NTNU
Norwegian University of
Science and Technology

Robotic welding of Tubes with Correction from 3D Vision and Force Control

Simen Hagen Bredvold

Master of Science in Mechanical Engineering
Submission date: June 2016
Supervisor: Olav Egeland, IPK

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

NTNU
Fakultet for ingeniørvitenskap og teknologi
Institutt for produksjons- og kvalitetsteknikk

Vår dato 2016-01-13 | Vår referanse OEG

MASTEROPPGAVE 2016

Simen Hagen Bredvold

Tittel: Robotisert sammensveising av rør ved korreksjon fra 3D kamera og kraftstyring

Tittel (engelsk): Robotic welding of tubes with correction from 3D vision and force control

Oppgavens tekst:

Sammensveising av rør inngår i stort omfang i produksjon av maritime og offshore systemer. Ved robotisert utførelse av sammensveising er det interessant å bruke to roboter til å holde rørene sammen og rotere denne sammenstillingen om røraksen mens en tredje robot brukes til å sveise sammen rørene. I denne operasjonen er det kritisk at sammenstillingen av rørene er tilstrekkelig nøyaktig også ved en viss unøyaktighet i geometrien til rørene, og at denne nøyaktigheten opprettholdes når rørene roteres om sin egen akse. I denne oppgaven skal implementering av denne operasjonen studeres slik at den kan sveise sammen rør uavhengig av dens radielle kast uten å omprogrammere robotene. 3D-syn og kraftstyring skal brukes til å korrigere for avvik i rørenes geometri. Systemet skal prøves ut i instituttets robotlaboratorium.

1. Beskriv hvordan Kinect kan brukes i 3D robotsyn.
2. Hvordan kan man bruke 3D syn til å justere sylinder posisjon og orientering.
3. Bruk robotkinematikk i Matlab til å simulerer bevegelsene robotene gjør for å sammenstille to rør.
4. Presenter en løsning for sammensveising av rør med korreksjon av rørenes posisjon og orientering basert på 3D-syn. I tillegg skal kraftstyring brukes for å begrense kontaktkrefter på grunn av unøyaktighet i styringen basert på robotsyn.
5. Prøv ut systemet i eksperimenter.

Oppgave utlevert: 2016-01-15
Innlevering: 2016-06-11
Utført ved Institutt for produksjons- og kvalitetsteknikk, NTNU.

Trondheim, 2016-01-15

Olav Egeland

Professor Olav Egeland
Faglærer

Preface

This Master's thesis is written as part of the five year Master program at the Department of Production and Quality Engineering. It was conducted during the spring semester 2016 from January to June. The Department of Production and Quality Engineering, with its Automation Department, provided for both facilities and a Master's supervisor, Olav Egeland.

After requests from the maritime industry, a pre-project concluded solutions to how one can handle and weld tubes together. Faced with the problem of tubes having an unknown run-out the tube handling was not possible to fit-up. This gave rise to the Master's thesis of using 3D-vision and force control to correct for positioning error and prevent re-programing.

Trondheim, 2016-06-10

Simen Hagen Bredvold

Simen Hagen Bredvold

Acknowledgment

I would like to thank my supervisor, Olav Egeland, for providing me with a meaningful Master's thesis and for guidance. The workshop employees have been very helpful in making the robot grippers and equipment used for this Master's thesis. Thank you for the help!

Also, I would thank the department for over the last years expanding the robotics lab, to provide the students with hands on experience working with robots.

Last, I would like to thanks the other students at the department for the academically cooperation and my girlfriend Marie Bjørnsgaard for supporting me throughout the semester.

S.HB.

Summary

The maritime industry are using steel tubes in both ship building and in the aquaculture industry. To keep labor cost down and to increase quality they want to expand their expertise in robotic welding to manufacture their products domestically. For this reason have the industry turned to NTNU and asked if their students could look into the handling and welding of tubes. This Master's thesis, on a general request from the maritime industry, have focused on how one can utilize a robot cell to handle and weld tubes with run-out together.

The approach to this thesis have been to use 3D computer vision and force control to correct for tube run-out by finding its error in position and orientation. To solve this the field of computer vision have been studied and presented, including various algorithms for data acquisition, filtering and object registration. For the later, *Random sample consensus*, *Iterative Closest Point*, *SAmple Consensus Initial Alignment* and a new method for aligning translation called *Search Method* are tested against the strict alignment precision required for welding. For safety of the robots, a Matlab application was developed to simulate the new poses generated by the alignment algorithms.

The solution was implemented in the robot cell at the *Department of Production and Quality Engineering* NTNU utilizing a Kinect 3D-camera for data acquisition and four KUKA robots for handling and welding. The new poses were obtained and given to the robots using C++, *Point Cloud Library* and the establishment of a client-to-server connection in Java which made it possible to control the robots using a remote computer. With a series of tests, each of the alignment algorithms were tested for precision and quality. The tests reviled that only the *Search Method* algorithm was good enough to align position for welding. The solution and its results led to the success of welding together tubes of different lengths and unknown run-out without re-programing the robots.

Sammendrag

Maritimindustrien bruker stålrør i både skipsbygging og i akvakulturindustrien. For å holde arbeidskostnadene nede og øke kvaliteten ønsker de å utvide deres kompetanse innenfor robotisert sveising, slik at de kan produsere deres produkter innenlands. På dette grunnlaget har industrien kontaktet NTNU og spurt om deres studenter kan utforske bruken av roboter til sveising og håndtering av rør. Denne masteroppgaven har med dette ønsket fra maritimindustrien utforsket hvordan en robotcelle kan utnyttes til å sveise sammen rør med radielt kast.

Tilnærmingen til denne oppgaven har vært å bruke 3D datasyn og kraftkontroll for korrigering av rørkast ved å finne feilen i rørets orienteringen og posisjonen. Fagområdet datasyn har derfor blitt studert og presentert i denne oppgaven. Dette inkluderer algoritmer for dataanskaffelses, datafiltrering og objektgjennkjennelse. For sistnevnte tema har «Random sample consensus», «Iterative closest point», "SAmple Consensus Initial Alignment", samt en nyutviklet metode for translasjonjustering kalt «Search Method» blitt testet mot de strenge presisjonskravene for sveising. For robotsikkerhet har en Matlab applikasjon blitt utviklet til å simulere de nye stillingen generert av justeringsalgoritmene.

Løsningen ble implementert i robotcellen hos Instituttet for Produksjons og Kvalitetsteknikk NTNU ved hjelp av et Kinect 3D-kamera for dataanskaffelse og fire KUKA-roboter for håndering og sveising. De nye stillingene ble generert og gitt til robotene ved hjelp av C++ programmering bibliotekene fra «Point Cloud Library» og ved etableringen av en klient-til-server kommukasjon i Java som gjorde det mulig styre robotene fra en ekstern datamaskin. En serie med testing ble gjennomført på hver algoritme for å utforske dens presisjon og kvalitet. Testene gjorde fast ved at bare "Search Method"-algoritmen var god nok til å justere rørets posisjon slik at sveising var mulig. Arbeidet og resultatene utført i denne masteroppgaven gjorde det mulig å sveise sammen rør av ukjent lengde og kast.

Contents

List of Tables

List of Figures

Abbreviations

OLP	=	Offline Programming
TCP	=	Tool Center Point
SRC	=	General Source code file
RSL	=	Robot Scripting Language
DH	=	Denavit-Hartenberg
PLC	=	Programmable Logic Controller
PCL	=	Point Cloud Library
Java	=	Programming Language
CAD	=	Computer-aided design
C++	=	Programming Language
SAC-IA	=	SAmple Consensus Initial Alignment
ICP	=	Iterative Closest Point
RANSAC	=	RANdom SAmple Consensus
Kinect	=	3D Sensor
MLS	=	Moving Least Squared
ToF	=	Time-Of-Flight
RGB camera	=	Camera delivers the three basic color components red, green, and blue
PCA	=	Principal Component Analysis
KR C4	=	Robot Controller for KUKA robots
FPFH	=	Fast Point Feature Histograms
SPFH	=	Simplified Point Feature Histogram
I/O	=	Input/Output communication between an information processing system
3DAutomate	=	3D factory simulation solution software
.txt	=	Text file
SDK	=	Software Development Kit from Microsoft
VTK	=	Visualization ToolKit
TCP/IP	=	End-to-End data communication
RPY-angles	=	Roll-Pitch-Yaw angles
pHRIWARE	=	physical Human-Robot Interaction Workspace Analysis, Research and Evaluation
KUKA FRAME	=	Data type containing pose of robot
MAG	=	Metal Active Gas

Chapter 1

Introduction

1.1 Background

The Norwegian aquaculture has in the period between 2005-2014 had an annual production increase of 6.5% and the government wants the aquaculture to be one of the industries to replace the oil industry in the long run [33]. The Norwegian Minister of Fisheries, Per Sandberg said on a press conference in January 2016 that he wants to speed up the technology development to maintain the growth in the industry. One of the biggest problems the industry is facing today is salmon louse which are becoming resistant to the current treatments. On of the measures to reduce louse is to locate the fish cages in rougher sea to increase the flow rate of water going through the cages, but this have been costly because the cages made of Polyethylene tubing keeps tearing. For the cages to endure the Norwegian climate and rough sea they need to be robust and rigid. This is the reason why companies like, Maritim Oppdrett AS, want to build the cages using steel tubes.

Steel tubes are used in many different industries and accounts for 8% of all global steel shipments according to the international trade center. One of the big consumer of steel tubes are the oil industry which uses them for drilling casings, tubing to carry the oil or gas to the ground surface, linepipe to transport the oil from well to the oil refinery etc. [21]. Falling oil prices the recent years have decreased the demand for steel pipes in the oil industry leading to price decrease together with the excess steel making capacity and falling raw material prices [26]. This have benefited the Norwegian shipping and aquaculture which uses steel tubes for their ships and fish cages. Instead of outsourcing the production of these cages to industries abroad they want manufacture them domestically. To be able to manufacture these metal cages in an economically sustainable way many companies are looking for the utilization of robots to weld the framework of the cages.

Problem Formulation

In order to help the maritime industry for the utilization of robotics technology and having a cost and quality efficient production, the need for research and development in the field is required. With the use of software, technology, expertise and the robot lab provided by the Department of Production and Quality Engineering at NTNU, this master thesis will discuss how one can splice two tubes together using robots, 3D-vision for position correction due to run-out in steel tubes and force control to reduce contact forces.

1.2 Objectives

The main objectives of this Master's thesis are:

1. Describe how the Kinect can be used as sensor for 3D-vision.
2. Describe how 3D-vision can be used to align cylinder position and orientation.
3. Use robot kinematics and Matlab to simulate the movements done by the two robots to hold two cylinders together.
4. Present a solution for weldment of tubes with correction of its position and orientation based on 3D-vision. Also, utilize force control to protect to robots when fitting up of tubes because of inaccuracies when using 3D-vision.
5. Test the solution in the robot lab.

1.3 Approach

In this thesis both theoretical and practical challenges have be solved. Literature about robotics and 3D-vision as well as numerous articles online have been studied to obtain the knowledge of creating a good solution.

With no experience with C++ all the exercises for the course "TDT4102 - Procedural and Object-Oriented Programming" was completed in the start of the semester to obtain the skills needed to be able to develop an 3D-vision application. The Point Cloud Library forum have been vividly used for discussion and learning about 3D perception topics.

Objective 1

General information about how the Kinect works is presented in chapter 3. In focus are the properties of how the Kinect works as a camera suitable for 3D image acquisition using Time-Of-Flight technology. The quality of captured scene is discussed by the means of noise and depth inaccuracy.

Objective 2

It is presented in the same chapter 3 the processing steps required for using 3D data from the Kinect for aligning cylindrical object. This includes how raw input data is filtered and the algorithms used for alignment. For alignment SAC-IA, ICP, RANSAC and a self composed method called Search Method are presented.

Objective 3

Section 4.5 presents the development of a safety program securing that the robots do not collide. The application was developed in Matlab using the robot kinematics from chapter 2. Forward and inverse kinematics, as well as joint space trajectory were used to simulate the robot motions.

Objective 4

Chapter 4 presentes the setup for the solution of utilizing 3D-vision, robots, force control and the safety program from objective 3 for welding two tubes together. This includes the Java application for controlling the robots, C++ application for 3D image acquisition and alignment algorithms and all the blocks used to complete the solution.

Objective 5

Using what have been studied in objective 1-4 made it possible to test how the robots in the lab could cooperate in welding two tubes together with the use of 3D-vision. In chapter 5 problems, solutions and the results of the implementation are presented.

1.4 Structure of the Report

1. Chapter 2 presents the kinematics used to simulate the robot motions. Also, general transformation matrix manipulation is covered.

2. Chapter 3 presents computer vision and how the Kinect together with point cloud processing and algorithms can be used to align cylindrical objects.

3. Chapter 4 presents the setup of the solution, software and the equipment used to weld two tubes together.

4. Chapter 5 presents the results from the final solution along with which algorithm worked best for alignment.

5. Chapter 6 summarizes the thesis with a discussion, conclusion and improvements.

Chapter 2

Robot kinematic

In this thesis a safety program have been developed to protect the robots and its environment against collisions. The collision detection uses robot kinematics to compute paths for the two handling robots KUKA 120 R2500 pro. This chapter will cover the robot kinematics used in the safety program.

2.1 Denavit-Hartenberg parameter

Denavit-Hartenberg uses four parameters to describe the pose of each link in the chain relative to the pose of the preceding link. To relate the kinematic information of the robot component, one attach a local coordinate frame to each link (i) at joint i+1 and then by following a standard method of rules the DH-parameters can be found. The four parameters needed at each link(i) are: link length a_i, link offset d_i, link twist α_i and joint angle θ_i.

2.1.1 Setting up the local coordinate frame to each link

Numbering of links starts from 0 for the immobile ground base link, to link n for the end-effector. While numbering of joints starts from 1 for the first movable link and increases up to n per joint. For the local coordinate frame to be determined, there are three rules to follow:

1. The z_{i-1} is axis of actuation of joint i.

2. Axis x_i is set so it is perpendicular to and intersects z_{i-1}.

3. Derive y_i from x_i and z_i using the right-hand rule.

The KUKA 120 R2500 pro will have a local coordinate system described in figure 2.1.

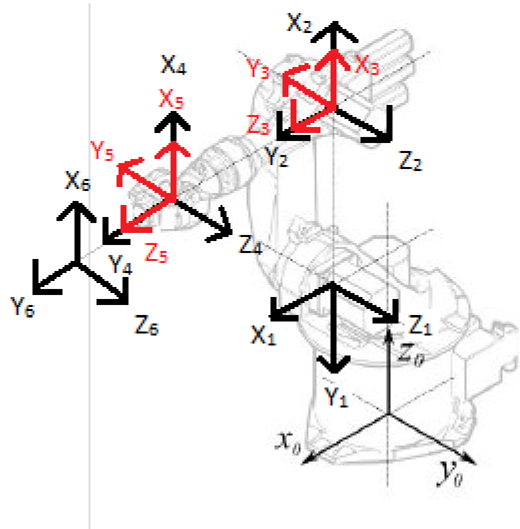

Figure 2.1: Local coordiante system for each joint for the KUKA 120 R2500 robot

2.1.2 Deriving the Denavit-Hartenberg parameters for the KUKA 120 R2500 robot

Using the chain of local coordinate system derived in subsection 2.1.1 together with the robots axis data found in figure 2.2, one can with a set of rules derive the DH- parameters. Rules of deriving the DH-parameters:

1. a_i is the distance from z_{i-1} to z_i measured along x_i
2. α_i is the angle from z_{i-1} to z_i measured about x_i
3. d_i is the distance from x_{i-1} to x_i measured along z_{i-1}
4. θ_i is the angle between x_{i-1} about z_{i-1} to become parallel to x_i

Using these rules, one obtain the DH-parameter found in table2.1.

*In joint 6 the link offset d_6 is 0.215m, but the TCP is translated 0.228m along the z_6 axis. Also, the KUKA robots have an offset of -90° in joint q_3.

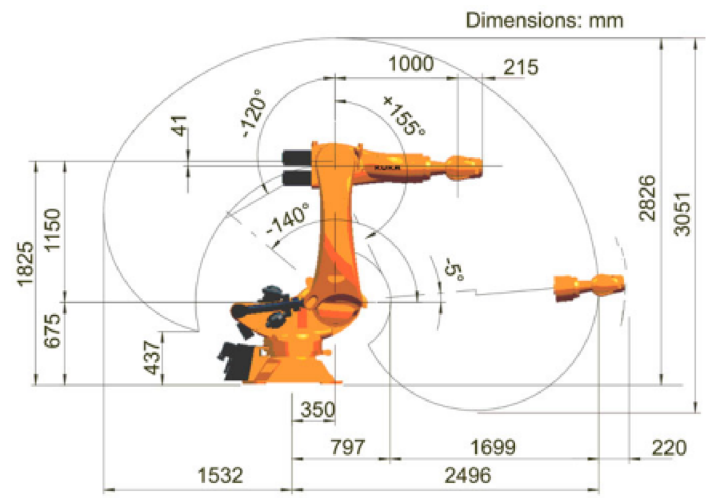

Figure 2.2: Axis data for KUKA 120 R2500. Figure taken from [1].

Table 2.1: DH- parameters for KUKA 120 R2500 pro in meter.

Joint (i)	θ	**d**	**a**	α
1	q_1	0.676	0.350	-90°
2	q_2	0	1.150	0
3	q_3-90°	0	-0.041	-90°
4	q_4	1	0	90°
5	q_5	0	0	-90°
6	q_6	0.215+0.228*	0	90°

Deriving transformation matrix using the DH-parameters

Every joint is given a local coordinate frame B_i. The necessary motion to transform from one coordinate B_i to B_{i-1} is represented as a product of four basic transformations using the DH-parameters of link (i).

1. Rotate θ_i about z_i

2. Translate along z_i a distance of d_i to make x axis of the two coorinate frames colinear.

3. Translate along z_i a distance of α_i to bring the origin together.

4. Rotate α_i about x_i

The equations are presented below, respectively:

$$Rot_{z,\theta_i} = \begin{bmatrix} \cos(\theta_i) & -\sin(\theta_i) & 0 & 0 \\ \sin(\theta_i) & \cos(\theta_i) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{2.1}$$

$$Trans_{z,d_i} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & d_i \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{2.2}$$

$$Trans_{x,\alpha_i} = \begin{bmatrix} 1 & 0 & 0 & \alpha_i \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{2.3}$$

$$Rot_{x,\alpha_i} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(\alpha_i) & -\sin(\alpha_i) & 0 \\ 0 & \sin(\alpha_i) & \cos(\alpha_i) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{2.4}$$

The product will give the transformation matrix between the two local coordinate frames.

$$^{i-1}T_i = Rot_{z,\theta_i} \cdot Trans_{z,d_i} \cdot Trans_{x,\alpha_i} \cdot Trans_{x,\alpha_i} \tag{2.5}$$

$$^{i-1}T_i = \begin{bmatrix} \cos(\theta_i) & -sin(\theta_i) & 0 & a_i \\ sin(\theta_i) \cdot cos(\alpha_i) & \cos(\theta_i) \cdot cos(\alpha_i) & -sin(\alpha_i) & -sin(\alpha_i) \cdot d_i \\ sin(\theta_i) \cdot sin(\alpha_i) & \cos(\theta_i) \cdot sin(\alpha_i) & cos(\alpha_i) & cos(\alpha_i) \cdot d_i \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{2.6}$$

2.2 Forward kinematics

Forward kinematics is the description of how one can find the coordinates (X Y Z) of the end-effector in Cartesian space relative to the base frame if the joint configuration is known.

The position and orientation of the end-effector, relative to the base frame, is described by the transformation matrix given in 2.7. Each term of the equation is taken from equation 2.6, ranging i from 1 to 6, with its suitable DH-parameters 2.1:

$$^{0}T_6 = {^{0}T_1} \cdot {^{1}T_2} \cdot {^{2}T_3} \cdot {^{3}T_4} \cdot {^{4}T_5} \cdot {^{5}T_6} \tag{2.7}$$

The rotation matrix (R_6^0) and origin (O_6^0) to the end-effector from base is derived from the transformation matrix 2.7:

$$^{0}T_6 = \begin{bmatrix} R_6^0 & O_6^0 \\ 0 & 1 \end{bmatrix} \tag{2.8}$$

$$O_6^0(q_i) = \begin{pmatrix} O_x \\ O_y \\ O_z \end{pmatrix} \tag{2.9}$$

From joint space using equation 2.7 one can obtain where in Cartesian space the end-effector is by using 2.9.

2.2.1 The Jacobian matrix

The Jacobian is a square matrix consisting of first-order partial derivatives of a vector-valued function which among others connects the joint velocity to the end-effector velocity.

$$\begin{bmatrix} V \\ \omega \end{bmatrix} = J \cdot \dot{q} \tag{2.10}$$

For the kinematics in the thesis, only the linear velocity (V) with all revolute joints is of interest.

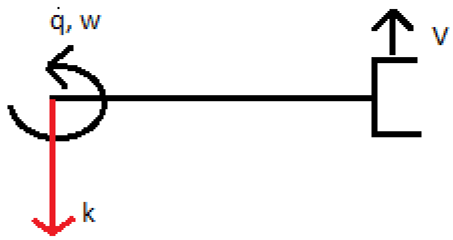

Figure 2.3: Velocity in a single revolute joint

Finding the Jacobian matrix

The velocity of the end-effector for an 6 linked manipulator is simply $\dot{O}_6^0$. By using the chain rule:

$$\dot{O}_6^0 = \sum_{i=1}^{6} \frac{\partial O_6^0}{\partial q_i} \dot{q}_i \tag{2.11}$$

Equation 2.11 is actually just another way of writing 2.10, so it is trivial to see that the *i*th column in the Jacobian matrix can be denoted as:

$$J_{v_i} = \frac{\partial O_6^0}{\partial q_i} \tag{2.12}$$

If the manipulator only consist of revolute joint, then equation 2.12 equals:

$$J_{v_i} = z_{i-1} \times (O_n - O_{i-1}) \tag{2.13}$$

Instead of proving the calculations from equation 2.12 to 2.13, it is easier to illustrates a second interpretation of 2.13.

Velocity in a single revolute joint:

$$\begin{aligned} \omega &= \dot{q} k \\ V &= \dot{q} k \times r \end{aligned} \tag{2.14}$$

Where k is the unit vector in z-direction (axis of actuation) and r is the vector between two local coordinates frames 2.3.

For the motion of the end-effector due to link i, see figure 2.4. The equation 2.13 is described as:

$$
\begin{aligned}
& r = O_n - O_{i-1} \\
& \omega = z_{i-1} \\
& J_{v_i} = \omega \times r \\
& \Rightarrow J_{v_i} = z_{i-1} \times O_n - O_{i-1}
\end{aligned}
\tag{2.15}
$$

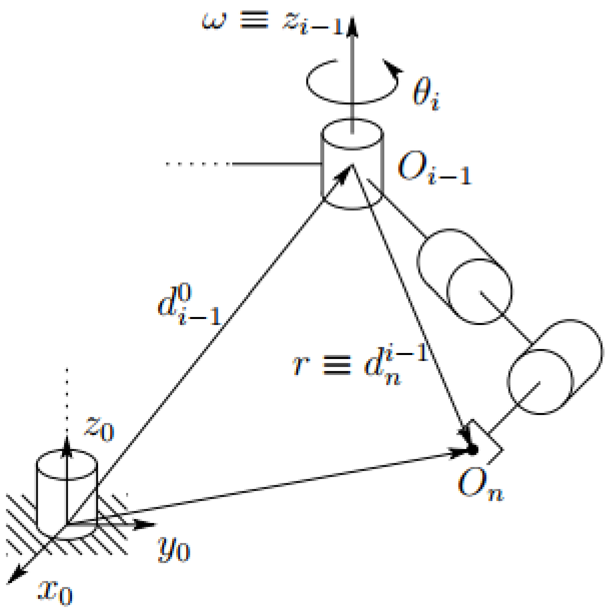

Figure 2.4: Velocity of end-effector due to link i

Where z_{i-1} is the three first elements in column three in ${}^0T_i$:

$$
\begin{aligned}
& z_0 = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} \\
& z_{1\to5} = {}^0T_{1\to5} \begin{bmatrix} \circ & \circ & z_{x_{1\to5}} & \circ \\ \circ & \circ & z_{y_{1\to5}} & \circ \\ \circ & \circ & z_{z_{1\to5}} & \circ \\ \circ & \circ & \circ & \circ \end{bmatrix}
\end{aligned}
\tag{2.16}
$$

O_n, in this case O_6, equals the three first elements in column four in the transformation matrix ${}^0T_6$. While the O_{i-1} equals:

$$O_0 = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$$
$$O_{1\to5} = {}^0T_{1\to5} \begin{bmatrix} \circ & \circ & \circ & O_{x_{1\to5}} \\ \circ & \circ & \circ & O_{y_{1\to5}} \\ \circ & \circ & \circ & O_{z_{1\to5}} \\ \circ & \circ & \circ & \circ \end{bmatrix} \tag{2.17}$$

Inverse Jacobian matrix

To compute the joint velocities for a given tool point velocity, one need to invert the Jacobian.

$$\dot{q} = J^{-1} \cdot \begin{bmatrix} V \\ \omega \end{bmatrix} \tag{2.18}$$

When taking the inverse of a matrix, one obtain a determinant which each element of the inverted matrix is divided by. If the determinant approaches zero, the inverse matrix approaches infinite. This is called a singularity and it occurs when two axes of revolute joints become parallel. Configurations that makes the determinant go to zero should be avoided.

2.3 Inverse Kinematics

In the opposite of forward kinematics, the inverse kinematics describes how to map the joint space from cartesian space. There are numerous approaches to finding the joint space if the end-effector coordinates are known. The one described in this thesis is called the *Newton-Raphson method* [24]. The method is based on searching for the joint configuration that gives the least error/residue between the wanted transformation matrix and the calculated one. The algorithm for finding the joint configuration:

1. Guess an initial joint configuration, q_k.

2. Using forward kinematics, determine the transformation matrix of the end-effector frame for the guessed joint, $T_k(q_k)$.

