



Norwegian University of  
Science and Technology

# Robotic Cleaning System for Salmon Slaughterhouses

**Emil Dale Bjørlykhaug**

Subsea Technology

Submission date: June 2016

Supervisor: Olav Egeland, IPK

Co-supervisor: Ola Jon Mork, NTNU i Ålesund

Norwegian University of Science and Technology  
Department of Production and Quality Engineering



## MASTEROPPGAVE 2016

**Emil Bjørlykhaug**

**Tittel: Robotisert vaskesystem for lakseslakteri**

**Tittel (engelsk): Robotic cleaning system for salmon slaughterhouses**

### **Oppgavens tekst:**

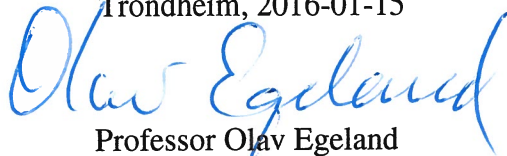
Ved slakting og prosessering av oppdrettslaks er det strenge krav til renhold. Dette utføres manuelt, og gir store kostnader og utfordringer relatert til HMS ved bruk av ulike renholdsmidler. I denne oppgave skal robotisert renhold for slike anlegg studeres, og en demonstrasjon skal utvikles i simulering.

1. Presenter teori for kinematikk og 3D robotsyn.
2. Design utkast av det robotiserte vaskesystemet.
3. Simuler de mest lovende design-utkastene.
4. Sett opp en prosedyre for robotisert renhold som inkluderer inspeksjon og oppdatering av anleggets geometri vha. 3D-syn.
5. Prøv ut 3D-syn-systemet i eksperimenter.

**Oppgave utlevert: 2016-01-15**

**Innlevering: 2016-06-10**

Utført ved Institutt for produksjons- og kvalitetsteknikk

Trondheim, 2016-01-15  
  
Professor Olav Egeland  
Faglærer



# Preface

This is the concluding Master's thesis of the study programme Subsea Engineering at NTNU. The work was carried out between January and June 2016.

This Master's thesis is the beginning of a PhD thesis regarding robotic cleaning, and was brought to my attention when Ola Jon Mork visited Olav Egeland at NTNU IPK in November 2015. My background as an industrial mechanic with a craft certificate, and my background as an Automation Engineer, has influenced the direction of this Master's thesis.

As part of the thesis, several trips to NTNU in Åleund were made where the project was discussed with fellow PhD student Lars André Giske, supervisor Ola Jon Mork and industry partners Amatec AS and SeaSide AS. The group also took a trip to Hannover Messe to find industrial components for the project.

Trondheim, June 6, 2016



Emil Dale Bjørlykhaug



# Acknowledgment

I would like to express my gratitude to Professor Olav Egeland for supervising this project and giving me valuable input when needed. I would also like to thank NTNU in Ålesund and SeaSide AS for funding the project and giving me an interesting project to work on. A special thanks to Associate Professor Ola Jon Mork at NTNU in Ålesund for also supervising and guiding me, and fellow PhD student Lars André Giske for guiding me in this project.

E.D.B





# Abstract

The purpose of this Master's thesis is to look at possible designs for a robotic cleaning solution for salmon slaughterhouses due to the daily need for cleaning and thus high labor costs, and simulate these solutions. In addition, the use of computer vision to aid in such a task is investigated. This thesis is part of a research project that aims at being able to clean a whole slaughterhouse. The focus on this thesis was mainly on cleaning an electric stunner, but many of the challenges with cleaning such a component are transferable to cleaning a whole slaughterhouse.

The main emphasis of this Master's thesis has been on developing different designs for the robotic cleaning solution. Finding a way to move a robot around in a salmon slaughterhouse is a big challenge, and various solutions to this have been developed and investigated. This thesis has both looked at alternatives for the robot, and also looked at smart solutions for the suspension of a robot. Both existing components already on the market have been evaluated, and building certain components from scratch have been investigated. A custom, modular robot have also been designed, with the goal of making a robot that is lighter and has a longer reach than any other robot in its weight class. Some of the designed solutions have been simulated, and it can be concluded that while probably none of the designed solutions will be the final solution when the project is done, some of the designs will greatly influence the next steps for the project. Further work will require trial and error and prototyping to reach a final design.

Another aspect of the robotic cleaning is the robot trajectories. Cleaning is usually performed every day after production has stopped. In order to be able to clean even though a component has been moved, requiring new robot trajectories, computer vision has been tested to see if the position and orientation of components can be established with such an accuracy that the robot trajectories can be updated according to the computer vision data, and resume the cleaning. It can be concluded that computer vision can be accurate enough to calibrate new trajectories if a component is moved, given a good enough sensor. It was also tested to see if computer vision can be used to find unwanted obstacles in the environment. The experiments showed good results, and it can be concluded that unwanted objects can be found as long as they are not too small.

To alleviate some of the problems with manually programming the hundreds of robot commands necessary to clean a whole electric stunner, some time was spent investigating the possibility to generate robot trajectories automatically from a CAD model of the electric stunner. This showed promising results for a simple geometry like a box on a table, but the complex geometries for the components in a salmon slaughterhouse proved to be difficult. Further work would be required to achieve a satisfactory result.



# Sammen drag

Formålet med denne masteroppgaven er å se på mulige løsninger for et robotisert vaskesystem for lakseslakteri. Det er strenge renholds krav i lakseslakteri, og de må derfor vaskes daglig. Dette fører til store utgifter årlig, noe som mulig kan reduseres vha. et robotisert vaskesystem. I tillegg vil en med et robotisert vaskesystem bedre kunne dokumentere vaske kvalitet vha. kamera og logging av robotbevegelser og vann/kjemikalie-forbruk. Denne oppgaven har hatt mest fokus på å vaske en elektrisk bedøver, men erfaringer gjort på dette området vil bane vei for å kunne vaske hele slakteri, noe som er målet med prosjektet.

Hovudfokuset i denne oppgaven har vært på det mekaniske designet til vaskesystemet. Det er store utfordringer knyttet til å bevege en robot rundt i et lakseslakteri, og ulike løsninger på dette problemet har vært undersøkt. Enkelte av løsningene er nesten bare basert på eksisterende komponenter på markedet, mens andre løsninger er mer basert på å bygge opp alt fra grunnen av. En av løsningene har også innbært å bygge en modulær robot opp fra grunnen av for å kunne spare vekt, samtidig som den har tilfresstillende rekkevidde.

Det kan sannsynligvis konkluderes med at mens ingen av de tenkte løsningene vil representere det endelige designet, så har de hjulpet å funnet kursen for videre arbeid. Sannsynligvis må det en del "prototyping" med ekte løsninger til for å skaffe mer kunnskap om utfordringene, før en endelig kan konkludere med et design.

Et viktig aspekt ved en robotisert vaskeløsning er bevegelsene til roboten. For å unngå at små forflytninger av utstyret skal føre til at vaskingen ikke kan gjennomføres, har mulighetene for å kunne bruke 3D-syn for å korrigere robotbanene vært undersøkt. Formålet med testene er å se om nøyaktigheten til 3D-syn er bra nok. Det kan konkluderes med at 3D-syn er nøyaktig nok til å kunne korrigere robotbaner om en sensor med høy presisjon blir brukt. Det ble også undersøkt om 3D-syn kan brukes til å finne uønskede objekter i arbeidsområdet til roboten. Eksperimentene ga gode resultater, og det kan konkluderes med at 3D-syn kan finne uønskede objekter, gitt de er av en viss størrelse.

Siden det er nødvendig med veldig mange robotbaner når en komplisert komponent som en elektrisk bedøver skal vaskes, ble muligheten for automatisk generering av robotbaner undersøkt. Dette gav lovende resultat for en enkel del som en firkantet boks, men det viste seg at en komplisert komponent som en elektrisk bedøver trenger mye bearbeiding før automatisk robotbane-generering kan fungere. Det kan dermed konkluderes med at dette vil ta for mye tid for dette prosjektet, og må legges til side.



# Contents

Preface . . . . .	ii
Acknowledgment . . . . .	iv
Summary . . . . .	vi
Terms and Abbreviations . . . . .	xviii
<b>1 Introduction</b>	<b>1</b>
1.1 Background . . . . .	1
1.2 Objectives . . . . .	1
1.3 Structure of the thesis . . . . .	2
<b>2 Background/Theory</b>	<b>3</b>
2.1 Introduction . . . . .	3
2.2 Kinematics . . . . .	3
2.3 Computer vision . . . . .	5
2.3.1 Introduction . . . . .	5
2.3.2 Different sensors . . . . .	5
2.3.3 Passthrough filter . . . . .	6
2.3.4 Statistical outlier filter . . . . .	6
2.3.5 Downsampling . . . . .	7
2.3.6 Planar Segmentation . . . . .	7
2.3.7 Ray tracing . . . . .	7
2.3.8 Feature registration . . . . .	7
2.3.9 Descriptors . . . . .	8
2.3.10 Sample Consensus Initial Alignment . . . . .	11
2.3.11 ICP . . . . .	12
2.3.12 Segment differences . . . . .	12
2.3.13 Normals . . . . .	12
2.3.14 Surface reconstruction . . . . .	12
2.4 Visual Components . . . . .	13
2.5 ROS . . . . .	14
2.6 PLC . . . . .	14
2.7 Water-based hydraulics . . . . .	14
<b>3 Mechanical design</b>	<b>15</b>
3.1 Introduction . . . . .	15
3.2 Challenges . . . . .	15
3.3 Solution 1 . . . . .	16
3.4 Solution 2 . . . . .	16
3.5 Solution 3 . . . . .	18
3.6 Solution 4 . . . . .	19

3.6.1	Vertical linear axis . . . . .	19
3.6.2	Horizontal linear axis . . . . .	21
3.6.3	Slewing ring . . . . .	22
3.6.4	Fold . . . . .	22
3.6.5	Robot . . . . .	24
3.7	Solution 5 . . . . .	24
3.7.1	Custom robot . . . . .	26
3.7.2	Curved guide . . . . .	29
3.7.3	Control system . . . . .	30
3.8	First prototype . . . . .	31
3.8.1	Robot . . . . .	31
3.8.2	Vertical linear axis . . . . .	31
3.8.3	Slewing ring . . . . .	31
3.8.4	Nozzle . . . . .	32
3.8.5	Fold . . . . .	32
3.8.6	Control system . . . . .	32
3.8.7	Final design of prototype . . . . .	33
3.9	Discussion . . . . .	36
<b>4</b>	<b>Simulation</b>	<b>37</b>
4.1	Introduction . . . . .	37
4.2	Programming movable robot base . . . . .	37
4.3	Programming the robot paths . . . . .	39
4.4	Point cloud import . . . . .	41
4.5	Results . . . . .	42
4.6	Discussion . . . . .	42
<b>5</b>	<b>Computer vision</b>	<b>45</b>
5.1	Introduction . . . . .	45
5.2	Object detection experiment . . . . .	45
5.3	Object detection results . . . . .	47
5.4	Segmentation experiment . . . . .	50
5.5	Segmentation results . . . . .	51
5.6	Discussion . . . . .	53
<b>6</b>	<b>Trajectory generation</b>	<b>55</b>
6.1	Introduction . . . . .	55
6.2	Method . . . . .	55
6.3	Result . . . . .	55
6.4	Discussion . . . . .	56
<b>7</b>	<b>Conclusion</b>	<b>61</b>
7.1	Future work . . . . .	61
	<b>Bibliography</b>	<b>62</b>
	<b>Appendix</b>	

<b>A Python script for solution 4</b>	<b>1</b>
<b>B The computer vision computation class</b>	<b>7</b>
<b>C The main class for CV</b>	<b>19</b>
<b>D Digital Appendix</b>	<b>35</b>





# List of Figures

2.1	Curve fitting with RANSAC, where outliers does not affect the result ( <a href="#">Wiki, 2016</a> ).	7
2.2	The two pipelines for object recognition and pose estimation ( <a href="#">Aitor Aldoma and Rusu, 2012</a> ).	8
2.3	Darboux frame with the three PFH angles ( <a href="#">Rusu, 2009</a> ).	9
2.4	The pair of points around point $p_s$ (here marked as $p_q$ ) and the relationships calculated (marked with lines) ( <a href="#">Rusu, 2009</a> ).	9
2.5	The influence region diagram for FPFH. Query point $p_q$ is connected only to its direct k-neighbours (gray circle). Each direct neighbour are then connected to its own neighbours. The result is then weighted from both the histogram of the query point and the histogram of its neighbours histogram ( <a href="#">Rusu, 2009</a> ).	10
2.6	The viewpoint $v_p$ and the associated angles between the normals and the viewpoint direction ( <a href="#">Radu Bogdan Rusu, 2010</a> ).	11
2.7	The normal estimation on the left has a good search radius. The right normal estimation has too big search radius, causing the curvature around the edge to affect the normals ( <a href="#">Rusu, 2009</a> ).	13
3.1	The fingers of the electric stunner. High voltage causes fish skin to burn in, making them difficult to keep clean. The electric stunner is here depicted with the fingers in an upright position for ease of cleaning. During normal operations, the arrays with fingers ly parallel to the conveyor. CAD model courtesy of SeaSide AS.	16
3.2	Rough sketch of the first solution, presented by Amatec AS. Here the planned railway in the roof can be seen.	17
3.3	Solution 2, looking a lot like a overhead crane. This solution would deliver unmatched accessibility for the robot.	18
3.4	Solution 3, here from the simulation in Visual Components.	19
3.5	Solution 4 is almost the same as solution 3, with the exception of the linear axis underneath the roof. CAD model of robot from <a href="#">KUKA (2016)</a> .	20
3.6	EPX 60 linear axis from RK Rose+Krieger, picture from datasheet.	20
3.7	Two alternatives from igus, but none of them which fully fullfills the requirements right out of the box.	21
3.8	PRT slewing ring from igus, picture taken from datasheet ( <a href="#">Available at PRT (2016)</a> ).	22
3.9	A system for folding the vertical arrangement. Here illustrated without the pins for the joints and cylinder attachments.	23
3.10	IGUS modular robot joints.	25
3.11	A worm gear.	25
3.12	Robot made from igus robot joints. CAD model of joint available at <a href="#">RoboLink (2016)</a> .	26

3.13	A long reach is possible with a modular robot. . . . .	27
3.14	The position of the vertical part of the robot that will result in the most torque on the horizontal guide. . . . .	29
3.15	Different types of spray pattern from a nozzle (Nozzle, 2016). . . . .	29
3.16	Rollon Curviline, picture taken from datasheet, available at Curviline (2016). . . . .	30
3.17	Rendering of first prototype. CAD models of UR10, EPX 60, PRT and servos from GrabCAD (2016), RoseKriegerEPX (2016), PRT (2016) and ServoCAD (2016), respectively. . . . .	34
3.18	The top assembly of the first prototype. . . . .	35
4.1	Under Node Feature Tree the different geometries in the Node (Link) Svivel are displayed. . . . .	38
4.2	Two of the most important aspects of programming custom functionality in VC: Links and joints. . . . .	38
4.3	The setup of the ServoController. . . . .	39
4.4	How the One to One Interface should be set up when making a movable robot base. . . . .	40
4.5	The statements made by the Python script. . . . .	40
4.6	A scan of the electronics lab at NTNU Ålesund imported in VC. . . . .	41
4.7	Meshes made from the scan of the room at NTNU Ålesund. . . . .	42
5.1	Experiment setup. . . . .	46
5.2	The modified pipeline. . . . .	47
5.3	A fairly good feature matching transformation shown in red. ICP shown in blue. . . . .	48
5.4	A less good feature matching transformation shown in red. ICP shown in blue. . . . .	49
5.5	The reference cloud without a box on the table. . . . .	51
5.6	The target cloud with a box on the table. . . . .	52
5.7	The result after the segmentation. The box is the only cluster bigger than 500 points, and thus the only one showed. . . . .	52
5.8	False depth reading at the edge of the box. . . . .	54
5.9	Erroneous reading due to concavities. . . . .	54
6.1	The box with the calculated normals. All normals point out of the box (difficult to see in 2D). . . . .	56
6.2	The normals for the table with the box. The points captured by the Kinect are white, red dots are the calculated robot positions. . . . .	57
6.3	A point cloud of 1 million points representing the electric stunner, with normals calculated for every 1000th point. . . . .	58
6.4	The problem with calculating the normals for a complex geometry. The normals on both the inside and outside of the plate point outwards. . . . .	59

# List of Tables

3.1 Comparison of robots . . . . .	24
------------------------------------	----



# Terms and Abbreviations

**Manipulator** A robot represented as a chain of rigid bodies which are connected by means of revolute or prismatic joints

**Revolute joint** A joint which provides single-axis rotation

**Prismatic joint** A joint which provides linear sliding

**Base** The start point coordinate system of a robot

**End-effector** The end point coordinate system of a robot

**TCP** Tool Center Point

**TCP/IP** Transmission Control Protocol/Internet Protocol

**PLC** Programmable Logic Controller

**PCL** Point Cloud Library

**CAD** Computer-aided design

**ICP** Iterative Closest Point

**MLS** Moving Least Squares

**C++** Programming language

**Python** Programming language

**ROS** Robot Operating System

**ToF** Time-of-Flight

**ST** Structured Light

**PFH** Point Feature Histogram

**FPFH** Fast Point Feature Histogram

**VFH** Viewpoint Feature Histogram

**DOF** Degree of freedom

**SIFT** Scale-invariant feature transform



# Chapter 1: Introduction

## 1.1 Background

The fishing industry is a multibillion dollar industry in Norway (DN, 2015). Salmon alone has a yearly revenue of over 40 billion NOK (Akvakultur, 2014). But the salmon industry is not without problems. With an ever increasing amount of lice and the constant danger of listeria infections during production and stricter requirements from the government each year, the industry has to find new ways of improving their breeding and slaughtering.

To cope with the high risk of listeria infections, the slaughterhouses must be thoroughly cleaned each day. This is done by cleaning crews at night after the production has been shut down. This costs millions in labor each year for every factory. In addition, there are high expenses in chemicals and water.

SeaSide AS is a company that produces machines for the salmon industry and has recognised the cleaning problem. They use the principal of design for cleaning when designing their components. This means that the components and machines should be easy to clean, with no inaccessible parts and locations where bacteria can grow unrestricted. But to take this a step further, SeaSide wanted to look into a robotic cleaning solution, making the cleaning crew obsolete. The project takes aim at being able to clean the whole factory after it is completed, but will initially only focus on cleaning the machines and components delivered by SeaSide.

That is where this thesis comes in. As part of this bigger project to design and develop a novel automated robot cleaning solution, this thesis will look into design and simulation of such a solution and the possibilities for using computer vision to check the state of the factory prior to cleaning - checking the position of equipment and that no unwanted objects are left inside the factory.

## 1.2 Objectives

The main objectives for this project work are:

1. Present theory for kinematics and 3D computer vision.
2. Design drafts of the robotic cleaning solution.
3. Simulate the most promising design drafts in either Delmia or Visual Components.
4. Set up a procedure for how 3D computer vision can detect positions of equipment and find obstacles that should not be present.
5. Do experiments on the computer vision system at the laboratory at the *Department of Production and Quality Engineering (IPK), NTNU*.

### 1.3 Structure of the thesis

The structure of this thesis does not follow the typical IMRAD (Introduction, Method, Results and Discussion) structure due to the varying nature of the different parts of the project. Instead, the different parts of the project are divided into separate chapters with both method, results and discussion in each chapter, as follows:

- Chapter 2 describes the relevant background and theory used in this project work. This includes kinematics, 3D computer vision, ROS.
- In Chapter 3 the different mechanical design solutions are presented. Both the thought process and the result from the work are presented in this chapter.
- Chapter 4 shows how to set up the simulation, and also presents the results from the simulations. The simulation results are given in the appendix.
- Chapter 5 includes the work regarding computer vision, both the method and results. The viability of the results are also discussed.
- In order to reduce the time required to program the robot path, automatic trajectory generation is investigated. This is presented in Chapter 6
- In Chapter 7, a brief conclusion of this project work is presented, along with recommendations for further work.
- The Appendix includes design drafts, source code and simulation videos.



# Chapter 2: Background/Theory

## 2.1 Introduction

This chapter presents the theory and background necessary for understanding the content of this thesis. However, some basic foreknowledge of mathematics and physics are required. Since this thesis will be about robotics and computer vision, theory for these fields will be presented.

## 2.2 Kinematics

A manipulator (robot) can be represented as a chain of rigid bodies (*links*) connected by means of revolute or prismatic *joints*. The start point of the chain is constrained to a *base*, while the *end-effector* is mounted on the end. The motion of the structure is obtained by composition of the elementary motions of each link with respect to the previous one. In order to manipulate an object in space, it is necessary to have a way of describing the *end-effector* position and orientation (Siciliano et al., 2009).

Kinematics is the branch of classical mechanics that describe the motion of joints and bodies without taking into account the masses of the bodies nor the forces action on them. Kinematics can be used to mathematically express the position of an end-effector of a robot as a function of the joints of said robot (forward kinematics). Inverse kinematics is exactly the opposite, an expression for the joints of the robot given a end effector position. Inverse kinematics can often have multiple solutions, while forward kinematics only have one solution.

### Rotation matrix

A rotation matrix is a way of describing rotation (orientation) about an arbitrary axis in space with respect to a reference frame. The 3x3 rotation matrices about axis X, Y and Z is given below (Siciliano et al., 2009).

$$R_x(\gamma) = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos\gamma & -\sin\gamma \\ 0 & \sin\gamma & \cos\gamma \end{pmatrix} \quad (2.1)$$

$$R_y(\beta) = \begin{pmatrix} \cos\beta & 0 & \sin\beta \\ 0 & 1 & 0 \\ -\sin\beta & 0 & \cos\beta \end{pmatrix} \quad (2.2)$$

$$R_z(\alpha) = \begin{pmatrix} \cos\alpha & -\sin\alpha & 0 \\ \sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 1 \end{pmatrix} \quad (2.3)$$

Consider a point in vector coordinates

$$p = \begin{pmatrix} p_x \\ p_y \\ p_z \end{pmatrix} \quad (2.4)$$

which is located in a Cartesian coordinate system with respect to a reference frame. This point can be rotated about an arbitrary axis by multiplying a rotation matrix with the point. Consider the point  $p$  to be rotated by 45 degrees by its Z-axis (Siciliano et al., 2009).

$$p_r = \begin{pmatrix} \cos 45 & -\sin 45 & 0 \\ \sin 45 & \cos 45 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} p_x \\ p_y \\ p_z \end{pmatrix} \quad (2.5)$$

### Homogeneous transformation matrix

As explained above, it is necessary to describe both position and orientation for the end-effector with respect to a reference frame. The position (translation) and orientation can be described by the use of the 4x4 *homogeneous transformation matrix*. Consider two joints, joint 0 and joint 1. The translation and rotation between the two joints is given as

$$A_1^0 = \begin{bmatrix} R_1^0 & p_1^0 \\ 0^T & 1 \end{bmatrix} \quad (2.6)$$

where  $R_1^0$  is the rotation from joint 0 to joint 1 and  $p_1^0$  is the translation (Siciliano et al., 2009). To get the transformation from the base of a robot and the end-effector, one can set up the transformation matrix for each joint and multiply them together to get the total transformation matrix.

### Euler angles

Euler angles is a way of representing orientation of a rigid body. This is done by using a set of three angles (Siciliano et al., 2009).

$$\phi = [\varphi \quad \vartheta \quad \psi]^T \quad (2.7)$$

The two most common sets of Euler angles are ZYZ angles and ZYX (roll, pitch, yaw) angles. The RPY angle set originates from the nautical field. The resulting rotation from a RPY set is obtained by the following sequence of rotation

- Rotate the reference frame by the angle  $\psi$  about x-axis (yaw). Rotation described by equation 2.1.
- Rotate the reference frame by the angle  $\vartheta$  about y-axis (pitch). Rotation described by equation 2.2.

- Rotate the reference frame by the angle  $\varphi$  about z-axis (roll). Rotation described by equation 2.3.

which can be written as.

$$R(\phi) = R_z(\varphi)R_y(\vartheta)R_x(\psi) \quad (2.8)$$

$R(\phi)$  is a 3x3 matrix, which can be written as

$$R = \begin{pmatrix} r_{11} & r_{12} & r_{13} \\ r_{21} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{pmatrix} \quad (2.9)$$

The resulting inverse solution from  $R(\phi)$  is given in two different sets of range for  $\vartheta$ .

$\vartheta$  in the range  $(-\pi/2, \pi/2)$ :

$$\begin{aligned} \varphi &= \text{Atan2}(r_{21}, r_{11}) \\ \vartheta &= \text{Atan2}(-r_{31}, \sqrt{r_{32}^2 + r_{33}^2}) \\ \psi &= \text{Atan2}(r_{32}, r_{33}) \end{aligned} \quad (2.10)$$

$\vartheta$  in the range  $(\pi/2, 3\pi/2)$ :

$$\begin{aligned} \varphi &= \text{Atan2}(-r_{21}, -r_{11}) \\ \vartheta &= \text{Atan2}(-r_{31}, -\sqrt{r_{32}^2 + r_{33}^2}) \\ \psi &= \text{Atan2}(-r_{32}, -r_{33}) \end{aligned} \quad (2.11)$$

## 2.3 Computer vision

### 2.3.1 Introduction

This theory section will for the most part be about 3D computer vision. To not confuse 3D computer vision with 3D video using stereo camera, the correct term is range imaging (often referred to as 2.5D). Most of the algorithms presented here are a part of PCL (Point Cloud Library).

### 2.3.2 Different sensors

In the field of range imaging there are a lot of different sensor utilizing different physical properties to capture the depth of an image. The two most common ways principals for capturing a range image is Structured Light (ST) and Time-of-Flight (ToF).

#### Structured Light

The structured light technique uses a 2D camera and a projected light pattern. This light pattern can take many forms, the most common being stripes. Depending on the surface of the captured geometry, the projected light will distort when viewed from different viewpoints,

making it possible to calculate the surface geometry. Two main sources for the projected light pattern exists, line lasers or projectors. When using line laser it is common to have two line lasers, one on each side of the camera to avoid obstructions. The laser will then have to move, either solo or with the camera, to scan the whole surface. With a projector as the structured light source, then outputted pattern can be changed without any movement. A common setup with a projector is to have the projector in the middle, and a camera on each side of it (David Fofi, 2004).

### ToF

Time-of-Flight sensors calculate the depth of points by measuring the time of flight for the light to hit a point. Since the speed of light is known, the calculation is a trivial task. However, since the speed of light is so fast, very small errors in time measurement will cause significant error in depth measurement. Advantages for TOF sensors is that they capture the whole scene in an instant. A scene with a depth of 100 meters is captured in less than a microsecond. TOF sensors are suitable for real-time application, reaching frame rates of up to 160Hz. Another big disadvantage of the TOF camera is that it gets false readings at depth edges and concavities (Sergi Foix and Torras, 2011). The false readings from depth edges is possible to remove with filters if the distances between the foreground and background are significant. However, the false readings due to concavities can be difficult to remove, as it is unknown if they are a measuring errors, or actually how the scanned object looks like.

### Kinect

The Kinect is a 3D camera from Microsoft. It is composed of a 2D camera, a range imaging sensor and a microphone. The first version of the Kinect used structured light for the range imaging, while the Kinect v2 uses ToF for range imaging. The range imaging of the Kinect 1 was of poor quality as it used a IR sensor with low resolution (Antoine Lejeune and Verly, 2011). The Kinect v2 was a big improvement as it uses a much higher resolution ToF sensor for the range imaging (with a corresponding higher resolution 2D camera), despite the drawback a TOF sensor has (Zugara, 2014) (IGN-board, 2013).

#### 2.3.3 Passthrough filter

In computer vision a passthrough filter is a filter that removes points that has x-, y- and z-values outside a given criteria. One can for instance use passthrough filtering to remove all points that has a z-value greater than 3 meters (Passthrough, 2016).

#### 2.3.4 Statistical outlier filter

When capturing a point cloud, some noise will almost always be present. StatisticalOutlierRemoval filter removes these points by calculating for all points the mean distance to the nearest points. The algorithm assumes all points follow a Gaussian distribution with a mean and standard deviation. If a point has a mean distance which is greater than the global distances mean and standard deviation, they are treated as outliers and removed (StatisticalOutlierRemoval, 2016).

### 2.3.5 Downsampling

Downsampling is used to reduce the number of points in a point cloud. In combination with Statistical Outlier Removal, this can be used to reduce computation time and remove noise. The way downsampling works is by creating a voxel grid (a grid of small boxes) of a given size throughout the whole point cloud. If a box contains one or more points, the centroid of all those point will be calculated and replace the other points. If the box contains no points, no point will be added to the downsampled point cloud (Downsampling, 2016).

### 2.3.6 Planar Segmentation

Planar segmentation can be used to filter out planes in the point cloud which are not interesting. The method in PCL is called SACsegmentation (Segmentation, 2016). This algorithm uses RANSAC to detect what belongs to the plane, and what does not (inliers and outliers). RANSAC stands for random sample consensus and is a method for estimating parameters for a mathematical model from a set of data that contains outliers, e.g. noise (Martin A. Fischler, 1980). An example of RANSAC is a curve fitting from a set of points, Figure 2.1.

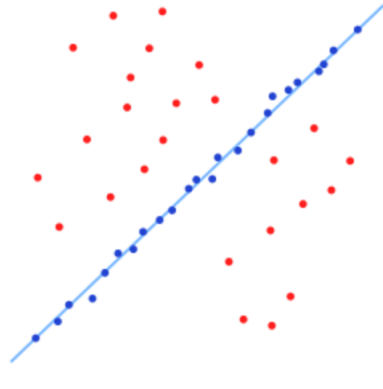


Figure 2.1: Curve fitting with RANSAC, where outliers does not affect the result (Wiki, 2016).

The same method can be used to find other surfaces than planes, like spheres, cylinders, etc.

### 2.3.7 Ray tracing

In order to match an object in point cloud to a CAD model, ray tracing of said model is necessary. Ray tracing is the act of viewing a given object from a viewpoint, and calculate how the model would look from that viewpoint. Imagine a sphere around the CAD model. This sphere is then tessellated into a number of planes, the center of these planes representing each of the viewpoints for the CAD model. The point cloud of the object from that viewpoint is then calculated (Appel, 1968).

### 2.3.8 Feature registration

Feature registration, sometimes called object recognition, is in 3D computer vision the process of finding similarities (features) in two point clouds and then matching them together based on those similarities. This will find a pose estimation and thus being able to roughly align the

point clouds according to each other. Then the transformation between the point clouds can be fine tuned with Iterative Closest Point (ICP). This is known as the *pipeline* (Aitor Aldoma and Rusu, 2012). Illustrated in Figure 2.2

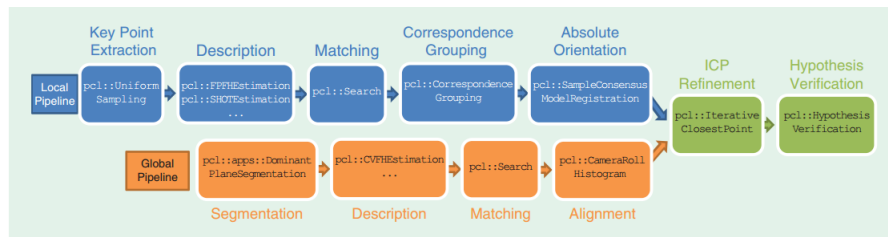


Figure 2.2: The two pipelines for object recognition and pose estimation (Aitor Aldoma and Rusu, 2012).

