance from the camera, the intensity will be low. Just like digital 2D images, a point cloud is limited to a given resolution. The resolution is not fixed in 3D space, and a nonuniform distribution of points can occur. A higher density of points in an area implies a higher computational cost for processing of that area. Voxels are therefore often used to set a uniform resolution in 3D space. A voxel represents a value on a regular grid in 3D space. Point clouds are often compressed by organizing them in a voxel grid because it gives the advantage of choosing the exact resolution of the grid [21], the same way the resolution in a 2D image is determined by the number of pixels.

Point cloud processing can be a computationally expensive task [22]. Kaldestad, Hovland and Anisi [23] shows how fast obstacle detection can be done through the octree method [24] and how it greatly reduces the computational cost of point cloud processing. Octree is a method that organizes data in a tree data structure where every node has exactly 8 children. It is often used to partition 3D space and can be closely related to point cloud processing. It divides space into octants and suboctants as shown in Figure 2.1 [24]. While a raw point cloud is just a list of unstructured points, the octree method is structured by nature. The advantage of the method lies within computational power. An octree can remove lots of unnecessary computations in areas without any data and it is expected to have time complexity  $O(\log N)$ .

[14] developed a multi-granularity environment perception algorithm that puts data from multiple Kinect sensors in an octree occupancy grid to create a uniformly distributed representation of

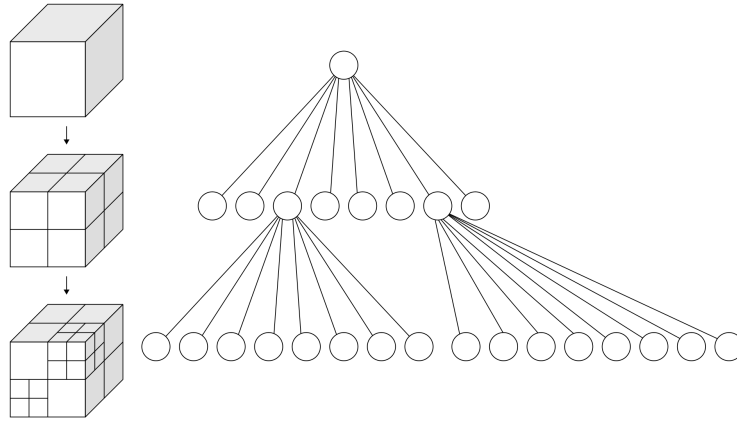


Figure 2.1: *Subdivision of a 3D cube into octants. On the right is the corresponding octree [25].*

the environment. They point out the scalability issue with computational cost growing cubically when the resolution or the space expands. By performing point cloud processing locally in an embedded system, the perception system could be scaled more easily by adding more sensors with local filtering algorithms. However, most embedded systems struggles with point cloud processing due to the computational cost [14]. Nevertheless, Dybedal, Aalerud and Hovland proposed in [7] a scalable embedded solution that compresses the point clouds locally at a Jetson TX2 development board using an octree based method. Further on, they use a novel method to generate intensity values which are not natively produced by the Kinect cameras. They then filter based on this intensity. The intensity filter removes many spurious points by setting a threshold for the intensity value of each point. The filter is very efficient, but for some applications the accuracy might not be sufficient.

As most sensors, lidars and 3D cameras induces noise. The noise appearing in the point cloud leads to outliers that corrupt the result. Outliers, by definition, is data points with a great distance from the main distribution of other points [26]. This can be a critical problem depending on the use case. Fortunately, there are methods to deal with such measurement errors. A commonly used method is a statistical outlier removal (SOR) filter. What this filter does, is calculating the distance from a point to its neighbours. Then, it removes the points, which do not meet a certain criterion. The filter used in this project assumes a Gaussian distribution of data points including a mean and a standard deviation. If a point's mean distance is outside an interval based on the global mean and standard deviation, it is considered an outlier and is therefore trimmed from the initial dataset [27, 26]. The SOR filter is more comprehensive compared to the intensity filter. However, its computational cost is much higher, but its reliability and accuracy makes it the preferred choice for many applications [7, 26].

### 2.2.1 Environment mapping

A way of mapping the environment with multiple 3D sensors was developed and implemented in a robot lab at the University of Agder by Aalerud et al. in [5], [6] and [7]. [5] covers the setup, placement and manual calibration of 6 Microsoft Kinect RGB-D sensors, which together achieved a mapping area of 10x15x5 m with an accuracy of 10 mm or better and a frame rate of 20 Hz. A problem they faced was the time it took to manually calibrate each sensor, approximately 2 hours per sensor node. Therefore, they proposed a solution of automatic calibration in [6] which led to an Euclidean error of 3 cm at distances up to 9.45 m from the sensors. They also decreased the area to 10x10x5 m to increase the point cloud density. The automatic calibration method proposed in [6]

had a more reliable and robust verification compared to the manual calibration [5], and therefore, Aalerud et al. chose to go on with the automatic calibration for further work.

Within robotics, such a point cloud can be used for obstacle avoidance by utilizing occupancy grid mapping. This method was first proposed by Moravec and Elfes in 1985 [28] where they used wide range sonars to create a map of an office. It remained nearly untouched by the industry until the 21<sup>st</sup> century, and in recent years, the use of this technology has expanded rapidly due to the decreased cost of computing power and sensors. In addition to being applied to industrial robot applications, occupancy grid mapping is even emerging at a consumer level through the commercialization of autonomous vehicles and even robot vacuum cleaners [29] because of the obstacle avoidance features the technology provides. Occupancy grid mapping are in many cases closely related to the octree method. Hornung et al. have created an open-source framework for 3D mapping called OctoMap [30]. It uses probabilistic occupancy estimations together with octrees to create a memory compact, multi-resolution 3D map.

### 2.2.2 Segmentation and model fitting

Segmentation is the task of specifying and labeling different regions within an image or point cloud. By labeling each single point and determining the corresponding class, e.g. plane or cylinder, semantic segmentation is performed.

Model fitting is the idea of matching predefined models with a data set. This is often used for finding primitive geometrical shapes such as cube, plane and cylinder. When dealing with images or point clouds, model fitting can be considered a segmentation approach. The most common model fitting algorithms are Hough Transform (HT) and Random Sample Consensus (RANSAC) [22].

RANSAC is a widely used and mature paradigm that fits models to experimental data, e.g. point clouds [31]. Two main advantages of RANSAC compared to HT is that RANSAC is superior within efficiency and success rate [32]. In addition, the RANSAC method is a very robust algorithm, even with noisy point clouds with many outliers [33]. This makes RANSAC a very suitable choice for cheap 3D sensors such as Microsoft Kinect. Sveier et al. shows in [18] how RANSAC can be used to detect primitive shapes, such as planes, spheres and cylinders, even with lots of outliers.

## 2.3 Autonomous pick-and-place

Pick-and-place procedures are a common sight within many industrial environments today, and it is desired to further exploit possibilities of making the procedures autonomous. With an autonomous procedure, perception and environmental mapping is often utilized, making the system more adaptable to a dynamically changing environment, also making the system more flexible and easier to set up in a new environment [34]. Perception could also provide distinct object localization algorithms that detects what and where to pick [35]. Kotthaus et al. demonstrates in [16] how an autonomous pick-and-place procedure can be used, carried out inside a hot environment with pick-and-place of nuclear fuel pellets, moving the perception systems outside the hot cell.

In recent studies, the use of collaborative robots are substantial in autonomous pick-and-place studies, as utilized in both [36] and [34]. These collaborative robots work well for their intended use, but lacks strength and durability against harsh environments often encountered in industrial applications. [37] states that the use of such robots is mainly used for smaller objects in a cleaner environment. The target of combining hard robotics with system environment awareness is not necessarily to obtain a fully secure system, but to make a base capable of cooperating with both

people and other systems, and also be independent from human and manual control. This could also be more valuable for industries already using hard industrial robots, being able to maintain its established selection of hardware while implementing a less hazardous environment.

# Chapter 3

## Experimental setup

The experimental setup is divided into two sections: *Hardware setup* and *Software setup*. The hardware setup will explain all the hardware used in the project and briefly dive into the relevant specifications. The software setup will explain what software tools was used and how the high level system architecture looks.

### 3.1 Hardware setup

All testing was performed in a lab where the hardware was stationed. A rack computer, an industrial robot and six 3D sensor nodes existed in the lab before the start of this project. The 3D sensor nodes were strategically placed around the lab, covering the working environment, and calibrated by Aalerud, Dybedal and Hovland in [5] and [6] to get an optimal 3D map of the robot's working range. Figure 3.1 shows the experimental setup including the 6 sensor nodes and two industrial robots, one of them (the left one in the figure) being used in this project. Figure 3.2 shows each Kinect V2 low cost cameras view of the environment, together covering the whole area. Each camera produces a point cloud representation of the environment used for object and obstacle detection.

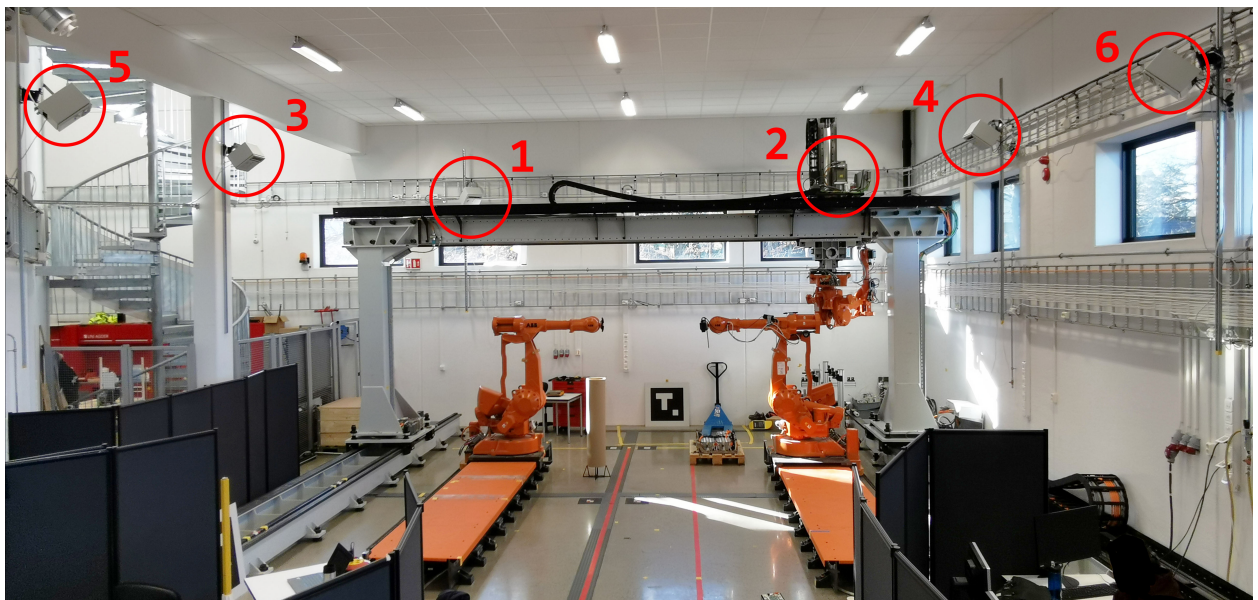


Figure 3.1: *Overview of the experimental setup*



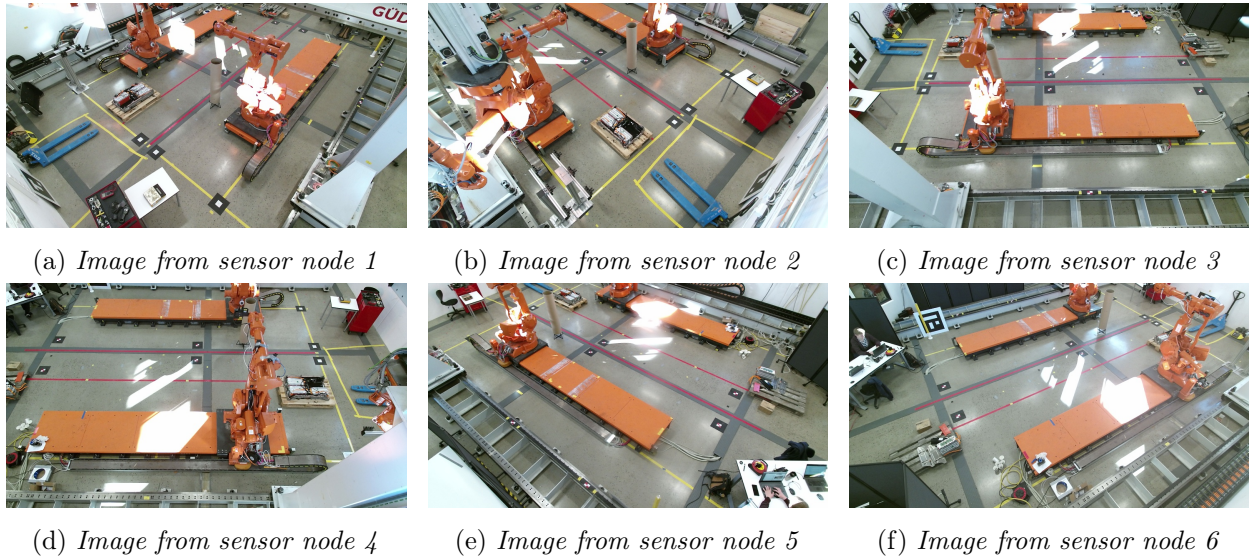
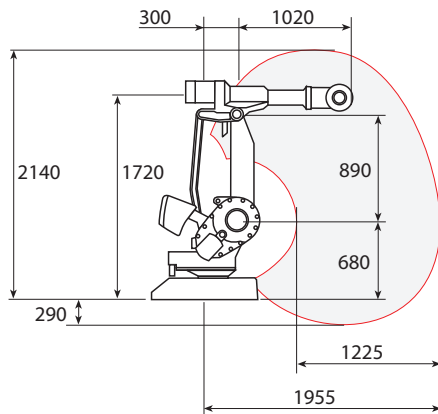


Figure 3.2: Images from all kinect cameras

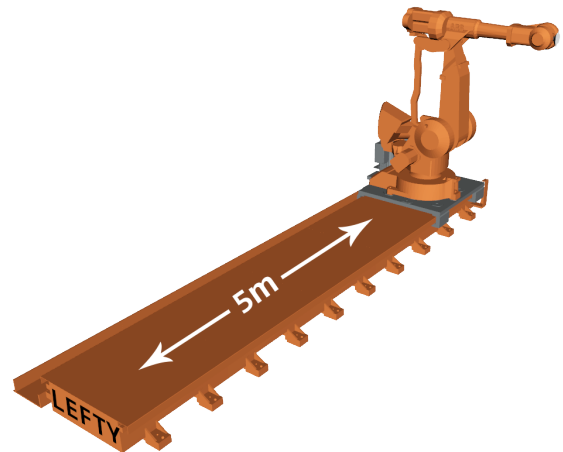
Every Kinect camera was connected to a NVIDIA Jetson TX2 development board which filtered the point clouds and published the filtered sensor data to the ROS environment to each ROS topic, explained in more detail in chapter 4. Every node was connected through ethernet to the rack computer. Both camera and board is shown in Figure 3.3.



Figure 3.3: The Kinect V2 and the NVIDIA Jetson TX2. Image of Kinect from Wikimedia Commons 2014, by courtesy of Evan-Amos. Image of Jetson TX2 from NVIDIA



(a) ABB IRB4400 dimensions and working range [Appendix B]



(b) ABB robot with track

Figure 3.4: Total working range of ABB IRB4400 with 5 m track (Lefty)

The robot used in this project was an ABB IRB 4400 industrial robot. This is a 6 degrees of freedom (DOF) robot capable of lifting 60 kg at high speeds [Appendix B]. The working range can be seen in Figure 3.4a. Further on, it can be seen in Figure 3.4b, that the ABB robot is placed on a 5 m long track which gives it one extra DOF making it a redundant robot. This means that the robot

has more DOF than needed to achieve a goal pose for the end-effector. The fact that it is redundant implies that it can achieve the goal pose in different ways, which might be necessary in an obstacle avoidance application. The robot used in this project goes by the name of Lefty, frequently used in the software.

As for the target object to be detected, localized and picked, a cylindrical shape was chosen. This primitive shape is very common on an offshore platform and in warehouses. It leads to some interesting and challenging grasping issues to be solved where the solutions can be applicable for many other use cases. To be able to detect the cylinder with the Kinect cameras in the lab, a relatively large casting pipe was used to compensate for the resolution. The hollow cylinder had an outer diameter of 26 cm and a length of 1.2 m.

## 3.2 Software setup

To be able to utilize both point clouds, segmentation, robot control and overall system knowledge a variety of software tools was used. Based on the already built operational environment with Kinect sensors and the industrial ABB robot, Robot Operating System (ROS) and MoveIt was desirable candidates for use as overall control architectures because of their compatibility within robotics. In addition, Point Cloud Library (PCL) was utilized to handle point cloud processing. To get an overall understanding as to how the system is built up and of the functionality of the system, it is desirable to have some general knowledge to these software tools. This section will briefly explain PCL, and how ROS and MoveIt were used to make the hardware components work together as one system.

### 3.2.1 Point Cloud Library

PCL is an open-source 3D processing library, especially designed for 3D geometry and point cloud processing:

*From an algorithmic perspective, PCL is meant to incorporate a multitude of 3D processing algorithms that operate on point cloud data, including: filtering, feature estimation, surface reconstruction, model fitting, segmentation, registration, etc. [38, p. 1]*

PCL is fully integrated within the ROS environment and C++.

### 3.2.2 Robot Operating System

Robot Operating System is a framework for writing robot software. It is a free and open source system that contains tools and libraries that makes it easier to develop software for different robotic disciplines [39]. ROS has a huge range of applications, and this section will only cover the parts used in this project.

ROS is built up by several ROS processes communicating through a peer-to-peer network. A ROS master holds the names and registration services and provides them to the different nodes. On the other hand, a ROS node is the process that performs calculations. A robot control system can contain many nodes that performs different computations in different coding languages such as C++ and Python. In this projects, C++ was used for perception algorithms because of the comprehensive support PCL provides for C++, while Python was used for robot control and gripping. There can for example be one node controlling the motion of the robot, one node detecting and localizing an object, one node planning the path and one node visualizing data from the system [39].



The nodes can talk to each other through topics. When one node publishes to a topic another node can subscribe to it. In this way, messages can be sent and received. Multiple nodes may publish/subscribe to one specific topic, and one node may publish/subscribe to multiple topics. ROS compares the topic to a message bus, where the bus has a name, and any node can connect to this bus to send or receive messages as long as it uses the correct data type [39].

Publishing and subscribing to ROS topics can be done through pre-made commands. The publisher publishes messages to a specific topic while the subscriber subscribes to a topic, and passes the message to a custom callback function. Further on, the callback function decides what to do with the information received [40].

## ROS setup

All components communicated with each other through a ROS network. Figure 3.5 shows how the components was physically connected using ethernet, and within each component the software packages used out of ROS, MoveIt and PCL. The ROS master was running on the rack computer, connected to the Jetson TX2 boards, the robot and the gripper and they communicated through SSH connection. The ROS master used all three software packages, the robot controller and the gripper only used ROS and the Jetson boards used ROS for communication and PCL for point cloud processing. An advanced explanation of the setup will be explained in chapter 6, having a better knowledge of the perception algorithms, gripper development and the robot navigation.

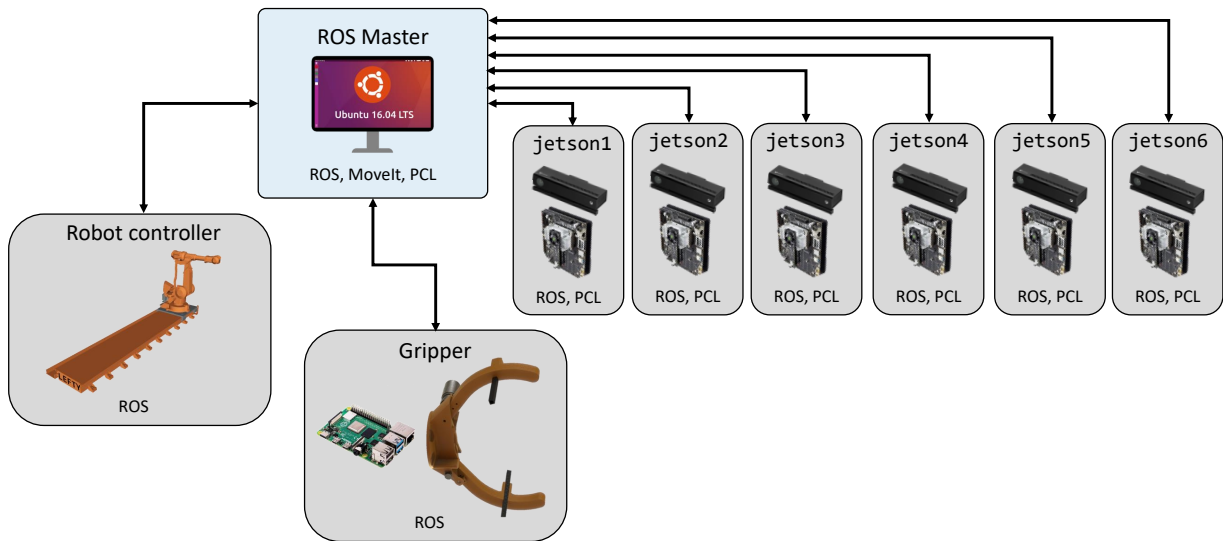


Figure 3.5: *ROS network*

### 3.2.3 MoveIt

MoveIt is an open-source state-of-the-art software, integrated within the ROS environment for safe and efficient use of robots within an operational environment [41]. It was used to connect the dots between object detection, obstacle mapping and navigation of the robot by combining the cylindrical object and the other obstacles into one planning scene which was used to do collision-free path planning. MoveIt consist of multiple software packages, each designed to cope with different challenges regarding robotics, including obstacle avoidance, surrounding awareness and efficient locomotion. This section will explain the parts of the MoveIt package which was used in this project.

For manipulation of robots MoveIt works as a path planning tool that interpolates movement between keypoints, giving waypoints. Keypoints are coordinates for the tip of the robot to "touch" given by e.g. algorithms, while waypoints are iterative keypoints making up the path between keypoints. MoveIt solves every inverse kinematic equations for the robots joints based on the Cartesian space, giving the corresponding joint space. Joint space consists of parameters of every single joint defining the translational and rotational displacements.

The system architecture of MoveIt is visualized in Figure 3.6. The figure shows how MoveIt handles data in this project. Every box is marked with a color, where the yellow boxes are packages that are part of MoveIt's ROS package while the blue boxes are part of the MoveIt core package. Lastly, the grey boxes represents external package dependencies [42].

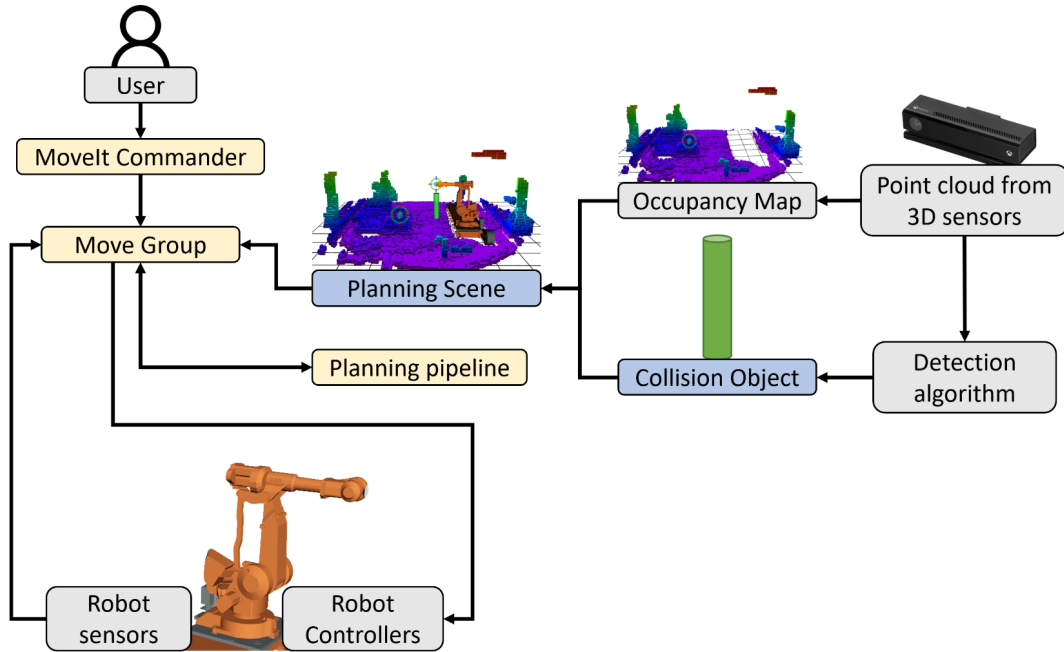


Figure 3.6: *MoveIt system architecture*

MoveIt has a node called `move_group`, which can be seen in the system architecture. This node serves as an integrator which pulls together all the individual components to provide the user with certain ROS services and actions. After all, `move_group` is a ROS node and it uses the ROS param server to access three kinds of information: URDF, SRDF and MoveIt configuration [42]. URDF is short for Unified Robot Description Format which is a way of representing a robot model in XML format. SRDF (Semantic Robot Description Format) on the other hand, is a representation of semantic information about robots. Finally, the MoveIt configuration includes information such as kinematics, joint limits, perception, motion planning, etc. [42]. These will be explained further in chapter 6.1 on building the robot model in MoveIt.

Planning scenes are used in MoveIt to represent the robots current state and the surroundings such as collision objects and other 3D objects. MoveIt has a great advantage when it comes to picking objects with a robot: The possibility to attach objects to the robot model [42].

The perception algorithms used PCL, ROS and MoveIt, and the next chapter will explain the methods used to develop algorithms to perform both object detection and localization and obstacle mapping.

# Chapter 4

## Perception

For a robotic application to become autonomous, it needs a perception of its environment and it needs to understand how to solve the task at hand. To be able to perform object detection and localization as well as model-free obstacle mapping, 3D sensors are used as explained in chapter 3. This chapter will explain three subjects related to perception: *Mapping the environment*, *Object detection and localization*, and *Obstacle mapping*, all used to enable an autonomous pick-and-place procedure. The source code for the perception can be seen Appendices E.1 and E.2

### 4.1 Mapping the environment

To be able to combine multiple sensor nodes to map the environment, transformations must be performed from each sensor node's local coordinate system to one common global coordinate system. This was done through calibration by Atle Aalerud et al. [6]. They placed ArUco markers around the lab and created an automatic calibration algorithm. The raw point clouds from the Kinect cameras was being processed and compressed to a voxel grid with a resolution of 4 cm because the accuracy achieved in [6] was 3 cm. Fusing the point clouds on the other hand, had not been done.

Based on the work explained above, the scope is expanded as summarized in the flow chart in Figure 4.1, which explains the flow of the code made to fuse the data from the sensor nodes. The orange area represents the work already performed by Aalerud et al. [5, 6, 7]. The compressed point clouds from the Jetson boards are available on 6 different ROS topics. The point cloud on each topic is filtered locally before being published, utilizing an intensity filter to remove some spurious points. This filter removes any point with an intensity value lower than a given threshold value of 1-255, where 255 represents maximum intensity. When subscribing to the ROS topics, the data type is `sensor_msgs::PointCloud2`, while the datatype required by the PCL functions is `pcl::PointCloud<pcl::PointXYZ>`. To convert the ROS message to the correct datatype, the `pcl::fromROSMsg()` function is used. The implementation is available in Appendix E.2.

After the conversion, the point clouds from the 6 sensors could be merged. With the PCL library, this is done by adding the point clouds together. The point clouds are saved to a variable called `cloud_merged`. A counter is used to count how many messages are received. Since all sensor nodes publish with the same frame rate, the different point clouds would come in sequence so that when the counter reached a total of 6, the algorithm has received messages from every sensor node. The algorithm summarized in Figure 4.1 is used inside the callback functions for the subscribers subscribing to the sensor nodes. The output of this algorithm is `cloud_merged`, which is a 3D mapped point cloud of the whole work area. This is used for object detection and localization and obstacle mapping.

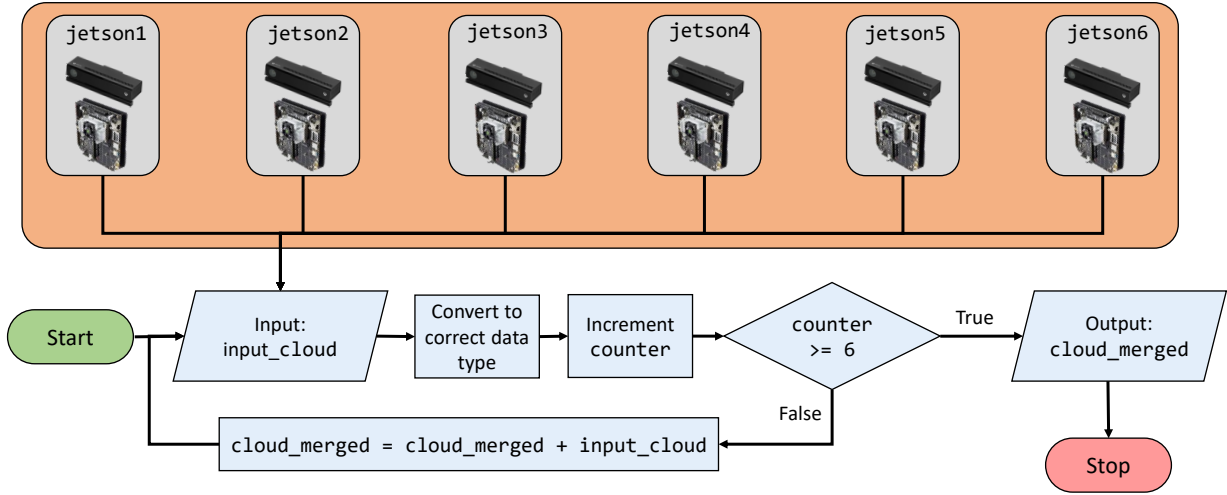


Figure 4.1: Flow chart for the point cloud merging

## 4.2 Object detection and localization

Within the reachable environment of the industrial robot, a target object (cylinder, explained in section 3.1) was placed to be detected, localized, grasped and then moved. This section will explain the process of establishing and testing a detection and localization algorithm, and in chapter 6, this algorithm will be used to perform a pick-and-place procedure on the cylinder. This depends heavily on a reliable estimation of the targets pose.

The lab was cluttered with different random objects, making it close to an industrial environment towards which the work is dedicated. If one tries to detect a cylindrical object in a 2D image, the cylinder will be close to a rectangle, and therefore many non-cylindrical 2D rectangles could be incorrectly interpreted as cylinders. 3D point clouds could be used to avoid this specific issue. In this way, the curvature of the cylinder could be measured by using the RANSAC method.

### 4.2.1 RANSAC

The cylinder segmentation was based on the RANSAC method which was introduced in chapter 2.2.2. The algorithm is built up by two main phases:

1. Hypothesis generation
2. Hypothesis evaluation / Model verification

Step 1. is done by using random samples to generate a hypothesis, while step 2. is to verify it to the data [43]. In addition, because this is a model fitting algorithm, the model has to be manually defined before step 1.

When generating a hypothesis, RANSAC chooses  $N$  random samples and estimates certain model parameters based on these points. For plane segmentation, 3 non-collinear points are enough to make out a plane, thus  $N=3$ . A plane model can be defined by the following equation [22]:

$$aX + bY + cZ + d = 0 \quad (4.1)$$

where  $[a, b, c, d]^T$  are the parameters to be estimated.

RANSAC uses equation 4.2 to solve the selection problem, which can be considered an optimization problem [43].

$$\hat{M} = \underset{d \in D}{\operatorname{argmin}} \left\{ \sum \operatorname{Loss}(\operatorname{Err}(d; M)) \right\} \quad (4.2)$$

where  $D$  is data,  $\operatorname{Loss}$  is a loss function and  $\operatorname{Err}$  represents an error function, for example geometrical distance errors.

To get the most out of the RANSAC method, some parameters can be tuned to fit the specific use case. The parameters that are tuned in this project are the following:

- Distance threshold: A distance threshold from each inlier point to the model
- Normal distance weight: Weight factor for the surface normals influence
- Radius limits: Limits the maximum radius deviations from the defined model
- Max iterations: How many iterations the algorithm will perform at maximum

#### 4.2.2 Segmentation

The implementation of the proposed algorithm for object detection from 3D point cloud is demonstrated in Figure 4.2. First, the point clouds from all the sensor nodes are subscribed to and merged as proposed in Figure 4.1. Then the point clouds are filtered to remove all points outside the working area which are considered spurious points. The direction vector of the cylinder is then estimated using the RANSAC algorithm followed by a sphere filter used to estimate the center of mass (COM). Filtration and RANSAC are both part of the `segment` function, while the sphere filter is implemented in a function called `passThroughFilterSphere`. These will be explained later in this section. To improve the precision by reducing the stochastic noise, a method of obtaining the result based on the average from multiple frames is proposed. The effect of this method will be presented in the results in section 4.2.4. The program will check if the desired number of frames is achieved. If not, it will repeat the segmentation process and keep on calculating the average of the obtained results. When the desired number of frames is achieved, the target will be created as a 3D object in MoveIt. The source code can be seen in Appendix E.1.

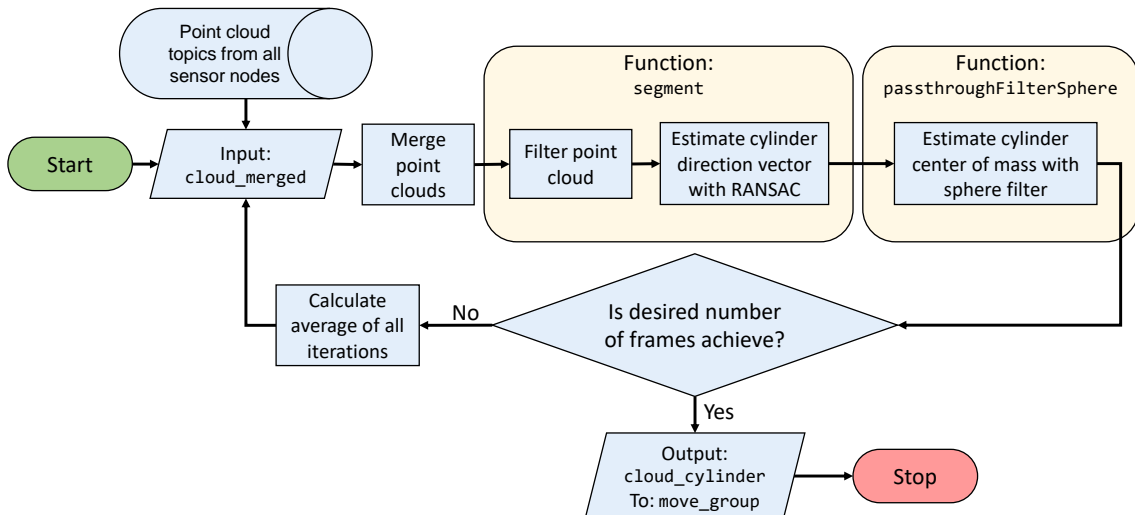


Figure 4.2: Flow chart for the object segmentation

Firstly, the cylinder segmentation algorithm was tested with the point cloud from one of the cameras as a proof of concept. It was observed that the target object was well detected using it. However, one camera is not sufficient to be able to detect the target object in the whole working area (not shown for brevity). All 6 point clouds are therefore merged as explained in section 4.1. This leads to a more versatile and reliable perception of the environment. Using 6 sensor nodes also makes the algorithm less prone to shadowed areas caused by potential obstacles.

Then, the point cloud are filtered using computationally inexpensive methods to reduce the number of points to be fed into the cylinder segmentation, making it computationally efficient. Figure 4.3 shows a flow chart of the `segment` function from Figure 4.2. Firstly, the `segment` function filters out all points outside the work area. This is done by using a passthrough filter that removes all points exceeding a given x-, y-, or z-value. These passthrough filters removed the floor and other unnecessary and spurious points from the sensors. The x-, y-, and z-limits are summarized in Table 4.1, which makes out the detection zone shown in Figure 4.4. Narrowing the detection zone from the initial 10 m x 10 m x 5 m to 1.8 m x 9.0 m x 1.95 m greatly reduced the computational cost of the algorithm while keeping a satisfactory detection zone for the task at hand, covering the entire reachable area. To test the reliability of the algorithm, the testing and tuning is done at the whole 10 m x 10 m x 5 m area to be able to spot possible issues.

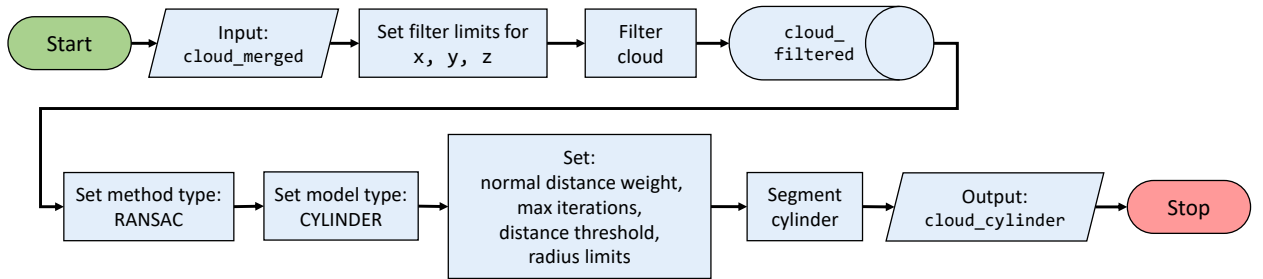


Figure 4.3: *Flow chart of the segmentation algorithm*

Table 4.1: *Coordinate limits for the cylinder segmentation*

	x_min	x_max	y_min	y_max	z_min	z_max
Limit [m]	4.2	6.0	1.0	10.0	0.05	2.0

Finally, when most of the irrelevant points were removed, the cylinder segmentation could start. RANSAC was considered the most suitable approach because of its robust and accurate results for primitive geometrical shapes, even when working with point clouds with noise and outliers as explained in chapter 2.2.2. The RANSAC algorithm had to be tuned to be able to achieve a high accuracy and precision. After tuning the algorithm the values in Table 4.2 was set.