3. From $T_k(q_k)$, derive the rotation matrix $R_k(q_k)$ and (R_d) from the desired transformation matrix T_d.

4. Find the deviation rotation matrix $(\tilde{R})$ between the current rotation matrix $R_k(q_k)$ and the desired rotation matrix $R_d(q)$ using the definition 2.19.

$$\begin{aligned} \tilde{R} &= \{\tilde{r}_{ij}\} \\ \tilde{R} &= R_k R_d^T \end{aligned} \tag{2.19}$$

5. Find the Euler rotation vector $\tilde{e}$ corresponding to the deviation $\tilde{R}$ using 2.20

$$\tilde{e} = \frac{1}{2}\begin{pmatrix} \tilde{r}_{32} - \tilde{r}_{23} \\ \tilde{r}_{13} - \tilde{r}_{31} \\ \tilde{r}_{21} - \tilde{r}_{12} \end{pmatrix} \tag{2.20}$$

6. Find the position error $\tilde{e}_p$.

$$\begin{aligned} T_d &= \{d_{ij}\} \\ T_k &= \{k_{ij}\} \\ \tilde{e}_p &= \begin{pmatrix} d_{14} - k_{k14} \\ d_{24} - k_{24} \\ d_{34} - k_{34} \end{pmatrix} \end{aligned} \tag{2.21}$$

7. Use the inverse Jacobian matrix for the current configuration to find the joint change done to get closer to T_d.

$$e = \begin{pmatrix} d_{14} - k_{k14} \\ d_{24} - k_{24} \\ d_{34} - k_{34} \\ \tilde{r}_{32} - \tilde{r}_{23} \\ \tilde{r}_{13} - \tilde{r}_{31} \\ \tilde{r}_{21} - \tilde{r}_{12} \end{pmatrix} \tag{2.22}$$

$$\partial q = (J_k)^{-1} \cdot e$$

8. Set the new joint configuration to be:

$$q_k = q_k + \partial q \tag{2.23}$$

9. Begin at step two with the new joint configuration q_k until ∂q goes to zero.

Also, the inverse kinematics should check if the wanted joint configuration is within the range of motion for each joint. For the KUKA KR 120 R2500 pro the range of motion is given in table 2.2.

Table 2.2: Range of motion for each joint

Axis	Range of motion
1	+/- 185°
2	-5° to -140°
3	+155° to -120°
4	+/-350°
5	+/-125°
6	+/-350°

2.4 Joint Space Trajectory

Given a starting and ending joint configuration, obtaining the intermediate joint configuration where time t is assumed to vary from 0 to 1 in m steps with separate joint space trajectory for each joint. Using an additional set of constraint and a quintic polynomial it is possible to fully determine the quintic trajectory space curve. The additional constraint are the starting and ending joint velocity and acceleration which gives the six constraints fitted with the 5th order quintic polynomial to obtain a smooth trajectory $q(t)$ between the m via points. The quintic polynomial for position $q(t)$, velocity $\dot{q}(t)$ and acceleration $\ddot{q}(t)$ are given in equation 2.24 with the constraints q_0, q_1, v_0, v_1, a_0 and a_1.

$$\begin{aligned} q(t) &= at^5 + bt^4 + ct^3 + dt^2 + et + f \\ \dot{q}(t) &= 5at^4 + 4bt^3 + 3ct^2 + 2dt + e \\ \ddot{q}(t) &= 20at^3 + 12bt^2 + 6ct + 2d \end{aligned} \tag{2.24}$$

The variables are obtained by equation 2.25.

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 5 & 4 & 3 & 2 & 1 & 0 \\ 0 & 0 & 0 & 2 & 0 & 0 \\ 20 & 12 & 6 & 2 & 0 & 0 \end{bmatrix} \begin{bmatrix} a \\ b \\ c \\ d \\ e \\ f \end{bmatrix} = \begin{bmatrix} q_0 \\ q_1 \\ v_0 \\ v_1 \\ a_0 \\ a_1 \end{bmatrix} \tag{2.25}$$

2.5 General Transformation

Consider two coordinate systems, in this case the camera C(X,Y,Z) and robot 0(x,y,z), which are employed to express the components of a vector r. There is always a transformation matrix T_C^O to map the components of r from the camera reference frame to the robot reference frame.

$$O_r = T_C^O C_r \tag{2.26}$$

The unit vectors of C(X,Y,Z) along the axes of O(x,y,z) introduces the rotation matrix R_C^O to map the camera frame to the robot frame. Each row of R_C^O is decomposition of a unit vector of the camera frame in the local robot frame. The translation in T_C^O is the distance the reference frame C have been translated with respect to O. Figure 2.5 graphically presents how r can be expressed in the O frame using the camera frame.

$$R_C^O = \begin{bmatrix} - & \vec{r}_x & - \\ - & \vec{r}_y & - \\ - & \vec{r}_z & - \end{bmatrix} \tag{2.27}$$

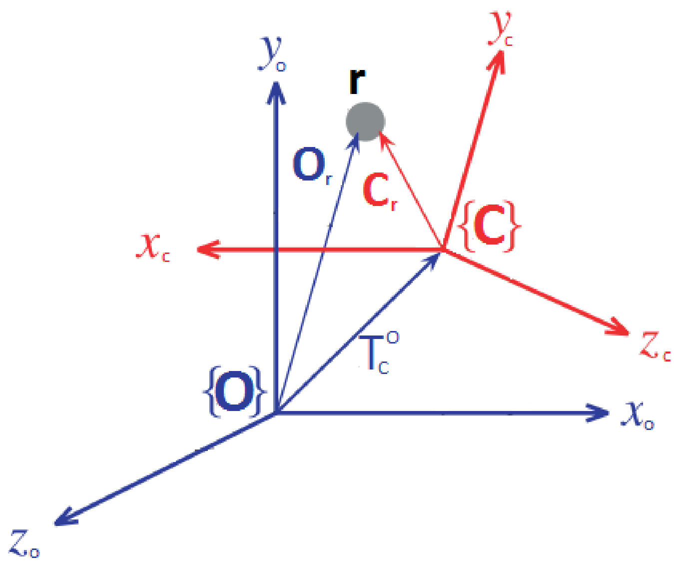

Figure 2.5: Two 3D coordinate frames O and C. C is rotated and translated with respect to O.

2.6 Roll, pitch, yaw-angles from transformation matrix

KUKA robots uses Z-Y-X Tait–Bryan angles (A,B,C), which is exactly the same as the often so called roll-pitch-yaw (RPY) convention. A,B and C are the rotation about the Z,Y and X axis, respectively. From a transformation matrix T, obtain the Tait–Bryan angles needed to assign the angles of rotation in Cartesian coordinates to the robot. These are found from the rotation matrix defined in 2.28 and the angles are denoted in equation 2.29,2.30 and 2.31.

$$T = \begin{bmatrix} R & t \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} r_{11} & r_{12} & r_{13} & x \\ r_{21} & r_{22} & r_{23} & y \\ r_{31} & r_{32} & r_{33} & z \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{2.28}$$

$$A = atan2(r_{21}, r_{11}) \tag{2.29}$$

$$B = atan2(-r_{31}, \sqrt{{r_{32}}^2 + {r_{33}}^2}) \tag{2.30}$$

$$C = atan2(r_{32}, r_{33}) \tag{2.31}$$

Chapter 3

Computer Vision

3.1 Introduction

Computer vision is a discipline of image understanding of a 3D scene from its 2D images using the characteristics of the structures present in the scene. The goal of computer vision is to go beyond the capabilities of human vision to model, replicate and analyse features of a scene using computer vision algorithms and their software implementation. Doing this, computer vision can be utilized in a wide range of applications areas as medicine, automation, security, entertainment industry and for this thesis in robotics. Implementation of computer vision in the field of robotics gives the rise to vision-based control of robots. Making the robots able to see have improved industrial robotic systems to the level for them to be used in applications as obstacles avoidance, assemble, visual serving, human robot interaction, safety, inspection etc.

While 2D imaging is most commonly used in machine vision, the use of 3D vision have its benefits in robot vision because robots work in a three dimensional world. 3D imaging allows a robot to sense variations in its physical environment and adapt accordingly, increasing flexibility, utility and velocity. Using 2D imaging for finding the pose of an object can be done with a series of assumptions, but if these assumptions are not obtained there will be a miscalculation of where the part is in space. This occurs if the objects size changes, if the object is moved closer to the camera or tilted differently, giving the robotic system little flexibility to changes and errors [10]. Hence, 3D imaging is chosen for this thesis because the objects used have a unknown run-out and length.

Existing sensor technologies for 3D image acquisition are, but not limited to, stereo vision, stereo vision using structured light, laser profiler and time of flight sensors [29]. Microsoft has created an advanced sensor input device called Kinect for its Xbox video gaming console which uses a time-of-flight (ToF) camera and a RGB camera for 3D image acquisition. The low cost of the Kinect, together with the open-source "Point Cloud Library" providing algorithms for data processing and manipulation which have made the Kinect vastly used by amateurs and professionals in robotics applications. For these reasons, the author of this thesis have chosen the technology of time-of-flight found in the new Kinect device to be the best option for correction of offline-programed robot poses for the two handling robots.

3.2 Kinect

As the Kinect was introduced to the marked in November 2010 it became the fastest selling consumer electronics device ever, selling 8 million units in its 60 first days [27]. Later on, in July 2011, Microsoft released a software development kit for Windows and just before in May 2010 the first version of the Point Cloud Library was released. This made the Kinect suitable for educational and industrial purposes, epically in the field of robotics.

The current version Kinect v2, hereinafter referred to as just Kinect, is using time of flight sensors for its 3D image acquisition, capturing depths from 0.5m - 4.6m. It has a depth image resolution of 512 x 424 pixels, meaning that it can create a 3D picture of the scene with 217088 points. Each point is represented in a coordinate system defined by the Kinect as in figure 3.1.

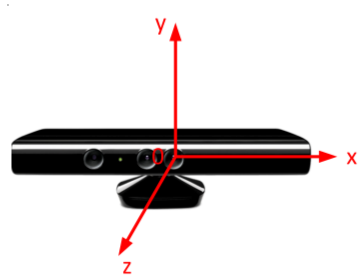

Figure 3.1: The coordinate system for the Kinect.

3.2.1 Time Of Flight - Distance measurement

A Time Of Flight camera works by translating the phase delay between the emitted signal and the reflected signal from the scene into distance. The phase delay can be measured using either a light source which is pulsed or modulated by a continuous-wave. All the commercial ToF cameras today has adopted the technology of illumination of modulated continuous waves. The scene is illuminated by a infra-red signal $S_e(t)$ of amplitude A_e and modulated by a sinusoid of frequency $f_{modulated}$, equation 3.1 where t is time. Figure 3.2 illustrates how the infra-red signal is emitted and reflected by the scene [8].

$$S_e(t) = A_e[1 + sin(2\Pi ft) \tag{3.1}$$

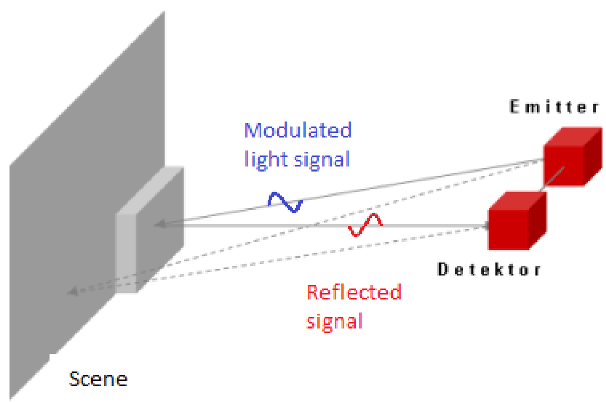

Figure 3.2: Illustration of how the modulated light is emittet from a source and reflected by the scene.

The infra-red signal is reflected by the scene and is registered by a receiver positioned close to the emitter. The received signal has an attenuated amplitude A_r because of energy absorption associated with the reflection. The phase delay is denoted $\Delta\phi$ and B_r is the ambient light, giving the equation of interest 3.2. The emittetd and reflected signal are shown in figure 3.3.

$$S_r(t) = A_r[1 + sin(2\Pi ft + \Delta\phi)] + B_r \tag{3.2}$$

The unknown variables A_r and B_r are measured in volt, while the phase delay is just a number. The phase delay can be expressed by equation 3.3 and the distance is found by solving the same equation with respect to distance ρ resulting in equation 3.10 where c is the speed of light.

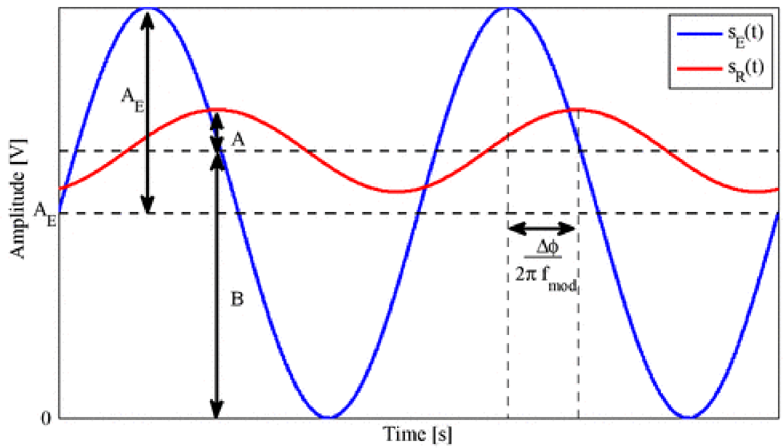

Figure 3.3: Emitted and reflected signal for Time of Flight technology. Figure from [8]

$$\Delta\phi = 2\Pi f \frac{2\rho}{c} \tag{3.3}$$

$$\rho = \frac{c}{4\Pi f}\Delta\phi \tag{3.4}$$

The received signal is sampled four times per period of the modulating signal, meaning that the sample frequency is four times $f_{modulated}$. The three unknown values are estimated using the four sampled values S_r^n in equation 3.5 and the algebraic manipulation from 3.5 to 3.6, 3.7 and 3.8 is described in [6] and [23].

$$(A_r, B_r, \Delta\phi) = arg \min_{A_r, B_r, \Delta\phi} \sum_{n=0}^{3} \{S_r^n - [A_r sin(\frac{\pi}{2} n) + \Delta\phi + B_r]\}^2 \tag{3.5}$$

$$A_r = \frac{\sqrt{(S_r^0 - S_r^2)^2 + (S_r^1 - S_r^3)^2}}{2} \tag{3.6}$$

$$B_r = \frac{S_r^0 + S_r^1 + S_r^2 + S_r^3}{4} \tag{3.7}$$

$$\Delta\phi = \arctan 2(S_r^0 - S_r^2, S_r^1 - S_r^3) \tag{3.8}$$

The final distance is obtained combining 3.4 and 3.8.

The above explanation with one emitter and one receiver only describes how to capture one point of the scene. To capture all the points in a scene, the ToF technology use a Matricial ToF camera. The Kinect is such a camera and these cameras uses several emitters providing an irradiation that is reflected back by the scene and collected by a multitude of receivers close to each other. The receiver, also called the camera sensor, consist of a CCD/CMOS lock-in pixels matrix that converts the received amount of light into a corresponding number of electrons. The stronger the light signal exposed to a pixel, the larger amount of electrons are generated. The amount of electrons is then converted into binary numbers, using A/D- conversion to measure the voltage from each pixel.

3.2.2 Time Of Flight - Noise

In practice there are several noise generating factors which influence the accuracy of the measured distance that must be taken into account. Both the generation of sinusoid frequency waves and the sampling of it are not ideal. Each of the four samples are done over a finite time interval, generating a harmonic distortion when estimating the phase delay. This again influences the accuracy of the measured distance.

Further, photon-shot noise is a phenomena caused by the nature of how light act [8]. If you measure the collection of photons from an unvarying source for a set of time the amount of photons will fluctuate around a mean value. This noise probability density function can be approximated by a Gaussian standard deviation equation 3.9 .

$$\sigma_p = \frac{c}{4\pi f_{\mathrm{mod}\,ulated}\sqrt{2}} \frac{\sqrt{B_r}}{A_r} \tag{3.9}$$

The standard deviation clearly indicates that if the modulation frequency $f_{modulated}$ increase the deviation will decrease, hence better accuracy. Furthermore, if the amplitude A_r of the reflected signal increases then the accuracy will do the same. The amplitude can vary due to inconsistencies at surfaces with low infrared-light reflectivity or the emitted signal waves are attenuated and scattered in the scene. Last, if the offset B_r is decreased by the means of increasing the interference by other sources of near-infrared light such as sunlight or other ToF cameras, the resulting distance would be less accuracy.

Saturation of the quantity of photons that the CCD/CMOS can collect is another noise generating problem. This happens if the camera is exposed to external IR illumination or reflection from highly reflective objects like a mirror.

Finally, the last type of noise this thesis will cover is the phenomena of motion blur. Just as for a standard camera, if the scene is in movement the result will be erroneous. The error is caused because of lower frame rates, so the scene when using ToF cameras should stand perfectly still.

3.2.3 ToF - Noise in practicality

The article *Evaluating and Improving the Depth Accuracy of Kinect for Windows v2* [31] preformed at the University of Ottawa evaluate properties for the Kinect as depth accuracy and depth resolution. To determine where the tubes in this project will be placed some of the relevant results from the article will be presented here.

The depth accuracy was mapped by evaluating the true distance with the mean distance of a planar surface measured by the Kinect. The results presented in figure 3.5 shows the accuracy error distribution for a planar surface at 40 key points in the horizontal and vertical plane. The results indicates that in the space between 0.5m -3.0m in Z-direction the accuracy is less than 2mm if kept inside the boundaries represented in figure 3.5.

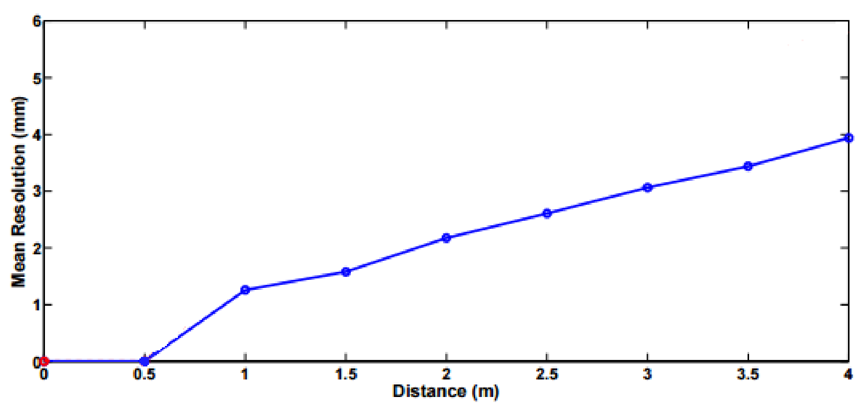

Figure 3.4: Distance between two adjacent pixels at different Z-values from the Kinect. Fig from [31].

Together with the best accuracy obtainable, it is also desirable to have the highest resolution to get the best results possible. The further away an object is located from the Kinect, the fewer points represents that object because of declining resolution. The results represented in figure

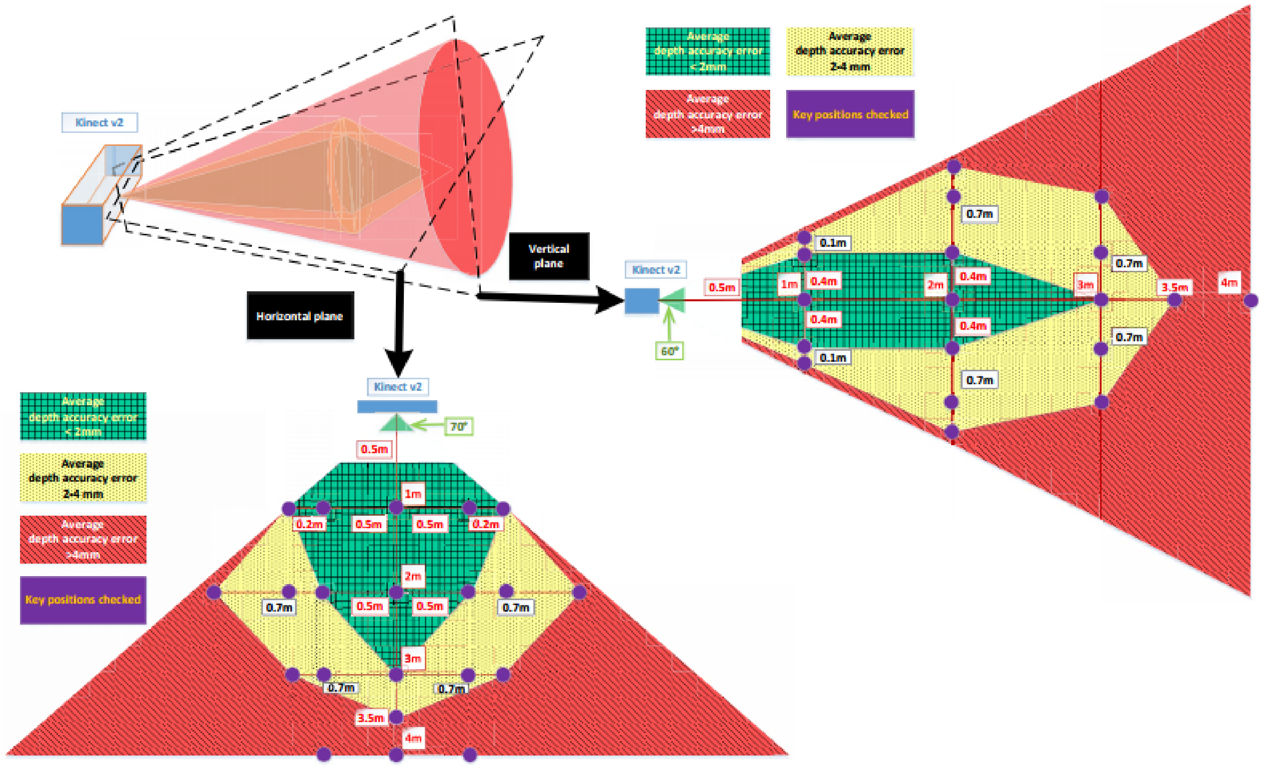

Figure 3.5: Accuracy error distribution of Kinect for Windows v2. Fig from [31]

3.4 was obtain by measuring the distance between two adjacent pixels at different distances from the Kinect. The further away from the Kinect, as expected the bigger the distance between two pixels. Knowing this the edge of the tubes were located at a distance of 0.6m in front of the camera.

3.2.4 Mapping coordinates from Kinect to robot

Using the general transformation described in section 2.5 the transformation matrix mapping coordinates from the camera local frame to the robot global frame was obtained by the method described below.

An object was attached to the end-effector of the robot and the position of the object was recorded both in the robot and camera frame. The object was located in the camera field of view and its position was recorded as the origin in both the camera frame and robot frame. The object was then translated by jogging the robot along Y and Z in the robot frame while recording the position of the object in both frames. The values are presented in table 3.1. With the positions presented in the table one can define a vector going from the origin two each point along Y and Z by equation 3.10 where O is the origin and P the position along one of the axes and the corresponding unit vector is defined by equation 3.11.

$$\vec{OP} = (x - x_0, y - y_0, z - z_0) \tag{3.10}$$

$$\hat{OP} = \frac{\vec{OP}}{\|\vec{OP}\|} \tag{3.11}$$

The unit vector for Y and Z for both robots in the camera frame are now defined, but the unit vector for X is still undefined. The unit vector representing the X-axis ($\hat{OX}$) is found by taking the cross product between unit vector $\hat{OY}$ and $\hat{OZ}$ shown in equation 3.12. To make the defined coordinate system accurate $\hat{OX}$ is crossed with $\hat{OY}$ to define a new $O\hat{Z}_{new}$ as in equation 3.13.

$$\hat{OX} = \hat{OY} \times \hat{OZ} \tag{3.12}$$

$$O\hat{Z}_{new} = \hat{OX} \times \hat{OY} \tag{3.13}$$

The unit vectors in the camera frame along the axis of the robot are used to derive the rotation matrix between the two frames. Equation 2.27 from section 2.5 explains how each of the three unit vectors are used to define the rows in the rotation matrix between camera and robot R_C and is presented in equation 3.14.

$$R_C = \begin{bmatrix} - & \hat{OX} & - \\ - & \hat{OY} & - \\ - & O\hat{Z}_{new} & - \end{bmatrix} \tag{3.14}$$

Once the rotation matrix is obtained the translation between the two frames can be computed. Given a coordinate vector r described in both the camera frame C_r and the robot frame O_r the translation between the two frames are obtained by equation 3.15.

Table 3.1: Positions recorded in the camera frame and the two robot frames to obtain the transformation matrix maping between them.

	Right robot			Left robot			Camera right robot			Camera left robot		
	x	y	z	x	y	z	x	y	z	x	y	z
Origo	1193.75	1223.59	1353.08	1235.02	-1419.58	1422.66	-222.22	-49.41	811.01	124.27	10.13	842.59
Point along y	1193.75	1383.15	1353.06	1235.02	-1556.57	1422.66	60.65	-55.51	822.86	-15.06	12.70	836.36
Point Along z	1193.75	1223.59	1545.68	1235.02	-1419.58	1591.46	-217.91	144.37	808.76	126.72	180.57	838.77

$$t = \begin{pmatrix} O_r(x) - R_C^O C_r(x) \\ O_r(y) - R_C^O C_r(y) \\ O_r(z) - R_C^O C_r(z) \end{pmatrix} \tag{3.15}$$

This method was executed for each robot with the values in table 3.1 giving the transformation matrices T_C^{right} and T_C^{left} in equation 3.16 and 3.17. The coordinate frames relative to each other is presented in figure 3.6.

$$T_c^{right} = \begin{bmatrix} -0.0416 & 0.0125 & 0.9991 & 0.3749 \\ 0.9989 & -0.0215 & 0.0418 & 1.4106 \\ 0.0222 & 0.9997 & -0.0116 & 1.4168 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{3.16}$$

$$T_c^{left} = \begin{bmatrix} -0.0443 & 0.0230 & 0.9987 & 0.3988 \\ 0.9988 & -0.0184 & 0.0447 & -1.5812 \\ 0.0144 & 0.996 & -0.0224 & 1.4296 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{3.17}$$

3.3 Point Cloud Processing

The depth image captured by the Kinect consist of 217088 points each presented by X,Y and Z values creating a depth frame, called Point Cloud. For application purposes raw data from the Kinect is not of much use alone, but with Point Cloud Processing one can with a toolbox of algorithms transform raw and noisy data into useful information. A overview of the processes of reducing data size and transforming a point cloud to useful data is presented in table 3.2.

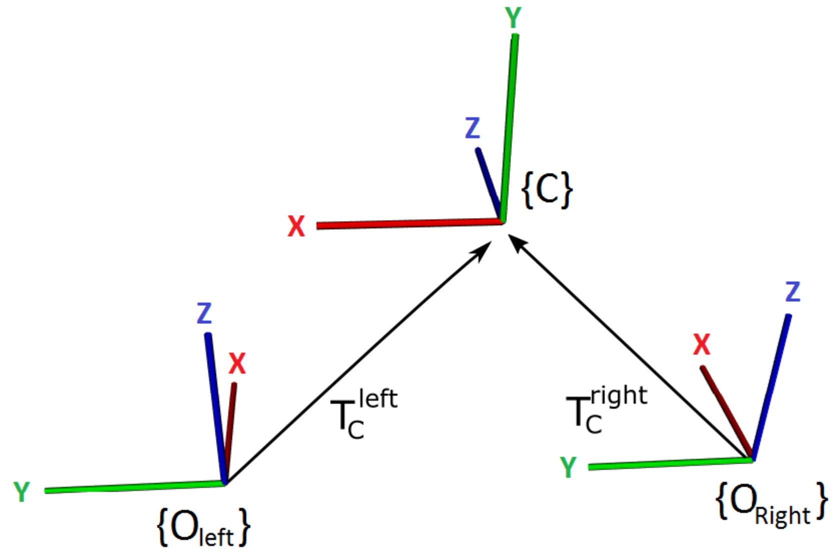

Figure 3.6: The robot frames $\{O_{right}\}$ and $\{O_{left}\}$ and the camera frame $\{C\}$.