### 2.3.9 Descriptors

In computer vision, a descriptor, also known as a feature descriptor, is a description of visual content in an image or a video. They describe characteristics such as shape, color or texture. For range images, a descriptor computes an expression for the geometry either around a point, or for the whole geometry. Descriptors can be used to match similarities of two images. This can then be used for feature registration and matching (Pipeline, 2016).

#### Local descriptors

A local descriptor is a descriptor that computes the geometry around a given point. Since the local descriptor will calculate the geometry around a given point with no notion of what kind of shape said geometry is, it is wise to choose points that have some key features in their local geometry, i.e. keypoints. Many local descriptors can also be used as global descriptors, setting the search radius high enough that it will cover all the points in a cluster, it will encode the whole geometry (Pipeline, 2016).

#### Global descriptor

Unlike a local descriptor, a global descriptor encodes object geometry, and uses all the points in a cluster to do this. It is therefore required to extract objects into clusters before the descriptor is calculated. This is done by segmentation (Pipeline, 2016).

#### PFH

Point Feature Histogram (PFH) is a local descriptor used to encode the geometry of an object in a cluster. It works by generalizing the curvature around a point  $p_s$ . This is done by looking at neighbour points and their respective normals in a radius around  $p_s$ . The difference between the normal of point  $p_s$  and a normal of any of the neighbour points can be expressed by three angles. If one attaches a Darboux frame to the point  $p_s$  (see Figure 2.3) with a vector  $p_t - p_s$  to the point  $p_t$ , and given their respective normals, one can calculate the three PFH angles  $\alpha$ ,

$\vartheta$  and  $\phi$  as follows:

$$\begin{aligned}\alpha &= v \cdot n_t \\ \phi &= u \cdot \frac{\mathbf{p}_t - bf p_s}{d} \\ \theta &= \arctan(w \cdot bf n_t, u \cdot bf n_t)\end{aligned}\tag{2.12}$$

where  $d$  is the Euclidean distance between  $\mathbf{p}_s$  and  $\mathbf{p}_t$  (Rusu, 2009).

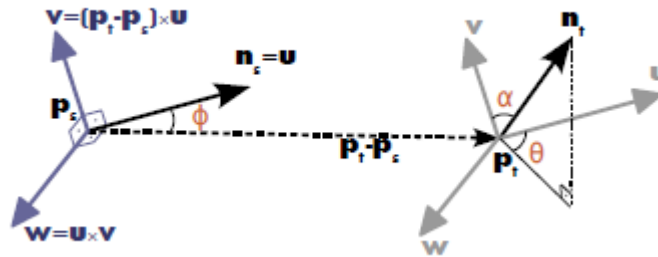


Figure 2.3: Darboux frame with the three PFH angles (Rusu, 2009).

The quadruplet  $(\alpha, \phi, \theta$  and  $d)$  is then computed for each pair of points in the sphere around the point  $p_s$  as illustrated in Figure 2.4.

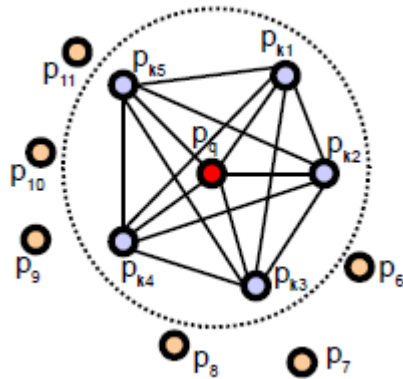


Figure 2.4: The pair of points around point  $p_s$  (here marked as  $p_q$ ) and the relationships calculated (marked with lines) (Rusu, 2009).

All the quadruplets for the pairs around point  $p_s$  are then binned into a histogram. The histogram is divided into a set of subdivision, and the number of occurrences falling inside each subdivision's value range are then added for all the point pairs. It can often be beneficial to remove the  $d$ -parameter in the quadruplet for 2.5D scans as the point density will vary depending on the distance of the scanned object in the scene.

### FPFH

Fast Point Feature Histogram (FPFH) is a simplification of PFH to reduce computation time. Instead of computing vectors and histograms for all the point pairs inside the sphere around point  $p_q$ , just the point pairs between the point  $p_q$  and its neighbours are computed (this is called Simplified Point Feature Histogram). In addition, for each point that is connected to  $p_q$ , the vectors and histograms for its  $k$ -nearest neighbours inside a radius equal to the initial one, are computed. The final histogram is weighted in the following manner (Rusu, 2009):

$$FPFH(\mathbf{p}_q) = SPFH(\mathbf{p}_q) + \frac{1}{k} \sum_{i=1}^k \frac{1}{\omega_k} \cdot SPFH(\mathbf{p}_k) \quad (2.13)$$

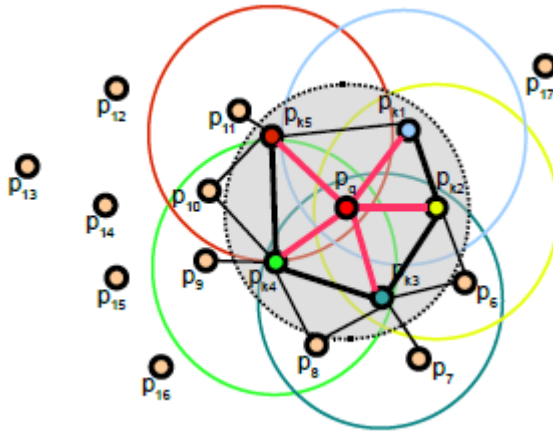


Figure 2.5: The influence region diagram for FPFH. Query point  $p_q$  is connected only to its direct  $k$ -neighbours (gray circle). Each direct neighbour are then connected to its own neighbours. The result is then weighted from both the histogram of the query point and the histogram of its neighbours histogram (Rusu, 2009).

### VFH

PFH and FPFH are invariant to scale and object pose, making them unusable to calculate pose. The Viewpoint Feature Histogram (VFH) is a descriptor based on FPFH designed to be able to encode pose in addition to the geometry. VFH uses FPFH as a global descriptor by having a search radius big enough to encode the whole geometry, and in addition it computes additional statistics between the viewpoint direction and the normals estimated at each point, see Figure 2.6. This additional feature is then binned into an extended histogram.

This extended feature of VFH makes it able to determine which ray traced model best matches the viewpoint of a model in a scene.

### SIFT

SIFT is a local descriptor that originally was designed for 2D images. SIFT stands for Scale-Invariant Feature Transform, and works by four major steps (Lowe, 1999):



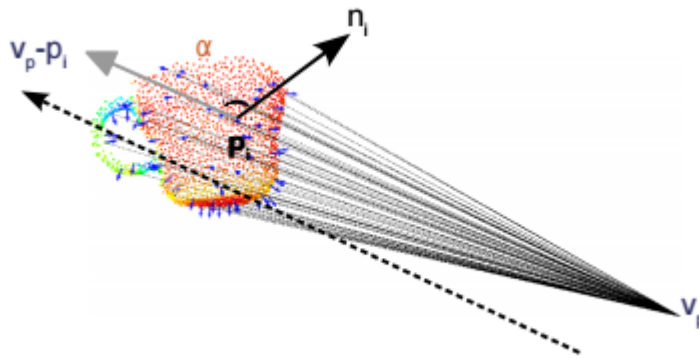


Figure 2.6: The viewpoint  $v_p$  and the associated angles between the normals and the viewpoint direction (Radu Bogdan Rusu, 2010).

1. Scale-space extrema detection: searches over all scales and image locations. Implemented using a Difference-of-Gaussian function to identify potential interest points that are invariant to scale and orientation.
2. Keypoint localization: at each candidate location, a detailed model is fit to determine location and scale. Keypoints are selected based on measures of their stability.
3. Orientation assignment: One or more orientations are assigned to each Keypoint location based on local image gradient directions. All future operations are performed on image data that has been transformed relative to the assigned orientation, scale and location for each feature, thereby providing invariance to these transformations.
4. Keypoint descriptor: The local image gradients are measured at the selected scale in the region around each keypoint. These are transformed into a representation that allows for significant levels of local shape distortion and change in illumination.

This approach generates a large amount of features. This makes the algorithm not ideal for real-time applications. To transform the algorithm over to 3D, (Alex Flint and Hengel, 2007) uses a 3D version of the Hessian to select interest points.

### 2.3.10 Sample Consensus Initial Alignment

Sample Consensus Initial Alignment (hereafter SAC-IA) is a method for registration (aligning two point clouds). SAC-IA is based on the Greedy Initial Alignment method (Radu Bogdan Rusu, 2008), but is designed to reduce the computational time. This is done by reducing the number of combinations of correspondence pairs. Instead, large numbers of correspondence candidates are sampled and ranked by the following steps (Radu Bogdan Rusu, 2009):

1. Select  $s$  sample points from  $P$  while making sure that their pairwise distances are greater than a user-defined minimum distance  $d_{min}$ .
2. For each of the sample points, find a list of points in  $Q$  whose histograms are similar to the sample points' histogram. From these, randomly select which will be considered that sample points' correspondence.

3. Compute the rigid transformation defined by the sample points and their correspondences and compute an error metric for the point cloud that computes the quality of the transformation.

### 2.3.11 ICP

Iterative Closest Point (ICP) is an algorithm to minimize the difference between two point clouds. Two point clouds are fed into the algorithm, a reference point cloud which is kept still, and a source point cloud which is transformed to best match the reference. ICP works in 4 steps (Paul J. Besl, 1992):

1. For each point in the source cloud, find the nearest point in the reference cloud.
2. Estimate the combination of rotation and translation using a mean squared error cost function.
3. Transform the source point cloud using the obtained transformation.
4. Re-iterate.

There are many different versions of the ICP algorithm. Standard version, with normals, non-linear, etc. (Pomerleau et al., 2013).

ICP requires a rough initial alignment of the two point clouds to converge to a correct result. If they are not overlapping, the algorithm will not converge.

### 2.3.12 Segment differences

In PCL there is a function called `SegmentDifferences`, which takes two point clouds and a distance as input. If point  $a$  in point cloud one is within a given distance of point  $a$  in point cloud two it is dismissed, or else it is kept. All the kept points are then used to form a new point cloud. This new point cloud represents the difference between the two input point clouds (`SegmentDifferences`, 2016).

### 2.3.13 Normals

Surface normals can be computed by looking at a point and its neighbours. The accuracy of the normals are dependent on the search radius around the point. Too big search radius and the estimated normals will be affected by curvatures in the neighbourhood. Figure 2.7 shows the effect of both a good and a bad search radius.

The direction of a normal is not explicitly given. To solve this, when calculating normals with PCL, the viewpoint can be set, and for each computed normal the scalar product between the vector from the point to the viewpoint, and the normal, is computed. If the scalar is less than zero, the direction of the normal is changed.

### 2.3.14 Surface reconstruction

Surface reconstruction in computer vision is to create a continuous mesh from a point cloud. A common way to represent a surface in computer science is to build it up using triangles, be

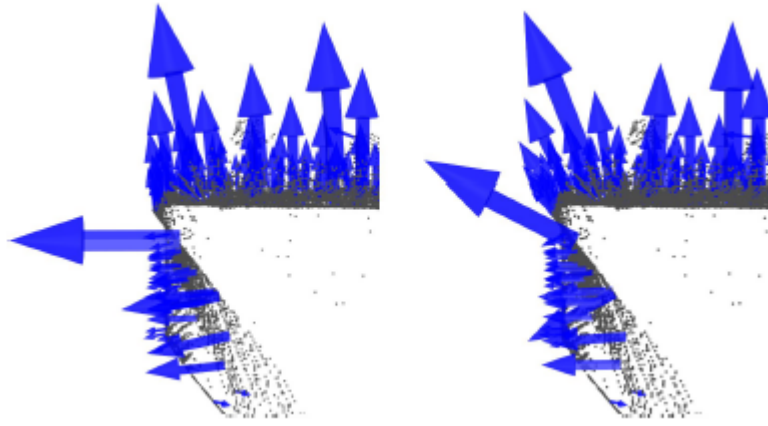


Figure 2.7: The normal estimation on the left has a good search radius. The right normal estimation has too big search radius, causing the curvature around the edge to affect the normals (Rusu, 2009).

it CAD models or video game characters. A common file format for representing CAD models is STL. This file format also uses triangles to represent surface geometries.

PCL has a function called GreedyProjectionTriangulation. This function is based on the paper "On Fast Surface Reconstruction Methods for Large and Noisy Point Clouds" (Zoltan Csaba Marton, 2009). The problem with growing a mesh of a 2.5D data set (a term for point clouds captured by a range image sensor from one viewpoint) is that the point density is not uniform. The closer the object is to the sensor, the more points it will have. Traditional surface reconstruction techniques does not account for this and will fail if the variation of point densities are big (Gopi and Krishnan, 2002).

The function creates a mesh by selecting a point and then searches the point's  $k$  nearest neighbours in a sphere with a radius  $r = \mu * d_0$ .  $d_0$  is the distance to the nearest neighbour, while  $\mu$  is a user-specified value. This way, the function adapts to the local point density, negating the non-uniform point density problem. Another feature that help the function reduce the problem with non-uniform point density is that it interpolates extra points in between existing ones using Robust Moving Least Squares algorithm (R. B. Rusu and Beetz, 2008). This re-sampling also help by generating more points where denser regions are needed, such as edges.

## 2.4 Visual Components

Visual Components is a simulation software for factory automation. The goal of using software like Visual Components is to plan a factory, and reduce installation and setup time for said factory. Visual Components has the ability to simulate in 3D everything from linear axes and pallets, to robots and workers. Out of the box Visual Components has a library of industrial components, like robots from known vendors, fences, tables and conveyors. If something is missing it is also possible to load CAD models and program with Python scripting how these should behave (VisualComponents, 2015).

## 2.5 ROS

ROS is short for Robot Operating System. ROS is an open-source meta-operating system for robotics which runs on Unix-based operating systems. It provides tools and libraries for building, writing and running code across multiple computers (ROS, 2014).

## 2.6 PLC

PLC stands for Programmable Logic Controller and is a widely used industrial automation component. PLCs can be used for controlling and regulating processes in the industry. It can read sensors such as pressure sensors, temperature sensors, limit switches, etc. and control DC motors, servo motors, pneumatic valves, etc. (Parr, 1998).

## 2.7 Water-based hydraulics

The most common fluid in hydraulics is oil. This is due to the lubricating and non-compressive abilities of oil. However, with the addition of the right additives, water-based hydraulics can often be a viable option. Water-based hydraulics are less dangerous to the environment, and it is fire proof. Newer technology has allowed one to develop hydraulic components that can be used with tap water, without the need of additives (Hydraulics, 2006).

# Chapter 3: Mechanical design

## 3.1 Introduction

One of the biggest challenges of this project was to design a smart suspension solution for the robot. The suspension will be mounted underneath the roof, and should be able to make the robot disappear after it has done its task. In a perfect world the robot will come from nowhere, clean the machine, and then disappear into nowhere. A big problem with hanging something like a robot underneath the roof is the structural integrity of the roof. Most of the factories where such a cleaning solution can be implemented will be very old, thus having varying quality of the roof.

One of the key principals when designing the solutions was to use ready-made industrial components. This is to reduce both the time spent designing, and more importantly, reduce the time which will be spent building the prototypes. Designing all the components from scratch is time-consuming, and will offer no advantages if there is already a product on the market which has adequate specifications. Making components from scratch may be viable if mass production will ensue a successful prototype, but this will probably not be the case with this system, as it has a relatively small market.

This chapter will present the work done on designing the mechanical solution. Some of the solutions are just in the concept phase, with no mechanical calculations for the different parts of the solution, while the more promising solutions have to a certain extent been fully designed with calculations done, and also with suggestions of parts from suppliers.

## 3.2 Challenges

There are several challenges with respect to the suspension, not just the structural integrity of the roof. One of the biggest dangers in the fish industry is listeria. This bacteria will grow wherever it can, and can possibly infect a whole batch of produce, resulting in total wash down of the factory being necessary.

Important aspects of the cleaning system:

- Low installation time.
- Little change of existing factory infrastructure.
- Must not introduce a risk of growing listeria.
- Reach of installation.
- Possible robot positions with the given suspension.
- Enough stiffness in the suspension if scrubbing is necessary.
- Must not pose a danger for contaminating the factory (oil leakage etc.).

- Price.

Probably the most challenging part of the cleaning is the "fingers" of the electrical stunner, see Figure 3.1. Between the "fingers" and the conveyor there is a voltage potential. When a salmon passes through the fingers it is electrocuted till stunned. This process causes some of the skin of the fish to burn in on the "fingers", requiring harsh chemicals to be cleaned. A possible solution is to scrub the fingers with a brush, thus reducing the amount of chemicals needed. However, this will require the fingers to be locked in position, which is something that has to be done by the robot.

The robot will also need to clean the underside of the equipment. This implies that the robot will need a horizontal reach of over 1 meter when it is over the equipment, but also when it is underneath the equipment.

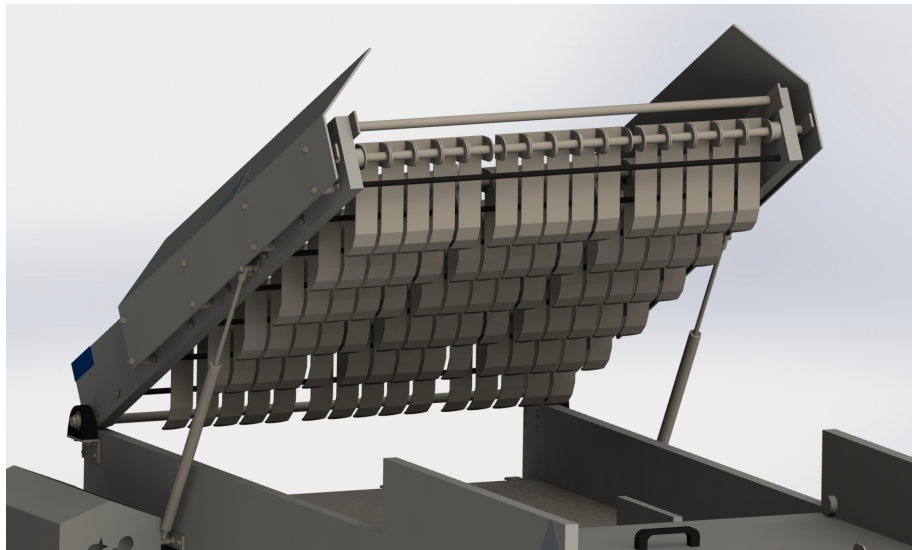


Figure 3.1: The fingers of the electric stunner. High voltage causes fish skin to burn in, making them difficult to keep clean. The electric stunner is here depicted with the fingers in an upright position for ease of cleaning. During normal operations, the arrays with fingers lie parallel to the conveyor. CAD model courtesy of SeaSide AS.

### 3.3 Solution 1

The first solution for the suspension of the robot was a custom built railway hanging from the roof. This railway would then be built in such a way that the robot hanging from it would reach all the components which should be cleaned. This solution was presented by Amatec AS, but was shut down by SeaSide AS due to the complexity of installing the suspension. A sketch of the solution made by Amatec can be seen in Figure 3.2

### 3.4 Solution 2

The second solution is the installation of two parallel runways with a beam spanning the gap, much like an overhead crane. In combination with a z-axis slider, this solution will give the robot unmatched positioning ability. This solution was thought of at a meeting at Sykkylven where

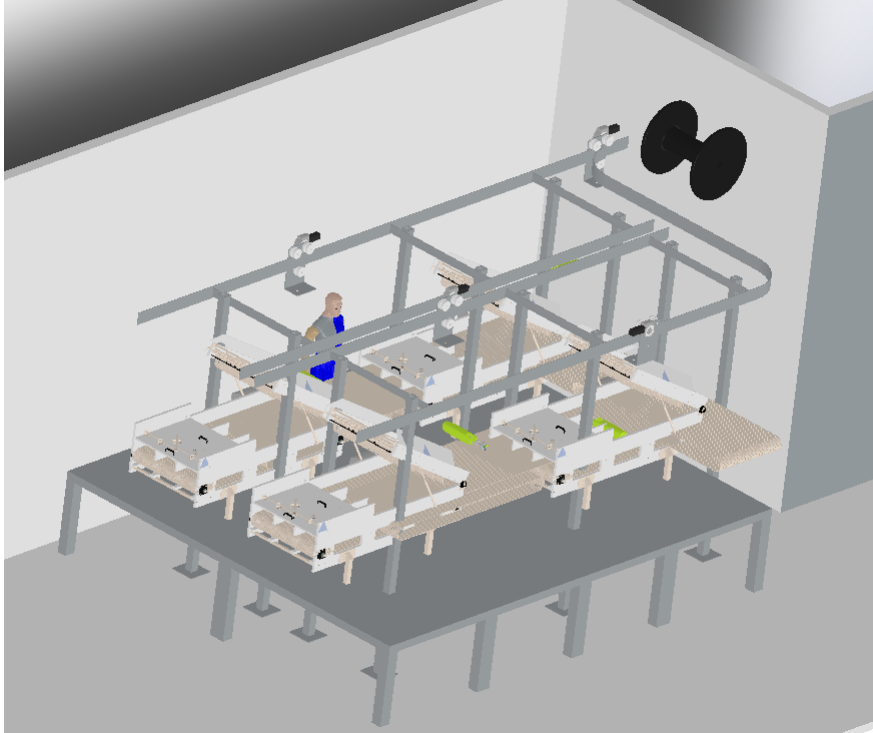


Figure 3.2: Rough sketch of the first solution, presented by Amatec AS. Here the planned railway in the roof can be seen.

PhD students, supervisors and industry partners were present. This solution does have some of the drawbacks as the first solution; it will require a lot of effort to install, and it may be impractical in a large slaughter house where the walls are far apart and a lot of installation is already hanging down from the roof, e.g. hoses and electrical wiring. A concept of this solution can be seen in Figure 3.3.

A big advantage with this solution, other than the unmatched positioning ability, is the rigidity. There is no requirements for the structural integrity of the roof, as the mechanical suspension can stand on the floor. The rigidity will also help the precision of the robot positioning.

Another big challenge of this solution is the z-axis. Using a standard linear z-axis will require a lot of ceiling height if the robot will have to steer clear the equipment in the fully retracted position of the z-axis, while also having the possibility to go low enough that the robot can stretch the full length of its envelope when it is underneath the equipment. This implies a stroke of at least 2 meters, which will require a ceiling height of at least 4 meters if the y-axis (the beam spanning the gap between the parallel runways) is 2 meters from the ground (which it will have to be to be able to steer clear the equipment). This is not a viable solution, as the ceiling heights in the factories are rarely that high, and in addition the height of the y-axis will probably have to be higher than 2 meters.

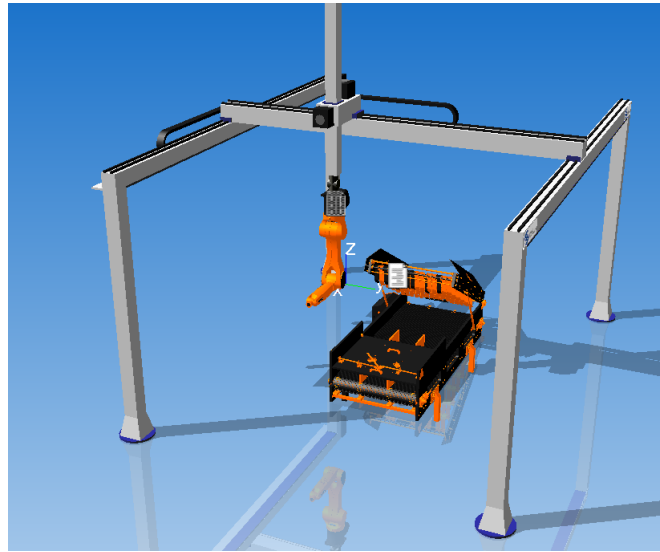


Figure 3.3: Solution 2, looking a lot like an overhead crane. This solution would deliver unmatched accessibility for the robot.

### 3.5 Solution 3

To reduce the complexity and time necessary for installation, a third solution was designed. The first draft of this solution was a single platform mounted underneath the roof, with a joint to swing down a beam with a linear axis on (referred to as the folding system hereafter). This beam could then rotate at the mount in the roof, and the robot would be placed on the linear axis on the beam. In addition, at the bottom of the beam there would be an extendable foot which would secure the beam onto the floor. The solution can be seen in Figure 3.4.

Using an extendable foot on the beam will remove all the torque on the base in the roof when the foot is planted on the floor, thus stiffening up the vertical beam and making the robot positioning more accurate. However, a fairly high amount of torque will be affecting the base when lowering the beam with the vertical axis and the robot. This torque can be resolved by having a bigger base mounted underneath the roof.

From simulations it seems that this solution will be able to clean two electrical stunners that are one meter apart, see Appendix D. This could be sufficient for a first edition of the cleaning robot.



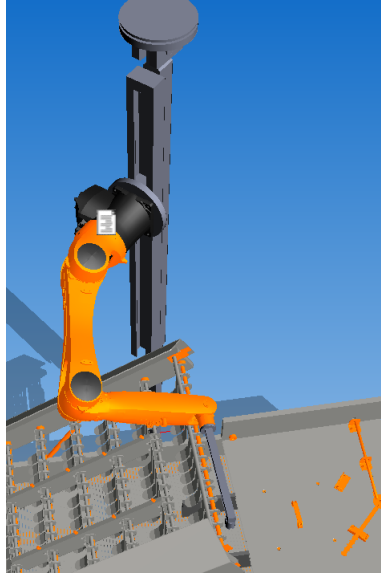


Figure 3.4: Solution 3, here from the simulation in Visual Components.

## 3.6 Solution 4

Solution 3 was presented to supervisors Olav Egeland and Ola Jon Mork, Fjordlaks and Seaside. It was well received, but everyone agreed its biggest drawback was the lack of disappearing after use, both with listeria infections in mind, and also the possible obstacle it will be when production resumes after the cleaning period. This led to the design of solution 4. Instead of having a mounted platform underneath the roof, it would be more desirable to have a linear axis mounted underneath the roof, allowing the robot to park in a garage after it is done cleaning. The robot can also clean the horizontal linear axis as it is on its way to the garage. The solution is illustrated in Figure 3.5. As this seemed to be a very viable solution, further work was done on the individual components.

### 3.6.1 Vertical linear axis

The vertical linear axis will allow better positioning of the robot. Critical demands for the vertical linear axis:

- Adequate motion velocity.
- Sufficient load carrying capacity in the vertical direction.
- Handle sufficient moment around the y-axis.
- Corrosion resistant.
- Self-lubricating.

The weight of a KR10 is 54kg. With additional equipment and wiring it is safe to assume the whole package will not exceed 70 kg. A safe assumption for the center of mass for the robot with equipment will be around 500mm, giving the robot a moment of 343Nm if it is standing perpendicular to the vertical axis. While it is not set in stone that the KR10 will be

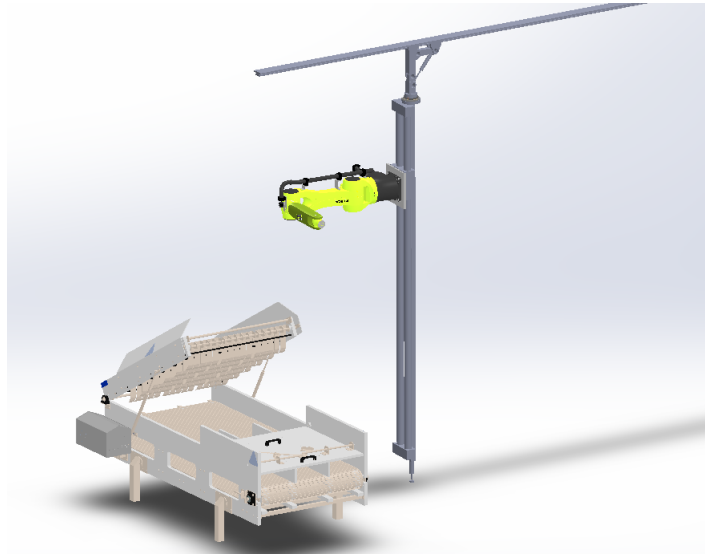


Figure 3.5: Solution 4 is almost the same as solution 3, with the exception of the linear axis underneath the roof. CAD model of robot from [KUKA \(2016\)](#).

the chosen robot, it will be a good estimate for the loads on the linear axis. A comparison of different robots can be seen in Table 3.1, and with the exception of the Motoman robot, which is probably too heavy, the KR10 is the heaviest.

The EPX series of linear axes from RK Rose+Krieger seemed to suit our needs, with the EPX 60 being the right compromise between strength, weight and price. The datasheet for the linear unit can be found at [RoseKrieger \(2016\)](#), a picture of the linear axis can be seen in Figure 3.6.

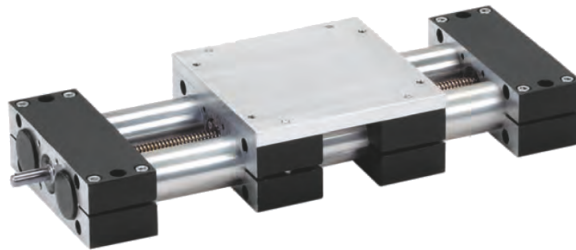


Figure 3.6: EPX 60 linear axis from RK Rose+Krieger, picture from datasheet.

In addition to the linear axis itself, a servomotor is required to drive the axis. A linear axis can be considered as a gear for lifting where the gear ratio is  $2\pi/\text{lift}$  per rotation. Calculation for torque of the servo with some safety margin, and considering the No-load torque from the data sheet of the EPX:

$$Torque_{req} = 1000N * \frac{5mm}{2\pi} + 1.5Nm = 2.3Nm \quad (3.1)$$

An appropriate servo would be a OMRON R88MK75030F. This servo delivers a rated torque of 2.4Nm, with a maximum torque of 7.1Nm (Datasheet available at [OmronServo \(2016\)](#)). Since



(a) Igus drylin ZLW which is corrosion resistant and lubrication free, but can't handle high enough loads (IGUS, 2016).



(b) Igus drylin T which can handle the load, but does not have propulsion (IGUS, 2016).

Figure 3.7: Two alternatives from igus, but none of them which fully fullfills the requirements right out of the box.

the linear axis is self locking, no brake on the servo will be needed, and the weight of the servo is just 2.3kg. This servo delivers a rpm of 3000. Another possibility is to use a smaller servo and a gear as the linear axis' maximum rpm is 80.