Table 4.2: *RANSAC parameters after tuning*

	Value	Unit
Distance threshold	0.08	m
Normal distance weight	0.2	-
Radius limits	$R_{cylinder} \pm 0.03$	m
Max iterations	10 000	-



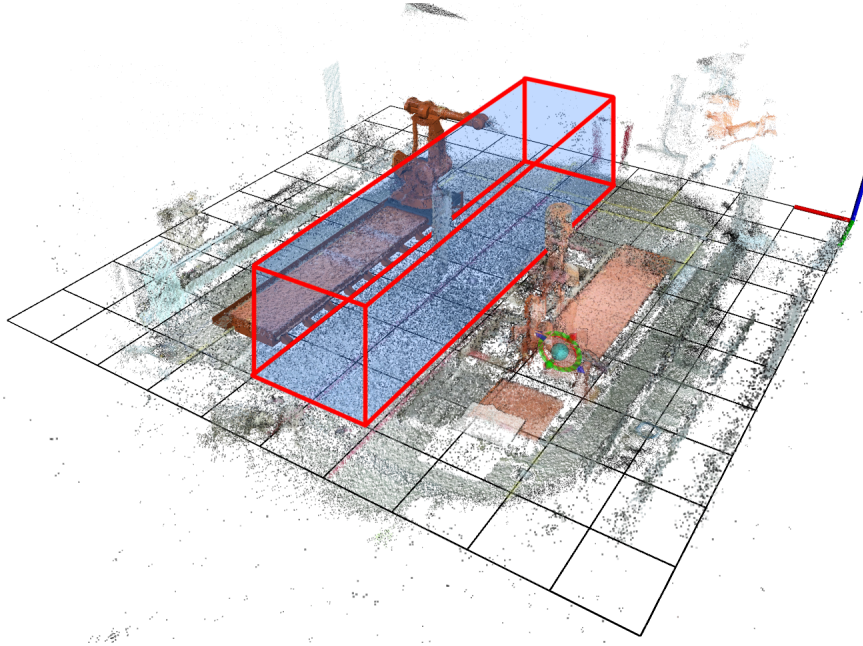


Figure 4.4: *Object detection zone*

### 4.2.3 Extracting the object's pose

To be able to pick up the cylinder, both the position and orientation must be estimated. The RANSAC algorithm is able to detect the cylinder. However, it has no conception of the cylinder length. Because of the lack of length conception, the segmentation included spurious data along the axis of the detected cylinder as shown as noise in Figure 4.5.

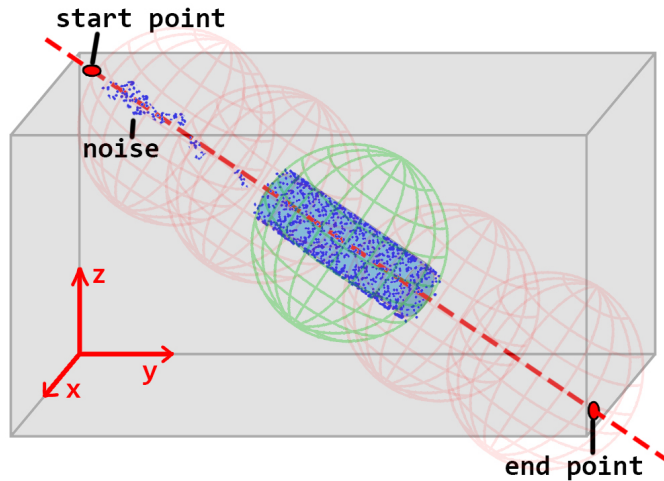


Figure 4.5: *Sphere filtering*

The RANSAC algorithm was tuned to give the direction vector representing the orientation of the cylinder as well as a point on the z-axis of the detected cylinder. The known length of the cylinder could be used to filter out the outlying points shown as noise in Figure 4.5. This is done through a sphere filter with a diameter making it large enough to include the complete cylinder. The sphere radius is calculated using Pythagoras' based on the cylinder length and radius as shown in equation 4.3.

$$R_{sphere} = \sqrt{(L_{cylinder}/2)^2 + R_{cylinder}^2} \quad (4.3)$$

The sphere filter removes all points outside the sphere and keeps all points inside. The sphere filter is then moved along the longitudinal axis of the cylinder. The center point is saved at the point where the sphere picked up the highest number of points from the cylinder point cloud obtained from the `segment` function. The sphere filter is visualized in Figure 4.5. In this figure, the grey rectangle represents the detection zone where the cylinder is known to be. The green sphere represents the sphere that picks up the highest number of points. After sphere filtering, all unwanted points outside the green sphere in Figure 4.5 is removed.

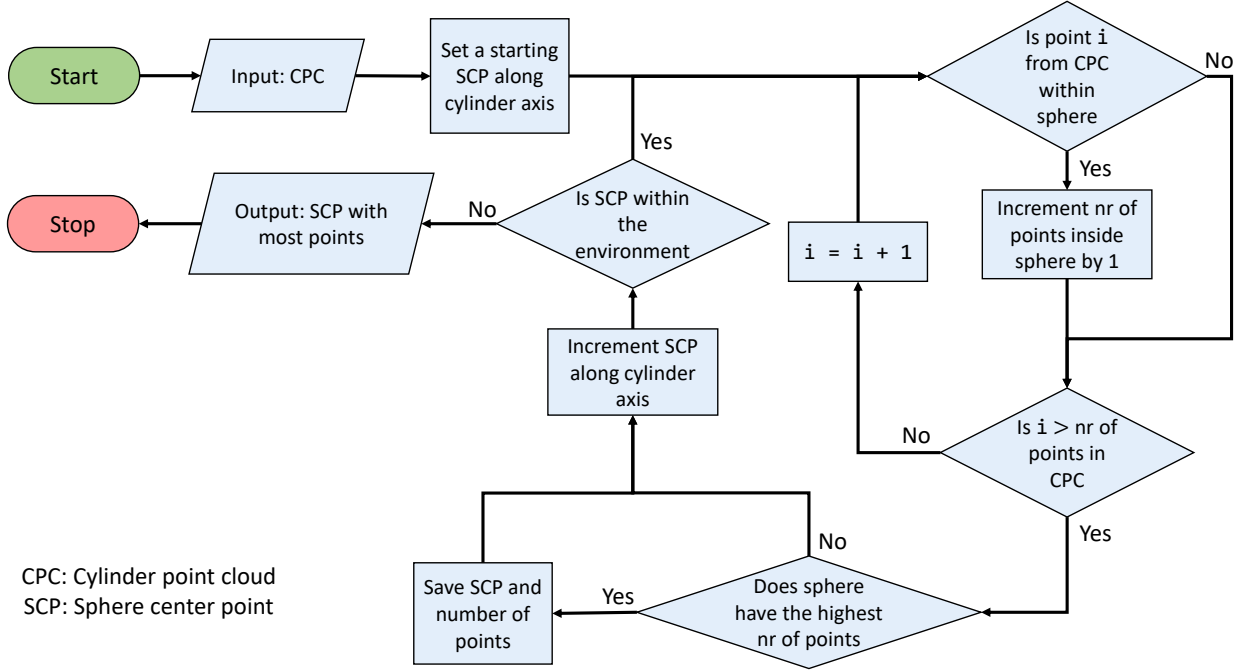


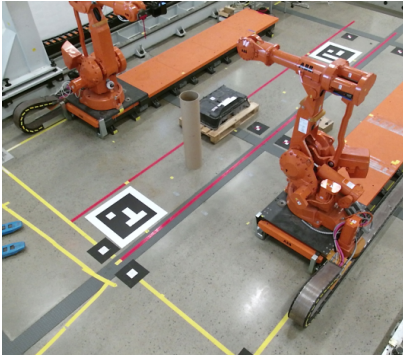
Figure 4.6: Flow chart of the sphere filter algorithm

Figure 4.6 shows the flow chart explaining the implementation of the sphere filter. The function can be seen in Appendix E.1.5. First it took the cylinder point cloud (CPC) as input. Then it set a starting sphere center point (SCP) at a point where the direction vector of the cylinder crossed the edge of the detection zone, see the red dots in Figure 4.5. The algorithm then checks the distance between SCP and the points in the CPC and discards all points outside  $R_{sphere}$ . If a sphere holds the highest number of points, the algorithm will save the SCP and number of points inside this sphere. The algorithm increments the position of the SCP with 1 mm along the cylinder's longitudinal axis. The process is repeated until the sphere has a center point outside the detection zone. Once the loops are finished, the SCP of the sphere with the most points will be considered the COM of the cylinder.

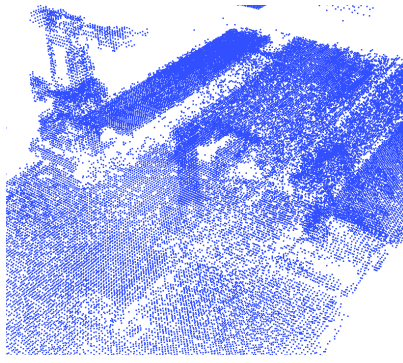
#### 4.2.4 Results

The direction vector from the RANSAC algorithm, COM from the sphere filter and the dimensions of the cylinder gave a fully defined 3D object that could be used for a pick-and-place procedure. However, to be able to grasp the target, the system is depending on high accuracy and precision of the segmentation. Figure 4.7a, 4.7d and 4.7g shows images from sensor node 1, and the cylinder can be seen in the middle of the picture. Further on, the merged point cloud from all sensor nodes is shown in Figure 4.7b, 4.7e and 4.7h. This point cloud was passed through the algorithm in Figure 4.2 and the result can be seen in Figure 4.7c, 4.7f and 4.7i with the detected cylinder marked in red.

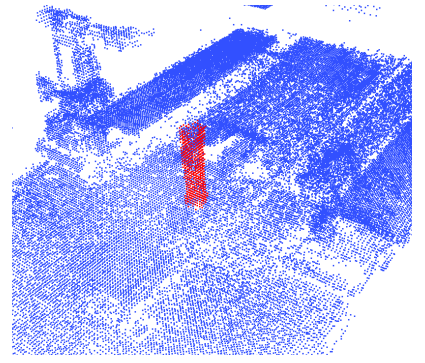




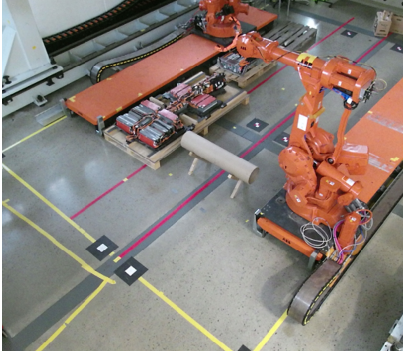
(a) Image from sensor node 1, vertical case



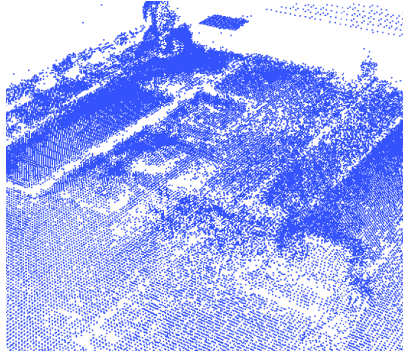
(b) Point cloud from all 6 sensor nodes, vertical case



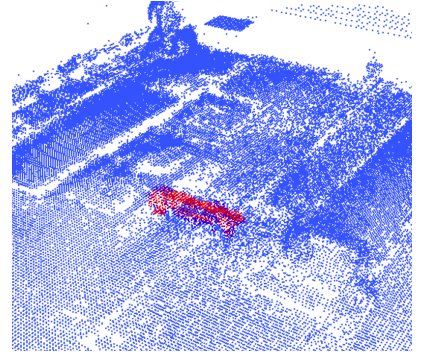
(c) Point cloud with the cylinder marked in red, vertical case



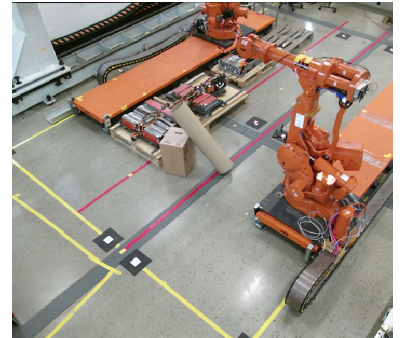
(d) Image from sensor node 1, horizontal case



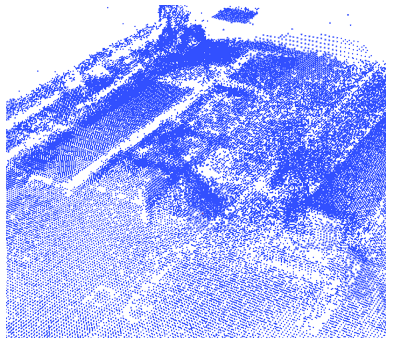
(e) Point cloud from all 6 sensor nodes, horizontal case



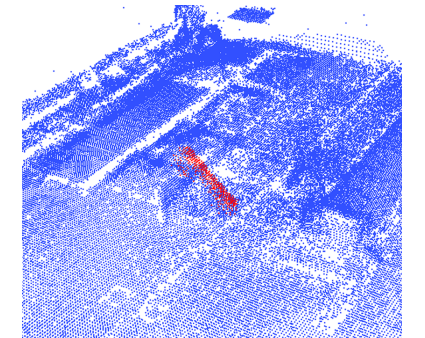
(f) Point cloud with the cylinder marked in red, horizontal case



(g) Image from sensor node 1, general case



(h) Point cloud from all 6 sensor nodes, general case



(i) Point cloud with the cylinder marked in red, general case

Figure 4.7: Results from cylinder segmentation for a vertical case (a,b,c), horizontal case (d,e,f) and general case (g,h,i)

To evaluate the accuracy and precision of the segmentation, the target was placed on a 3D printed base built specifically for this use. It was placed accurately on a reference point that Aalerud, Dybedal and Hovland [6] had measured by using a Leica AT960 high-precision laser tracker. This reference point has a measured distance in x- and y-direction in the global coordinate system and it was considered the ground truth for testing. The z-direction was the height of the base plus half the target length. The frame and the reference point can be seen in Figure 4.8, the ground truth coordinates for the targets COM can be seen in Table 4.3.

Table 4.3: Validation of the reference point

	x [m]	y [m]	z [m]
Ground truth (laser)	5.27510	5.28855	0.75200

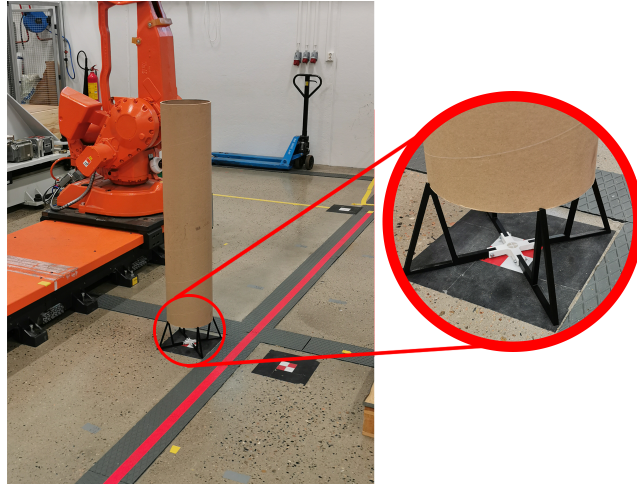


Figure 4.8: Validation of the cylinder segmentation using laser measured point as ground truth

To evaluate the results, 10 000 measurements was obtained using the algorithm in Figure 4.2. Figure 4.9 shows the histogram of the error compared to the ground truth, and it shows how the z-value has two peaks with approximately 2-3 cm between them. This is likely due to the voxel size of 4 cm. The issue is that the number of voxels on the top layer of the object is varying. With the target in a vertical position, the z-axis has the same direction as the cylinder's direction vector.

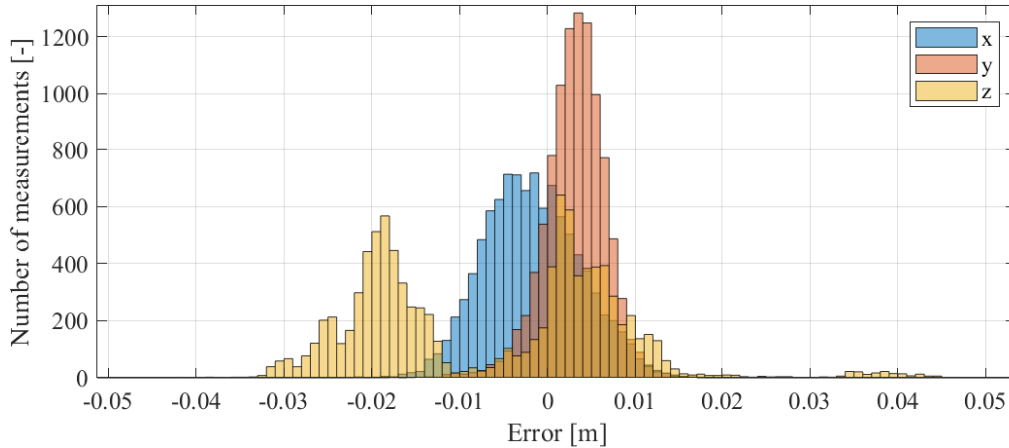


Figure 4.9: Histogram of the errors in x-, y- and z-direction

The accuracy of 3D sensor systems is often given as the mean Euclidean distance between the ground truth and the measured point [6]. This Euclidean distance is calculated for each measurement point using equation 4.4.

$$e = \sqrt{(\hat{x} - x_{ref})^2 + (\hat{y} - y_{ref})^2 + (\hat{z} - z_{ref})^2} = \sqrt{e_x^2 + e_y^2 + e_z^2} \quad (4.4)$$

where,

	Description	Unit
$e$	Euclidean distance between measured point and ground truth	[m]
$e_x, e_y, e_z$	x-,y- and z-error between the measured point and the reference value	[m]
$\hat{x}_t, \hat{y}_t, \hat{z}_t$	Estimated x, y- and z-position	[m]
$x_{ref}, y_{ref}, z_{ref}$	Reference x-, y- and z-position	[m]

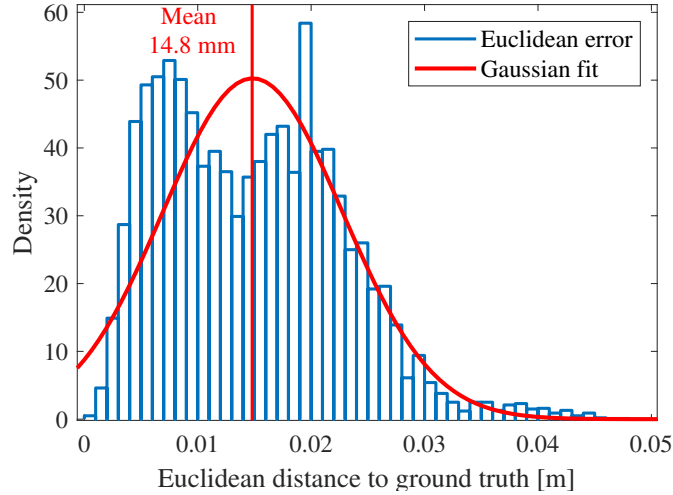


Figure 4.10: *Histogram of the euclidean error between measurement and ground truth*

A histogram of the Euclidean distance between the measured point and the ground truth is shown in Figure 4.10. The graph also shows the Gaussian fit of the histogram, marked with red, obtained using Matlab distribution fitting. The mean,  $\mu$ , of this normal distribution is the accuracy of the system, found to be 14.8 mm. Further on, the precision for COM,  $P_{com}$ , is calculated as shown in equation 4.5

$$P_{com} = 2 \cdot \sigma_{com} = 2 \cdot 8.9mm = 17.8mm \quad (4.5)$$

Where  $\sigma_{com}$  is the standard deviation of the normal distribution found to be 8.9 mm.

The orientation was also important to be able to grasp with the right angle. The angle between the ground truth (vertical) and estimated direction vector was calculated using equation 4.6 [44, p. 5]. This angle is illustrated in Figure 4.11.

$$\theta = \arccos \left( \frac{a_x \cdot b_x + a_y \cdot b_y + a_z \cdot b_z}{\sqrt{a_x^2 + a_y^2 + a_z^2} \cdot \sqrt{b_x^2 + b_y^2 + b_z^2}} \right) \quad (4.6)$$

where,

	Description	Unit
$\theta$	The angle between the two vectors	[°]
$a$	Ground truth direction vector	[m]
$b$	Estimated direction vector	[m]

Figure 4.12 shows the histogram of the angular error illustrated in Figure 4.11. The accuracy is equal to  $\mu$  of the normal distribution in Figure 4.12, found to be  $0.84^\circ$ . The precision,  $P_{angle}$ , is calculated the same way as for the COM, and equation 4.7 shows that the angular precision is  $0.88^\circ$ .

$$P_{angle} = 2 \cdot \sigma_{angle} = 2 \cdot 0.44^\circ = 0.88^\circ \quad (4.7)$$

Both stochastic noise and the fact that z had two peaks in the histogram in Figure 4.9 limited the precision, and the accuracy and precision of 14.8 and 17.8 mm respectively, is not sufficient for a

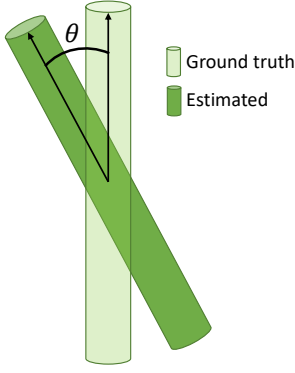


Figure 4.11: *Angle error illustration*

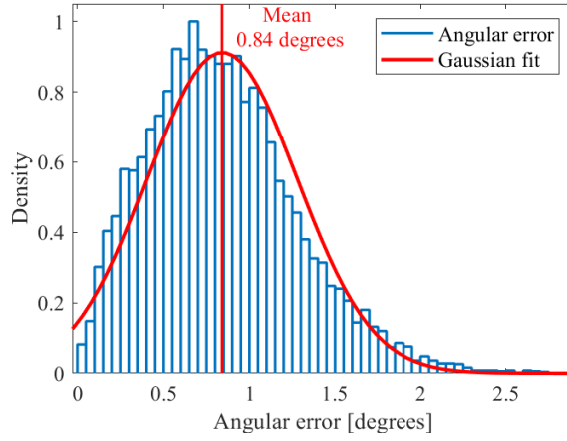
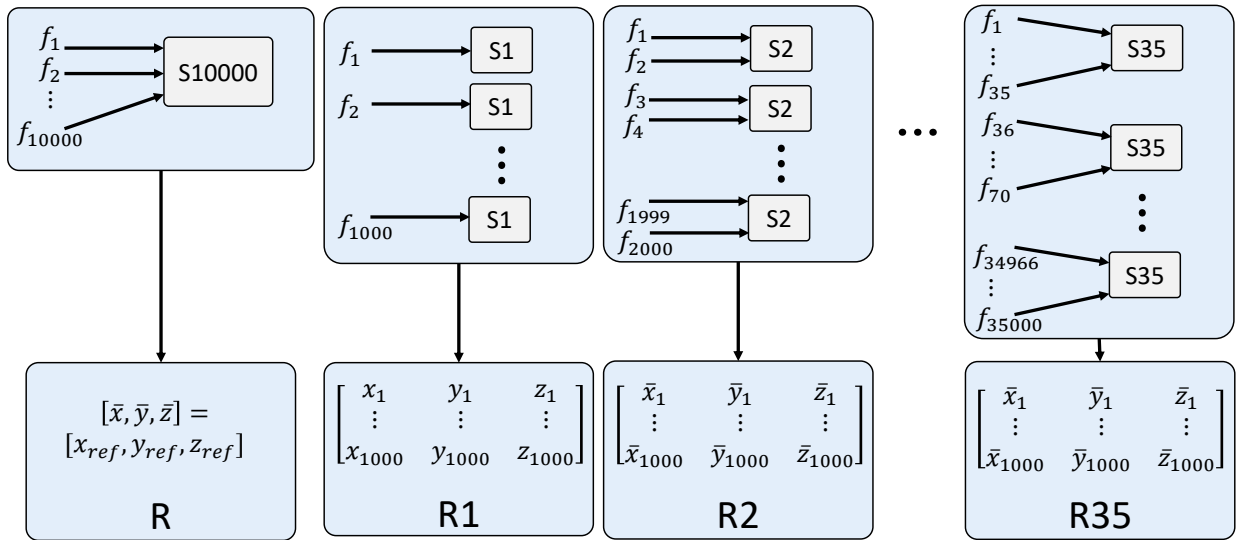


Figure 4.12: *Histogram of the angular error*

sensible gripper. Since the precision is caused by systematic error, it can be improved by increasing the sample size. A method of calculating the average COM and direction vector obtained from multiple point cloud frames,  $f$ , was proposed as a solution to the problem. The hypothesis was that this would lead to lesser outliers, i.e. a better precision.



$f_i$ : Frame number  $i$  containing point cloud data from all sensors nodes  
 $S[nr]$ : Segmentation algorithm from Figure 4.2 with  $[nr]$  as desired number of frames  
 $R[nr]$ : Result from  $S[nr]$

Figure 4.13: *Explanation of testing of object segmentation*

To test this hypothesis, the cylindrical target is placed at a random position in the detection zone. The object is first segmented on 10 000 frames outputting the average COM. This can be seen as R in Figure 4.13, which was considered the most precise achievable result from the system. Therefore, R is used as reference position when evaluating the effect of using multiple frames. Further on, the desired number of frames is set to 1 and the algorithm S1 in Figure 4.13 corresponds to the segmentation algorithm in Figure 4.2 with 1 as the desired number of frames. S1, S2, S3 and so on, is performed 1000 times, giving 1000 measurements of the center of mass, where the results are matrices of size 1000x3. Since every row of the matrix R35 is the mean of the segmentation on 35 frames, it is expected that the result from R35 will have a lower error than R1 when comparing with the reference, R. The reason it stops at 35 is that it is observed that the errors starts to converge



with 20-30 as desired number of frames. These results will be presented later.

The error between the R1-35 and R was calculated by using the root-mean-square error (RMSE) method as shown in equation 4.8.

$$E_{RMS} = \sqrt{\frac{\sum_{t=1}^T e_t^2}{T}} \quad (4.8)$$

where,

	Description	Unit
$E_{RMS}$	The root mean square error	[m]
$T$	Number of measurements	[-]
$e_t$	Euclidean error between reference position and position from measurement $t$	[m]

The results are from now on split into two use cases, vertical and horizontal position. When a suitable number of frames are chosen, the accuracy and precision with the optimal number of frames will be presented as final results for object detection and localization.

### Vertical position

First, the experiment was performed with the cylinder in a vertical position. The RMSE between R1-35 and R is shown in Figure 4.14. It can be seen that increasing the number of frames decreased this error significantly, and at 15-20 frames (R15-R20), it starts to converge.

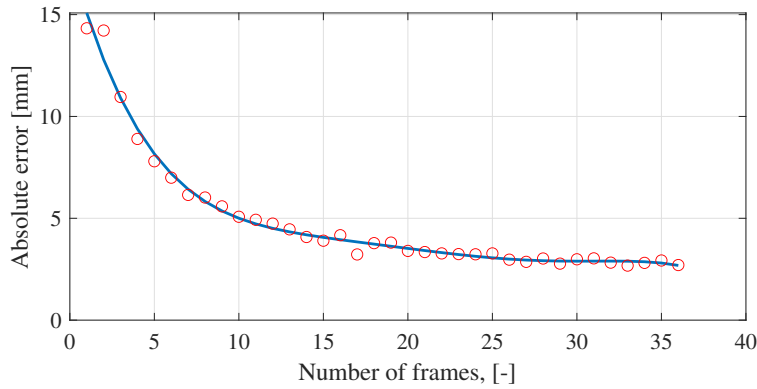


Figure 4.14: *RMSE on COM with the cylinder in a vertical position*

The reduction of outliers when increasing the number of frames was also significant, as shown in Figure 4.15, where the left graph shows the number of points with an Euclidean error larger than 2 cm and the right graph shows the number of points with an Euclidean error larger than 1 cm. By using one frame only, 321 out of 1000 measurements (from R1) had an Euclidean error of 2 cm or more as the left plot in Figure 4.15 shows. If the number of frames are increased, e.g. 20, it can be seen on the right plot in Figure 4.15 that the number of outliers above 1 cm is only 15 (from R20).

The reliability function is a way to show the probability of an object surviving beyond a given parameter value, thus it is also known as the survival function. In engineering, this statistical approach can be used to represent the reliability of a product based on the products lifetime or its accuracy and precision. In this case, the object was the cylinder localization, and the parameter of interest is the measurement error. The reliability function can give the probability of the error being larger than a given size [45, 46].

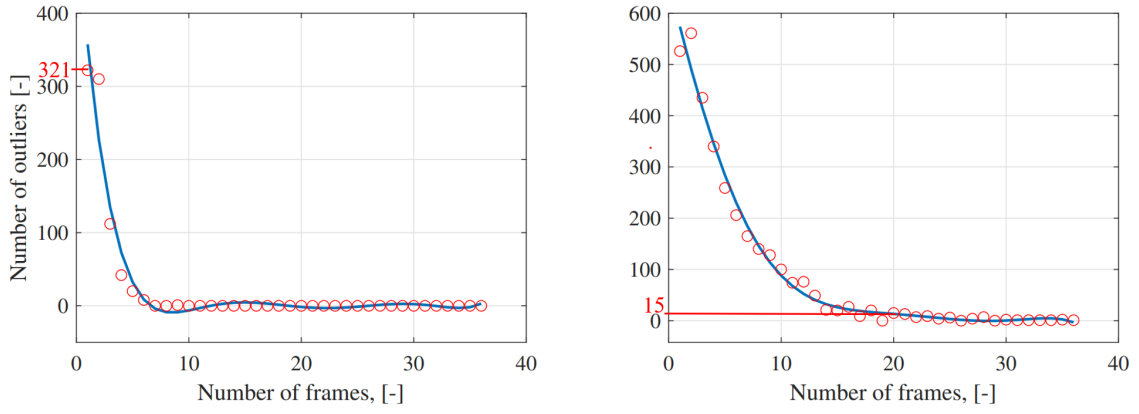


Figure 4.15: *Nr of outliers with the cylinder in a vertical position. The left graph shows outliers outside 2 cm while the right graph shows outliers outside 1 cm*

The reliability function is defined by letting  $E$  be a random continuous variable. It has a cumulative distribution function (CDF)  $F(e)$  on the interval  $[0, \infty)$ . The reliability function is then [46]:

$$S(e) = P(E > e) = \int_e^{\infty} f(u) du = 1 - F(e) \quad (4.9)$$

The reliability function is shown in Figure 4.16. This figure shows the probability of getting an euclidean error larger than a given value with different numbers of frames. It could be observed that the probability of getting a large error drops significantly when taking the average of multiple frames. It seems to converge at around 20 frames. Based on 1000 measurements, the probability of getting an euclidean error above 10 mm is 1.5% from R20. For comparison, it can be seen in the curve that there is the probability of reaching such an error with only 1 frame is 53.2%.

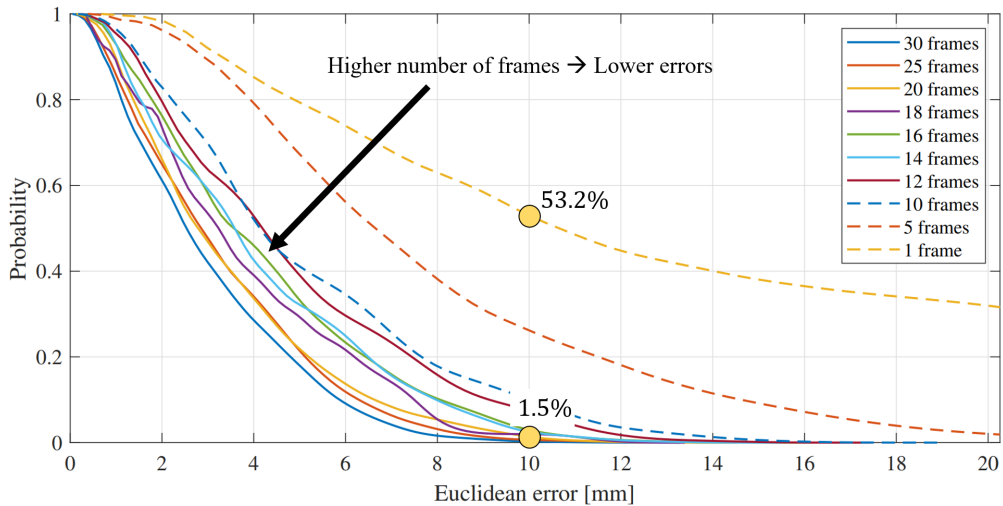


Figure 4.16: *Reliability function with the cylinder in vertical position. It shows the probability of the euclidean error being larger than a given error*

To get a closer look at where the error came from, the error in x-, y-, and z-direction was plotted, and it can be seen in Figure 4.17 that the error was still undoubtedly largest in z-direction both for 1 and 20 frames. Figure 4.17a shows the results from R1 in Figure 4.13 while Figure 4.17b shows the results from R20.

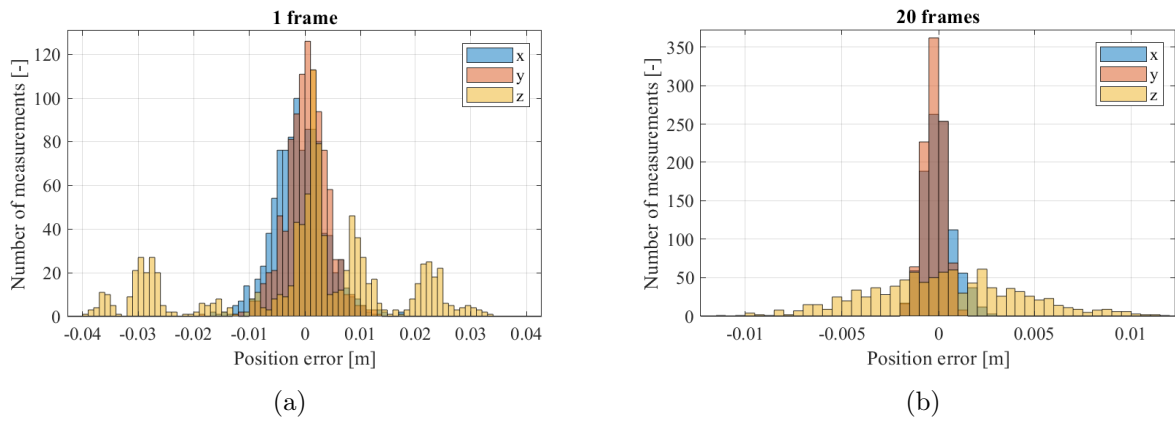


Figure 4.17: Comparison between the error in  $x$ -,  $y$ - and  $z$ -direction for 1 frame and 20 frames with the cylinder in a vertical position along the  $z$ -axis

### Horizontal position

The same approach as for the vertical position was used to estimate the reference position,  $R$ , of the cylinder. Equivalents to the results with the cylinder in a vertical position results are presented in this section for a horizontal position. The horizontal case shows no significant difference in the RMSE compared to the vertical case, see Figure 4.18.

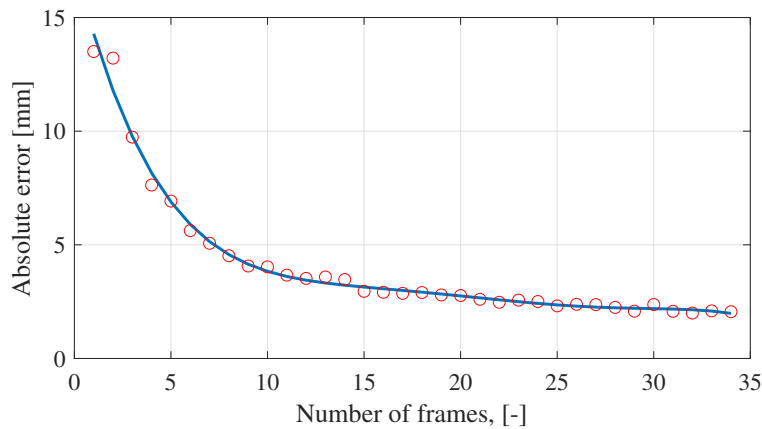


Figure 4.18: Average absolute error on the COM with the cylinder in a horizontal position

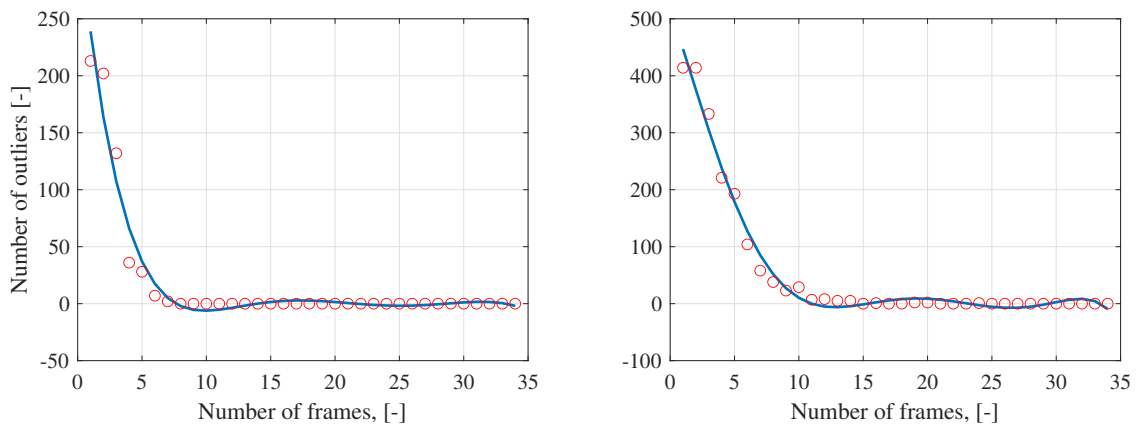


Figure 4.19: Nr. of outliers with the cylinder in a horizontal position. The left graph shows outliers outside 2 cm while the right graph shows outliers outside 1 cm

The number of outliers with regards to Euclidean error shown in Figure 4.19, is however a bit lower for low number of frames compared to the vertical case, but overall, there is no remarkable difference when comparing to Figure 4.15.

The reliability function in Figure 4.20 shows that the probability of getting an error above 10 mm with the target placed horizontally with 20 frames is 0.2% based on the 1000 measurements done (from R20). For comparison, the probability of getting such an error when using only one frame is 41.8%.