Table 3.2: Report mapping of the Point Cloud Processes.

Subsection	**Point Cloud Process**
3.3.1	Passthrough filter
3.3.2	Normal estimation
3.3.3	Voxel Grid Down Sampling
3.3.4	Random Sample Consensus
3.3.5	Smoothing - Moving Least Squares

3.3.1 Passthrough filter

By define a volume of interest by setting minimum and maximum values on each axis an loop runs through all the data points in the point cloud deleting all the points not satisfying the boundary conditions. To increase the processing time for later algorithms this method can greatly reduce the number of data points depending on the volume of the boundaries.

3.3.2 Normal estimation

Many algorithms used in point cloud processing needs the normal estimation of the surface for computation. Given a geometric surface, the normal for a certain point is the vector being perpendicular to the surface in that point. For computing a normal for a query point the neighboring points are used to describe the local surface feature. A popular method of estimating surface normals is called *principal component analysis* (PCA) developed by Hoppe, H [17] in 1992.

For each point P_i a covariance matrix denoted C is analyzed for its eigenvectors and eigenvalues. A point P_i uses its k-nearest neighbours to compute the covariance matrix as in equation 3.18. $\bar{p}$ is the 3D centroid from equation 3.22 of the k neighbours. The eigenvalues λ_j and eigenvectors $\vec{V}_j$ are computed analytically by equation 3.19. If two eigenvalues are close together and one is significantly smaller, then the eigenvectors for the first two will define a plane and the eigenvector with the smallest eigenvalue determines the normal to this plane. The plane allocated with a point on a cylinder and its normal vector are shown in figure 3.7.

$$C = \frac{1}{k}\sum_{i=1}^{k}(p_i - \bar{p})(p_i - \bar{p})^T \tag{3.18}$$

$$C \cdot \vec{V}_j = \lambda_j \cdot \vec{V}_j, j \in \{0,1,2\} \tag{3.19}$$

The PCA method makes the normal orientation ambiguous, either pointing inwards or outwards of the surface. This problem is solved by knowing the viewpoint V_p and that every normal vector $\vec{n}_i$ have to satisfy the equation given in 3.20. Figure 3.29 shows how a plane is fitted to a point representing the surface of a cylinder and the normal to this plane is the normal for the cylinder at that point.

$$\vec{n}_i \cdot (v_p - p_i) \tag{3.20}$$

3.3.3 Voxel Grid Down Sampling

A voxel is a volume element, while a voxel grid is the composition of several voxels creating a grid covering the entire scene. The volume of a voxel is defined by the leaf size which is a distance measure in x,y and z. Every point representing the scene will be contained by a voxel and the number of data points lying inside the boundary condition of that specific voxel will be reduced by the means of being represented by the voxel centroid. The centroid of voxel i is defined by equation 3.21 and the arithmetic mean values X,Y and Z representing the N points inside the voxel are found using 3.22. Figure 3.8 shows how the colored points constrained by a voxel are

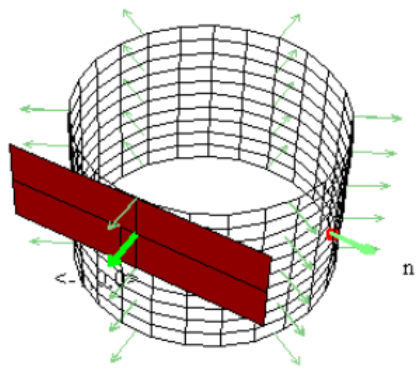

Figure 3.7: Normal vector for a cylinder. Figure taken from [20].

down sampled to the black centroid and how voxel grid covers all the points in the scene.

$$Centroid_i = (\bar{X}_i, \bar{Y}_i, \bar{Z}_i) \tag{3.21}$$

$$\begin{aligned} \bar{X}_i &= \tfrac{1}{N}\sum_{i=1}^{N} X_i \\ \bar{Y}_i &= \tfrac{1}{N}\sum_{i=1}^{N} Y_i \\ \bar{Z}_i &= \tfrac{1}{N}\sum_{i=1}^{N} Z_i \end{aligned} \tag{3.22}$$

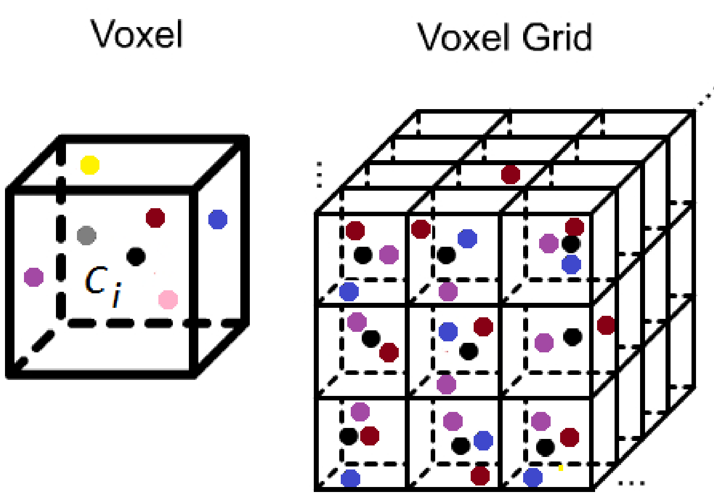

Figure 3.8: A voxel and a voxel grid where the colored points are down sampled to the black centroid.

The leaf size allows the user to decide the resolution of the down sampled data. Small leaf size results in a larger resolution because the number of voxel i.e centroids increase and the distance between each centroid decrease.

3.3.4 Random sample consensus

The Random Sample Consensus (RANSAC) algorithm is an approach developed from within the computer vision community to estimate parameters of a mathematical model from a set of observed data which contains outliers. These parameters should mathematically describe a model consisting of only inlier data. The estimation of parameters for a model is a learning technique where a random subset containing minimal data is taken from the input data set [12]. Within a subset of data the parameters for the model are estimated. These parameters are then tested against the rest of the data and given a score, namely the number of inliers fitting these parameters. If the score of inliers is not big enough, new parameters are found from another random data subset. This is repeated N times until the score of inliers are above an acceptable level.

The number of iterations N is set high enough to ensure that the probability of at least one of the sets of random subset does not include an outlier, which should be greater than $P = 0.99$. Let u be the probability of selecting an inlier from the data set and v = 1 - u the probability of selecting an outlier. The probability of all selected data m are inliers is u^m. This gives the equality 3.23 and with some manipulation the equation 3.24.

$$1 - p = (1 - u^m)^N \tag{3.23}$$

$$N = \frac{\log(1-p)}{\log(1-(1-v)^m)} \tag{3.24}$$

To define the parameters used in RANSAC the model of interest needs to be defined. If the model can be mathematically defined such as for boxes, spheres, cones, cylinders, lines, planes etc. then the RANSAC method can be used to estimate the parameters of the model together with model specific algorithms [22]. For this thesis RANSAC is used to obtain the parameters for a cylindrical model.

Table 3.3: Parameters found to estimate a cylinder using the RANSAC algorithm.

Cylinder parameter found by RANSAC
Point on center axis defined by a X-value
Pointon center axis defined by a Y-value
Point on center axis defined by a Z-value
Center axis in X-direction
Center axis in Y-direction
Center axis in Z-direction
Radius of cylinder

Cylinder estimation algorithm

For a cylinder there are seven parameters of interest, listed in table 3.3, and they can be estimated using the RANSAC criteria described in section 3.3.4 and the cylinder estimation algorithm described below.

From a subset of data, randomly select three non collinear points $P(x_i, y_i, z_i)$. With the three randomly selected points one can define a plane by equation 3.25. The constants A,B and C are found by solving the set of equations in 3.26. These equations are parametric in D and by setting D equal to any non-zero number and substituting it into these equations will yield one solution set.

$$Ax + By + Cz + D = 0 \tag{3.25}$$

$$\begin{aligned} Ax_1 + By_1 + Cz_1 + D &= 0 \\ Ax_2 + By_2 + Cz_2 + D &= 0 \\ Ax_3 + By_3 + Cz_3 + D &= 0 \end{aligned} \tag{3.26}$$

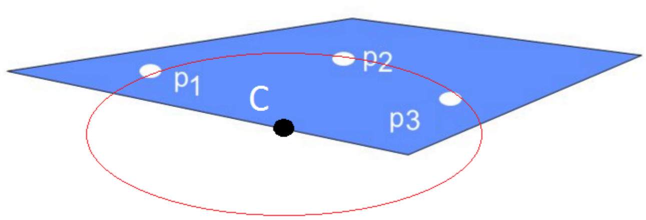

Figure 3.9: A plane and a circle in that plane can be defined by three points.

When A,B,C and D are obtained a circle lying in the plane is defined, see figure 3.9. The center of the circle C(x_0, y_0, z_0) and the radius r are found by equation 3.27 and 3.28 which states that the distance from each point P_i to the center should equal each other and the radius.

$$d(p_1, c) = d(p_2, c) = d(p_3, c) = r \tag{3.27}$$

$$d(p_i, c) = \sqrt{(x_i - x_0)^2 + (y_i - y_0)^2 + (z_i - z_0)^2} \tag{3.28}$$

The normal vector to the plane has the values given from 3.25 (A,B,C), giving a normal vector described by equation 3.29. A line parallel to the normal vector and intersects the plane through the center point of the circle, is equivalent to the center axis for a cylinder given by equation 3.30.

$$\vec{n} = A\vec{i} + B\vec{j} + C\vec{k} \tag{3.29}$$

$$\Upsilon \equiv \begin{cases} x = x_0 + tA \\ y = y_0 + tB \\ z = z_0 + tC \end{cases} \tag{3.30}$$

Once the center axis is obtained the shortest distance between the axis Υ and every point in the data are calculated. For this, the vector passing through the center point C(x_0, y_0, z_0) and the data point of interest P_j is defined as $c\vec{p}_j$. The cross product of two 3D vectors, $c\vec{p}_j$ and $\vec{n}$ are the same as the area of the parallelogram spanned by them. The same area can also be calculated by multiplying the length of the base $|\vec{n}|$ times the height d(P,Υ), see figure 3.10 and equation 3.31. Manipulation of 3.31 gives the equation 3.32 used to calculate the minimum distance between every point in the data set and the center axis Υ.

$$\left|\vec{n} \times c\vec{p}_j\right| = d(P_j, \Upsilon) \cdot |\vec{n}| \tag{3.31}$$

$$d(P_j, \Upsilon) = \frac{\left|\vec{n} \times c\vec{p}_j\right|}{|\vec{n}|} \tag{3.32}$$

If the distance of point $d(P_j)$ is within the boundaries of the radius plus/minus an error threshold $r \pm \varepsilon$ it means that the point is an inlier, while all points outside the boundaries is an outlier [15]. This method are repeated N times and the parameters with the most inliers are chosen for the best estimation of the cylinder. In figure 3.11 the boundaries creates an inner

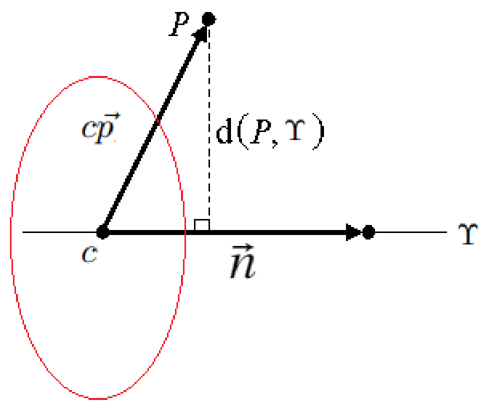

Figure 3.10: The distance between a point and the center axis

and outer radius, all points not bounded by them are removed and the data have been down sampled. A point cloud representing a cylinder captured by the Kinect is shown in figure 3.11 where the yellow points are outliers and red points inliers.

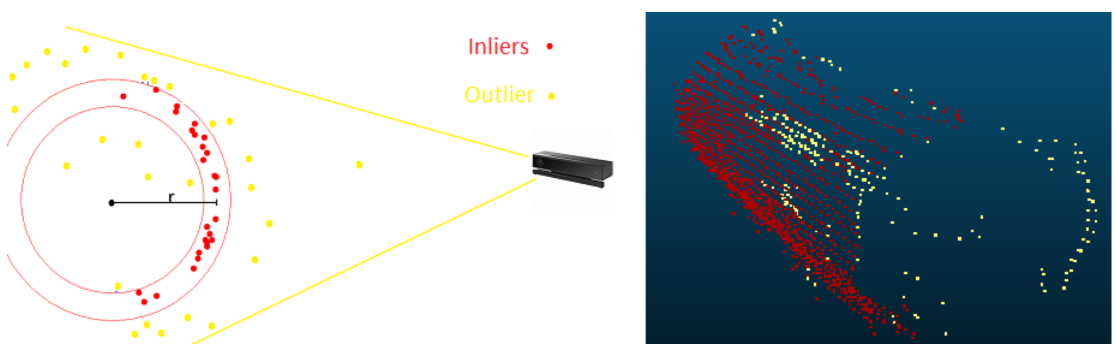

Figure 3.11: The yellow points are outliers and are removed from the data. Left side demonstrates the boundaries describing a cylinder. Right side demonstrates a data set captured by the Kinect where it is down sampled using RANSAC.

3.3.5 Smoothing - Moving Least Squares

After down sampling a data set, the surface still contain irregularities caused by measurements error and inherent noise 3.2.2. These are very hard to remove using statistical analysis, but the method *Moving Least Squares* (MLS) has shown to be very useful to reconstruct and smoothening of surfaces. Moving least-squares is insensitive to noise using algorithms of higher order polynomial interpolations between the surrounding data points. The algorithm starts with a weighted least squares formulation for an arbitrary fixed point and moves this points over the entire domain. At each point a weighted least squares fit is computed and evaluated individually. The detailed computation behind this method is presented by P. Lancaster and K. Salkauskas in their article *Surfaces Generated by Moving Least Squares Methods* and is beyond the reach of this thesis.

3.4 Model a cylinder model

Instead of importing CAD meshes and transforming them into point clouds, the target model is mathematically composed given an arbitrary axis with unit vector w and a point (x_0, y_0, z_0) in which the axis goes through. Further, suppose that u and v are unit vectors that are both mutually perpendicular and are perpendicular to the axis. By taking a random point P and finding the vector from (x_0, y_0, z_0) to P then u is obtained by taking the cross product of this vector and w. v is obtained by the taking the cross-product of u and w. Then the surface of a cylinder can be expressed by the parametric equations 3.33 using the vectors shown in figure 3.12 [28].

$$
\begin{aligned}
x &= \mathrm{x}_0 + r cos(\theta)\, u_x + r sin(\theta)\, v_x + t w_x \\
\mathrm{y} &= \mathrm{y}_0 + r cos(\theta)\, u_y + r sin(\theta)\, v_y + t w_y \\
\mathrm{z} &= \mathrm{z}_0 + r cos(\theta)\, u_z + r sin(\theta)\, v_z + t w_z
\end{aligned}
\tag{3.33}
$$

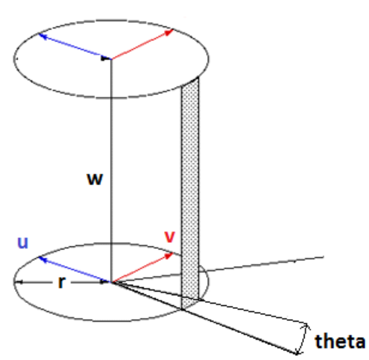

Figure 3.12: The surface of a generic cylinder is mathematically represented by knowing the axis and a starting point.

Theta (θ) ranges from the interval 0 to 2π and t ranges over the set of real numbers. Figure

3.13 shows a generated point cloud created using equations 3.33. This method of generating a generic cylinder makes it easy to change pose, length and radius rather than drawing a CAD and meshing it in MeshLab.

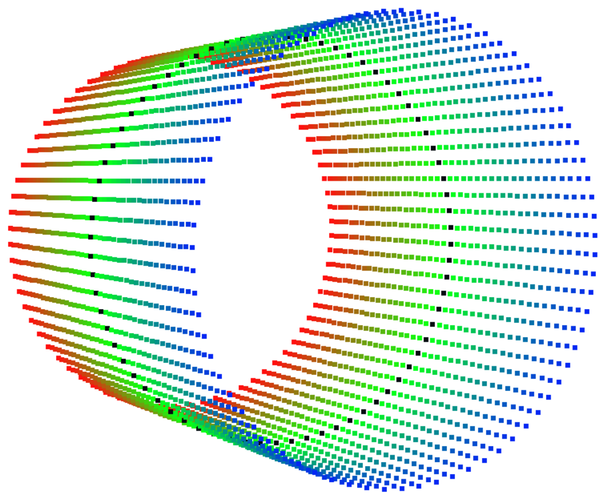

Figure 3.13: A created point cloud using parametric equations for the surface of a cylinder.

3.4.1 Finding center axis of model cylinder in the camera coordinate system

The center axis for a perfect cylinder without run-out should in this project have a unit vector $\hat{w}_p$ of (0,1,0) in the robot coordinate system. In subsection 3.2.4 the transformation matrix to map vectors from the camera frame to the two robot frames was presented. Using the inverse of the transformation matrix one can map vectors from the robot frame to the camera frame and the center axis vector w in the camera frame can be obtained using equation 3.34.

$$\hat{w} = T_C^{-1} \hat{w}_p \tag{3.34}$$

3.5 Alignment of cylinders

The goal of this thesis are to locate two cylinders in the scene captured by the Kinect and obtain the transformation matrix between their actual pose and a wanted pose. The transformation matrix consist of the rotation and translation needed to align a model in the data set from the scene onto the target model.

To effectively and successfully detect an object and its pose in a large data set it is required that most of the points not belonging to the object is removed using the processes described above in section 3.3. When as much of the points not describing the object are filtered, the alignment can be executed.

Aligning two point clouds in 3D is called registration and in this thesis different methods were tested for alignment, namely Iterative Closest Point (ICP) and SAmple Consensus Initial Alignment (SAC-IA) for a adequate transformation matrix and a self composed method using RANSAC to obtain rotation and an algorithm denoted *Search Method* to find translation. These are described in subsection 3.5.2, 3.5.1 and 3.5.3 respectively.

3.5.1 SAmple Consensus Initial Alignment - SAC-IA

Finding an object and aligning it with a target in a scene can be done using feature descriptors which describes local geometry such as corners, edges, ridges and shape of surfaces. The feature descriptors are derived for both the target and the object in the scene. When the feature descriptors are computed for both target and object the search for matching correspondence pairs between them is computed and the alignment which minimized the error metric is chosen. *SAmple Consensus Initial Alignment* (SAC-IA) is an algorithm that uses *Fast Point Feature Histograms*(FPFH) to realize a first alignment between two different point clouds. The process of using SAC-IA for initial alignment is shown in figure 3.14. Down sampling, removing of outliers, Moving Least Squares and normal estimation are already explained in section 3.3.

Descriptor - Fast Point Feature Histograms

The Fast Point Feature Histograms (FPFH) algorithm originates from Point Feature Histograms (PFH) which describe the local geometry around a point p for 3D point cloud datasets. FPFH is a faster way to compute descriptors still able to retaining most of the descriptive power of the PFH. The PFH computation relies on finding the mean curvature around a point p by looking at its k-neighbor points surface normals. The accuracy of the PFH is strictly related to how good the points normal describes the underlying surface. The computation of the histogram for a point p can be described in three steps [25].

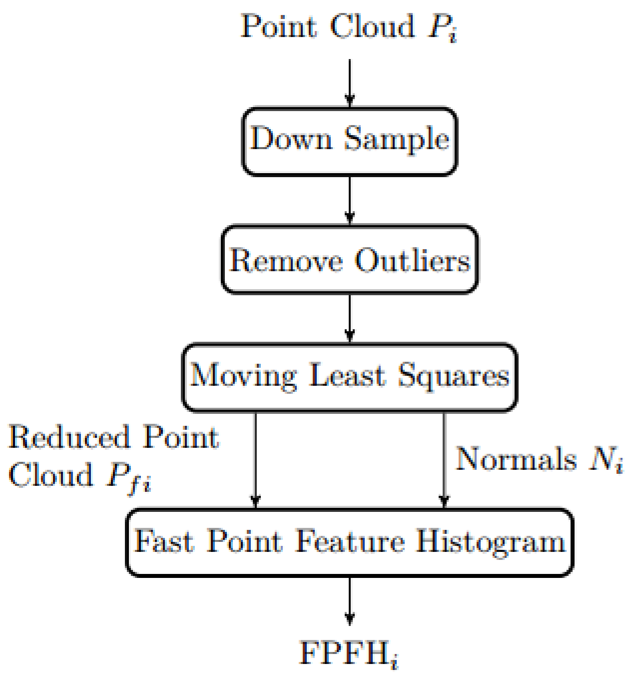

Figure 3.14: The process of using FPFH for the SAC-AI alignment.

1. For each point p all surrounding points enclosed by a sphere with radius r is selected. These points are denoted k-neighborhood. See figure 3.15 for the enclosed speher and the points in the k-neighborhood.

2. For every pair of points p_i and p_j (j $\neq$ i, $j < i$) in the k-neighborhood and their estimated normals n_i and n_j where p_i being the one having the smaller angle between the associated normal and the line connecting the points [7]. Define a Darboux *uvw* frame where $u = n_i$, $v = (p_j - p_i) \times u$ and $w = u \times v$.

3. Obtain the angular variations of n_i and n_j using equation 3.35.

$$\begin{aligned} \alpha &= v \cdot n_j \\ \phi &= (u \cdot (p_j - p_i)) / \left\| p_j - p_i \right\| \\ \theta &= \arctan(w \cdot n_j, u \cdot n_j) \end{aligned} \tag{3.35}$$

PFH is computationally costly $O(k^2)$ opposed to FPFH O(k), where k is the number of neighbors for each point. To get the FPFH for a point p a Simplified Point Feature Histogram (SPFH)

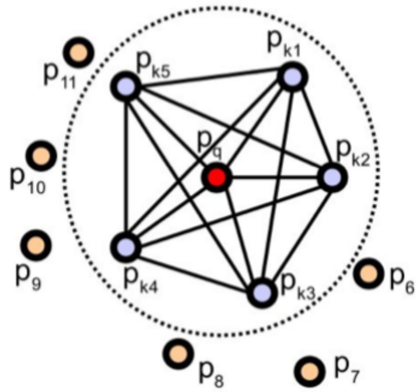

Figure 3.15: The influence region diagram for a Point Feature Histogram, figure from [7].

is obtained which only calculates the relationship between itself and its neighbors, see figure 3.16. Further, for each point the k-neigbors is re-determined and their SPFH values are used to weight the final histogram of p. Equation 3.36 computes FPFH for a point p using the simplified version, ω_k is the weight representing the distance between query point p and a neighbor point p_k.

$$FPFH(p) = SPFH(p) + \frac{1}{k}\sum_{i=1}^{k}\frac{1}{\omega_k}\cdot SPFH(p_k) \tag{3.36}$$

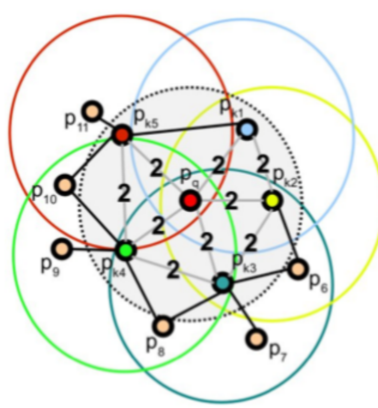

Figure 3.16: The influence region diagram for Fast Point Feature Histogram using Simplified Point Feature Histogram , figure from [7].

SAC-IA algorithm

After FPFH sample large numbers of correspondence candidates and rank each of them very quickly using the following scheme:

1. Select a number of sample points from the data point representing the object. The sample points are denoted s and their pairwise distances must be larger than a defined distance d_{min}.

2. For each point s compare its histogram to the histograms for points on the target and make a list of points which has similar histograms. From the list, select randomly one point which will be considered that sample points correspondence.

3. Calculate the rigid transformation matrix to align the sample points and their correspondences. An error metric for the quality of the transformation is computed.

These steps are repeated and the transformation with the lowest error metric is used for initial alignment.

3.5.2 Iterative Closest Point - ICP

ICP starts with two sets of data, point clouds, and an initial guess for their relative rigid-body transform. In this case SAC-IA has been used to obtain the initial guess. It then iteratively refines the transform by repeatedly generating pairs of corresponding points in the point cloud and minimizing the error metric. Figure 3.17 shows a cylinder represented by white points being aligned with a target model with yellow points using ICP. The algorithm scheme for ICP [32]:

1. Create a pairing between point sets, closest points are matched.

2. Compute the rigid registration given the pairing.

3. Apply the transformation to the data and compute the mean distance between point sets.

4. If change in the mean distance is not below a given limit or the number of iterations has not reached a maximum number, repeat steps 1,2 and 3.

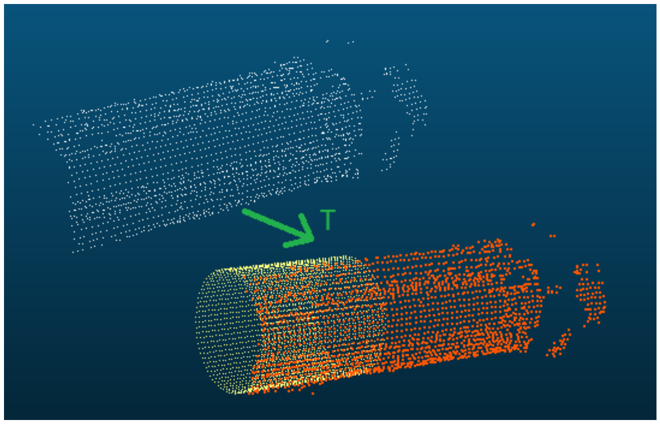

Figure 3.17: The white points represents a cylinder which is aligned with the yellow target model.

Given the data points $\{\vec{D}_i\}$ and target model points $\{\vec{M}_i\}$, find the rigid transformation with translation $\vec{T}$ and rotation R which minimizes the sum of the squared distance of equation 3.37 [19].

$$d_i{}^2 = [\vec{M}_i - (R\vec{D}_i + \vec{T})]^2 \tag{3.37}$$

For the equation to be able to handle inconsistent points and outliers Chen And Medioni [9] dynamically weighted every point so that the error is calculated according to equation 3.38.

$$error = \left(\frac{1}{n}\right)\sum_{i=1}^{n} w_i * d_i{}^2 \tag{3.38}$$

w_i is the weight for point i and $d_i{}^2$ is the squared distance from a data point to the model surface.

The result of using ICP for alignment of cylindrical object is found in 5.2.1

3.5.3 RANSAC to align orientation

Described in section 3.3.4 the RANSAC algorithm is used to down sample the data set by removing outliers not within a distance threshold of the cylinder surface. Three of the parameters found when estimating a cylinder are used to define the direction vector of the center axis. The direction vector of the captured cylinder can be rotated so that it becomes parallel to the center axis of the target model. This is done by defining a coordinate system for the cylinder using the direction vector as one of the axis and the two other axis being perpendicular to this vector. The direction vector found by RANSAC is denoted v_k and the wanted direction vector $\hat{w}$.