### 3.6.2 Horizontal linear axis

There are a lot of linear axes on the market suitable for the roof. With the leg stabilizing the column with the vertical linear axis and the robot, the horizontal linear axis does not need to handle any moment. One of the key features for the horizontal linear axis is the same as for the vertical linear axis; it has to be self lubricating. This is even more essential for the horizontal linear axis, as it will be present in the factory not just during cleaning, but also during production. The length of the horizontal linear axis is hard to decide on a general level as the factories and the placement of the equipment varies a lot. A minimum required stroke of 3 meters is fair to assume.

When it comes to lubrication free linear axes, igus seems to be the number one supplier on the market. A suitable axis for the roof could be a igus drylin with belt drive. This system is lubrication free, corrosion resistant and delivers the speed and precision needed. Figure 3.7a shows a igus drylin ZLW. However, the linear axes with belt propulsion has a limited loading capacity. The biggest ZLW can handle a load of 100kg. This would probably not be sufficient as the weight of the vertical linear axis in all likelihood exceeds 46kg (100kg load capacity, 54kg robot). An igus drylin T, Figure 3.7b, would handle high enough loads, but this guide is without propulsion, requiring a custom propulsion system. A third alternative would be to use a igus drylin system with a lead screw. The problem then is the stroke, as this is limited to 500mm.

As will be discussed in Section 3.6.4, in order to eliminate the torque acting on the horizontal linear axis when the beam is folded up, it may be a good solution to have two sliders on the linear guide. This will require a custom linear axis, as none of the off the shelf linear axes provided by igus have dual sliders. Having dual sliders might also increase the load bearing capacity of drylin ZLW, making it a possible solution. However, this will require igus to be willing to deliver a custom linear axis, which they did seem willing to do when visiting their stand at the Hannover Messe.

### 3.6.3 Slewing ring

The slewing ring will accompany the linear axis in giving the robot a bigger working envelope. Requirements for the slewing ring is the same as for the vertical linear axis. In addition it must have an external gear ring to allow for rotation of the lower part of the assembly. A suitable slewing ring would be the igus PRT slewing rings. Their rings have bearing material made of plastic, making them corrosion resistant and lubrication free.

The slewing ring will have to withstand similar forces and torque as the vertical linear axis. Worst case scenario for the forces are when the beam is hanging straight down, before the foot has been planted on the floor. The whole assembly will then be suspended through the slewing ring. The robot has a weight of 54 kg, but the weight of the EPX linear axis is not stated in the datasheet, so an assumption has to be made. A safe assumption will be a total force of 2000N. The worst case scenario is when the beam is at top position. To minimize the torque in the slewing ring, the robot will have to be as close to it as possible. Measurements taken in SolidWorks give a distance of approximately 300mm from the slewing ring to the center of the robot base. In addition, the linear axis on the beam will introduce a torque at the slewing ring. Assuming a weight of 100kg and the mass center at the middle of the length, i.e. 1500mm, this combined will give a torque requirement on the slewing ring of 1630Nm. This is just outside the spec of the PRT-01-100 slewing ring. However, the slewing ring will not need to rotate at this load, making it reasonable to assume a overload of less than 10% will be acceptable. A PRT slewing ring can be seen in Figure 3.8.



Figure 3.8: PRT slewing ring from igus, picture taken from datasheet (Available at [PRT \(2016\)](#)).

### 3.6.4 Fold

The vertical arrangement with the linear axis and the slewing ring will be over 3000mm tall, making it difficult to navigate it. Similar to solution 3, a function for folding the arrangement would be necessary to implement, in this case to ease the transportation along the horizontal linear axis. A solution for how this can be solved can be seen in Figure 3.9. This system uses

two guides on the same slider with attachments of the hydraulic cylinder and slave arm at the same joint resulting in no moment at the sliders, only tensile and compressive forces.

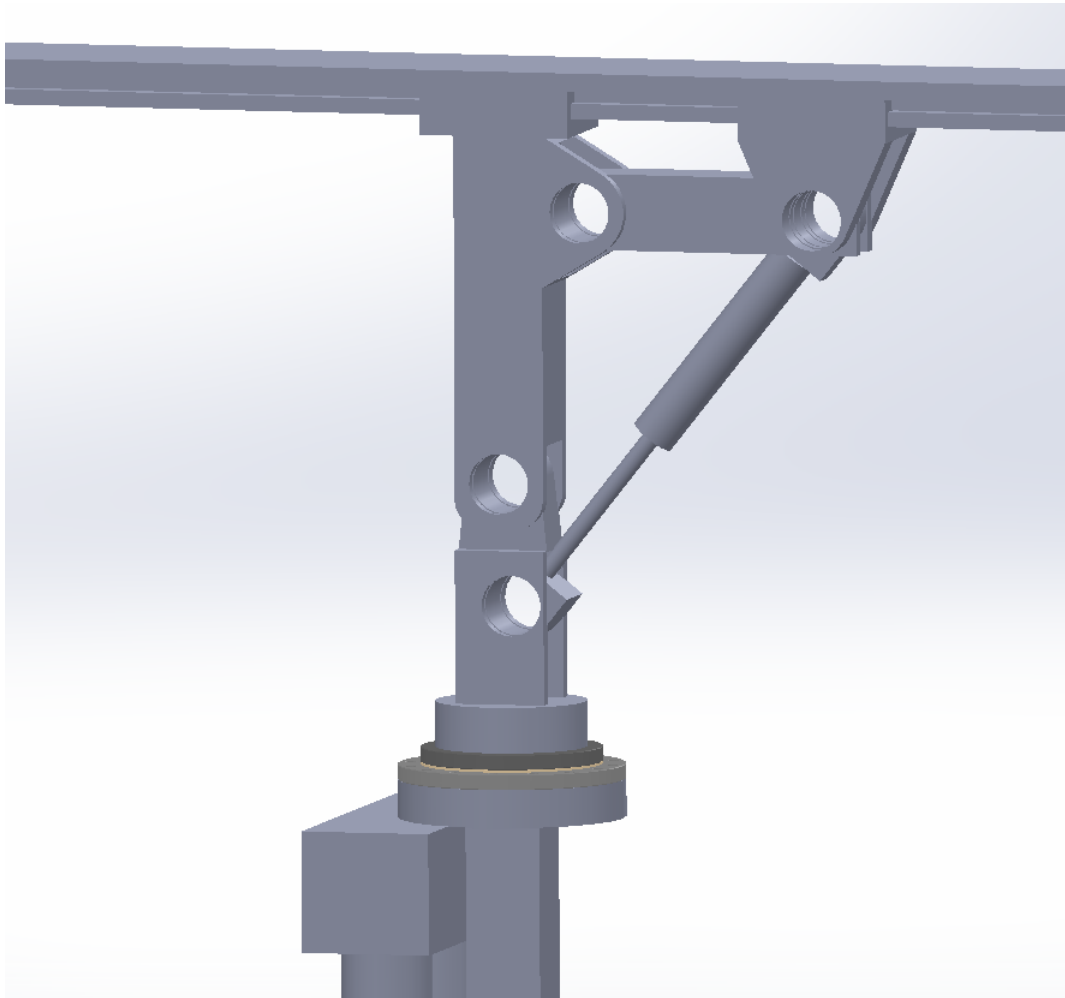


Figure 3.9: A system for folding the vertical arrangement. Here illustrated without the pins for the joints and cylinder attachments.

Since the robot shall be working in a clean environment, and the danger of leakage is always present with hydraulics, it may be beneficial to use water-based hydraulics. Especially if the hydraulics are pure water (with no additives) the environmental benefit will be substantial. A leakage will then cause no threat to the slaughterhouse. The main drawback of pure water hydraulics versus conventional oil hydraulics is the price. Since water is corrosive, and has poor lubrication properties compared to oil, the surfaces of the movable hydraulic parts need to be corrosion resistant. In addition, water has a lower viscosity than oil, requiring tighter clearances for seals, which combined with the corrosiveness and poor lubrication properties results in the need for very hard surfaces with smooth finishes, which drives the price up (Hydraulics, 2006).

The maximum force the hydraulic cylinder will have to deliver can be calculated for when the beam is at the top position. With the attachment of the cylinder as shown in the Figure 3.9 the spacing between the hinged joint of the beam and the attachment of the cylinder is 80mm.

To deliver a torque of 1630Nm, the force from the cylinder will have to be 20375N. To reduce the cylinder force, the attachment can be moved further from the hinge.

### 3.6.5 Robot

There are a lot of factors for choosing the right robot. It will have to be agile (many joints) with a good reach, while still keeping a low weight making it possible to suspend the robot from a roof. Other important qualities of the robot include corrosion resistance, high IP grade (preferably IP67), easy programming, preferably internal guide for cabling and hoses. The price of the robot is also an important factor.

Alternatives:

- KUKA KR10 IP67
- Motoman EPX 1250
- Stäubli TX60L
- Universal Robot UR10
- FANUC LR Mate 200iD/7LC

A comparison of the different robots can be seen in Table 3.1. This table does not compare precision as this is acceptable for all the robots.

Table 3.1: Comparison of robots

	Reach [mm]	Payload [kg]	Weight [kg]	IP grade
KUKA KR10	1101	10	54	67
Motoman EPX 1250	1256	5	110	Protective cover
Stäubli TX60L	920	2	52	65
Universal Robot UR10	1300	10	29	Protective cover
FANUC LR Mate	911	7	27	67

From this comparison two of the robots stand out as good alternatives, the KR10 and UR10. These have the highest payload, while also having the highest reach-to-weight ratio. The UR10 seems to be the best choice if a protective cover does not introduce any problems.

## 3.7 Solution 5

The two major difficulties with solution 4 is the reach. Two major factors limit the reach: the reach of the robot, and the horizontal linear axis in the roof. A better solution would be a robot with a longer arm, and an axis/slider in the roof that can have curvature.

The problem with robots that has a long reach is weight. A robot with a reach of 1000mm often weighs somewhere around 50-60kg, and can handle 5-15kg at the end effector. As soon as this reach is extended, say to 2000mm, the weight of the robot increase dramatically, often with an associated increase in weight-handling at the end effector. The only alternative to get a robot with a long reach, will still keeping weight down, is to design one ourselves. IGUS has already started working on making a modular robot where joints are made of the slewing rings,

where the inner ring has a ring gear on the outer surface. This ring gear is then mated to a screw, making a worm gear inside the assembly, Figure 3.10. The downside with this IGUS product, is that it is made of plastic, resulting in a maximum torque for the joint of 35Nm. A worm gear is shown in Figure 3.11.

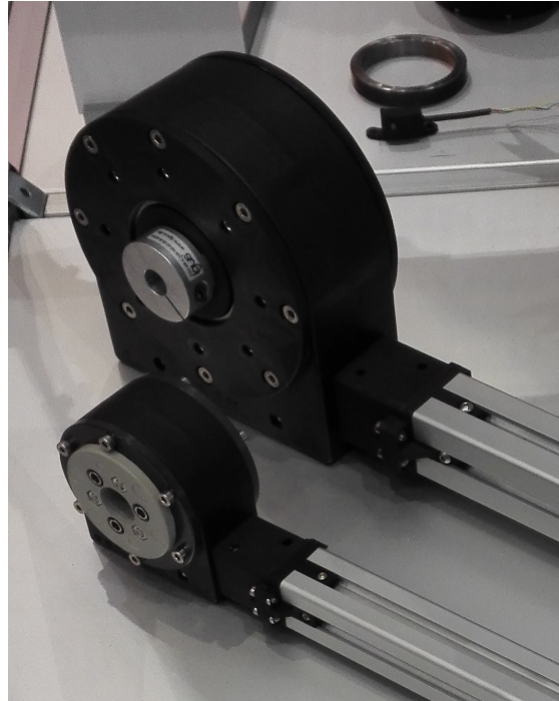


Figure 3.10: IGUS modular robot joints.

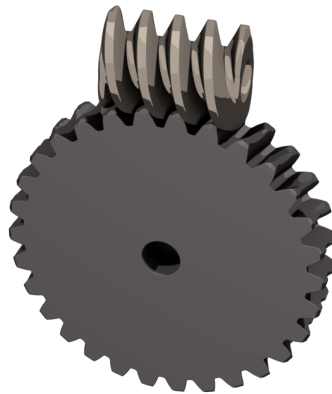


Figure 3.11: A worm gear.

Linear axes with propulsion in the form of a screw or a belt is a common product in the automated industry. Guides with curvature is also a common product. But a guide with curvature and propulsion at the same time is not that common. There are already some solutions on the market for this problem, but they use magnets for propulsion, much like a maglev train. This solution is not valid for the fish industry since the environment is highly corrosive, and the magnets are not very corrosion resistant. This requires a custom solution,

not totally different from the solution proposed by Amatec.

Another benefit with making a custom robot and railway is that the robot can be made in such a way that there is no need for a vertical linear axis. This feature can be implemented in the robot, and if the robot is made light enough, there may also be no need for a leg coming down from the glider all the way to the floor, since the moments will be reduced by the lighter weight.

### 3.7.1 Custom robot

There are several approaches for designing the robot. One of the simplest forms of the robot would be making a 4DOF robot which would be mounted on a vertical linear axis such as in solution 4. As discussed, it would be better if the robot had the functionality of the vertical linear axis. This can reduce weight, and also simplify the kinematics of the setup. One solution for making the robot function like a vertical linear axis, is to have a system like a delta robot for the vertical portion. However, this would probably result in too short travel distance for the vertical portion. A stroke of 2500mm or more would be suitable, making the robot reach all the way to the floor in most factories, which is necessary to clean the equipment from the underside. A better solution to achieve such a long reach is to built a foldable arm, as illustrated in Figure 3.12 and 3.13.

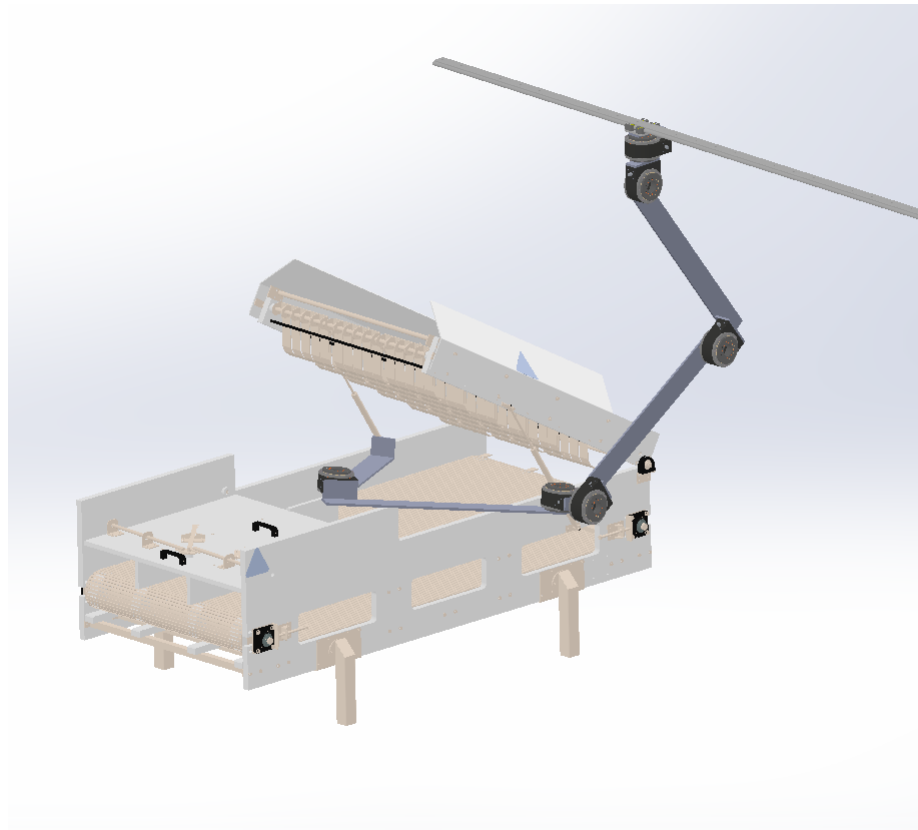


Figure 3.12: Robot made from igus robot joints. CAD model of joint available at [RoboLink \(2016\)](#).





Figure 3.13: A long reach is possible with a modular robot.

As discussed, the igus robot joints are made of plastic only, and an upgrade of the worm and the worm gear would be necessary to increase the torque of the joint. Luckily, igus supplies CAD drawings of their parts, although the drawings does not show how the internals work. However, this is a small problem for a mechanical engineer to design. One drawback of substituting the worm and worm gear with steel or similar is that the worm gear will now require lubrication.

The biggest uncertainty regarding the modular robot is the force from the nozzle acting on the end-effector. This will of course be affected by what nozzle will be used. The pressure in the water system can be upwards of 25 bar, and if a nozzle with a hole diameter of 5mm is chosen, this can give a maximum force of:

$$Force = 2.5MPa * \frac{0.005m^2 * \pi}{4} = 49N \quad (3.2)$$

This is the worst case scenario, and is the equivalent of a 5kg payload. Assuming the worm and worm gear of the igus robot joint is made of some type of common plastic like PVC, ABS or Nylon, a steel counterpart could be upwards of 100 times stronger (1-5MPa vs 210MPa elastic modulus ([Elasticity, 2016](#))), giving a probable torque rating for the upgraded robot joint of over 1000Nm.

Another possibility is to buy slewing drives made from steel. These slewing drives are most commonly used for industrial equipment where machinery or tools are placed upon the slewing drive, such as excavators, cranes and CNC milling machines. Therefore, the slewing drives are not symmetrical, and only meant for loads on one side. This poses a disadvantage for such a solution to build a robot arm, as it requires an additional slewing ring to be installed to take the axial forces in the opposite direction of what the slewing drive is made for.

A better solution would be to build the robot from both steel slewing drives and plastic slewing drives. The joints that need to deliver the most torque around the joint axis can then be of steel, while the joints that does not need to deliver such high torque can be made of plastic. Another aspect of the slewing drive is how much axial torque it can withstand, and how high loads in axial and radial directions it can withstand. The axial torque and radial force is not listed on igus' web page ([Robot-joint, 2016](#)) or on a preliminary data sheet found on their web page. The only thing listed other than the radial torque is the axial load, which is over 1200N for the biggest joint. It is however safe to assume that the axial torque can be way higher than the radial torque. For comparison: the igus PRT of the same size can handle a torque of 800Nm, datasheet available at [PRT \(2016\)](#). This would be sufficient for a reach of:

$$Reach = \frac{800Nm}{49N} = 16.3m \quad (3.3)$$

This is when ignoring the moment caused by the weight of the individual joints and links. A safe assumption would be at least 10 meters. Using aluminium profiles, or even carbon fiber, for the links, and plastic joints wherever possible will keep the weight of the whole construction to a minimum, resulting in longer possible reach.

The three joints that will have to deliver the most radial torque are joint 2,3 and 4 (from top to bottom), and here it would be wise to use steel slewing drives. A company called Cone Drive produce slewing drives in small sizes. The smallest drive they produce weighs 12kg and can deliver a radial torque of 400Nm (SE3C), datasheet can be found on [SlewingDrive \(2016\)](#). Three of these joints will then weigh 36kg, and probably upwards of 50kg with servomotors, additional slewing ring and links. But the good thing about this weight is that it will have a center of mass which is not too far from the robot base when seen from above, resulting in relatively low torque on the guide. Simplifying by saying that all the mass is uniformly distributed between joint 2, 3 and 4, and assuming this part of the robot will just be used for vertical movement, it can be calculated that the max torque this assembly with 1 meter links will introduce to the guide is:

$$Torque = 50kg \cdot \frac{2}{3} \cdot 9.81m/s^2 \cdot 0.5m = 153.5Nm \quad (3.4)$$

when standing in the position shown in Figure 3.14. Doubling the reach of the vertical portion of the robot will then double the torque on the guide.

Using the assumptions just made, it is possible to make an estimate of the forces and torque the guide will need to withstand. Most of the curved guides are not specifically made to handle a lot of torque, they are made to just have something standing or hanging from them. Strengthening of the guide will be discussed in Section 3.7.2.

To get better control of the spray of the nozzle it would be beneficial to have at least a 1DOF joint on the tip of the robot, preferably a 2DOF joint, something similar to a pan tilt bracket used for cameras. Depending on the nozzle it could also help with a third degree of rotation around the axis of the spray. If the nozzle is a flat spray nozzle, the rotation of the nozzle has to be adjusted according to the direction the robot travels. A simple solution is to use a nozzle which has a cone shaped spray pattern, as the liquid distribution does not have to be uniform. See Figure 3.15 for illustration.

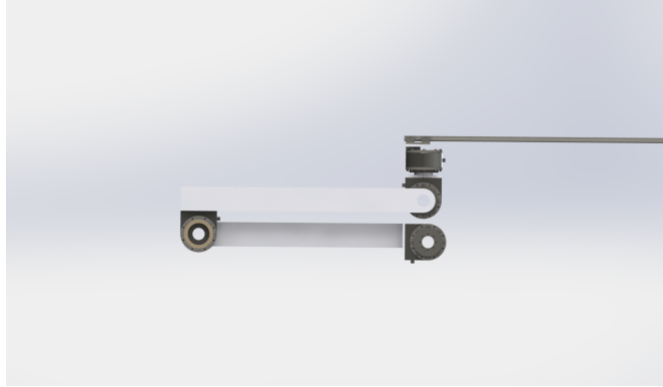


Figure 3.14: The position of the vertical part of the robot that will result in the most torque on the horizontal guide.

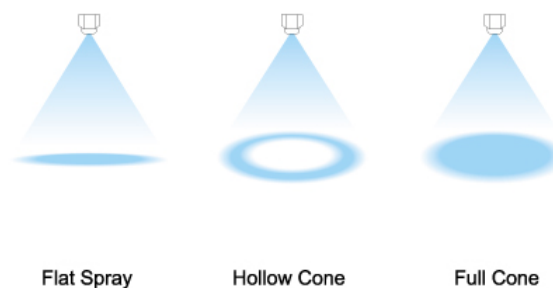


Figure 3.15: Different types of spray pattern from a nozzle (Nozzle, 2016).

### 3.7.2 Curved guide

Rollon makes curved guides, but without propulsion, which can be seen in Figure 3.16. A possible solution is to use their guides, and combine them with a chain welded to the side of the guide for propulsion. The slider on the guide will then have to be fitted with an electric motor and a sprocket to be able to move around. This is however not a very industrial solution. The welding of the chain will be a lot of work and it can break over time. From the data sheet of the Rollon curved guide called Curviline it is obvious that these guides are not meant to handle any torque. One possible solution to make the guide able to withstand torque is to have some sort of locking mechanism that immobilizes and stiffens up the carriage. A simple mechanism that does this could be a hydraulically operated clamp, similar to a robot gripper. It is not the rail itself that can not withstand the force, it is the rolling mechanism of the carriage, making a clamping mechanism a valid solution for increasing the torque possible to apply to the carriage. As long as the rail is then secured firmly enough underneath the roof, the rigidity of the suspension and robot is catered to.

Another solution is to make the curved guide using two tubes as guides, and then a slider with bearings mated to the tubes, much like with the EPX linear axis, with the exception that it allows curvature of the pipes by having a swivel between the link from the bearing on one pipe to the other, and the carriage, same as the Curviline which can be seen in Figure 3.16. The propulsion would then again have to be accomplished by a chain or similar.

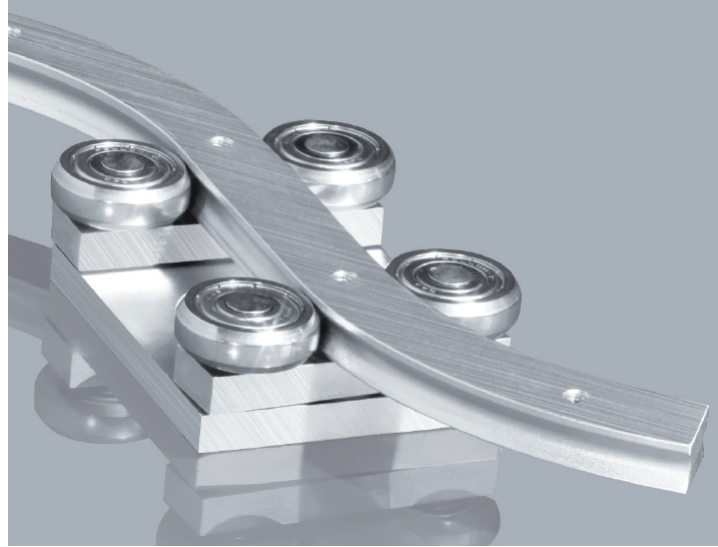


Figure 3.16: Rollon Curviline, picture taken from datasheet, available at [Curviline \(2016\)](#).

### 3.7.3 Control system

A modular robot made from scratch would need a control system. When buying a complete robot from a supplier they always come delivered with a control system which has the kinematics and trajectory planning built in. One solution is to buy a PLC-based motion controller from OMRON. The motion controllers from OMRON can handle up to 256 axes with a cycle time of  $125\mu s$ . They support position control, velocity control and torque control, with the possibility of multi-axes coordinated control with linear interpolation. See [OmronPLC \(2016\)](#) for more details. Advantages with this solution is that it is a industrial solution which has been tested and proven. However, these PLCs are only intended for programming relatively small programs, possibly making them very incomprehensible when a lot of robot movements need to be programmed. It is not unlikely to exceed hundreds of robot commands for the complete cleaning process.

Another possibility is to build a servo controller and use a computer running ROS as a motion controller. ROS has features for motion control, kinematics and trajectory planning. Servomotors have built in encoders, thus having simple interfaces. With this solution it will be easy to implement computer vision, and changing the trajectories according to the 3D scans. For the interface between the PC and the servo controller a microcontroller would be necessary. This microcontroller would then just work as the hardware interface of the PC.

The simplest solution would be to use a microcontroller to do all the heavy lifting. A micro controller can do all the calculations required for the inverse kinematics and the trajectory generation, and have a simple interface with TCP/IP for controlling it using a computer. The PC can then just have a simple stand-alone program for programming the paths of the robot, and sending them to a queue on the microcontroller. This solution would give very low latency at a low cost. The drawback of such a solution is that it's not very industrial. Micro controllers are generally not delivered with any form of water resistance, making an upgrade of their IP grade necessary for industrial use. The advantage of this solution is the programming of

the robot paths. With a custom stand-alone program to program the robot paths in, a big emphasize can be placed in making the programming interface as user friendly as possible for a big number of robot actions.

### 3.8 First prototype

At the end of writing this thesis it has been decided that a prototype to start testing the cleaning will be built at Stranda from mid June, using a UR10 as robot. This is due to NTNU in Ålesund has a UR10 which will be free to use the whole summer. The prototype will be based on solution 3, to keep the design fairly simple, while also testing the viability of solution 3 and 4. The key components for the prototype that needs to be ordered:

- The robot: UR10, available at NTNU in Ålesund.
- Vertical linear axis: RK EPX 60, already ordered.
- Slewing ring with servo.
- Nozzle.
- Control system.
- Folding system: Optional.
- Stabilizing leg: Optional.

To perform the testing, a structure standing on the floor with a beam spanning over it will act as the mechanical suspension base. An electric stunner will be used for test cleaning.

In addition, brackets for the servo motors, wiring, etc. must be made/ordered.

#### 3.8.1 Robot

As is obvious from Table 3.1, the UR10 robot is a very good alternative for a cleaning robot. The only upgrade it will need is a protective cover to shield it from water spray. To extend the reach, an extender on the robot end effector before the nozzle may be viable. Since a UR10 is readily available at NTNU in Ålesund, the choice was simple.

#### 3.8.2 Vertical linear axis

Regardless of what solution we would ultimately choose, it was obvious we needed a vertical linear axis capable of holding and moving a robot. Thus, it was decided to order a vertical linear axis before the final design was established. The choice fell on the stainless steel version of the EPX 60 from RK Rose+Krieger.

The servo for the linear axis can be the same as in solution 3. A servo controller from OMRON is also necessary.

#### 3.8.3 Slewing ring

Same as for solution 3, this prototype will base its slewing ring on the one chosen in Section 3.6.3, the igus PRT. The loads in the prototype will be almost the same as in solution 3, the only difference being the weight from the robot (UR10 vs KR10). The choice landed on the

PRT-01-150, with a outer ring with spur gear. This outer ring will have 126 teeth according to the datasheet.

Since there are no direct forces acting on the servo rotating the slewing ring while the vertical linear axis is not folded up, the torque requirement for the servo will be dependant on the wanted angular acceleration of the assembly. Assuming a center of mass at the middle of the reach of the robot when fully extended, and neglecting the moment of inertia for the linear axis, the moment of inertia for the vertical assembly becomes:

$$I = 29kg \cdot 0.65^2m^2 = 12.25kgm^2 \quad (3.5)$$

Choosing a servo with a torque rating of 1Nm and a cogwheel with 20 teeth will give an angular acceleration of:

$$\alpha = \frac{1Nm}{12.25kgm^2} \cdot \frac{126}{20} = 0.51 \frac{rad}{s^2} \quad (3.6)$$

which should be sufficient. To keep it simple, the same servo as recommended for the linear axis can be chosen, the OMRON R88M-K75030F. The only thing that has to be manufactured is a bracket for mounting the servo, and the cogwheel.

#### 3.8.4 Nozzle

A nozzle designed for cleaning will also have to be ordered. It is unknown at this point in time whether a flat cone nozzle or full cone nozzle is the best for cleaning, so this will have to be tested with the first prototype. HL Skjong is a partner of Seaside, and will take the responsibility of ordering correct nozzles for the application.

#### 3.8.5 Fold

Neglecting the folding function of the system will simplify things quite a bit. However, if the final solution chosen resembles solution 3/4, the testing of the stability of the folding with the accompanying stabilizing leg will have to be done at some point. On the other hand, if the cleaning system does not work at all, having tested the folding system and the stabilizing leg would have been pointless.

#### 3.8.6 Control system

A control system is needed for controlling the robot, the servo for the linear axis, and the servo for the slewing ring. Choosing the servos from OMRON, an additional servo controller is necessary. The servo controllers from OMRON can either be used with EtherCAT interface, or analog inputs, depending on which model is chosen.

One alternative is to use a servo controller with EtherCAT interface. Then an accompanying PLC with EtherCAT interface is necessary. This PLC can then be used to control both the servos and the robot. The robot paths will then have to be programmed on the robot controller, and the PLC can then only start and stop robot programs. Another alternative is to use a microcontroller and the analog servo controller. The microcontroller can then either control both the servos and the robot, or just control the servos while a computer controls the robot. This will give easier control of the robot.

### 3.8.7 Final design of prototype

The final design of the prototype can be seen in Figure 3.17 and 3.18. Part list for the prototype:

- 1 x UR10
- 1 x EPX 60
- 1 x PRT-150-ST
- 2 x OMRON R88M-K75030F
- 2 x Servo controller

In addition to what can be seen in the figure, the prototype will also need wiring and a control system.