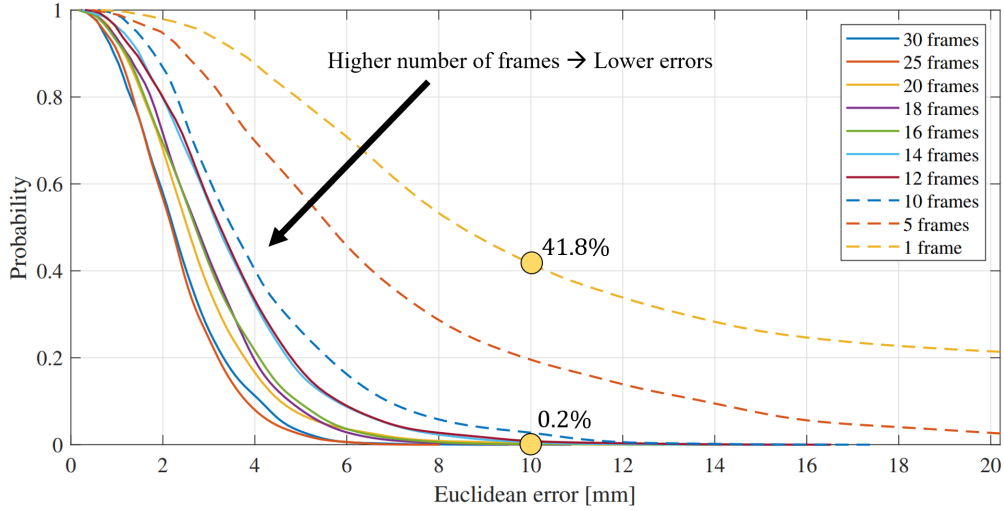


Figure 4.20: Reliability function with the cylinder in a horizontal position

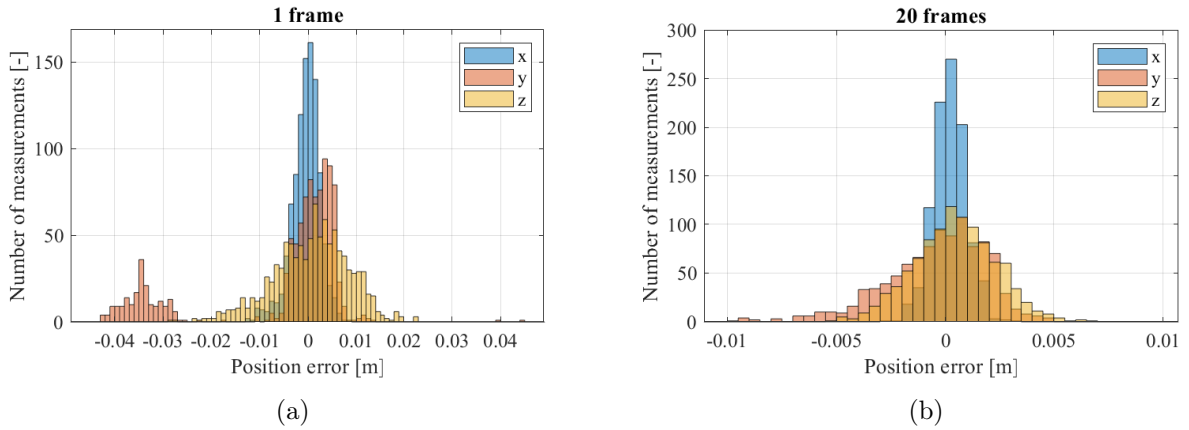


Figure 4.21: Comparison between the error in  $x$ -,  $y$ - and  $z$ -direction for 1 frame and 20 frames with the cylinder in a horizontal position along the  $z$ -axis

When looking at the absolute error in  $x$ -,  $y$ -, and  $z$ -direction with the cylinder with the longitudinal direction along the  $y$ -axis, it was observed that the error was now largest in  $y$ -direction, see Figure 4.21. This was the longitudinal direction, and the two peaks in Figure 4.21a is similar to what was seen in the longitudinal direction in the vertical case. The peaks have a distance of 3-4 cm between them, which corresponds to the voxel size of 4 cm. The two peaks became one when increasing the number of frames as shown in Figure 4.21b.

### Summary of object segmentation results

After evaluating the results from the two use cases, 20 was chosen as the default number of frames which was considered a nice trade off between precision and computational cost. Without any



obstacles in the frame, it segments the target with a frequency of approximately 9 Hz. For 20 frames that means around 2.2 seconds which is satisfactory for this application since the segmentation only happens once before every pick-and-place procedure. However, the computation time increases drastically when introducing obstacles inside the detection zone.

To compare 1 and 20 frames in a visual and intuitive way, they are plotted as points in the xy- and xz-plane. Figure 4.22 shows the result with 1 frame, while equivalently Figure 4.23 shows the same results, but with 20 frames. These figures shows how the precision was improved significantly when increasing the number of frames. The mean COM is placed a bit differently with 1 and 20 frames. Likely due to the fact that the tests were performed on different occasions, and the placement of the target object was therefore slightly different.

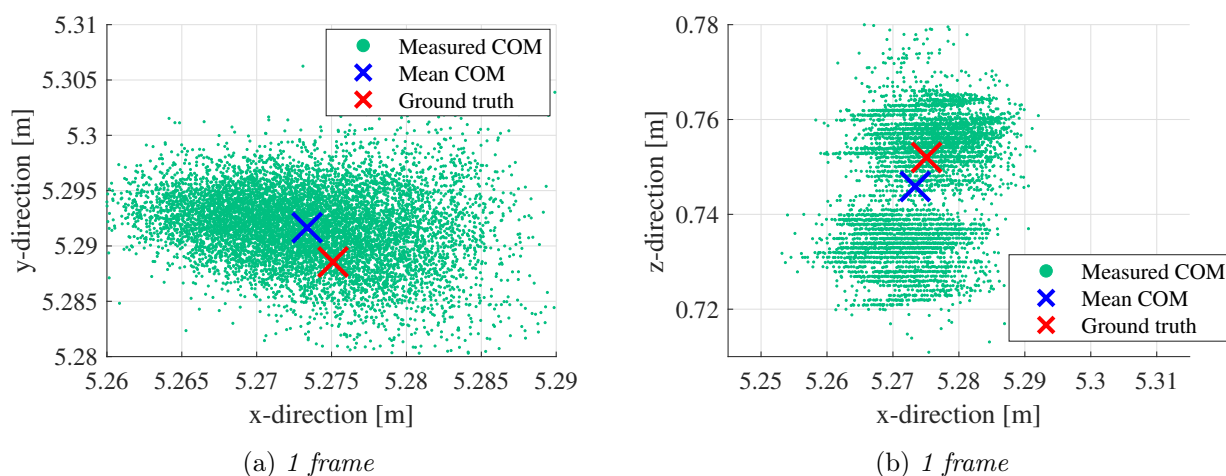


Figure 4.22: 10 000 measurements of the COM with 1 frame. The ground truth is the laser measured reference point

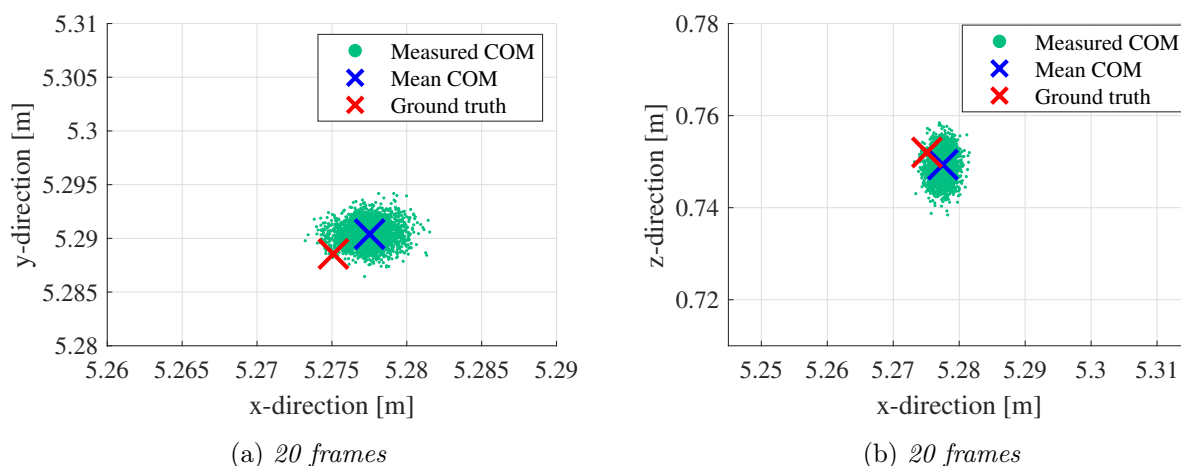
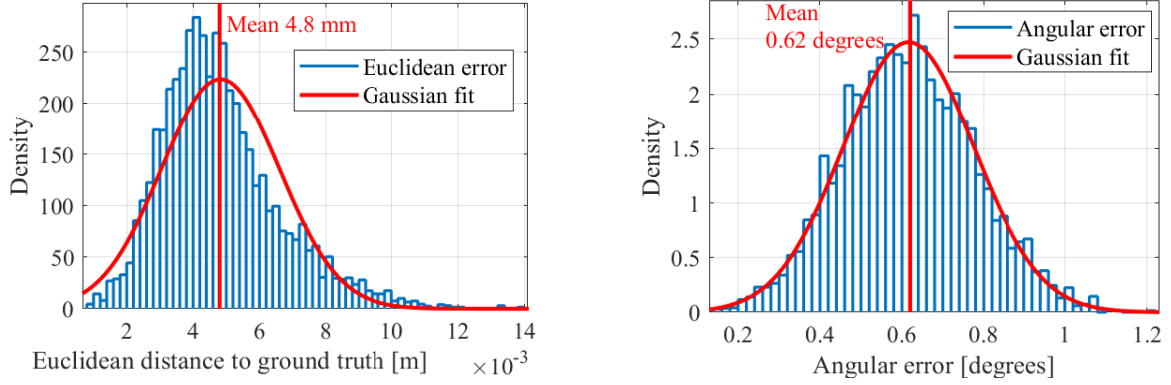


Figure 4.23: 10 000 measurements of the COM with 20 frames. The ground truth is the laser measured reference point

With an increased number of frames, the euclidean distance decreases, and the histogram in Figure 4.24a shows the equivalent to Figure 4.10, but with 20 frames. In addition it shows the Gaussian fit in red. The accuracy of the system is the mean of the normal distribution,  $\mu$ , which is 4.8 mm.



(a) Histogram of the euclidean error between measurement and ground truth

(b) Histogram of the angular error

Figure 4.24: Results with 20 frames

The precision for COM,  $P_{com}$ , on the other hand, is defined in equation 4.10

$$P_{com} = 2 \cdot \sigma_{com} = 2 \cdot 1.8[mm] = 3.6[mm] \quad (4.10)$$

Where  $\sigma_{com}$  is the standard deviation of the normal distribution. It is worth mentioning that the precision in the transverse plane of the object is significantly better than in the longitudinal direction as the histograms for x-, y- and z-position shows (Figure 4.9, 4.17, 4.21). Lastly, the results from orientation is shown in Figure 4.24b where the red line shows the Gaussian fit. The mean of this normal distribution is  $0.62^\circ$  while the angular precision,  $P_{angle}$  is calculated using equation 4.11

$$P_{angle} = 2 \cdot \sigma_{angle} = 2 \cdot 0.16^\circ = 0.32^\circ \quad (4.11)$$

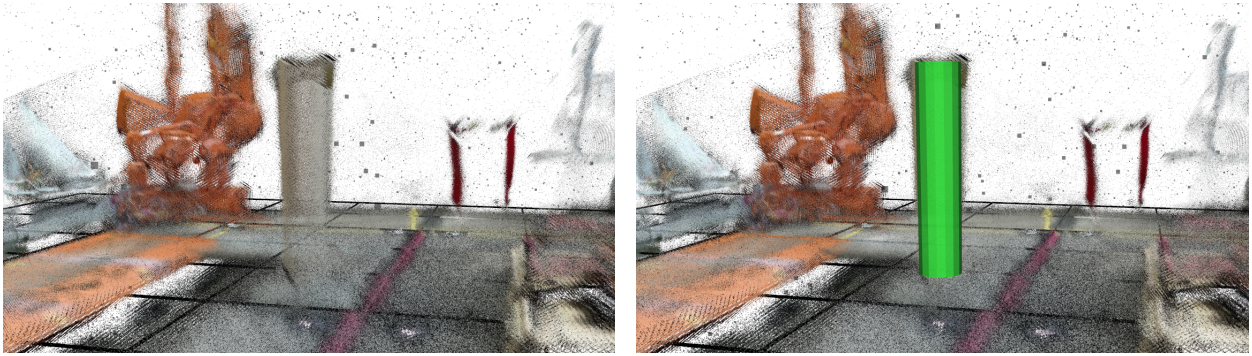
Table 4.4: Accuracy and precision of the system from 10000 measurements comparing 1 and 20 frames

	1 frame	20 frames
Euclidean precision [mm]	17.8	3.6
Euclidean accuracy [mm]	14.8	4.8
Angular precision [°]	0.88	0.32
Angular accuracy [°]	0.84	0.62

#### 4.2.5 Adding the cylinder to MoveIt

When the target's pose is determined, the next step is to add it to MoveIt. This is done through a function called `addCylinder` based on the script from [47]. This function was modified to take cylinder parameters as input and to output a MoveIt compatible collision object. The cylinder parameters are stored in a struct called `AddCylinderParams`. This struct held 4 parameters: radius, direction vector, centre of mass and length. The direction vector and the COM are arrays with 3 elements (x, y and z). Further on, this collision object can be published to the `collision_object` topic. From this topic it is subscribed to by the planning scene [48, p. 7]. Figure 4.25 shows how the cylinder is visualized in RViz. RViz is a 3D visualisation tool for ROS. First, in Figure 4.25a, only

the RGB point cloud is shown while Figure 4.25b shows the point cloud with the cylinder object upon.



(a) *RViz RGB pointclouds with the cylinder in the middle*

(b) *Cylinder object added to RViz*

Figure 4.25: *With and without the cylinder object from MoveIt*

#### 4.2.6 Real world validation

When testing the pick-and-place procedure later in the project, the cylinder segmentation was validated by moving the robot automatically to the target’s pose. Figure 4.26 shows where the robot stopped when told to move its end effector to a distance of 20 cm from the cylinder COM. Since the cylinder has a radius of 13 cm, the expected result should be 7 cm between the robot’s end effector and the cylinder edge. The test shows a deviation of 1 mm in the transverse direction and 10 mm in the longitudinal direction. Similar results were obtained for the horizontal and general use case.

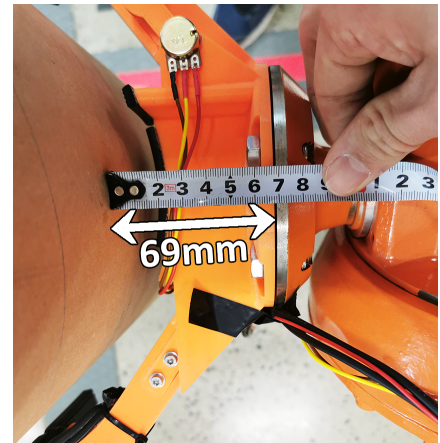


Figure 4.26: *Validation of the cylinder segmentation with the robot*

### 4.3 Obstacle mapping

To be able to perform obstacle avoidance, the system needs a 3D map of the environment to locate obstacles. This can be done by simply modeling the whole work area offline using CAD software before moving the robot. However, this is a very fixed approach resulting in a low adaptability to potential changes. This report presents a method based on online mapping using multiple 3D sensors that creates one point cloud of the whole work area. In this way, the environment can be mapped and obstacles can be detected with a completely model-free approach, making it highly adaptable for changes to the environment. The approach also opens the possibility for real-time collision avoidance.

3D sensor based collision avoidance can be a computationally expensive task [11]. However, there are tools to make this process more efficient when working with point clouds. One method is to map the environment with point clouds using the octree method introduced in chapter 2.2.

Within MoveIt, there is functionality for adding an occupancy map to the robot environment based on the octree method, and in that way it creates a map with a variable resolution based on the localization of datapoints. This functionality is provided through a plugin package called OctoMap. The great advantage of utilizing MoveIt’s functionality for mapping the environment is how it integrates with the ROS system. Because the target object was segmented before the mapping, and the robot model was predefined, the inliers to these objects could be removed. Then, a map could be created with a given resolution resulting in an octree based point cloud of all other unknown objects in the lab. It was then possible to have an accurate model of the objects of interest, in this case the cylinder and the robot, while keeping a low resolution of the obstacles to reduce computational cost and to add a safety margin to the algorithm.

### 4.3.1 Denoising

To create such a map, point clouds from all cameras are merged using the method proposed in chapter 4.1. An issue with using these point clouds directly is noise. Practically, that meant lots of points in thin air as shown in Figure 4.27. This would mean that the robot would struggle with planning around all these individual points.

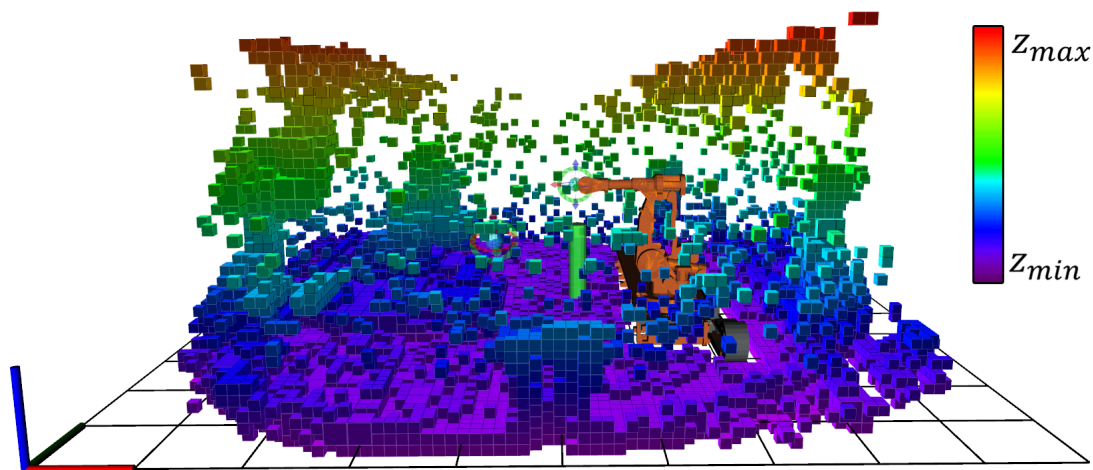


Figure 4.27: *Noise in the occupancy map*

A way to reduce the noise is to remove outliers by using a SOR filter introduced in chapter 2.2. `cloud_merged`, obtained from the algorithm shown with a flow chart in Figure 4.1, is used as input for the SOR filter. The output of the SOR filter is `cloud_filtered`, which is the cloud that is used as input for the occupancy map generator. Applying the filter directly resulted in a significant improvement. However, some outlying points were still appearing mid air. This issue caused problems when planning with obstacle avoidance. There were two main areas where outlying points was still present after applying the filter. These are highlighted in Figure 4.28 where all the coloured points are spurious. These spurious points could cause MoveIt to plan a strange motion around them, or even fail planning. To get the most out of the SOR filter, it can be tuned by changing two parameters:

- Number of neighbour points to analyze for each point (KNN)
- Standard deviation multiplier

When tuning the SOR filter, the optimal standard KNN was found at 50, with the deviation multiplier threshold set to 0.25. This reduced the number of spurious points caused by *Problem 2* from Figure 4.28 significantly. However, *Problem 1* was still a frequent issue.



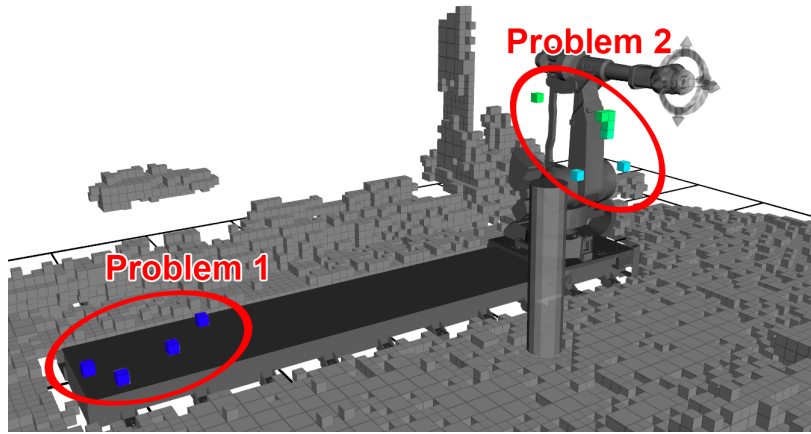


Figure 4.28: *Problem in the occupancy map*

By looking at the point clouds from the different Kinect cameras, it was discovered that the calibration of the Kinect cameras connected to `jetson1` and `jetson4` was slightly offset. They registered points around 10 cm above the track which was assumed to be the reason for *Problem 1* in Figure 4.28. `jetson1` was calibrated by adjusting the pitch and `jetson4` was adjusted downwards in the z-direction. They were calibrated by using the robot, floor, track and other point clouds as reference. *Problem 1* was eliminated by this calibration. It is worth mentioning that the result obtained for object detection and localization was performed after this calibration.

At first, the tuning and calibration seemed to be sufficient. However, by monitoring many consecutive frames, it was discovered that *Problem 2* was still present. After monitoring 200 frames, it was found that in 2% of the frames, points would appear right in front of the robot. It was discovered through eliminating one Kinect camera at a time that this point was caused by `jetson4` and `jetson6`. They were placed in the opposite end of the lab relative to the robot. The robot was just inside the sensors range, thus causing inaccuracies with a collection of low intensity points in this area.

Since the point clouds from the Kinect cameras held information about intensity, a threshold value for intensity could be set locally at the Jetson boards as proposed in [7]. Since this is just a threshold, it is a computationally cheap operation with a negligible increase in computational cost. The intensity is represented as a value between 0 and 255. By increasing the threshold from 0 to 10, the issue shown in Figure 4.28 disappeared and the accuracy of the occupancy map's representation of the environment did not undergo any noticeable change. Another upside of this intensity filter was the reduction in number of points to undergo the more comprehensive SOR filtering. The average total number of points from all Kinect cameras dropped from 216650 points to 215030 when applying the intensity filter, in other words a reduction of 0.75%. What this shows is that even though the intensity filter removed the issue, it did not lead to any significant loss of information.

### 4.3.2 Results

Figure 4.29 shows the results from the different filters used. It shows the mean distance to K nearest neighbours or the 50 nearest neighbours in this case. The unfiltered cloud (blue) had a huge amount of outliers. The intensity filter (red) removed lots of those spurious points, but it can be seen that there are still many outliers present. When applying the SOR filter (green) on the intensity filtered cloud, most outliers disappeared, and the graph reflects the significance of the SOR filter. Comparing the unfiltered occupancy map in Figure 4.27 with the intensity and SOR filtered map in Figure 4.30 gives a new perspective of the results shown in Figure 4.29.

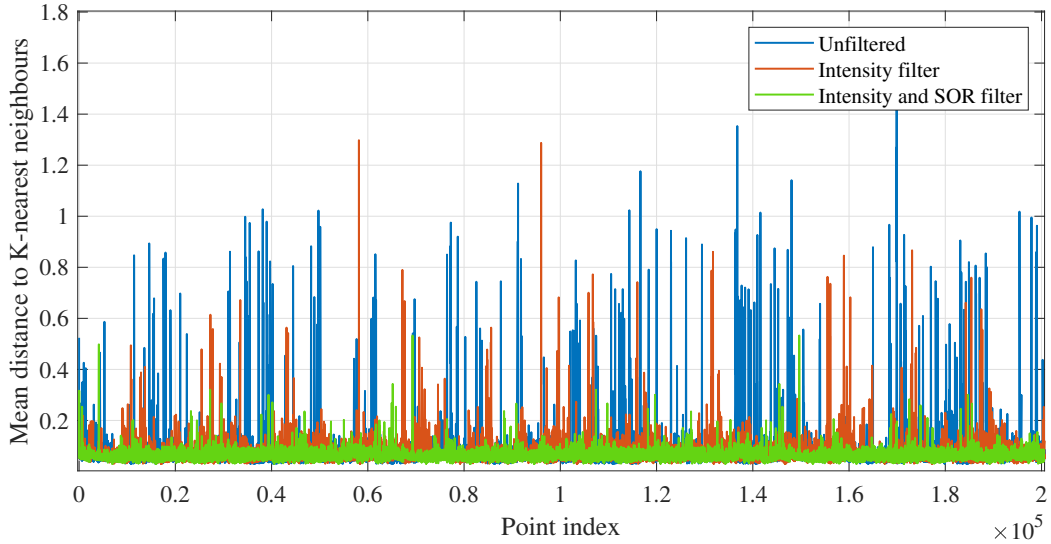


Figure 4.29: *Results on filtering*

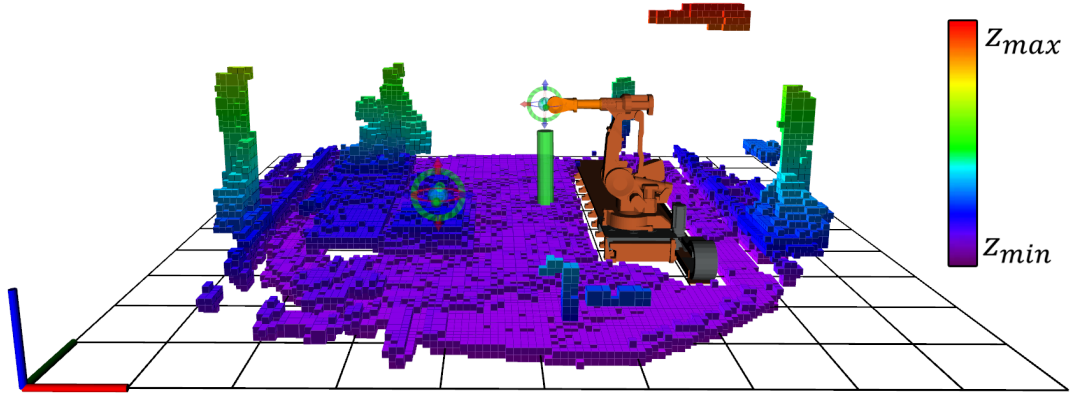


Figure 4.30: *Filtered occupancy map after tuning*

### 4.3.3 Creating an occupancy map with MoveIt

Integrating the result from Figure 4.30 with MoveIt is necessary to be able to use this map for collision-free motion planning. To create an occupancy map with MoveIt, the first step was to make a config file that can be seen in Appendix E.2.1. The most important parameters in this file are listed below:

- `point_cloud_topic` : Specifies what ROS topic the occupancy map should listen to.
- `max_range` : Specifies the maximum distance from the origin to the points to be included in the occupancy map.
- `point_subsample` : Specifies how many points are required within a voxel to be included as a point in the occupancy map.

The point cloud topic was set to `/master/merged_point_cloud` which was the output from merged and filtered (intensity and SOR) point cloud. Max range was set to 15 m to include the whole work area. Lastly, point subsample was set to 1 to include all registered points. It was found that increasing this led to holes in the floor and some loss of information on objects near the edge of the work area because they were not covered by multiple sensor nodes and therefore had fewer points.

To make sure that the config file from Appendix E.2.1 is used, the following command was added to one of the launch files:

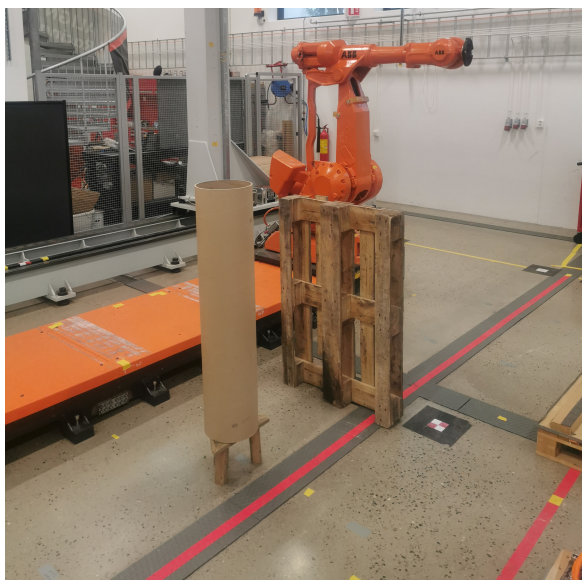
```
<roscpp command="load" file="$(find
→ p26_lefty_moveit_config)/config/sensors_kinect_pointcloud.yaml" />
```

The parameters set in the `sensor_manager.launch` file in Appendix E.2.2 was used to configure the occupancy map. These parameters are listed below:

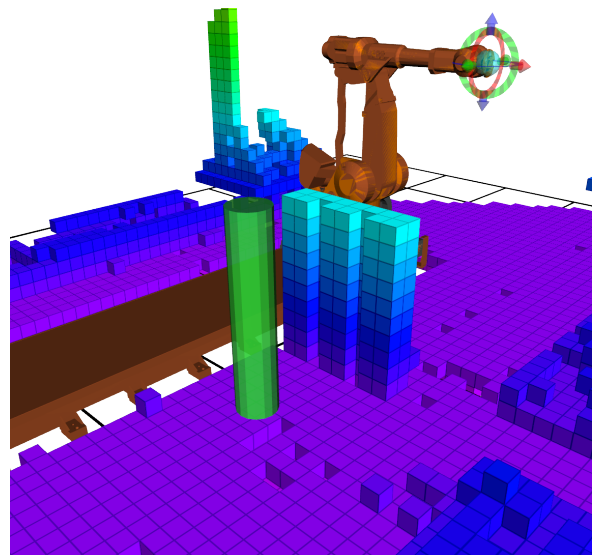
- `octomap_frame` : Specifies the coordinate frame in which this representation will be stored. When working with a mobile robot, this frame should be a fixed frame in the world.
- `octomap_resolution` : Specifies the resolution of the OctoMap (in meters).

First, the OctoMap frame was changed to the base frame of the robot, `lefty_track_left` which was the left side of the fixed track. Further on, the OctoMap resolution was changed to a relatively low resolution of 12 cm which corresponds to the size of 3 voxels from the voxel grid with 4 cm resolution. This induces an uncertainty about the points true position inside the blocks of the occupancy map because the point can be anywhere inside the 12 cm blocks. For object detection, that could have been a problem, but since the only use of this occupancy map is to map obstacles without any detection algorithms, a low resolution will be cheaper computationally and it will add a safety margin to the occupancy map for obstacle avoidance.

When the pick-and-place procedure starts, `move_group.launch` is launched. This launch file includes all the configurations done above, and made MoveIt create an occupancy map based on these configurations. Figure 4.31b shows how MoveIt perceived the target object (green) and a pallet, as well as other surroundings. Figure 4.31a shows the real world for comparison.



(a) Photo of robot, cylinder and pallet.



(b) MoveIt's perception of robot, cylinder and pallet with a resolution of 12 cm

Figure 4.31: Comparison between real world and MoveIt's perception



# Chapter 5

## Gripper development

A gripper was developed as a proof of concept for the complete pipeline pick-and-place procedure, with a goal of no manual intervention. The gripper must fit the cylindrical target object used in this project, whilst using cheap materials and components that were easily accessible to achieve a low-cost system. It should be able to grasp and pick the target from random poses inside the reachable environment, and holding it while performing the locomotion of the industrial robot from one point, then releasing it at the goal position.

In this chapter the conceptual and design phase of the mechanical system and its parts are reviewed, in addition to the setup of the manipulators hardware and software including controller, electronics and actuation. The grasping of the target must work in an automatic sequence with the rest of the pick-and-place procedure, without any interruption of the complete sequence.

An autonomous model-free grasping is not focused for this project, because the target object is known and due to the complexity of developing a completely autonomous gripper. Therefore, the gripper itself is limited to being automatic.

### 5.1 Mechanical design

This section includes the selection of overall mechanical architecture based on evaluation of concepts made. With a concept chosen, a compatible actuator was found from the university's collection. A detailed and final design was performed for every component of the chosen architecture, completed by final assembly and improvements.

#### 5.1.1 Concepts

Some guidelines were made in front of the conceptual phase based on the task at hand. These reduced the amount of considerations that had to be made in addition to simplifying the system software integration. The guidelines are listed below:

- Grasp around a 26 cm wide cylindrical target object radially
- Utilize rotational movement from one electric motor for actuation
- Simplify design for efficient production performed preferably with 3D-printing
- Reduce the distance from object COM to base to minimize bending moment at base
- Synchronous movement of multiple arms, ensuring a centered target COM when grasped

Three conceptual designs for grasping the target were made with the guidelines in mind. All designs utilized arms moving radially towards the target, but differs with the placement, joints and actuation

of the arms. Arms are what moves and interacts with the target. The idea was to apply force on the side of the target to obtain sufficient friction force to be able to lift the target while stabilizing it.

A scoring system was made to distinguish the concepts qualities, evaluating each concept upon given parameters. Every concept received a score between 1-5 for each parameter, where 5 is best. Intuition and sense was used to score each concept, in addition to comparing the concepts qualities. Every parameter is weighted equal, and the highest total score determined what concept to proceed with. The parameters are listed and explained below:

- Ease of production • Ease of design • Modularity • Robustness • Load applied

Ease of design and production, i.e. the estimated time and complexity needed to construct a final and working product of each concept. The estimated time includes, but was not limited to, 3D design with various considerations, 3D printing, processing with cutting and fastening, and assembly. Ease of production also includes possible difficulties with different components, such as potential for plastic breaking, and other fastening methods implying uncertainties. Modularity was each concepts potential for interchangeable components as well as possibility for use with different cylinder sizes with the same design. The robustness was estimated based on joint positions and overall design, with focus on potential weak points. Load applied indicated the utilization of torque from the motor directly translated to the nominal force applied on the target, thus increasing both friction and rigidity.

Each concept shown below are simplified versions, and was utilized as guidance for a final design. The chosen concept was to be designed and produced as a working prototype capable of grasping the cylindrical target object, in addition to holding it steady while the robot was moving. Some of each concepts advantages could also be implemented in another concept, if the final concept benefited from this.

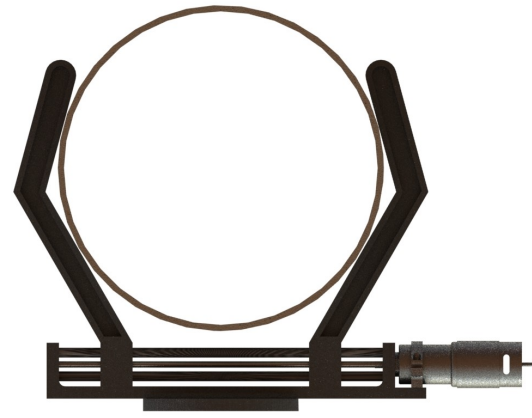
### **Concept 1, linear actuator**

The first concept is based on the linear translation from a linear actuator with lead screws and nuts, the concept can be seen in Figure 5.1. With the additional gearing obtained from the lead screw the maximum force applied would be greater, and a smaller motor could be implemented compared to a direct drive. Due to the design of the arms and the linear translation, the centre of the cylinders with different diameters would all have equal distance from the center to the base. By having these distances similar for any size of the cylinders, the grasping action and cylinder position in MoveIt were independent for the different cylinders.

Friction could become an issue between the arms and sliders, especially with tension increased. The wide base and the placement of the motor could limit the movement of the robot by inducing constraints. However, the base could be narrowed by increasing the lower part angle on the arms, and reduced to the required travelling length from extended position to grasped on the smallest cylinder. The slider rods would be smooth steel or carbon fiber and coated with a friction reducing lube. As the translational movement of the arms are opposite, the lead screw had to consist of two parts, where one side is links. The production of this concept, with a wide base, fastening of opposite lead screws and assembly of rods could be difficult and time consuming.



(a) *Side view of concept 1, linear actuator*



(b) *Front view of concept 1, linear actuator with 26 cm cylinder*

Figure 5.1: *Illustrations of concept 1, linear actuator*

### Concept 2, Circular arms

With this concept the motor acts directly on one of the arms which is connected to the non-actuated arm through gears making sure for synchronous movement. The conceptual illustrations can be seen in Figure 5.2. This concept had the fewest number of parts and was assumed to be the easiest to both design, produce and assemble. The arms could be made straight like in concept 1, adding more modularity but with less strength and fewer stability possibilities at the base as well as a smaller contact area. For this explicit design the circular gears were limited in size to minimize the distance from the base to the COM due to their radius. With non-circular gears, as in concept 3, this distance could be kept to a minimum and the joint distance would be widened. A widened base and joint distance would increase the strength of the base and make the arms move more perpendicular to the cylindrical target.



(a) *Side view of concept 2, circular arms*



(b) *Front view of concept 2, circular arms with 26 cm cylinder*

Figure 5.2: *Illustrations of concept 2, circular arms*

### Concept 3, Rod-type

The last concept, and the most complex with the moving arms disconnected from the direct drive of the motor. The concept illustrations can be seen in Figure 5.3. Here, rods are placed between the direct gears and the translating arms. The idea was to apply load towards the opposite side of the target, pushing the target towards the center of the gripper base. By applying force from this direction, more of the circumference would be attached to the gripper base, ensuring more friction and rigidity with a rigidity-enhancing design placed here.

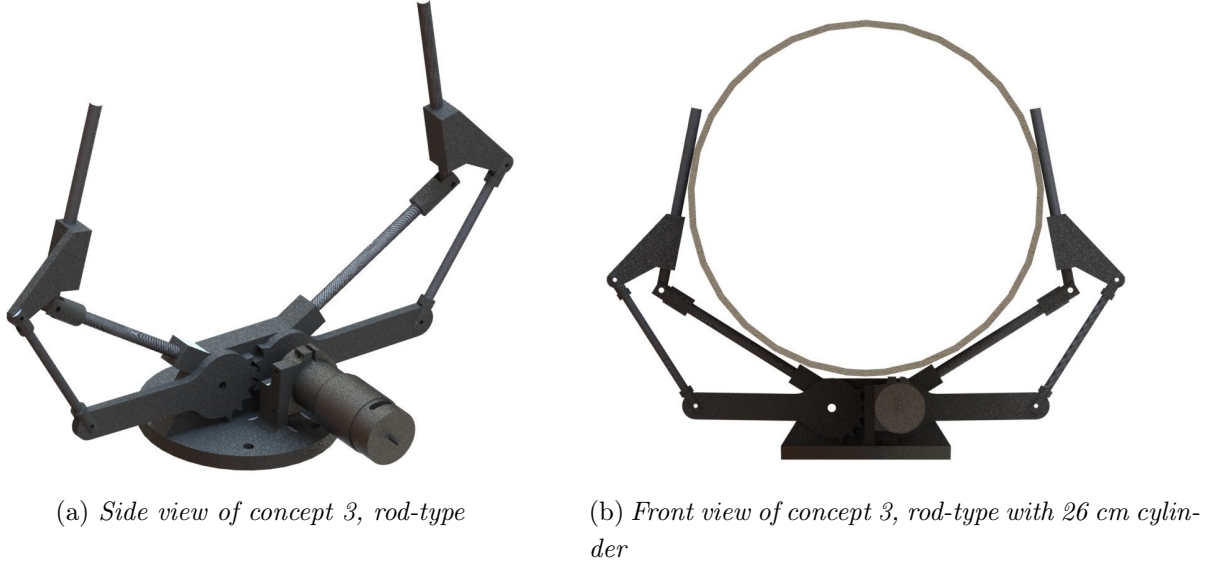


Figure 5.3: Illustrations of concept 3, rod-type

All rods could be of carbon fibre, ensuring a light and strong construction together with 3D-printed leavers and rod connectors. To fasten the rods and leavers, glue could be applied, implying a non-reversible action of the assembly. Otherwise, bolts could be used through the rods, thus weakening the carbon fibre with a hole. In addition, the difference of the cylinder COM position is vast between a smaller and a bigger cylinder.