Henceforth, the estimated cylinder and the modeled cylinder will be referred to as "cylinder" and "target".

First find the unit vector for the direction vector for the cylinder $\hat{v}_k$ [11].

$$\hat{v}_k = \frac{\hat{v}_k}{|\hat{v}_k|} \tag{3.39}$$

Define a coordinate system for the cylinder by obtaining two vectors which are perpendicular to the direction unit vector and each other. There exists an infinite number of vectors in three dimension that are perpendicular to a fixed one. Pick *any* non-zero vector $\hat{v}$ that is not parallel to $\hat{v}_k$. The cross product between the unit vector $\hat{v}$ and $\hat{v}_k$ will define one of the axis denoted $\hat{r}$, as in equation 3.40. The other axis is defined by the cross product between $\hat{r}$ and $\hat{v}_k$ and is denoted $\hat{a}$, see equation 3.41.

$$\hat{r} = \hat{v}_k \times \hat{v} \tag{3.40}$$

$$\vec{a} = \vec{r} \times \vec{m} \tag{3.41}$$

These three unit vectors defines the cylinder coordinate system denoted K and the rotation matrix which can map the components of any vector r between the cylinder frame and the camera frame consists of rows being the unit vector defining the cylinder frame, see 2.5. The rotation matrix R_K is found in equation 3.42.

$$R_K = \begin{bmatrix} - & \hat{v}_k & - \\ - & \vec{r} & - \\ - & \vec{a} & - \end{bmatrix} \tag{3.42}$$

The wanted frame also needs to be defined. The cylinder wants to have a direction vector parallel to the approach axis of the robot (Y-axis) in the robot frame $\hat{w}$, but because the robot and camera frame are not perfectly aligned a rotation matrix is needed to map from the camera to the wanted orientation. In the camera frame, the wanted frame for the cylinder is the same as the robot frame and this rotation matrix was found when calibrating the camera in the two robot frames. This means that the wanted frame can be defined according to equation 3.43 and this gives the two rotation matrices 3.44 and 3.45 between the camera and wanted orientation.

$$R_w = \begin{bmatrix} - & \hat{OY} & - \\ - & O\hat{Z}_{new} & - \\ - & \hat{OX} & - \end{bmatrix} \tag{3.43}$$

$$R_{w_{right}} = \begin{bmatrix} 0.9989 & -0.0215 & 0.0418 \\ 0.0222 & 0.9997 & -0.0116 \\ -0.0416 & 0.0125 & 0.9991 \end{bmatrix} \tag{3.44}$$

$$R_{w_{left}} = \begin{bmatrix} 0.9988 & -0.0184 & 0.0447 \\ 0.0144 & 0.9996 & -0.0224 \\ -0.0443 & 0.0230 & 0.9987 \end{bmatrix} \tag{3.45}$$

The rotation for the robot frame is denoted R_R and is defined in equation 3.46. R_C is the rotation matrix between camera and robot, R_w is the wanted frame and R_K is the actual frame.

$$R_R = R_C R_w R_k \tag{3.46}$$

Figure 3.18 represents a cylinder and its coordinate system in the camera frame and figure 3.19 shows the setup for the right robot.

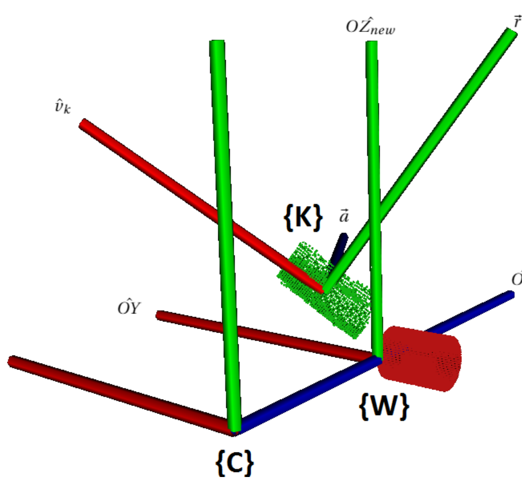

Figure 3.18: Camera frame C, wanted frame W and cylinder frame K.

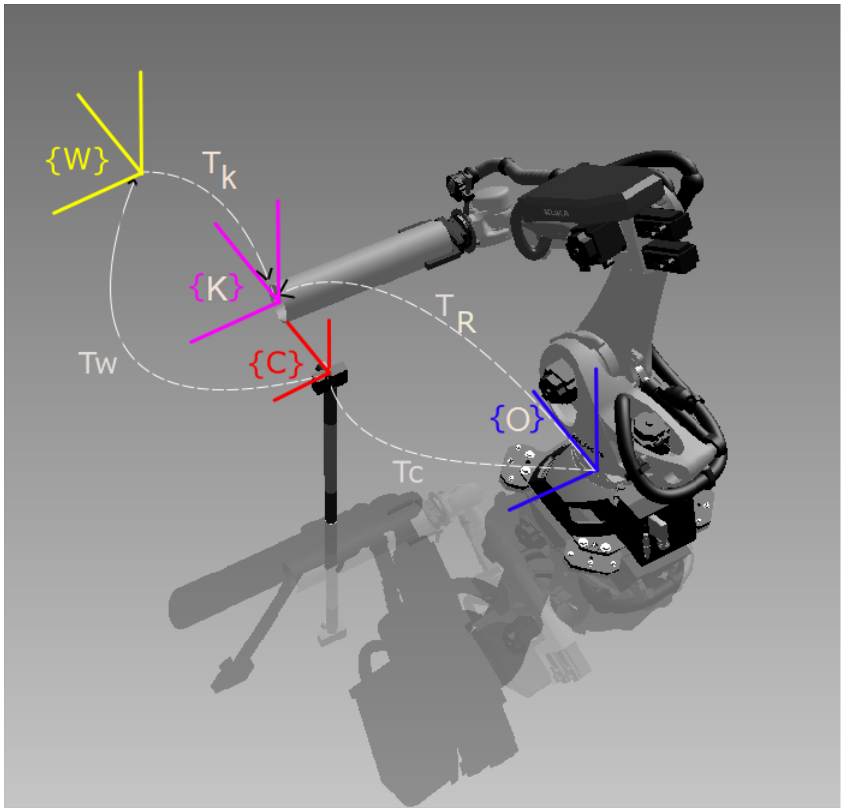

Figure 3.19: The setup of the right robot and the frames used in this thesis.

3.6 Align position using Search Method

As the rotation matrix is found using RANSAC the missing part to fully define a transformation matrix is the translation needed to align a cylinder given a wanted pose. To find the translation the implementation of an algorithm developed by the author of this thesis called "Search Method" is used.

The algorithm consist of searching for a set of points, represented by the black and purple points in figure 3.20, near the edge of the cylinder and use their values to represent the point $M_{edge} = (x_m, y_m, z_m)$ denoted with orange color in the same figure. The black points are used to estimate y_m and z_m, while the purple points are used to estimate x_m. How the algorithm filters and finds these points are explained later. As the values for the target value are known $P_{edge} = (x_p, y_p, z_p)$, the translation t consist of the difference between M_{edge} and P_{edge} described in equation 3.47.

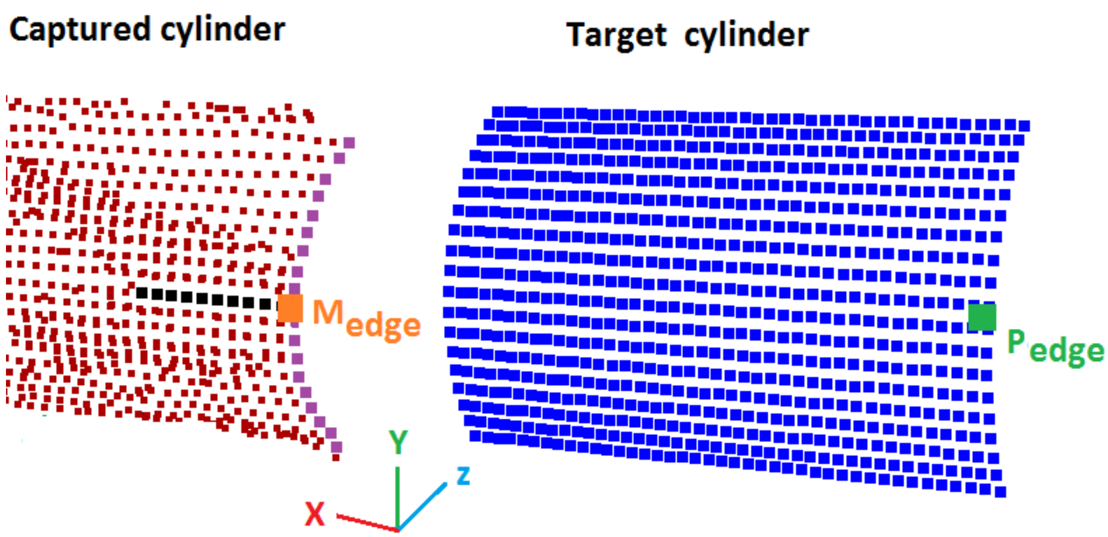

Figure 3.20: Points of interest in black and purple are used to represent M_{edge} while P_{edge} is a known position on the target model.

$$t = \begin{pmatrix} x_m - x_p \\ y_m - y_p \\ z_m - z_p \end{pmatrix} \tag{3.47}$$

There are two cylindrical object to be align in this thesis and they are both held by the robots shown in 4.2. Each cylinder is denoted as *left cylinder* and *right cylinder* where all the points representing the right cylinder have negative x-values in the camera frame and left wtih all postive values. For the following algorithm the conditions change depending on which cylinder that will be aligned.

The first step in the Search Method is to roughly locate the end of the right cylinder going through every data point and searching for the biggest value lying on the axis represented by the center axis. For the left cylinder the smallest value is wanted. The target models in this thesis will have a center axis, earlier denoted as $\hat{OY}$ to be approximately equal to (1,0,0) and to be parallel with the robot y-axis (0,1,0). Hence, search for the point with the biggest and smallest value of x in the camera frame when estimating the edge of the right and left cylinder. This point is denoted $M_{estimate}$ and found according to equation 3.48 and 3.49 for the point cloud Q representing the right cylinder and D for left cylinder repectivly. Because of noise, specially around the edge in which the point processing described in 3.3 fails to perfectly filter, this value can not represents any values of M_{edge}. Figure 3.21 is an exaggerated example of how noise near the edge is not perfectly filtered, resulting in a point $M_{estimate}$ not describing any part of the cylinder.

$$M_{estimate}(x_{right}, y_{right}, z_{right}) = \max_x \vec{Q}_i \tag{3.48}$$

$$M_{estimate}(x_{left}, y_{left}, z_{left}) = \min_x \vec{D}_i \tag{3.49}$$

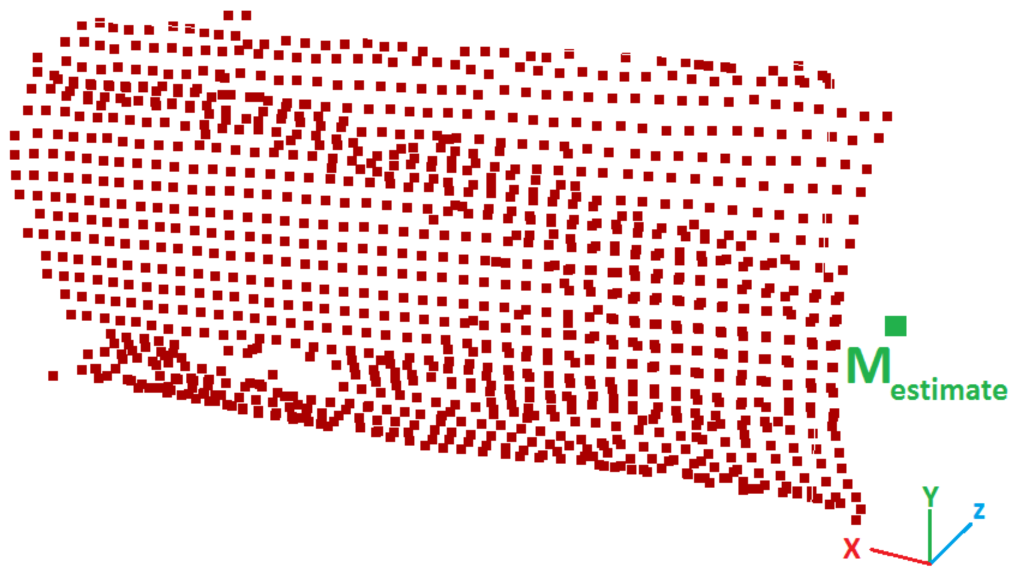

Figure 3.21: When searching for a roughly value for the end of a cylinder noise will falsifying the results representing the actual end. This is the left cylinder where the edge has a postive x-value.

The next step in finding the set of points used to estimate M_{edge} is to divide parts of the cylinder lying close to the value of $M_{estimate}$ into n intervals along the x-axis. Figure 3.22 shows how the cylinder is divided over a length L_x starting from $M_{estimate}$ into intervals of width δ.

Starting at the first interval N_1 and going through until N_n the algorithm does the following:

1. Search for the point with the smallest z-value (closest to the camera) which is lying within the boundaries of interval N_i and store the z and y value for the point.

2. Calculate the normal vector for the point found in step 1.

3. Count the number of point lying inside interval N_i.

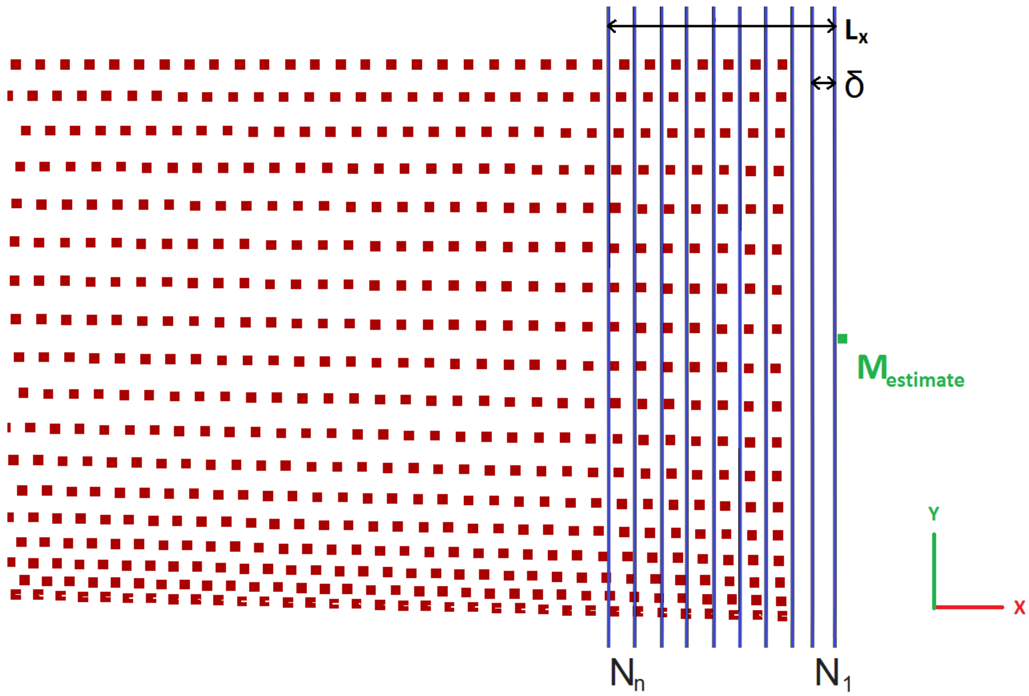

Figure 3.22: The end of the cylinder is divided into interval of width δ.

4. Check if the number of points in interval N_i is above a given limit. If so, calculate the mean of the x-values contained by that interval.

5. Repeat 1-3, with interval N_{i+1} until N_n.

The data given from the scheme is stored in a two-dimensional array as the one given in table 3.4. The purple points are defined as every point inside the first interval containing over a given number of points. These are used to estimate x_m by taking the mean of the x_j-values of the C_i number of points inside interval i.

$$x_m = \frac{1}{C_i} \sum_{j=1}^{C_i} x_j \tag{3.50}$$

Further, duo to noise and irregularities on the surface representing the cylinder the data stored from the search scheme above is filtered before computing the estimation of y_m and z_m. All data from interval i is removed from the array if its data do not meet the two following criteria:

1. The normal vector in Z direction $n\vec{k}_i$ is outside the value of [0.99,1.00],

2. The number of points C_i in interval i is below a given threshold.

Table 3.4: Data captured in each of the N intervals. The data represents the coordinate values and normal vector in z direction of the point being the closest to the Kinect.

Interval	$\mathbf{y}_i$	$\mathbf{z}_i$	**Normal**Z_i	**PointCount**$_i$
0	y_0	z_0	$n\vec{k}_0$	C_0
1	y_1	z_1	$n\vec{k}_1$	C_1
2	y_2	z_2	$n\vec{k}_2$	C_2
...				
N x_n	y_n	z_n	$n\vec{k}_n$	C_n

Looking at a cylinder with a center axis perpendicular to the observer, the normal vector for the closest point on a cylinder will always point back towards the observer. This means that if the center axis is in the XY-plane of the camera frame and the observer is looking down the z-axis the points with the smallest z-values should have a normal vector of (0,0,-1), see figure 3.23. This gives that all points stored in table 3.4 with normal vectors deviating greatly from (0,0,-1) will give an erroneous estimation of M_{edge} and are therefor removed. Secondly, the algorithm for calculating the estimation of the normal vector for a point utilizes the surrounding data neighbors. If there is a lack of sufficient neighbors the normal vector can be wrongfully calculated passing an erroneous point through the first criteria. For this reason intervals not containing enough points are removed as well.

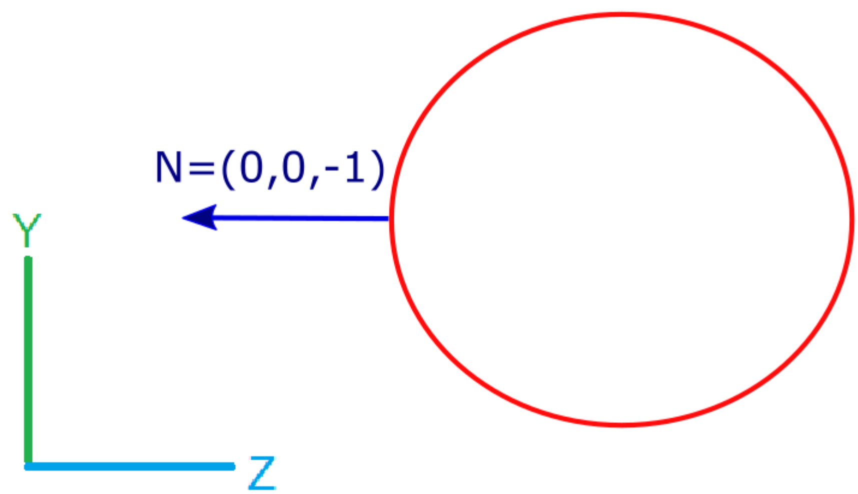

Figure 3.23: The point on a cylinder with the smallest z-values have a normal vector of (0,0,-1).

Because of the high accuracy needed in this thesis to be able to fit-up two tubes for welding, an outlier detection algorithm is applied on the remaining data as well. If the weight of noise inside a voxel grid wrongfully place the centroid or if two grids splits a dense sampling of points into two separate centroids, the representation of the underlying surface by that single point will be inaccurate. To remove potential outliers in both y_i and z_i the *THE MODIFIED Z-SCORE* outlier detection algorithm by Iglewicz and Hoaglin [18] was implemented. Each value of y and z in the data set is given a Z-Score, where absolute scores over 3.5 are denoted an outlier and removed.

The median and the median of the absolute deviation of the median (MAD) given in equation 3.51 where $\tilde{x}$ is the median of the remaining x_i values.

$$MAD = median\{|x_i - \tilde{x}|\} \tag{3.51}$$

The Z-Score (Z_i) is computed by equation 3.52 where $E(MAD) = 0.6545\sigma$ for data with over 10 samples.

$$Z_i = \frac{0.6745(x_i - \tilde{x})}{MAD} \tag{3.52}$$

The highly filtered remaining data is finally ready to be used for calculating y_m and z_m. The values are computed taking the mean value of the remaining y_i and z_i values.

$$y_m = \frac{1}{N}\sum_{i=1}^{n} y_i \tag{3.53}$$

$$z_m = \frac{1}{N}\sum_{i=1}^{n} z_i \tag{3.54}$$

As all the values in M_{edge} are found the translation t can be computed. The results of the filtering and the performance of the algorithm are presented in section 5.13

3.6.1 Search Method together with RANSAC

Explained in 3.5.3 RANSAC can only be used to find and align orientation, but to obtain an adequate transformation matrix one need to find translation as well. For this the Search Method algorithm described above is utilized. To search for the point earlier denoted as M_{edge} the point cloud must be orientated such that the center axis lies in the XY-plane. Any rotation of a point cloud captured by the Kinect will be rotated about the camera frame, changing the position value of every point in the cloud including M_{edge}. To reduce the position change of M_{edge} due to rotation one should align it with the origin of the camera frame, but since M_{edge} is still unknown the centroid of the point cloud is aligned instead. The centroid of a point cloud is computed using equation 3.22. When the centroids position is aligned with the camera origin the cylinder frame is oriented to have the same frame as the camera. It is at this stage the Search Method algorithm can start searching for M_{edge}, but since this point have been shifted when rotating around the centroid it is denoted as $M^{|}_{edge}$. To find the true value of M_{edge} multiply with the inverse rotation matrix of equation **??** and add the translation done when aligning the centroid with the origin t^c_0. The steps of finding the true value of M_{edge} is described in figure 3.24. This gives the equation 3.55 which precisely estimates the value of the end of the cylinder in any orientation.

$$M_{edge} = (R_K)^{-1} M^{|}_{edge} + t^c_0 \tag{3.55}$$

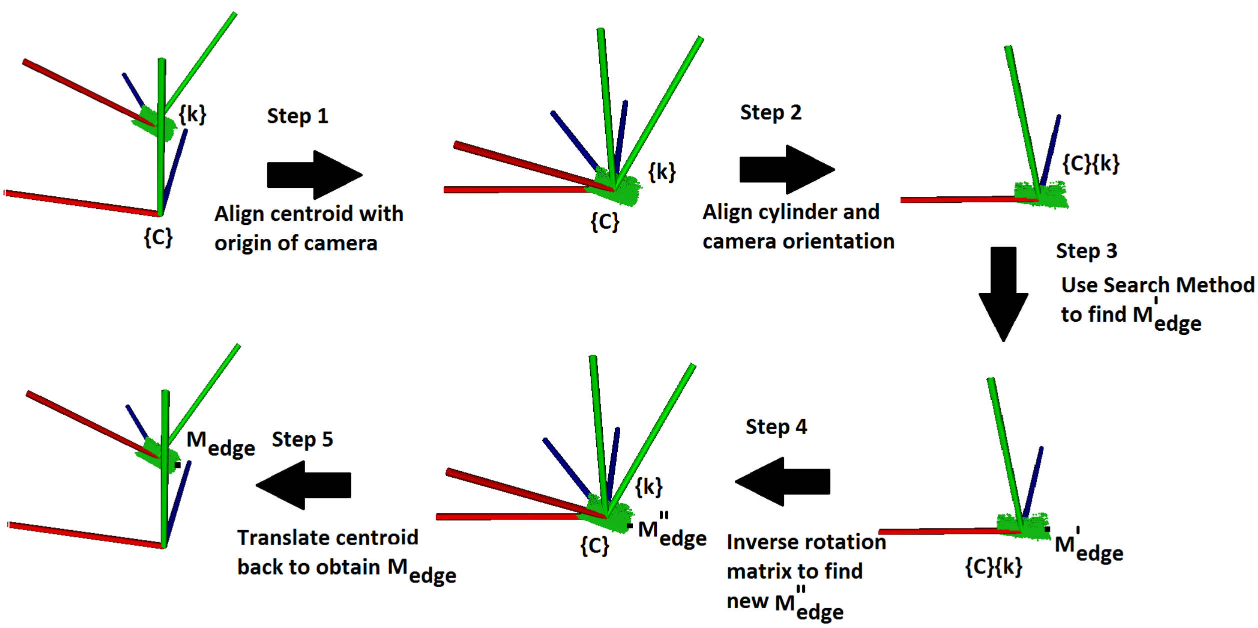

Figure 3.24: When searching for a roughly value for the end of a cylinder noise will falsifying the results representing the actual end.

For the robots to rotate about M_{edge} the tool center point is translated along the approach axis of the end effector by the value of the length of the cylinder. This value is computed when the robots picks up the cylinder with unknown length by storing the value of z in the robot frame when it grabs the tube. This is shown in figure 3.25.

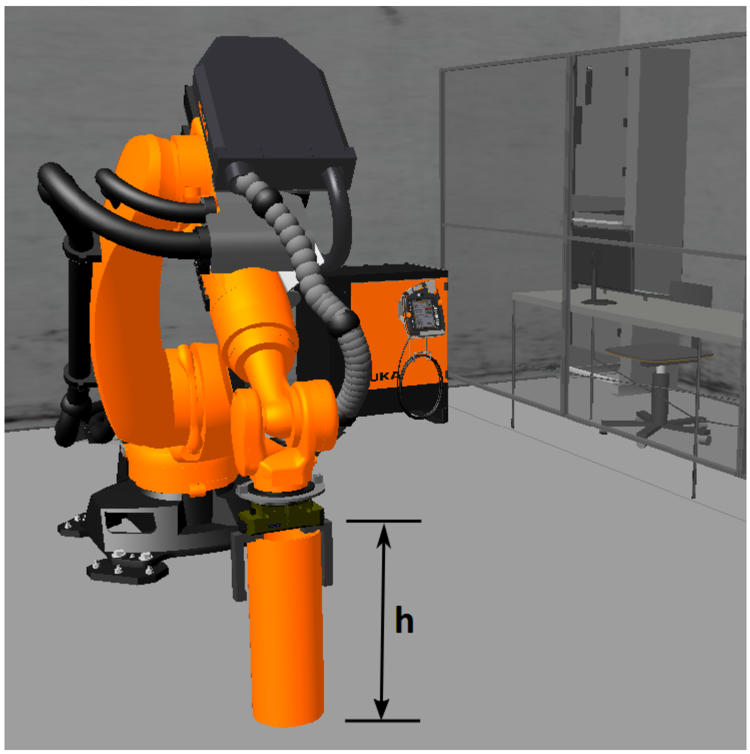

Figure 3.25: The translation value of the tool center point along the approach axis of the robot is found when the robots picks up its tube.