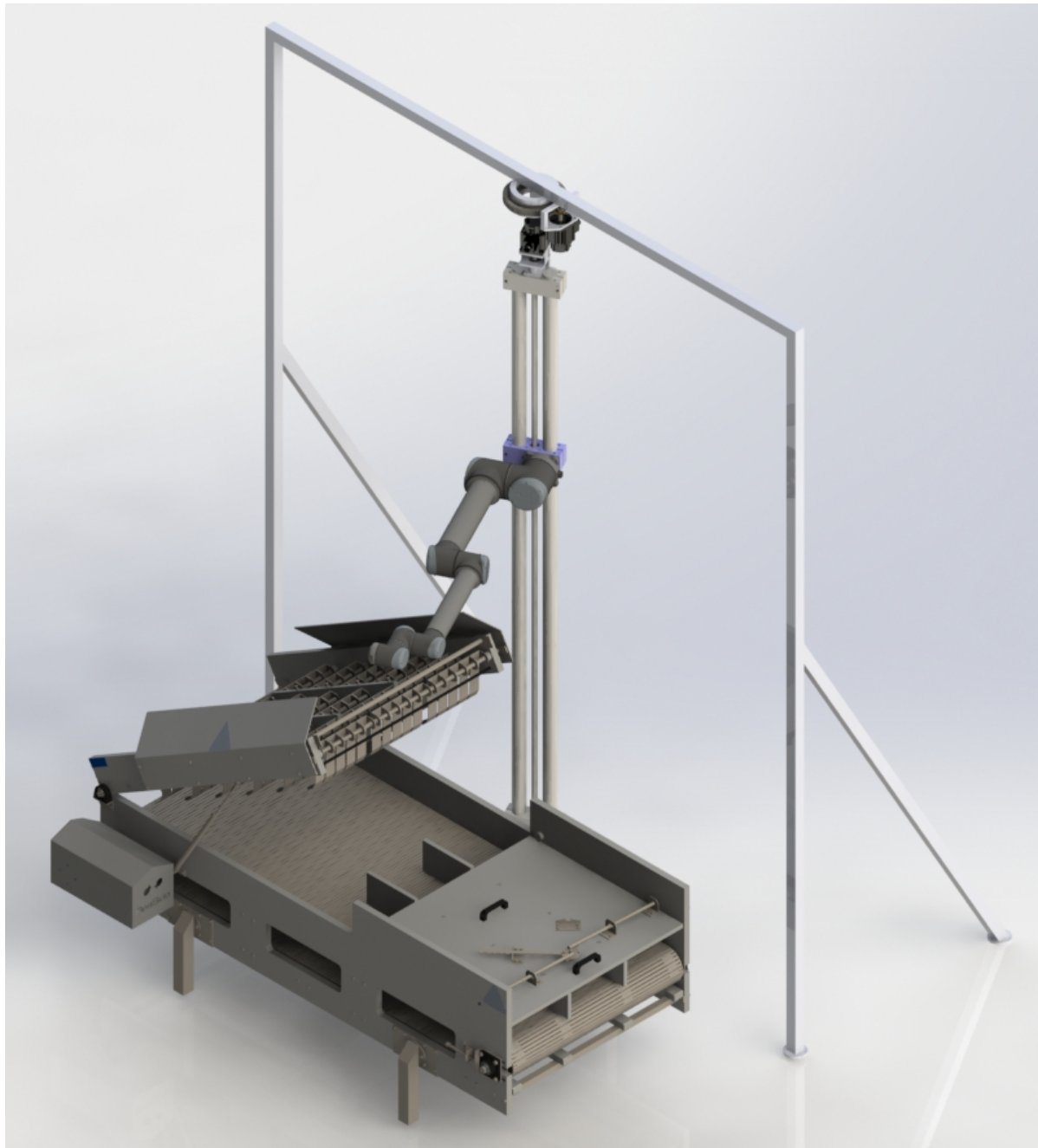


Figure 3.17: Rendering of first prototype. CAD models of UR10, EPX 60, PRT and servos from GrabCAD (2016), RoseKriegerEPX (2016), PRT (2016) and ServoCAD (2016), respectively.



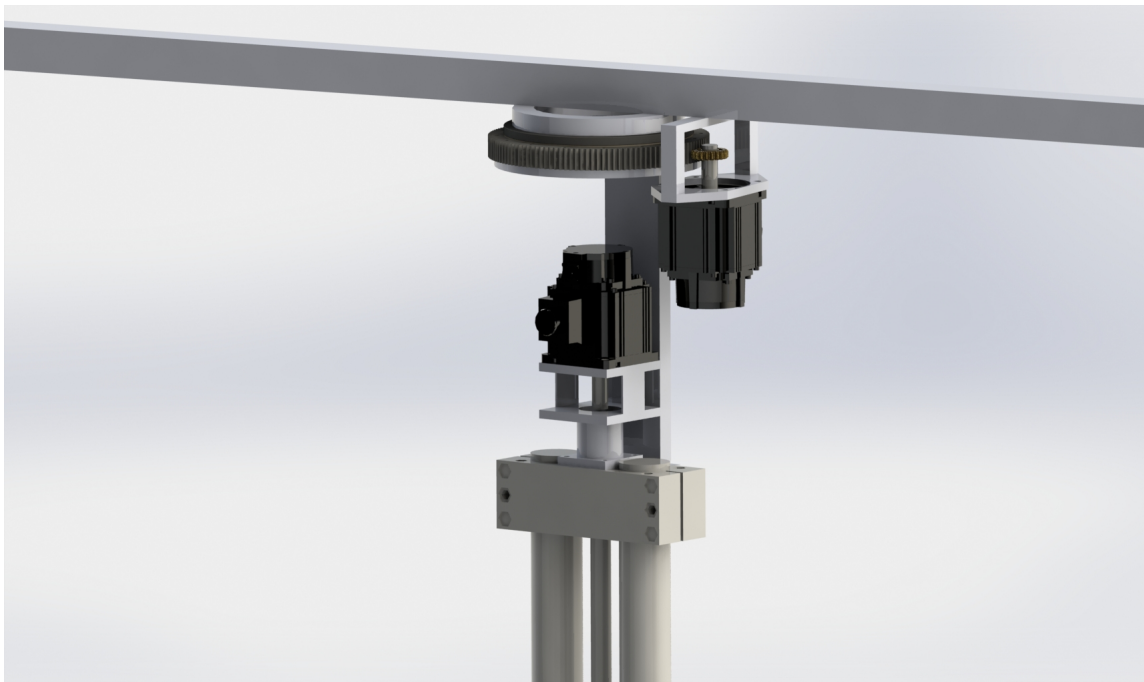


Figure 3.18: The top assembly of the first prototype.

### 3.9 Discussion

As we have just seen, there is no one single solution that is ideal to this problem. All solutions have their advantages and disadvantages. It seems like the design proposals have come full circle, i.e. solution 5 is in some way similar to solution 1. Of course, one of the big advantages with solution 5 is the custom robot, but the suspension which was the culprit for solution 1, is very similar to the one in solution 5. This might emphasize that the ideal solution is not the most obvious one, nor the one that instinctively feels the most promising. There will probably have to be a some trial and error with the prototype before a final solution is established, no matter how much work is put into design.

It is apparent that plastic components provide benefits when working with a clean but still corrosive environment. The lubricating properties while still being corrosion resistant solves many of the challenges present. However, the range of suppliers that makes plastic based linear axes are limited, resulting in fewer components to choose from. As seen in Section 3.6.2 this may result in a need for specialized components, driving up the design and manufacturing time, and cost.

# Chapter 4: Simulation

## 4.1 Introduction

A lot of work has gone into the design of a robotic cleaning solution for salmon slaughterhouses. During the writing of this thesis several solutions has been presented and designed. It has therefore not been suitable to do simulations of all the solutions, especially the ones that were presented and designed at the end of the writing of this thesis (solution 5 and first prototype). Instead, the solutions that looked the most promising at the time were simulated, this being solutions 2 and 4. The simulations were carried out in Visual Components.

The purpose of the simulation is to gain insight in the validity of the designed solution. The following aspects which are important for a cleaning robot can be examined with simulation:

- Required robot reach.
- Global reach/accessibility with a given transportation system.
- Local reach/accessibility with a given robot.
- Alternatives/requirements for a vertical transportation system (vertical linear axis solution 3 and 4).

Many of these aspects can be almost fully examined with an assembly in SolidWorks. However, a simulation can complete the whole picture of the cleaning process, making it easier to communicate with others working on the project, subcontractors, etc. exactly how the robot should move and clean.

## 4.2 Programming movable robot base

The first thing to do when programming joints in Visual Components (VC) is to either import a CAD model of the joint, or make the model in VC itself. A CAD model created in another software will contain different geometries, depending on the complexity of the drawn part. The different geometries should then be put in Links (also called Nodes in VC), see Figure 4.1. The base Link will have the base geometry and will have no joint, as this is the base and it will stand still. The next Link will then have another geometry, and this will be set up to either revolve around a given axis or slide along a given axis. As can be seen in Figure 4.2, the node (Link) Svivel has a joint that will rotate around the z-axis. The first thing to do before starting to make all the Links is to make a ServoController in the base Node. This ServoController is then set up with all the joints that will ultimately represent the whole manipulator. As can be seen in Figure 4.3, each joint gets a name, whether it is revolute or prismatic (linear in VC), and the lower and upper limit of the joint value. These joints are then fed into the joint fields of the links, as seen in Figure 4.2b. For each joint there will also have to be a Jog Info under the Behavior tab for each Link. This Jog Info will export the joint variable, making it possible to manipulate the variable from scripting.

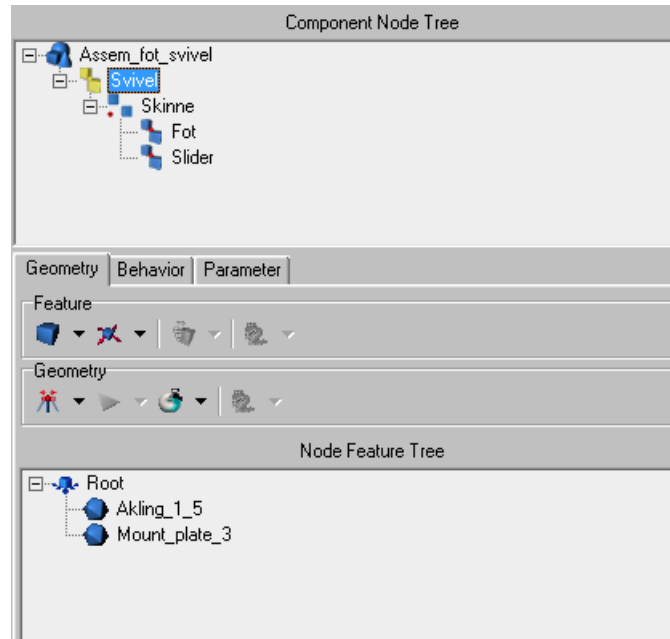
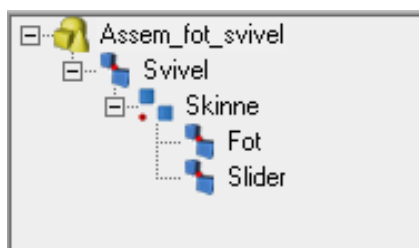
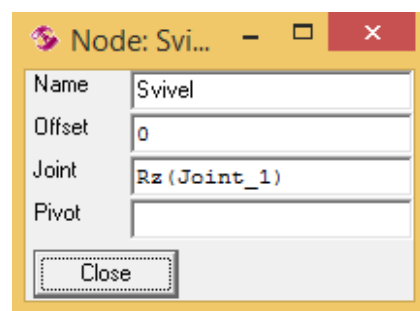


Figure 4.1: Under Node Feature Tree the different geometries in the Node (Link) Svivel are displayed.

The last thing that is needed when making a movable robot base is the exportation of the base and its links. This is done by making a One to One Interface in the Node the robot will connect to. Before setting up this interface, a frame where the robot should connect is needed. This interface is then set up with a hierarchy where the frame is the parent. In addition there will have to be a Joint Export field to export the joints in the ServoController. This will enable the robot to manipulate the joints in the movable robot base. An example can be seen in Figure 4.4



(a) Overview of the Links.



(b) How to setup a joint.

Figure 4.2: Two of the most important aspects of programming custom functionality in VC: Links and joints.

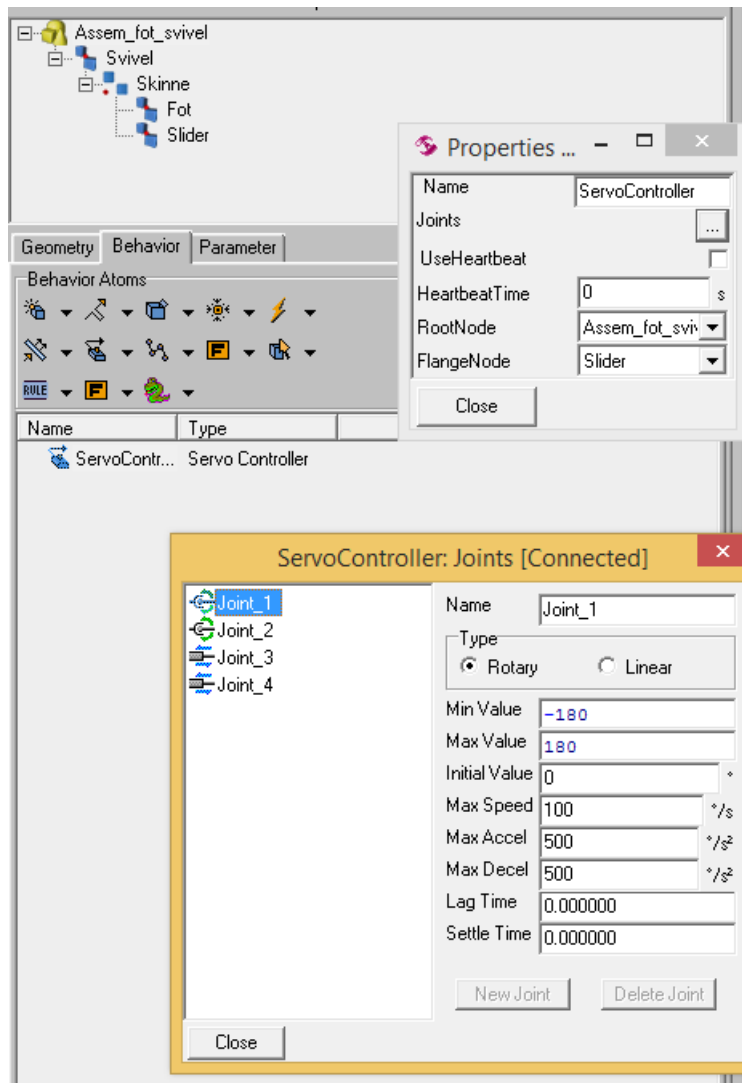


Figure 4.3: The setup of the ServoController.

### 4.3 Programming the robot paths

The programming of the robot paths were done utilizing Python scripting. The script for solution 4 can be seen in Appendix A. This results in multiple statements the robot will go to, as seen in Figure 4.5.

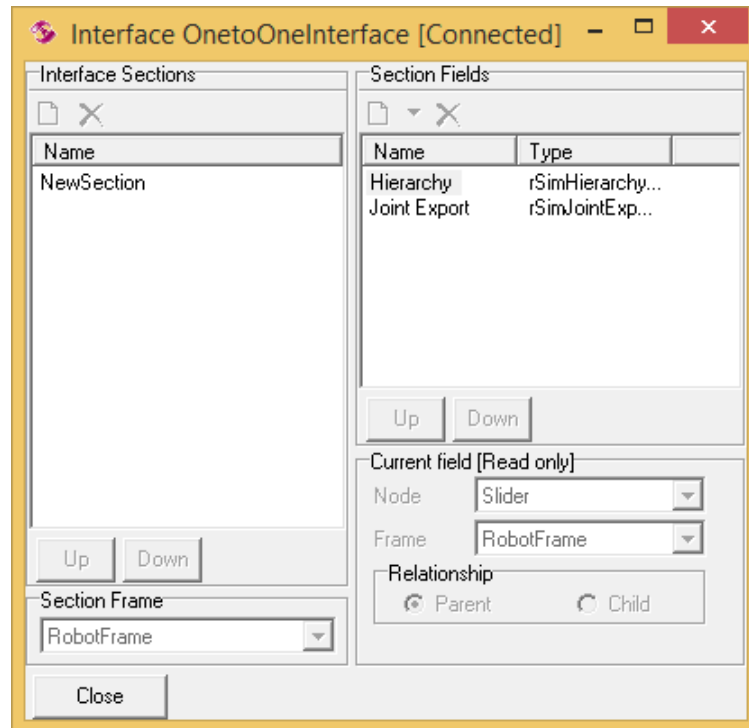


Figure 4.4: How the One to One Interface should be set up when making a movable robot base.

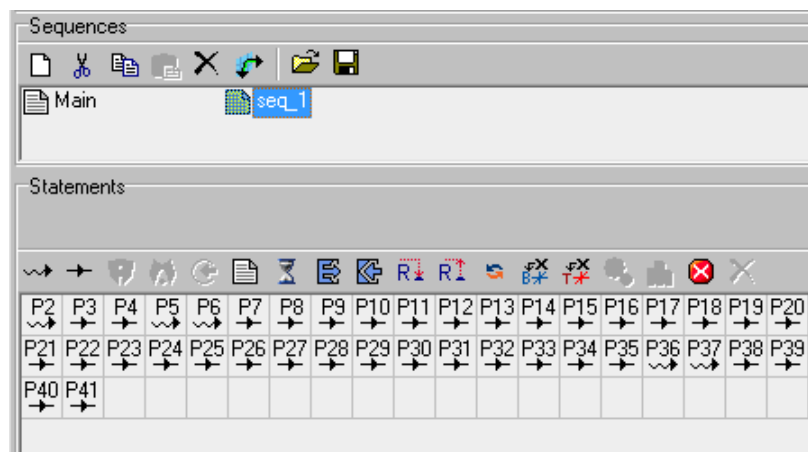


Figure 4.5: The statements made by the Python script.

#### 4.4 Point cloud import

One aspect which would enhance the importance of a simulation is the import of a scan from a factory where such a cleaning solution can be implemented. This scan will then have all the dimensions and obstacles that will be in that given factory, making it possible to predict the difficulties of installing the cleaning solution. It was considered to do a scan of Fjordlaks, but this was not completed. However, a test scan of the electronic lab at NTNU Ålesund was completed for testing purposes.

Importing a scan into VC is a trivial task of deleting the header in a PCD file and changing the file extension to xyz. Figure 4.6 shows the scan imported into VC. The scanner is a Topcon GLS-2000. Unfortunately, the software for the scanner only exports the scan in the file extension .pcd without color.

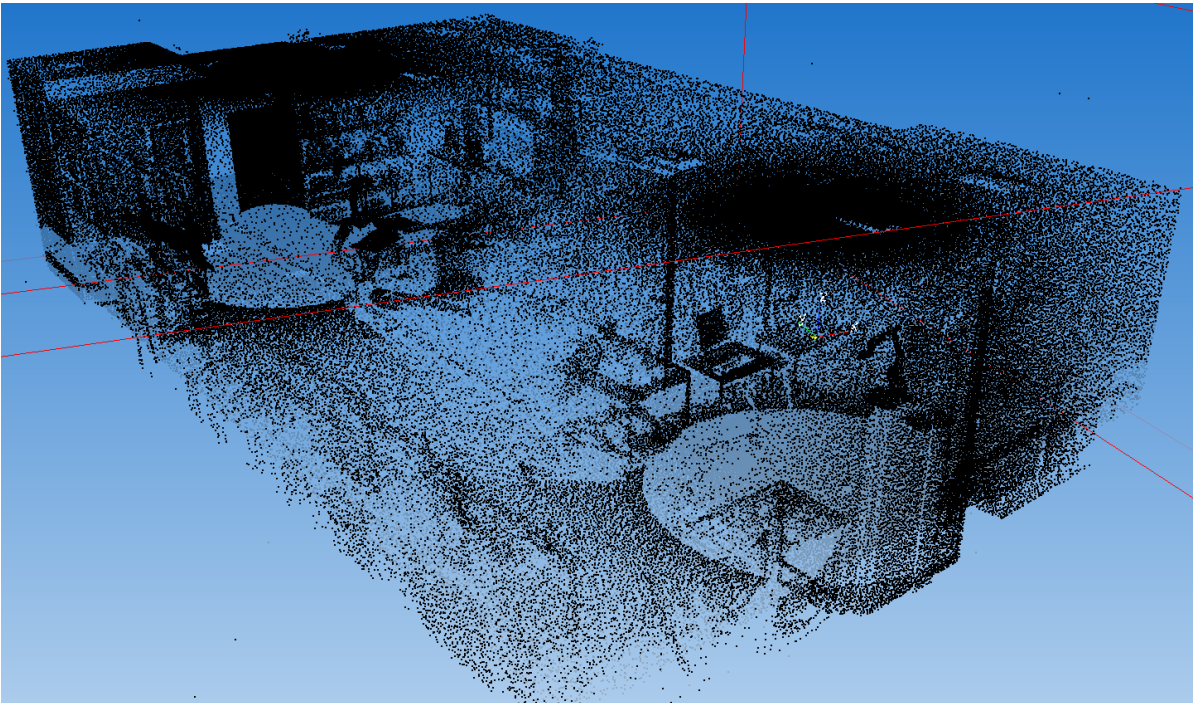
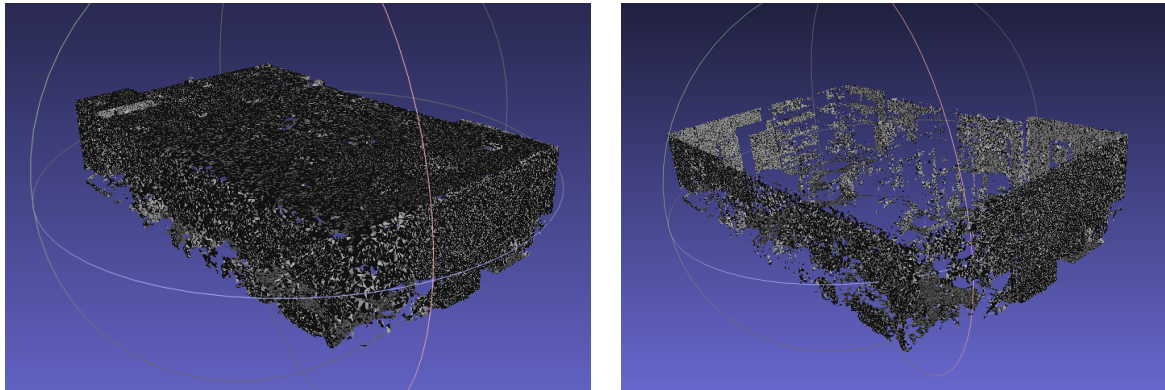


Figure 4.6: A scan of the electronics lab at NTNU Ålesund imported in VC.

Further work with the point cloud was to make a mesh of it, thus enabling it to be imported in non-Premium SolidWorks as a background. In PCL there is a feature to make a mesh of a point cloud. The method is called GreedyProjectionTriangulation. This method requires a point cloud with its associated normals as input. In addition, the maximum of nearest neighbours, maximum and minimum angle of the triangles and some other parameters need to be set, see Section 2.3.14. The parameters suggested from the PCL website was used as a baseline, and with some small adjustments the results proved adequate. The result from the mesh can be seen in Figure 4.7a. Using a passthrough filter it is possible to remove the roof of the room, see Figure 4.7b.



(a) Mesh made from the scan.

(b) The roof of the room removed from the mesh.

Figure 4.7: Meshes made from the scan of the room at NTNU Ålesund.

## 4.5 Results

The simulation of solution 2 can be seen in Appendix D. The simulation shows the reach of the KR10 on a suspension almost like an overhead crane. As can be seen, the positioning possibilities of the robot is excellent. No need for a long arm on the robot end effector to reach the whole component that should be cleaned. Expanding the robot to cleaning other components and machines within the reach of the mechanical suspension becomes a trivial task when the robot base can be positioned precisely.

The simulation of solution 3/4 can be seen in Appendix D. Unlike the simulation of solution 2, the robot needs an arm mounted on the end effector to reach the whole component. However, with the arm attached to the end effector, the reach of the robot is acceptable as it can reach two electric stunners if they are placed side by side. The big drawback is that expanding the reach of the cleaning solution is not possible. That is unfortunate as one of the main features making the solution viable in the long run is if it can clean the whole factory.

## 4.6 Discussion

The simulations of the solutions shows the possibilities with them. The reach and accessibility is shown, even though a complete cleaning process is not simulated. A complete simulation of the whole cleaning process could have been implemented, but the advantages are minimal. The paths created in VC can be exported to the real world robot, but since no data on the width of the nozzle spray, the appropriate velocity of end-effector, etc. exists, the paths in simulation are probably far from ideal, and must be established by manual programming and trail and error.

The mesh is good enough to help in a design process. It has accurate measurement of distances, obstacles etc. making the planning of a factory layout easier. However, there are many programs that can create meshes from point clouds. Some of these programs have many features that recognises features such as floors, walls, tables etc. making it easier to create more realistic meshes. Premium versions of SolidWorks and Siemens NX does support import of raw point clouds, making it unnecessary to first make a mesh of the point cloud.



A scan of Fjordlaks where the first prototype will be installed would have helped in designing and simulating the solutions. For instance, the whole curved guide underneath the roof could have been designed, and then made from the computer model without taking any measurements in the factory.



# Chapter 5: Computer vision

## 5.1 Introduction

Before the cleaning of the slaughter house should start, it would be desirable to check that everything is as it should be, i.e. no unwanted parts (such as a shovel) are left inside the slaughter house. One way to do this is to use a 3D camera such as a Kinect to map the slaughter house. If any differences from a reference point cloud of the slaughter house is found, actions should have to be taken. It is also possible that some of the equipment has moved since the slaughter house was scanned. It would then be beneficial to detect how much it has moved so that the robot trajectories can be adjusted accordingly, making the robot able to clean regardless.

The idea is to mount a camera on the robot before each cleaning and then move through the area which it is supposed to clean, scanning this area. After the scanning is completed, and no obstacles are found inside the slaughter house, and the corrected equipment positions are calculated, the robot can then put away the camera and start the cleaning.

### Two approaches to the object detection

As just mentioned, being able to detect objects and their position and orientation can be beneficial in a robotic cleaning solution. There are several ways this can be solved for a slaughter house. It could be possible to assume that the equipment will not move very far (in centimeter range), making it possible to just use ICP correcting the position and orientation of the equipment. There would however be a requirement to roughly position the CAD models of all the equipment after the first scan was complete, as the ICP algorithm requires a rough alignment before it starts. Then after the manual positioning, and the first run of ICP, all the default locations for the equipment is established. These positions can then be used as base when the new positions for the equipment are calculated.

A better solution to the object detection would be to use feature detection first for the rough initial alignment. Feature detection does not require the parts to overlap to work. Feature detection calculates features in the point clouds, and tries to see if some of the features are in both point clouds. After the initial alignment, the same procedure with ICP can be done. With this approach, the assumption that the equipment is not moved more than a couple of centimeter does no longer apply.

## 5.2 Object detection experiment

In order to test the possibilities with computer vision, a simple test procedure was set up. This test procedure was as simple as finding a box on a table using a Kinect v2. PCL was utilized for the testing. When placing a rectangular box on a table it can either have the back facing the front, or the front facing the front. To eliminate this problem, a hole was cut on one of



(a) The box used for testing.



(b) The Kinect and its view over the table where the testing was done.

Figure 5.1: Experiment setup.

the sides of the box, seen in Figure 5.1a, making the front and back of the box explicitly given. The Kinect was mounted in the robot cell as can be seen in Figure 5.1b. A CAD model which matched the real world box was made.

The first step before doing anything with the point cloud from the Kinect was to make a database of how the CAD model of the box would look like from different viewpoints. This is called ray tracing, explained in Section 2.3.7. A method for this was already developed by fellow PhD student Adam Leon Kleppe. VFH descriptors for the point cloud from different viewpoints were then calculated.

The point cloud of the scene was captured using a Kinect. The first step after acquiring the point cloud was to remove everything but the table. This was done using a passthrough filter. Then the point cloud was downsampled and the table was segmented out. The remaining points were then grouped in clusters. Same as with the CAD point clouds, VFH descriptors were calculated for the clusters. An algorithm then checked which CAD cluster matched best the cluster from the Kinect, and an initial alignment could be done. Next up was using the ICP algorithm for getting the final transformation. This is a modified version of the pipeline in Section 2.3.8, shown in Figure 5.2.

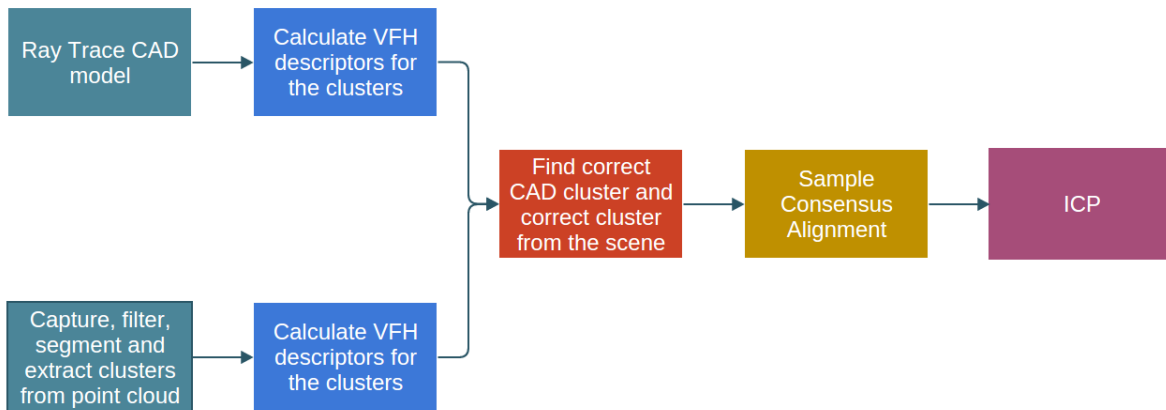


Figure 5.2: The modified pipeline.

The initial alignment was done using Sample Consensus Initial Alignment. First SIFT was used to generate the keypoints in both the CAD cluster and the scene cluster which matched best according to the VFH descriptor. Then FPFH descriptors are calculated for those keypoints. The keypoints and their respective descriptors are then used by the Sample Consensus Initial Alignment algorithm to find the best match, and its respective transformation. ICP then does the final transformation.

All the programming was done using C++ and Qt. Fellow student Sindre Raknes helped with setting up *slots* and *signals*.

### 5.3 Object detection results

Figure 5.3 and 5.4 represents the results from two tests which were done on exactly the same data (same ray traced models, same captured scene). The red dots represents the feature matching transformation of the ray traced CAD model, while the blue represents the ray traced CAD model which has been transformed with ICP in addition to the feature matching transformation. As can be seen in the figures, the feature matching does not deliver the same result even if the input is the same.