### Load applied

Each concept has a different utilization of the motor torque, thus having different applied force with an equal motor. Figure 5.4 shows the geometries of concept 2 and 3, and their lengths affecting the applied force. Equations (5.1-5.3) shows how the different lengths and designs utilizes the torque. Every length is based on concept CAD-models, and losses are not included. It is clear that the applied force achieved with concept 1 and lead screw is superior, and a smaller motor could be used here, also reducing the width of the base. For concept 3, with the outer link between  $L_{s2}$  and  $L_{s3}$  not being perpendicular to the actuated arm, some torque is lost in the translation, thus further reducing the applied force. However, as concept 3 applies force more towards the base, it is believed that more stability could be achieved due to pressure between the target and base.

$$F_{N,1} = T_m \cdot \frac{2\pi}{L_s \cdot L_L} \quad (5.1)$$

$$F_{N,2} = \frac{T_m}{L_L} \quad (5.2)$$

$$F_{N,3} = T_m \cdot \frac{L_2}{L_1 \cdot L_3} \quad (5.3)$$

where,

	Description	Value	Unit
$F_{N,1}$	Applied force for concept 1	$28 \cdot T_m$	$N$
$F_{N,2}$	Applied force for concept 2	$7 \cdot T_m$	$N$
$F_{N,3}$	Applied force for concept 3	$3 \cdot T_m$	$N$
$T_m$	Motor torque	–	$Nm$
$L_s$	Lead screw pitch distance	1.5	$mm$
$L_L$	Length arm concept 1 and 2	0.15	$m$
$L_1$	Length 1	0.03	$m$
$L_2$	Length 2	0.1	$m$
$L_2$	Length 3	0.1	$m$

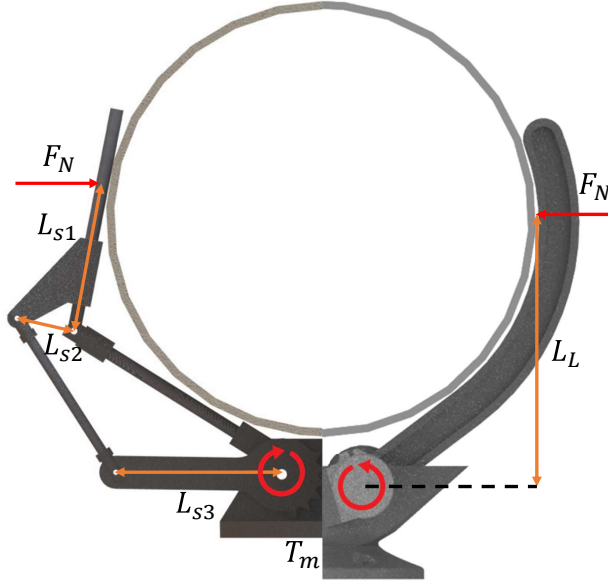


Figure 5.4: *Concepts 2 and 3 with corresponding force-influencing lengths*

### 5.1.2 Soft robotics gripper

Another approach could be to make a soft robotics gripper. Within soft robotics, the idea is to make robots less hazardous for people and the surroundings. One part is the danger of rigid-body grippers and their powerful actuators and applied force, where many industrial "hard" grippers have a gripper force sufficient to potentially hurt people. Soft robotic grippers with soft and flexible materials tend to utilize vacuum, air pressure or wires to actuate the gripper components, and different adhesives to increase friction. An example can be found in [49], which utilize air pressure to inflate the outside bladders to control the actuation of the arms, and a gecko-inspired adhesive to enhance friction. The gripper can be seen in Figure 5.5, and a similar and bigger design could be used to pick the target used in this project. However, the combination of different materials and actuation control was assumed to be time consuming and hard to implement in rapid and low-cost prototyping. With soft materials and bodies, more complex force sensing and control would be required due to less rigidity in the assembly [50].

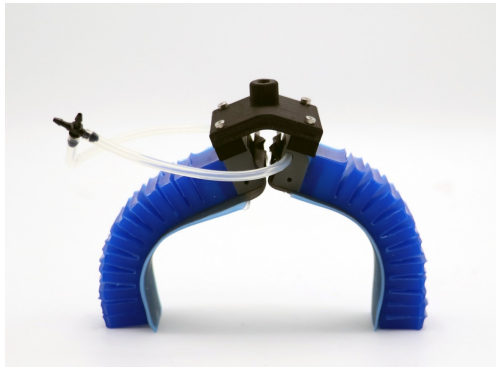


Figure 5.5: *Example of a soft gripper with air pressure and special adhesive[49]*

### 5.1.3 Concept evaluation

Each concept were evaluated upon the described parameters and compared to each other, and their result can be seen in Figure 5.6, including the total score next to their names. Based on the overall technical impression and total score, concept 2 with circular arms was determined to be proceeded with. Due to its primitive design it is considered the most efficient to both design and produce, with fewer parts and joints. It is also considered the most robust with a solid base and arms in addition to directly connected motor to the actuated arm, thus giving a fair utilization of motor torque.

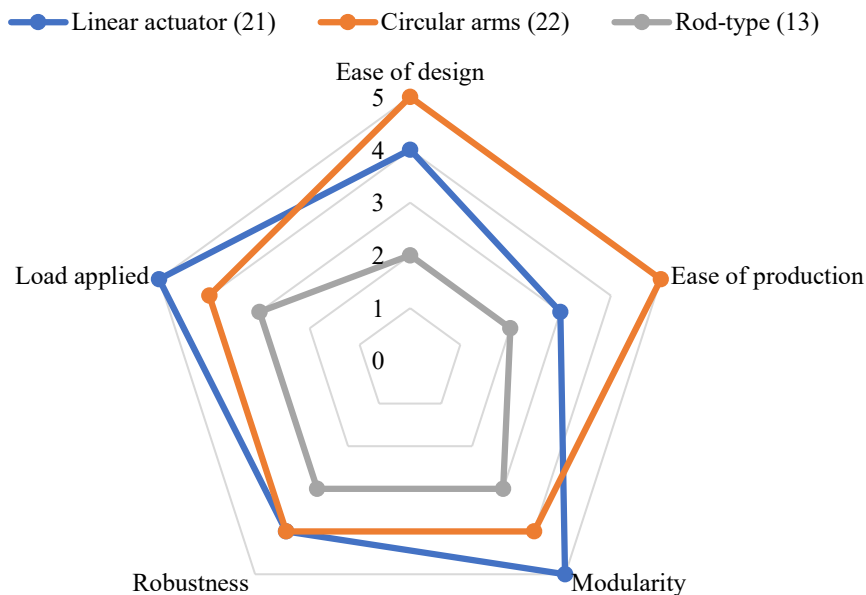


Figure 5.6: *Radar diagram of concepts score and total score*

### 5.1.4 Gripper overall architecture

With the concept chosen, the components could be further developed with regards to smaller details and to improve the overall design together with actuator and position sensor.

#### Motor selection

Motor speed, size, weight and stall torque were important parameters when selecting a compatible motor to be used on the gripper. Its size and weight were to be kept to a minimum reducing the necessary support from the base. The motor speed should be slow as the arms only needed to rotate

a few degrees. The required torque had to be estimated based on preliminary design and objects to be grasped, the minimum required torque was calculated as shown in equations (5.4-5.7) based on the free body diagram illustrated in Figure 5.7.

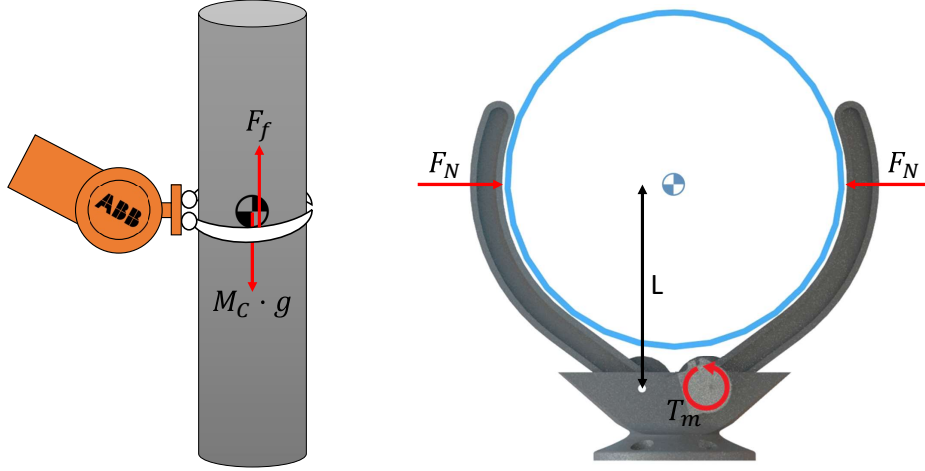


Figure 5.7: Free body diagram of gripper acting on target

The friction force comes from the arms being pushed radially towards the target. With arms made from 3D-printed PLA and the target made from cardboard the friction coefficient was found to be relatively low. By attaching sealant strips made from rubber on the arms the friction was increased. The point of contact was assumed to be at the outside of the COM of the target, the distance from the arm joints to the contact point is illustrated as  $L$  in Figure 5.7. This distance was not set at the time of the motor selection, but was assumed to be somewhat larger than the radius of the cylindrical target.

$$F_f = \mu \cdot 2 \cdot F_N \quad (5.4)$$

$$F_N = \frac{T_m}{2 \cdot L} \quad (5.5)$$

$$F_f > M_c \cdot g \quad (5.6)$$

$$\Downarrow$$

$$T_m > \frac{M_c \cdot g \cdot L}{\mu} \quad (5.7)$$

where,

	Description	Value	Unit
$T_m$	Minimum required motor stall torque	9.7	$Nm$
$M_c$	26 cm cylinder mass	3.3	$kg$
$g$	Gravitational acceleration	9.81	$\frac{m}{s^2}$
$L$	Length from joint center to nominal force acting on cylindrical target	0.15	$m$
$\mu$	Friction coefficient between rubber and cardboard [51]	0.5	–
$F_f$	Friction force		$N$
$F_N$	Nominal force		$N$



Now, a motor could be selected from the university's assortment of actuators based on the calculated minimum required torque calculated and an idea of its size, weight and speed. An Actobotics 12 V DC motor with a vast gearing was found suitable for the task at hand, with a small size, slow and controllable speed and high rated stall torque. The minimum stall torque was rated at 418 kgf-cm which corresponds to 41 Nm, greater than the estimated required torque. Its dimensions and characteristics can be seen in Appendix C.

### 5.1.5 Base

The base is the foundation of the complete gripper with both arms, motor, position sensor and gears connected to the base. It carries all the weight, including the target and should be designed tough, yet not difficult nor time consuming to produce and 3D-print.

By making the base as shallow as possible the bending moment caused by the target mass would be lower, applying less stress on the base. The arms should also have a widened joint separation causing a more horizontal movement of the arms as illustrated in Figure 5.8. By doing this the change in position for smaller and bigger cylinders was reduced, with a more horizontal movement than vertical. In addition, the arms could be made shorter while keeping the same property of reaching around the cylindrical target, pushing it towards the base reducing the risk of it slipping out.

The combination of a widened and shallower base, and the placement of the motor meant that it would interact with the six-bolt pattern for the fastening to the industrial robot tip. It was decided to only use four bolts, and assume that this would be adequate to fasten the base and gripper to the base. By twisting the top of the base from the flat part bolted to the robot end, the motor would also not interact with the bolt pattern. The four-bolt pattern and twisting can be seen in Figure 5.9. The notches around each bolt hole facilitates space for the hex socket needed for fastening the bolts and also spread the tension from the bolts at an even and flat area.

A double-D notch, similar to the one found on the motor front chassis was made in the base where the motor was fastened to better counteract the torque. A vertical plate was implemented to the side of the motor, by attaching the motor bracket to this plate, the motor was completely fastened to the base, both counteracting its weight and induced torque. The plate can be seen in Figure 5.9b and is the vertical part besides the right hole. A similar plate was placed on the other side of the base, and both were extended in height and used as stabilizers for the target. The other plate was also made with a bracket facilitating the use of rotational position sensor, such as potentiometer or encoder.

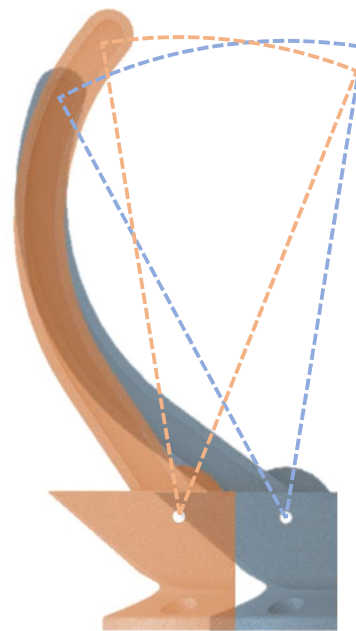
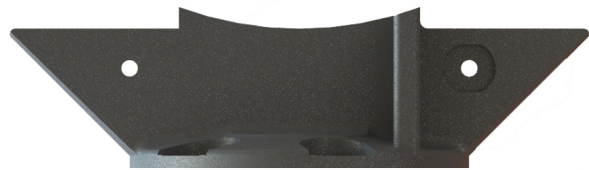


Figure 5.8: *Movement with wider base*



(a) *Top side of base*



(b) *Side view of base*

Figure 5.9: *Top and side view of base*

### **Motor connector**

With a minimum stall torque of more than 40 Nm, a 6 mm D-shaft on the motor and a corresponding connection in the 3D-printed plastic, the plastic would likely break at an early stage. A leverage that would fit both the motor shaft and distribute the forces at a larger area on the arm was implemented. A tail rotor hub from a RC-helicopter was found suitable and was cut in half and used as leverage. A notch in the arm was made making a snug fit for the leverage, and a set screw was used to connect the leverage to the motor shaft through a hole in the arm. An illustration of the assembly in exploded view can be seen in Figure 5.10. The hole in the motor shaft had to be fabricated to fit the smooth end of the set screw.

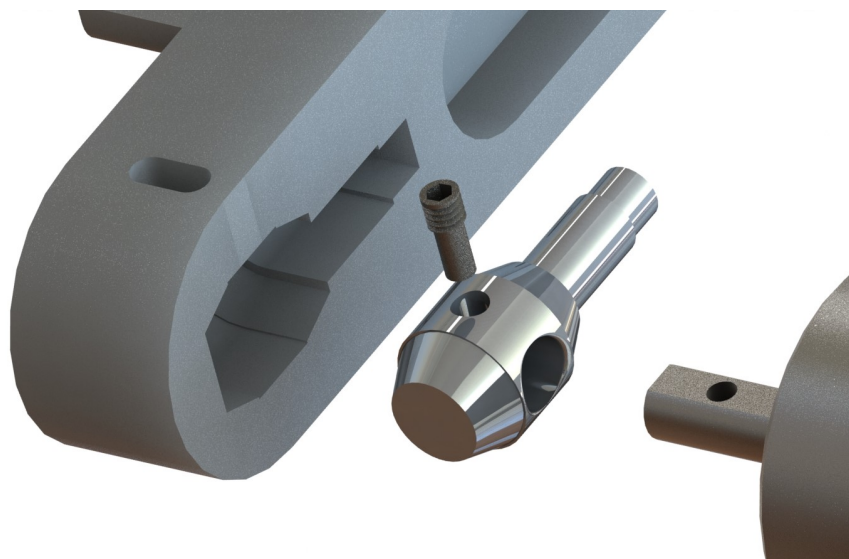


Figure 5.10: *Exploded view of the motor connection with leverage*

### 5.1.6 Arms

With this design, the arms are what mainly holds the target in place, thus requiring a tough design capable of stabilizing the target. The arms were designed quite long, being able to reach above center of the circumference of the cylindrical target, pushing the target towards the base, utilizing the stabilizing elements there as well. A notch in the upper part of the arm was made, where a longer rod was placed to further stabilize the target, especially in the longitudinal direction. The actuated arm and rod can be seen in Figure 5.11. The stabilizing rods and the shaft between the arms and base were of carbon fiber to maintain a low weight of the complete assembly. A rubber sealing strip was added with double-sided tape to the inside of the arms and stabilizing rods to add more friction between the gripper and target.

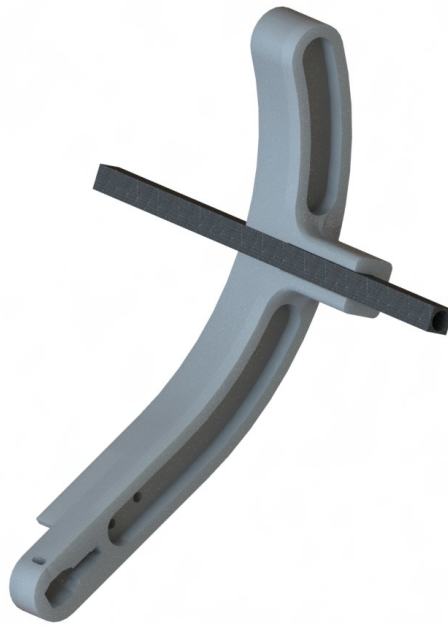


Figure 5.11: *Arm with stabilizing carbon rod*

### 5.1.7 Synchronous gears

A widened base and a greater distance between the joints would directly impact the required size of the gears providing synchronous movement of the non-actuated arm. If a circular gear was to be used, the vast radius would require the target to be moved away from the base due to the size of such gears, increasing the required torque at base and the required arm lengths. The gears overall radius had to be reduced to minimize the distance from the base to the centre of the target. This was done by only producing a fraction of a gear, which was possible due to the small rotation needed of the arms to grasp the target. 3D-printing was seen as the best solution to be able to produce these custom gears although lacking the strength and integrity of "ordinary" circular cast plastic or metal gears. As the arms were quite large, and took some time to print, it was desired to maintain a level of modularity for the printed parts. By implementing the custom gears as separate parts they could be easily interchanged if broken or in the need of modifications.

The gears were fastened to the arms with the narrow part at the outside tip which can be seen in Figure 5.12. These fits into the arms and were additionally fastened with bolts and nuts. It was not considered necessary to have more than one complete tooth at the passive side due to the limited rotation, the passive gear is the left one in Figure 5.12. The highest load was applied when grasping the target, and the gears were designed such that the complete face of the teeth were interacting in

this position, the same position are shown in Figure 5.12. The gears induced minor slack, but this was not a problem when actuating in one direction at a time, which is the case for such a gripper.

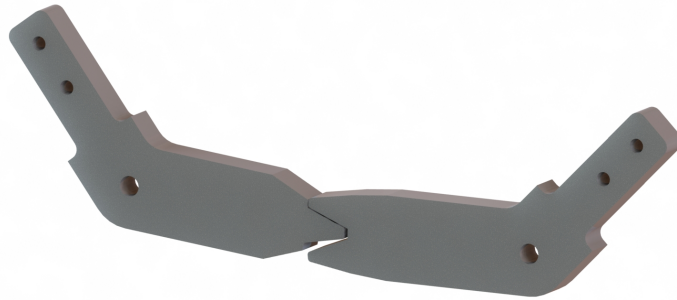


Figure 5.12: *Synchronous gears*

### 5.1.8 Position sensor

A rotational position sensor was added to the assembly to have control over the movement of the arms. This was fastened to the base at the center of the arm joint and to the non-actuated arm through a link. The position sensor was not added directly to the motor to have control even if the motor should be slipping in its attachments or detect if the arms would move differently, i.e. broken gears or arms.

### 5.1.9 Complete assembly



(a) *Real life assembly with cables, fasteners and rubber seals*



(b) *CAD-model of assembly*

Figure 5.13: *Real and CAD-model of complete gripper assembly*

With all parts designed and printed these were assembled with bolts, nuts, tape, cable ties and carbon shafts. It was chosen to use an orange filament to better fit the orange industrial robot, in addition to it being a hazardous color easier to spot. A comparison between the clean CAD-model and the real life assembly with all accessories and fasteners can be seen in Figure 5.13. The complete assembly and corresponding parts can be found at [GrabCAD](#). The placement of the sealant strips can also be seen in Figure 5.13a, where the gripper is bolted to the industrial robot end-effector

link. The black tape was holding the cables to the motor and position sensor in position, keeping them from being squeezed.

## 5.2 Hardware setup

In this section different hardware components are chosen, set up and combined giving a sufficient control of the gripper actuation.

### 5.2.1 Controller

A controller communicating over ROS as a node was determined to use together with the gripper, enabling a running sequence together with the main computer. Through ROS, the idea was to communicate when to extend and retract the arms, and feedback the applied force and arms position so the sequence could proceed when feedback values are within defined threshold values. With an actuator and with sensors, I/O-control was implemented to be able to perform the different actions of the gripper.

The industrial ABB-robot had some I/O-ports, see Appendix B, which could feed through signals controlling actuation and containing sensor feedback data. However, the signals would only be passing through the robot, and all calculations and controlling signals would come from the main computer. In addition, the internal RAPID-code on the ABB-robot had to be manipulated in order to pass these signals through ROS, and compatible physical connectors were not available at the university during this project.

An additional ROS node extending the existing ROS network, controlling the actuator directly was seen as a more versatile solution and reduced the computational resources on the main computer. This node would not be dependant on a specific kind of robot, and could be implemented in any sort of ROS network.

### 5.2.2 Hardware components

All major hardware used for the gripper are listed below:

- Raspberry Pi 4B w/Power Supply
- 12 V DC motor with gearing
- DC Motor Driver Module
- Arduino Uno
- Potentiometer

#### Raspberry Pi 4B

A Raspberry Pi 4B (RPi) was used as the gripper controller and ROS node. RPi was able to run Ubuntu, with ROS Kinetic through ethernet connection to the local LAN. The RPi included many general purpose input-output (GPIO) digital pins, including pulse width modulation (PWM)-channels that could control the actuator. The RPi 4B-version also included WiFi, which was handy when performing troubleshooting from a laptop, without the need of connection to the local LAN.

The RPi 4B may be small in size, but did not lack the computational resources required for the intended use in this project, the processor runs at 1.5 GHz and the version used in this project

had 4GB of memory (RAM). The main reason for using the RPi was due to its small footprint, possibilities for using Ubuntu and ROS, the ethernet connection enabling use with the local LAN and the 40 pins GPIO, in addition to the fact that it was available at the university. However, some issues became prominent while setting up the RPi, it did not include analog pins, being able to interpret analog sensor signals. Also, the pins were only designed for 3.3V, as oppose to many electrical components that works with 5V, which if sent to the RPi, may permanently damage the ports as it does not include protection for the GPIO [52, p. 14].

## **DC Motor Driver Module**

From the university, a DC motor driver module was given to be able to control the input voltage to the DC motor. The driver module was built upon a VNH2SP30 H-bridge motor driver for automotive applications and used PWM and two logical inputs to determine speed and direction of the motor, see Appendix D. With the VNH2SP30, PWM could be run at up to 20kHz and current sense could be implemented, facilitating force sensor when grasping the target. Even without the analog pins on the RPi, PWM enables the use of a similar approach by using the RPi's hardware PWM or by software.

The current sense voltage runs approximately proportional with the current through the motor, and enabled the use of resemblance of force feedback on the force from the gripper arms applied on the target. This was possible due to the DC motor characteristics and torque being inverse proportional to the rotational speed of the motor shaft, and the proportional relationship between torque and current drawn, resulting in increased current drawn with more load applied i.e. less speed. For this purpose a separate and more precise current sensor was not deemed necessary, as the specific current drawn was of no interest as opposed to the proportional and consistent value corresponding to the load applied from the gripper arms. Also, the planned use of the force feedback would be as a threshold value for the motor to stop when reached. This threshold value would be set based upon empirical testing and validation.

## **Potentiometer**

Many solutions for observing the arms position exists, both digital (e.g. encoder and limit switch) and analog (e.g. potentiometer and ultrasonic depth sensor). An encoder or potentiometer were prominent methods to be able to constantly observe the arms rotational movement. However, the need of calibrating the encoder for every boot-up made the potentiometer the best solution for the intended use. From the mechanical design the potentiometer was placed at the non-actuated arm joint, and connected through a link to the arm following the rotational movement.

## **Arduino Uno**

An Arduino was utilized as an interpreter for the analog signals from the current sensor and potentiometer, as the RPi did not have analog pins. The university did not have a separate ADC nor a motor driver running with serial communication. The Arduino had a built-in ADC and could communicate serially over USB to the RPi using different libraries and protocols. In addition, the Arduino had multiple analog inputs and outputs which prepared the manipulator for extension with more sensors or actuators if necessary.



## Connections

A connection diagram for the different hardware parts can be seen in Figure 5.14 showing all connections between the different hardware components. The USB-cable represent the connection between the Raspberry Pi and the Arduino Uno, which both powered the Arduino and handled signal transfer. The Raspberry Pi was connected to a power supply and ethernet to the local LAN. The 12V DC power supply to the motor driver comes from the 220V AC - 12V DC transformer.

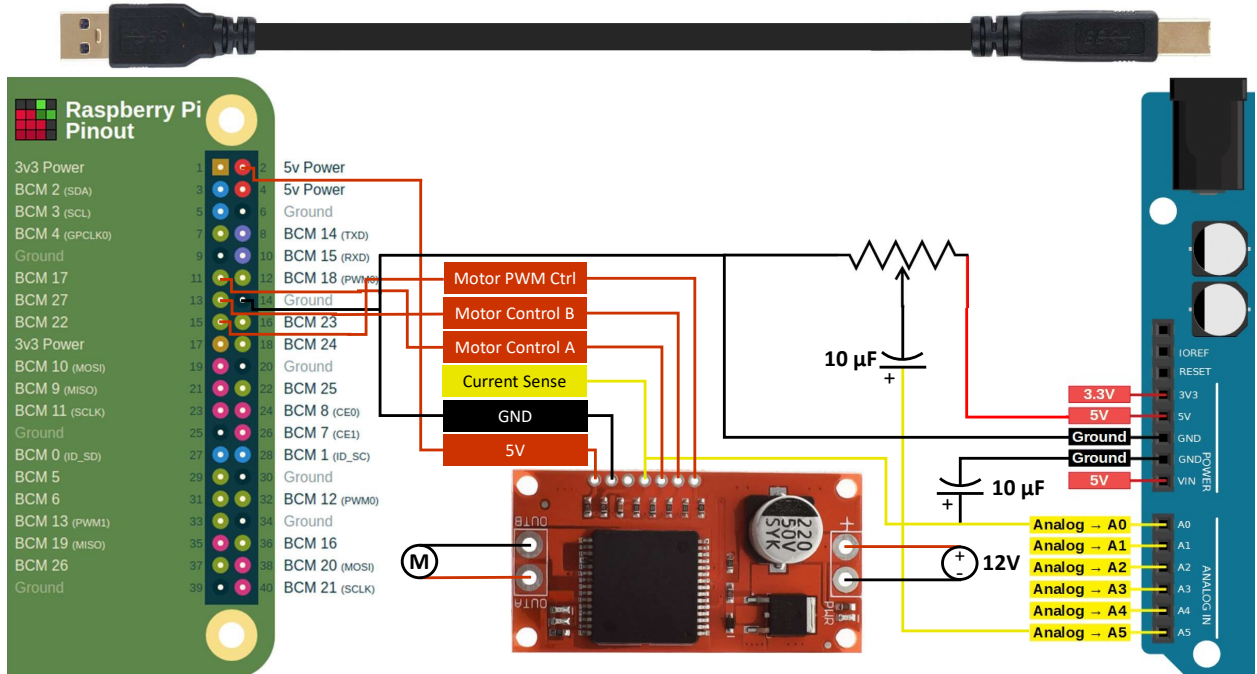


Figure 5.14: *Electrical diagram for connections between Raspberry Pi, Arduino Uno, potentiometer and DC Motor Driver*

### 5.2.3 Installation and software setup

#### Raspberry Pi

As the rest of the system utilize ROS Kinetic for communication, this was also desired to use on the RPi to reduce the risk of compatibility issues across the different devices. To be able to run ROS Kinetic, Ubuntu 16.04 was considered the most suitable and compatible operating system [53]. A complete Ubuntu 16.04 image installation, made especially for RPi 3/4, was found [54] and installed on the RPi. This image was pre-installed with ROS Kinetic for robotic use, but required some changes regarding network setup and time synchronization.

#### Ubiquity Robotics Ubuntu 16.04

Ubiquity Robotics was the manufacturer of the installed Ubuntu 16.04 operating system on the RPi, and stated that it worked on both RPi 3 (Model B and B+) and RPi 4B. The image is based on Ubuntu 16.04 and came pre-installed with ROS Kinetic and a catkin workspace already setup. However, as this image was primarily built for use with their robots some startup scripts had to be disabled, including the automatic startup of `roscore`, which is not applicable for the lab setup with the main computer running `roscore`. The startup scripts were disabled by running the following commands [55]:

```
sudo systemctl disable magni-base
```



```
sudo systemctl disable roscore
```

## Network

To be able to connect the Raspberry Pi to the local net in the lab, a static IP, with corresponding gateway, DNS and netmask had to be configured for the ethernet connection. The following connection properties were used:

- IP: 10.225.120.56
- Netmask: 255.255.255.0
- Gateway: 10.225.120.0
- DNS: 10.225.120.50

The local LAN was a closed net without connection to the internet to prevent any impact from intruders. The lack of internet access was unfortunate when there was a need of installing different packages and libraries on the RPi. To solve this a hotspot from a laptop was setup and connected at boot whenever working on the RPi. With successful connection to both the local net and the internet over WiFi the next step was to setup the different ROS network parameters to be able to connect through the main computer running the master node `roscore`. With this the following parameters had to be declared in `~\.bashrc` to be initialized at boot, with the command `export ROS_...` [56]:

- `ROS_MASTER_URI = "http://10.225.120.50:11311"`
- `ROS_IP = "10.225.120.56"`
- `ROS_HOSTNAME = "p26_raspberry.local"`

Pinging and the commands `rostopic list` and `rostopic hz <rostopic>` were used to confirm the connection to the local net and ROS setup in the lab with the RPi. However, there were some trouble when trying to use the latter and subscribing to a ROS topic. From [56] the idea of time discrepancy between the different components appeared.

## Time synchronization

By installing chrony with `sudo apt install chrony`, initializing it at boot by including it in `~\.bashrc` with `systemctl enable chronyd` and setting up the main computer as the reference client with its IP, the RPi clock was following the reference clock in the main computer as can be seen in Figure 5.15. However, the RPi was still not able to subscribe to any topic found by `rostopic list`. By looking up the time and date settings on the RPi with `timedatectl`, it was observed that the realtime clock (RTC) was not similar to both the local and universal time as can be seen in Figure 5.16. This was due to the fact that Raspberry Pis do not have a realtime clock built-in, which was presumably used by ROS. However, there was a quick work-around that manipulated the RPi to mirror the reference time as RTC. This was performed by installing `fake-hwclock` [57] and enabling it with `sudo systemctl enable fake-hwclock`. Now, the realtime clock was following chrony with the main computer as host, and subscribing to different topics in ROS worked without any noticeable issues.

```

ubuntu@p26_raspberry:~/catkin_ws$ chronyc tracking
Reference ID      : 10.225.120.50 (wp3rack.wp3.local)
Stratum          : 4
Ref time (UTC)   : Thu Apr  8 10:56:32 2021
System time      : 0.000025046 seconds fast of NTP time
Last offset      : +0.000087191 seconds
RMS offset       : 0.000163176 seconds
Frequency        : 18.387 ppm fast
Residual freq    : +2.147 ppm
Skew             : 0.339 ppm
Root delay       : 0.016791 seconds
Root dispersion  : 0.001514 seconds
Update interval  : 64.7 seconds
Leap status      : Normal

```

Figure 5.15: *Time synchronization between RPi and main computer*

```

ubuntu@p26_raspberry:~/catkin_ws$ timedatectl
Local time: Thu 2021-04-08 12:51:26 CEST
Universal time: Thu 2021-04-08 10:51:26 UTC
RTC time: n/a
Time zone: Europe/Oslo (CEST, +0200)
Network time on: yes
NTP synchronized: yes
RTC in local TZ: no

```

Figure 5.16: *Raspberry without real-time clock (RTC)*

## General Purpose Input Output

The general purpose input/output (GPIO) on the RPi consists of a 40-pin header with digital pins, continuous 3.3 and 5V power outputs and ground pins. Two of the pins included hardware operated PWM channels, enabling control of the DC driver and speed of the motor. Together with the DC motor driver, five pins and connections were required as can be seen in Figure 5.14, with 5V power, ground, PWM and two connections for controlling the state of the bridge. With only digital GPIO on the RPi, and to be able to perceive the current sense and potentiometer voltage, a separate analog to digital-converter (ADC) was used in the form of an Arduino Uno serially connected to the RPi using USB.

The serial communication between the RPi and Arduino was feasible with the use of Firmata[58], a software communication protocol between host computer and different microcontrollers. On the Arduino, Firmata was implemented using an example script given directly from the Arduino app called StandardFirmata. With this script all Arduino pins could be read and controlled using a host computer, i.e. RPi connected serially with the Arduino. On the RPi and with Firmata for python, pyFirmata [59] implemented, the Arduino pins could be accessed. The implementation code can be seen in Appendix E.4.1. Using pyFirmata, the signal input range on the RPi was 0 to 1.

## Motor actuation and sensor filtration

With the current sense and potentiometer connected and set up, motor actuation was initialized using PWM. It was determined to further evaluate two libraries with the RPi and PWM, either pigpio[60] with hardware timed PWM or RPi.GPIO[61] with software PWM. Every output pin could be used with software PWM, whereas hardware PWM on the RPi only has two channels with a total of 4 original PWM pins. However, pigpio controls the GPIO on the RPi and every pin can be used with hardware PWM, but limited to two channels i.e. a total of two PWM frequencies running simultaneously. With only the motor driver utilizing the PWM-signal, the accuracy was of utmost importance when choosing between pigpio and RPi.GPIO. With the motor running at constant speed with no load the current sense voltage was measured with both methods, the comparison can be seen in Figure 5.17.

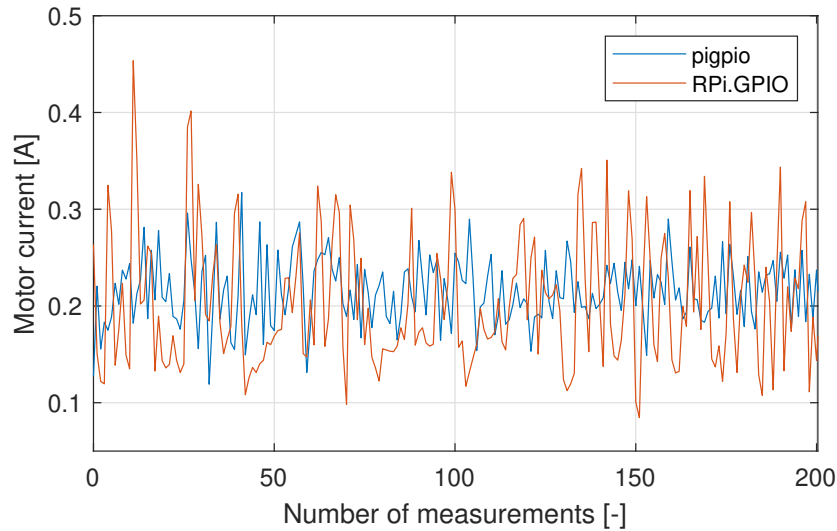


Figure 5.17: *PWM noise for pigpio and RPi.GPIO*

From Figure 5.17 it was noticeable that pigpio with hardware PWM had the least amount of noise and higher accuracy than RPi.GPIO. Jerk was also prominent when using RPi.GPIO with the motor having sudden and rapid accelerations. However, with RPi.GPIO and software PWM the motor would stop running if the script failed or was killed. In the same case with pigpio and hardware PWM, the motor would continue to run, and potentially destroy mechanical components if assembled. By including a try block containing the source code, and an exception block containing a function that stopped the motor, this issue was suppressed.

The gripper would be actuating while the current sense would be within a set threshold value. The high spikes seen when using RPi.GPIO would make this type of actuating less accurate, and the threshold value could randomly be exceeded. A low-pass filter with a small capacitor was implemented for the current sensor and potentiometer as seen in Figure 5.14. The low-pass filter greatly reduced the noise for both analog sensors, the reduction for the current sensor as seen in Figure 5.18.

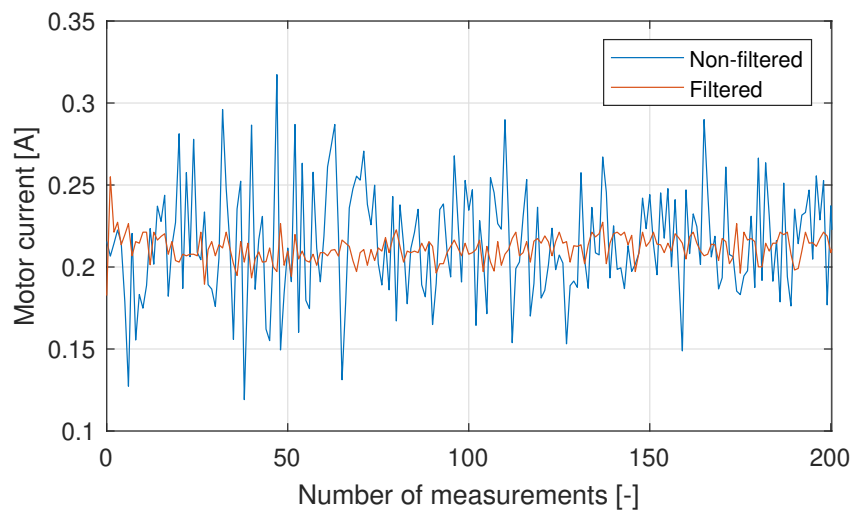


Figure 5.18: *Current sense noise with and without low-pass filtering*

The average of every 40 measurements was utilized to further smoothen the current sense signal. The signal was also multiplied with  $\frac{5}{0.13}$  to obtain an estimation of the real current, 5 acquired

the Arduino voltage, 0.13 is the approximate  $V/A$  from the current sense. However, as the current sense voltage was proportional to the real current, the estimation was only utilized as an indication and prevent potential overloading of the motor driver.

### 5.3 Control setup

With the motor rotating the arms, sensors working and every component assembled, the control architecture could be developed. The idea for the gripper was to work as a state-machine with actuation of the arms in both directions working in sync with the navigation of the industrial robot. The planned sequence of the grasping action is shown in Figure 5.19, and represents a simple procedure with the navigation to the COM of the target (1), followed by the retraction of the gripper arms gripping around the cylindrical target (2). In this section the control algorithm for the extension and retraction of the arms will be explained utilizing sensor data in combination with threshold values.

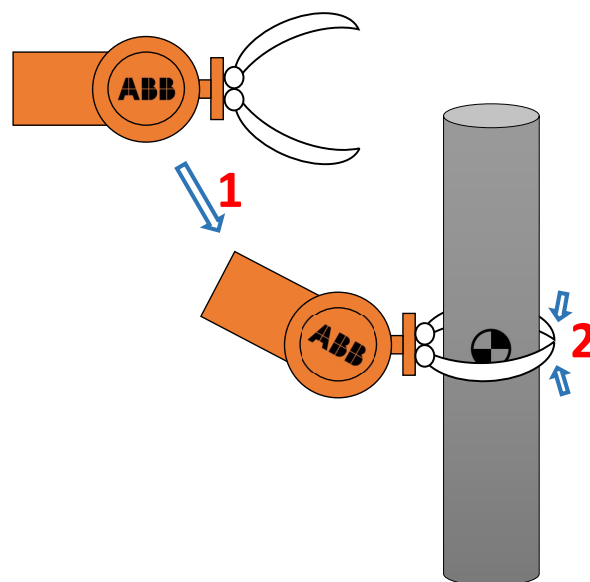


Figure 5.19: *Sequence of grasping*

#### 5.3.1 Control architecture

To control the arms, two different algorithms were made, extension and retraction of the arms. The different algorithm flow charts can be seen in Figures 5.20-5.21 and their source code can be seen in Appendix E.4.2 called `extend` and `retract`. Both are similar, but differs mainly with the pre-check of gripper is already extended with the `extend`-function and the motor brake with applied torque with retraction. The main component is the while-loop running with motor spinning while the sensor measurements are within the set threshold values. If the position or current sense data is greater than the set threshold values the motor will stop and hold.