Chapter 4

Setup and Robot Control

4.1 Robot Lab

For this thesis the utilization of four robots are used to handle cylindrical tubes and weld them together. Two KUKA 120 R2600 pro are used as handling robots, being able to pick up cylindrical object using pneumatic 3-finger centric grippers. The coordinate system for the KUKA KR 120 is shown in figure 4.1. For welding the lab is equipped with a Fronius TransSteel 5000 welding machine which has a welding gun connected to the KUKA KR 16-2 while the control of the welding is integrated on the KUKA KR 5 Arc robot. This is a Metal Active Gas (MAG) weld which uses a shielding gas to protect the process from being contaminated by air. Further, the Kinect is located between the two handling robots. In the direction of the Kinect, the KUKA 120 on the left side is now denoted as *LEFT* robot and the other as *RIGHT* robot. Figure 4.2 shows a visualization of the lab setup.

The robot controllers used by the KUKA robots at the lab are four KR C4 and they can integrate robot control, PLC control, motion control and safety control. The SoftPLC option makes the KR C4 controller able to control complete robot cells by I/O handling. However, the CR 4 controllers at the lab do not support this because there is no implementation of the physical connection opportunities to other devices. For this reason the I/O handling is done through the PLC at the lab. The only exception is the Fronius TransSteel 5000 welding machine which is connected to the KR C4 for the KUKA KR 5 robot. This means that the welding machine can be controlled independently by the KR 5 without the use of an external PLC.

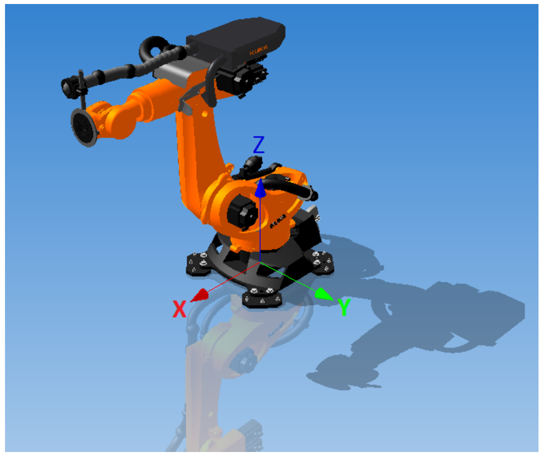

Figure 4.1: The coordinate system of the KUKA 120 R2500 robot.

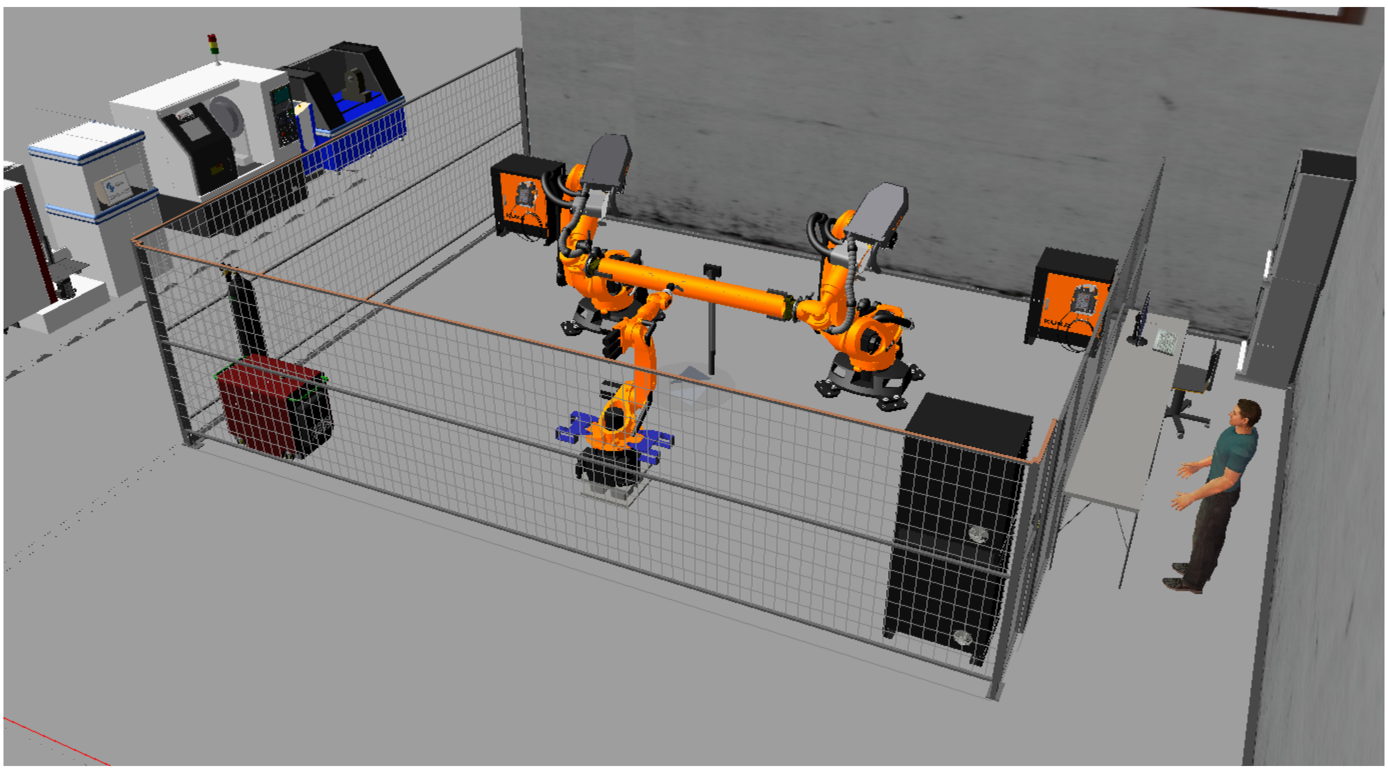

Figure 4.2: A visualization of the lab containing the three robots and its CR4 controller used in this thesis, the welding machine and the computer controlling the robots through the PLC server.

4.2 Offline Programming

Offline Programming is a method to develop a simulation of a robot in a virtual environment similar to the real one in the robot cell. There are a number of steps to follow to successfully generate robot programs for a robot.

When the program is tested for errors, collision detection and optimized motion planing, the next step is to convert the program to a language that the robot can understand. The post processing for KUKA is done in KUKA-SIM, which converts RSL (Robot Scripting Language) into the native KUKA source code file (SRC).

The program is then installed and tested on the robot, but very often there is deviation between the simulation and the real robot that needs to be calibrated. It is often caused in step 2 when creating the environment with coordinate offset error, but also deviation occur because of geometric parameters and non-geometric parameter [16]. Geometric parameters are play between parts and mechanical deflection due to load. While non-geometric parameters are joint and link flexibility and thermal strain [30].

For the deviation in the program, one need to go back to the simulation and correct it until the deviation is minimized. To ensure that the coordinate system in the simulation is translatable to the real coordinate system, the robot need to be completely and correctly mastered. Only then can the robot preform poses and path accurately, and be moved using programmed motions at all. This includes calibration of the tool and base and teaching offsets using load correction of tool and workpiece [3]. This is called Robot Calibration and is preformed using the KUKA-pendant.

All the robots and peripheral devices needs to be controlled so that they can work together. Depending on how many I/O counts the system runs and the complexity of the system logic, the choice of control method is decided. If the system is complex the need for an external PLC which manage I/O processing over various different bus level networks is required. The PLC has the role of being 'master' and the robots/devices connected to the PLC being 'nodes'.

4.3 C++ application to align cylinders

An application was developed for this thesis using the programming language C++ in the integrated development environment from Visual Studio 2013. For the development of the application the *Point Cloud Library* and a set of 3rd party libraries were used. These are open-source libraries of algorithms for point cloud processing tasks and 3D geometry processing. The libraries used in the development are listed in table 4.1

Table 4.1: The libraries used in the C++ application.

Library	Includes	Function
3rd party library dependencies	Boost	Shared pointers and threading
	Eigen	Matrix and Vector operations
	VTK	Visualization of point clouds
	FLANN	Kdtree for fast approximate nearest neighbors search.
Point Cloud Library	Registration	ICP, SAC-IA and RANSAC
	Features	Normal estimation and FPFH
	Filters	Voxel Grid and passthrough
	Surface	MLS
	IO	Point cloud handling

The operations in this application are to locate two cylinders in the scene and calculate a correction transformation matrix to align them with two target models using the two robots. The different steps in the application are listed below.

1. Capture the scene using the Kinect and store it in a point cloud.
2. Filter the point cloud using pass through filter, voxel down sampling, RANSAC down sampling and MLS.
3. Store each cylinder in separate point clouds.
4. Create two target cylinders point clouds and set their position and orientation.
5. Run one of the alignments algorithms described in 3.5.
6. Obtain two transformation matrices for alignment of both cylinders and find the corresponding values in the robot frame.
7. Write and store the transformation matrices into two separate .txt files.
8. Visualize the alignment of the transformed cylinders.

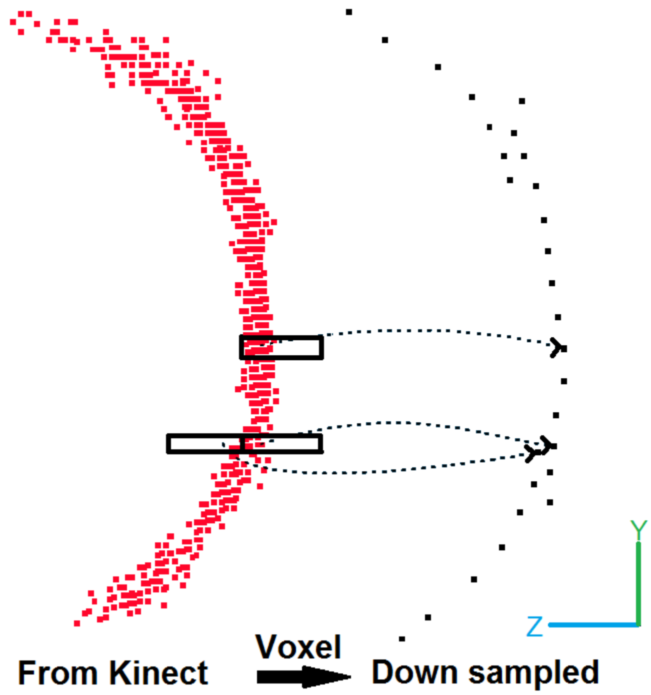

Figure 4.3: Left side: Cross section of point cloud representing a cylinder with 10 pictures. Right side: The same point cloud after down sampling using a rectangle sized voxel grid.

Step 1: Capture the scene

Using a class in C++ called *Kinect2Grabber* developed by Tsukasa SUGIURA one can grab a point cloud of the scene using the Kinect using the third party dependencies, Point Cloud Library (PCL), Windows SDK v2.0 and Visual Studio 2013. A series of 10 pictures were saved in the same point cloud because the representation from one captured point cloud can give erroneous results because of the fluctuating depth values. This can be seen on the left side in figure 4.3 where the cross section of a cylinder composed by 10 pictures are represented. The consequence of the depth inaccuracy is that points with relative similar x and y values have varying z-values, giving a row of points in the YZ-plane.

Step 2: Down sampling

A passthrough filter will greatly reduce the number of data points. The cylinders are held in the air by two robots in front of the Kinect fare a way from interfering floors or walls, making this filter efficient in removing insignificant data points. All data points lying outside the boundaries of Z[0.6,2], Y[-0.7,0.7] and X[-1.4,1.4], given in meter will be removed.

To deal with the rows of points voxel grid down sampling method explained in 3.3.3 is used, with a large leaf length in Z-direction. The red points in figure 4.3 are reduced to either one or two black points representing the underlying surface. The figure also shows how the grid system can split up an adjacent group of points into two points representing the surface. In figure 4.4 the red point cloud is a tube represented by 98,978 points, while the black point cloud is the same tube down sampled using voxel grid represented by 1,351 points.

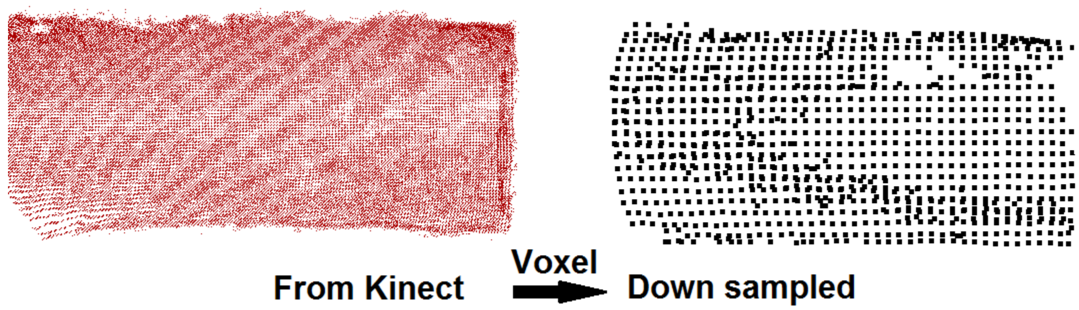

Figure 4.4: Left side represents a tube without voxel down sampling, while the right side is down sampled.

The RANSAC algorithm is used to remove every point not representing the surface of the cylinder or not being within a threshold distance of the surface, denoted outliers. Figure 3.11 shows how outliers are removed.

Further, for better representation of the cylinder surface the application runs the point cloud through the Moving Least Squares algorithm for smoothing the surface. The cross section of a down sampled point cloud to a smooth surface is shown in figure 4.5.

Step 3: Splitting the scene

The camera is positioned between the two handling robots and they both hold a cylindrical object in such a way that all points representing the cylinder held by the *RIGHT* KUKA robot has negative x-values. Going through every data point filtering and storing data with positive x-values and negative x-values into two separate point clouds.

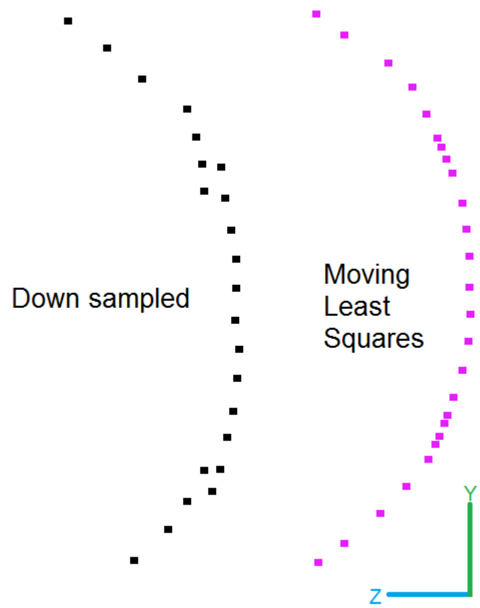

Figure 4.5: Left side: Cross section of down sampled point cloud . Right side: The same point cloud after smoothing using MLS.

Step 4: Create a target model

Two target cylinder are mathematically generated using the basics of equation 3.13. For simplification the cylinders are modeled along the x-axis for a length of 0.3m and then later translated 0.6m in z-direction and rotated so the direction vector of the target cylinder is parallel to the y-axis of the robot. The radius of the tube equals the radius of the cylinder used in the lab, 84mm.

Step 5: Select alignment algorithm

The algorithm for the selected method for alignment is started. The three methods are RANSAC together with Search Method, SAC-IA with ICP and Search Method for correction of translation and lastly the Search Method used without any rotation. These methods, RANSAC, SAC-IA, ICP and Search method are explained separately in subsection 3.5.3, 3.5.1, 3.5.2 and 3.6 respectively.

Step 6-7: Find the transformation matrix for each robot

The output from each of the alignment algorithms are the correction translation in each of the robot frames which are needed to align the position of the cylinders and a rotation needed to align orientation. The transformation matrices are written to two separate files *tranformation-matrix_RIGHT.txt* and *transformationmatrix_LEFT.txt* which are used by Matlab and a Java application later.

Step 8: Visualization

The Visualization ToolKit (VTK) is used to visualize the point clouds of importance for this project. This includes the starting pose, the target pose and the aligned point cloud which have been translated and rotated onto the target point cloud together with a live stream of the scene.

4.4 Communication application in Java

The software KUKA.RobotSensorInterface makes it possible to influence the robot motion or program execution via sensor data. The sensor data and signals can be read by a field bus, processed and forwarded to the robot controller. Or it is possible to use the software package KUKA.Ethernet KRL XML which makes it possible to set up under KUKA.RobotSensorInterface an anticyclic Ethernet link between a robot controller and up to nine external systems, like the Kinect. The data are transmitted via the Ethernet TCP/IP protocol as XML strings. The problem is that neither of these software packages are acquired at the robot lab.

The solution is to use a Java open-source cross-platform called JOpenShowVar. This allows for communication with all KUKA robots connected to a KR C4 controller. The communication allows for reading and writing variables and data structures of the controlled manipulators. The JOpenShowVar works as a client middleware between the Java application running on a remote computer and the KUKAVARPROXY acting as a server on the KR4 controller connected via TCP/IP [14]. Figure 4.6 show the architecture for JOpenShowVar communication with the KUKA robots.

To be able to read and write information to the robots, all variables need to be predefined as global variables in the system data list $CONFIG.DAT. The type of the global variables needs to be declared according to the information required, for this thesis BOOL and FRAME. The Boolean variable are used to signal the start and stop of robot programs and welding while the FRAME variable can store robot poses. Additional, there are global variables which are already

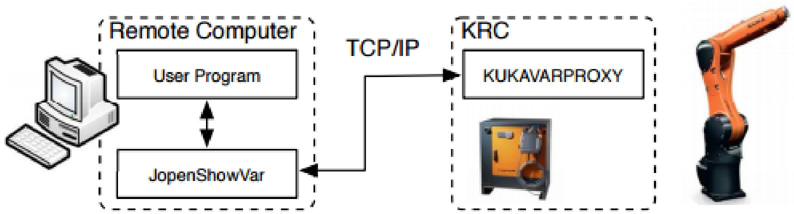

Figure 4.6: The client-server model architecture between the robot running KUKAVARPROXY and OpenShowVar. Figure from [14].

declared in the system for READ-ONLY purposes such as $POS_ACT and $AXIS_ACT. These variables are constantly updated and contains the joint configuration and pose of the robot. Below is a code snippet of a function from the Java application "ControlSystem" which takes as input a string *frameName*, double presicion variable position and orientation. The sting contents must be a FRAME variable declared in $CONFIG.DAT. An object takes as input the *frameName* and the position and orientation are written to the object. Using the *CrossComClient* class the object is send to the robot using *writeVariable* method.

```
public void writeFrame(String frameName, double X, double Y, double Z...
                                              , double A, double B, double C){
        KRLFrame frame = new KRLFrame(frameName);
        frame.setX(X);
        frame.setY(Y);
        frame.setZ(Z);
        frame.setA(A);
        frame.setB(B);
        frame.setC(C);
        try{
            this.connection.writeVariable(frame);
        }
        catch(Exception e){
            System.out.println("Error writing frame to Robot");
```

```
        }
    }
```

Not limited to setting up communication, the Java script is used to control the sequencing of every part of the operation for welding together two tubes. This including starting robot programs by sending Boolean signals to each of the robots, starting and stopping the weld, running the C++ application and Matlab Safety program described below 4.5 and it reads the two matrices from the C++ application and calculates the new PRY-angles using the equations described in 2.6.

4.5 Safety Program

The two transformation matrices found by the C++ application are written to two separate .txt files and then read by a safety application in Matlab. From the matrices the new RPY-angles and position correction for each robot are obtained. For safety reasons the two new poses are simulated and visualized using a safety application developed in Matlab. The Matlab application is developed using the *Robotic toolbox for Matlab* [10] by Peter Corke for plotting two generic robot using the *SerialLink* class that generates an object of a serial-link arm-type robot by taking the KUKA 120 Denavit-Hartenberg parameters as input. The class offers a method for checking collision between the robot object and a solid model which belongs to the class *CollisionModel* found in *physical Human-Robot Interaction Workspace Analysis, Research and Evaluation* (pHRIWARE) toolbox for solid object construction. Further, the application uses forward and inverse kinematics to compute joint configuration and poses while joint space trajectory planning is used to simulate the path between the current and wanted joint configuration. The robot kinematics used are described in chapter 2.

The application is developed to protect the robots and its environment against collisions if the poses obtained are erroneously calculated. The Matlab application tracks the robot configuration from the offline programmed robot programs using the known joint configurations for when the tubes are picked up and for holding the tubes in front of the camera. The following motion for each robot are simulated in Matlab by reading the two transformation matrices to be able to weld together the two tubes. Inverse kinematics are used on the transformation matrices to obtain the wanted joint configurations q_{RIGHT} and q_{LEFT} and joint space trajectory between the current configurations and the wanted configurations are computed to estimate the robot paths. When the wanted pose is visualized two cylinders are added to check if they collide. The length of the cylinders are calculated when they are picked up by reading the Z-values of the end-effectors at contact. Before these motions are executed on the robots, the user have to verify if the trajectory and wanted pose for each robot including the attached cylinders do not crash. If the cylinders crash with either the other cylinder or robot it will be colored red for transparency. This application is not only useful to see if they crash or not, but also to visualize where the two cylinder will be held. If they are held at poses obviously not suitable for welding the operator can chose to cancel the welding operation. To summarize the algorithm in the application:

1. The joint configurations when holding the tubes in front of the Kinect is known because they are programmed using 3DAutomate.

2. The wanted transformation matrices are given to the Matlab application from the C++ application.

3. Inverse kinematics are utilized to obtain the wanted joint configuration for each robot.

4. Compute a joint space trajectory between current and wanted configurations.

5. Move the end-effector of each robot according to the trajectory computed in (4).

6. Add two cylinders located at the grippers for each robot with the actual length of the cylinders picked up.

7. The program checks any collisions have occurred and colors the crashing cylinder red if so.

8. The user is prompted if the wanted poses are OK for welding or not. If they are OK, the user tells the program to continue. IF not, no further motions are executed.

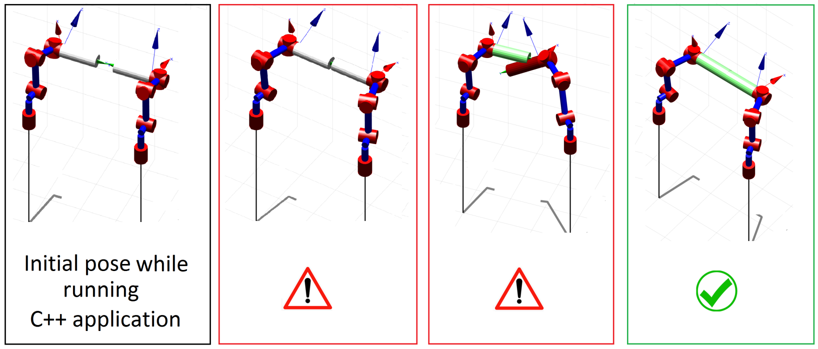

Figure 4.7: The first picture on the left shows the pose for both robots when the C++ application is running. The two picture in the middle shows configurations not satisfying for welding, while the picture on the right side is good for welding.

Figure 4.7 shows the initial pose during the run-time of the C++ application and three examples of the robots being moved to the poses given from the C++ application. It is clear to see that the two in the middle are not acceptable since the spacing is to big for welding and the other collide. On the right side the cylinders are suitable for welding and the welding process can start.

4.6 Robotic welding of cylindrical objects

The KUKA KR16-2 is attached with a Fronius TransSteel 5000 welding machine feeding a welding rod with thickness of 1.0 mm, see figure 4.9. Figure 4.8 shows how the two KUKA 120 robots hold their tubes against each other and how the KUKA KR16-2 is used to weld them together. The robot programming for the KUKA KR16-2 and welding parameters was developed and tested in the project thesis, but for the reader to fully understand this thesis the results are presented next.

Table 4.2: Fronius TransSteel 5000 on KUKA robot description for figure 4.9.

	Description
1	Power source
2	Wire feeder
3	Adapter Flange
4	Collision Box
5	Weld Torch

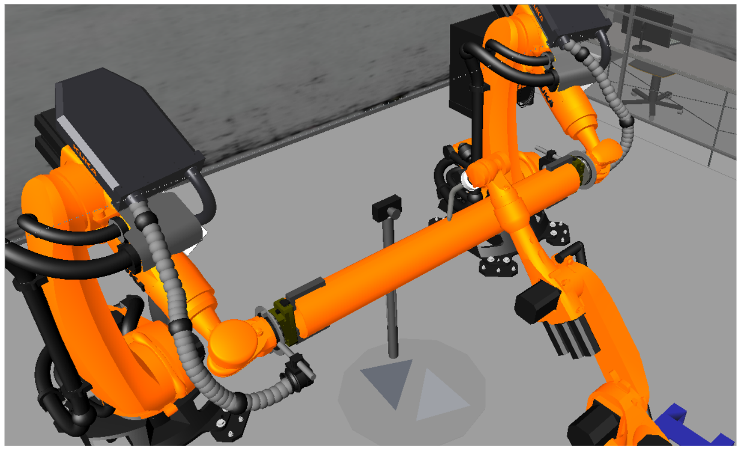

Figure 4.8: The tubes are handled by the two KUKA KR120 robots and welded together by the KUKA KR16-2 with its attached welding gun. The position of the Kinect is also shown.

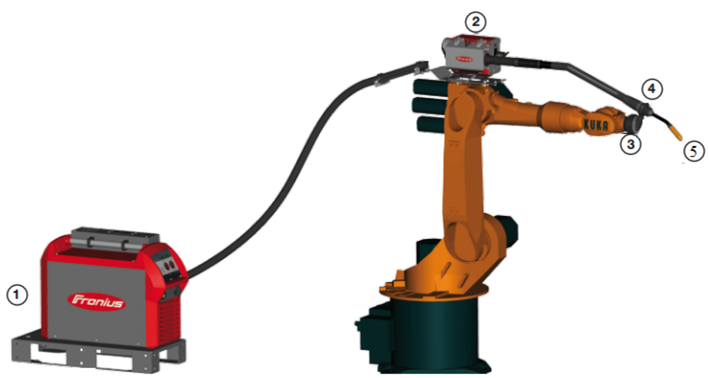

Figure 4.9: Fronius TransSteel 5000 welding machine connected with a KUKA robot. Figure from [5]

4.6.1 Welding Programming and Parameters

The two tubes must be fitted up together in such a way that they can be welded together. For two tubes with wall thickness less than 6mm, the internal misalignment can not exceed 25% of its wall thickness [4]. This thesis uses tubes with wall thickness of 5mm giving a misalignment tolerance of 1.25mm. Further, the root opening between the two tubes should be 1.6mm.

Given that the tubes are fitted up correctly using the Kinect for correction the first operation for the welding robot is to tack weld the tubes together. Tack welding is a temporary weld used to create the initial joint between two pieces of metal being welded together. The paths for tack

Table 4.3: The parameters for figure 4.10

Parameter	Description	Dimention
t	Wall-thickness	5mm
A	Root opening	1.6mm

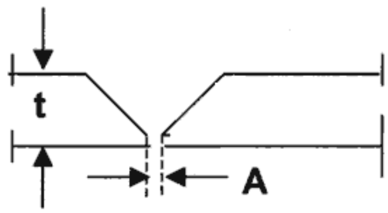

Figure 4.10: Weld joint parameter. Figure from [13]

welding a 1.9cm long seam on four points evenly spaced around the tube was developed in 3DAutomate using a Python script to create several linear points lying on the natural curvature defined by the radius of the tubes. Since the goal of the thesis focuses on the welding preparation of fitting up the tubes correctly the 360° welding of the tubes is left out.