Notice how the box is not a complete box, this is due to the ray tracing. This is how the box looks like from the viewpoint that best matches the viewpoint of the box seen from the Kinect.

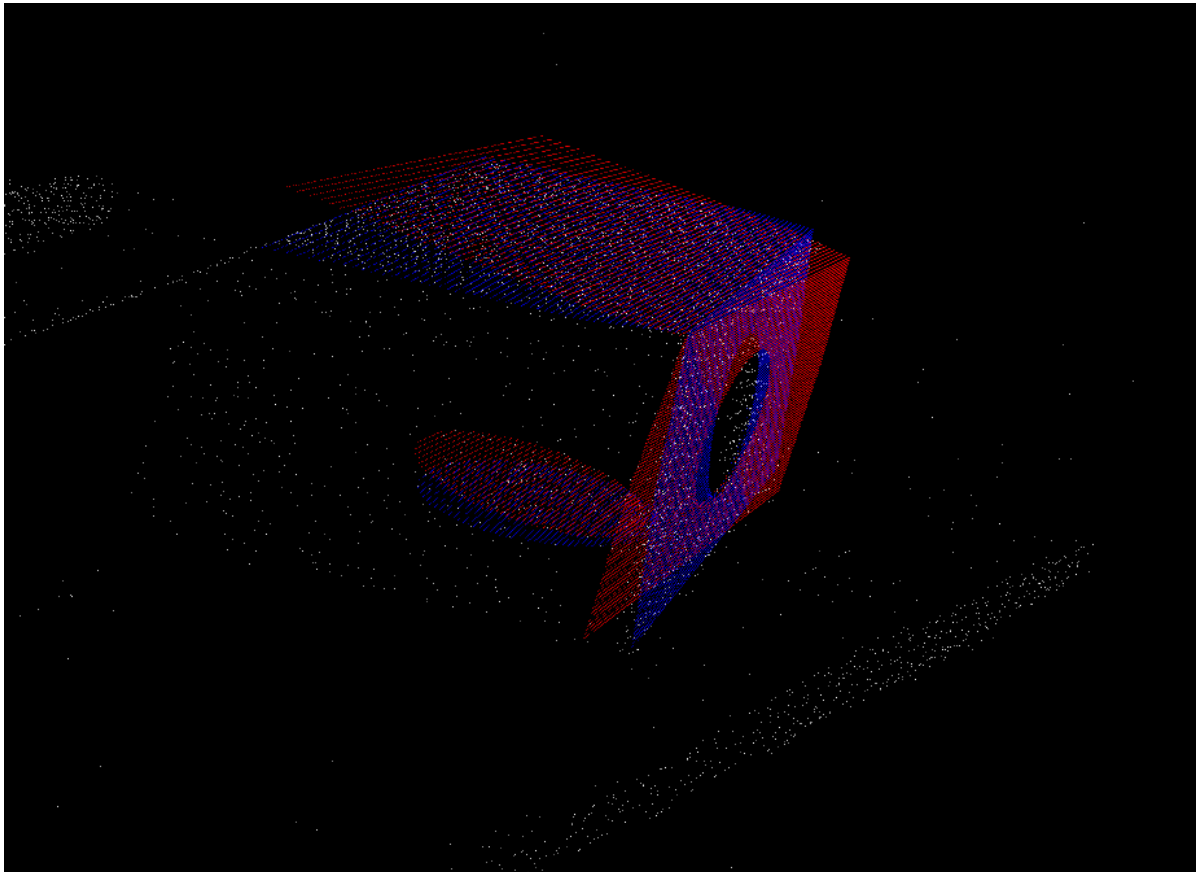


Figure 5.3: A fairly good feature matching transformation shown in red. ICP shown in blue.

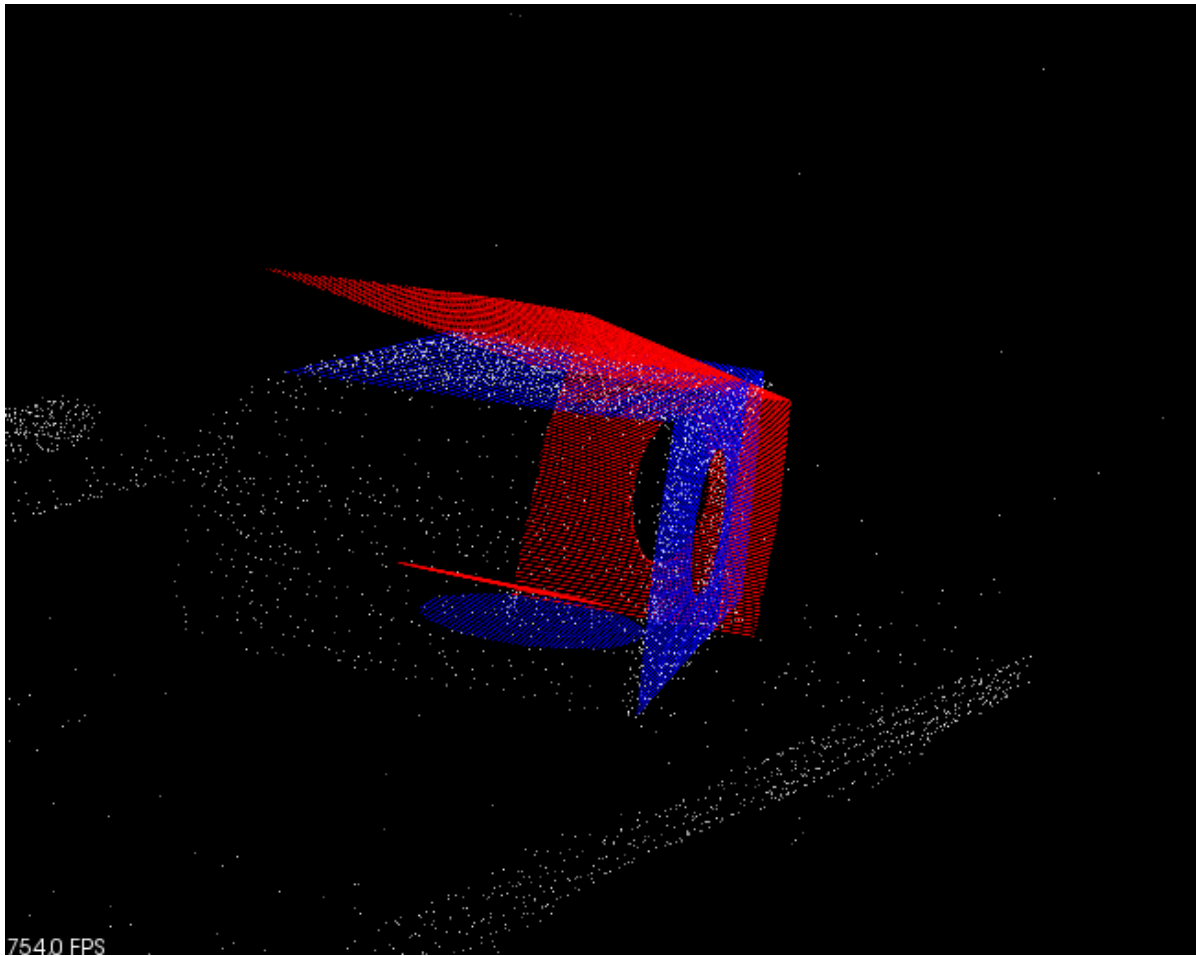


Figure 5.4: A less good feature matching transformation shown in red. ICP shown in blue.

The different transformation matrices can be seen below. Even though the feature matching transformation differs quite a bit, the final transformation after the ICP is almost the same.

$$SACIA_1 = \begin{bmatrix} 0.0455127 & -0.976051 & -0.212727 & 0.236714 \\ 0.929239 & 0.119526 & -0.349611 & 0.00252712 \\ 0.366665 & -0.181763 & 0.912425 & 0.679168 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (5.1)$$

$$ICP_1 = \begin{bmatrix} 0.992716 & 0.11998 & -0.0112267 & 0.0402233 \\ -0.117827 & 0.985973 & 0.118239 & -0.152422 \\ 0.025256 & -0.116054 & 0.992923 & -0.0156832 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (5.2)$$

$$Total_1 = \begin{bmatrix} 0.152555 & -0.95256 & -0.263368 & 0.267891 \\ 0.954197 & 0.211364 & -0.211758 & -0.0975172 \\ 0.257377 & -0.218999 & 0.941169 & 0.664364 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (5.3)$$

$$SACIA_2 = \begin{bmatrix} -0.211146 & -0.967466 & -0.139382 & 0.196236 \\ 0.959156 & -0.177613 & -0.220168 & -0.121475 \\ 0.188249 & -0.180176 & 0.965453 & 0.654992 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (5.4)$$

$$ICP_2 = \begin{bmatrix} 0.926095 & 0.373453 & -0.0538526 & 0.166735 \\ -0.376393 & 0.924344 & -0.0626539 & 0.129486 \\ 0.0263823 & 0.0782926 & 0.996586 & 0.0158975 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (5.5)$$

$$Total_2 = \begin{bmatrix} 0.152521 & -0.952592 & -0.263295 & 0.26783 \\ 0.954269 & 0.211261 & -0.211538 & -0.097698 \\ 0.25713 & -0.218991 & 0.941242 & 0.66432 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (5.6)$$

## 5.4 Segmentation experiment

As mentioned in the introduction of this chapter, it would be convenient to find out if something has been misplaced or left behind inside the slaughter house before cleaning. Tests were carried out using the same experimental setup as with object detection. The tests were now done using the SegmentDifferences algorithm in PCL. As explained in Section 2.3.12, this algorithm takes two point clouds and a minimum distance criteria, and finds the differences between the point clouds. Since noise in the point clouds will always be different from point cloud to point cloud (even when the point clouds are two consecutive ones in a 3D stream), a StatisticalOutlierRemoval was run before the SegmentDifferences algorithm. All the differences in the point clouds are then put in clusters.



## 5.5 Segmentation results

In Figure 5.7 the result of the SegmentDifferences algorithm can be seen. The only step done before SegmentDifferences is a StatisticalOutlierRemoval filter. The reference and target cloud can be seen in Figure 5.5 and 5.6, respectively. From the result it is clear that the algorithm only keeps the points that are different, in only one cloud. If not, the part of the table that is missing due to the box casting a "shadow" would also be included in the result. To eliminate noise that was not removed by the StatisticalOutlierRemoval filter, only clusters with a size of over 500 points are shown. In this case that is only the box. This value could be lower is was likely to find object smaller than the box.

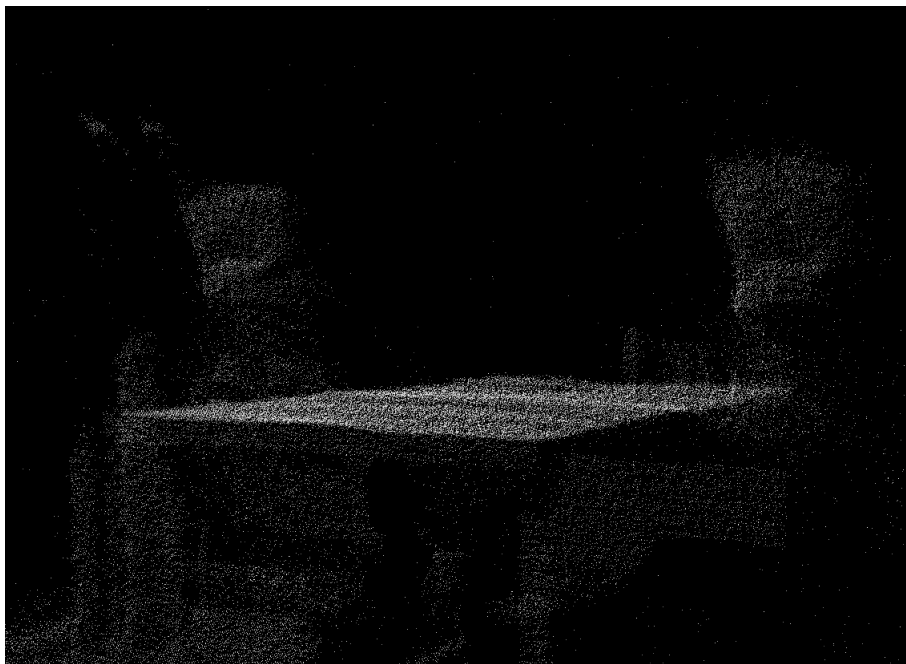


Figure 5.5: The reference cloud without a box on the table.

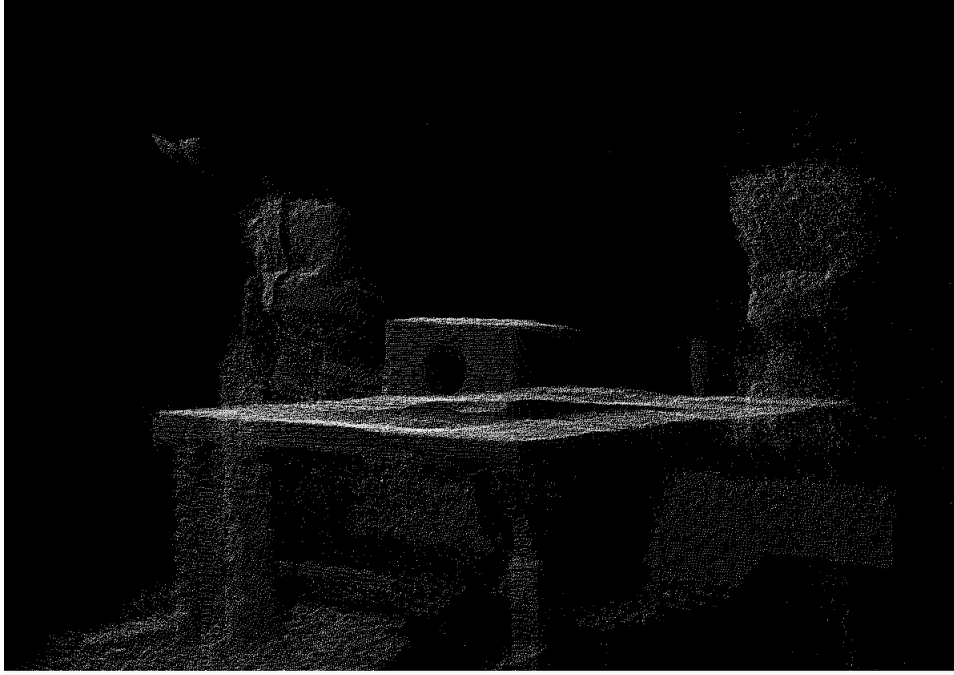


Figure 5.6: The target cloud with a box on the table.

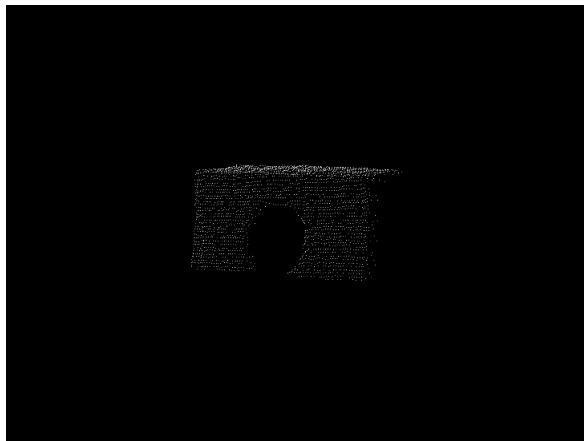


Figure 5.7: The result after the segmentation. The box is the only cluster bigger than 500 points, and thus the only one showed.

## 5.6 Discussion

We can see there is potential in computer vision for robotic cleaning of salmon slaughter houses. Using a library such as PCL, it is a relatively simple task to achieve acceptable results. There are however a lot of descriptors and detectors, each having its strengths and weaknesses, making it time consuming finding the right one for the application.

The results from the transformation using both features and ICP seemed good. While no absolute measurement of error is available, it would be reasonable to claim an accuracy of 1cm based of Figure 5.3 and 5.4. If this is sufficient or not for cleaning is unknown at this point in time, as no tests on cleaning has been performed.

The difference in results from feature matching is probably from the Sample Consensus Initial Alignment algorithm as it just picks a random number of samples, and randomly picks a sample points correspondence. A better alternative might have been to use the Greedy Initial Alignment algorithm as it is more thorough, as explained in Section 2.3.10. However, it is not a problem as the ICP algorithm can cope with a lot of orientational deviation as long as the positioning is not too far off.

In these tests, no experimentation regarding what descriptors and detectors to use were done. Using the ones recommended in [Silvio Filipe \(2014\)](#) and [Aitor Aldoma and Rusu \(2012\)](#) proved to work well enough. Many of the newer descriptors are more suited to real-time applications where the calculation of the descriptors are a huge factor. In the case of cleaning of a salmon slaughterhouse, the computation time is not that important. The robot has to move and cover the whole area to clean, and while it moves between snapshots, there is a lot of time for computation.

The same can be said for the ICP. As outlined in Section 2.3.11, many versions of the ICP algorithm have been developed. While some of them are more favored than the standard version, the standard version made a good match in less than a second. The biggest challenges for ICP is the varying point density for the point clouds to be matched, and noise from the sensor. In addition, the viewpoint of the camera has to be reasonably similar to the viewpoint of one of the ray traced models. As discussed in Section 2.3.14, the closer an object is to the range imaging sensor, the more points it gets. The distance will thus affect the ICP matching to some degree. Another source of error is the range imaging sensor used in the experiment. The Kinect uses a ToF sensor, which is less accurate than structured light sensors. The problem with false depth reading at edges also become apparent when zooming in to the edge of the box, as can be seen in Figure 5.8. The same can be seen for erroneous readings at concavities as seen in Figure 5.9.

Regarding the SegmentDifferences, this seemed to work very good. The only variable that needs to be tweaked is the distance criteria, and with a value of 0.001, which is equal to 1mm, the results were satisfying. However, this number will probably depend upon the sensor used to capture the range image.

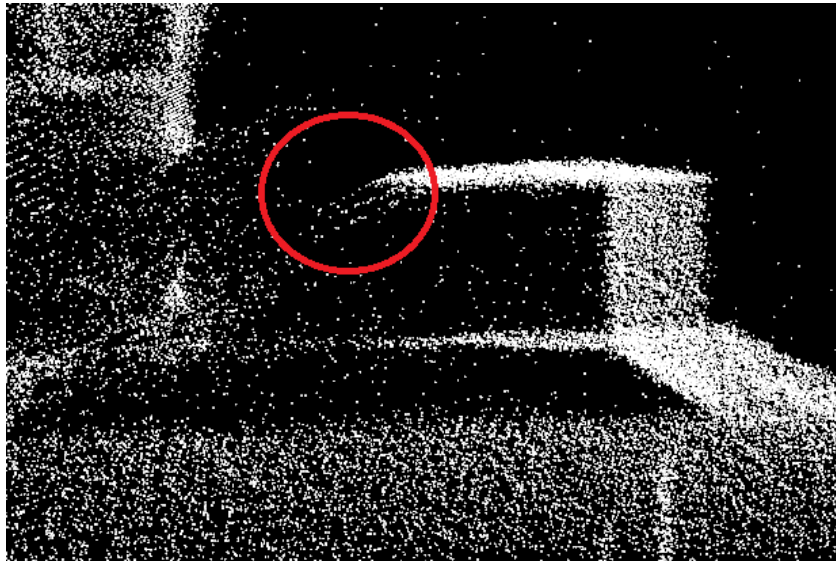


Figure 5.8: False depth reading at the edge of the box.

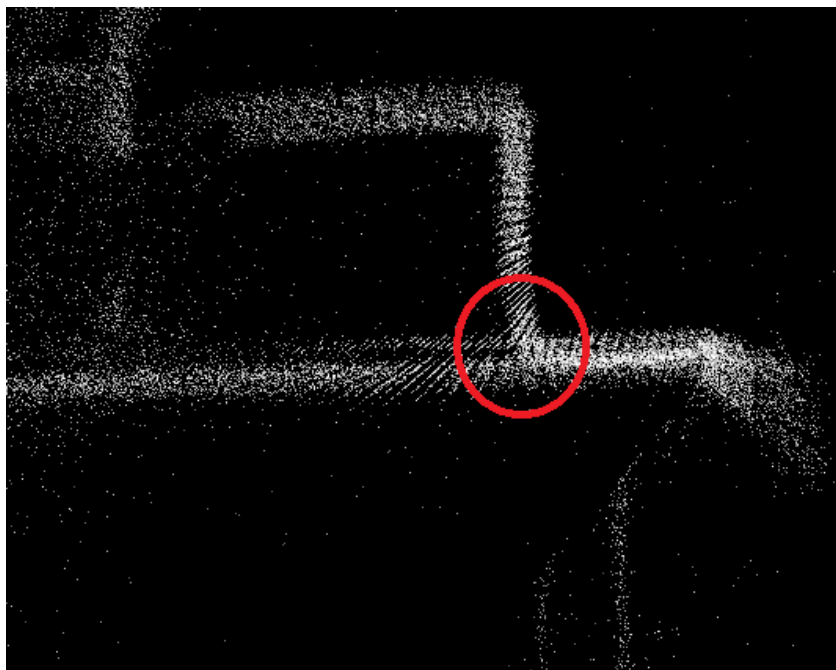


Figure 5.9: Erroneous reading due to concavities.

# Chapter 6: Trajectory generation

## 6.1 Introduction

To be able to clean the machines with a good result, the robot has to move at an appropriate distance from the surfaces of the machine. In addition, it needs to spray the whole surfaces, preferably with some overlap. The machines themselves does have a somewhat complex geometry, possibly leading to the necessity to develop a system to scan a surface that needs to be cleaned, and then automatically derive the trajectories. The points in the trajectory can be derived by calculating the normals to the scanned surface, and then use the normals times the wanted distance to get a point and orientation of in the trajectory.

Some work has already been done on this field (Chen et al., 2002) in a spray painting context. Spray painting and washing is a very similar process, making the approach viable for washing. The big drawback of this method is the complexity of the scanned geometry. A relatively simple curved geometry like a car door will give reasonable result with any modification to the geometry as it is a continuous surface with few sudden changes in normals to the surface.

The hard way is to manually program all the paths for the robot. This will be very time consuming, at least for a big factory. It will probably also require some adjustment from the first draft of the robot paths to gain a satisfactory cleaning result.

## 6.2 Method

The works was carried out using PCL. PCL has methods for calculating normals from a point cloud. All the normals have to point in the same direction relative to the surface, i.e. either out of the surface, or inwards in the component. As explained in Section 2.3.13 a normal will point towards the viewpoint which is set.

The test was done both using a CAD model of the same box as used in Section 5.2 and with a point cloud captured by the Kinect.

## 6.3 Result

Figure 6.1 shows the box and the calculated normals. In order to get all the normals to point outwards the viewpoint is set in the middle of the box, and the direction of each normal reversed. Figure 6.2 shows the normals calculated for the table with the box on it. Some noise from the Kinect are present, making the image more difficult to interpret. The viewpoint is here set above the table to have all the normals point in the right direction. Robot positions are then calculated by using the point from the Kinect and its associated normal with a specified distance from the surface. These points are visualized by red dots. To reduce the number of points and thus the robot positions, and to get a uniform point density, a downsampling is done with a voxel size of 0.1m.

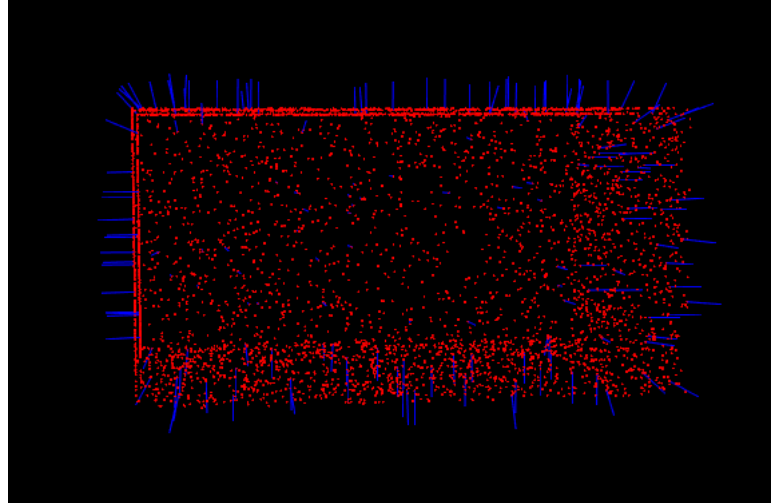


Figure 6.1: The box with the calculated normals. All normals point out of the box (difficult to see in 2D).

The results from automatic trajectory generation on an electric stunner can be seen in Figure 6.3. However, some of the normals does not point in the wanted direction, as can be seen in Figure 6.4.

All these points can then be used as points the robot need to go to, either with a linear motion, or a three point curve. The points are organized according to the viewpoint of the Kinect, and with a viewpoint as in the test, the points starts at the back of the table, and moves to the front, and then starts at the back again. This makes it possible in this case to use the points as is, with the need to reorganize the order. A possible solution to organize the points is to align the point cloud and the normals according to a coordinate system.

#### 6.4 Discussion

As can be seen in Figure 6.2, noise from the Kinect is present. In this case this will result in robot positions that are useless. No noise filtering was done on the captured point cloud, and this would probably have removed most of the noise, but not all. However, when using a CAD model, the issue with noise is removed.

The problem with automatic trajectory generation is the complex geometry of the components. With a simple box and a table, all the points will be easily accessible, but with the complex geometry of the electric stunner that is not the case. First of all, there is no set viewpoint that will result in all the surface normals to point outwards of their respective surfaces, as is obvious in Figure 6.4. In addition, certain areas are not accessible to the robot, and advanced collision avoidance would be necessary to be able to reach all the different positions on the components. A possible solution to automatically generate the trajectories would be to "smooth out" the surfaces, making them less complex. For instance, make the whole finger assembly as seen in Figure 3.1 just a box, and calculate the trajectories from that. Another solution can be to split all the parts that make up a component, and calculate the normals for each part individually.

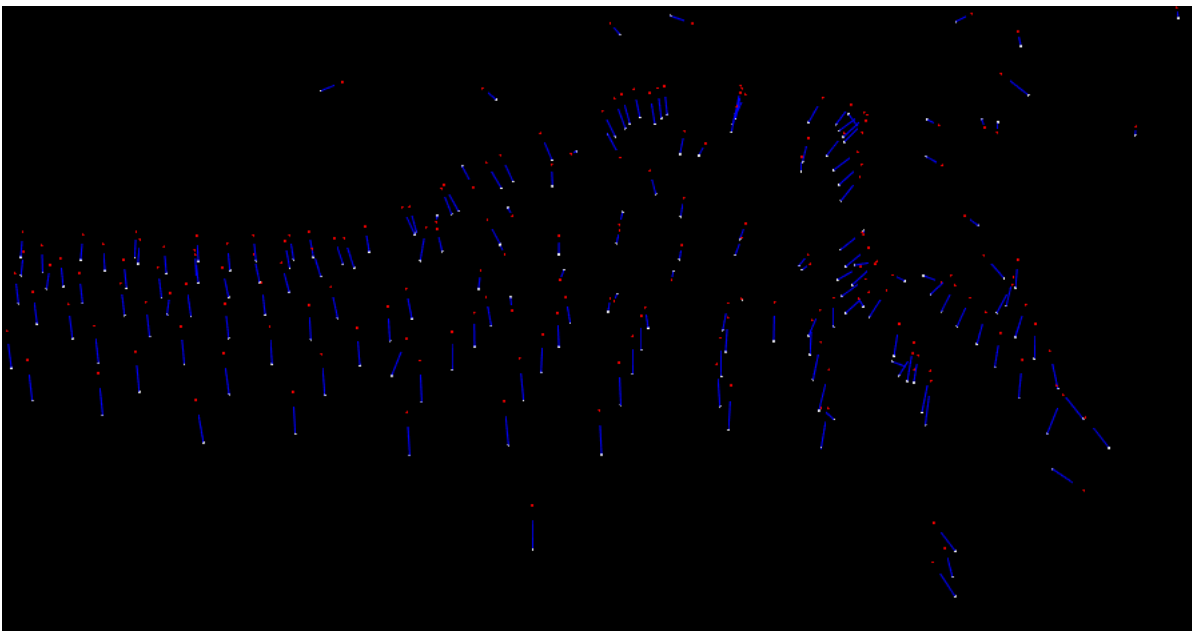


Figure 6.2: The normals for the table with the box. The points captured by the Kinect are white, red dots are the calculated robot positions.

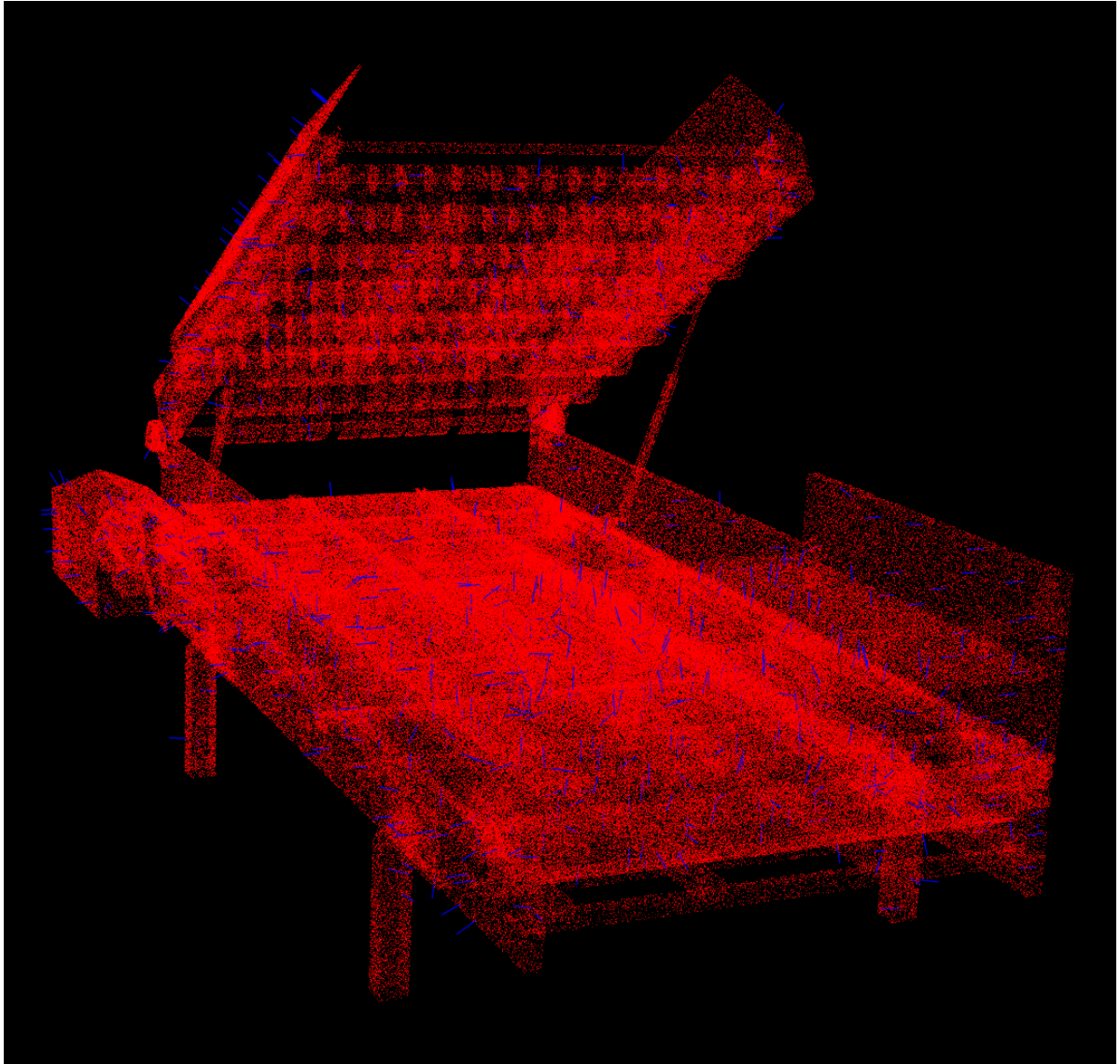


Figure 6.3: A point cloud of 1 million points representing the electric stunner, with normals calculated for every 1000th point.