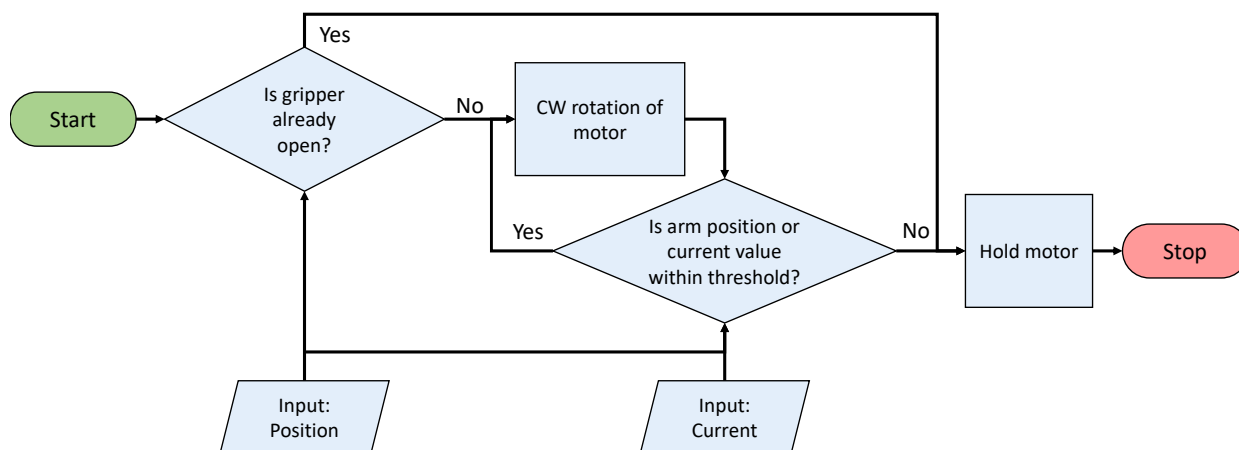


Figure 5.20: *Flow chart of the extension of the gripper arms*

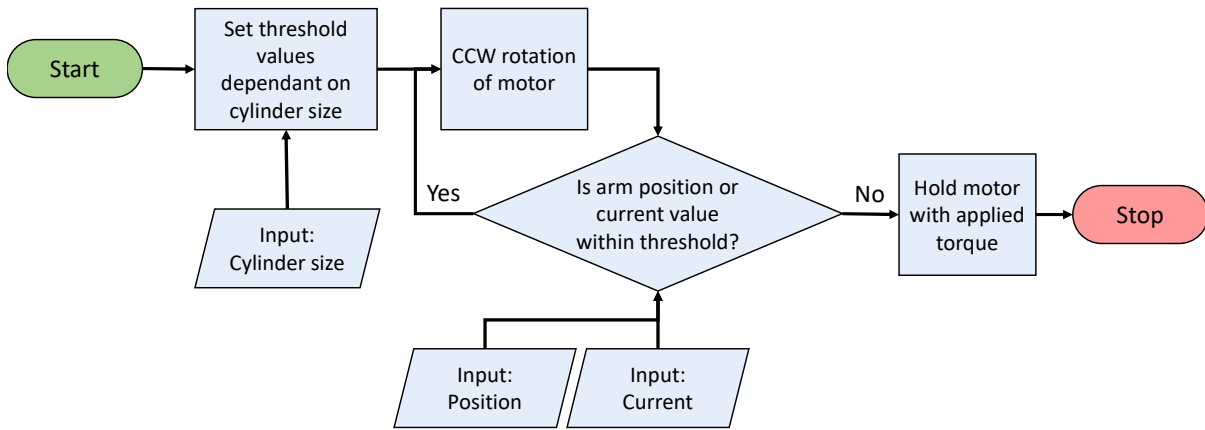


Figure 5.21: *Flow chart of the retraction of the gripper arms*

### 5.3.2 Sensor threshold values

Both sensors would work together with threshold values, determining where and when the arm should stop extending or retracting. These threshold values were found by empirical testing with and without the target grasped. The different sensors to first interact were dependant on the actuation of the arms, if they were retracting or extending. If extending, the position sensor should reach the threshold first, not applying too much stress and potential fatigue on the mechanical stop, i.e. the 3D-printed base. When extending, the current sensor is working as a backup if the position sensor should fail, but with a lower threshold than for the retraction on the target. For the retraction of the arms on the target, the arms position, when applied sufficient force, could be different for each grasping due to a potential change in grasping pose and target placement. Here, the current sensor is used as the main stopping threshold as the gripping force on the target is of utmost importance.

While testing to find the maximum threshold current, i.e. the applied force on the target to get sufficient friction force, the motor started spin within its mounting due to the double-D socket in the base being rounded from excessive stress. A threshold current value just below the limit of spin on the motor gave sufficient force to be able to lift the target. The maximum threshold current value when extending the arms was found by observing the current value while extending. The threshold value was set somewhat higher than the minimum required for the arms not to stop prematurely.

### 5.3.3 Motor control

The DC motor driver was controlled by PWM and two direction pins. The direction pins were set to high and low, and vice versa to control the rotational direction of the motor. The PWM was set to 20 kHz to be at the end of the audible range, this was also the maximum PWM input frequency of the motor driver [62]. Four different control cases for the motor was established, rotation in both directions, low torque hold and high torque hold. The different control cases can be seen in Table 5.1 and shows how the direction pins were set and the motor speed for each case. The motor speed represents the PWM duty cycle in percentage.

A high torque braking for the motor was deemed necessary to maintain a sufficient gripping force while the arms were still. At first the low torque hold was applied with braking to ground, i.e. both direction pins low. This was not adequate and a torque had to be constantly applied. By setting the motor to apply force with a smaller speed setting in the same direction as for retraction the motor would brake with sufficient force to maintain the gripping force. With a small speed setting

Table 5.1: *Motor control on direction pins and speed*

	CCW for retraction	CW for extension	Low torque hold	High CCW torque hold
Pin 11	Low	High	Low	Low
Pin 13	High	Low	Low	High
Speed	30%	20%	0%	9%

the motor would not start to spin, but act as a brake for the opposite direction.

## 5.4 Complete prototype

The gripper assembly could now be attached to the industrial robot with the control system running and actuating the gripper in a controlled manner within threshold values.

### 5.4.1 Mounting onto industrial robot

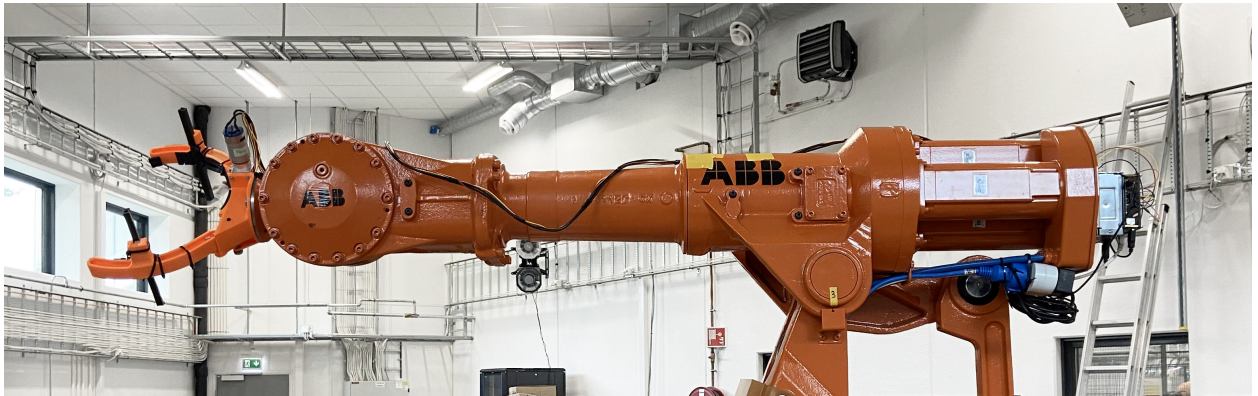
Both an ethernet and power cable had to be drawn up to the industrial robot, to be able to communicate with and power the gripper. On the industrial robot with track, the cables were drawn inside the cable track arrangement and through the robot arm's main vertical body as seen in Figure 5.22b, the blue cable is the power cable. With the cables managed in this way the robot could navigate without any constraint made from the cables. The power connections for the gripper, i.e. power supply for DC motor and RPi can be seen in Figure 5.22d with the use of a multi power outlet from the power cable.

The gripper control unit with RPi, Arduino, DC motor driver w/power source can be seen mounted on the industrial robot in Figure 5.22c. The 12V power source was mounted to the robot using double sided tape. The RPi, Arduino and motor driver were mounted to the power source using double sided tape and cable ties. From the control components, the different signal and power cables to the gripper were jointed and stretched, with tape holding it in place at the gripper and at the industrial robot. The placement of the tape on the robot can be seen with the yellow tape just above the right "ABB"-logo in Figure 5.22a.

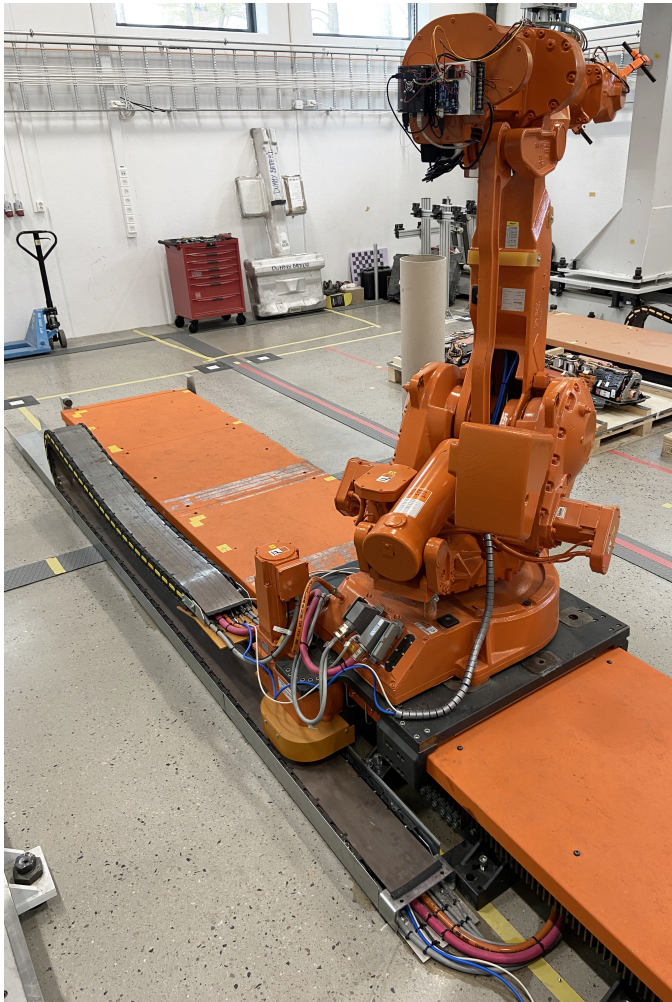
### 5.4.2 Gripped target

The grasped, picked and lifted target can be seen in Figures 5.23a-5.23b in both vertical and horizontal pose. With the lifted target the industrial robot was manually navigated with multiple poses of the target, and with rapid and sudden movements to verify a stable and firm grip. The manual navigation included greater and more rapid movements of the target than what was expected with the automatic navigation using MoveIt due to the smoothness from the motion planning. A limited automatic speed of the industrial robot was a requirement for use in the lab, this reduced the end-effector maximum speed with 75% to 250 mm/s [63].

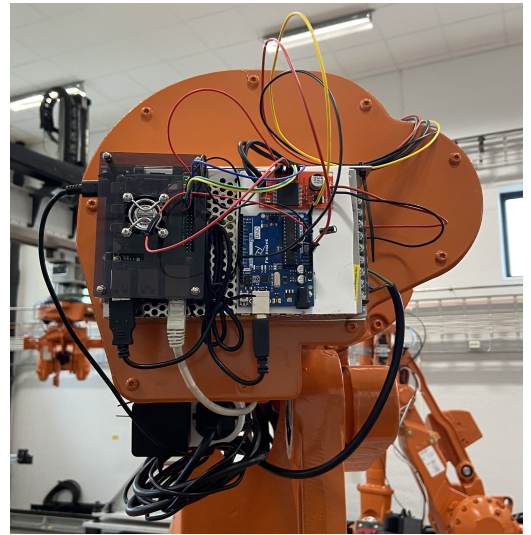




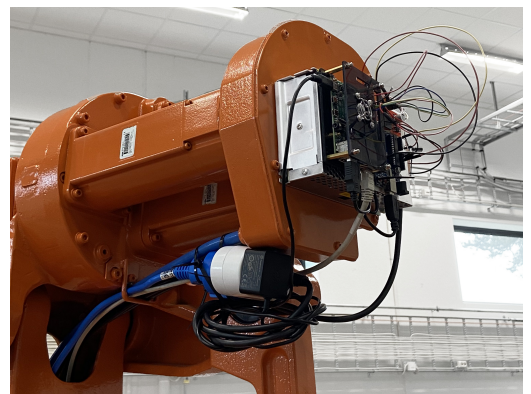
(a) Side view of robot with complete gripper assembly mounted



(b) Cable tray at rear of robot



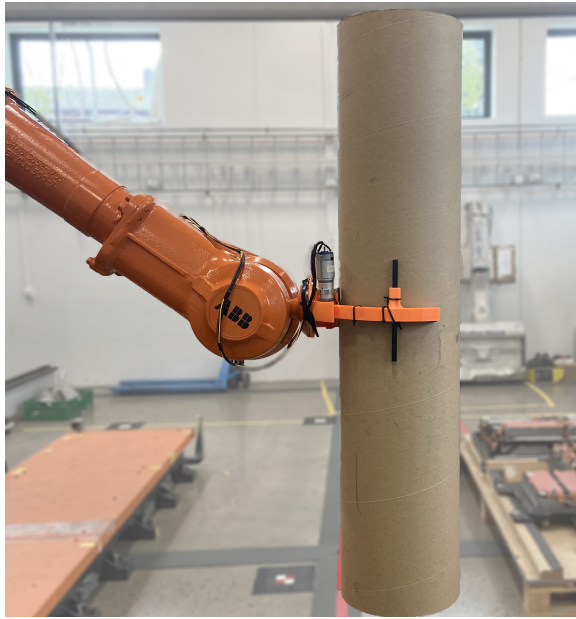
(c) Gripper control mounted



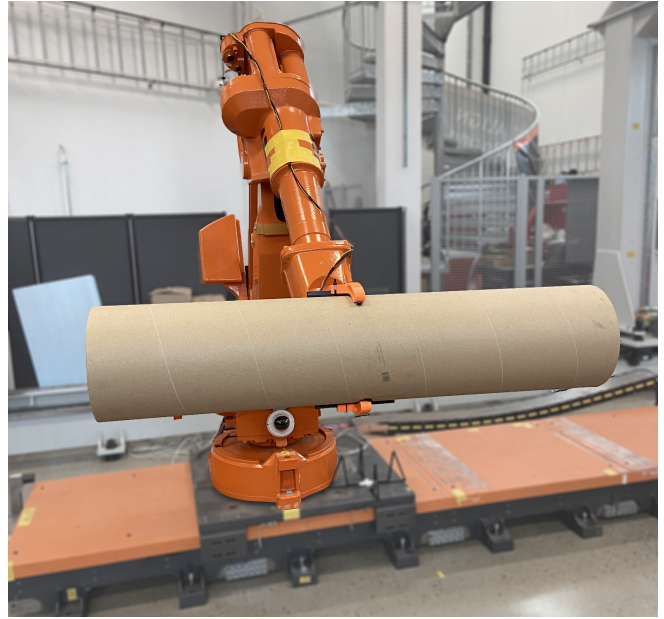
(d) Gripper control power source

Figure 5.22: Industrial robot with complete gripper assembly mounted





(a) *Side view of vertically lifted target*



(b) *Front view of horizontally lifted target*

Figure 5.23: *Lifted target*

With the target gripped near dead center with the object detection having an accuracy of 4.8 mm, minimum torque would be statically applied due to the equilibrium as performed in the tests seen in Figures 5.23a-5.23b. However, to verify the integrity and strength of the gripper design a longitudinal offset from the centre of the target larger than the accuracy of the object detection was applied. In Figure 5.24, the target was lifted with a longitudinal offset of 20 mm, and tested with the gripper on top. With the gripper on top, the target objects mass was pulling it away from the base, reducing the longitudinal rigidity. Less rigidity caused some slack in the gripping of the target, but not significant in regards to creating issues with obstacles. In addition, the gripper would not be in this position relative to the target at all times while navigating the robot.

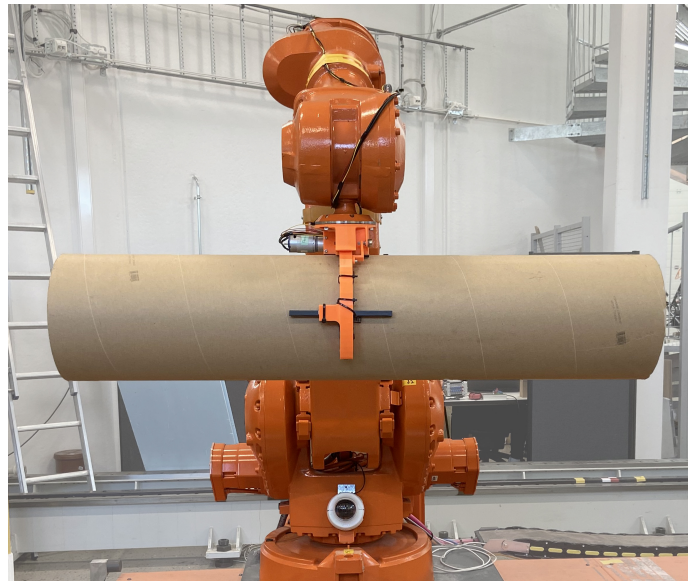


Figure 5.24: *Target lifted from top*

# Chapter 6

## Autonomous pick-and-place

The previous chapters explained cylinder segmentation, mapping of the environment and the development of a custom gripper. The last task of this thesis is to combine all these topics to be able to perform a collision-free autonomous pick-and-place procedure. This chapter will first explain how the robot model was configured in MoveIt, integrating the gripper to be used for collision checking. Further on, the concepts of navigating the robot will be explained, followed by the system integration explaining the technical details on the integration in ROS. Lastly, the results on autonomous pick-and-place with obstacle avoidance will be presented.

### 6.1 Building the robot model

Before the pick-and-place operation could begin, the robot had to be configured correctly in MoveIt. That meant adding the custom gripper, defining the new base link, end-effector link and self collision detection as well as setting up the inverse kinematic solver. A robot model of the ABB IRB4400 itself was already configured and made available by ABB. MoveIt has a GUI called MoveIt Setup Assistant, which is used to configure robot models. This program was used to add the gripper to the robot model.

In the lab, there were two robots connected to the same computer. This complicates the use of ROS and MoveIt because the default setup does not account for the utilization of multiple separate robotic systems. This can cause nodes with identical names to collide when doing similar operations on both robots simultaneously. This would likely become an issue as another group was working on the other robot in the lab at the time of this project. By giving them different names and using namespaces for the ROS nodes common for both robots, the issue was prevented. `p26_lefty` was used as our robot name and as namespace for all necessary ROS nodes in this project. `p26` is the project number given from the university. This is a safe approach to avoid node names colliding. For example, when controlling a robot through MoveIt the `move_group` topic needs to be launched. If two users launches `move_group` without a namespace, the first will crash while the most recently started node will run. With a namespace, the ROS node will now be `p26_lefty/move_group` which is independent and does not collide with other similar nodes.

#### 6.1.1 Unified Robot Description Format

Unified Robot Description Format (URDF) is a file format in ROS, which represent different parameters of a robot model, such as links and joints [41]. A chain of these kinematic joints including corresponding parent and child links combines with the robots geometric and collision model and meshes. The URDF also implies the different joint limits and facilities for internal collision detection for the robots links and joints. Here the different use of the geometric and collision mesh applies,

where the collision mesh is preferably much less apprehensive in details, thereby smaller in size. This reduces the required processing power when performing collision detection.

When combining different robots or adding tools to a robot model, xacro can be used in a simple and clean manner to make an updated combined URDF-file. Xacro is a macro XML language that performs simple math, constants and macros [64], which is beneficial when adding e.g. end-effectors with offset origin including coordinates and Cartesian's angles relative to the specified parent link. With URDF, all joints and links are dummies and only specifies their relative position and limitations, it is desired to combine these into groups and configurations semantically.

### 6.1.2 Semantic Robot Description Format

Semantic Robot Description Format (SRDF) covers semantic information about the robot which is not covered in the URDF. The SRDF file tells MoveIt how to use the URDF file correctly. Therefore, it holds information such as passive joints, group states, base and tip link and self collision detection. Group states can be a state for a group of joints, for example a group state called `all_zero` was used as an initial state in this project. Passive joints are a list of joints without any form of actuation.

### 6.1.3 Base link and end-effector link

MoveIt needs to know the base link and the end-effector link on the robot. The base link was in this case `lefty_track_left` which is the part of the track where the robot is placed when in the initial position. The end-effector on the other hand, is the tool, or the gripper of the robot, called `lefty_tool`. The two links can be seen in Figure 6.1 where `lefty_track_left` is red and `lefty_tool` is white.

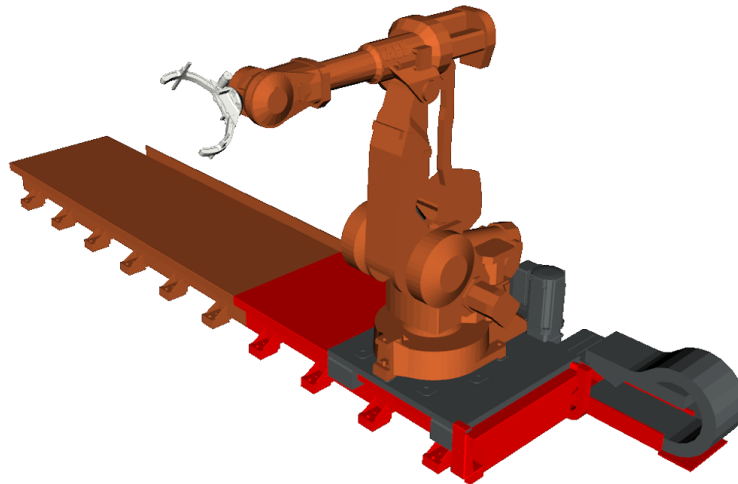


Figure 6.1: *Base link (red) and end-effector (white)*

### 6.1.4 Collision detection

Self collision detection is an important step to a collision-free motion planning. In the MoveIt Setup Assistant, a collision matrix could be defined. This matrix is used to determine the links which are able to collide, thus checking for collision between these. By removing collision detection between links unable to collide, less computational resources are required. By default, the robot's self collision matrix is setup for the robot to avoid collision between any parts. A problem encountered with this setup was that MoveIt could not plan a path where the robot moved on the track because collision

was detected between the track and the base of the robot. Therefore, collision detection was disabled between the base of the robot called `lefty_base_link` and the track which was divided into 4 links: `lefty_track_left`, `lefty_track_mid_1`, `lefty_track_mid_2` and `lefty_track_mid_3`. These names came directly from the xacro models from ABB, with "lefty\_" as a prefix. Several other parts on the robot model was not able to collide and was therefore excluded from collision detection. Appendix A.1 shows the full collision matrix from MoveIt Setup Assistant.

The geometries used for collision detection in MoveIt were defined by CAD-models consisting of different STL-files. By utilizing distinct collision object geometries with reduced level of details and comprehensiveness the computational cost of collision detection was further reduced. From ABB, a collision geometry was given of the industrial robot as seen in Figure 6.3 with a slightly larger volume than for the original model seen in Figure 6.1. The larger volume establishes a safety margin when performing collision detection. For the gripper such a geometry with a greater volume and less details was made. The embracing collision geometry around the more detailed version can be seen in Figure 6.2 and placed on the robot in Figure 6.3. Notice how the gripper collision geometry is not covering the top of the base. This was intentionally performed to acquire sufficient control with the collision detection while grasping the cylinder. With the base covering the top, the minimum allowable planning distance was too long from the gripper to the target without it colliding, resulting in a imperfect grip. Once during testing, one of the stabilizing rods collided with an obstacle. It was clear that the rod was not aligned correctly on the physical model, not corresponding to the collision geometry.



Figure 6.2: *Collision geometry of the gripper embracing the more detailed CAD-model*

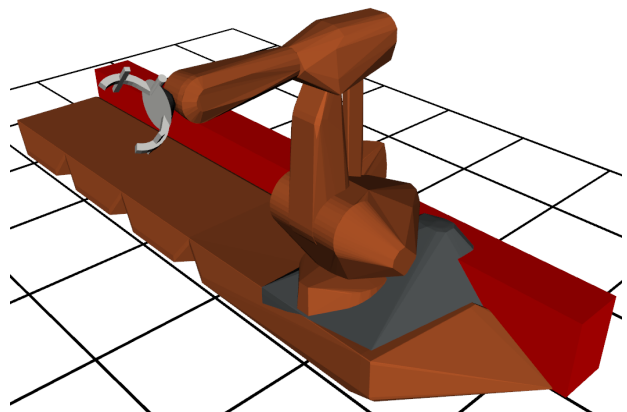


Figure 6.3: *Collision model of the robot*

### 6.1.5 Inverse kinematics solver

After defining the collision matrix, the next step was to define what kinematic solver to use. A widely used plugin-based solver for MoveIt is based on IKFast [48, p. 6]. Unlike many kinematic solvers, IKFast analytically solves kinematics equations and generates code for later use. This results in stable and fast solutions, and is therefore the choice for most robotic applications in MoveIt [65].

## 6.2 Navigation and locomotion of the robot

This section will explain the concepts of navigating the robot to pick-and-place the target with the optimal pose.

### 6.2.1 Grasping

The closest point to the robot on the central periphery of the cylindrical target was assumed to be the most efficient way to grasp. This point would be the tip of the green arrow on Figure 6.4. All the points on this periphery could be reached by approaching the target with a tool orientation perpendicular to the direction vector of the target which is illustrated as a blue arrow in Figure 6.4.

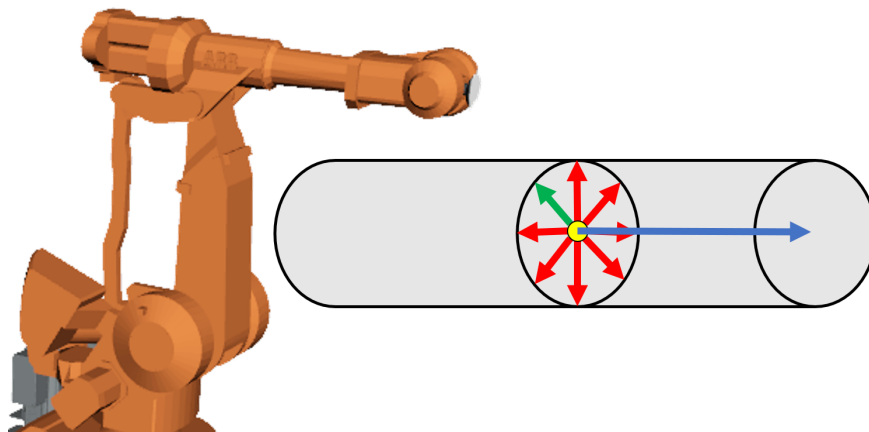


Figure 6.4: *Up vector candidates*

### 6.2.2 Orientation

MoveIt uses quaternions to represent the orientation of the tool. To achieve the desired quaternions to approach the target perpendicularly, a direction vector is not sufficient. This is where the up vector comes in. The red and green arrows in Figure 6.4 are up vectors, and in this case, the optimal up vector would be the green vector which is pointing towards the tool. These up vectors could be achieved by calculating the cross product between the direction vector and any vector not parallel to the itself. Lets call such a vector the dummy vector. This dummy vector was set to be a vertical unit vector. However, this could cause an issue if the direction vector was perfectly vertical, i.e. being parallel to the dummy vector. In this case, the cross product would be equal to zero. Therefore, if the direction vector was close to being vertical, the dummy vector would be set to be horizontal pointing in positive x-direction, to avoid this issue. See the code below where `v0` is the dummy vector, `D` is the direction vector of the cylinder, `D[2]` is the z-component of `D` while `v1` is a unit vector perpendicular to the direction vector generated from the cross product between `D` and `v0`.

```
v0 = [0,0,1]
if D[2] > 0.9: # if z component is larger than 0.9
    v0 = [1,0,0]

u = np.cross(D, v0)
u = normalize(u)
```

A list of candidate vectors was then created by rotating  $\mathbf{u}$  around  $\mathbf{D}$  with a resolution of  $1^\circ$  between each, resulting in 360 possible up vectors. To perform such a transformation on the vector, an adaptation of Rodrigues' rotation formula [66] was applied. Rodrigues' rotation formula is an efficient algorithm for rotating a vector in space. In this case, the vector to be rotated is the up vector,  $\vec{u}$ , and the axis of rotation is the direction vector  $\vec{D}$ . Firstly, the vector  $\vec{u}$  has to be decomposed into components parallel and perpendicular to  $\vec{D}$ :

$$\vec{u} = \vec{u}_{\parallel} + \vec{u}_{\perp} \quad (6.1)$$

The fact that they are perpendicular to each other makes the calculations less complex as the parallel component disappears.

$$\vec{u}_{rot} = \vec{u}_{\perp rot} = \vec{u} \cdot \cos(\theta) + (\vec{u} \times \vec{D}) \cdot \sin(\theta) \quad (6.2)$$

where,

	Description
$u_{rot}$	Rotated up vector
$u$	Initial up vector
$\theta$	Angle of rotation
$D$	Direction vector of the cylinder, and axis of rotation in this case.

The candidates could now be generated with  $\theta$  varying from  $0-2\pi$  to generate up vectors in all directions. They were represented as a class object of type `GraspingPointCandidate` which held two variables:

- $\mathbf{u}$  - The rotated vector after Rodrigues' algorithm
- $\mathbf{d}$  - Distance between the tool and the center of mass plus  $\mathbf{u}$

The member variable  $\mathbf{d}$  was calculated as shown in equation 6.3

$$d = |P_{tool} - (P_{com} + u)| \quad (6.3)$$

The chosen candidate is the one with the shortest distance,  $\mathbf{d}$ . This candidate's up vector would then be the one pointing closest to the robot's tool. From here, this up vector will be referred to as  $\vec{U}$ .

At this point, the direction vector,  $\vec{D}$  and the up vector,  $\vec{U}$ , of the cylinder was known, meaning a fully defined rotation. In terms of the robot direction, the cylinder's up vector,  $\vec{U}$ , will represent the direction vector of the robot's tool, while the cylinder's direction vector will represent the up vector of the robot's tool. For the following calculations,  $\vec{U}$  and  $\vec{D}$  will represent the cylinder's up vector and direction vector while the goal of the calculations is to find the desired robot orientation represented in quaternions. Two ways of converting such vectors to quaternions are by going through Euler angles or rotation matrices. By going through Euler angles, the desired yaw,  $\psi$ , and pitch,  $\theta$ ,



of the robot's tool could be calculated by simple trigonometry equations as shown in equation 6.4 and 6.5 as functions of the cylinder's up vector,  $\vec{U}$ .

$$\psi = \text{atan2}(U_y, U_x) \quad (6.4)$$

$$\theta = \text{asin}(U_z) \quad (6.5)$$

Finally, for the roll angle, a vector parallel to the ground and perpendicular to the direction vector is created:

$$\vec{W}_0 = [-U_y, U_x, 0] \quad (6.6)$$

The cross product between the direction vector,  $\vec{D}$ , and this horizontal vector,  $\vec{W}_0$ , will be equal to a reference up vector,  $\vec{U}_0$ , (equation 6.7) representing no roll.

$$\vec{U}_0 = \vec{W}_0 \times \vec{D} \quad (6.7)$$

This means that if  $\vec{U}$  is equal to  $\vec{U}_0$ , the roll angle is equal to zero. However, if they are not equal, the angle between them represents the roll angle,  $\phi$ , which is then calculated as shown in equation 6.8

$$\phi = \text{atan2}\left(\frac{\vec{W}_0 \cdot \vec{D}}{|\vec{W}_0|}, \frac{\vec{U}_0 \cdot \vec{D}}{|\vec{U}_0|}\right) \quad (6.8)$$

Now that the desired orientation of the robot's tool is defined in terms of Euler angles, the conversion to quaternions can be done. The approach is shown in the pseudocode below [67]:

```

cy = cos(yaw * 0.5)
sy = sin(yaw * 0.5)
cp = cos(pitch * 0.5)
sp = sin(pitch * 0.5)
cr = cos(roll * 0.5)
sr = sin(roll * 0.5)

qw = cr * cp * cy + sr * sp * sy
qx = sr * cp * cy - cr * sp * sy
qy = cr * sp * cy + sr * cp * sy
qz = cr * cp * sy - sr * sp * cy

```

The results from the calculations above was tested through the simulated environment in RViz, and in that way, it could be validated virtually before testing it in the real world, leading to safe and controlled testing.

### 6.2.3 Motion planning

MoveIt supports several different types of planners. The most commonly used planner is called Open Motion Planning Library (OMPL) [68, 69, 70] which is an open-source library. The creators of OMPL define motion planning as the following:

*Motion planning is the problem of finding a continuous path that connects a given start state of a robotic system to a given goal region for that system, such that the path satisfies a set of constraints (e.g., collision avoidance, bounded forces, bounded acceleration). [68]*

OMPL is directly integrated in MoveIt and it is the most supported motion planner. It is based on randomized, abstract motion planners, meaning that the planner has no conception of the robot. However, MoveIt configures OMPL such that it can work with complex problems in robotics [48, p. 7].

### 6.2.4 Moving the robot with MoveIt

MoveIt has many tools for moving a robot through a script with great support for both Python and C++. According to [48, p. 5], the Python API is recommended for building applications and scripting demos by using the `moveit_commander` package. Python was therefore chosen as the preferred language when developing this pick-and-place procedure. The most important class objects for moving the robot are listed below:

- `RobotCommander` - An outer level interface to the robot.
- `PlanningSceneInterface` - An interface to the representation of the world surrounding the robot.
- `MoveGroupCommander` - An interface for a group of joints, i.e. all robot joints. Used to plan and execute motions on the robot.

To move the robot, a start pose was set based on the current pose of the robot. Then a goal state was set to the desired pose. By calling `group.go()`, where `group` was a `MoveGroupCommander` object, MoveIt would plan the motion with OMPL, and the robot would start to move. When the motion plan was done executing, a stop function was called to make sure there were no residual movement. This is summarized by the code snippet below:

```
group.set_start_state(state)
group.set_pose_target(pose_goal)
group.go(wait=True)
group.stop()
```

## 6.3 System integration

All the blocks under the WBS diagram in Figure 1.2 has been explained earlier in the report, and to perform an autonomous pick-and-place procedure, all the work had to be integrated as one complete system. Mechanically, the gripper was mounted on the robot as shown in chapter 5. It was then connected to the rack computer. ROS was used to combine the perception systems with the industrial robot and the gripper. First, the functional description of the program will be presented, followed by a technical explanation of the ROS nodes and topics.

### 6.3.1 Pick-and-place functional description

A state machine was implemented to have control over the program flow. Every procedure starts with the extension of the gripper to make sure that the gripper is in the correct position. Further on comes object detection and obstacle mapping. The industrial robot (IR) is then moved to the target's pose and the gripper is closed. For safety reasons, the obstacle map is refreshed so that the system is updated in case of a changed environment due to human intervention or obstacles tipping over if a collision avoidance failure was to happen. The industrial robot then moves to the goal pose, and the target is then released by extending the gripper. A pick-and-place procedure is then finished, and the program can either stop or be repeated several times. The transitions are explained in Figure 6.5.

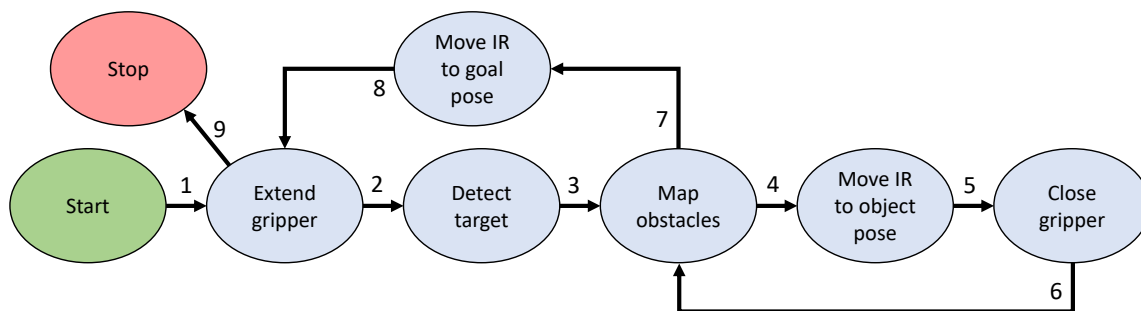


Figure 6.5: *State machine*

#### Transitions

1. All ROS nodes are done launching
2. Gripper is extended
3. Target is detected
4. Obstacles are mapped
5. IR is moved to objects pose
6. Gripper is closed
7. Obstacles are mapped
8. IR is moved to goal pose
9. Program is physically stopped. Transition 2 can also be initiated manually

### 6.3.2 ROS nodes and topics

7 ROS packages was created to perform the task at hand in addition to the packages from Aalerud et al. [5, 6, 7] and ABB. The blue boxes in Figure 6.6 shows ROS nodes running during a pick-and-place procedure. The texts on the arrows between these boxes are the ROS topic they use to talk to each other. Black arrows are unidirectional while orange arrows represents bidirectional communication. Note that this is not the complete schematic of nodes and topics. There are lots of nodes running in the background talking on many different topics, but Figure 6.6 is simplified to include the core components important to understand the software behind the pick-and-place procedure. Further on, the nodes and topics from this figure are explained in Table 6.1 and Table 6.2.