In the C++ application the wanted position for the two tubes are defined according to the coordinate system of the camera and the positioning of the camera in the environment. Meaning that the position of the tubes for this master project differs from the project thesis. To be able to use the old robot programs the base of the KUKA KR 16-2 is translated the same amount as the tubes. To find the translation needed a simple method was executed using the old robot program for the top tack weldment. First contact between the welding rod and the tubes for the old position had the coordinate values x_{old},y_{old} and z_{old} and marked on the tubes. The tubes were placed at the wanted position for this thesis and the welding robot was moved by jogging it with constant orientation until the welding rod touched the marked spot. This new robot position is denoted as x_{new},y_{new} and z_{new}. The translation of base t_{base} is defined in equation 4.1.

$$t_{base} = \begin{Bmatrix} x_{new} - x_{old} \\ y_{new} - y_{old} \\ z_{new} - z_{old} \end{Bmatrix} \tag{4.1}$$

4.7 Robot programs

In Automatic mode the main program in the each robot runs a while-loop constantly checking Boolean values, true and false. The Boolean values are declared as global variables and can be read and written to from a remote computer. Every Boolean value are by default set to FALSE, but when a value is changed to TRUE from the remote computer the loop pauses and the wanted sub-program is executed. When the given sub-program has ended, the main loops continues and waits for further operations. The utilization of Boolean values makes it possible to sequence the sub-programs in a wanted order. The communication architecture and pseudo code to start the sub-program denoted as *ONE* is described in figure 4.11.

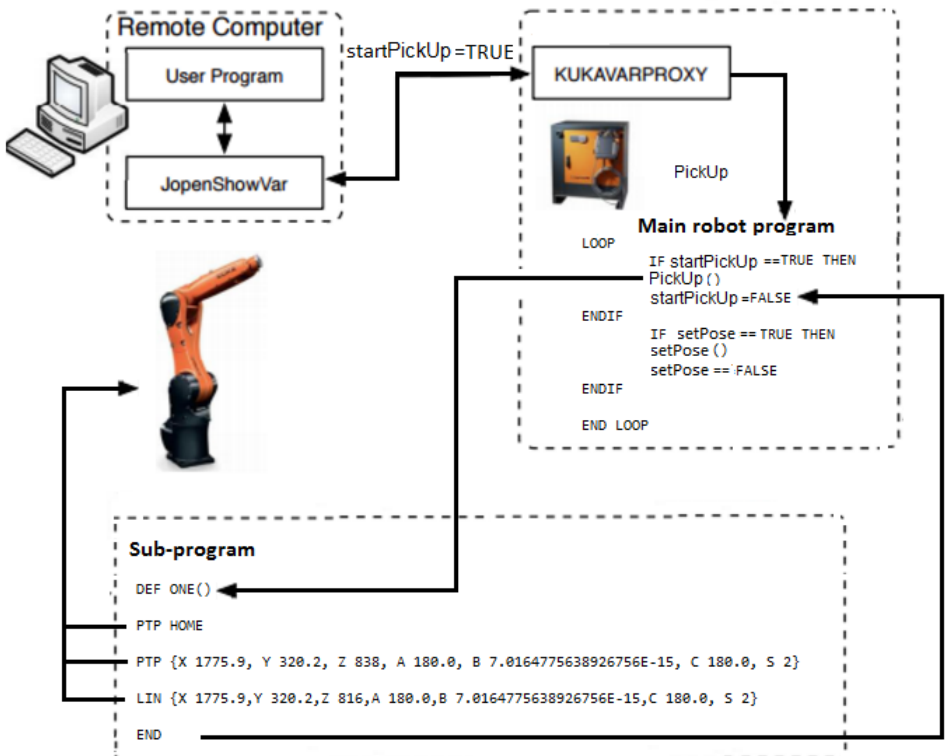

Figure 4.11: Communication architecture and pseudo code of how a sub-program with robotic motions are started from a Remote Computer.

To set a new pose the java application writes a global variable of type FRAME which can store position and orientation, this is shown in the code snippet in 4.4. When the new pose is written a sub-program is executed which reads the FRAME and moves the robot to this pose in a linear motion. Figure 4.12 shows how the Boolean variable *setPose* is set to TRUE and the sub-program called setPose is started and how it reads the values in the FRAME variable.

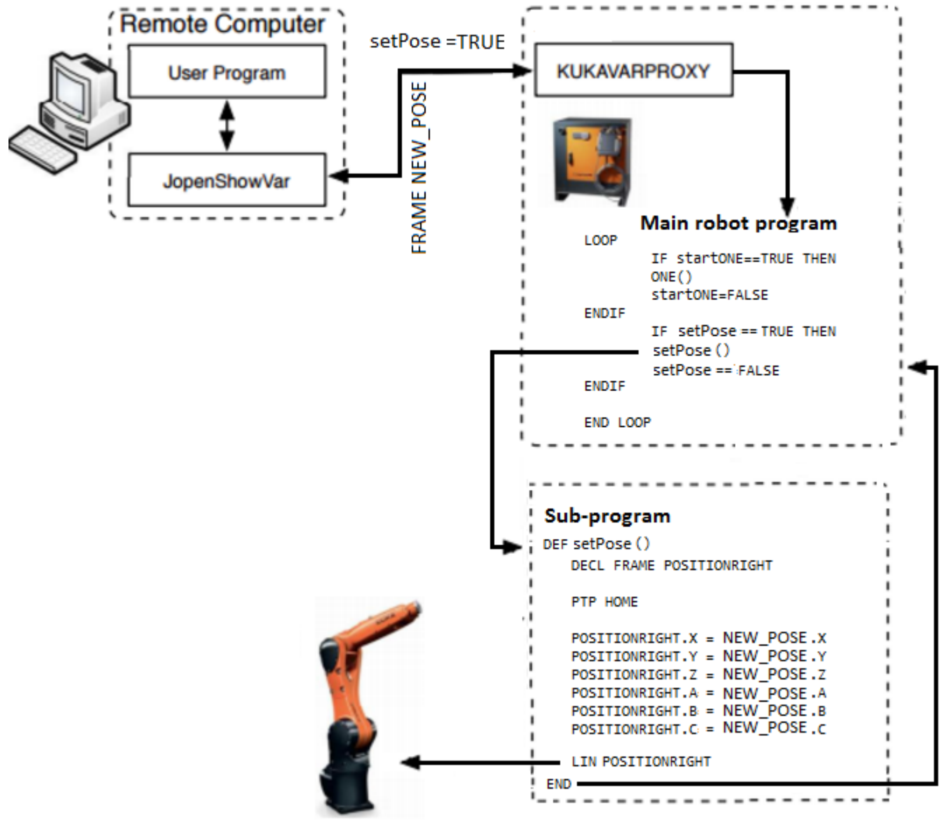

Figure 4.12: Communication architecture and pseudo code of how a new pose declared in a FRAME type is passed to the controller and how the sub-program reads the FRAME and moves to this pose in a linear motion.

4.7.1 Force Control

The tubes used in this thesis are of a unknown length making the force sensor attached to the robot end-effector very useful. Instead of re-programming the robot program for picking up the tubes for varying length the approach is controlled by a force control application to prevent collision. Also, when setting the tubes against each other force control is implemented to protect the robots against the uncertainty from the Kinect and C++ application for alignment. The KUKA.ForceTorqueControl 3.0 is an add-on technology package which together with the multi-axis force and torque sensor "ATI OMEGA 160" sensor can simultaneously measures forces F_x, F_y and F_z and torques T_x, T_y and T_z [2], see figure 4.13. The value of forces and torques are obtained using silicon strain guages by measuring the voltage running through them. When the guages are strained the electrical conductor becomes narrower and longer which decrease its electrical resistance and voltage.

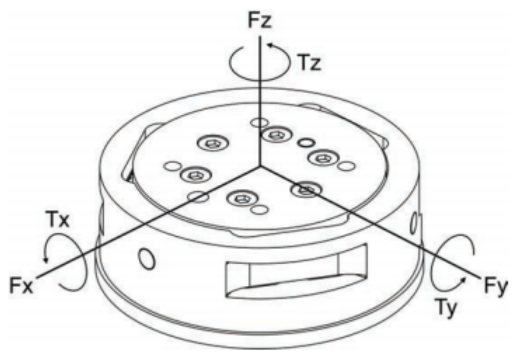

Figure 4.13: The ATI omega 160 force/torque sensor placed on the robot end-effector measures forces F_x, F_y, and F_z and torques T_x, T_y, and T_z.

The force control is implemented as an application on the teach-pendant where the wanted option is to preform a "sensor-guided: make contact" operation. The application is called by the robot program when the end-effector is about to make contact with another object in the environment. The predefined approach speed, main direction of the resistance force and the set-point force, in which the robot stops when reached is all defined in the application options. In the case of picking up the tubes from the floor, the main force direction is in the z-axis of the world coordinate frame while in the y-axis for placing the tubes together. The set-point force for both cases is set to 50N and the approach speed 0.01m/s.

4.8 Architecture of the process

The above sections describes the different applications and programs used in the correction process of making it possible to weld two cylindrical object with a unknown run-out and length together. The process to achieve this is presented in figure 4.14 showing how the C++ application, Java application, Matlab application collaborates with the Kinect sensor, pre-made .src files and human operator to execute a correction of the cylinders pose making it possible to weld them together. Table 4.4 lists the different tasks for each application. The solution provides a graphical user interface showing a robot control table, robot simulation from the Safety

Table 4.4: Tasks for each application.

Application:	**Tasks:**
Java	Sequence robot programs Calculate RPY-angles Calculate new pose Read and write variables to robots Start C++ application Start Matlab application Control welding ON/OFF Prompt user for OK poses
Matlab	Visualize robot movements Simulate new pose Check for collision
C++	Visualization of 3D image Visualization of wanted and current point cloud Calculate correction transformation matrices
Force/torque control	Prevent large forces/torques on robot
3DAutomate	Create robot program for welding robot and handling robots

Program and a stream including point clouds representing the uncorrected pose and the point clouds transformed to the wanted pose. This is shown in figure 4.15.

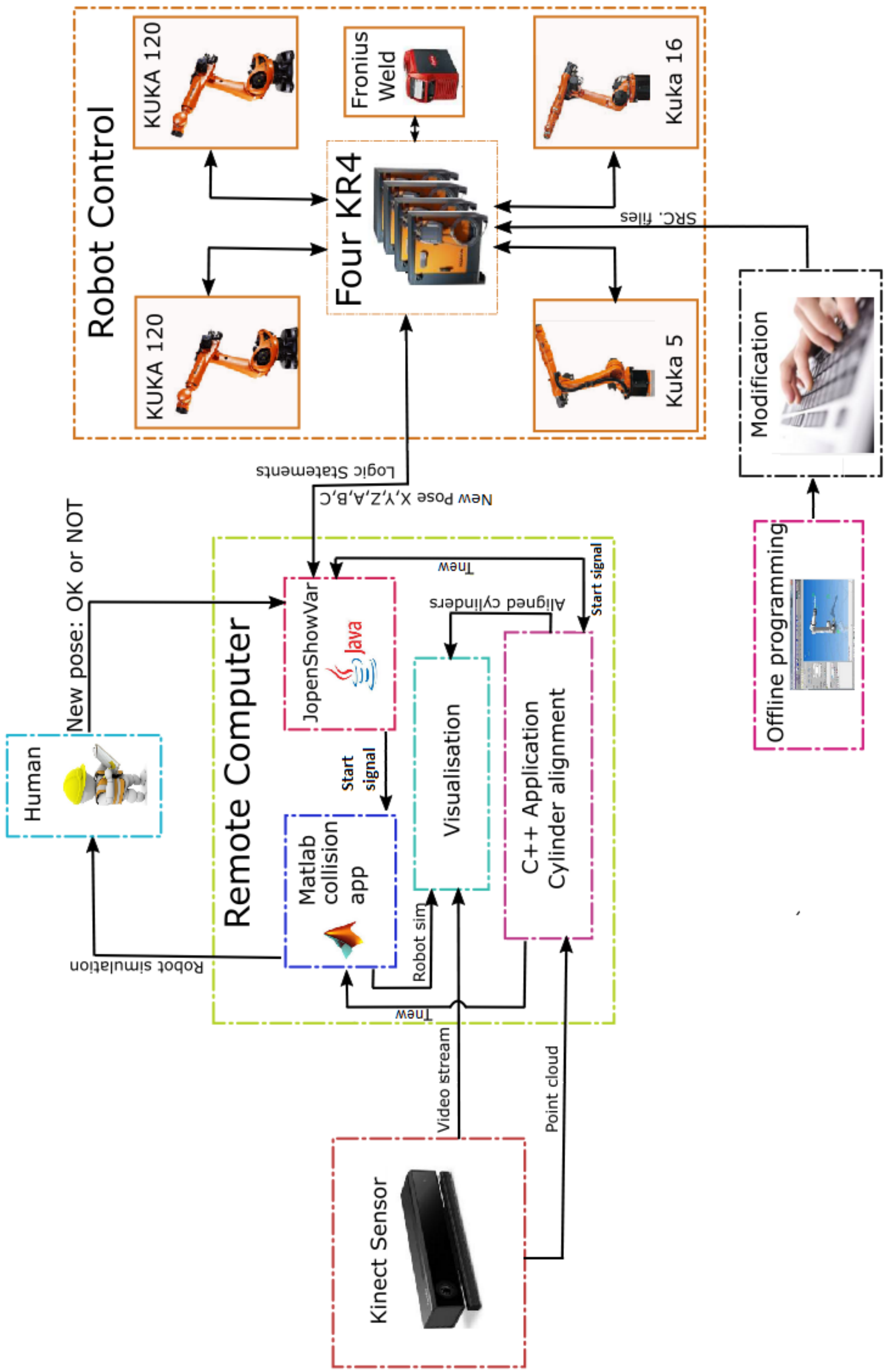

Figure 4.14: Architecture of the process of obtaining a sufficiently good fit-up for two cylindrical object to be able to weld them together.

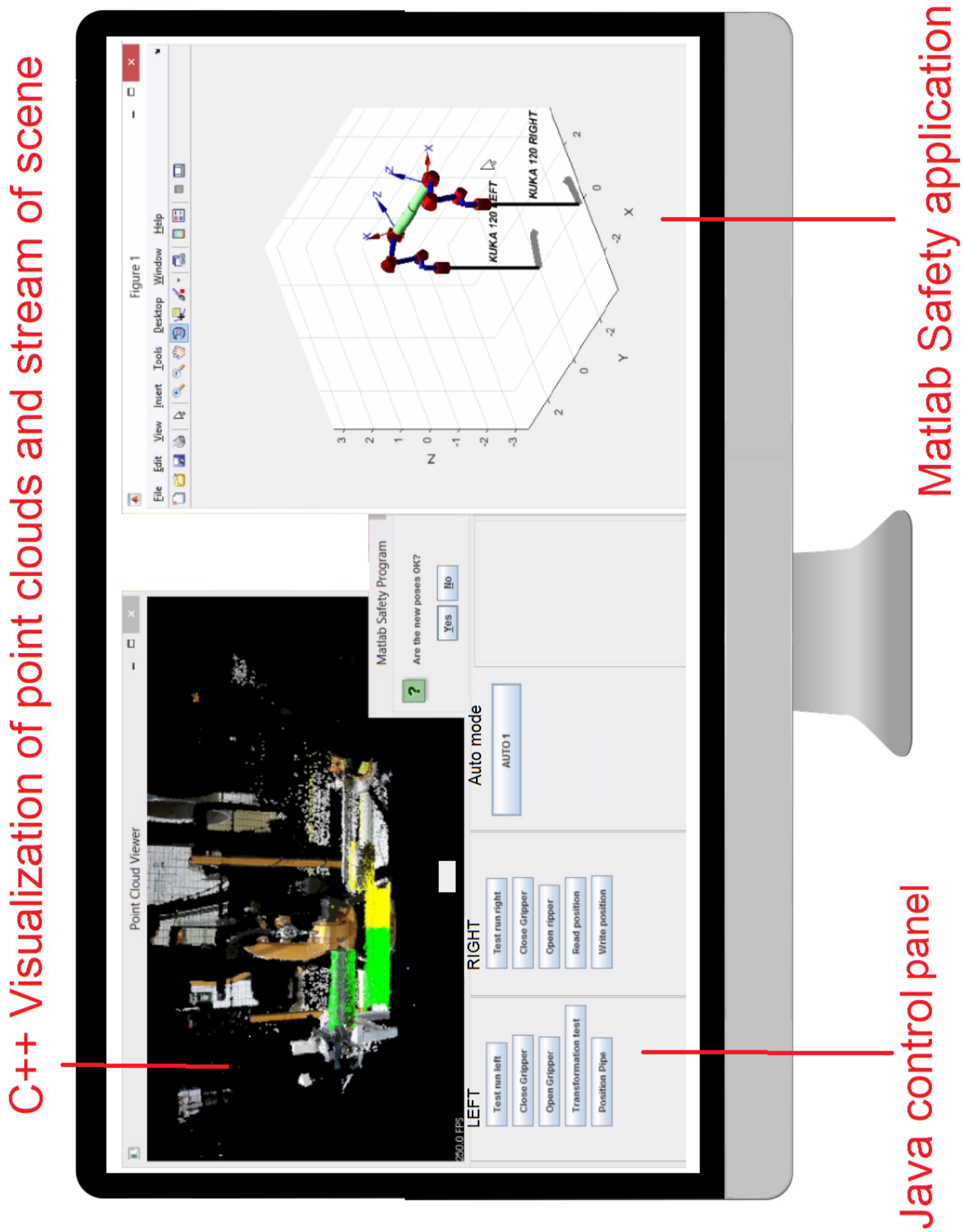

Figure 4.15: The graphical user interface provided to the operator. Java control used to control robots and auto operations, Matlab safety application for simulation and collision testing and a visual stream of the scene including the point clouds representing current and wanted pose.

Chapter 5

Results

5.1 Run-Out

The origin of this master thesis is based on the run-out of industrial steel tubes making them impossible to fit-up by offline programmed robot motions without any feedback. To test for run-out a dial gauge measured the translation in z-axis of the robot while the tube was rotated 360°. Figure 5.1 shows the setup for circular radial run-out measurement and the resulting translation in mm is presented in figure 5.2. The results proved a difference between minimum and maximum z-value to be 4.56mm.

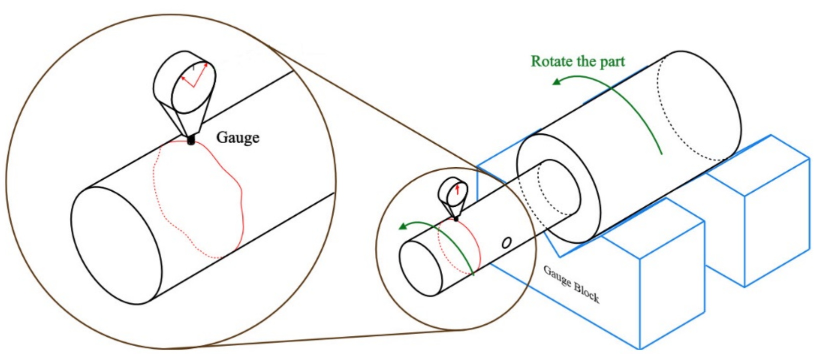

Figure 5.1: The setup of how a dial gauge was utilized to measure the translation of the tube in the z-axis of the robot.

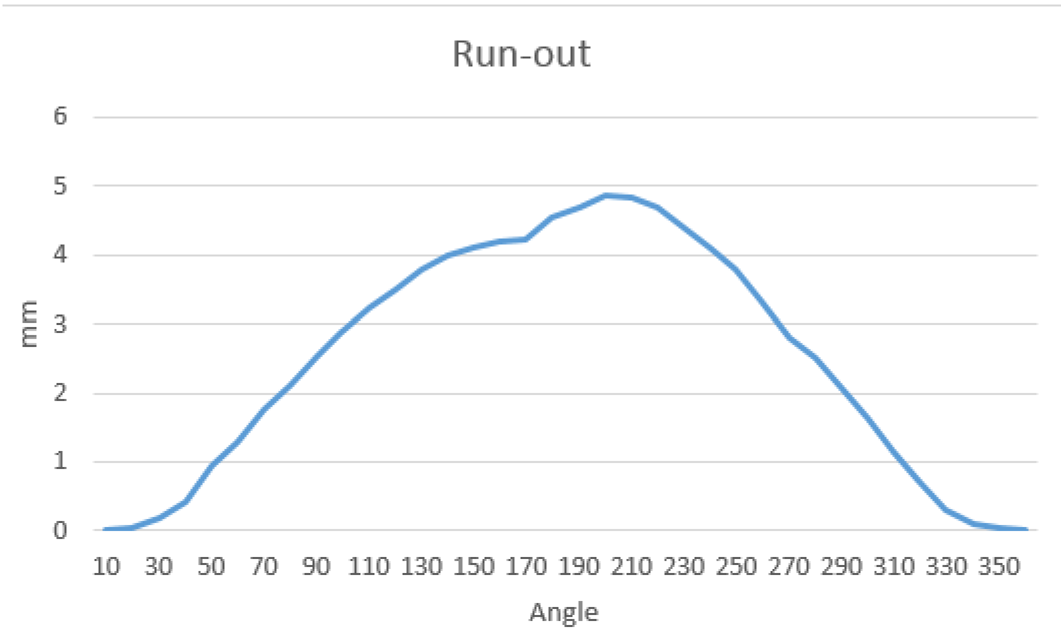

Figure 5.2: x-axis showing the angle of rotation, while the y-axis represents the translation in mm.

Figure 5.3: With colinear approach axis the two tubes exceeding the allowed tolerance for fit-up before welding.

5.1.1 Fit-up without alignment

When the two approach axes for each of the end-effectors are collinear the resulting run-out will produce a fit-up which exceeds the allowed misalignment error of 1.25mm. The graph in 5.2 shows that the tube is translated 4.56mm from the starting point when rotated to 210 °. This results in a fit-up greatly exceeding the tolerance of 1.25mm and is not suitable for welding, see figure 5.3.

5.2 Alignment results

This section will cover the alignment results for testing SAC-IA, ICP, RANSAC and Search Method to aligning cylindrical objects.

5.2.1 Alignment with SAC-IA and ICP

The results of an alignment algorithm can be determined by three factors, namely the robustness, quality and time consummation. For this thesis computation time used by an algorithm is not of interest and will not be discussed further. For the SAC-IA and ICP a fitness score describes the quality of the alignment by obtaining the sum of squared distances between corresponding points in the transformed cloud and the target. The robustness is obtained by looking at the deviation of the quality over many samples.

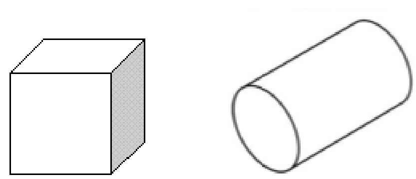

Figure 5.4: The box and cylinder used as test objects for the SAC-IA and ICP alignments algorithm.

For testing SAC-IA and ICP two objects were used, a cubic box and a cylindrical tube as the one shown in figure 5.4. The cubic box was designed in *SolidWorks* and imported to the software *CloudCompare* which can sample points on a mesh into a Point Cloud Data (PCD) file. As explained in 3.4 the cylinder was modeled mathematically in C++.

Cubic box alignment

There is no correlation between the box alignment testing and welding of tubes, but the test was executed to see if the SAC-IA and ICP alignment worked correctly or not. Running a series of test where a box was located at different positions and orientations relative to the camera concluded that the combination of SAC-IA and ICP managed to align a box captured with the Kinect and align it with high quality to the target box model. SAC-IA is often used as an initial alignment tool and the ICP used for fine adjustment. Due to the box geometrical features including corners and edges the SAC-IA becomes very efficient and precise because the alignment is based on corresponding feature points. Applying the ICP algorithm using the SAC-IA alignment as an initial guess resulted in a transformation matrix closely to a identity matrix meaning that almost no rotation or translation was executed by the ICP alignment. This method of aligning cubic boxes proved to be both robust and with high quality. Figure 5.5 shows the result of aligning a box in a noisy point cloud with a target box.

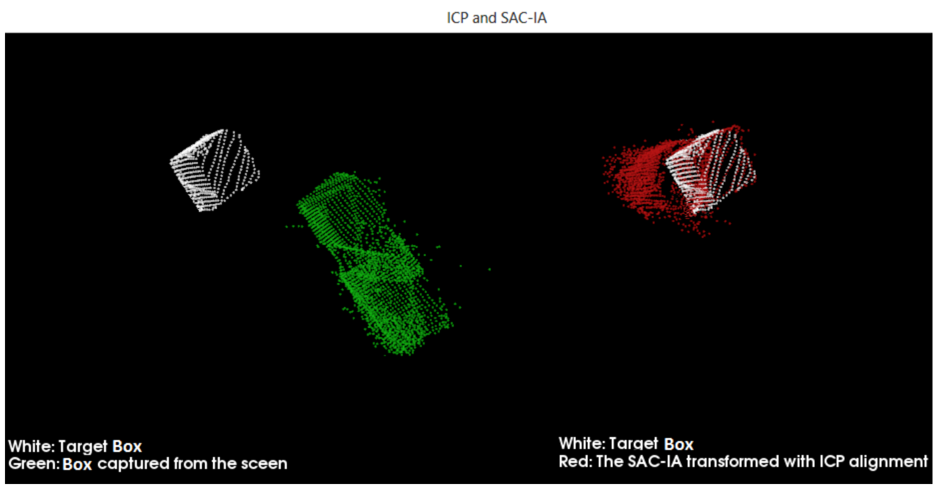

Figure 5.5: A noisy point cloud including a box is aligned with a target box using SAC-IA and ICP.

Cylindrical tube alignment

The combination of SAC-IA and ICP resulted in a highly accurate alignment for a cubic box and the reason why this was tested was because the two algorithms had problems with aligning cylindrical object. The geometrical feature descriptors for a straight cylinder is highly homogeneous and similar throughout the length of the tube making the process of matching corresponding features "confusing" for the lack of a better word. Also, the fitness score which describes the quality of the alignment by the means of the squared distances between corresponding points is misleading. Figure 5.6 shows a point cloud which have been processed and aligned twice by the ICP algorithm. The fitness score for both cases are the same, but the alignments are not. This together with a varying initial alignment by the SAC-IA resulted in a transformation not possible to use for aligning two cylindrical objects for welding because the positioning along the length of the cylinder varies to much.

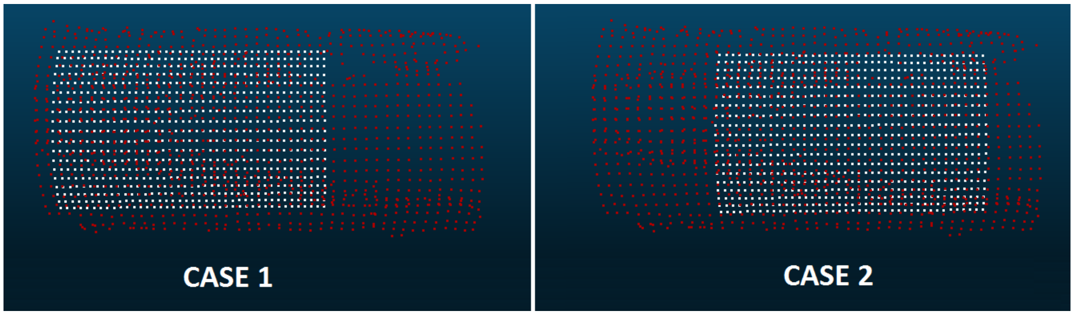

Figure 5.6: Two different alignments done by ICP with the same fitness score.