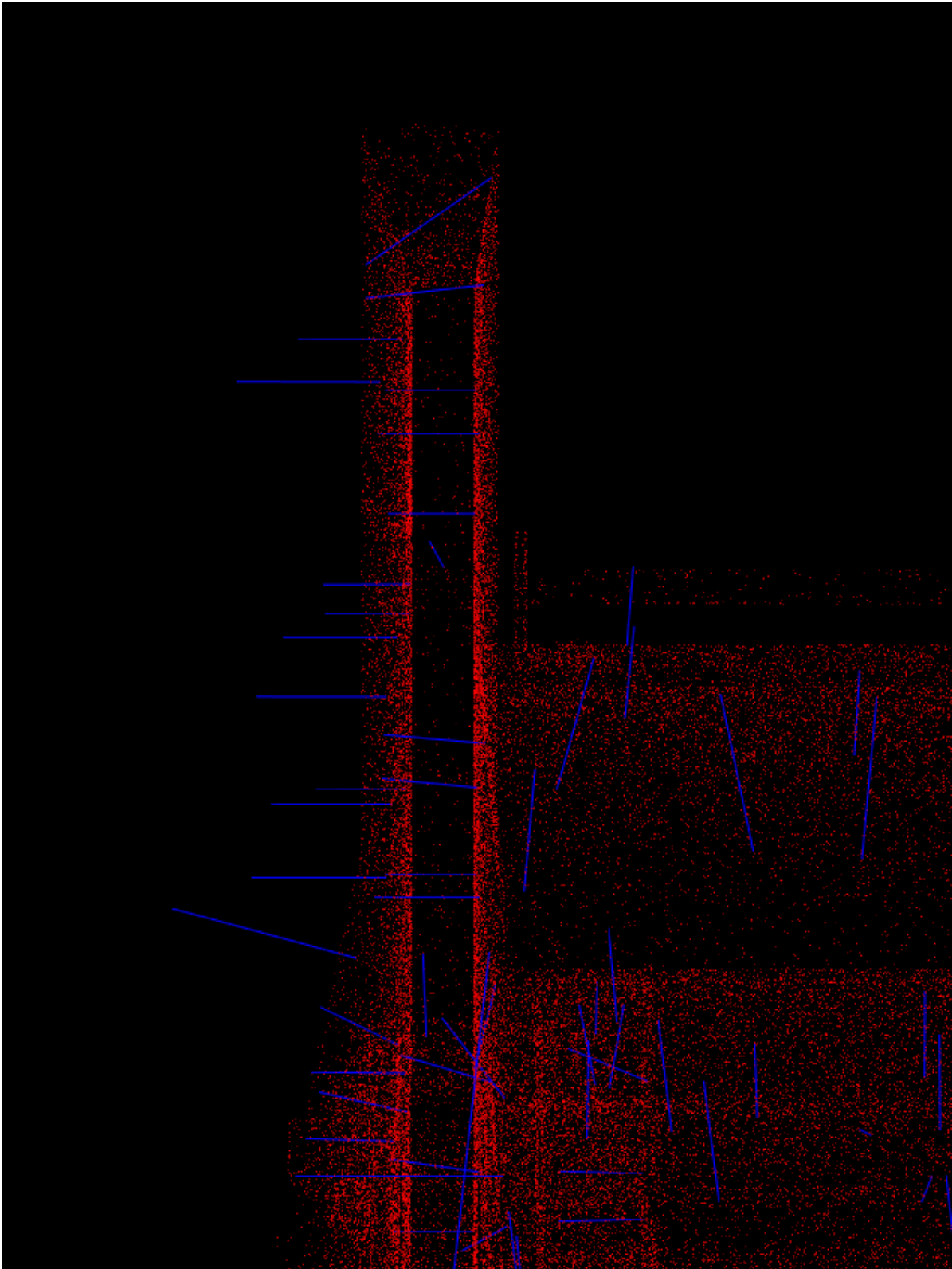


Figure 6.4: The problem with calculating the normals for a complex geometry. The normals on both the inside and outside of the plate point outwards.



# Chapter 7: Conclusion

In this thesis we have seen various solutions to the robotic cleaning problem, and simulations of those solutions. We have also seen tools that may be helpful to achieve good cleaning quality, i.e. computer vision for generating robot trajectories, and automatically adjust them if components are moved.

The results from the mechanical designs has made it possible to build a first prototype, and one can conclude that there is no perfect solution to this problem. Further work with trail and error will be necessary to complete the project. The slaughterhouses are complicated environments with relatively long distances for a robot, high requirements for cleanliness and with an infrastructure that can not easily be modified. Many of the solutions seemed promising, with the custom robot seeming the most promising. It can almost be concluded that the custom robot is the only solution that meet the requirements for the salmon industry if the whole slaughterhouse shall be cleaned by a robot.

It can be concluded that computer vision is beneficial for a robotic cleaning solution. However, computer vision is not without problems. The results did vary even though none of the parameters were changed, giving cause of worry. As discussed, this was probably due to the SAC-IA algorithm, and Greedy Initial Alignment would maybe have solved this. Some problems with noise were also present, and the tests were done at fairly short distances, which indicate a more expensive sensor may be required for industrial applications. However, it can be concluded that computer vision easily can detect objects that should not be in a scene.

Automatic generation of trajectories for robotic cleaning is an interesting idea, but the complex geometries for the components makes it difficult to achieve. For this project it will be too complex and time consuming, making the project harder to complete than it has to be. It would, however, have been an interesting area of research if time allowed it.

## 7.1 Future work

This thesis is a preliminary study of certain aspects of the bigger project. Further work will have to be done to complete a fully working robotic cleaning solution. As mentioned in this thesis, the next phase of the project will be to build a prototype to start testing. This prototype will test the positioning ability of solution 3 and 4, help with establishing robot trajectories for cleaning, give insight in the forces applied by the nozzle and thus the required rigidity of the robot and suspension, and help determine if there are certain problem areas on the components, e.g. the fingers on the electric stunner, that will affect the requirements for agility/accessibility of a final design. However, it is also recommended that a full scale custom robot is built, to test the validity of that solution also.

Future work should also include more work on computer vision to make sure it delivers the same result time after time, starting by trying the Greedy Inital Alignment algorithm, and be using a more accurate sensor.



# Bibliography

- Aitor Aldoma, Zoltan-Csaba Marton, F. T. W. W. C. P. and Rusu, R. (2012). Point cloud library tutorial: Three-dimensional object recognition and 6 dof pose estimation. *IEEE Robotics and Automation Magazine* 19.
- Akvakultur (2014). <https://www.ssb.no/jord-skog-jakt-og-fiskeri/statistikker/fiskeoppdrett/aar-foreloepige/2015-06-02>.
- Alex Flint, A. D. and Hengel, A. (2007). Local 3d structure recognition in range images. *In 9th Biennial Conference of the Australian Pattern Recognition Society on Digital Image Computing Techniques and Applications*.
- Antoine Lejeune, M. V. D. and Verly, J. (2011). The secrets of the kinect. <https://orbi.ulg.ac.be/bitstream/2268/104993/1/Lejeune2011TheSecrets.pdf>.
- Appel, A. (1968). Some techniques for shading machine renderings of solids. *AFIPS Conference*.
- Chen, H., Xi, N., Sheng, W., Song, M., and Chen, Y. (2002). Cad-based automated robot trajectory planning for spray painting of free-form surfaces. *The Industrial Robot*, 29(5):426–433.
- Curviline (2016). <http://www.rollon.com/IN/en/products/linear-line/5-curviline/>.
- David Fofi, Tadeusz Sliwa, Y. V. (2004). A comparative survey on invisible structured light. *Machine Vision Applications in Industrial Inspection XII*.
- DN (2015). <http://www.dn.no/nyheter/2015/01/07/1243/Fisk/norsk-fisk-verdt-69-milliarder>.
- Downsampling (2016). Point cloud library. [http://pointclouds.org/documentation/tutorials/voxel\\_grid.php](http://pointclouds.org/documentation/tutorials/voxel_grid.php).
- Elasticity (2016). [http://www.engineeringtoolbox.com/young-modulus-d\\_417.html](http://www.engineeringtoolbox.com/young-modulus-d_417.html).
- Gopi, M. and Krishnan, S. (2002). A fast and efficient projection-based approach for surface reconstruction. *SIBGRAPI '02: Proceedings of the 15th Brazilian Symposium on Computer Graphics and Image Processing*, pages 179–186.
- GrabCAD (2016). <https://grabcad.com/library/universal-robot>.
- Hydraulics (2006). <http://machinedesign.com/archive/water-hydraulics-your-future>.
- IGN-board (2013). On the differences between kinect 1, kinect 2 and the sony camera. <http://www.ign.com/boards/threads/on-the-differences-between-kinect-1-kinect-2-and-the-sony-camera.453537413/>.
- IGUS (2016). [http://www.igus.com/wpck/7181/drylin\\_zlw](http://www.igus.com/wpck/7181/drylin_zlw).

- KUKA (2016). [http://www.kuka-robotics.com/en/products/industrial\\_robots/small\\_robots/kr10\\_r1100\\_sixx/](http://www.kuka-robotics.com/en/products/industrial_robots/small_robots/kr10_r1100_sixx/).
- Lowe, D. G. (1999). Object recognition from local scale-invariant features. *Proceedings of the International Conference on Computer Vision*.
- Martin A. Fischler, R. C. B. (1980). Random sample consensus: A paradigm for model fitting with applications to image analysis and automated cartography. *Journal of WSCG 21*.
- Nozzle (2016). [http://www.ikeuchiusa.com/technical\\_information.html](http://www.ikeuchiusa.com/technical_information.html).
- OmronPLC (2016). <https://industrial.omron.us/en/products/nx7#features>.
- OmronServo (2016). <https://www.ia.omron.com/products/family/2644/download/catalog.html>.
- Parr, E. (1998). *Industrial Control Handbook*. Industrial Press.
- Passthrough (2016). Point cloud library. <http://pointclouds.org/documentation/tutorials/passthrough.php>.
- Paul J. Besl, N. D. M. (1992). A method for registration of 3-d shapes. *IEEE Transactions on Pattern Analysis and Machine Intelligence*.
- Pipeline (2016). Pcl/openni tutorial 4: 3d object recognition (descriptors). [http://robotica.unileon.es/mediawiki/index.php/PCL/OpenNI\\_tutorial\\_4:\\_3D\\_object\\_recognition\\_\(descriptors\)](http://robotica.unileon.es/mediawiki/index.php/PCL/OpenNI_tutorial_4:_3D_object_recognition_(descriptors)).
- Pomerleau, F., Colas, F., Siegwart, R., and Magnenat, S. (2013). Comparing icp variants on real-world data sets. *Autonomous Robots*, 34(3):133–148.
- PRT (2016). [http://www.igus.com/wpck/3734/iglidur\\_prt?C=US](http://www.igus.com/wpck/3734/iglidur_prt?C=US).
- R. B. Rusu, Z. C. Marton, N. B. M. D. and Beetz, M. (2008). Towards 3d point cloud based object maps for household environments. *Robotics and Autonomous Systems Journal*.
- Radu Bogdan Rusu, Nico Blodow, M. B. (2009). Fast point feature histograms (fpfh) for 3d registration. *IEEE International Conference on Robotics and Automation*.
- Radu Bogdan Rusu, Gary Bradski, R. T. J. H. (2010). Fast 3d recognition and pose using the viewpoint feature histogram. *Intelligent Robots and Systems*.
- Radu Bogdan Rusu, Nico Blodow, Z. C. M. M. B. (2008). Aligning point cloud views using persistent feature histograms. *International Conference on Intelligent Robots and Systems*.
- RoboLink (2016). [http://www.igus.eu/wpck/15257/roboLink\\_D\\_2x\\_PRT\\_lager](http://www.igus.eu/wpck/15257/roboLink_D_2x_PRT_lager).
- Robot-joint (2016). [http://www.igus.co.uk/wpck/13351/N15\\_10\\_04\\_roboLink\\_D\\_Direktantrieb?C=GB&L=en](http://www.igus.co.uk/wpck/13351/N15_10_04_roboLink_D_Direktantrieb?C=GB&L=en).
- ROS (2014). <http://wiki.ros.org/ROS/Introduction>.
- RoseKrieger (2016). <https://www.rk-rose-krieger.com/english/products/linear-technology/for-those-familiar-with-our-products/linear-units/linear-units-with-spindle-drive/>.

- RoseKriegerEPX (2016). <http://rose-krieger.partcommunity.com/3d-cad-models/sso>.
- Rusu, R. B. (2009). *Semantic 3D Object Maps for Everyday Manipulation in Human Living Environments*. PhD thesis, Technischen Universität München.
- Segmentation (2016). Point cloud library. [http://www.pointclouds.org/documentation/tutorials/planar\\_segmentation.php](http://www.pointclouds.org/documentation/tutorials/planar_segmentation.php).
- SegmentDifferences (2016). Segmentdifferences pcl api documentation. [http://docs.pointclouds.org/1.5.1/classpcl\\_1\\_1\\_segment\\_differences.html](http://docs.pointclouds.org/1.5.1/classpcl_1_1_segment_differences.html).
- Sergi Foix, G. A. and Torras, C. (2011). Lock-in time-of-flight (tof) cameras: A survey. *IEEE sensors journal*, vol 11.
- ServoCAD (2016). <http://www.omron.com.au/products/family/2644/download/cad.html>.
- Siciliano, B., Sciavicco, L., Villani, L., Oriolo, G., and SpringerLink (2009). *Robotics : Modeling, Planning and Control*. Springer London, London.
- Silvio Filipe, L. A. A. (2014). A comparative evaluation of 3d keypoint detectors in a rgb-d object dataset. *FCT*.
- SlewingDrive (2016). <http://conedrive.com/Products/Slewing-Solutions/slewing-drives.php>.
- StatisticalOutlierRemoval (2016). Point cloud library. [http://pointclouds.org/documentation/tutorials/statistical\\_outlier.php](http://pointclouds.org/documentation/tutorials/statistical_outlier.php).
- VisualComponents (2015). <http://www.visualcomponents.com/>.
- Wiki (2016). Ransac wiki article. <https://en.wikipedia.org/wiki/RANSAC>.
- Zoltan Csaba Marton, Radu Bogdan Rusu, M. B. (2009). On fast surface reconstruction methods for large and noisy point clouds. *IEEE International Conference on Robotics and Automation*, pages 3218–3223.
- Zugara (2014). How does the kinect 2 compare to the kinect 1? <http://zugara.com/how-does-the-kinect-2-compare-to-the-kinect-1>.

## Appendix A: Python script for solution 4

```
1 from vcScript import *
2 import vcMatrix
3
4 comp_track = getApplication().findComponent("Assem_fot_svivel")
5 servo = comp_track.findBehaviour("ServoController")
6
7 comp = getComponent().findBehaviour('Executor')
8 list = comp.SubRoutines
9 testroutine = list[0]
10
11 def OnStart():
12     for i in servo.Joints:
13         print i.Name, i.InitialValue
14
15
16 def OnSignal( signal ):
17     pass
18
19 def OnRun():
20     mtx = vcMatrix.new()
21     mtx.translateAbs(1000,-300,150)
22     mtx.rotateRelY(180)
23     mtx.rotateRelX(180)
24     my_statement = testroutine.createStatement(VC_STATEMENT_PTPMOTION)
25     my_statement.Tool='Tool_1'
26     my_statement.ExternalJoint1 = 0
27     my_statement.ExternalJoint2 = 90
28     my_statement.ExternalJoint3 = 0
29     my_statement.ExternalJoint4 = 0
30     my_statement.Target = mtx
31
32     mtx = vcMatrix.new()
33     mtx.translateAbs(1000,-300,150)
34     mtx.rotateRelY(180)
35     mtx.rotateRelX(180)
36     mtx.rotateRelY(-45)
37     mtx.rotateRelZ(-45)
38     my_statement = testroutine.createStatement(VC_STATEMENT_LINMOTION)
39     my_statement.Tool='Tool_1'
40     my_statement.ExternalJoint1 = 0
41     my_statement.ExternalJoint2 = 0
42     my_statement.ExternalJoint3 = 0
43     my_statement.ExternalJoint4 = 0
44     my_statement.Target = mtx
45
46     mtx = vcMatrix.new()
47     mtx.translateAbs(1000,-300,150)
48     mtx.rotateRelY(180)
49     mtx.rotateRelX(180)
```



```

50 mtx.rotateRelY(-45)
51 mtx.rotateRelZ(-45)
52 my_statement = testroutine.createStatement(VC_STATEMENT_LINMOTION)
53 my_statement.Tool='Tool_1'
54 my_statement.ExternalJoint1 = 0
55 my_statement.ExternalJoint2 = 0
56 my_statement.ExternalJoint3 = 0
57 my_statement.ExternalJoint4 = 900
58 my_statement.Target = mtx
59
60 mtx = vcMatrix.new()
61 mtx.translateAbs(1000,300,650)
62 mtx.rotateRelY(180)
63 mtx.rotateRelX(180)
64 my_statement = testroutine.createStatement(VC_STATEMENT_PTPMOTION)
65 my_statement.Tool='Tool_1'
66 my_statement.ExternalJoint1 = 0
67 my_statement.ExternalJoint2 = 0
68 my_statement.ExternalJoint3 = 0
69 my_statement.ExternalJoint4 = 900
70 my_statement.Target = mtx
71
72 mtx = vcMatrix.new()
73 mtx.translateAbs(0,1500, 1280)
74 mtx.rotateRelY(180)
75 mtx.rotateRelX(180)
76 mtx.rotateRelZ(90)
77 my_statement = testroutine.createStatement(VC_STATEMENT_PTPMOTION)
78 my_statement.Tool='Tool_1'
79 my_statement.ExternalJoint1 = 45
80 my_statement.ExternalJoint2 = 0
81 my_statement.ExternalJoint3 = 500
82 my_statement.ExternalJoint4 = 900
83 my_statement.Target = mtx
84
85 mtx = vcMatrix.new()
86 mtx.translateAbs(0,500, 1280)
87 mtx.rotateRelY(180)
88 mtx.rotateRelX(180)
89 my_statement = testroutine.createStatement(VC_STATEMENT_LINMOTION)
90 my_statement.Tool='Tool_1'
91 my_statement.ExternalJoint1 = 90
92 my_statement.ExternalJoint2 = 0
93 my_statement.ExternalJoint3 = 500
94 my_statement.ExternalJoint4 = 900
95 my_statement.Target = mtx
96
97 for k in range(1,10):
98     mtx = vcMatrix.new()
99     mtx.translateAbs(0-k*70,1500, 1280+k*40)
100     mtx.rotateRelY(180)
101     mtx.rotateRelX(180)
102     mtx.rotateRelZ(90)
103     my_statement = testroutine.createStatement(VC_STATEMENT_LINMOTION)
104     my_statement.Tool='Tool_1'
105     my_statement.ExternalJoint1 = 45
106     my_statement.ExternalJoint2 = 0

```

```
107 my_statement.ExternalJoint3 = 500+k*30
108 my_statement.ExternalJoint4 = 900
109 my_statement.Target = mtx
110
111 mtx = vcMatrix.new()
112 mtx.translateAbs(0-k*70,500, 1280+k*40)
113 mtx.rotateRelY(180)
114 mtx.rotateRelX(180)
115 my_statement = testroutine.createStatement(VC_STATEMENT_LINMOTION)
116 my_statement.Tool='Tool_1'
117 my_statement.ExternalJoint1 = 90
118 my_statement.ExternalJoint2 = 0
119 my_statement.ExternalJoint3 = 500+k*30
120 my_statement.ExternalJoint4 = 900
121 my_statement.Target = mtx
122
123 for k in range(11,16):
124     mtx = vcMatrix.new()
125     mtx.translateAbs(0-k*70,1500, 1280+k*40)
126     mtx.rotateRelY(180)
127     mtx.rotateRelX(180)
128     mtx.rotateRelZ(90)
129     my_statement = testroutine.createStatement(VC_STATEMENT_LINMOTION)
130     my_statement.Tool='Tool_1'
131     my_statement.ExternalJoint1 = 45
132     my_statement.ExternalJoint2 = 0
133     my_statement.ExternalJoint3 = 500+k*30
134     my_statement.ExternalJoint4 = 900
135     my_statement.Target = mtx
136
137     mtx = vcMatrix.new()
138     mtx.translateAbs(0-k*70,500, 1280+k*40)
139     mtx.rotateRelY(180)
140     mtx.rotateRelX(180)
141     mtx.rotateRelZ(90)
142     my_statement = testroutine.createStatement(VC_STATEMENT_LINMOTION)
143     my_statement.Tool='Tool_1'
144     my_statement.ExternalJoint1 = 90
145     my_statement.ExternalJoint2 = 0
146     my_statement.ExternalJoint3 = 500+k*30
147     my_statement.ExternalJoint4 = 900
148     my_statement.Target = mtx
149
150
151     mtx = vcMatrix.new()
152     mtx.translateAbs(0, 1500, 1180)
153     mtx.rotateRelY(180)
154     mtx.rotateRelX(180)
155     mtx.rotateRelZ(90)
156     my_statement = testroutine.createStatement(VC_STATEMENT_PTPMOTION)
157     my_statement.Tool='Tool_1'
158     my_statement.ExternalJoint1 = 0
159     my_statement.ExternalJoint2 = 0
160     my_statement.ExternalJoint3 = 0
161     my_statement.ExternalJoint4 = 900
162     my_statement.Target = mtx
163
```

```
164 mtx = vcMatrix.new()
165 mtx.translateAbs(570, 1500, 1200)
166 mtx.rotateRelY(180)
167 mtx.rotateRelX(180)
168 mtx.rotateRelZ(90)
169 mtx.rotateRelX(135)
170 my_statement = testroutine.createStatement(VC_STATEMENT_PTPMOTION)
171 my_statement.Tool='Tool_1'
172 my_statement.ExternalJoint1 = -45
173 my_statement.ExternalJoint2 = 0
174 my_statement.ExternalJoint3 = 500
175 my_statement.ExternalJoint4 = 900
176 my_statement.Target = mtx
177
178 mtx = vcMatrix.new()
179 mtx.translateAbs(100, 1500, 1700)
180 mtx.rotateRelY(180)
181 mtx.rotateRelX(180)
182 mtx.rotateRelZ(90)
183 mtx.rotateRelX(135)
184 my_statement = testroutine.createStatement(VC_STATEMENT_LINMOTION)
185 my_statement.Tool='Tool_1'
186 my_statement.ExternalJoint1 = -45
187 my_statement.ExternalJoint2 = 0
188 my_statement.ExternalJoint3 = 500
189 my_statement.ExternalJoint4 = 900
190 my_statement.Target = mtx
191
192 mtx = vcMatrix.new()
193 mtx.translateAbs(70, 500, 1720)
194 mtx.rotateRelY(180)
195 mtx.rotateRelX(180)
196 mtx.rotateRelZ(90)
197 mtx.rotateRelX(135)
198 mtx.rotateRelZ(-90)
199 my_statement = testroutine.createStatement(VC_STATEMENT_LINMOTION)
200 my_statement.Tool='Tool_1'
201 my_statement.ExternalJoint1 = -45
202 my_statement.ExternalJoint2 = 0
203 my_statement.ExternalJoint3 = 500
204 my_statement.ExternalJoint4 = 900
205 my_statement.Target = mtx
206
207 mtx = vcMatrix.new()
208 mtx.translateAbs(-125, 1500, 1860)
209 mtx.rotateRelY(180)
210 mtx.rotateRelX(180)
211 mtx.rotateRelZ(90)
212 mtx.rotateRelX(135)
213 mtx.rotateRelZ(-45)
214 my_statement = testroutine.createStatement(VC_STATEMENT_LINMOTION)
215 my_statement.Tool='Tool_1'
216 my_statement.ExternalJoint1 = -45
217 my_statement.ExternalJoint2 = 0
218 my_statement.ExternalJoint3 = 700
219 my_statement.ExternalJoint4 = 900
220 my_statement.Target = mtx
```

```
221
222     mtx = vcMatrix.new()
223     mtx.translateAbs(-125, 500, 1860)
224     mtx.rotateRelY(180)
225     mtx.rotateRelX(180)
226     mtx.rotateRelZ(90)
227     mtx.rotateRelX(135)
228     mtx.rotateRelZ(-90)
229     my_statement = testroutine.createStatement(VC_STATEMENT_LINMOTION)
230     my_statement.Tool='Tool_1'
231     my_statement.ExternalJoint1 = -45
232     my_statement.ExternalJoint2 = 0
233     my_statement.ExternalJoint3 = 700
234     my_statement.ExternalJoint4 = 900
235     my_statement.Target = mtx
236
237
238 def OnReset():
239     for k in range(0, 500):
240         testroutine.deleteStatement(len(testroutine.Statements)-1)
```



## Appendix B: The computer vision computation class

```
1 //
2 // Created by Emil Bjoerlykhaug 06.04.16
3 //
4
5 #include "hahaha/computing.hpp"
6
7
8 /**
9  * @brief Constructor of the Computing class. Takes no arguments
10  */
11 Computing::Computing(){
12
13 }
14
15 /**
16  * @brief Calculate VFH descriptors for each point cloud in the input cluster
17  * @param clusters The input cluster to calculate the descriptors for
18  * @return A vector with the descriptors for the point cloud in the cluster
19  */
20 std::vector<pcl::PointCloud<pcl::VFHSignature308>::Ptr> Computing::
    calculateVFHDescriptors(std::vector<pcl::PointCloud<pcl::PointXYZ>::Ptr>
        clusters)
21 {
22     //Debug
23     std::cout << "Hi!" << std::endl;
24     //Create a variable for the descriptors
25     std::vector<pcl::PointCloud<pcl::VFHSignature308>::Ptr> descriptors;
26
27     //Iterate through every item in the cluster, and calculate descriptors for
    it
28     for(int i=0;i<clusters.size();i++){
29         // Object for storing the normals.
30         pcl::PointCloud<pcl::Normal>::Ptr normals(new pcl::PointCloud<pcl::
            Normal>);
31         // Object for storing the VFH descriptor.
32         pcl::PointCloud<pcl::VFHSignature308>::Ptr descriptor(new pcl::
            PointCloud<pcl::VFHSignature308>);
33
34         // Estimate the normals.
35         pcl::NormalEstimation<pcl::PointXYZ, pcl::Normal> normalEstimation;
36         normalEstimation.setInputCloud(clusters[i]);
37         normalEstimation.setRadiusSearch(0.03);
38         pcl::search::KdTree<pcl::PointXYZ>::Ptr kdtree(new pcl::search::KdTree<
            pcl::PointXYZ>);
39         normalEstimation setSearchMethod(kdtree);
40         normalEstimation.compute(*normals);
```

```

41
42     // VFH estimation object.
43     pcl::VFHEstimation<pcl::PointXYZ, pcl::Normal, pcl::VFHSignature308>
         vfh;
44     vfh.setInputCloud(clusters[i]);
45     vfh.setInputNormals(normals);
46     vfh.setSearchMethod(kdtree);
47     // Optionally, we can normalize the bins of the resulting histogram,
48     // using the total number of points.
49     vfh.setNormalizeBins(true);
50     // Also, we can normalize the SDC with the maximum size found between
51     // the centroid and any of the cluster's points.
52     vfh.setNormalizeDistance(false);
53     //Compute the descriptor
54     vfh.compute(*descriptor);
55     descriptors.push_back(descriptor);
56 }
57 return descriptors;
58 }
59
60 /**
61  * @brief Passthrough filter on the input cloud
62  * @param cloud_before_filter The input cloud
63  * @param zFilter Bool if the filter also should filter in the z-axis
64  * @return the filtered cloud
65  */
66 pcl::PointCloud<pcl::PointXYZ>::Ptr Computing::passThroughFilter(pcl::
        PointCloud<pcl::PointXYZ>::Ptr cloud_before_filter, bool zFilter)
67 {
68     //Initialize variables
69     pcl::PointCloud<pcl::PointXYZ>::Ptr cloud(new pcl::PointCloud<pcl::PointXYZ
        >);
70     pcl::PointCloud<pcl::PointXYZ>::Ptr cloud1(new pcl::PointCloud<pcl::
        PointXYZ>);
71     pcl::PointCloud<pcl::PointXYZ>::Ptr cloud2(new pcl::PointCloud<pcl::
        PointXYZ>);
72     // Pass through filter
73     pcl::PassThrough<pcl::PointXYZ> pass;
74     pass.setInputCloud (cloud_before_filter);
75     //pass.setFilterFieldName ("z");
76     //pass.setFilterLimits (0, 2.7);
77     //Filter in the y-direction
78     pass.setFilterFieldName ("y");
79     pass.setFilterLimits (-2, 0);
80
81     pass.filter (*cloud1);
82     //Filter in the x-direction
83     pass.setInputCloud(cloud1);
84     pass.setFilterFieldName ("x");
85     pass.setFilterLimits (-0.45, 0.45);
86     pass.filter (*cloud2);
87
88     //Check if the z-direction should be filtered as well.
89     if(zFilter){
90         pass.setInputCloud(cloud2);
91         pass.setFilterFieldName ("z");
92         pass.setFilterLimits (0, 2.350);