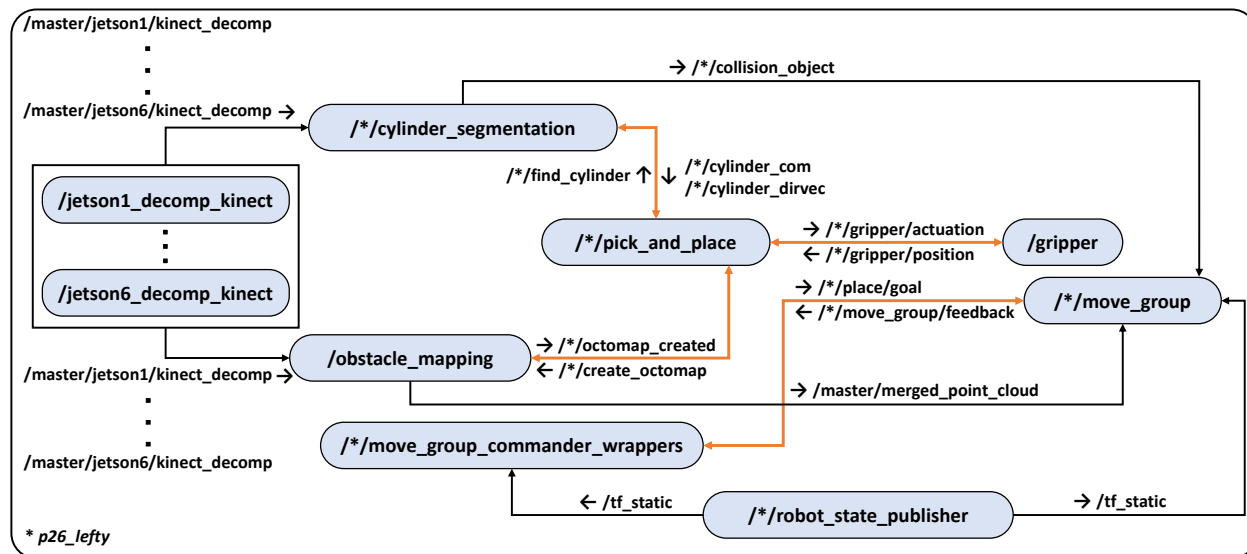


Figure 6.6: *Simplified schematic of the ROS topics*

Table 6.1: *Explanation of ROS nodes. \*p26\_lefty*

ROS node	Description
/jetson[nr]_decomp_kinect	Jetson boards publishing point clouds from the Kinect sensors
/*/cylinder_segmentation	Cylinder segmentation using point clouds and RANSAC.
/*/obstacle_mapping	Merges point clouds, filters them and creates an octomap of the obstacles
/*/pick_and_place	Controls the pick-and-place procedure by sending poses to /*/move_group_commander_wrappers and actuation commands to the gripper
/*/move_group_commander_wrappers	A wrapper for the functionalities provided by MoveIt
/gripper	Gripper actuation and sensing
/*/move_group	Explained in chapter 3.2.3
/*/robot_state_publisher	Takes the robot joint positions obtained from internal sensors in the robot as input, and outputs the pose of all the robot links

Table 6.2: *Explanation of ROS topics. \*p26\_lefty*

ROS topic	Description	Type
/master/jetson[nr]/kinect_decomp	Point cloud from sensor node [nr]. [nr] is a number from 1-6	sensor_msgs/PointCloud2
/*/collision_object	Used for cylinder object	moveit_msgs/CollisionObject
/*/find_cylinder	Initiates the cylinder segmentation	std_msgs/Bool
/*/cylinder_com	Cylinder center of mass	geometry_msgs/Point
/*/cylinder_dirvec	Cylinder direction vector	geometry_msgs/Point
/*/gripper/actuation	Used for gripper setpoints	std_msgs/UInt8
/*/gripper/position	Info on the gripper position	std_msgs/UInt8
/*/octomap_created	High when octomap has been created	std_msgs/Bool
/*/create_octomap	Initiates the obstacle mapping	std_msgs/Bool
/*/place/goal	Goal position	moveit_msgs/PlaceActionGoal
/*/move_group/feedback	Information on what the action server is doing	moveit_msgs/MoveGroupActionFeedback
/master/merged_point_cloud	Merged point cloud for the octomap	sensor_msgs/PointCloud2
/tf_static	Poses for all the robot links	tf2_msgs/TFMessage

## 6.4 Results

This section will present the results from gripping followed by the complete pick-and-place procedure results.

### 6.4.1 Gripper

When grasping the target, it was observed in approximately 1 out of 5 times that the gripper could not establish an ideal rigidity of the target making it wobbly within the gripper base. The issue was prominent while the detection was at its upmost inaccurate, resulting in space between the gripper base and target at grasping. This distance minimized the utilization of the rigidity properties of the base. However, the target was grasped with sufficient rigidity to avoid collisions and to be able to place it at a goal position.

From the beginning it was determined not to use hard surfaces as stand for the cylinder, making a suspension weaker than the cylinder, gripper and robot, deliberately compressing the stand if something were to fail. The inaccuracy of the object detection could sometimes move the gripper base into the cylinder. With some suspension in the stand, this was never an issue with the cylinder placed horizontally. The gripper base encountered some stress in this situation, compressed onto the cylinder, but endured it without suffering during testing. The compressed stand from one test can be seen in Figure 6.7.



Figure 6.7: *Object stand compressed from applied robot force*

#### 6.4.2 Pick-and-place

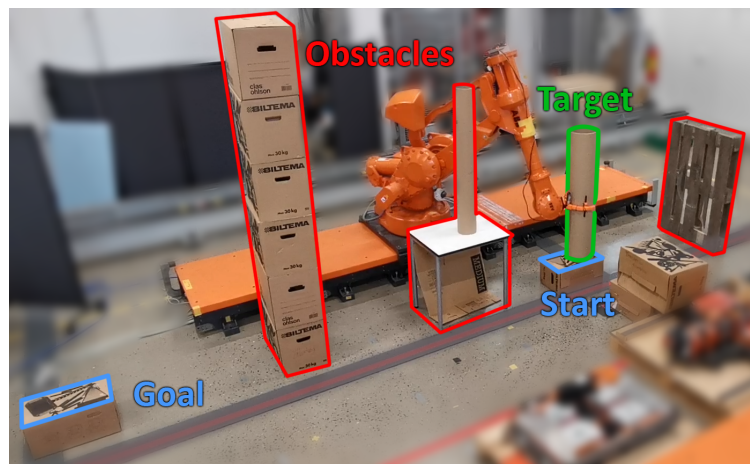


Figure 6.8: *Pick-and-place test conditions*

Merging all the work presented until now resulted in a consistent and versatile pick-and-place procedure. To test the consistency of the system, 30 tests was performed with the setup seen in Figure 6.8. The test was considered successful if the target was placed on the blue goal area (31 cm x 48 cm) without falling over. The tests was divided into 3 different sections, vertical, horizontal and at an angle, see Figure 6.9, where all three was tested 10 times each. The target was placed on cardboard boxes to avoid breaking the gripper or the target object. This could be a problem for the horizontal case especially, where the target or the gripper could be crushed if the detection algorithm detected the target slightly too low and the target was on the hard floor. In addition, the detection zone of the target was set to 5 cm above the ground as explained in chapter 4.2.2.

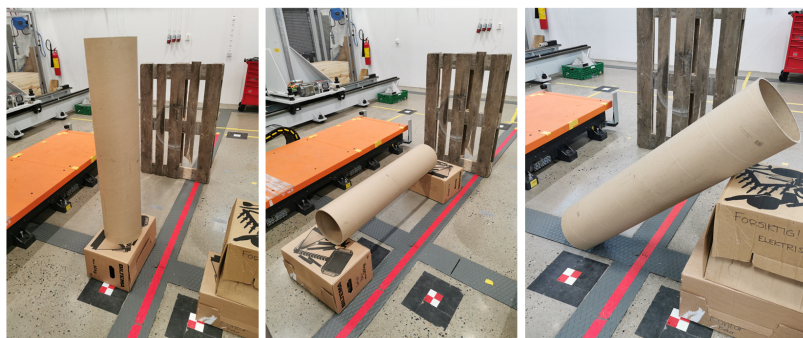


Figure 6.9: *Vertical, horizontal and angled test conditions*



The 30 tests displayed no collisions and a 100% success rate for the repositioning of the target. An example from the vertical case is shown in Figure 6.10. To increase the complexity with regards to motion planning and obstacle mapping, a bridge was made for the robot and target to move underneath, see Figure 6.11. Both tests were performed multiple times without collisions and with successful gripping of the target. However, sometimes the precision of the object detection and localization caused a slightly skewed gripping causing the target to be a bit wobbly. The errors propagated through the pick-and-place procedure resulting in a lower precision with regards to target placement. The skewed gripping did not cause any collisions with obstacles during testing.

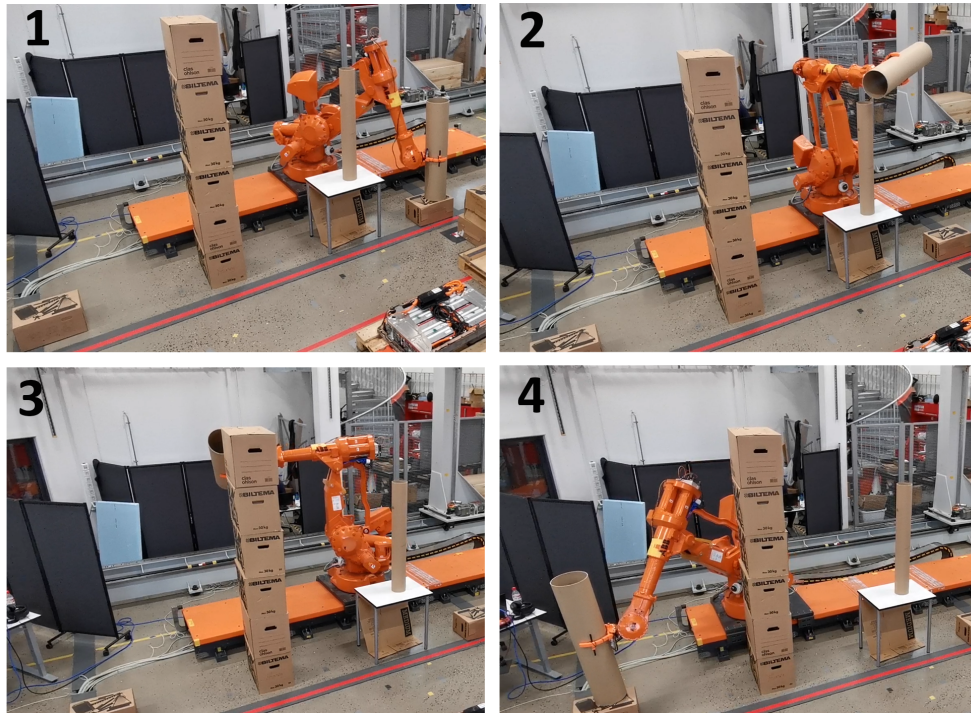
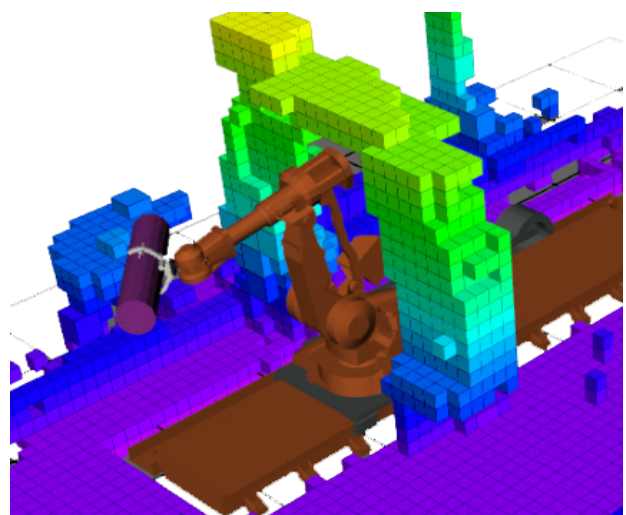


Figure 6.10: 4 images showing the sequence of a pick-and-place procedure with a vertical target starting pose



(a) Photo of the test



(b) RViz visualization of the test

Figure 6.11: Pick-and-place test with a bridge over the robot

# Chapter 7

## Discussions

The autonomous pick-and-place procedure carried out in this project worked well as a prototype, and included both advantages and disadvantages compared to other methods. As a prototype, it had some improvement potential regarding different parts of the autonomous pick-and-place procedure within perception, gripping and motion planning. This chapter will discuss these and propose methods to solve these issues found for further work.

### 7.1 Object detection and localization

The results from the object detection and localization shows a larger error in one specific direction depending on the cylindrical target's pose. Chapter 4.2.4 shows that when the target was in a vertical position, the error in z-direction was significantly larger than the error in x- and y-direction. However, when the target was in a horizontal position, the y-error was largest. What these two had in common was that the largest error was found in the longitudinal direction of the target. On the other hand, the error in the transverse plane was low. Fortunately, the transverse plane proved to be the most critical when gripping the target. Nevertheless, the error in the longitudinal direction was massively decreased by the method of averaging based on multiple frames. Thus, the error in both the transverse and longitudinal direction was considered well within what was required to be able to grasp the target with the proposed gripper.

The accuracy in the transverse plane shows the power of the RANSAC method. The RANSAC method did not consider the length of the cylinder and the larger error in longitudinal direction may question the sphere filter explained in chapter 4.2.3. If one were to improve the error in the longitudinal direction, the error might be low enough for 1 frame to be sufficient. However, the work-around of taking the average of multiple frames made it possible to perform the task at hand comfortably. The disadvantage with this method was of course computational cost, but since the segmentation only happened once before the pick-and-place procedure started, this was not considered a problem of significance for this application. It was considered more effectively to have one accurate segmentation at the start, and let MoveIt keep track of the position in real-time when manipulating the target, rather than having a real-time cylinder segmentation which would have been less accurate and computationally more expensive.

To reduce the computational cost even further, a RANSAC based plane segmentation was considered to reduce the number of measurement points in the point cloud. However, the planar segmentation was not necessary as the floor and the robot tracks are excluded by limiting the detection zone. However, it could be desirable to remove planar surfaces from obstacles. When testing with the target placed horizontally, it was observed that the planar segmentation removed parts of the target in the same plane as planar surfaces (e.g. from a box). This caused the algorithm to fail finding the cylindrical object. Therefore, the plane segmentation was discarded, but it is worth keeping in

mind if testing in different environments where planar surfaces creates lots of data points.

With a reliable segmentation achieved with the 26 cm cylinder, the algorithm was tested on a smaller casting pipe with a diameter of 16 cm to evaluate the algorithm's versatility. The voxel grid resolution made it hard to see the curvature of the small cylinder as shown in Figure 7.1, showing the difference in curvature of both 16 cm and 26 cm cylinder. The lack of curvature on the smaller cylinder made it difficult to distinguish it from other flat objects.

A higher resolution of the perception system would make it possible to detect smaller objects with a reliable result. There are multiple ways to establish a higher resolution. Three methods are proposed below:

1. Replacing the existing sensor nodes with higher resolution 3D sensors
2. Mounting an additional 3D sensor on the robot to get a closer look at the object to be detected (eye-in-hand)
3. Placing the nodes closer to the work area improving the resolution



Figure 7.1: *The large and the small cylinder shown from above from a photo (left) and a point cloud (right)*

These solutions would have their own disadvantages. 1. and 2. would have increased the overall cost. 2. would also increase the complexity of the system and it could introduce a potential precision issue when gathering data during robot motion as stated in [14]. Lastly, 3. would decrease the area covered by the 3D sensors.

In addition, by utilizing 2. and mounting a 3D sensor on the industrial robot, the accuracy of the object detection could be further improved. The eye-to-hand global perception system could first make a coarse detection, giving an indication of the position of the object, or points of interests if the objects are too small to be detected. Then moving the robot within the area of interest, making a closer sweep and scan giving an updated point cloud preferably with lesser noise and higher resolution of the object to be grasped. The updated point cloud would establish better conditions for the detection algorithm and would also enable detection of smaller objects. Otherwise, the eye-to-hand system would only cover the obstacle mapping, specifically utilizing the eye-in-hand for object detection.

A 3D sensor placed on the robot would be exposed to more of the potential filth and dust often occurring in an industrial environment, likely reducing the quality of the sensed data over time. The eye-to-hand 3D sensors would not be similarly exposed, placed above and away from the most harsh parts of the environment. Eye-in-hand would also induce a more lumpy geometry on the robot, reducing its agility and access in tight spaces.

## 7.2 Obstacle mapping

With the placement of the 3D sensors, there was only coverage from top, making it difficult to evaluate the obstacle's geometrical properties from underneath. This never led to any collisions during testing. A bridge for the robot to move under is shown in Figure 6.11. Since the roof used in the test was thin, the points mapped covered the whole roof avoiding any motion planning through

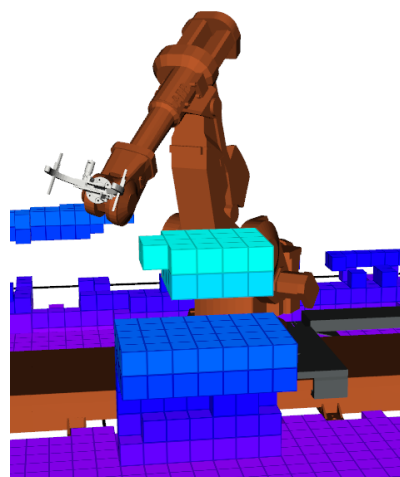


it. If the roof was a thick box, the box would only be covered at top, thus be considered hollow, enabling motion planning inside it.

As the 3D sensors utilize time-of-flight principle with IR light they are dependent on the parts being somewhat reflective. On dark objects intensity issues were prominent, some not appearing at all in the occupancy map. Figure 7.2 shows the robot colliding with a black chair not detected and present in the occupancy map. It also shows how the combination of low resolution, dark color and the intensity filter neglects the thin swivel and back on the chair in the occupancy map. The size and color of obstacles should therefore be a consideration for them to be detected.



(a) Photo of the robot colliding with the chair



(b) Parts of the chair not present in the occupancy map

Figure 7.2: Issue with dark objects

The gripper control unit seen in Figure 7.3 mounted at the rear of the industrial robot was not added to the robot collision geometry. Sometimes the control unit was considered an obstacle in the occupancy map making motion planning more challenging. The robot would therefore move around the spurious points from the control unit it perceived as obstacles, causing interesting detours. The issue was encountered in approximately 1 out of 20 procedures, and the robot managed to plan around the point every time the issue was encountered. The problem could have been solved by creating an STL file of the control unit, then adding it to the corresponding robot link the same way the gripper was added when building the robot model in chapter 6.1.

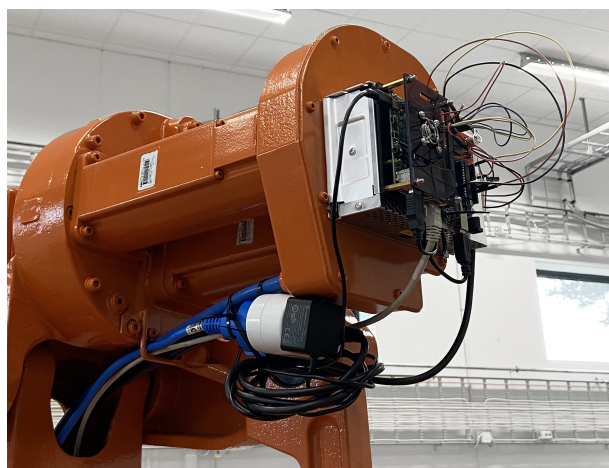


Figure 7.3: Gripper control unit

### 7.3 Gripper

The complete gripper assembly gave sufficient gripping force and stabilization to be able to lift a cylinder with a radius of 26 cm from both horizontal and vertical picking position. However, while finding the maximum applicable motor torque, i.e. current threshold, and verifying the assembly while grasping, the flat plastic part of the double-D socket in the base broke, the O-socket and motor mount can be seen in Figure 7.4. The gripping force could have been higher if the motor did not spin within the socket, increasing stability even further. The double-D socket were presumed to be a solid piece holding a fair amount of torque, thus implying an even more rugged construction on the rest of the assembly. Now, with only the motor bracket remaining, the maximum applicable force is somewhat lower, but still sufficient to obtain friction and stability to lift and move the target in a rigid and safe manner.

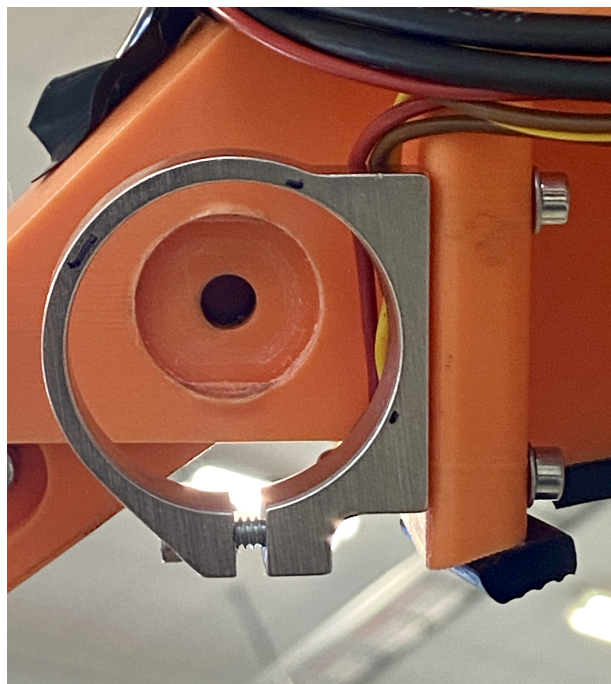


Figure 7.4: *Motor bracket and former D-socket*

With a further reduction of grip, the lack of rigidity could cause the gripper to loose the target and dropping it. In addition, it could have imposed collisions with obstacles. However, the lack of rigidity never inflicted with obstacles or the robot and the target was never dropped with more than 50 pick-and-place procedures completed. Sealant strips were utilized to increase the friction between the gripper and target. However, with a slightly skewed position while grasping, especially present when a distance between the base and the target occurred, the retraction of the arms did not manage to push the cylinder towards the base, inducing less rigidity between the gripper and target. Spring loaded stabilization parts could be implemented, always pushing towards the grasped cylinder. This induce more friction and rigidity with lesser dependency of the distance to the target when grasping. Another alternative could be to redesign the arms, applying more pressure on the top of the cylinder at the start of retraction, pushing it more towards the base. Alternatively using wider or twin-acting arms.

Ordinary rigid-body gripper concepts with hard materials were chosen to be proceeded with. Neither is the ABB IRB4400 industrial robot considered soft, e.g. being able to knock a person down without noticing and crushing body parts between different links and joints. However, with the development of this project the overall goal is to achieve a more alert and less hazardous robotic cell, especially for surroundings, imposing a safer robotic system, even without soft robotics hardware.

### 7.4 Motion Planning

Due to the static obstacles used in this project, dynamic motion planning was not required to prove the use of multiple 3D sensors to achieve an autonomous pick-and-place procedure with obstacle avoidance. However, dynamic motion planning could be used as a task to further exploit and develop the autonomous possibilities for the system. Together with the SOR-filter, an octomap refresh rate

of 0.5-1 Hz was achieved. It could be worth looking into a Fast Cluster Statistical Outlier Removal (FCSOR) filter [71] to increase the frequency. FCSOR improves the time complexity compared to the more traditional SOR-filter, coming closer to a real-time obstacle mapping. The low refresh rate would not be able to catch any rapid dynamic changes in the environment to utilize dynamic motion planning without slowing down the robotic movements significantly. Also, MoveIt and OMPL are not yet compatible with continuous collision checking [72]. By utilizing this system, dynamic obstacle avoidance could be implemented by making new motion planning with new occupancy maps for every given seconds, or any traveled distance, or by making an algorithm that discovers if updated obstacles interacts with the set trajectory. However, with the motion planning taking up to multiple seconds, the system would be slower than whats reasonable for an industrial application.



# Chapter 8

## Conclusions

For safer and efficient use of robotics, it is desired to make processes autonomous with surrounding awareness. This thesis proposes a complete pipeline autonomous pick-and-place procedure including object detection and localization, obstacle avoidance, gripper development and robot control.

The experimental setup used in this project, including placement, configuration and calibration of 3D sensors with embedded intensity filtering was proposed by [5], [6] and [7]. We propose the integration of their work together with proposed perception algorithms, robot control and gripper design to one complete autonomous pick-and-place procedure. The proposed perception algorithms involved merging the point clouds from 6 RGB-D cameras of type Kinect V2 for object detection and localization and model-free obstacle mapping. Further on, a low-cost automatic gripper was developed, including actuation, sensorization, production and implementation with the experimental setup.

The empirical results together with the statistical analysis show that the proposed methodology is able to map the environment of volume of 10 m x 10 m x 5 m with lesser noise, determine the target position with accuracy of 4.8 mm and precision of 3.6 mm, and orientation with accuracy of  $0.62^\circ$  and precision of  $0.32^\circ$  from 10000 measurements. The autonomous pick-and-place procedure was tested 30 times successively with multiple obstacles and with the target object placed vertically, horizontally and angled. The tests displayed no collisions and 100% success rate on both gripping and placement of the target at a given goal position.

The proposed methods used only open-source software and it is highly applicable for industrial applications as a cheap way to make industrial robots safer and more efficient through advanced perception and control algorithms. This project shows how many components in multiple engineering disciplines could be developed and implemented in a relatively short period of time. However, object detection and localization, obstacle mapping, gripper development and motion planning are all components with room for improvements.

To further exploit the setup, achieving a higher degree of safety and modularity for pick-and-place applications, some key properties could be further developed. Within most industrial environments the scene is continuously changing, imposing multiple challenges regarding collision-free motion. Dynamic obstacle avoidance would be a valuable addition, significantly reducing the hazard of working close to an industrial robot. By also implementing a human detection algorithm as proposed in [20], an additional safety precaution could be made, e.g. maintaining a greater distance around people. With the resolution of the point cloud mapping the environment, minor objects and obstacles could not be detected. By upgrading the 3D sensors, placing them closer to the environment or introducing an eye-in-hand camera, a higher resolution could be obtained also increasing the accuracy of the perception algorithms.

# Bibliography

- [1] K. A. Iden et al. “Vær, is og andre fysiske utfordringer ved Barentshavet sørøst”. In: (2012).
- [2] Valemon. URL: <https://www.equinor.com/no/what-we-do/norwegian-continental-shelf-platforms/valemon.html>. (accessed: 15.02.2021).
- [3] Joakim (22) skal styre denne plattformen fra land. URL: <https://www.vg.no/nyheter/innenriks/i/zRKn9/joakim-22-skal-styre-denne-plattformen-fra-land>. (accessed: 15.02.2021).
- [4] Senad Burak and Fikret Veljovic. “Ergonomic Analysis and Redesign of Workspace in Order to Minimize Workers’ Workload and Optimize Their Nutrition”. eng. In: *TEM Journal* 8.2 (2019), pp. 572–576. ISSN: 2217-8309.
- [5] Atle Aalerud et al. “Industrial Environment Mapping Using Distributed Static 3D Sensor Nodes”. In: *2018 14th IEEE/ASME International Conference on Mechatronic and Embedded Systems and Applications (MESA)*. 2018, pp. 1–6. DOI: [10.1109/MESA.2018.8449203](https://doi.org/10.1109/MESA.2018.8449203).
- [6] Atle Aalerud, Joacim Dybedal, and Geir Hovland. “Automatic Calibration of an Industrial RGB-D Camera Network Using Retroreflective Fiducial Markers”. eng. In: *Sensors (Basel, Switzerland)* 19.7 (2019), p. 1561. ISSN: 1424-8220.
- [7] Joacim Dybedal, Atle Aalerud, and Geir Hovland. “Embedded Processing and Compression of 3D Sensor Data for Large Scale Industrial Environments”. eng. In: (2019). ISSN: 1424-8220. URL: <https://hdl.handle.net/11250/2648520>.
- [8] D. L. Schacter. *Psychology*. New York, NY : Worth Publishers, 2011.
- [9] M.S Shehata et al. “Video-Based Automatic Incident Detection for Smart Roads: The Outdoor Environmental Challenges Regarding False Alarms”. eng. In: *IEEE transactions on intelligent transportation systems* 9.2 (2008), pp. 349–360. ISSN: 1524-9050.
- [10] Mozhddeh Shahbazi et al. “Range camera self-calibration based on integrated bundle adjustment via joint setup with a 2d digital camera”. eng. In: *Sensors (Basel, Switzerland)* 11.9 (2011), pp. 8721–8740. ISSN: 1424-8220.
- [11] K. B. Kaldestad, G. Hovland, and D. A. Anisi. “Implementation of a Real-Time Collision Avoidance Method on a Standard Industrial Robot Controller”. In: *IEEE International Conference on Intelligent Robots and Systems, Chicago, Illinois* (2014).
- [12] S Khan, L Aragão, and J Iriarte. “A UAV-lidar system to map Amazonian rainforest and its ancient landscape transformations”. eng. In: *International journal of remote sensing* 38.8-10 (2017), pp. 2313–2330. ISSN: 0143-1161.
- [13] Atle Aalerud, Joacim Dybedal, and Dipendra Subedi. “Reshaping Field of View and Resolution with Segmented Reflectors: Bridging the Gap between Rotating and Solid-State LiDARs”. eng. In: *Sensors (Basel, Switzerland)* 20.12 (2020), p. 3388. ISSN: 1424-8220.
- [14] Ge Zhang et al. “Multi-granularity environment perception based on octree occupancy grid”. eng. In: *Multimedia tools and applications* 79.35-36 (2020), pp. 26765–26785. ISSN: 1380-7501.
- [15] Paolo Bellandi, Franco Docchio, and Giovanna Sansoni. “Roboscan: a combined 2D and 3D vision system for improved speed and flexibility in pick-and-place operation”. eng. In: *International journal of advanced manufacturing technology* 69.5 (2013), pp. 1873–1886. ISSN: 0268-3768.

- [16] Tobias Kotthaus and Georg F. Mauer. “Vision-based autonomous robot control for pick and place operations”. In: *2009 IEEE/ASME International Conference on Advanced Intelligent Mechatronics*. 2009, pp. 1851–1855. DOI: [10.1109/AIM.2009.5229792](https://doi.org/10.1109/AIM.2009.5229792).
- [17] V Lippiello, B Siciliano, and L Villani. “Position-Based Visual Servoing in Industrial Multi-robot Cells Using a Hybrid Camera Configuration”. eng. In: *IEEE transactions on robotics* 23.1 (2007), pp. 73–86. ISSN: 1552-3098.
- [18] Aksel Sveier et al. “Object Detection in Point Clouds Using Conformal Geometric Algebra”. eng. In: *Advances in applied Clifford algebras* 27.3 (2017), pp. 1961–1976. ISSN: 0188-7009.
- [19] Xinyu Wang et al. “Robot manipulator self-identification for surrounding obstacle detection”. eng. In: *Multimedia tools and applications* 76.5 (2017), pp. 6495–6520. ISSN: 1380-7501.
- [20] Atle Aalerud and Geir Hovland. “Dynamic Augmented Kalman Filtering for Human Motion Tracking under Occlusion Using Multiple 3D Sensors”. eng. In: *2020 15th IEEE Conference on Industrial Electronics and Applications (ICIEA)*. IEEE, 2020, pp. 533–540. ISBN: 1728151694.
- [21] J. D. Foley et al. *Computer Graphics: Principles and Practice*. Addison–Wesley, 1990.
- [22] Y. Xie, J. Tian, and X. X. Zhu. “Linking Points With Labels in 3D: A Review of Point Cloud Semantic Segmentation”. In: *IEEE Geoscience and Remote Sensing Magazine* 8.4 (2020), pp. 38–59. DOI: [10.1109/MGRS.2019.2937630](https://doi.org/10.1109/MGRS.2019.2937630).
- [23] K. B. Kaldestad, G. Hovland, and D. A. Anisi. “3D Sensor-Based Obstacle Detection Comparing Octrees and Point clouds Using CUDA”. In: *Modeling, identification and control* 33.4 (2012), pp. 123–130. DOI: <http://dx.doi.org/10.4173/mic.2012.4.1>.
- [24] D. J. R. Meagher. “Octree encoding : a new technique for the representation, manipulation and display of arbitrary 3-D objects by computer.” In: (1980).
- [25] *Octree*. URL: <https://en.wikipedia.org/wiki/Octree>. (accessed: 31.03.2021).
- [26] F. Pirotti et al. “Implementation and assessment of two density-based outlier detection methods over large spatial point clouds”. In: *Open Geospatial Data, Software and Standards* (2018). DOI: [10.1186/s40965-018-0056-5](https://doi.org/10.1186/s40965-018-0056-5).
- [27] *Removing outliers using a StatisticalOutlierRemoval filter*. URL: [https://pcl.readthedocs.io/en/latest/statistical\\_outlier.html](https://pcl.readthedocs.io/en/latest/statistical_outlier.html). (accessed: 09.04.2021).
- [28] H. Moravec and A. Elfes. “High resolution maps from wide angle sonar”. In: *Proceedings. 1985 IEEE International Conference on Robotics and Automation*. Vol. 2. 1985, pp. 116–121. DOI: [10.1109/ROBOT.1985.1087316](https://doi.org/10.1109/ROBOT.1985.1087316).
- [29] Francisco Rubio, Francisco Valero, and Carlos Llopis-Albert. “A review of mobile robots: Concepts, methods, theoretical framework, and applications”. eng. In: *International Journal of Advanced Robotic Systems* 16.2 (2019), p. 172988141983959. ISSN: 1729-8814.
- [30] Armin Hornung et al. “OctoMap: An Efficient Probabilistic 3D Mapping Framework Based on Octrees”. In: *Autonomous Robots* (2013). Software available at <http://octomap.github.com>. DOI: [10.1007/s10514-012-9321-0](https://doi.org/10.1007/s10514-012-9321-0). URL: <http://octomap.github.com>.
- [31] Martin A. Fischler and Robert C. Bolles. “Random Sample Consensus: A Paradigm for Model Fitting with Applications to Image Analysis and Automated Cartography”. In: *Commun. ACM* 24.6 (June 1981), pp. 381–395. ISSN: 0001-0782. DOI: [10.1145/358669.358692](https://doi.org/10.1145/358669.358692). URL: <https://doi.org/10.1145/358669.358692>.
- [32] F. Tarsha-Kurdi, T. Landes, and P. Grussenmeyer. “Hough-transform and extended ransac algorithms for automatic detection of 3d building roof planes from lidar data”. In: *ISPRS Workshop on Laser Scanning 2007 and SilviLaser* 36 (2007), pp. 407–412.
- [33] R. Schnabel, R. Wahl, and R. Klein. “Efficient ransac for point-cloud shape detection”. In: *Computer graphics forum* 26 (2007), pp. 214–226.

- [34] Daniel Wahrmann et al. “An Autonomous and Flexible Robotic Framework for Logistics Applications”. eng. In: *Journal of intelligent robotic systems* 93.3 (2019), pp. 419–431. ISSN: 0921-0296.
- [35] Wenhai Liu, Jie Hu, and Weiming Wang. “A Novel Camera Fusion Method Based on Switching Scheme and Occlusion-Aware Object Detection for Real-Time Robotic Grasping”. eng. In: *Journal of intelligent robotic systems* 100.3-4 (2020), p. 791. ISSN: 0921-0296.
- [36] Salvatore D’Avella, Paolo Tripicchio, and Carlo Alberto Avizzano. “A study on picking objects in cluttered environments: Exploiting depth features for a custom low-cost universal jamming gripper”. eng. In: *Robotics and computer-integrated manufacturing* 63 (2020), p. 101888. ISSN: 0736-5845.
- [37] George M Whitesides. “Soft Robotics”. eng. In: *Angewandte Chemie (International ed.)* 57.16 (2018), pp. 4258–4273. ISSN: 1433-7851.
- [38] R. B. Rusu and S. Cousins. “3D is here: Point Cloud Library (PCL)”. In: *2011 IEEE International Conference on Robotics and Automation*. 2011, pp. 1–4. DOI: [10.1109/ICRA.2011.5980567](https://doi.org/10.1109/ICRA.2011.5980567).
- [39] Steve Cousins. “Welcome to ROS Topics. (English)”. In: *IEEE Robotics Automation Magazine* (2010).
- [40] *ROS Tutorials*. URL: <http://wiki.ros.org/ROS/Tutorials/WritingPublisherSubscriber%28c%2B%2B%29>. (accessed: 07.01.2021).
- [41] S. Chitta, I. Sucan, and S. Cousins. “MoveIt! [ROS Topics]”. In: *IEEE Robotics Automation Magazine* 19.1 (2012), pp. 18–19. DOI: [10.1109/MRA.2011.2181749](https://doi.org/10.1109/MRA.2011.2181749).
- [42] *MoveIt Concepts*. URL: <https://moveit.ros.org/documentation/concepts/>. (accessed: 18.02.2021).
- [43] S. Choi, T. Kim, and W. Yu. “Performance evaluation of ransac family”. In: *Proceedings of the British Machine Vision Conference* (2009).
- [44] Hal Voepel et al. “Development of a vector-based 3D grain entrainment model with application to X-ray computed tomography scanned riverbed sediment”. eng. In: 44.15 (2019), pp. 3057–3077. ISSN: 0197-9337.
- [45] S. O. Nyberg. *Statistikk - en bayesiansk tilnærming*. Universitetsforlaget 2016, 2017.
- [46] D. G. Kleinbaum and M. Klein. *Survival analysis: A Self-learning text*. Springer, 2012.
- [47] *GitHub moveit\_tutorials*. URL: [https://github.com/ros-planning/moveit\\_tutorials/blob/master/doc/perception\\_pipeline/src/cylinder\\_segment.cpp](https://github.com/ros-planning/moveit_tutorials/blob/master/doc/perception_pipeline/src/cylinder_segment.cpp). (accessed: 18.02.2021).
- [48] Sachin Chitta. “MoveIt!: An Introduction”. eng. In: *Robot Operating System (ROS)*. Studies in Computational Intelligence. Cham: Springer International Publishing, 2016, pp. 3–27. ISBN: 3319260529.
- [49] Paul Glick et al. “A Soft Robotic Gripper With Gecko-Inspired Adhesive”. In: *IEEE Robotics and Automation Letters* PP (Jan. 2018), pp. 1–1. DOI: [10.1109/LRA.2018.2792688](https://doi.org/10.1109/LRA.2018.2792688).
- [50] Jun Shintake et al. “Soft Robotic Grippers”. eng. In: *Advanced materials (Weinheim)* 30.29 (2018), e1707035–n/a. ISSN: 0935-9648.
- [51] *Friction and Friction Coefficients*. URL: [https://www.engineeringtoolbox.com/friction-coefficients-d\\_778.html](https://www.engineeringtoolbox.com/friction-coefficients-d_778.html). (accessed: 08.04.2021).
- [52] Stewart Watkiss. *Learn Electronics with Raspberry Pi : Physical Computing with Circuits, Sensors, Outputs, and Projects*. eng. Berkeley, CA, 2016.
- [53] *Ubuntu install of ROS Kinetic*. URL: <http://wiki.ros.org/kinetic/Installation/Ubuntu>. (accessed: 03.03.2021).
- [54] *Raspberry Pi Images*. URL: <https://downloads.ubiquityrobotics.com/pi.html>. (accessed: 04.03.2021).