A series of 10 tests were conducted on a cylindrical Polyvinylklorid (PVC) plastic tube with no run-out to test orientation and translation accuracy. The tube was held by the right robot in such a way that no rotation were needed to align the tubes, hence all the correction RPY-angles from the alignment should equal 0°. One of the results is shown in figure 5.7 where the translation along the length of the cylinder is inaccurate. The white point cloud is the target cylinder while the blue is the initial guess done by SAC-IA and the red point cloud is the final alignment by ICP. The results for the 10 tests are presented in table 5.1.

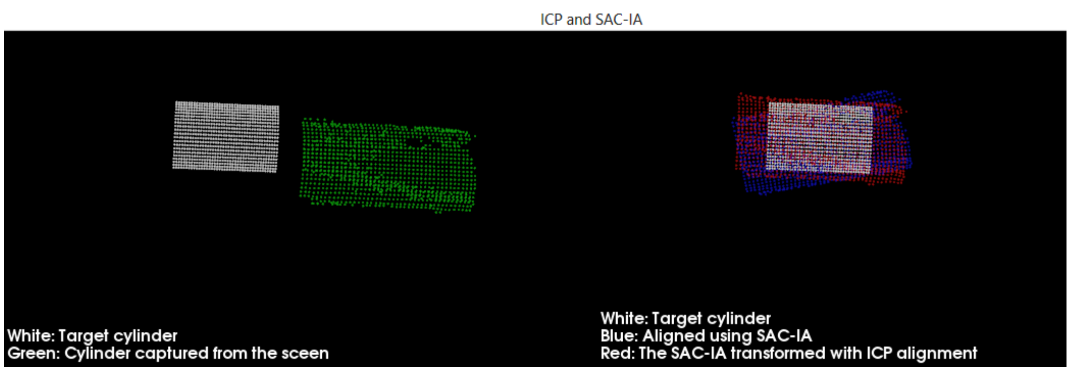

Figure 5.7: The green point cloud represents the cylinder held by the robot. Blue is the alignment done by SAC-IA and the red point cloud is the alignment by ICP.

From table 5.1 one can observe that the deviation along the Y-axis in the robot frame is ±27.1mm which will cause the two tubes to either crash or be to far apart for welding. Further, the deviation along the X-axis is also to big, resulting in a misalignment greater than the specified 1.25mm.

For orientation the mean value, mean error angle, about the Z-axis and X-axis of the robot frame, given as A and C have the values of 1.67° and 1.21° respectively. The error in orientation

Table 5.1: Aligning results using SAC-IA and ICP.

Pose / Test #	x[mm]	y[mm]	z[mm]	A[Degrees]	C[Degrees]
Test1:	3.10	11.11	-0.93	2.10°	1.30°
Test2:	1.71	-36.95	-0.30	2.06°	0.76°
Test3:	-1.33	11.73	-0.41	2.20°	0.60°
Test4:	-1.64	10.28	0.76	2.19°	1.10°
Test5:	-0.71	1.70	0.46	3.10°	0.85°
Test6:	-0.90	28.36	-0.82	0.10°	1.60°
Test7:	-0.66	11.50	-0.19	1.40°	1.50°
Test8:	-1.37	12.01	-0.14	0.80°	1.50°
Test9:	-0.60	-4.77	1.51	0.20°	2.40°
Test10:	3.58	-62.51	0.61	2.50°	0.46°
Results:					
Mean Value:	0.12	-1.75	0.12	1.67°	0.95°
Standard Deviation:	1.83	25.91	0.73	1.61°	0.55°

will result in either a gap or a welding seam not going along the root opening, see figure 5.8. The gap can be expressed by equation 5.1 where D is the diameter of the tubes and ϕ and θ are error angles of alignment for the right robot and left robot. The error angles in the equation are either for error orientation about the Z-axis or X-axis.

$$\Delta\varepsilon = D(sin(\phi) + sin(\theta)) \tag{5.1}$$

Using this equation and the values of the mean error of 1.67° and 1.22° with a diameter of 168mm gives gaps of 9.79mm and 7.02mm respectively. Both values will give gapes not possible to weld because they are bigger than the root opening of 1.6mm. The deviation in both position and orientation clearly exceeds misalignment and orientation limit which makes SAC-IA and ICP not suitable for aligning two tubes for welding.

5.2.2 Rotation using RANSAC

As explained in 3.5.3 the RANSAC algorithm for cylinder parameter estimation can be used to obtain the center axis for a cylinder found in the scene captured by the Kinect. To test the performance of the RANSAC algorithm the same test as for the SAC-IA and ICP were executed. A series of 10 test on a cylinder with no run-out was orientated in such a way that there should be

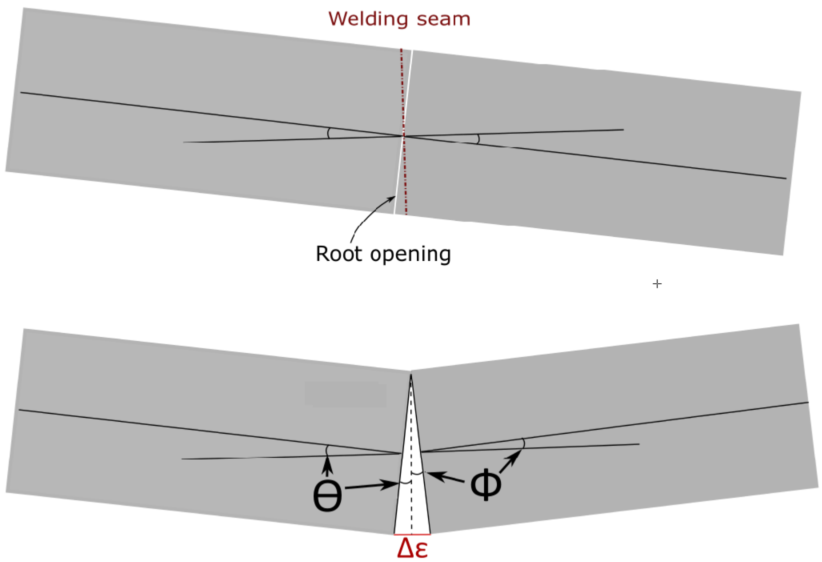

Figure 5.8: Fit-up results for alignment with orientation deviation.

no change in the RPY-angles to align the cylinder. The results are represented in table 5.2.

Table 5.2: Orientation results for RANSAC.

Orientation / Test #	A[Degrees]	C[Degrees]
Test1:	1.82°	1.29°
Test2:	1.81°	1.65°
Test3:	1.61°	1.72°
Test4:	1.29°	1.04°
Test5:	1.05°	1.78°
Test6:	1.81°	1.11°
Test7:	1.75°	1.23°
Test8:	1.81°	1.73°
Test9:	1.31°	1.25°
Test10:	1.21°	1.05°
Results:		
Mean Value:	1.54°	1.38°
Standard Deviation:	0.28°	0.28°

The RANSAC algorithm is a orientation alignment only with no translation calculation. It has a mean error angle of 1.54° and 1.38° for A and C respectively. The reason for the orientation error is because of the mathematically model given in 3.5.3 which is used to define a cylinder and the fact that the data from the Kinect is noisy. When estimating the center axis for a cylinder the normal of the transversal plane defined by three randomly chosen points are used. Since these points are randomly selected from points lying on the surface of the cylinder it is almost impossible to locate three points which are actually located on the transversal plane to the original cylinder. For this reason the normal, center axis, is always a bit off the actual center axis [15] which gives deviating results. Given the results above, the RANSAC algorithm provides a standard deviation of 0.28° for both A and C, see figure 5.9, which makes it more accurate than the SAC-IA and ICP for orientation. Still, it do not provide for a sufficiently good alignment for welding because the gaps calculated with equation 5.1 to be 9.02mm and 8.09mm using the mean angle error of 1.54° and 1.38° respectively.

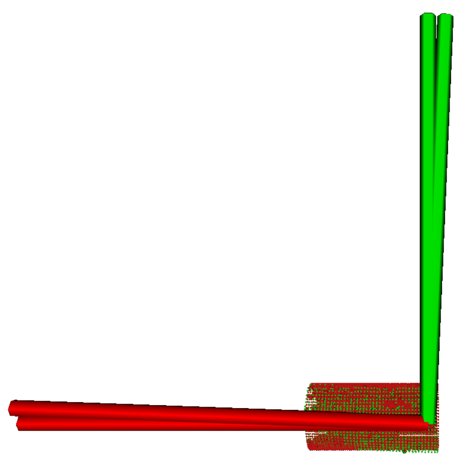

Figure 5.9: The orientation error of 0.28° to align coordinates system are shown by aligning the green point cloud with the target red point cloud.

5.2.3 Aligning position using Search Method

In the preparation prior to welding the tubes are processed by turning to obtain two ends which are perfectly parallel. This ensures that the root opening distance is homogeneous throughout the fit-up. This means that the resulting run-out can be fixed by translation only. After ICP proved to generate inaccurate translational results an algorithm denoted *Search Method* was developed to search for a specific point on the edge of cylindrical object and generate a translation between this point and a wanted point.

Described in 3.6 the algorithm search for a set of data points used to estimate M_{edge} by finding the points with lowest z-values, being closest to the camera, within the boundaries of interval N_i. The y and z-values for each interval are presented in figure 5.10 which shows very noisy data.

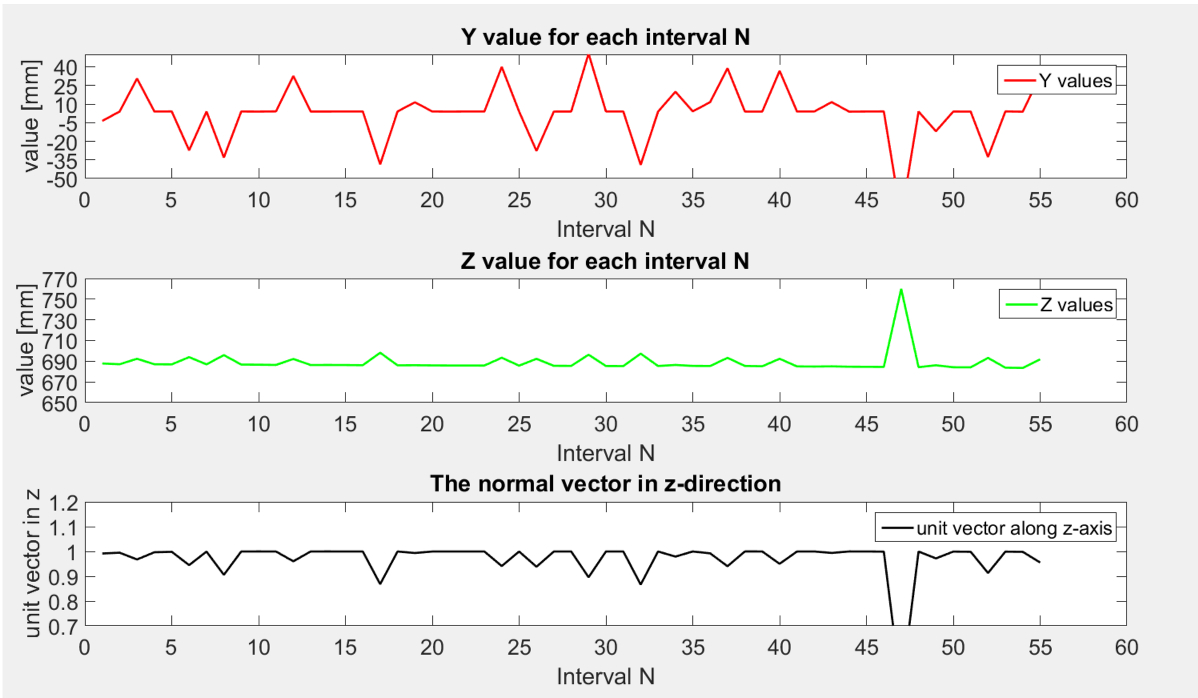

Figure 5.10: The y and z values of the point being closest to the camera within the boundaries of interval N_i. Also the z component of the unit normal vector is presnted in the bottom graph.

Depth variations in the Kinect results in varying y and z-value because the wanted point can be wrongfully measured to be further away then its surrounding neighbors, resulting in either the point above or below can have a smaller z-value than the wanted point. This is illustrated in figure 5.11 where an unwanted point is selected because it has lower z-values than the wanted point. The unwanted point still has a higher z-value than the rest of the intervals and this can be seen by the small spikes in the middle graph in figure 5.10. The increase in z-value from the mean results in that a unwanted point is selected which have either lower or higher y-values than the mean. The correlation between the small spikes in z-values and the y-values is clearly shown in figure 5.10. Negative spikes means that the point below has been captured, while positive is points above. Further, all the points being represented by spikes in the y and z graph will have normal vectors deviating from (0,0,1) which again will give spikes when plotting the z value of the normal vector shown in the bottom graph in 5.10. Figure 5.11 illustrates how a wrongfully captured point gives spikes in y,z and normal value.

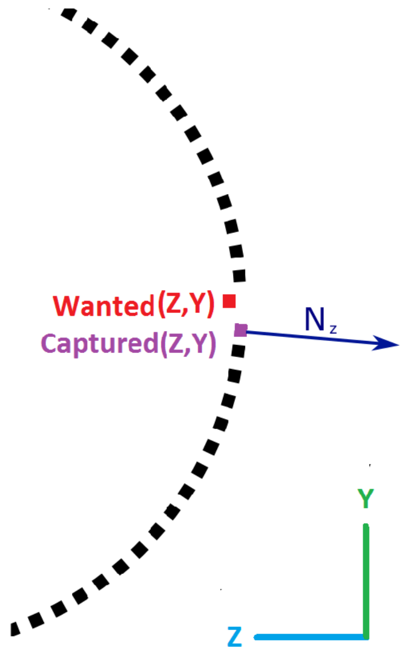

Figure 5.11: The wanted point is not captured because of depth inaccuracy in the Kinect. This results in a y,z and normal vector value which is not desirable for estimating M_{edge}

Filtering these results by removing any points having a normal vector in z-direction below 0.99 reduces the data from 55 points to 13. The remaining points are further filtered by *THE MODIFIED Z-SCORE* which removes potential outliers. The results after normal and outlier filtering are presented by the graphs in figure 5.12.

The mean of the remaining point will be used to estimate the y and z-values for M_{edge}. The estimated x-value is the mean of the points within the first interval N_i which contain more than 10 points. Because of noise near the edge the first intervals often contain less than 10 points and do not represents the edge of the cylinder.

The algorithm was tested by placing the two tube with unknown run-out at 10 different positions in front of the camera and by estimating M_{edge} it calculated the correction position values needed to fit-up the two tubes. Figure 5.13 visualises the results for one of the tests showing how two tubes apart is translated to a consisted fit-up. For these tests the perfect position with the corresponding run-out was found to be at robot position $p_{RIGHT}(976.19, 924.36, 1422.81)$ and $p_{LEFT}(1002.27, -1044.16, 1416.88)$. The results given in table 5.3 presents the deviation between the new position computed by *Search Method* and the perfect position p_{LEFT} and p_{RIGHT}. Also, the table present the internal misalignment for the fit-up due to the position deviation by com-

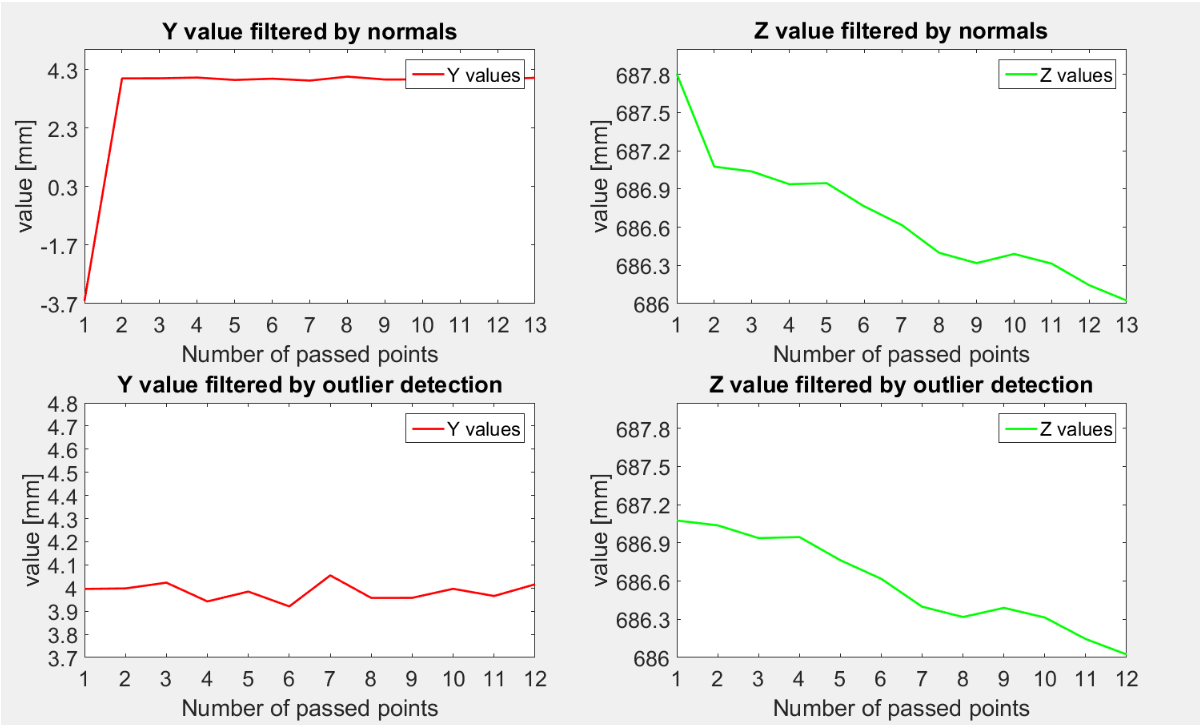

Figure 5.12: The wanted point is not captured because of depth noise in the Kinect. This results in a y,z and normal vector value which is not desirable for estimating M_{edge}.

puting the distance between the two deviating points in the XZ-plane using equation 5.2.

$$d = \sqrt{(x_{right} - x_{left})^2 + (z_{right} - z_{left})^2} \tag{5.2}$$

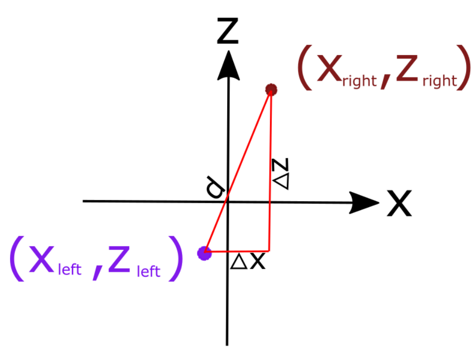

Figure 5.13: The distance (internal misalignment) between two point in the XZ-plane.

All cells in the last column in table 5.3 are marked by green, meaning that they all have an in-

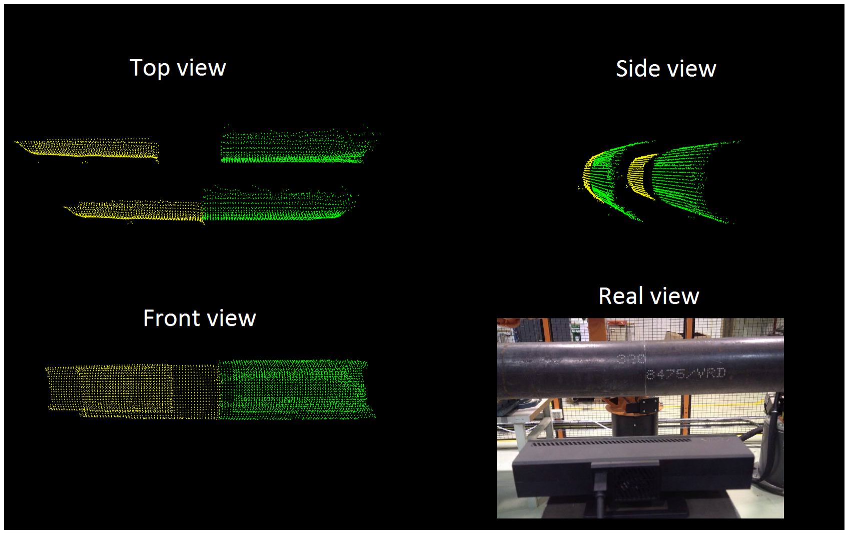

Figure 5.14: The Search Method translation of two tubes in three different view points. The resulting fit-up is presented in the lower right corner.

Table 5.3: Results using Search Method to align position for the two robots.

Position / Test #	Right Robot x[mm]	Right Robot y[mm]	Right Robot z[mm]	Left Robot x[mm]	Left Robot y[mm]	Left Robot z[mm]	Internal misalignment between the tubes held by two robots
Test1:	0.30	-0.18	0.15	0.37	0.10	-0.08	0.24
Test2:	0.32	-0.63	0.02	-0.18	0.28	-0.43	0.67
Test3:	-0.21	0.04	0.05	-0.45	0.02	0.01	0.24
Test4:	-0.40	0.18	-0.01	0.08	-0.06	-0.02	0.47
Test5:	-0.21	0.23	-0.12	-0.18	0.17	-0.12	0.02
Test6:	-0.55	-0.15	0.04	0.09	-0.05	-0.22	0.69
Test7:	0.26	0.18	0.08	-0.46	-0.12	-0.06	0.73
Test8:	0.49	0.20	0.00	-0.21	0.27	-0.18	0.72
Test9:	-0.31	0.21	-0.03	-0.02	-0.17	0.08	0.31
Test10:	0.30	-0.03	0.00	0.08	0.10	0.58	0.63
Results:							
Mean Value:	0.03	-0.02	0.03	-0.09	0.05	-0.05	0.49
Standard Deviation:	0.35	0.26	0.07	0.24	0.16	0.26	0.25

ternal misalignment less than 1.25mm which satisfy the requirement for fit-up for welding. The mean value for the internal misalignment was obtained to be 0.49mm with a standard deviation of 0.25mm which by the means of the samples taken indicates that the method is very accurate and robust. Further, with a mean value and deviation in y-direction of only -0.02±0.26mm and

0.05±0.16mm for the two robots it clearly does not exceed the root opening of 1.6mm. Figure 5.14 shows two captured point clouds representing the left and right cylinder at start position and the translated position.

5.2.4 Using the rotation from SAC-IA/ICP or RANSAC together with Search Method

As explained in subsection 5.2.1 the SAC-IA/ICP alignment had a varying value along the length of the tube and the RANSAC rotation alignment described in subsection 5.2.2 has no translation. To obtain a complete as possible alignment the Search Method was implemented together with the SAC-IA/ICP and RANSAC to align tubes not only having a run-out, but also if the tubes are tilted by the robots. Because of the orientation deviation for both RANSAC and SAC-IA/ICP the resulting fit-up was either oblique or it had gaps not suitable for welding.

Figure 5.15 shows the visualization of how two point clouds representing two tubes are first by SAC-IA and ICP tried to be aligned with two target cylinders and then with translation correction using Search Method. Left side in the figure represents the SAC-IA and ICP alignment while the right side Search method is implemented. Even though the alignment might seem good, it was proven in 5.2.1 that the orientation deviation for SAC-IA and ICP is to big for welding. Figure 5.16 shows the actual result of the alignment using SAC-IA and ICP.

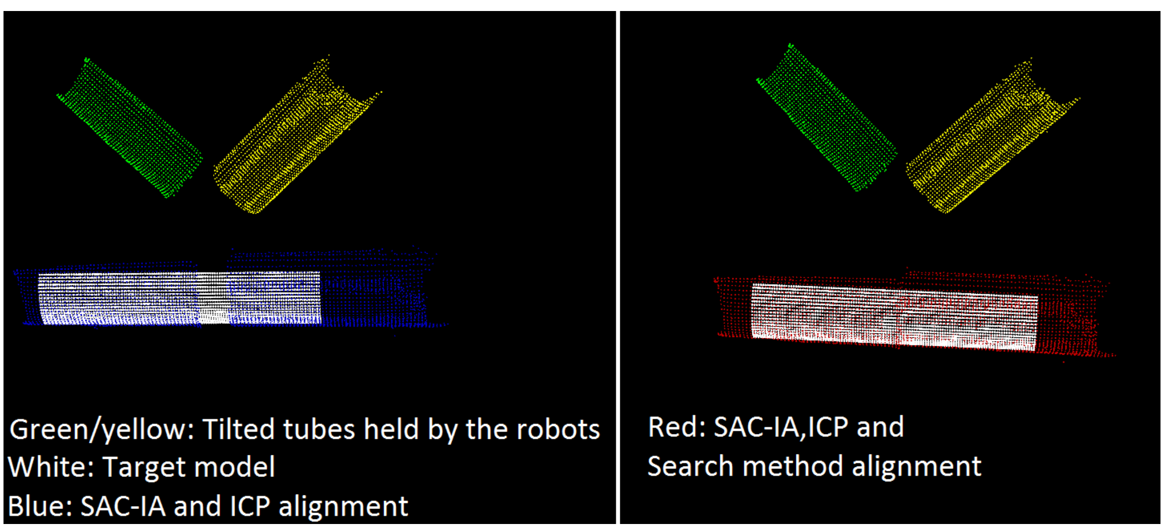

Figure 5.15: The alignment with only SAC-IA and ICP on yhe left side and with Search Method correction on the right side.