```

```

93     pass.filter (*cloud);
94 }
95 else{
96     cloud = cloud2;
97 }
98
99     return cloud;
100 }
101
102 /**
103  * @brief Downsample an input cloud using a voxel grid
104  * @param cloud The input cloud
105  * @param leaf_size The size of the voxels. All the points inside a voxel will
106  *       be replaced with the centroid
107  * @return The downsampled cloud
108  */
109 pcl::PointCloud<pcl::PointXYZ>::Ptr Computing::downsample_voxel(pcl::PointCloud
110 <pcl::PointXYZ>::Ptr cloud, float leaf_size)
111 {
112     pcl::VoxelGrid<pcl::PointXYZ> vg;
113     pcl::PointCloud<pcl::PointXYZ>::Ptr cloud_filtered (new pcl::PointCloud<pcl
114     ::PointXYZ>);
115     vg.setInputCloud (cloud);
116     vg.setLeafSize (leaf_size, leaf_size, leaf_size);
117     vg.filter (*cloud_filtered);
118     return cloud_filtered;
119 }
120
121 /**
122  * @brief A method which will remove the biggest planar face in a point cloud.
123  * @param inCloud The input cloud
124  * @return The point cloud without the planar face
125  */
126 pcl::PointCloud<pcl::PointXYZ>::Ptr Computing::segmentation(pcl::PointCloud<
127 pcl::PointXYZ>::Ptr inCloud)
128 {
129     pcl::PointCloud<pcl::PointXYZ>::Ptr cloud_f(new pcl::PointCloud<pcl::
130     PointXYZ>);
131     // Create the segmentation object for the planar model and set all the
132     parameters
133     pcl::SACSegmentation<pcl::PointXYZ> seg;
134     pcl::PointIndices::Ptr inliers (new pcl::PointIndices);
135     pcl::ModelCoefficients::Ptr coefficients (new pcl::ModelCoefficients);
136     pcl::PointCloud<pcl::PointXYZ>::Ptr cloud_plane (new pcl::PointCloud<pcl::
137     PointXYZ> ());
138     seg.setOptimizeCoefficients (true);
139     seg.setModelType (pcl::SACMODEL_PLANE);
140     seg.setMethodType (pcl::SAC_RANSAC);
141     seg.setMaxIterations (100);
142     seg.setDistanceThreshold (0.01);
143
144     seg.setInputCloud (inCloud);
145     seg.segment (*inliers, *coefficients);
146     //Check if there are not planar faces in the model
147     if (inliers->indices.size () == 0)

```



```

143     {
144         std::cout << "Could not estimate a planar model for the given dataset."
145             << std::endl;
146         //break;
147     }
148     // Extract the planar inliers from the input cloud
149     pcl::ExtractIndices<pcl::PointXYZ> extract;
150     extract.setInputCloud (inCloud);
151     extract.setIndices (inliers);
152     extract.setNegative (false);
153
154     // Get the points associated with the planar surface
155     extract.filter (*cloud_plane);
156     std::cout << "PointCloud representing the planar component: " <<
157         cloud_plane->points.size () << " data points." << std::endl;
158
159     // Remove the planar inliers, extract the rest
160     extract.setNegative (true);
161     extract.filter (*cloud_f);
162     return cloud_f;
163 }
164
165 /**
166  * @brief Divide an input point cloud into clusters of points
167  * @param inCloud The input point cloud
168  * @return the clusters
169  */
170 std::vector<pcl::PointCloud<pcl::PointXYZ>::Ptr> Computing::extract_clusters(
171     pcl::PointCloud<pcl::PointXYZ>::Ptr inCloud)
172 {
173     std::vector<pcl::PointCloud<pcl::PointXYZ>::Ptr> clusters;
174     // Creating the KdTree object for the search method of the extraction
175     pcl::search::KdTree<pcl::PointXYZ>::Ptr tree (new pcl::search::KdTree<pcl::
176         PointXYZ>);
177     tree->setInputCloud (inCloud);
178
179     std::vector<pcl::PointIndices> cluster_indices;
180     pcl::EuclideanClusterExtraction<pcl::PointXYZ> ec;
181     ec.setClusterTolerance (0.01); // 2cm
182     ec.setMinClusterSize (200);
183     ec.setMaxClusterSize (25000);
184     ec.setSearchMethod (tree);
185     ec.setInputCloud (inCloud);
186     ec.extract (cluster_indices);
187
188     //Put all the point cloud in a vector, the cluster vector. Also prints the
189     //size of the each cluster
190     for(int i = 0; i < cluster_indices.size(); i++){
191         pcl::PointCloud<pcl::PointXYZ>::Ptr tmpcloud (new pcl::PointCloud<pcl::
192             PointXYZ>);
193         pcl::copyPointCloud(*inCloud, cluster_indices[i],*tmpcloud);
194         clusters.push_back(tmpcloud);
195         std::cout << "PointCloud representing the Cluster: " << tmpcloud->size
196             () << " data points." << std::endl;
197     }
198 }

```

```

193     return clusters;
194 }
195
196 /**
197  * @brief A method for generating a search tree. A search tree is required for
198  * finding the correct cluster with the help of the VFH descriptor. Method is
199  * required for matching clouds
200  * @param inDescriptor the descriptor to generate a search tree for
201  * @return the search tree for given descriptor
202  */
203 pcl::KdTreeFLANN<pcl::VFHSignature308>::Ptr Computing::generate_search_tree(std
204 ::vector<pcl::PointCloud<pcl::VFHSignature308>::Ptr> inDescriptor)
205 {
206     // Take global descriptors and put them in a kd tree
207
208     pcl::PointCloud<pcl::VFHSignature308>::Ptr global_descriptor (new pcl::
209     PointCloud<pcl::VFHSignature308>());
210     pcl::KdTreeFLANN<pcl::VFHSignature308>::Ptr search_tree(new pcl::
211     KdTreeFLANN<pcl::VFHSignature308>);
212     for(int i = 0; i < inDescriptor.size(); i++){
213
214         pcl::PointCloud<pcl::VFHSignature308>::Ptr temp_descriptor =
215         inDescriptor.at(i);
216         *global_descriptor += *(temp_descriptor);
217     }
218
219     search_tree->setInputCloud(global_descriptor);
220     std::cout << "Size of tree: " << global_descriptor->size() << std::endl;
221     return (search_tree);
222 }
223
224 /**
225  * @brief Match an input descriptor to a search tree made of a vector of
226  * descriptors. Will return which descriptor in the tree that best matches,
227  * and the square distance of that match
228  * @param global_descriptor The descriptor to match
229  * @param inTree A search tree of descriptors
230  * @return A vector with descriptor in the search tree that best matches, and
231  * the square distance of that match
232  */
233 std::vector<float> Computing::match_cloud(pcl::PointCloud<pcl::VFHSignature308
234 >::Ptr global_descriptor, pcl::KdTreeFLANN<pcl::VFHSignature308>::Ptr
235 inTree)
236 {
237     std::vector<float> return_values;
238     std::vector<int> best_match(1);
239     std::vector<float> square_distance(1);
240
241     inTree->nearestKSearch(global_descriptor->points[0], 1, best_match,
242     square_distance);
243     return_values.push_back(best_match[0]);
244     return_values.push_back(square_distance[0]);
245     return return_values;
246 }
247
248 /**

```

```

238 * @brief A method for finding the best match from two sets of descriptors.
239 * @param cluster_descriptors Descriptors for one of the clusters
240 * @param cad_descriptors Descriptors for the ray traced CAD model
241 * @return Which of the models in the CAD cluster that best matches one of the
        descriptors in the cloud cluster
242 */
243 int Computing::search_for_correct_cluster(std::vector<pcl::PointCloud<pcl::
VFHSignature308>::Ptr> cluster_descriptors, std::vector<pcl::PointCloud<pcl
::VFHSignature308>::Ptr> cad_descriptors)
244 {
245     //Generate a search tree from the CAD descriptors
246     pcl::KdTreeFLANN<pcl::VFHSignature308>::Ptr tree = generate_search_tree(
        cad_descriptors);
247
248     float min_distance = 10000;
249     int correct_cluster = 50;
250     std::vector<float> search_result;
251
252     //For each of the descriptors in the cloud cluster, match it against the
        search tree from the CAD descriptors
253     for (int i = 0; i < cluster_descriptors.size(); i++){
254         search_result = match_cloud(cluster_descriptors.at(i),tree);
255         std::cout << search_result.at(1) << std::endl;
256         //If the new search result is less than the previous minimum distance,
            change the new minimum distance to this one. And save what is the
            new correct cluster
257         if(search_result[1] < min_distance){
258             min_distance = search_result[1];
259             correct_cluster = i;
260         }
261     }
262     std::cout << "Correct cluster: " << correct_cluster << std::endl;
263     return correct_cluster;
264 }
265 }
266
267 /**
268 * @brief Compute FPFH descriptor for a input cloud. Normals to the cloud,
        keypoints, and search radius is necessary as inputs
269 * @param cloud The input cloud
270 * @param normals The normals to the cloud
271 * @param keypoint Keypoints in the cloud
272 * @param radius The search radius for calculating the descriptor
273 * @return The caculated descriptors
274 */
275 pcl::PointCloud<pcl::FPFHSiganture33>::Ptr Computing::compute_FPFH(pcl::
        PointCloud<pcl::PointXYZ>::Ptr cloud,
276
                pcl::
                PointCloud
                <pcl::
                Normal
                >::Ptr
                normals,
                pcl::
                PointCloud
                <pcl::
                PointXYZ

```

```

>::Ptr
keypoint
, float
radius)
277 {
278     //Initiate object
279     pcl::FPFHEstimationOMP<pcl::PointXYZ, pcl::Normal, pcl::FPFHSignature33>
        fpfh_obj;
280     //To speed up the process
281     fpfh_obj.setNumberOfThreads(8);
282     fpfh_obj.setSearchMethod(pcl::search::Search<pcl::PointXYZ>::Ptr (new pcl::
        search::KdTree<pcl::PointXYZ>));
283     fpfh_obj.setRadiusSearch(radius);
284     fpfh_obj.setSearchSurface(cloud);
285     fpfh_obj.setInputNormals(normals);
286     fpfh_obj.setInputCloud(keypoint);
287     pcl::PointCloud<pcl::FPFHSignature33>::Ptr descriptors(new pcl::PointCloud<
        pcl::FPFHSignature33>());
288     fpfh_obj.compute(*descriptors);
289     return descriptors;
290 }
291 }
292
293 /**
294  * @brief A method to find keypoints in a input point cloud
295  * @param inCloud The input cloud
296  * @param min_scale Minimum scale
297  * @param nr_octaves Number of octaves
298  * @param nr_scale_pr_octave Number of scale per octave
299  * @return
300  */
301 pcl::PointCloud<pcl::PointXYZ>::Ptr Computing::generate_keypoints_sift(pcl::
    PointCloud<pcl::PointXYZ>::Ptr inCloud, float min_scale, int nr_octaves,
    int nr_scale_pr_octave)
302 {
303     pcl::PointCloud<pcl::PointXYZRGB>::Ptr rgbcloud(new pcl::PointCloud<pcl::
        PointXYZRGB>);
304     pcl::copyPointCloud(*inCloud, *rgbcloud);
305     for(int i = 0; i < rgbcloud->size(); i++){
306         rgbcloud->points[i].r = 255;
307         rgbcloud->points[i].g = 255;
308         rgbcloud->points[i].b = 255;
309     }
310
311     pcl::SIFTKeypoint<pcl::PointXYZRGB, pcl::PointWithScale> sift;
312     sift.setSearchMethod(pcl::search::Search<pcl::PointXYZRGB>::Ptr (new pcl::
        search::KdTree<pcl::PointXYZRGB>));
313     sift.setScales(min_scale, nr_octaves, nr_scale_pr_octave);
314     sift.setInputCloud(rgbcloud);
315     sift.setMinimumContrast(0.0);
316     pcl::PointCloud<pcl::PointWithScale> tmp;
317     sift.compute(tmp);
318     pcl::PointCloud<pcl::PointXYZ>::Ptr keyPoints (new pcl::PointCloud<pcl::
        PointXYZ>);
319     pcl::copyPointCloud(tmp, *keyPoints);
320     return keyPoints;
321 }

```

```

322
323 /**
324  * @brief Aligns two point clouds with feature matching
325  * @param source_cloud The source cloud
326  * @param target_cloud The target cloud
327  * @param min_sample Minimum sample
328  * @param max_corr_dist Maximum distance between correspondences
329  * @return The transformation matrix
330  */
331 Eigen::Matrix4f Computing::alignment(pcl::PointCloud<pcl::PointXYZ>::Ptr
    source_cloud, pcl::PointCloud<pcl::PointXYZ>::Ptr target_cloud, int
    min_sample, int max_corr_dist)
332 {
333     // Sample consensus alignment
334     pcl::SampleConsensusInitialAlignment<pcl::PointXYZ, pcl::PointXYZ, pcl::
        FPFHSignature33> initial_alignment;
335
336     initial_alignment.setMinSampleDistance(min_sample);
337     initial_alignment.setMaxCorrespondenceDistance(max_corr_dist);
338     initial_alignment.setMaximumIterations(100);
339
340
341     //Compute keypoints
342     pcl::PointCloud<pcl::PointXYZ>::Ptr source_key = generate_keypoints_sift(
        source_cloud, 0.001, 3, 3);
343     std::cout << source_cloud->size() << std::endl;
344     std::cout << source_key->size() << std::endl;
345     // Estimate the normals.
346     pcl::PointCloud<pcl::Normal>::Ptr normals1(new pcl::PointCloud<pcl::Normal
        >);
347     pcl::NormalEstimation<pcl::PointXYZ, pcl::Normal> normalEstimation1;
348     normalEstimation1.setInputCloud(source_cloud);
349     normalEstimation1.setRadiusSearch(0.03);
350     pcl::search::KdTree<pcl::PointXYZ>::Ptr kdtree1(new pcl::search::KdTree<pcl
        ::PointXYZ>);
351     normalEstimation1.setSearchMethod(kdtree1);
352     normalEstimation1.compute(*normals1);
353     //Compute local descriptors
354     pcl::PointCloud<pcl::FPFHSignature33>::Ptr source_descriptors =
        compute_FPFH(source_cloud, normals1, source_key, 0.15);
355     initial_alignment.setInputSource(source_key);
356     initial_alignment.setSourceFeatures(source_descriptors);
357
358
359     //Compute keypoints
360     pcl::PointCloud<pcl::PointXYZ>::Ptr target_key = generate_keypoints_sift(
        target_cloud, 0.001, 3, 3);
361     std::cout << target_cloud->size() << std::endl;
362     std::cout << target_key->size() << std::endl;
363     // Estimate the normals.
364     pcl::PointCloud<pcl::Normal>::Ptr normals2(new pcl::PointCloud<pcl::Normal
        >);
365     pcl::NormalEstimation<pcl::PointXYZ, pcl::Normal> normalEstimation2;
366     normalEstimation2.setInputCloud(target_cloud);
367     normalEstimation2.setRadiusSearch(0.03);
368     pcl::search::KdTree<pcl::PointXYZ>::Ptr kdtree2(new pcl::search::KdTree<pcl
        ::PointXYZ>);

```

```

369     normalEstimation2.setSearchMethod(kdtree2);
370     normalEstimation2.compute(*normals2);
371     //Compute local descriptors
372     pcl::PointCloud<pcl::FPFHSignature33>::Ptr target_descriptors =
        compute_FPFH(target_cloud, normals2, target_key,0.15);
373
374     initial_alignment.setInputTarget(target_key);
375     initial_alignment.setTargetFeatures(target_descriptors);
376
377
378     pcl::PointCloud<pcl::PointXYZ> registration_output;
379     initial_alignment.align(registration_output);
380
381     std::cout << initial_alignment.getFinalTransformation() << std::endl;
382
383     return (initial_alignment.getFinalTransformation());
384 }
385
386 /**
387  * @brief Aligns two point cloud with ICP
388  * @param source_cloud Source cloud
389  * @param target_cloud Target cloud
390  * @return The transformation matrix
391  */
392 Eigen::Matrix4f Computing::icp_alignment(pcl::PointCloud<pcl::PointXYZ>::Ptr
    source_cloud, pcl::PointCloud<pcl::PointXYZ>::Ptr target_cloud)
393 {
394     // Compute the ICP alignment
395     pcl::PointCloud<pcl::PointXYZ>::Ptr finalCloud(new pcl::PointCloud<pcl::
        PointXYZ>);
396
397     pcl::IterativeClosestPoint<pcl::PointXYZ, pcl::PointXYZ> icp_b;
398     icp_b.setInputSource(source_cloud);
399     icp_b.setInputTarget(target_cloud);
400     icp_b.setMaximumIterations(100);
401
402     icp_b.align(*finalCloud);
403     if (icp_b.hasConverged())
404     {
405         std::cout << "ICP converged." << std::endl
406             << "The score is " << icp_b.getFitnessScore() << std::endl;
407         std::cout << "Transformation matrix:" << std::endl;
408         std::cout << icp_b.getFinalTransformation() << std::endl;
409     }
410     else std::cout << "ICP did not converge." << std::endl;
411     return icp_b.getFinalTransformation();
412 }
413
414 /**
415  * @brief Finds the differences between two point cloud using the
     SegmentDifferences method in PCL
416  * @param source_cloud Source point cloud
417  * @param target_cloud Target point cloud
418  * @param distThresh The distance threshold
419  * @return the resulting point cloud
420  */
421 pcl::PointCloud<pcl::PointXYZ>::Ptr Computing::find_differences(pcl::PointCloud

```

```

422     <pcl::PointXYZ>::Ptr source_cloud, pcl::PointCloud<pcl::PointXYZ>::Ptr
423     target_cloud, float distThresh)
424 {
425     // The output cloud
426     pcl::PointCloud<pcl::PointXYZ>::Ptr segmentedCloud(new pcl::PointCloud<pcl
427     ::PointXYZ>);
428
429     pcl::search::KdTree<pcl::PointXYZ>::Ptr tree(new pcl::search::KdTree<pcl::
430     PointXYZ>);
431     pcl::SegmentDifferences<pcl::PointXYZ> seg_diff;
432     seg_diff.setSearchMethod(tree);
433     seg_diff.setDistanceThreshold(distThresh);
434     seg_diff.setInputCloud(source_cloud);
435     seg_diff.setTargetCloud(target_cloud);
436     seg_diff.segment(*segmentedCloud);
437     return segmentedCloud;
438 }
439
440 /**
441  * @brief A statistical outlier removal filter
442  * @param inCloud The cloud to process
443  * @param meanK Global distance mean deviation
444  * @param stdDev Standard deviation of distance
445  * @return Point cloud with noise filtered
446  */
447 pcl::PointCloud<pcl::PointXYZ>::Ptr Computing::statOutlierRemoval(pcl::
448 PointCloud<pcl::PointXYZ>::Ptr inCloud, int meanK, float stdDev)
449 {
450     pcl::PointCloud<pcl::PointXYZ>::Ptr cloud_filtered (new pcl::PointCloud<pcl
451     ::PointXYZ>);
452
453     std::cout << "Cloud before filtering: " << std::endl;
454     std::cout << *inCloud << std::endl;
455
456     // Create the filtering object
457     pcl::StatisticalOutlierRemoval<pcl::PointXYZ> sor;
458     sor.setInputCloud (inCloud);
459     sor.setMeanK (meanK);
460     sor.setStddevMulThresh (stdDev);
461     sor.filter (*cloud_filtered);
462
463     //Print to see the difference in points after filtering
464     std::cout << "Cloud after filtering: " << std::endl;
465     std::cout << *cloud_filtered << std::endl;
466
467     return cloud_filtered;
468 }
469
470 /**
471  * @brief Calculates normals for a input point cloud
472  * @param inCloud The input cloud
473  * @return The normals for the input cloud
474  */
475 pcl::PointCloud<pcl::Normal>::Ptr Computing::calculateNormals(pcl::PointCloud<
476 pcl::PointXYZ>::Ptr inCloud)
477 {
478     pcl::PointCloud<pcl::Normal>::Ptr normals1(new pcl::PointCloud<pcl::Normal

```

```

    >);
472   pcl::NormalEstimation<pcl::PointXYZ, pcl::Normal> normalEstimation1;
473   normalEstimation1.setInputCloud(inCloud);
474   normalEstimation1.setRadiusSearch(0.2);
475   pcl::search::KdTree<pcl::PointXYZ>::Ptr kdtree1(new pcl::search::KdTree<pcl
        ::PointXYZ>);
476   normalEstimation1.setSearchMethod(kdtree1);
477   normalEstimation1.compute(*normals1);
478
479
480   return normals1;
481 }
482
483 /**
484  * @brief Calculate normals for a point cloud, given a viewpoint
485  * @param inCloud The input cloud
486  * @param x X-coordinate for the viewpoint
487  * @param y Y-coordinate for the viewpoint
488  * @param z Z-coordinate for the viewpoint
489  * @return The calculated normals
490  */
491 pcl::PointCloud<pcl::Normal>::Ptr Computing::calculateNormalsGivenViewPoint(pcl
    ::PointCloud<pcl::PointXYZ>::Ptr inCloud, float x, float y, float z)
492 {
493   pcl::PointCloud<pcl::Normal>::Ptr normals1(new pcl::PointCloud<pcl::Normal
        >);
494   pcl::NormalEstimation<pcl::PointXYZ, pcl::Normal> normalEstimation1;
495   normalEstimation1.setInputCloud(inCloud);
496   normalEstimation1.setRadiusSearch(0.03);
497   normalEstimation1.setViewPoint(x,y,z);
498   pcl::search::KdTree<pcl::PointXYZ>::Ptr kdtree1(new pcl::search::KdTree<pcl
        ::PointXYZ>);
499   normalEstimation1.setSearchMethod(kdtree1);
500   normalEstimation1.compute(*normals1);
501
502
503   return normals1;
504 }
505
506 /**
507  * @brief Flips the direction of the normals in the input
508  * @param inNormal The normals to flip
509  * @return The flipped normals
510  */
511 pcl::PointCloud<pcl::Normal>::Ptr Computing::flipNormals(pcl::PointCloud<pcl::
    Normal>::Ptr inNormal)
512 {
513   std::cout << "test" << std::endl;
514   pcl::PointCloud<pcl::Normal>::Ptr outNormal = inNormal;
515   for (int i=0;i<inNormal->size();i++){
516     //std::cout << "x: " << outNormal->points[i].normal_x << std::endl;
517     //std::cout << "y: " << outNormal->points[i].normal_y << std::endl;
518     //std::cout << "z: " << outNormal->points[i].normal_z << std::endl;
519     outNormal->points[i].normal_x = -1.0*outNormal->points[i].normal_x;
520     outNormal->points[i].normal_y = -1.0*outNormal->points[i].normal_y;
521     outNormal->points[i].normal_z = -1.0*outNormal->points[i].normal_z;
522     //std::cout << "x2: " << outNormal->points[i].normal_x << std::endl;

```



```

523     //std::cout << "y2: " << outNormal->points[i].normal_y << std::endl;
524     //std::cout << "z2: " << outNormal->points[i].normal_z << std::endl;
525 }
526 return outNormal;
527 }
528
529 /**
530  * @brief Calculates new points at a given distance from the input point, along
531  *        the input normal
532  * @param inCloud The input cloud
533  * @param inNormal The normals to the input cloud
534  * @param distance The distance from the old point the new point should be
535  * @return The new points, organized in a cloud
536  */
537 pcl::PointCloud<pcl::PointXYZ>::Ptr Computing::
538 calculateNewPointsFromPCandNormals(pcl::PointCloud<pcl::PointXYZ>::Ptr
539 inCloud, pcl::PointCloud<pcl::Normal>::Ptr inNormal, float distance)
540 {
541     pcl::PointCloud<pcl::PointXYZ>::Ptr outCloud (new pcl::PointCloud<pcl::
542     PointXYZ>);
543     pcl::PointCloud<pcl::Normal>::Ptr outNormal (new pcl::PointCloud<pcl::
544     Normal>);
545     *outCloud = *inCloud;
546     *outNormal = *inNormal;
547     std::cout << "test" << std::endl;
548     for (int i=0; i < inCloud->size();i++){
549         std::cout << "x: " << outCloud->points[i].x << std::endl;
550         outCloud->points[i].x = outCloud->points[i].x+outNormal->points[i].
551             normal_x*distance;
552         std::cout << "x: " << outNormal->points[i].normal_x << std::endl;
553         std::cout << distance << std::endl;
554         std::cout << "x: " << outCloud->points[i].x << std::endl;
555         outCloud->points[i].y = outCloud->points[i].y+outNormal->points[i].
556             normal_y*distance;
557         outCloud->points[i].z = outCloud->points[i].z+outNormal->points[i].
558             normal_z*distance;
559     }
560     return outCloud;
561 }

```

## Appendix C: The main class for CV

```
1  /**
2  * @file /src/main_window.cpp
3  *
4  * @brief Implementation for the qt gui.
5  *
6  * @date February 2011
7  */
8  /*****
9  ** Includes
10 *****/
11
12 #include <QtGui>
13 #include <QMessageBox>
14 #include <iostream>
15 #include "../include/hahaha/main_window.hpp"
16
17 /*****
18 ** Namespaces
19 *****/
20
21 namespace hahaha {
22
23 using namespace Qt;
24
25 //Function used in uniform sampling
26 inline double uniform_deviate(int seed)
27 {
28     double ran = seed * (1.0 / (RAND_MAX + 1.0));
29     return ran;
30 }
31
32 //Function used in uniform sampling
33 inline void randomPointTriangle(float a1, float a2, float a3, float b1, float
34     b2, float b3, float c1, float c2, float c3,
35     Eigen::Vector4f& p)
36 {
37     float r1 = static_cast<float>(uniform_deviate(rand()));
38     float r2 = static_cast<float>(uniform_deviate(rand()));
39     float r1sqr = sqrtf(r1);
40     float OneMinR1Sqr = (1 - r1sqr);
41     float OneMinR2 = (1 - r2);
42     a1 *= OneMinR1Sqr;
43     a2 *= OneMinR1Sqr;
44     a3 *= OneMinR1Sqr;
45     b1 *= OneMinR2;
46     b2 *= OneMinR2;
47     b3 *= OneMinR2;
48     c1 = r1sqr * (r2 * c1 + b1) + a1;
49     c2 = r1sqr * (r2 * c2 + b2) + a2;
```

```

49     c3 = r1sqr * (r2 * c3 + b3) + a3;
50     p[0] = c1;
51     p[1] = c2;
52     p[2] = c3;
53     p[3] = 0;
54 }
55
56 //Function used in uniform sampling
57 inline void randPSurface(vtkPolyData * polydata, std::vector<double> *
    cumulativeAreas, double totalArea, Eigen::Vector4f& p)
58 {
59     float r = static_cast<float> (uniform_deviate(rand()) * totalArea);
60
61     std::vector<double>::iterator low = std::lower_bound(cumulativeAreas->begin
        (), cumulativeAreas->end(), r);
62     vtkIdType el = vtkIdType(low - cumulativeAreas->begin());
63
64     double A[3], B[3], C[3];
65     vtkIdType npts = 0;
66     vtkIdType *ptIds = NULL;
67     polydata->GetCellPoints(el, npts, ptIds);
68     polydata->GetPoint(ptIds[0], A);
69     polydata->GetPoint(ptIds[1], B);
70     polydata->GetPoint(ptIds[2], C);
71     randomPointTriangle(float(A[0]), float(A[1]), float(A[2]),
72         float(B[0]), float(B[1]), float(B[2]),
73         float(C[0]), float(C[1]), float(C[2]), p);
74 }
75
76 //Transforms a vtkSmartPointer to a PointCloud with a given amount of points.
Code from PCL
77 void uniform_sampling(vtkSmartPointer<vtkPolyData> polydata, size_t n_samples,
    pcl::PointCloud<pcl::PointXYZ> &
78     cloud_out)
79 {
80     polydata->BuildCells();
81     vtkSmartPointer<vtkCellArray> cells = polydata->GetPolys();
82     double p1[3], p2[3], p3[3], totalArea = 0;
83     std::vector<double> cumulativeAreas(cells->GetNumberOfCells(), 0);
84     size_t i = 0;
85     vtkIdType npts = 0, *ptIds = NULL;
86     for (cells->InitTraversal(); cells->GetNextCell(npts, ptIds); i++) {
87         polydata->GetPoint(ptIds[0], p1);
88         polydata->GetPoint(ptIds[1], p2);
89         polydata->GetPoint(ptIds[2], p3);
90         totalArea += vtkTriangle::TriangleArea(p1, p2, p3);
91         cumulativeAreas[i] = totalArea;
92     }
93
94     cloud_out.points.resize(n_samples);
95     cloud_out.width = static_cast<pcl::uint32_t > (n_samples);
96     cloud_out.height = 1;
97     for (i = 0; i < n_samples; i++) {
98         Eigen::Vector4f p;
99         randPSurface(polydata, &cumulativeAreas, totalArea, p);
100         cloud_out.points[i].x = p[0];
101         cloud_out.points[i].y = p[1];

```

```

102     cloud_out.points[i].z = p[2];
103 }
104 }
105
106 *****
107 ** Implementation [MainWindow]
108 *****
109
110 MainWindow::MainWindow(int argc, char** argv, QWidget *parent)
111     : QMainWindow(parent)
112     , qnode(argc,argv)
113 {
114     qRegisterMetaType<pcl::PointCloud<pcl::PointXYZ>::Ptr >("pcl::PointCloud<
115         pcl::PointXYZ>::Ptr");
116     qRegisterMetaType<pcl::PointCloud<pcl::PointXYZRGB>::Ptr >("pcl::PointCloud
117         <pcl::PointXYZRGB>::Ptr");
118     ui.setupUi(this); // Calling this incidentally connects all ui's triggers
119         to on_...() callbacks in this class.
120
121     setWindowIcon(QIcon(":/images/icon.png"));
122     ui.tab_manager->setCurrentIndex(0); // ensure the first tab is showing
123         - qt-designer should have this already hardwired, but often loses
124         it (settings?).
125     QObject::connect(&qnode, SIGNAL(rosShutdown()), this, SLOT(close()));
126     QObject::connect(this, SIGNAL(getTopics()), &qnode, SLOT(findTopics()));
127     QObject::connect(&qnode, SIGNAL(sendTopics(QStringList)), this, SLOT(
128         updateTopics(QStringList)));
129     QObject::connect(&qnode, SIGNAL(setPointCloud(pcl::PointCloud<pcl::PointXYZ
130         >::Ptr)), this, SLOT(getPointCloud(pcl::PointCloud<pcl::PointXYZ
131         >::Ptr)));
132     QObject::connect(&qnode, SIGNAL(setPointCloudRGB(pcl::PointCloud<pcl::
133         PointXYZRGB>::Ptr)), this, SLOT(getPointCloudRGB(pcl::PointCloud<pcl::
134         PointXYZRGB>::Ptr)));
135     QObject::connect(this, SIGNAL(subscribeToPointCloud2(QString)), &qnode,
136         SLOT(subscribeToPointCloud2(QString)));
137     QObject::connect(this, SIGNAL(takePictures(int, QString, bool)), &qnode,
138         SLOT(takePicture(int, QString, bool)));
139     QObject::connect(this, SIGNAL(connect_agilus_robot()), &qnode, SLOT(
140         connect_agilus()));
141
142     *****
143     ** Auto Start
144     *****
145     qnode.init();
146     rayTraceLoader = new RayTraceLoader("Test");
147     rayTraceLoader->populateLoader();
148
149
150     w = new QVTKWidget();
151     viewer.reset(new pcl::visualization::PCLVisualizer("Viewer", false));
152     w->SetRenderWindow(viewer->getRenderWindow());
153     w->update();
154     QHBoxLayout *layout = new QHBoxLayout;
155     ui.frame_3->setLayout(layout);
156     ui.frame_3->layout()->addWidget(w);