- [55] *Using Our Raspberry Pi Image Without A Magni*. URL: [https://learn.ubiquityrobotics.com/image\\_no\\_magni](https://learn.ubiquityrobotics.com/image_no_magni). (accessed: 04.03.2021).
- [56] *ROS Network Setup*. URL: <http://wiki.ros.org/ROS/NetworkSetup>. (accessed: 05.03.2021).
- [57] *fake-hwclock*. URL: <https://packages.debian.org/unstable/fake-hwclock>. (accessed: 12.03.2021).
- [58] *Arduino - Firmata*. URL: <https://www.arduino.cc/en/reference/firmata>. (accessed: 12.04.2021).
- [59] *pyFirmata*. URL: <https://pypi.org/project/pyFirmata/>. (accessed: 16.03.2021).
- [60] *pigpio library*. URL: <http://abyz.me.uk/rpi/pigpio/>. (accessed: 12.04.2021).
- [61] *RPi.GPIO Python Module*. URL: <https://sourceforge.net/p/raspberry-gpio-python/wiki/Home/>. (accessed: 12.04.2021).
- [62] *Automotive fully integrated H-bridge motor driver VNH2SP30*. URL: <https://www.pololu.com/file/0J52/vnh2sp30.pdf>. (accessed: 20.03.2021).
- [63] *Product specification IRB4400*. LTC3600. Rev. D. Linear Technology. 2011. URL: <https://www.analog.com/media/en/technical-documentation/data-sheets/3600fd.pdf>.
- [64] *Using Xacro to Clean Up a URDF File*. URL: <http://wiki.ros.org/urdf/Tutorials/Using%20Xacro%20to%20Clean%20Up%20a%20URDF%20File>. (accessed: 26.02.2021).
- [65] *IKFast: The Robot Kinematics Compiler*. URL: [http://openrave.org/docs/latest\\_stable/openravepy/ikfast/#ikfast-the-robot-kinematics-compiler](http://openrave.org/docs/latest_stable/openravepy/ikfast/#ikfast-the-robot-kinematics-compiler). (accessed: 02.04.2021).
- [66] Rodrigues. “Des lois géométriques qui régissent les déplacements d’un système solide dans l’espace, et de la variation des coordonnées provenant de ces déplacements considérés indépendamment des causes qui peuvent les produire.” fre. In: *Journal de Mathématiques Pures et Appliquées* (1840), pp. 380–440. URL: <http://eudml.org/doc/234443>.
- [67] D. M Henderson. *Euler angles, quaternions, and transformation matrices for space shuttle analysis*. eng. 1977.
- [68] Ioan A. Şucan, Mark Moll, and Lydia E. Kavraki. “The Open Motion Planning Library”. In: *IEEE Robotics & Automation Magazine* 19.4 (Dec. 2012). <https://ompl.kavrakilab.org>, pp. 72–82. DOI: [10.1109/MRA.2012.2205651](https://doi.org/10.1109/MRA.2012.2205651).
- [69] Mark Moll, Ioan A. Şucan, and Lydia E. Kavraki. “Benchmarking Motion Planning Algorithms: An Extensible Infrastructure for Analysis and Visualization”. In: *IEEE Robotics & Automation Magazine* 22.3 (Sept. 2015), pp. 96–102. DOI: [10.1109/MRA.2015.2448276](https://doi.org/10.1109/MRA.2015.2448276).
- [70] Zachary Kingston, Mark Moll, and Lydia E. Kavraki. “Exploring Implicit Spaces for Constrained Sampling-Based Planning”. In: *Intl. J. of Robotics Research* 38.10–11 (Sept. 2019), pp. 1151–1178. DOI: [10.1177/0278364919868530](https://doi.org/10.1177/0278364919868530).
- [71] Haris Balta et al. “Fast Statistical Outlier Removal Based Method for Large 3D Point Clouds of Outdoor Environments”. eng. In: *IFAC PapersOnLine* 51.22 (2018), pp. 348–353. ISSN: 2405-8963.
- [72] *MoveIt OMPL Planner*. URL: [http://docs.ros.org/en/kinetic/api/moveit\\_tutorials/html/doc/ompl\\_interface/ompl\\_interface\\_tutorial.html](http://docs.ros.org/en/kinetic/api/moveit_tutorials/html/doc/ompl_interface/ompl_interface_tutorial.html). (accessed: 08.04.2021).

# Appendix A

## MoveIt Setup Assistant

### A.1 Self-collision checking

**Optimize Self-Collision Checking**

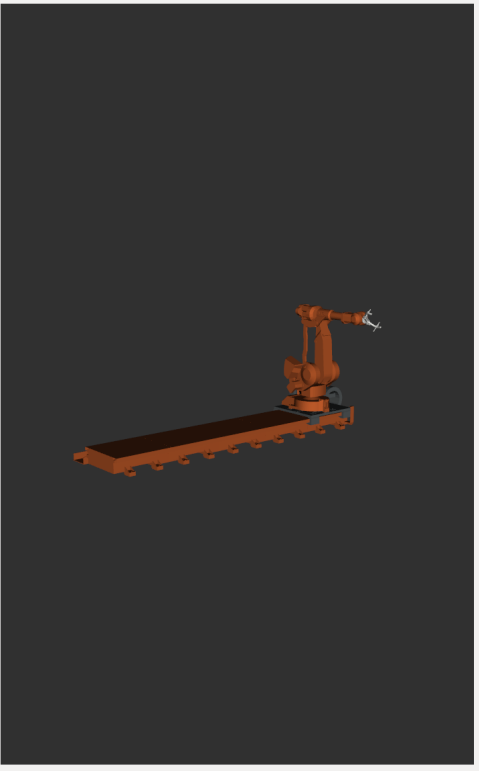
This searches for pairs of robot links that can safely be disabled from collision checking, decreasing motion planning time. These pairs are disabled when they are always in collision, never in collision, in collision in the robot's default position, or when the links are adjacent to each other on the kinematic chain. Sampling density specifies how many random robot positions to check for self collision.

Sampling Density: Low  High 90000

Min. collisions for "always"-colliding pairs: 95%

	lefty_track_left	lefty_track_mid_1	lefty_track_mid_2	lefty_track_mid_3	lefty_belt_0	lefty_belt_1	lefty_belt_2	lefty_belt_3	lefty_belt_4	lefty_belt_collision	lefty_belt_loop	lefty_carriage	lefty_base_link	lefty_link_1	lefty_link_2	lefty_link_3	lefty_link_4	lefty_link_5	lefty_link_6	lefty_tool	lefty_link_d2	lefty_link_d1
lefty_track_left	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
lefty_track_mid_1	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
lefty_track_mid_2	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
lefty_track_mid_3	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
lefty_belt_0	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
lefty_belt_1	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
lefty_belt_2	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
lefty_belt_3	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
lefty_belt_4	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
lefty_belt_collision	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
lefty_belt_loop	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
lefty_carriage	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
lefty_base_link	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
lefty_link_1	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
lefty_link_2	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
lefty_link_3	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
lefty_link_4	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
lefty_link_5	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
lefty_link_6	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
lefty_tool	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
lefty_link_d2	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
lefty_link_d1	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>

link name filter   linear view  matrix view





# Appendix B

## ABB IRB4400 Datasheet

## IRB 4400

### Fast, compact and versatile industrial robot



IRB 4400 is an extremely fast, compact robot for medium to heavy handling. It has exceptional all-round capabilities which makes it suitable for a variety of manufacturing applications. The load capacity of 60 kg at very high speeds usually permits handling of two parts at a time.

#### Fast, compact and versatile industrial robot

IRB 4400 is a rigid, well-balanced design and patented TrueMove™ function provide smooth and fast movement throughout the entire working range. This ensures very high quality in applications such as cutting. Rapid maneuverability makes the IRB 4400 perfectly matched for applications where speed and flexibility are important. The compact design and protected versions enables use in situations where conventional robots cannot work, such as foundry and spraying applications. The Foundry Plus 2 version is IP 67 protected and can be washed with high pressure steam, which makes it ideal for use in harsh environments.

#### Reliability and economy

The robust, rigid construction means long intervals between routine maintenance. Well-balanced steel arms with double bearing joints, a torque-strut on axis 2 and use of maintenance-free gearboxes and cabling also contribute to the very high levels of reliability. The drive train is optimised to give high torque with the lowest power consumption for economic operation.

#### Extensive communication for easy integration

The extensive communication capabilities include serial links, network interfaces, PLC, remote I/O and field bus interfaces. This makes for easy integration in small manufacturing stations as well as large scale factory automation systems.

#### Global service and support

For worry-free operation, ABB also offers Remote-Service, which gives remote access to equipment for monitoring and support. Moreover, ABB customers can take advantage of the company's service organization; with more than 35 years of experience in the arc welding sector, ABB provides service support in over 100 locations in 53 countries.

#### Main Applications

- Cutting/Deburring
- Die Spraying
- Dispensing
- Grinding/Polishing
- Measuring

## Specification

Robot version	Reach (m)	Handling capacity (kg)
IRB 4400/60	1.96	60
IRB 4400/L10	2.53	10
Supplementary load		
on axis 2	35 kg	
on axis 3	15 kg	
on axis 4	0-5 kg	
Number of axes	6	
Protection	Standard version IP 54, Foundry Plus 2 IP 67 and high pressure steam washable	
Mounting	Floor	
Controller	IRC5 Single Cabinet	
Integrated signal supply	23 signals and 10 power on upper arm	
Integrated air supply	Max. 8 bar on upper arm	

## Performance (according to ISO 9283)

	Position repeatability	Path repeatability*
IRB 4400/60	0.06 mm	0.09 mm
IRB 4400/L10	0.05 mm	0.16 mm

\*At 1.6 m/s.

## Technical information

### Electrical Connections

Supply voltage	200-600 V, 50/60 Hz
Rated power transformer rating	7.8 kVA

### Physical

Robot base	920 x 640 mm
Robot weight	1040 kg

### Environment

Ambient temperature for mechanical unit	
During operation	+5° C (41° F) to + 45° C (113° F)
Relative humidity	Max. 95%
Noise level	Max. 70 dB (A)
Safety	Double circuits with supervision, emergency stops and safety functions, 3-position enable device

### Emission

EMC/EMI-shielded

Data and dimensions may be changed without notice.

## Movement

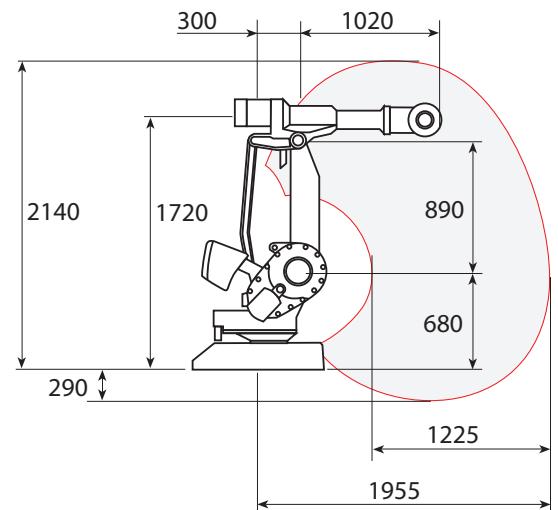
Axis movement	Working range	Axis max speed IRB 4400/60	Axis max speed IRB 4400/L10
Axis 1, Rotation	+165° to -165°	150°/s	150°/s
Axis 2, Arm	+95° to -70°	120°/s	150°/s
Axis 3, Arm	+65° to -60°	120°/s	150°/s
Axis 4, Rotation	+200° to -200°	225°/s	370°/s
Axis 5, Bend	+120° to -120°	250°/s	330°/s
Axis 6, Turn	+400° to -400°	330°/s	381°/s
	Max. rev: +200° <sup>1</sup> to -200° <sup>2</sup>		

<sup>1</sup>Max. rev: +183 to -183 valid for IRB 4400/L10

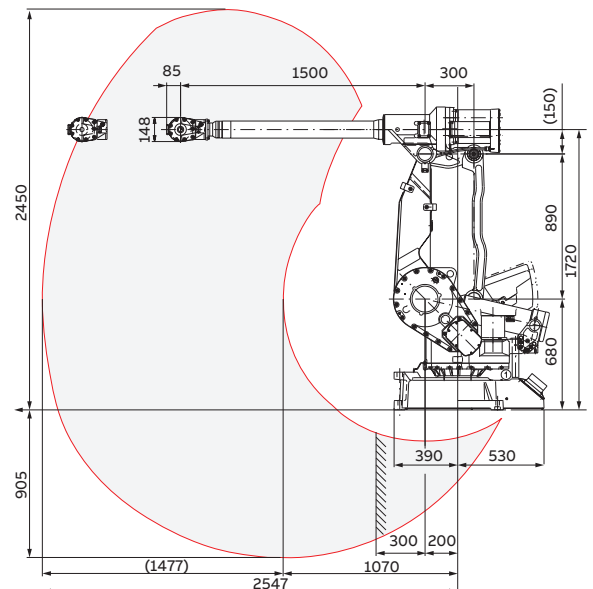
<sup>2</sup>The default working range for axis 6 can be extended by changing parameter values in the software.

There is a supervision function to prevent overheating in applications with intensive and frequent movements.

### Working range, IRB 4400/60



### Working range, IRB 4400/L10



# Appendix C

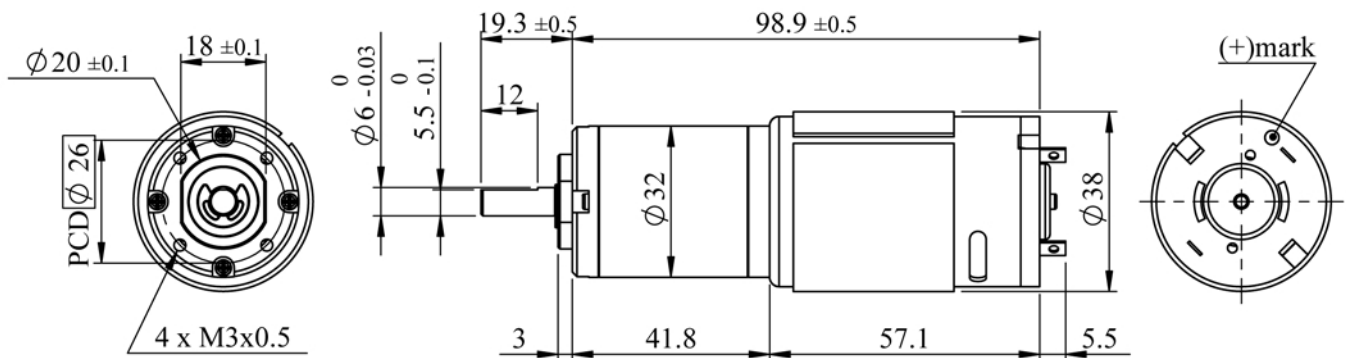
## DC Motor w/Gearing

### DIRECTION OF ROTATION



## SPECIFICATIONS

### DC Planetary Gear Brush Motor



(unit:mm)

### A. Operating Conditions:

1	Operating Voltage Range	6~12	VDC	4	Operating Temperature	-10~+60	°C
2	Rated Voltage	12	VDC	5	Storage Temperature	-30~+85	°C
3	Rated Load	21	kgf-cm	6	Test Position	Horizontal	~

### B. Electrical Characteristics:

1	Max. No-load Current	0.54	A	6	Max. Stall Current	20	A
2	No-load Speed	16±3	rpm	7	Insulation Resist.(500V)	20	MΩ
3	Rated-load Current	0.9	A	8	Dielectric Strength	250	VAC
4	Rated-load Speed	15.6±2	rpm	9	Motor Brush Type	Graphite	~
5	Min. Stall Torque	418	kgf-cm	10	Output Power at Max.Eff.	9	W

### C. Mechanical Characteristics:

1	Gear Type	Planetary	~	7	Max. Shaft Radial Load	3.5	kgf
2	Gear Ratio	1/516	~	8	Max. Shaft Runout	0.05	mm
3	Gear Material	Metal	~	9	Max. Shaft End Play	0.30	mm
4	Rated Tolerance Torque	22	kgf-cm	10	Bearing Type	Dual Ball	~
5	Moment. Tolerance Torque	44	kgf-cm	11	Net Weight	380±20	grams
6	Max. Shaft Axial Load	2.5	kgf				

# Appendix D

## DC Motor Driver VN12SP30

The complete datasheet can be found here [\[62\]](#).



## Automotive fully integrated H-bridge motor driver

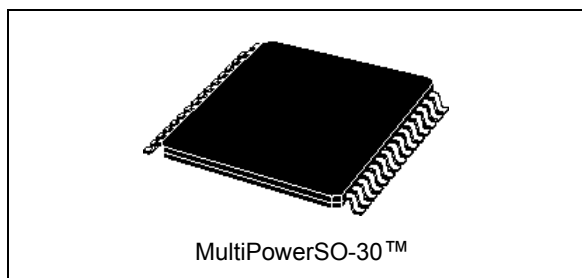
### Features

Type	$R_{DS(on)}$	$I_{out}$	$V_{CCmax}$
VNH2SP30-E	19m $\Omega$ max (per leg)	30A	41V

- 5V logic level compatible inputs
- Undervoltage and overvoltage shut-down
- Overvoltage clamp
- Thermal shut down
- Cross-conduction protection
- Linear current limiter
- Very low stand-by power consumption
- PWM operation up to 20 kHz
- Protection against loss of ground and loss of  $V_{CC}$
- Current sense output proportional to motor current
- Package: ECOPACK<sup>®</sup>

### Description

The VNH2SP30-E is a full bridge motor driver intended for a wide range of automotive applications. The device incorporates a dual monolithic high side driver and two low side switches. The high side driver switch is designed using STMicroelectronic's well known and proven proprietary VIPower™ M0 technology which permits efficient integration on the same die of a true Power MOSFET with an intelligent signal/protection circuitry.



The low side switches are vertical MOSFETs manufactured using STMicroelectronic's proprietary EHD ('STripFET™') process. The three die are assembled in the MultiPowerSO-30 package on electrically isolated leadframes. This package, specifically designed for the harsh automotive environment offers improved thermal performance thanks to exposed die pads. Moreover, its fully symmetrical mechanical design allows superior manufacturability at board level. The input signals  $IN_A$  and  $IN_B$  can directly interface to the microcontroller to select the motor direction and the brake condition. The  $DIAG_A/EN_A$  or  $DIAG_B/EN_B$ , when connected to an external pull-up resistor, enable one leg of the bridge. They also provide a feedback digital diagnostic signal. The normal condition operation is explained in [Table 12: Truth table in normal operating conditions on page 14](#). The motor current can be monitored with the CS pin by delivering a current proportional to its value. The speed of the motor can be controlled in all possible conditions by the PWM up to 20 kHz. In all cases, a low level state on the PWM pin will turn off both the  $LS_A$  and  $LS_B$  switches. When PWM rises to a high level,  $LS_A$  or  $LS_B$  turn on again depending on the input pin state.

**Table 1. Device summary**

Package	Order codes	
	Tube	Tape and Reel
MultiPowerSO-30	VNH2SP30-E	VNH2SP30TR-E

# Appendix E

## Source code

### E.1 Object detection and localization

#### E.1.1 main.h

[https://github.com/evenfl/p26\\_master/blob/master/p26\\_cylinder\\_segmentation/src/main.h](https://github.com/evenfl/p26_master/blob/master/p26_cylinder_segmentation/src/main.h)

```
1  #pragma once
2
3  #include <pcl/ModelCoefficients.h>
4  #include <pcl/io/pcd_io.h>
5  #include <pcl/point_types.h>
6  #include <pcl/filters/extract_indices.h>
7  #include <pcl/filters/passthrough.h>
8  #include <pcl/features/normal_3d.h>
9  #include <pcl/sample_consensus/method_types.h>
10 #include <pcl/sample_consensus/model_types.h>
11 #include <pcl/segmentation/sac_segmentation.h>
12 #include <pcl/common/distances.h>
13 #include "pcl_ros/point_cloud.h"
14 #include <boost/foreach.hpp>
15 #include <iostream>
16 #include <string>
17 #include <fstream> //For saving to text file
18 #include <ros/ros.h>
19 #include "std_msgs/String.h"
20 #include "std_msgs/Bool.h"
21 #include <sstream>
22 #include <pcl_conversions/pcl_conversions.h>
23 #include <sensor_msgs/PointCloud.h>
24 #include <sensor_msgs/PointCloud2.h>
25 #include <sensor_msgs/point_field_conversion.h>
26 #include <sensor_msgs/point_cloud_conversion.h>
27 #include <math.h>      /* round, floor, ceil, trunc */
28 #include <cmath>      /* std::abs */
29 #include "geometry_msgs/Point.h"
```

```

30
31 typedef pcl::PointXYZ PointT;
32 typedef pcl::PointCloud<pcl::PointXYZ> PointCloud;
33
34 //const double cylinderDiameter = 0.169; // Yellow pipe in lab
35 //const double cylinderLength = 0.53; // Yellow pipe in lab
36 const double cylinderDiameter = 0.26;
37 const double cylinderRadius = cylinderDiameter/2;
38 const double cylinderLength = 1.2;
39
40 const float x_min = 4.2;
41 const float x_max = 6.0;
42 const float y_min = 1.0;
43 const float y_max = 10.0;
44 const float z_min = 0.05;
45 const float z_max = 2.0;
46
47
48 struct AddCylinderParams
49 {
50     /* Radius of the cylinder. */
51     double radius;
52     /* Direction vector towards the z-axis of the cylinder. */
53     double direction_vec[3];
54     /* Center point of the cylinder. */
55     double center_pt[3];
56     /* Height of the cylinder. */
57     double height;
58 };

```

## E.1.2 addCylinder.h

[https://github.com/evenfl/p26\\_master/blob/master/p26\\_cylinder\\_segmentation/src/addCylinder.h](https://github.com/evenfl/p26_master/blob/master/p26_cylinder_segmentation/src/addCylinder.h)

```

1  #pragma once
2
3  #include "main.h"
4  #include <moveit/planning_scene_interface/planning_scene_interface.h>
5  #include <moveit_msgs/CollisionObject.h>
6
7  // MoveIt
8  #include <moveit/robot_model_loader/robot_model_loader.h>
9  #include <moveit/planning_scene/planning_scene.h>
10
11 #include <moveit/kinematic_constraints/utils.h>
12

```

```

13 moveit_msgs::CollisionObject addCylinder(const AddCylinderParams
    ↪ cylinder_params);

```

### E.1.3 addCylinder.cpp

[https://github.com/evenfl/p26\\_master/blob/master/p26\\_cylinder\\_segmentation/src/addCylinder.cpp](https://github.com/evenfl/p26_master/blob/master/p26_cylinder_segmentation/src/addCylinder.cpp)

```

1  #include "addCylinder.h"
2
3  moveit_msgs::CollisionObject addCylinder(const AddCylinderParams
    ↪ cylinder_params)
4  {
5      // Define a collision object ROS message.
6      moveit_msgs::CollisionObject collision_object;
7      collision_object.header.frame_id = "world";
8      collision_object.id = "cylinder";
9
10     // Define a cylinder which will be added to the world.
11     shape_msgs::SolidPrimitive primitive;
12     primitive.type = primitive.CYLINDER;
13     primitive.dimensions.resize(2);
14     /* Setting height of cylinder. */
15     primitive.dimensions[0] = cylinder_params.height;
16     /* Setting radius of cylinder. */
17     primitive.dimensions[1] = cylinder_params.radius;
18
19     // Define a pose for the cylinder (specified relative to frame_id).
20     geometry_msgs::Pose cylinder_pose;
21     /* Computing and setting quaternion from axis angle representation. */
22     Eigen::Vector3d cylinder_z_direction(cylinder_params.direction_vec[0],
    ↪ cylinder_params.direction_vec[1],
23                                         cylinder_params.direction_vec[2]);
24     Eigen::Vector3d origin_z_direction(0., 0., 1.);
25     Eigen::Vector3d axis;
26     axis = origin_z_direction.cross(cylinder_z_direction);
27     axis.normalize();
28     double angle = acos(cylinder_z_direction.dot(origin_z_direction));
29     cylinder_pose.orientation.x = axis.x() * sin(angle / 2);
30     cylinder_pose.orientation.y = axis.y() * sin(angle / 2);
31     cylinder_pose.orientation.z = axis.z() * sin(angle / 2);
32     cylinder_pose.orientation.w = cos(angle / 2);
33
34     // Setting the position of cylinder.
35     cylinder_pose.position.x = cylinder_params.center_pt[0];
36     cylinder_pose.position.y = cylinder_params.center_pt[1];
37     cylinder_pose.position.z = cylinder_params.center_pt[2];

```

```

38
39     // Add cylinder as collision object
40     collision_object.primitives.push_back(primitive);
41     collision_object.primitive_poses.push_back(cylinder_pose);
42     collision_object.operation = collision_object.ADD;
43
44     return collision_object;
45
46 }

```

#### E.1.4 segment.h

[https://github.com/evenfl/p26\\_master/blob/master/p26\\_cylinder\\_segmentation/src/segment.h](https://github.com/evenfl/p26_master/blob/master/p26_cylinder_segmentation/src/segment.h)

```

1  #pragma once
2
3  #include "main.h"
4
5  pcl::PointCloud<PointT>::Ptr segment(const pcl::PointCloud<PointT>::Ptr&
   → input, pcl::ModelCoefficients::Ptr coefficients_cylinder);
6
7  pcl::PointCloud<pcl::PointXYZ>::Ptr
8  passThroughFilterSphere(pcl::PointCloud<PointT>::Ptr& cloud,
9  pcl::PointXYZ sphereCenterPoint, const double radius, bool remove_outside);

```

#### E.1.5 segment.cpp

[https://github.com/evenfl/p26\\_master/blob/master/p26\\_cylinder\\_segmentation/src/segment.cpp](https://github.com/evenfl/p26_master/blob/master/p26_cylinder_segmentation/src/segment.cpp)

```

1  #include "segment.h"
2
3  pcl::PointCloud<PointT>::Ptr segment(const pcl::PointCloud<PointT>::Ptr&
   → input, pcl::ModelCoefficients::Ptr coefficients_cylinder) //const
   → sensor_msgs::PointCloudConstPtr& input)
4  {
5
6     // All the objects needed
7     //pcl::PCDReader reader;
8     pcl::PassThrough<PointT> pass;
9     pcl::NormalEstimation<PointT, pcl::Normal> ne;
10    pcl::SACSegmentationFromNormals<PointT, pcl::Normal> seg;
11    pcl::PCDWriter writer;
12    pcl::ExtractIndices<PointT> extract;
13    pcl::ExtractIndices<pcl::Normal> extract_normals;
14    pcl::search::KdTree<PointT>::Ptr tree (new pcl::search::KdTree<PointT> ());

```

```

15
16 // Datasets
17 //pcl::PointCloud<PointT>::Ptr cloud (new pcl::PointCloud<PointT>);
18 pcl::PointCloud<PointT>::Ptr cloud_filtered (new pcl::PointCloud<PointT>);
19 pcl::PointCloud<PointT>::Ptr filter_x (new pcl::PointCloud<PointT>);
20 pcl::PointCloud<PointT>::Ptr filter_y (new pcl::PointCloud<PointT>);
21 pcl::PointCloud<PointT>::Ptr filter_z (new pcl::PointCloud<PointT>);
22 pcl::PointCloud<pcl::Normal>::Ptr cloud_normals (new
    ↪ pcl::PointCloud<pcl::Normal>);
23 pcl::PointCloud<PointT>::Ptr cloud_filtered2 (new pcl::PointCloud<PointT>);
24 pcl::PointCloud<pcl::Normal>::Ptr cloud_normals2 (new
    ↪ pcl::PointCloud<pcl::Normal>);
25 pcl::ModelCoefficients::Ptr coefficients_plane (new
    ↪ pcl::ModelCoefficients); //, coefficients_cylinder (new
    ↪ pcl::ModelCoefficients);
26 pcl::PointIndices::Ptr inliers_plane (new pcl::PointIndices),
    ↪ inliers_cylinder (new pcl::PointIndices);
27
28 // Read in the cloud data
29 //reader.read ("table_scene_mug_stereo_textured.pcd", *cloud);
30 //std::cerr << "PointCloud has: " << cloud->size () << " data points." <<
    ↪ std::endl;
31 //std::cerr << "PointCloud has: " << input->size () << " data points." <<
    ↪ std::endl;
32
33 // Build a passthrough filter to remove spurious NaNs
34 //pass.setInputCloud (cloud);
35 pass.setInputCloud (input);
36 pass.setFilterFieldName ("x");
37 pass.setFilterLimits (x_min, x_max); //5.5 works
38 pass.filter (*cloud_filtered);
39
40 pass.setInputCloud (cloud_filtered);
41 pass.setFilterFieldName ("y");
42 pass.setFilterLimits (y_min, y_max); //5.5 works
43 pass.filter (*cloud_filtered);
44
45 pass.setInputCloud (cloud_filtered);
46 pass.setFilterFieldName ("z");
47 pass.setFilterLimits (z_min, z_max); //5.5 works
48 pass.filter (*cloud_filtered);
49
50 //std::cerr << "PointCloud after filtering has: " << cloud_filtered->size ()
    ↪ << " data points." << std::endl;
51
52 // Estimate point normals

```



```

53 ne.setSearchMethod (tree);
54 ne.setInputCloud (cloud_filtered);
55 ne.setKSearch (50);
56 ne.compute (*cloud_normals);
57
58 // Create the segmentation object for the planar model and set all the
   ↪ parameters
59 // seg.setOptimizeCoefficients (true);
60 // seg.setModelType (pcl::SACMODEL_NORMAL_PLANE);
61 // //seg.setNormalDistanceWeight (0.1);
62 // seg.setMethodType (pcl::SAC_RANSAC);
63 // //seg.setMaxIterations (100);
64 // seg.setDistanceThreshold (0.01); //0.31 if 1 segmentation
65 // seg.setInputCloud (cloud_filtered);
66 // seg.setInputNormals (cloud_normals);
67 // // Obtain the plane inliers and coefficients
68 // seg.segment (*inliers_plane, *coefficients_plane);
69 // //std::cerr << "Plane coefficients: " << *coefficients_plane <<
   ↪ std::endl;
70
71 // // Extract the planar inliers from the input cloud
72 // extract.setInputCloud (cloud_filtered);
73 // extract.setIndices (inliers_plane);
74 // extract.setNegative (false);
75
76 // // Write the planar inliers to disk
77 // pcl::PointCloud<PointT> cloud_plane; //(new pcl::PointCloud<PointT> ());
78 // extract.filter (cloud_plane);
79 // //std::cerr << "PointCloud representing the planar component: " <<
   ↪ cloud_plane->size () << " data points." << std::endl;
80 // //writer.write ("src/P26_cylinder_segmentation/pointclouds/plane.pcd",
   ↪ *cloud_plane, false);
81
82 // // Remove the planar inliers, extract the rest
83 // extract.setNegative (true);
84 // extract.filter (*cloud_filtered2);
85 // extract_normals.setNegative (true);
86 // extract_normals.setInputCloud (cloud_normals);
87 // extract_normals.setIndices (inliers_plane);
88 // extract_normals.filter (*cloud_normals2);
89
90 // Create the segmentation object for cylinder segmentation and set all the
   ↪ parameters
91 seg.setOptimizeCoefficients (true);
92 seg.setModelType (pcl::SACMODEL_CYLINDER);
93 seg.setMethodType (pcl::SAC_RANSAC);

```

```

94  seg.setNormalDistanceWeight (0.2); //0.01 works (0.1 seperated the robot
    ↪ from the cylinder)
95  seg.setMaxIterations (10000);
96  seg.setDistanceThreshold (0.08);//0.1 works
97  seg.setRadiusLimits (cylinderRadius-0.03, cylinderRadius+0.03);//0.08, 0.16
    ↪ works
98  seg.setInputCloud (cloud_filtered);
99  seg.setInputNormals (cloud_normals);
100
101
102  // Obtain the cylinder inliers and coefficients
103  seg.segment (*inliers_cylinder, *coefficients_cylinder);
104
105  // Write the cylinder inliers to disk
106  extract.setInputCloud (cloud_filtered);
107  extract.setIndices (inliers_cylinder);
108  extract.setNegative (false);
109  pcl::PointCloud<PointT>::Ptr cloud_cylinder (new pcl::PointCloud<PointT>
    ↪ ());
110  extract.filter (*cloud_cylinder);
111  if (cloud_cylinder->points.empty ()) {
112  // Can't find the cylindrical component.
113      std::cerr << "x";
114  }
115
116  // writer.write ("input_cloud.pcd", *input, false);
117  // writer.write ("cloud_filtered.pcd", *cloud_filtered, false);
118  // writer.write ("cylinder.pcd", *cloud_cylinder, false);
119
120
121  return cloud_cylinder;
122
123 }
124
125 pcl::PointCloud<pcl::PointXYZ>::Ptr
126 passThroughFilterSphere(pcl::PointCloud<PointT>::Ptr& cloud,
127 pcl::PointXYZ sphereCenterPoint, const double radius, bool remove_outside)
128 {
129  pcl::PointCloud<pcl::PointXYZ>::Ptr filteredCloud(new
    ↪ pcl::PointCloud<pcl::PointXYZ>);
130  float distanceFromSphereCenterPoint;
131  bool pointIsWithinSphere;
132  bool addPointToFilteredCloud;
133  for (int point_i = 0; point_i < cloud->size(); ++point_i)
134  {

```

```

135     distanceFromSphereCenterPoint = pcl::euclideanDistance(cloud->at(point_i),
    ↪     sphereCenterPoint);
136     pointIsWithinSphere = distanceFromSphereCenterPoint <= radius;
137     addPointToFilteredCloud = (!pointIsWithinSphere && remove_outside) ||
    ↪     (pointIsWithinSphere && !remove_outside);
138     if (addPointToFilteredCloud){
139         filteredCloud->push_back(cloud->at(point_i));
140     }
141 }
142 return filteredCloud;
143 }

```

## E.1.6 main.cpp

[https://github.com/evenfl/p26\\_master/blob/master/p26\\_cylinder\\_segmentation/src/main.cpp](https://github.com/evenfl/p26_master/blob/master/p26_cylinder_segmentation/src/main.cpp)

```

1  #include "main.h"
2  #include "segment.h"
3  #include "addCylinder.h"
4
5  typedef pcl::PointCloud<pcl::PointXYZ> PointCloud;
6
7  PointCloud::Ptr cloud_merged (new PointCloud);
8  PointCloud::Ptr cloud_cylinder (new PointCloud);
9  PointCloud::Ptr cloud_cylinder_tmp (new PointCloud);
10 void callback(const sensor_msgs::PointCloud2ConstPtr& input);
11 void callback_find_cylinder(const std_msgs::Bool::ConstPtr& data);
12 int counter = 0;
13 bool findCylinder = true;
14
15 // Declare a variable of type AddCylinderParams and store relevant values from
    ↪ ModelCoefficients.
16 AddCylinderParams cylinder_params;
17
18 int main(int argc, char** argv)
19 {
20     ros::init(argc, argv, "cylinder_segmentation");
21     ros::NodeHandle nh;
22     ros::Subscriber sub1 = nh.subscribe ("/master/jetson1/kinect_decomp", 1,
    ↪     callback);
23     ros::Subscriber sub2 = nh.subscribe ("/master/jetson2/kinect_decomp", 1,
    ↪     callback);
24     ros::Subscriber sub3 = nh.subscribe ("/master/jetson3/kinect_decomp", 1,
    ↪     callback);
25     ros::Subscriber sub4 = nh.subscribe ("/master/jetson4/kinect_decomp", 1,
    ↪     callback);

```

```

26  ros::Subscriber sub5 = nh.subscribe ("/master/jetson5/kinect_decomp", 1,
    ↪  callback);
27  ros::Subscriber sub6 = nh.subscribe ("/master/jetson6/kinect_decomp", 1,
    ↪  callback);
28  ros::Subscriber sub_find_cylinder = nh.subscribe
    ↪  ("/p26_leftty/find_cylinder", 1, callback_find_cylinder);
29  // ros::Subscriber sub6 = nh.subscribe ("/master/merged_point_cloud", 1,
    ↪  callback); // merged point cloud (Lower frame rate)
30  ros::Publisher pub = nh.advertise<PointCloud> ("cloud_cylinder", 1);
31  ros::Publisher cylinder_object_publisher =
    ↪  nh.advertise<moveit_msgs::CollisionObject>("collision_object", 1);
32  ros::Publisher pub_com = nh.advertise<geometry_msgs::Point> ("cylinder_com",
    ↪  1);
33  ros::Publisher pub_dirvec = nh.advertise<geometry_msgs::Point>
    ↪  ("cylinder_dirvec", 1);
34
35  ros::Rate rate(20);
36
37  int iterations = 0;
38
39  PointT point_com_avg;
40  PointT dirvec_avg;
41  const int nrOfIterations = 20;
42  const float deviance = 0.06; // For 26 cm cylinder
43  // const float deviance = 0.1; // For 16 cm cylinder
44
45  unsigned int inconsistencyCounter = 0;
46
47  std::cerr << "Segmenting cylinder." << std::endl;
48
49  std::cerr << "0%";
50  for (int i = 0; i < nrOfIterations-6; i++){std::cerr << " ";}
51  std::cerr << "100%" << std::endl;
52
53  while (ros::ok())
54  {
55      ros::spinOnce();
56      rate.sleep();
57
58      if (counter >= 6 && findCylinder == true)
59      {
60          // If all pointclouds are received, find the pose
61
62          pcl::ModelCoefficients::Ptr coefficients_cylinder (new
    ↪  pcl::ModelCoefficients);
63          cloud_cylinder = segment(cloud_merged, coefficients_cylinder);

```

```

64
65 // BEGIN SPHERE FILTER
66 PointT point;
67 point.x = coefficients_cylinder->values[0];
68 point.y = coefficients_cylinder->values[1];
69 point.z = coefficients_cylinder->values[2];
70 PointT dirvec;
71 dirvec.x = coefficients_cylinder->values[3];
72 dirvec.y = coefficients_cylinder->values[4];
73 dirvec.z = coefficients_cylinder->values[5];
74
75 float adj = 0.0f;
76 float increment = 0.001f; //Check again every 1 mm
77
78 adj = (x_min - point.x)/dirvec.x;
79 point.x = x_min;
80 point.y = point.y + adj*dirvec.y;
81 point.z = point.z + adj*dirvec.z;
82
83 if (point.y < y_min)
84 {
85     adj = (y_min - point.y)/dirvec.y;
86     point.x = point.x + adj*dirvec.x;
87     point.y = y_min;
88     point.z = point.z + adj*dirvec.z;
89 }
90 else if (point.y > y_max)
91 {
92     adj = (y_max - point.y)/dirvec.y;
93     point.x = point.x + adj*dirvec.x;
94     point.y = y_max;
95     point.z = point.z + adj*dirvec.z;
96 }
97 if (point.z < z_min)
98 {
99     adj = (z_min - point.z)/dirvec.z;
100    point.x = point.x + adj*dirvec.x;
101    point.y = point.y + adj*dirvec.y;
102    point.z = z_min;
103 }
104 else if (point.z > z_max)
105 {
106    adj = (z_max - point.z)/dirvec.z;
107    point.x = point.x + adj*dirvec.x;
108    point.y = point.y + adj*dirvec.y;
109    point.z = z_max;

```

```

110     }
111     if (point.x + increment*dirvec.x > x_max || point.x + increment*dirvec.x
    ↪ < x_min ||
112         point.y + increment*dirvec.y > y_max || point.y + increment*dirvec.y
    ↪ < y_min ||
113         point.z + increment*dirvec.z > z_max || point.z + increment*dirvec.z
    ↪ < z_min)
114     {
115         increment = std::abs(increment)*(-1);
116     }
117
118     PointT point_com;
119     unsigned long biggestCloudSize = 0;
120
121     while (point.x >= x_min && point.x <= x_max && point.y >= y_min &&
    ↪ point.y <= y_max && point.z >= z_min && point.z <= z_max)
122     {
123         //     cloud_cylinder_tmp = passThroughFilterSphere(cloud_cylinder, point,
    ↪ cylinderLength/2, false);
124         cloud_cylinder_tmp = passThroughFilterSphere(cloud_cylinder, point,
    ↪ sqrt((cylinderLength/2)*(cylinderLength/2)+cylinderRadius*cylinderRadius),
    ↪ false);
125         if (cloud_cylinder_tmp->size () > biggestCloudSize)
126         {
127             biggestCloudSize = cloud_cylinder_tmp->size ();
128             point_com = point;
129         }
130         point.x = point.x + increment*dirvec.x;
131         point.y = point.y + increment*dirvec.y;
132         point.z = point.z + increment*dirvec.z;
133     }
134
135     cloud_cylinder = passThroughFilterSphere(cloud_cylinder, point_com,
    ↪ sqrt((cylinderLength/2)*(cylinderLength/2)+cylinderRadius*cylinderRadius),
    ↪ false);
136
137     // END SPHERE FILTER
138
139     counter = 0;
140
141     if (iterations == 0 && point_com.x != 0.0)
142     {
143         point_com_avg.x = point_com.x;
144         point_com_avg.y = point_com.y;
145         point_com_avg.z = point_com.z;
146         dirvec_avg.x = dirvec.x;

```



```

147     dirvec_avg.y = dirvec.y;
148     dirvec_avg.z = dirvec.z;
149     iterations++;
150     inconsistencyCounter = 0;
151     std::cerr << "#";
152 }
153
154 if ( std::abs(dirvec.x - dirvec_avg.x) > 0.5 || std::abs(dirvec.y -
↪ dirvec_avg.y) > 0.5 || std::abs(dirvec.z - dirvec_avg.z) > 0.5 )
155 {
156     dirvec.x = -dirvec.x;
157     dirvec.y = -dirvec.y;
158     dirvec.z = -dirvec.z;
159 }
160
161 if ( std::abs(point_com.x - point_com_avg.x) > deviance ||
↪ std::abs(point_com.z - point_com_avg.z) > deviance ||
↪ std::abs(point_com.z - point_com_avg.z) > deviance && point_com.x !=
↪ 0 )
162 {
163     inconsistencyCounter++;
164     std::cerr << "-";
165     if (inconsistencyCounter >= round(0.25*nrOfIterations))
166     {
167         iterations = 0;
168         inconsistencyCounter = 0;
169         std::cerr << " Inconsistent result, starting over." << std::endl;
170     }
171 }
172 else if (point_com.x != 0.0)
173 {
174     point_com_avg.x = (point_com_avg.x*iterations +
↪ point_com.x)/(iterations+1);
175     point_com_avg.y = (point_com_avg.y*iterations +
↪ point_com.y)/(iterations+1);
176     point_com_avg.z = (point_com_avg.z*iterations +
↪ point_com.z)/(iterations+1);
177     dirvec_avg.x = (dirvec_avg.x*iterations + dirvec.x)/(iterations+1);
178     dirvec_avg.y = (dirvec_avg.y*iterations + dirvec.y)/(iterations+1);
179     dirvec_avg.z = (dirvec_avg.z*iterations + dirvec.z)/(iterations+1);
180     iterations++;
181     //inconsistencyCounter = 0;
182     std::cerr << "#" << iterations;
183 }
184 if (iterations >= nrOfIterations)
185 {

```

```

186 //cylinder_object_publisher.publish(addCylinder(cylinder_params));
187 std::cerr << std::endl << "Direction vector: [ " << dirvec_avg.x << ",
    ↪ " << dirvec_avg.y << ", " << dirvec_avg.z << " ]" << std::endl;
188 std::cerr << std::endl << "Centre of mass: [ " << point_com_avg.x <<
    ↪ " , " << point_com_avg.y << " , " << point_com_avg.z << " ]" <<
    ↪ std::endl << std::endl;
189 iterations = 0;
190
191 cylinder_params.center_pt[0] = point_com_avg.x;
192 cylinder_params.center_pt[1] = point_com_avg.y;
193 cylinder_params.center_pt[2] = point_com_avg.z;
194
195 cylinder_params.height = cylinderLength;
196 cylinder_params.radius = cylinderRadius;
197
198 cylinder_params.direction_vec[0] = dirvec_avg.x;
199 cylinder_params.direction_vec[1] = dirvec_avg.y;
200 cylinder_params.direction_vec[2] = dirvec_avg.z;
201 cylinder_object_publisher.publish(addCylinder(cylinder_params));
202 geometry_msgs::Point pub_com_msg;
203 geometry_msgs::Point pub_dirvec_msg;
204 pub_com_msg.x = point_com_avg.x;
205 pub_com_msg.y = point_com_avg.y;
206 pub_com_msg.z = point_com_avg.z;
207 pub_dirvec_msg.x = dirvec_avg.x;
208 pub_dirvec_msg.y = dirvec_avg.y;
209 pub_dirvec_msg.z = dirvec_avg.z;
210 pub_com.publish(pub_com_msg);
211 pub_dirvec.publish(pub_dirvec_msg);
212
213 moveit_msgs::CollisionObject collision_object;
214 collision_object.header.frame_id = "world";
215 collision_object.id = "box";
216
217 shape_msgs::SolidPrimitive primitive;
218 primitive.type = primitive.BOX;
219 primitive.dimensions.resize(3);
220 primitive.dimensions[0] = 0.3;
221 primitive.dimensions[1] = 0.4;
222 primitive.dimensions[2] = 0.25;
223 geometry_msgs::Pose box_pose;
224 box_pose.orientation.w = 1.0;
225 box_pose.position.x = 5.42;
226 box_pose.position.y = 9.23;
227 box_pose.position.z = 0.125;
228

```

```

229     collision_object.primitives.push_back(primitive);
230     collision_object.primitive_poses.push_back(box_pose);
231     collision_object.operation = collision_object.ADD;
232
233     cylinder_object_publisher.publish(collision_object);
234
235     // Write pointcloud to pcd files
236     //     pcl::PCDWriter writer;
237     //     writer.write
238     → ("src/p26_master/p26_cylinder_segmentation/pointclouds/cylinder_filtered.pcd",
239     → *cloud_cylinder, false);
240     //     writer.write
241     → ("src/p26_master/p26_cylinder_segmentation/pointclouds/cloud_merged.pcd",
242     → *cloud_merged, false);
243
244     findCylinder = false;
245     //     ros::shutdown();
246     }
247     cloud_merged->header.frame_id = "world";
248     pub.publish(*cloud_merged);
249     cloud_merged->clear();
250     }
251     else if(findCylinder == false){
252     cloud_merged->clear();
253     counter = 0;
254     }
255     }
256     }
257
258     void callback(const sensor_msgs::PointCloud2ConstPtr& input)
259     {
260     pcl::PointCloud<PointT> cloud;
261     pcl::fromROSMsg (*input, cloud); //cloud is the output
262     *cloud_merged += cloud;
263     counter++;
264     }
265
266     void callback_find_cylinder(const std_msgs::Bool::ConstPtr& data)
267     {
268     findCylinder = true;
269     }

```

## E.2 Obstacle mapping

### E.2.1 sensor\_kinect\_pointcloud.yaml

```

1  sensors:
2    - sensor_plugin: occupancy_map_monitor/PointCloudOctomapUpdater
3      point_cloud_topic: /master/merged_point_cloud
4      max_range: 15.0
5      point_subsample: 1
6      padding_offset: 0.1
7      padding_scale: 1.0
8      max_update_rate: 1.0
9      filtered_cloud_topic: filtered_cloud

```

## E.2.2 sensor\_manager.launch

```

1  <launch>
2
3    <!-- This file makes it easy to include the settings for sensor managers
4     → -->
5
6    <!-- Params for 3D sensors config -->
7  <!-- <rosparam command="load" file="$(find
8     → p26_lefty_moveit_config)/config/sensors_3d.yaml" />-->
9
10   <!-- Params for the octomap monitor -->
11   <param name="octomap_frame" type="string" value="lefty_track_left" />
12   <!-- <param name="octomap_frame" type="string" value="lefty_tool" />-->
13   <param name="octomap_resolution" type="double" value="0.12" />
14   <param name="max_range" type="double" value="15.0" />
15
16   <!-- Load the robot specific sensor manager; this sets the
17     → moveit_sensor_manager ROS parameter -->
18   <arg name="moveit_sensor_manager" default="p26_lefty" />
19   <include file="$(find p26_lefty_moveit_config)/launch/$(arg
20     → moveit_sensor_manager)_moveit_sensor_manager.launch.xml" />
21
22 </launch>

```

## E.2.3 main.h

[https://github.com/evenfl/p26\\_master/blob/master/p26\\_pick\\_and\\_place/src/main.h](https://github.com/evenfl/p26_master/blob/master/p26_pick_and_place/src/main.h)

```

1  #pragma once
2
3  #include <pcl/ModelCoefficients.h>
4  #include <pcl/io/pcd_io.h>
5  #include <pcl/point_types.h>
6  #include <pcl/filters/extract_indices.h>
7  #include <pcl/filters/passthrough.h>
8  #include <pcl/features/normal_3d.h>

```

```

9  #include <pcl/sample_consensus/method_types.h>
10 #include <pcl/sample_consensus/model_types.h>
11 #include <pcl/segmentation/sac_segmentation.h>
12 #include <pcl/common/distances.h>
13 #include <pcl/filters/statistical_outlier_removal.h>
14 #include "pcl_ros/point_cloud.h"
15 #include <boost/foreach.hpp>
16 #include <iostream>
17 #include <string>
18 #include <fstream> //For saving to text file
19 #include <ros/ros.h>
20 #include "std_msgs/String.h"
21 #include <sstream>
22 #include <pcl_conversions/pcl_conversions.h>
23 #include <sensor_msgs/PointCloud.h>
24 #include <sensor_msgs/PointCloud2.h>
25 #include <sensor_msgs/point_field_conversion.h>
26 #include <sensor_msgs/point_cloud_conversion.h>
27 #include <math.h>      /* round, floor, ceil, trunc */
28 #include <cmath>     /* std::abs */
29 #include "geometry_msgs/Point.h"
30
31 #include <moveit/planning_scene_interface/planning_scene_interface.h>
32 #include <moveit_msgs/CollisionObject.h>
33
34 // MoveIt
35 #include <moveit/robot_model_loader/robot_model_loader.h>
36 #include <moveit/planning_scene/planning_scene.h>
37
38 #include <moveit/kinematic_constraints/utils.h>
39
40
41
42 typedef pcl::PointXYZ PointT;
43 typedef pcl::PointCloud<pcl::PointXYZ> PointCloud;
44
45
46 //const double cylinderDiameter = 0.169; // Yellow pipe in lab
47 const double cylinderDiameter = 0.25;
48 const double cylinderRadius = cylinderDiameter/2;
49 //const double cylinderLength = 0.53; // Yellow pipe in lab
50 const double cylinderLength = 1.2;
51
52 const float x_min = 3.0;
53 const float x_max = 6.0;
54 const float y_min = 1.0;

```

```

55  const float y_max = 10.0;
56  const float z_min = 0.05;
57  const float z_max = 2.0;
58
59
60  struct AddCylinderParams
61  {
62      /* Radius of the cylinder. */
63      double radius;
64      /* Direction vector towards the z-axis of the cylinder. */
65      double direction_vec[3];
66      /* Center point of the cylinder. */
67      double center_pt[3];
68      /* Height of the cylinder. */
69      double height;
70  };

```

## E.2.4 main.cpp

[https://github.com/evenfl/p26\\_master/blob/master/p26\\_pick\\_and\\_place/src/main.cpp](https://github.com/evenfl/p26_master/blob/master/p26_pick_and_place/src/main.cpp)

```

1  #include "main.h"
2
3  #include <std_srvs/Empty.h>
4
5  typedef pcl::PointCloud<pcl::PointXYZ> PointCloud;
6
7  //PointCloud::Ptr cloud_merged (new PointCloud);
8
9
10  pcl::PointCloud<pcl::PointXYZ>::Ptr cloud_merged (new
    ↪  pcl::PointCloud<pcl::PointXYZ>);
11  pcl::PointCloud<pcl::PointXYZ>::Ptr cloud_filtered (new
    ↪  pcl::PointCloud<pcl::PointXYZ>);
12
13  void callback(const sensor_msgs::PointCloud2ConstPtr& input);
14  void callback_create_octomap(const std_msgs::String::ConstPtr& msg);
15  int counter = 0;
16  bool createOctomap = false;
17  //bool createOctomap = true;
18
19  int main(int argc, char** argv)
20  {
21      ros::init(argc, argv, "collision_map");
22      ros::NodeHandle nh;
23      ros::Publisher pub = nh.advertise<PointCloud>("/master/merged_point_cloud",
    ↪  1);

```



```

24  ros::Publisher pub_octomap_created =
    ↪  nh.advertise<std_msgs::String>("/p26_lefty/octomap_created", 1);
25
26  ros::Subscriber sub1 = nh.subscribe ("/master/jetson1/kinect_decomp", 1,
    ↪  callback);
27  ros::Subscriber sub2 = nh.subscribe ("/master/jetson2/kinect_decomp", 1,
    ↪  callback);
28  ros::Subscriber sub3 = nh.subscribe ("/master/jetson3/kinect_decomp", 1,
    ↪  callback);
29  ros::Subscriber sub4 = nh.subscribe ("/master/jetson4/kinect_decomp", 1,
    ↪  callback);
30  ros::Subscriber sub5 = nh.subscribe ("/master/jetson5/kinect_decomp", 1,
    ↪  callback);
31  ros::Subscriber sub6 = nh.subscribe ("/master/jetson6/kinect_decomp", 1,
    ↪  callback);
32
33  ros::Subscriber sub_create_octomap = nh.subscribe
    ↪  ("/p26_lefty/create_octomap", 1, callback_create_octomap);
34
35  // ros::Publisher pub = nh.advertise<PointCloud>
    ↪  ("/jetson/wp3/points_nocolor", 1);
36
37  ros::Rate rate(20);
38
39  while (ros::ok())
40  {
41      ros::spinOnce();
42      rate.sleep();
43
44      if (counter >= 12 && createOctomap == true)
45      {
46          cloud_merged->header.frame_id = "world";
47
48
49          // Create the filtering object
50          pcl::StatisticalOutlierRemoval<pcl::PointXYZ> sor;
51          sor.setInputCloud (cloud_merged);
52          sor.setMeanK (50);
53          sor.setStddevMulThresh (0.25);
54          sor.filter (*cloud_filtered);
55
56
57          // pcl::PCDWriter writer;
58          // writer.write ("cloud_unfiltered.pcd", *cloud_merged, false);
59          // writer.write ("cloud_filtered_sor.pcd", *cloud_filtered, false);
60

```

```

61
62     //ros::service::waitForService("p26_left/clear_octomap"); //this is
        ↪ optional
63     ros::ServiceClient clearClient =
        ↪ nh.serviceClient<std_srvs::Empty>("p26_left/clear_octomap");
64     std_srvs::Empty srv;
65     clearClient.call(srv);
66     pub.publish(*cloud_filtered);
67
68     //     std::cerr << cloud_merged->size() << std::endl;
69
70     cloud_merged->clear();
71     cloud_filtered->clear();
72     counter = 0;
73     createOctomap = false;
74     std_msgs::String msg;
75     std::stringstream ss;
76     int count = 0;
77     ss << "1" << count;
78     msg.data = ss.str();
79     pub_octomap_created.publish(msg);
80     //     ros::shutdown();
81
82     }
83     else if(createOctomap == false){
84         cloud_merged->clear();
85         counter = 0;
86     }
87
88
89     }
90
91 }
92
93
94 void callback(const sensor_msgs::PointCloud2ConstPtr& input)
95 {
96     //cloud_merged->clear();
97     pcl::PointCloud<pcl::PointXYZ> cloud;
98     pcl::fromROSMsg (*input, cloud); //cloud is the output
99     *cloud_merged += cloud;
100    counter++;
101 }
102
103 void callback_create_octomap(const std_msgs::String::ConstPtr& msg)
104 {

```

```
105 createOctomap = true;
106 }
```

## E.3 Pick-and-place

### E.3.1 p26\_move.py

```
1  #!/usr/bin/env python
2
3  # Software License Agreement (BSD License)
4  #
5  # Copyright (c) 2013, SRI International
6  # All rights reserved.
7  #
8  # Redistribution and use in source and binary forms, with or without
9  # modification, are permitted provided that the following conditions
10 # are met:
11 #
12 # * Redistributions of source code must retain the above copyright
13 # notice, this list of conditions and the following disclaimer.
14 # * Redistributions in binary form must reproduce the above
15 # copyright notice, this list of conditions and the following
16 # disclaimer in the documentation and/or other materials provided
17 # with the distribution.
18 # * Neither the name of SRI International nor the names of its
19 # contributors may be used to endorse or promote products derived
20 # from this software without specific prior written permission.
21 #
22 # THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
23 # "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
24 # LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS
25 # FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE
26 # COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT,
27 # INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING,
28 # BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES;
29 # LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER
30 # CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
31 # LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN
32 # ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
33 # POSSIBILITY OF SUCH DAMAGE.
34 #
35 # Author: Acorn Pooley, Mike Lautman
36
37
38 ## To use the Python MoveIt! interfaces, we will import the
   → `moveit_commander` namespace.
```

```

39  ## This namespace provides us with a `MoveGroupCommander`_ class, a
    ↪ `PlanningSceneInterface`_ class,
40  ## and a `RobotCommander`_ class. (More on these below)
41  ##
42  ## We also import `rospy`_ and some messages that we will use:
43  ##
44
45  import sys
46  import copy
47  import rospy
48  import moveit_commander
49  import moveit_msgs.msg
50  import geometry_msgs.msg
51  from math import pi, atan2, cos, sin, sqrt, asin
52  from std_msgs.msg import String, Int64, Float32, Int8, UInt8, Bool
53  from moveit_commander.conversions import pose_to_list
54  import numpy as np
55  from sensor_msgs.msg import PointCloud2
56  #from moveit_python import PlanningSceneInterface
57
58  def euler_to_quaternion(roll, pitch, yaw):
59
60      qx = np.sin(roll/2) * np.cos(pitch/2) * np.cos(yaw/2) - np.cos(roll/2)
        ↪ * np.sin(pitch/2) * np.sin(yaw/2)
61      qy = np.cos(roll/2) * np.sin(pitch/2) * np.cos(yaw/2) + np.sin(roll/2)
        ↪ * np.cos(pitch/2) * np.sin(yaw/2)
62      qz = np.cos(roll/2) * np.cos(pitch/2) * np.sin(yaw/2) - np.sin(roll/2)
        ↪ * np.sin(pitch/2) * np.cos(yaw/2)
63      qw = np.cos(roll/2) * np.cos(pitch/2) * np.cos(yaw/2) + np.sin(roll/2)
        ↪ * np.sin(pitch/2) * np.sin(yaw/2)
64
65      return [qx, qy, qz, qw]
66
67  def normalize(vec):
68      length = 0.0
69      for i in range(len(vec)):
70          length = length + vec[i]*vec[i]
71      length = sqrt(length)
72      for i in range(len(vec)):
73          vec[i] = vec[i]/length
74      return vec
75
76  def vec_length(vec):
77      length = 0.0
78      for i in range(len(vec)):
79          length = length + vec[i]*vec[i]

```

```

80     length = sqrt(length)
81     return length
82
83 class GraspingPointCandidate:
84     def __init__(self, theta, dirvec, u_init, P_com, P_eef):
85
86         # Rodrigues' rotation formula
87         w = np.cross(u_init, dirvec)
88
89         x1 = cos(theta)
90         x2 = sin(theta)
91
92         u_rot = x1*u_init + x2*w
93
94         u_rot = normalize(u_rot)
95
96         self.d = vec_length([P_eef[0]-(P_com[0]+u_rot[0]),
97                               ↪ P_eef[1]-(P_com[1]+u_rot[1]), P_eef[2]-(P_com[2]+u_rot[2])])
98         self.u = u_rot
99
100 def all_close(goal, actual, tolerance):
101     """
102     Convenience method for testing if a list of values are within a tolerance of
103     ↪ their counterparts in another list
104     @param: goal      A list of floats, a Pose or a PoseStamped
105     @param: actual    A list of floats, a Pose or a PoseStamped
106     @param: tolerance A float
107     @returns: bool
108     """
109     all_equal = True
110     if type(goal) is list:
111         for index in range(len(goal)):
112             if abs(actual[index] - goal[index]) > tolerance:
113                 return False
114
115     elif type(goal) is geometry_msgs.msg.PoseStamped:
116         return all_close(goal.pose, actual.pose, tolerance)
117
118     elif type(goal) is geometry_msgs.msg.Pose:
119         return all_close(pose_to_list(goal), pose_to_list(actual), tolerance)
120
121     return True
122
123 class MoveGroupPickAndPlace(object):
124     """MoveGroupPickAndPlace"""
125     def __init__(self):

```

```

124     super(MoveGroupPickAndPlace, self).__init__()
125
126     # First initialize `moveit_commander` and a `rospy` node:
127     moveit_commander.roscpp_initialize(sys.argv)
128
129     # Instantiate a `RobotCommander` object. This object is the outer-level
130     ↪ interface to
131     # the robot:
132     robot = moveit_commander.RobotCommander()
133
134     # Instantiate a `PlanningSceneInterface` object. This object is an
135     ↪ interface
136     # to the world surrounding the robot:
137     scene = moveit_commander.PlanningSceneInterface()
138
139     # Instantiate a `MoveGroupCommander` object. This object is an
140     ↪ interface
141     # to one group of joints.
142     # This interface can be used to plan and execute motions on the robot:
143     group_name = "p26_lefty_tcp"
144     group =
145     ↪ moveit_commander.MoveGroupCommander(group_name)#,robot_description='/p26_lefty/robot_description',
146     ↪ ns='/p26_lefty')
147
148     ## We create a `DisplayTrajectory` publisher which is used later to
149     ↪ publish
150     ## trajectories for RViz to visualize:
151     display_trajectory_publisher =
152     ↪ rospy.Publisher('p26_lefty/move_group/display_planned_path',
153     ↪ moveit_msgs.msg.DisplayTrajectory,
154     ↪ queue_size=20)
155
156     # Getting Basic Information
157     # ~~~~~
158     # We can get the name of the reference frame for this robot:
159     planning_frame = group.get_planning_frame()
160     print "==== Reference frame: %s" % planning_frame
161
162     # We can also print the name of the end-effector link for this group:
163     eef_link = group.get_end_effector_link()
164     print "==== End effector: %s" % eef_link
165
166     # We can get a list of all the groups in the robot:
167     group_names = robot.get_group_names()
168     print "==== Robot Groups:", robot.get_group_names()

```



```

162
163     # Sometimes for debugging it is useful to print the entire state of the
164     # robot:
165     #print "===== Printing robot state"
166     #print robot.get_current_state()
167     #print ""
168
169     # Misc variables
170     self.box_name = ''
171     self.robot = robot
172     self.scene = scene
173     self.group = group
174     self.display_trajectory_publisher = display_trajectory_publisher
175     self.planning_frame = planning_frame
176     self.eef_link = eef_link
177     self.group_names = group_names
178
179     # Allow replanning to increase the odds of a solution
180     group.allow_replanning(True)
181     # Allow some leeway in position (meters) and orientation (radians)
182     group.set_goal_position_tolerance(0.01)
183     group.set_goal_orientation_tolerance(0.05)
184
185     def go_to_joint_state(self):
186         group = self.group
187
188         ## Planning to a Joint Goal
189         ## ~~~~~
190         joint_goal = group.get_current_joint_values()
191         print "moving from "
192         print joint_goal
193         joint_goal[0] = 0
194         joint_goal[1] = 0
195         joint_goal[2] = 0
196         joint_goal[3] = 0
197         joint_goal[4] = 0
198         joint_goal[5] = 0
199         joint_goal[6] = 0
200         print "moving to "
201         print joint_goal
202
203         # The go command can be called with joint values, poses, or without any
204         # parameters if you have already set the pose or joint target for the
205         ↪ group
206         group.go(joint_goal, wait=True)

```

```

207     # Calling ``stop()`` ensures that there is no residual movement
208     group.stop()
209
210     current_joints = self.group.get_current_joint_values()
211     return all_close(joint_goal, current_joints, 0.01)
212
213 def go_to_pose_goal(self, x, y, z, xd, yd, zd, distance):
214     group = self.group
215
216     ## Planning to a Pose Goal
217     ## ~~~~~
218     ## We can plan a motion for this group to a desired pose for the
219     ## end-effector:
220
221     eef_pose = group.get_current_pose().pose
222     P_eef = [eef_pose.position.x, eef_pose.position.y, eef_pose.position.z]
223     P_com = [x,y,z]
224
225     D = [xd, yd, zd]
226
227     v0 = [0,0,1]
228     if zd > 0.9:
229         v0 = [1,0,0]
230
231     u = np.cross(D, v0)
232     u = normalize(u)
233
234     a = np.array([])
235     for i in range(360):
236         a = np.append(a, GraspingPointCandidate(i*(2*np.pi)/360, D, u, P_com,
237             ↪ P_eef))
238
239     smallest_distance = 1000000
240     for i in range(360):
241         if a[i].d < smallest_distance:
242             smallest_distance = a[i].d
243             a_saved = a[i]
244
245     U = a_saved.u
246
247     W0 = [ -U[1], U[0], 0 ]
248     U0 = np.cross(W0, U)
249     angle_H=atan2(U[1],U[0])
250     angle_P=asin(U[2])
251     angle_B = atan2( np.dot(W0,D) / vec_length(W0), np.dot(U0,D) /
252         ↪ vec_length(U0) )

```

```

251
252     q = euler_to_quaternion( -angle_B+np.pi/4, angle_P, np.pi+angle_H )
253     q = normalize(q)
254
255     #q = euler_to_quaternion(0, np.pi/2, np.pi) # Straight down
256     #q = euler_to_quaternion(0, np.pi, np.pi) # Straight backwards and upside
        ↪ down
257     #q = euler_to_quaternion(0, 0, np.pi) # Straight forwards
258     #q = euler_to_quaternion(0, 0, 0) # Straight backwards
259
260     pose_goal = geometry_msgs.msg.Pose()
261     pose_goal.orientation.x = q[0]
262     pose_goal.orientation.y = q[1]
263     pose_goal.orientation.z = q[2]
264     pose_goal.orientation.w = q[3]
265
266     pose_goal.position.x = x+distance*U[0]
267     pose_goal.position.y = y+distance*U[1]
268     pose_goal.position.z = z+distance*U[2]
269     state = self.robot.get_current_state()
270     group.set_start_state(state)
271     group.set_pose_target(pose_goal)
272
273     ## Now, we call the planner to compute the plan and execute it.
274     plan = group.go(wait=True)
275     #     plan = move_group.plan()
276     #     if plan.joint_trajectory.points: # True if trajectory contains points
277     #         move_success = move_group.execute(plan)
278     #     else:
279     #         rospy.logerr("Trajectory is empty. Planning was unsuccessful.")
280
281     # Calling `stop()` ensures that there is no residual movement
282     group.stop()
283     # It is always good to clear your targets after planning with poses.
284     # Note: there is no equivalent function for clear_joint_value_targets()
285     group.clear_pose_targets()
286
287     # For testing:
288     current_pose = self.group.get_current_pose().pose
289     print "moving to:"
290     print pose_goal
291     return all_close(pose_goal, current_pose, 0.01)
292
293     cylinder_com = geometry_msgs.msg.Point()
294     cylinder_dirvec = geometry_msgs.msg.Point()
295     cylinder = moveit_msgs.msg.CollisionObject()

```

```

296
297 def callback_com(data):
298     global cylinder_com
299     cylinder_com = data
300     rospy.loginfo("Received cylinder center of mass: %f , %f , %f",
301                 ↪ cylinder_com.x, cylinder_com.y, cylinder_com.z)
302
303 def callback_dirvec(data):
304     global cylinder_dirvec
305     cylinder_dirvec = data
306     rospy.loginfo("Received cylinder direction vector: %f , %f , %f",
307                 ↪ cylinder_dirvec.x, cylinder_dirvec.y, cylinder_dirvec.z)
308
309 def callback_CollisionObject(CollisionObject):
310     global cylinder
311     cylinder = CollisionObject
312
313 gripper_position = 0
314
315 def gripper_pos(data):
316     global gripper_position
317     gripper_position = data.data
318     rospy.loginfo("Received gripper position: %i", gripper_position)
319
320 def wait_for_gripper():
321     rospy.wait_for_message("gripper/position", UInt8, timeout=None)
322
323 def main():
324     try:
325         # The anonymous=True flag means that rospy will choose a unique
326         # name for our 'listener' node so that multiple listeners can
327         # run simultaneously.
328         rospy.init_node('pick_and_place', anonymous=True)
329
330         pub = rospy.Publisher('gripper/actuation', UInt8, queue_size=10)
331         pub_find_cylinder = rospy.Publisher('find_cylinder', Bool, queue_size=10)
332         pub_create_octomap = rospy.Publisher('create_octomap', Bool,
333                 ↪ queue_size=10)
334         # rospy.Subscriber("octomap_created", String, gripper_pos)
335         rospy.Subscriber("gripper/position", UInt8, gripper_pos)
336         rospy.Subscriber("cylinder_com", geometry_msgs.msg.Point, callback_com)
337         rospy.Subscriber("cylinder_dirvec", geometry_msgs.msg.Point,
338                 ↪ callback_dirvec)
339
340         # Wait for the cylinder segmentation to complete.

```

```

337 rospy.wait_for_message("cylinder_dirvec", geometry_msgs.msg.Point,
    ↪ timeout=None)
338 rospy.sleep(1)
339 lefty_robot = MoveGroupPickAndPlace()
340
341 pub_create_octomap.publish(1)
342 rospy.wait_for_message("octomap_created", Bool, timeout=None)
343
344 print "=====  

    ↪ operation..."
345 raw_input()
346
347 while not rospy.is_shutdown():
348
349     # Open gripper
350     actuation = 1;
351     pub.publish(actuation)
352     #wait_for_gripper()
353     while gripper_position != 1:
354         pub.publish(actuation)
355         rospy.sleep(1)
356         if gripper_position == actuation:
357             break
358
359     pub_create_octomap.publish(1)
360     rospy.wait_for_message("octomap_created", Bool, timeout=None)
361     rospy.sleep(1)
362
363     lefty_robot.go_to_pose_goal(cylinder_com.x, cylinder_com.y,
    ↪ cylinder_com.z, cylinder_dirvec.x, cylinder_dirvec.y,
    ↪ cylinder_dirvec.z, 0.1925)
364
365     if gripper_position == 1:
366         # Move to gripping position
367         rospy.loginfo("Moving robot to target 3")
368     #     lefty_robot.go_to_pose_goal(cylinder_com.x, cylinder_com.y,
    ↪ cylinder_com.z, cylinder_dirvec.x, cylinder_dirvec.y, cylinder_dirvec.z,
    ↪ 0.14)
369
370     #Close gripper
371     actuation = 2
372     pub.publish(actuation)
373
374     else:
375         # Gripper malfunction
376         rospy.loginfo("Gripper malfunction")

```

```

377
378     # Waiting for completion from gripper
379     wait_for_gripper()
380
381     if gripper_position == 2:
382         rospy.loginfo("Cylinder attatched")
383
384     lefty_robot.group.attach_object('cylinder')
385
386     #print "===== Press `Enter` when the cylinder is physically
387     ↪  attatched..."
388     #raw_input()
389     pub_create_octomap.publish(1)
390     rospy.wait_for_message("octomap_created", Bool, timeout=None)
391     rospy.sleep(1)
392
393     lefty_robot.go_to_pose_goal(cylinder_com.x, cylinder_com.y,
394     ↪ cylinder_com.z+0.1, cylinder_dirvec.x, cylinder_dirvec.y,
395     ↪ cylinder_dirvec.z, 0.1925)
396
397     lefty_robot.go_to_pose_goal(5.42, 9.23-0.13, 0.86+0.05, 0, 0, 1, 0)
398
399     #print( "===== Press `ENTER` to release cylinder")
400     #raw_input()
401
402     #Open gripper
403     actuation = 1
404     pub.publish(actuation)
405     wait_for_gripper()
406
407     lefty_robot.group.detach_object('cylinder')
408
409     print( "===== Press `ENTER` when the cylinder is repositioned")
410     raw_input()
411
412     pub_find_cylinder.publish(1);
413
414     # Wait for the cylinder segmentation to complete.
415     rospy.wait_for_message("cylinder_dirvec", geometry_msgs.msg.Point,
416     ↪ timeout=None)
417
418     # lefty_robot.go_to_joint_state()
419
420     print( "===== Pick-and-place operation complete!")
421 except rospy.ROSInterruptException:
422     return

```

```

419     except KeyboardInterrupt:
420         return
421
422 if __name__ == '__main__':
423     main()

```

## E.4 Gripper

The complete package can be found in Github here: [https://github.com/sindreb/p26\\_gripper](https://github.com/sindreb/p26_gripper)

### E.4.1 actuation.py

This source code can also be found at Github here: [https://github.com/sindreb/p26\\_gripper/blob/main/src/actuation.py](https://github.com/sindreb/p26_gripper/blob/main/src/actuation.py)

```

#!/usr/bin/env python3.7

import time
import sys
import numpy as np
import RPi.GPIO as GPIO
from pyfirmata import Arduino, util
import pigpio as io

# Setup for communication with Arduino
board = Arduino('/dev/ttyACM0') # Define device connection
it = util.Iterator(board)      # Receive data into iterator
it.start()
time.sleep(0.1)                # Wait for iterator to start
current_sense = board.analog[0] # Define analog pin 0 as current sensor
arm_pos = board.analog[5]      # Define analog pin 5 as potmeter

# Setup for GPIO on Raspberry
GPIO.setwarnings(False) # Disable warnings from GPIO
GPIO.setmode(GPIO.BOARD) # Use onboard GPIO on Raspberry
pi = io.pi()
pi.set_mode(22, io.OUTPUT)
pi.set_PWM_frequency(22, 20000) # Set pin 15 (BCM 22) as motor PWM output at 20kHz
GPIO.setup(11, GPIO.OUT) # Set pin 11 as output
GPIO.setup(13, GPIO.OUT) # Set pin 13 as output

max_opening_current = 0.55 # Maximum current value for motor at opening
max_closing_current = 2.4 # Maximum current value for motor at closing
max_opening_pos = 0.53 # Maximum opening position value
min_closing_pos_200 = 0.47 # Minimum closing position value for 200mm cylinder
min_closing_pos_260 = 0.485 # Minimum closing position value for 260mm cylinder

```



```

def CCW(speed):
    GPIO.output(11, GPIO.HIGH)
    GPIO.output(13, GPIO.LOW)
    pi.set_PWM_dutycycle(22, speed/100 * 255)

def CW(speed):
    GPIO.output(11, GPIO.LOW)
    GPIO.output(13, GPIO.HIGH)
    pi.set_PWM_dutycycle(22, speed/100 * 255)

def current_average(): # Finds and calculates average/filtered current
    current_it = 0
    measurement = 40
    while current_sense.read() == None: # Passes through initial None value
        pass
    for current_count in np.arange(1, measurement, 1): # 40 measurements
        current_it += current_sense.read()
        time.sleep(0.1/measurement) # Time between each measurement

    current_avg = np.round(5/0.13*current_it/measurement, 5) # Average of 40 measurement
    return current_avg

def hold(): # Holds torque at the motor by braking to ground
    GPIO.output(11, GPIO.LOW)
    GPIO.output(13, GPIO.LOW)
    pi.set_PWM_dutycycle(22, 0)

def extend(): # Opening of the arms
    print("Opening gripper...")
    current_sense.enable_reporting() # Start reporting from current_sense and arm_pos
    arm_pos.enable_reporting()
    extended = 0
    current_avg = 0
    current_max_count = 0
    arm_pos_max = 0

    while current_sense.read() == None: # Passes through initial None values
        pass
    while arm_pos.read() == None:
        pass

    if arm_pos.read() >= max_opening_pos: # Checks if arms already in open position
        hold()
        extended = 1
        print("Gripper already open")

```

```

while extended == 0: # Runs while arms are not in open position
    CW(20)

    while current_max_count < 3 and arm_pos_max < 2: # Runs while position and current
        current_avg = current_average()
        arm_position = arm_pos.read()

        if current_avg > max_opening_current:
            current_max_count += 1

        if arm_position > max_opening_pos:
            arm_pos_max += 1

        print(current_avg, ", ", arm_position)
    extended = 1

hold()
current_sense.disable_reporting() # Start reporting from current_sense
arm_pos.disable_reporting()

def retract(cylinder_size):
    print("Closing gripper...")
    current_sense.enable_reporting() # Start reporting from sensors
    arm_pos.enable_reporting()
    CCW(30)
    current_avg = 0
    current_max_count = 0
    arm_pos_min_count = 0

    if cylinder_size == 260:
        arm_pos_min = min_closing_pos_260

    elif cylinder_size == 200:
        arm_pos_min = min_closing_pos_200

    while current_sense.read() == None: # Passes through initial None value
        pass
    while arm_pos.read() == None:
        pass

    while current_max_count < 3 and arm_pos_min_count < 2:
        if current_avg > max_closing_current:
            current_max_count += 1

        arm_position = arm_pos.read()
        if arm_position < arm_pos_min:

```

```

        arm_pos_min_count += 1
        current_avg = current_average()
        print(current_avg, ", ", arm_position)
CCW(9) # Hold motor still with sufficient strength
current_sense.disable_reporting() # Stop reporting from sensors
arm_pos.disable_reporting()

```

## E.4.2 gripping.py

This source code can also be found at Github here: [https://github.com/sindreb/p26\\_gripper/blob/main/src/gripping.py](https://github.com/sindreb/p26_gripper/blob/main/src/gripping.py)

```

#!/usr/bin/env python3.7

import rospy
from std_msgs.msg import UInt8
import time
from actuation import extend, retract, hold

pub = rospy.Publisher('p26_lefty/gripper/position', UInt8, queue_size=100)

def grip_procedure(data): # Callback function that stores global actuation variable from r
    global actuation
    actuation = data.data

rospy.init_node('gripper', anonymous=True)
rate = rospy.Rate(10)
rospy.Subscriber("p26_lefty/gripper/actuation", UInt8, grip_procedure)

def open_grip(): # Opening gripper and publish position
    extend()
    gripper_pos = 1 #1 open, 2 closed
    rospy.loginfo("Gripper open")
    pub.publish(gripper_pos)

def close_grip(cylinder_size): # Close gripper and publish position
    retract(cylinder_size)
    gripper_pos = 2 #1 open, 2 closed
    rospy.loginfo("Gripper closed")
    pub.publish(gripper_pos)

def main():

    rospy.wait_for_message("p26_lefty/gripper/actuation", UInt8, timeout=None)

    # Actuation value describes what action to perform, 1 open, 2 and 3 closes to correspo
    if actuation == 1:

```

```

    rospy.loginfo("Opening gripper...")
    open_grip()

elif actuation == 2:
    rospy.loginfo("Closing gripper for 260mm cylinder...")
    close_grip(260)

elif actuation == 3:
    rospy.loginfo("Closing gripper for 200mm cylinder...")
    close_grip(200)

else:
    rospy.loginfo("Malfunction")

try:
    gripper_pos = 0
    pub.publish(gripper_pos) #Publishes that Raspberry initialized and ready for input
    while True:
        main()

except: # Sets all pins low if error occurs, this stops motor
    hold()
    rospy.loginfo("An error occurred, stopping motor")

```