```

```

146 }
147
148 MainWindow::~MainWindow() {}
149
150 /*****
151 ** Implementation [Slots]
152 *****/
153
154 void MainWindow::updateTopics(QStringList list)
155 {
156     ui.comboBox->clear();
157     ui.comboBox->addItem(list);
158 }
159
160 /**
161  * @brief Connects the viewer in the GUI to the chosen input from the pull down
162     menu.
163  * @param check
164  */
165 void MainWindow::on_pushButton_2_clicked(bool check)
166 {
167     if(ui.comboBox->currentText().length() != 0){
168         ui.pushButton_2->setEnabled(true);
169         Q_EMIT subscribeToPointCloud2(ui.comboBox->currentText());
170     }
171 }
172 /**
173  * @brief Refreshes the topics published in ROS
174  * @param check
175  */
176 void MainWindow::on_pushButton_clicked(bool check)
177 {
178     Q_EMIT getTopics();
179 }
180 /**
181  * @brief Saves the current point cloud shown in the viewer to file.
182  * @param check
183  */
184 void MainWindow::on_button_take_pic_clicked(bool check)
185 {
186     QString savefile = QFileDialog::getSaveFileName(this, tr("savePictureString
187         "), "/home/minions/Workspaces/PointClouds/",tr("PointCloud (*.pcd)"));
188     //QString url = ui.textbox_path->toPlainText();
189     //url.append("/");
190     //url.append(ui.fileName->text());
191     Q_EMIT takePictures(1, savefile, false);
192 }
193
194 /**
195  * @brief Sets up a connection with the robot
196  * @param check
197  */
198 void MainWindow::on_connect_robot_button_clicked(bool check)
199 {
200     Q_EMIT connect_agilus_robot();

```

```

201 }
202
203
204 /**
205  * @brief Method intended for moving the robots in the cell according to user
      input. Is not completed
206  * @param check
207  */
208 void MainWindow::on_move_robot_button_clicked(bool check)
209 {
210
211 }
212
213 /*!
214  * \brief Method linked to the make STL from PCD button. Will create a STL mesh
      from a Point Cloud
215  * \param check Standard singals/slot argument
216  */
217 void MainWindow::on_pushButton_5_clicked(bool check){
218     // Debug
219     cout << "Button clicked!" << std::endl;
220
221     //Load a PCD file
222     QString modelName;
223     pcl::PointCloud<pcl::PointXYZ>::Ptr cloud_before_filter (new pcl::
      PointCloud<pcl::PointXYZ>);
224     pcl::PointCloud<pcl::PointXYZ>::Ptr cloud (new pcl::PointCloud<pcl::
      PointXYZ>);
225     modelName = QFileDialog::getOpenFileName(this, tr("Choose MODEL cloud"), "/"
      home/", tr("PCD File (*.pcd *.PCD)"));
226     pcl::io::loadPCDFile(modelName.toStdString(), *cloud_before_filter);
227
228     //Filter out the roof from the loaded PCD file.
229     pcl::PassThrough<pcl::PointXYZ> pass;
230     pass.setInputCloud (cloud_before_filter);
231     pass.setFilterFieldName ("z");
232     pass.setFilterLimits (-10000000, 1.1);
233     //pass.setFilterLimitsNegative (true);
234     pass.filter (*cloud);
235
236     // Code from PCL used to construct the STL mesh. The standard parameters
      have been tweaked to some degree
237     // Normal estimation*
238     pcl::NormalEstimation<pcl::PointXYZ, pcl::Normal> n;
239     pcl::PointCloud<pcl::Normal>::Ptr normals (new pcl::PointCloud<pcl::Normal
      >);
240     pcl::search::KdTree<pcl::PointXYZ>::Ptr tree (new pcl::search::KdTree<pcl::
      PointXYZ>);
241     tree->setInputCloud (cloud);
242     n.setInputCloud (cloud);
243     n.setSearchMethod (tree);
244     n.setKSearch (20);
245     n.compute (*normals);
246
247     // Concatenate the XYZ and normal fields*
248     pcl::PointCloud<pcl::PointNormal>::Ptr cloud_with_normals (new pcl::
      PointCloud<pcl::PointNormal>);

```

```

249     pcl::concatenateFields (*cloud, *normals, *cloud_with_normals);
250
251     getPointCloudNormal(cloud, normals);
252
253     // Create search tree*
254     pcl::search::KdTree<pcl::PointNormal>::Ptr tree2 (new pcl::search::KdTree<
        pcl::PointNormal>);
255     tree2->setInputCloud (cloud_with_normals);
256
257     // Initialize objects
258     pcl::GreedyProjectionTriangulation<pcl::PointNormal> gp3;
259     pcl::PolygonMesh triangles;
260
261     // Set the maximum distance between connected points (maximum edge length)
262     gp3.setSearchRadius (0.1);
263
264     // Set typical values for the parameters
265     gp3.setMu (3.5);
266     gp3.setMaximumNearestNeighbors (100);
267     gp3.setMaximumSurfaceAngle(M_PI/4); // 45 degrees
268     gp3.setMinimumAngle(M_PI/18); // 10 degrees
269     gp3.setMaximumAngle(2*M_PI/3); // 120 degrees
270     gp3.setNormalConsistency(false);
271
272     // Get result
273     gp3.setInputCloud (cloud_with_normals);
274     gp3.setSearchMethod (tree2);
275     gp3.reconstruct (triangles);
276
277     pcl::io::savePolygonFileSTL("/home/minions/output2.stl",triangles);
278     pcl::io::saveVTKFile("/home/minions/output1.vtk",triangles);
279
280     //A fix for a bug in the savePolygonFileSTL method. The STL file should
        have dots and not comma as decimal mark.
281     //Load file
282     QFile file("/home/minions/output2.stl");
283     if(!file.open(QIODevice::ReadOnly)) {
284         QMessageBox::information(0, "error", file.errorString());
285     }
286
287     QTextStream in(&file);
288
289     QString text;
290
291     //Change the commas for dots.
292     while(!in.atEnd()) {
293         QString line = in.readLine();
294         line.replace(",",".");
295
296         //builds up the new file. \n is newline in unix
297         text.append(line);
298         text.append("\n");
299     }
300
301     //close the file
302     file.close();
303

```

```

304     //std::cout << text.toStdString() << std::endl;
305
306     //save the fixed file
307     QFile output_file("/home/minions/updated.stl");
308     output_file.open(QIODevice::WriteOnly);
309     QTextStream roflcopter(&output_file);
310     roflcopter << text;
311     output_file.close();
312     //debug
313     std::cout << "finished with updated file" << std::endl;
314
315 }
316
317 /**
318  * @brief Insert CAD model and ray traces it. The ray trace function is made by
319     Adam Leon Kleppe
320  * @param check
321  */
322 void MainWindow::on_pushButton_6_clicked(bool check)
323 {
324     //debug
325     cout << "Button clicked!" << std::endl;
326     //Load file
327     QString modelName;
328     pcl::PolygonMesh mesh;
329     modelName = QFileDialog::getOpenFileName(this, tr("Choose MODEL cloud"), "/"
330         home/", tr("STL File (*.stl *.STL)"));
331     pcl::io::loadPolygonFileSTL(modelName.toStdString(), mesh);
332     std::cout << "Loaded file." << std::endl;
333
334
335     //Scale the CAD model. PCL uses meters as standard measure, while STL files
336         are in millimeters
337     pcl::PointCloud<pcl::PointXYZ> temp_trans_cloud;
338     fromPCLPointCloud2(mesh.cloud, temp_trans_cloud);
339     Eigen::Matrix4f scaleCAD = Eigen::Matrix4f::Identity();
340     float scaleFactor = 0.001;
341     scaleCAD(0, 0) = scaleFactor;
342     scaleCAD(0, 1) = 0;
343     scaleCAD(1, 1) = scaleFactor;
344     scaleCAD(1, 0) = 0;
345     scaleCAD(2, 2) = scaleFactor;
346     scaleCAD(0, 3) = 0;
347     scaleCAD(1, 3) = 0;
348     scaleCAD(2, 3) = 0;
349     pcl::transformPointCloud(temp_trans_cloud, temp_trans_cloud, scaleCAD);
350     toPCLPointCloud2(temp_trans_cloud, mesh.cloud);
351
352     //Initiate the rayTraceLoader object
353     rayTraceLoader = new RayTraceLoader(mesh, "Test");
354     rayTraceLoader->setCloudResolution(200);
355     rayTraceLoader->populateLoader();
356
357     pcl::PointCloud<pcl::PointXYZ>::Ptr platePC(new pcl::PointCloud<pcl::

```



```

        PointXYZ>);
358
359 //Create a point cloud from the mesh
360 vtkSmartPointer<vtkPolyData> meshVTK2;
361 pcl::VTKUtils::convertToVTK(mesh, meshVTK2);
362 uniform_sampling(meshVTK2, 5000, *platePC);
363
364 //Visualize the point cloud made from the mesh.
365 addCADinPointCloud(platePC, "cad_1",10);
366
367 }
368
369 /**
370  * @brief Insert a point cloud in the viewer
371  * @param check
372  */
373 void MainWindow::on_insert_cloud_button_clicked(bool check)
374 {
375     //Debug
376     cout << "Button clicked!" << std::endl;
377     //Load a point cloud
378     QString modelName;
379     pcl::PointCloud<pcl::PointXYZ>::Ptr cloud_before_filter (new pcl::
        PointCloud<pcl::PointXYZ>);
380     pcl::PointCloud<pcl::PointXYZ>::Ptr cloud (new pcl::PointCloud<pcl::
        PointXYZ>);
381     modelName = QFileDialog::getOpenFileName(this,tr("Choose point cloud"),"/
        home/",tr("PCD File (*.pcd *.PCD)"));
382     pcl::io::loadPCDFile(modelName.toStdString(), *cloud_before_filter);
383
384     pcl::visualization::PCLVisualizer visTmp;
385
386     //Stand-alone ROS visualizer. Has more options to change point size etc.
387     visTmp.addPointCloud(cloud_before_filter, "base");
388
389     visTmp.spin();
390
391     //Add point cloud in viewer.
392     addCADinPointCloud(cloud_before_filter, modelName.toStdString(),100);
393 }
394
395
396 /**
397  * @brief Method for matching a point cloud to the ray traced model. Will
        segment the table and filter noise. Finds the correct cluster
398  * of the ray traced model, and the correct cluster from the point cloud after
        the cloud has been segmented. Will then do a initial alignment with SAC-IA
399  * using SIFT keypoints and FPFH descriptors. A final alignment is done using
        standard ICP. Mehtod is based on the modified pipeline.
400  * A model for ray tracing has to be saved in a default location
401  * @param check
402  */
403 void MainWindow::on_insert_cloud_segment_button_clicked(bool check)
404 {
405     //Debug
406     cout << "Button clicked!" << std::endl;
407     //Load point cloud

```

```

408     QString modelName;
409     pcl::PointCloud<pcl::PointXYZ>::Ptr cloud_before_filter (new pcl::
        PointCloud<pcl::PointXYZ>);
410     //pcl::PointCloud<pcl::PointXYZ>::Ptr cloud (new pcl::PointCloud<pcl::
        PointXYZ>);
411     modelName = QFileDialog::getOpenFileName(this, tr("Choose point cloud"), "/"
        home/", tr("PCD File (*.pcd *.PCD)"));
412     pcl::io::loadPCDFile(modelName.toStdString(), *cloud_before_filter);
413
414     pcl::PointCloud<pcl::PointXYZ>::Ptr cloud(new pcl::PointCloud<pcl::PointXYZ
        >);
415
416     // Pass through filter to get just the table
417     cloud = compute->passThroughFilter(cloud_before_filter, false);
418
419     // To see if the downsampling has worked. There is a bug in the library
        cause too small leaf size not to work
420     std::cout << "PointCloud before filtering has: " << cloud->points.size ()
        << " data points." << std::endl; /*
421     pcl::PointCloud<pcl::PointXYZ>::Ptr cloud_filtered = compute->
        downsample_voxel(cloud, 0.007f);
422     std::cout << "PointCloud after filtering has: " << cloud_filtered->points.
        size () << " data points." << std::endl; /*
423
424     // Segmentation to get rid of the table
425     cloud_filtered = compute->segmentation(cloud_filtered);
426
427     // Extract clusters of points. This then makes us able to find out which
        cluster we are interested in
428     std::vector<pcl::PointCloud<pcl::PointXYZ>::Ptr> clusters = compute->
        extract_clusters(cloud_filtered);
429
430
431     // Calculate VFH descriptors for the clusters from the point cloud
432     std::vector<pcl::PointCloud<pcl::VFHSignature308>::Ptr> descriptors =
        compute->calculateVFHDescriptors(clusters);
433     // Print the number of clusters
434     std::cout << descriptors.size() << std::endl;
435
436     // Ray trace the saved model if it has not already been ray traced.
437     std::vector<RayTraceCloud> rayTraceList = rayTraceLoader->getPointClouds(
        true);
438     std::cout << rayTraceList.size() << std::endl;
439     std::vector<pcl::PointCloud<pcl::PointXYZ>::Ptr> cad_clusters;
440
441     // Add the clusters in a vector
442     for(int i=0;i<rayTraceList.size();i++){
443         cad_clusters.push_back(rayTraceList.at(i).cloud);
444     }
445     // Calculate the VFH descriptors for clusters of the cad model
446     std::vector<pcl::PointCloud<pcl::VFHSignature308>::Ptr> cad_descriptors =
        compute->calculateVFHDescriptors(cad_clusters);
447     std::cout << cad_descriptors.size() << std::endl;
448
449     // Find the which of the clusters in the point cloud we are interested in
450     int corr;
451     corr = compute->search_for_correct_cluster(descriptors, cad_descriptors);

```

```

452
453 // Do an initial alignment with feature detection
454 Eigen::Matrix4f initAlignment = compute->alignment(cad_clusters[corr],
         cloud_filtered,0.01,1);
455 pcl::PointCloud<pcl::PointXYZ>::Ptr initModel(new pcl::PointCloud<pcl::
         PointXYZ>());
456
457 // Transform the point cloud according to the initial alignment (red color)
458 pcl::transformPointCloud(*cad_clusters[corr], *initModel, initAlignment);
459 // Visualize it
460 addCADinPointCloud(initModel, "one",0);
461
462 // Do the final transformation with ICP
463 Eigen::Matrix4f finalAlignment = compute->icp_alignment(initModel,
         cloud_filtered);
464 // Add the transformations to get it from start to finish
465 Eigen::Matrix4f totAlignment = finalAlignment*initAlignment;
466
467 std::cout << "Total alignment matrix:" << std::endl;
468 std::cout << totAlignment << std::endl;
469
470 // Transform the model according to the total transformation
471 pcl::PointCloud<pcl::PointXYZ>::Ptr finalModel(new pcl::PointCloud<pcl::
         PointXYZ>());
472 pcl::transformPointCloud(*cad_clusters[corr], *finalModel, totAlignment);
473
474 // Visualize the total transformation in blue
475 addCADinPointCloud(finalModel, "two",1);
476
477 // Visualize the original cloud without the table
478 addCADinPointCloud(cloud_filtered, "clusters",10);
479
480 }
481
482 /**
483  * @brief Method intended for calculate descriptors. Depricated
484  * @param check
485  */
486 void MainWindow::on_descriptors_button_clicked(bool check)
487 {
488     //TODO matche descriptors. lage og lagre descriptors nar load CAD.
489 }
490
491 /**
492  * @brief Method to find the difference between two point clouds. Intended for
         finding onwanted obstacles in a cloud, measured against a reference cloud.
493  * @param check
494  */
495 void MainWindow::on_segment_two_clouds_button_clicked(bool check)
496 {
497     //Debug
498     cout << "Button clicked!" << std::endl;
499     //Load the point cloud in question
500     QString modelName1;
501     pcl::PointCloud<pcl::PointXYZ>::Ptr cloud1 (new pcl::PointCloud<pcl::
         PointXYZ>);
502     pcl::PointCloud<pcl::PointXYZ>::Ptr cloud1_filtered (new pcl::PointCloud<

```

```

    pcl::PointXYZ>);
503 //pcl::PointCloud<pcl::PointXYZ>::Ptr cloud (new pcl::PointCloud<pcl::
    PointXYZ>);
504 modelName1 = QFileDialog::getOpenFileName(this, tr("Choose point cloud 1"),
    /home/", tr("PCD File (*.pcd *.PCD)"));
505 pcl::io::loadPCDFile(modelName1.toStdString(), *cloud1);
506 cloud1_filtered = compute->statOutlierRemoval(cloud1, 50, 0.7);
507
508 //Load the reference point cloud
509 QString modelName2;
510 pcl::PointCloud<pcl::PointXYZ>::Ptr cloud2 (new pcl::PointCloud<pcl::
    PointXYZ>);
511 pcl::PointCloud<pcl::PointXYZ>::Ptr cloud2_filtered (new pcl::PointCloud<
    pcl::PointXYZ>);
512 //pcl::PointCloud<pcl::PointXYZ>::Ptr cloud (new pcl::PointCloud<pcl::
    PointXYZ>);
513 modelName2 = QFileDialog::getOpenFileName(this, tr("Choose point cloud 2"),
    /home/", tr("PCD File (*.pcd *.PCD)"));
514 pcl::io::loadPCDFile(modelName2.toStdString(), *cloud2);
515 cloud2_filtered = compute->statOutlierRemoval(cloud2, 50, 0.7);
516
517 pcl::PointCloud<pcl::PointXYZ>::Ptr cloud_seg (new pcl::PointCloud<pcl::
    PointXYZ>);
518
519 //cout << "before compute!" << std::endl;
520 // computes the differences according to the SegmentDifferences method in
    PCL. 0.001 is the maximum allowed deviation
521 cloud_seg = compute->find_differences(cloud1_filtered, cloud2_filtered,
    0.001);
522
523 //cout << "After compute!" << std::endl;
524 pcl::PointCloud<pcl::PointXYZ>::Ptr cloudSegFiltered (new pcl::PointCloud<
    pcl::PointXYZ>);
525 //filter the segment-cloud
526 cloudSegFiltered = compute->statOutlierRemoval(cloud_seg, 50, 0.7);
527
528 //extract clusters from the segment-cloud
529 std::vector<pcl::PointCloud<pcl::PointXYZ>::Ptr> clusters = compute->
    extract_clusters(cloudSegFiltered);
530 for(int i=0; i<clusters.size(); i++){
531     if(clusters.at(i)->size()>500)
532         std::cout << "Found something that is not meant to be there!" << std::
            endl;
533 }
534
535
536 //visualize the first cluster in the vector.
537 addCADinPointCloud(clusters.at(0), "points", 10);
538
539 }
540
541 /**
542  * @brief This method creates the normals for a chosen cloud, then calculates
    new points a given distance from the old point along its normal
543  * This method is intended for automatic trajectory generation.
544  * @param check
545  */

```

```

546 void MainWindow::on_estimate_normals_button_clicked(bool check)
547 {
548     //Load file
549     QString modelName1;
550     pcl::PointCloud<pcl::PointXYZ>::Ptr cloud_before_filter (new pcl::
        PointCloud<pcl::PointXYZ>);
551     pcl::PointCloud<pcl::PointXYZ>::Ptr tmpCloud (new pcl::PointCloud<pcl::
        PointXYZ>);
552     pcl::PointCloud<pcl::PointXYZ>::Ptr cloud_filtered (new pcl::PointCloud<pcl
        ::PointXYZ>);
553     //pcl::PointCloud<pcl::PointXYZ>::Ptr cloud (new pcl::PointCloud<pcl::
        PointXYZ>);
554     modelName1 = QFileDialog::getOpenFileName(this, tr("Choose point cloud 1"), "
        /home/", tr("PCD File (*.pcd *.PCD)"));
555     pcl::io::loadPCDFile(modelName1.toStdString(), *cloud_before_filter);
556     //remove everything but the table
557     tmpCloud = compute->passThroughFilter(cloud_before_filter, true);
558     // downsample the cloud
559     cloud_filtered = compute->downsample_voxel(tmpCloud, 0.1);
560
561     // calculate the normals
562     pcl::PointCloud<pcl::Normal>::Ptr normals = compute->calculateNormals(
        cloud_filtered);
563
564     // calculate new point a given distance from the old point, along the
        normal
565     pcl::PointCloud<pcl::PointXYZ>::Ptr distanceCloud = compute->
        calculateNewPointsFromPCandNormals(cloud_filtered, normals, 0.03);
566
567     //Visualize the original points, the normals and the new points at a
        distance from the old points.
568     addNormalsInPointCloud(cloud_filtered, normals, "normals", 1, 0.02);
569     addCADinPointCloud(cloud_filtered, "points", 10);
570     addCADinPointCloud(distanceCloud, "traj", 0);
571
572     //Visualize the same as above, just in a stand-alone visualizer, which
        gives the opportunity to change the point size.
573     pcl::visualization::PCLVisualizer visTmp;
574     pcl::visualization::PointCloudColorHandlerCustom<pcl::PointXYZ> red (
        distanceCloud, 255, 0, 0);
575     visTmp.addPointCloud(distanceCloud, red, "distCloud");
576     visTmp.addPointCloud(cloud_filtered, "base");
577     visTmp.addPointCloudNormals<pcl::PointXYZ, pcl::Normal>(cloud_filtered,
        normals, 1, 0.02, "lulz", 0);
578     visTmp.setPointCloudRenderingProperties(pcl::visualization::
        PCL_VISUALIZER_COLOR, 0.0, 0.0, 1.0, "lulz");
579     visTmp.spin();
580
581     //save the normals.
582     pcl::io::savePCDFileASCII("/home/minions/Workspaces/PointClouds/lal.pcd", *
        normals);
583
584 }
585
586 /**
587  * @brief This method creates the normals for a chosen cad model, then
        calculates new points a given distance from the old point along its normal

```

```

588  * This method is intended for automatic trajectory generation.
589  * @param check
590  */
591 void MainWindow::on_estimate_cad_normals_button_clicked(bool check)
592 {
593     //Load CAD model
594     QString modelName;
595     pcl::PolygonMesh mesh;
596     modelName = QFileDialog::getOpenFileName(this, tr("Choose MODEL cloud"), "/"
597         home/", tr("STL File (*.stl *.STL)"));
598     pcl::io::loadPolygonFileSTL(modelName.toString(), mesh);
599     std::cout << "Loaded file." << std::endl;
600
601     //Scale the CAD model. PCL uses meters, STL is in millimeters.
602     pcl::PointCloud<pcl::PointXYZ> temp_trans_cloud;
603     fromPCLPointCloud2(mesh.cloud, temp_trans_cloud);
604     Eigen::Matrix4f scaleCAD = Eigen::Matrix4f::Identity();
605     float scaleFactor = 0.001;
606     scaleCAD(0, 0) = scaleFactor;
607     scaleCAD(0, 1) = 0;
608     scaleCAD(1, 1) = scaleFactor;
609     scaleCAD(1, 0) = 0;
610     scaleCAD(2, 2) = scaleFactor;
611     scaleCAD(0, 3) = 0;
612     scaleCAD(1, 3) = 0;
613     scaleCAD(2, 3) = 0;
614     pcl::transformPointCloud(temp_trans_cloud, temp_trans_cloud, scaleCAD);
615     toPCLPointCloud2(temp_trans_cloud, mesh.cloud);
616
617
618
619     pcl::PointCloud<pcl::PointXYZ>::Ptr platePC(new pcl::PointCloud<pcl::
        PointXYZ>);
620
621     // Create a point cloud from the mesh
622     vtkSmartPointer<vtkPolyData> meshVTK2;
623     pcl::VTKUtils::convertToVTK(mesh, meshVTK2);
624     uniform_sampling(meshVTK2, 1000000, *platePC);
625
626     //Calculate normals
627     pcl::PointCloud<pcl::Normal>::Ptr normals = compute->
        calculateNormalsGivenViewPoint(platePC, 0.1, 0.1, 0.1);
628     //Flip the normals since they all point inwards due to the viewpoint.
629     pcl::PointCloud<pcl::Normal>::Ptr invNormals = compute->flipNormals(normals
        );
630
631     //Visualize the normals
632     addNormalsInPointCloud(platePC, invNormals, "normals", 30000, 0.2);
633     //pcl::PointCloud<pcl::Normal>::Ptr invNormals = compute->flipNormals(
        normals);
634
635     //Save the normals
636     pcl::io::savePCDFFileASCII("/home/minions/Workspaces/PointClouds/lal.pcd", *
        normals);
637     //Visualize the points
638     addCADinPointCloud(platePC, "test", 0);

```

```

639
640 //Visualize the same as above, just in a stand-alone visualizer, which
        gives the opportunity to change the point size.
641 pcl::visualization::PCLVisualizer visTmp;
642 pcl::visualization::PointCloudColorHandlerCustom<pcl::PointXYZ> red (
        platePC,255,0,0);
643 visTmp.addPointCloud(platePC,red, "placePC");
644 visTmp.addPointCloudNormals<pcl::PointXYZ, pcl::Normal>(platePC,invNormals
        ,1000,0.04,"lulz",0);
645 visTmp.setPointCloudRenderingProperties(pcl::visualization::
        PCL_VISUALIZER_COLOR,0.0,0.0,1.0,"lulz");
646 visTmp.spin();
647 }
648
649
650 /**
651  * @brief Update the point cloud in the viewer.
652  * @param c the point cloud
653  */
654 void MainWindow::getPointCloud(pcl::PointCloud<pcl::PointXYZ>::Ptr c)
655 {
656
657     if(!viewer->updatePointCloud(c, "cloud")){
658         viewer->addPointCloud(c, "cloud");
659         w->update();
660     }
661     w->update();
662 }
663
664 /**
665  * @brief Add a CAD model, or another point cloud in the viewer.
666  * @param c The point cloud
667  * @param name Name of the point cloud
668  * @param color The color of the point cloud
669  */
670 void MainWindow::addCADinPointCloud(pcl::PointCloud<pcl::PointXYZ>::Ptr c, std
        ::string name, int color)
671 {
672     if(color==0){
673         pcl::visualization::PointCloudColorHandlerCustom<pcl::PointXYZ> red (c
        ,255,0,0);
674         viewer->addPointCloud(c,red,name);
675     }
676     else if(color == 1){
677         pcl::visualization::PointCloudColorHandlerCustom<pcl::PointXYZ> blue (c
        ,0,0,255);
678         viewer->addPointCloud(c,blue,name);
679     }
680     else{
681         viewer->addPointCloud(c, name);
682     }
683     w->update();
684 }
685
686 /**
687  * @brief Add normals in the viewer.
688  * @param c The point cloud

```

```

689  * @param n The normals
690  * @param name Name of the normals
691  * @param level Ratio of how many points to normals. 1 give one normal for
        every point, 10 gives one normal for every 10 points.
692  * @param scale
693  */
694  void MainWindow::addNormalsInPointCloud(pcl::PointCloud<pcl::PointXYZ>::Ptr c,
        pcl::PointCloud<pcl::Normal>::Ptr n, std::string name, int level, float
        scale)
695  {
696      //pcl::visualization::PointCloudColorHandlerCustom<pcl::PointXYZ> red (c
        ,255,0,0);
697      //viewer->addPointCloud(c,red,name);
698      viewer->addPointCloudNormals<pcl::PointXYZ, pcl::Normal>(c,n,level,scale,
        name,0);
699      viewer->setPointCloudRenderingProperties(pcl::visualization::
        PCL_VISUALIZER_COLOR,0.0,0.0,1.0,name);
700
701      w->update();
702  }
703
704  /**
705   * @brief Adds both the point cloud and the normals. Level and scale is already
        set
706   * @param c The point cloud
707   * @param n The normals
708   */
709  void MainWindow::getPointCloudNormal(pcl::PointCloud<pcl::PointXYZ>::Ptr c, pcl
        ::PointCloud<pcl::Normal>::Ptr n)
710  {
711
712      if(!viewer->updatePointCloud(c, "cloud")){
713          viewer->addPointCloud(c, "cloud");
714          viewer->addPointCloudNormals<pcl::PointXYZ, pcl::Normal>(c,n,10,0.05, "
        normals");
715          w->update();
716      }
717      //w->update();
718  }
719
720
721  /**
722   * @brief Insert a point cloud in the viewer, with colors
723   * @param c The point cloud with colors.
724   */
725  void MainWindow::getPointCloudRGB(pcl::PointCloud<pcl::PointXYZRGB>::Ptr c)
726  {
727      if(!viewer->updatePointCloud(c, "cloud")){
728          viewer->addPointCloud(c, "cloud");
729          w->update();
730      }
731      w->update();
732  }
733
734
735  /** Implementation [Slots][manually connected]
736  */

```



```
737  *****/
738
739
740
741  /***/
742  ** Implementation [Menu]
743  *****/
744
745
746
747  /***/
748  ** Implementation [Configuration]
749  *****/
750
751
752
753  void MainWindow::closeEvent(QCloseEvent *event)
754  {
755      QMainWindow::closeEvent(event);
756  }
757
758  } // namespace hahaha
```

# Appendix D: Digital Appendix

```
Simulation
├── Simulation of solution 2.pdf
├── Simulation of solution 3-4.pdf
├── Solution 2.vcm
├── Solution 3/4.vcm
Computer Vision code
├── computing.hpp
├── computing.cpp
├── main.cpp
├── main_window.hpp
├── main_window.cpp
├── qnode.hpp
├── qnode.cpp
├── ray_trace.hpp
├── ray_trace.cpp
CAD models
├── Solution 4.stl
├── Solution 5.stl
└── First prototype.stl
```

**Simulation of solution2.pdf** PDF video of simulation 2.

**Simulation of solution3-4.pdf** PDF video of simulation 3/4.

**Solution 2.vcm** The Visual Components project of solution 2.

**Solution 3-4.vcm** The Visual Components project of solution 3/4.

**computing.hpp** Header file for computing.cpp.

**computing.cpp** Main file for point cloud manipulation.

**main.cpp** Main launch file for the computer vision program.

**main\_window.hpp** Header file for main\_window.

**main\_window.cpp** Main file for GUI interactions.

**qnode.hpp** Header file for qnode.cpp.

**qnode.cpp** QNode for the GUI.

**ray\_trace.hpp** Header file for ray\_trace.cpp. Developed by Adam Leon Kleppe.

**ray\_trace.cpp** File for ray tracing CAD models. Developed by Adam Leon Kleppe.

**Solution 4.stl** STL file of assembly of solution 4.

**Solution 5.stl** STL file of assembly of solution 5.

**First prototype.stl** STL file of assembly of the first prototype that will be built.