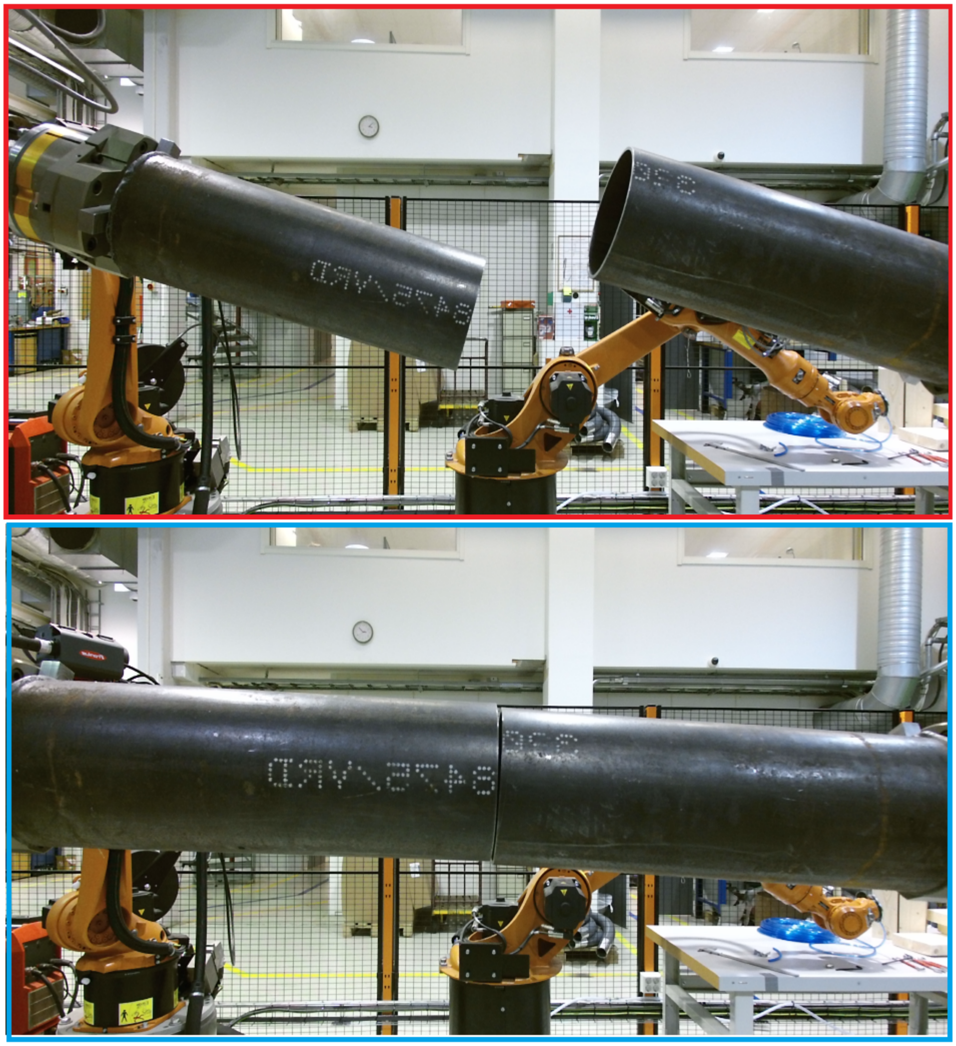

Figure 5.16: The upper part shows two tubes being tilted by the robots while the bottom figure shows the resulting fit-up using SAC-IA/ICP and Search Method.

The same test were executed using the RANSAC together with Search Method to obtain a complete transformation. The result is shown in figure 5.17 where the coordinate system is located on M_{edge} with the orientation of the tilted cylinder and the red point cloud represents the final alignment for the two point clouds using RANSAC and Search Method. Also this method was proven to be insufficiently accurate to use for welding.

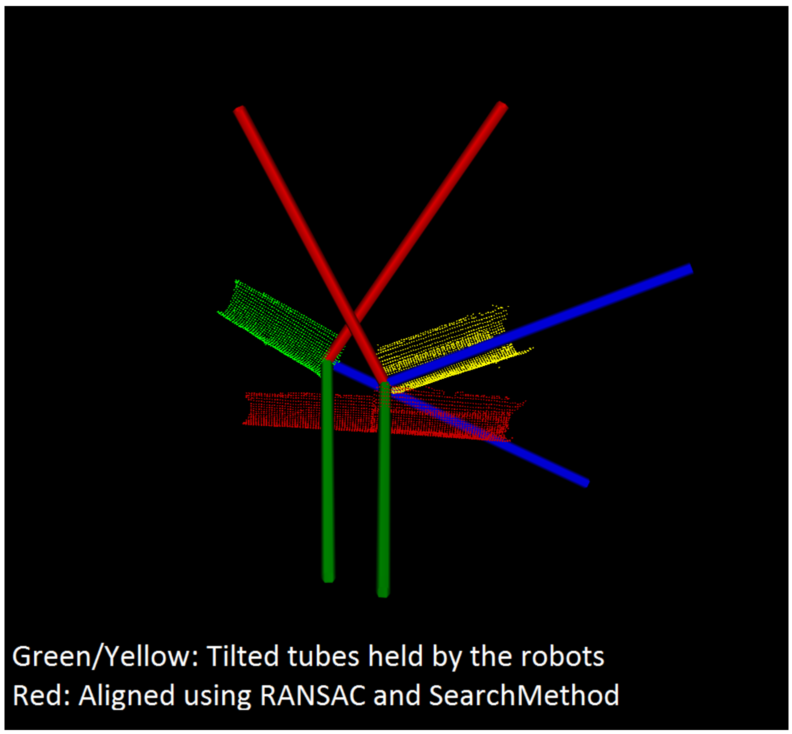

Figure 5.17: Coordinate system defined by the orientation of the cylinder and its origin is placed at M_{edge}. The red point cloud show the resulting alignment using RANSAC and Search Method.

Chapter 6

Concluding Remarks

6.1 Discussion

In the process of developing a solution using the Kinect for pose corrections of the two tubes for welding, a considerable large amount of time was used to learn the language of programming. It was invested much effort in learning how Visual Studio 2013 worked and how to utilize C++ to develop a working script. By balancing one block of code on top of the other and testing for error and bugs the script became long and with unorthodox structure. With no experience in how to build a script from the ground using functions, classes, methods and structs a good programmer would probably set question marks on the coding approach. But with persistence the final code ables to preform every intended implementation with success, even though the structure and handling of data could have been done more efficiently. That being said, if the author of this thesis could go back in time the development of a script with better user interface would have been developed. Not having a GUI Graphical user interface (GUI) for the C++ application made small changes like changing the volume for passthrough filtering very time consuming because the whole code had to be recompiled. Recompiling and testing small changes along the way was considerable time consuming and could have been avoided.

Further more, the knowledge of computer vision and 3D data processing was absent, making the handling of 3D data and extraction of vital information from it time consuming to master. Traditional pipelines for point cloud registration was looking promising at start being able to align object with simple geometry found in the office, but when tested on cylindrical objects these alignment methods struggled because of the cylinder's homogeneous geometry. After the fact that several implementation of registration methods failed, it was decided that the already well documented alignments methods was not suitable for aligning two tubes with both the translational and orientational accuracy needed for this project. The development of a method to find the precise position of a tubes lead to the *Search Method* algorithm. This method made it possible to reposition the two tubes in such a way that they could be welded together. It became also an excessive tool to improve the bad translation results using ICP.

For communication between the remote computer and the robots a solid work had already been done in the project thesis by knowing how to sequence robot programs and staring/stopping the weld using Boolean values. Still, how to write and read the robot poses using the CrossComClient class and the JOpenShowVar had to be mastered. Since the Java "ControlSystem" application had to be used for communication it was decided that it would also be used for controlling the Matlab and C++ application. Java presented it self as an easier programing-language than C++ and was the obvious choice for programing the control of the solution.

6.2 Conclusion

In this Master's thesis solutions to align cylindrical object with a target cylinders to preform sufficiently good fit-ups for welding was tested. Using SAmple Consensus for Initial Alignment and Iterative Closest Point for fine adjustment resulted in a standard deviation y-value of 25.91mm making the translation too inaccurate for welding. The translation was fixed by Search Method, but the deviation in orientation makes the ICP together with SAC-IA not accurate enough to meet the welding criteria of having a root opening of 1.6mm and internal misalignment of 1.25mm. The other aligning method was to use RANSAC to find orientation and Search Method to obtain translation. This method had a lower standard deviation than the ICP for finding orientation, but still not good enough for welding. This concludes that for orientation neither RANSAC or ICP are suitable for aligning two cylinders for welding. For translation the ICP showed great weakness because of the homogeneous geometry of a cylinder, while the Search Method tailored for finding the edge of a cylinder proved to be very precise with a stunning mean internal misalignment error of only 0.48±0.25mm making it suitable for fitting up two tubes for welding.

The alignment methods could due to data noise generate erroneous alignment poses which could possible damage the robot or its environment. To protect the robots, two safety measures were implemented. First, because the two tubes have to be closer than 1.6mm from each other to be weldable, the slightest error could cause the cylinders to crash and potentially damage the robots. A force controlled approach was implemented to stop the robot protecting the fit-up against damaging reaction forces. Secondly, a robot safety application was developed in Matlab to simulate the calculated new pose obtained by the Kinect and C++ alignment application. By collision testing and visual observation of the simulation a operator can decide if the new poses are weldable or if the robots collide.

6.3 Recommendations for Further Work

The task of welding together tubes with unknown run-out is solved, but there could be improvements. For starters, the C++ code could be cleaned up making it more readable for third party readers and it could have had a better user interface. With better user interface it could be possible to change processing parameters in real time instead of recompiling the code for every change. Qt GUI is a cross-platform application framework that could be used for developing a better graphical user interface.

The calibration of the camera frame relative to the robot frames was done manually by jogging the robots and recoding different positions. If the Kinect had been bumping into or moved, the calibration had to be repeated. To prevent the time-consuming effort of manually re-calibrate the camera regularly and calibration application should be developed. This could be done by attaching a ball of known radius to the robots end-effector and by utilizing RANSAC for spheres to track the ball origin position. Pre-made robot motions with known position and the position detection using RANSAC could generate the needed positions to calculate the transformation matrix between the two frames.

Next, for better flexibility a parameterized solution could be developed to weld together tubes of different diameter. The Search Method algorithm works for any tube independent of its diameter, but new welding paths would have to be generated and written to the KR-16 robot.

Last, the JOpenShowVar solution for communication only provides for soft real-time access to the manipulator to be controlled. Soft real-time means that the communication delay is to big to control the manipulators in real-time. By acquiring KUKA.RobotSensorInterface and KUKA.Ethernet KRL XML a real-time control system can be developed opening for the opportunities of using the already installed KUKA.ForceTorqueControl to move the manipulators according the input forces.

Bibliography

[1] *KUKA KR QUANTEC PRO Specification.*

[2] *Net F/T Network Force/Torque Sensor System Installation and Operation Manual.*

[3] *Robot programming 1 Kuka System Software 8 Training Documentation.*

[4] specification for pipework welding, inspection, supporting and testing.

[5] *TRANSSTEEL ROBOTICS CONFIGURATION.*

[6] B. Buttgen, T. Oggier, M. L. R. K. and Lustenberger, F. (2005). Ccd/cmos lock-in pixel for range imaging: Challenges, limitations and state-of-the-art. *Swiss Center for Electronics and Microtechnology.*

[7] Burgard, W. (2009). *Intelligent Autonomous Systems 10: IAS-10.* IOS Press.

[8] Carlo Dal Mutto, Pietro Zanuttigh, G. M. C. (2012). *Time-of-Flight Cameras and Microsoft Kinect.* Springer.

[9] Chen, Y. and Medioni, G. (1991). Object modeling by registration of multiple range images. *Image and Vision Computing.*

[10] Corke, P. (2011). *Robotic Vision and Control Fundamental Algorithm in MATLAB.* Springer.

[11] Corral, M. (2008). *Vector Calculus.* LL.

[12] David Forsyth, Philip Torr, A. Z. (2008). *Computer Vision - ECCV 2008: 10th European Conference on Computer Vision, Marseille, France, October 12-18, 2008. Proceedings.* Springer Science & Business Media.

[13] Davison, R. (2000). General guidelines, practical aspects for production welding, a fabricators view. *TAPPI journal 2000, volume 83, no.9.*

[14] F. Sanfilippo, L. I. Hatledal, H. Z. M. F. and Pettersen, K. Y. (2014). Jopenshowvar: an open-source cross-platform communication interface to kuka robots. *Information and Automation (ICIA), 2014 IEEE International Conference on.*

[15] Garcia, S. (2009). Fitting primitive shapes to point clouds for robotic grasping. Master's thesis, School of Electrical Engineering - Royal Institute of Technology.

[16] H. Nakamura, K. Yamamoto, T. I. and Koyama, T. (1993). *Development of off-line programming system for spot welding robot.* IEEE.

[17] HOPPE, H., D. T. D. T. M. J. and STUETZLE, W. (1992). Surface reconstruction from unorganized points. *Computer Graphics (SIGGRAPH'92 Proceedings) 26.*

[18] Iglewicz, B. and Hoaglin, D. (1993). *How to Detect and Handle Outliers.* ASQC Quality Press.

[19] Jason Luck, C. L. and Hoff, W. (2000). Rigistration of range data using a hybrid simulated annealing and iterative closest point algorithm. *IEEE 2000 International Conference on Robotics & Automation.*

[20] Jeff Knisley, K. S. (2014). *Calculus: A Modern Approach.* John Wiley & Sons,.

[21] Katsumi, M. and Kenji*2, O. (2013). Steel products for energy industries. JFE TECHNICAL REPORT No. 18, JFE TECHNICAL REPORT.

[22] Klein†, R. S. R. W. R. (2007). Efficient ransac for point-cloud shape detection. *The Eurographics Association and Blackwell Publishing 2007.*

[23] Muft, F. and Mahony, R. (2009). Statistical analysis of measurement processes for time-of flight cameras. *Proceedings of SPIE the International Society for Optical Engineering.*

[24] N.Jazar, R. (2010). *Theory of Applied Robotics Kinematics, Dynamics, and Control.* Springer.

[25] Radu Bogdan Rusu, Nico Blodow, M. B. (2009). Fast point feature histograms (fpfh) for 3d registration. *2009 IEEE International Conference on Robotics and Automation.*

[26] Sekiguchi, N. (2015). Steel market developments.

[27] Smith, C. (2011). Microsoft kinect: World's fastest-selling consumer electronics device. http://www.washingtonpost.com/wp-dyn/content/article/2009/03/27/AR2009032701576.html.

[28] T. Huysmans, J. S. and Verdonk, B. (2005). Parameterization of tubular surfaces on the cylinder. *Journal of the Winter School of Computer Graphics*.

[29] Wajahat Kazmia, Sergi Foixb, G. A. H. J. A. (2014). Indoor and outdoor depth imaging of leaves with time of flight and stereo vision sensors: Analysis and comparison. *ISPRS J. Photogram. Remote Sens.*

[30] Yu Liu, Zainan Jiang, H. L. and Xu, W. (2012). Geometric parameter identification of a 6-dof space robot using a laser-ranger. *Journal of Robotics Volume 2012 (2012), Article ID 587407,*.

[31] Zhang, L. and Saddik, A. E. (2015). Evaluation and improving the depth accuracy of kinect for windows v2. *IEEE Sensors Journal*.

[32] Ziv (2010). Rigid registration: The iterative closest point algorithm. Master's thesis, The Hebrew University, Jerusalem, Isreal.

[33] Ø. Karlsen, B., F. O. U. o. T. S. (2015). Kunnskapsstatus som grunnlag for kapasitetsjustering innen produksjonsområder basert på lakselus som indikator. Technical report, havforskningsinstituttet.

Appendix A

Source code

A.1 Matlab Safety Application

A.2 Safety Application

```

% Simen Hagen Bredvold 05.06.2016
% Master's Thesis
% NTNU
% This is a program for to simulate the new poses given from
    the C++
% application and the Kinect. To protect the environment and
    robots against
% collision this program will test the new poses against
    collision.

%Read in the DH parameters for KUKA 120 robot
DH=getDH();
Radconverter=(2*pi)/360;

```

```
% Start position right robot
qrightstart=[0 (-pi/2) pi/2 0 0 0]; %need a offsett on theta3
qrightstart(3)=qrightstart(3)-(pi/2);
%Right object using the SerialLink class. Plot the start
    position
RIGHT = SerialLink(DH, 'name', 'KUKA 120 RIGHT');
RIGHT.plot(qrightstart,'notiles');

 hold on % to plot in the same figure

% Start position left robot
qleftstart=[0 (-pi/2) pi/2 0 0 0]; %need a offsett on theta3
qleftstart(3)=qleftstart(3)-(pi/2);
LEFT = SerialLink(DH, 'name', 'KUKA 120 LEFT'); %LEFT object
LEFT.base=transl([0 -3.12 0]); %
LEFT.plot(qleftstart,'notiles')

% qPickUpRight and qPickUpLeft are the joint configuration
    when the robots picks up their
% tubes from the floor.

% RIGHT
qPickUpRight=[-0.02 -55.7 90.10 0.07 55.56 89.92]; %need a
    offsett on theta3
qPickUpRight(3)=qPickUpRight(3)-((90));
qPickUpRightTH=(-(qrightstart/Radconverter)+qPickUpRight)/10;

%ONE LEFT
qPickUpLeft=[0.77 -54.8 87.07 -0.04 57.76 0.77]; %need a
    offsett on theta3
```

```
qPickUpLeft(3)=qPickUpLeft(3)-((90));
qPickUpLeftTH=(-(qleftstart/Radconverter)+qPickUpLeft)/10;

%The program waits until the robot program starts
B=1;
while B==1
    A=fileread('C:\Users\simen_000\Desktop\NetBeansProjectsmededit\JavaControlSystem\ControlSystem\MatlabStarter.txt');
    B=strfind(A, 'FALSE');
    pause(1);
    disp(B)
end

%Move to the pick up position for the tubes.
% The jtraj method can only visualize one robot moving at the time.
% To display them at the "same" time the robot movements are split
% up into 10 parts.
t = [0:.1:0.1]';
for k=1:10
    CurrentRight=(qPickUpRightTH*(k-1)*Radconverter) + qrightstart;
    NextRight=(qtwoTH*(k)*Radconverter) + qrightstart;
    CurrentLeft=(qPickUpLeftTH*(k-1)*Radconverter) + qleftstart;
    NextLeft=(qPickUpLeftTH*(k)*Radconverter) + qleftstart;
    %Joint space trajectory
    qright = jtraj(CurrentRight, NextRight, t);
```

```
    qleft = jtraj(CurrentLeft, NextLeft, t);
    RIGHT.plot(qright,'notiles');
    LEFT.plot(qleft,'notiles');
end
t = [0:.05:2]';
%Wait for the robots to finish picking up the tubes.
B=1;
while B==1
    A=fileread('C:\Users\simen_000\Desktop\
    NetBeansProjectsmededit\JavaControlSystem\ControlSystem\
    MatlabTubePickedUp.txt');
    B=strfind(A, 'FALSE');
    C = textscan(A,'%s %f %f'); % save the number values in
    the file. Length of tube.
    pause(1);
    disp(B)
end

% Position the robots infront of the camera using program
    SETPOSITIONLEFT and SETPOSITIONRIGHT:

% Program SETPOSITIONRIGHT RIGHT joint configuration
qSETPOSITIONRIGHT=[-8.2 -92.88 114.67 -93.2 82.39 127.05]; %
    need a offsett on theta3
qSETPOSITIONRIGHT(3)=qSETPOSITIONRIGHT(3)-((90));
[Tsetpositionright ,Jsetpositionright]=forwardkinematics(DH,
    qSETPOSITIONRIGHT*Radconverter);

%Program SETPOSITIONLEFT LEFT joint configuration
```

```
qSETPOSITIONLEFT=[1.77 -91.03 113.38 90.51 88.64 -3.78]; % need a offsett on theta3
qSETPOSITIONLEFT(3)=qSETPOSITIONLEFT(3)-((90));
[Tsetpositionleft ,Jsix]=forwardkinematics(DH, qSETPOSITIONLEFT*Radconverter);

qSETPOSITIONRIGHTTH=(-(qSETPOSITIONRIGHT) + qSETPOSITIONRIGHT)/10;
qSETPOSITIONLEFTTH=(-(qSETPOSITIONLEFT) + qSETPOSITIONLEFT)/10;

%Display the robot movements:
t = [0:.1:0.1]';
for k=1:10
    CurrentRight=(qSETPOSITIONRIGHTTH*(k-1)*Radconverter) + qPickUpRight*Radconverter;
    NextRight=(qSETPOSITIONRIGHTTH*(k)*Radconverter) + qPickUpRight*Radconverter;
    CurrentLeft=(qSETPOSITIONLEFTTH*(k-1)*Radconverter) + qPickUpLeft*Radconverter;
    NextLeft=(qSETPOSITIONLEFTTH*(k)*Radconverter) + qPickUpLeft*Radconverter;
    %Joint space trajectory
    qright = jtraj(CurrentRight, NextRight, t);
    qleft = jtraj(CurrentLeft, NextLeft, t);
    RIGHT.plot(qright,'notiles');
    LEFT.plot(qleft,'notiles');
end

```

```
%Create two cylinders in the plot with the same pose as the end-effectore.
location= [0 0 0]';
scale_right = [0.168 0.168 C{2}]; %diameter of cylder and length of tube
scale_left = [0.168 0.168 C{1}]; %diameter of cylder and length of tube
Trotx=[rotx(-pi/2), location; 0,0, 0, 1];

location_RIGHT = [0 0 0]';
T_RIGHT_cylinder_HOLD = [eye(3), location_RIGHT; 0, 0, 0, 1];
Tnew_RIGHT_HOLD=T_RIGHT_cylinder_HOLD*Tsetpositionright*Trotx; %Pose of cylinder
base_RIGHT_HOLD = Cylinder(Tnew_RIGHT_HOLD, scale_right, 'FaceColor', [1 1 1], 'EdgeColor', 'none'); %Object of class Cylinder
cylinder_RIGHT_HOLD=CollisionModel(base_RIGHT_HOLD);
hold_right=cylinder_RIGHT_HOLD.plot;

%Left robot
location_LEFT = [0 -3.12 0]';
T_LEFT_cylinder_HOLD = [eye(3), location_LEFT; 0, 0, 0, 1];
Tnew_LEFT_HOLD=T_LEFT_cylinder_HOLD*Tsetpositionleft*Trotx;
base_LEFT_HOLD = Cylinder(Tnew_LEFT_HOLD, scale_left, 'FaceColor', [1 1 1], 'EdgeColor', 'none');
cylinder_LEFT_HOLD=CollisionModel(base_LEFT_HOLD);
hold_left=cylinder_LEFT_HOLD.plot;

%Waits until the C++ application is finished. The next movement is based
```

```
%on the information from the Kinect and it checks if the new
    pose is
%possible to reach without crashing.

B=1;
while B==1
    A=fileread('C:\Users\simen_000\Desktop\
    NetBeansProjectsmededit\JavaControlSystem\ControlSystem\
    MatlabSafetyProgram.txt');
    B=strfind(A, 'FALSE');
    pause(1);
    disp(B)
end
delete(hold_left); % remove the cylinders.
delete(hold_right); % remove the cylinders.

% The current transformation matrix for each robot.
[Tright ,Jtuberight]=forwardkinematics(DH,qSETPOSITIONRIGHT*
    Radconverter);
[Tleft ,Jtubeleft]=forwardkinematics(DH,qSETPOSITIONLEFTTH*
    Radconverter);

% Read the new transformation matrix from each robot.
% Also, the dimmensjion is in mm while matlab uses meter.
%Function to convert the commas to dots because matlab can't
    read commas.
comma2point_overwrite('C:\Users\simen_000\Desktop\
    NetBeansProjectsmededit\JavaControlSystem\ControlSystem\
    transformationmatrix_RIGHT.txt');
```

```
comma2point_overwrite('C:\Users\simen_000\Desktop\NetBeansProjectsmededit\JavaControlSystem\ControlSystem\transformationmatrix_LEFT.txt');

%Read the transformation matrices greated by the C++ application.
filename = 'C:\Users\simen_000\Desktop\NetBeansProjectsmededit\JavaControlSystem\ControlSystem\transformationmatrix_RIGHT.txt';
RightTransformation=importdata(filename);
filename = 'C:\Users\simen_000\Desktop\NetBeansProjectsmededit\JavaControlSystem\ControlSystem\transformationmatrix_LEFT.txt';
LEFTTransformation=importdata(filename);

%Convert to meter from mm:
LEFTTransformation(1,4)=LEFTTransformation(1,4)/1000;
LEFTTransformation(2,4)=LEFTTransformation(2,4)/1000;
LEFTTransformation(3,4)=LEFTTransformation(3,4)/1000;

RightTransformation(1,4)=(RightTransformation(1,4)/1000);
RightTransformation(2,4)=RightTransformation(2,4)/1000;
RightTransformation(3,4)=RightTransformation(3,4)/1000;

% Add the correction translation to the new transformation matrix
NewTransformationMatrixLeft(1,4)=LEFTTransformation(3,4)+Tleft(1,4); %Translation
NewTransformationMatrixLeft(2,4)=LEFTTransformation(1,4)+Tleft(2,4); %Translation
```

```
NewTransformationMatrixLeft(3,4)=LEFTTransformation(2,4)+Tleft(3,4); %Translation

NewTransformationMatrixRight(1,4)=RightTransformation(3,4)+Tright(1,4);
NewTransformationMatrixRight(2,4)=RightTransformation(1,4)+Tright(2,4);
NewTransformationMatrixRight(3,4)=RightTransformation(2,4)+Tright(3,4);

%Inverse kinematics to find the new joint configurations for the two new
%poses.
qright = InverseKinematics(DH, NewTransformationMatrixRight, qSETPOSITIONRIGHT*Radconverter);
qleft = InverseKinematics(DH, NewTransformationMatrixLeft, qSETPOSITIONLEFTTH*Radconverter);
t=[0:0.05:2]';
%Move the end effector from the current configuration to the new
%configuration
q1 = jtraj(qSETPOSITIONRIGHT*Radconverter, qright, t);
q2 = jtraj(qSETPOSITIONLEFTTH*Radconverter, qleft, t);

RIGHT.plot(q1,'notiles');
LEFT.plot(q2,'notiles');

%Check if they crash:
%This is done by preforming collision checking which takes the SerialLink
```

```
%object and its joint configuration and check if it
   intersects with a solid
%model.

%Create the solid model object for left and right robot:
location_RIGHT = [0 0 0]';
T_RIGHT_cylinder = [eye(3), location_RIGHT; 0, 0, 0, 1];
Tnew_RIGHT=T_RIGHT_cylinder*NewTransformationMatrixRight*
   Trotx;
base_RIGHT = Cylinder(Tnew_RIGHT, scale_right, 'FaceColor',
   [0 1 0], 'EdgeColor', 'none'); % Green tube
cylinder_RIGHT=CollisionModel(base_RIGHT);

location_LEFT = [0 -3.12 0]';
T_LEFT_cylinder = [eye(3), location_LEFT; 0, 0, 0, 1];
Tnew_LEFT=T_LEFT_cylinder*NewTransformationMatrixLeft*Trotx;
base_LEFT = Cylinder(Tnew_LEFT, scale_left, 'FaceColor', [0 1
    0], 'EdgeColor', 'none');
cylinder_LEFT=CollisionModel(base_LEFT);

%Create two new SerialLink object which have an extended last
    link equal to
% the length of the cylinder.
DH_right=DH;
DH_right(6,2)=DH_right(6,2)+C{2}; %C{2} is the lenght of the
   right tube.
RIGHT_collision = SerialLink(DH_right, 'name', 'KUKA 120
   RIGHT');

DH_left=DH;
```

```
DH_left(6,2)=DH_left(6,2)+C{1}; %C{1} is the lenght of the
    right tube.
LEFT_collision = SerialLink(DH_left, 'name', 'KUKA 120 RIGHT'
    );

%Check if the right robot crash with the left tube and the
    right tube::
C_right=RIGHT_collision(q1,cylinder_LEFT);
C_left=LEFT_collision(q2,cylinder_RIGHT);

%If either C_left or C_right is true then on of them crashes
    and the pose
%must not be executed by the real robots. To visualize plot
    the two robots
%holding their tube and color the tube that crashes red.

if C_left==1
    base_LEFT_red = Cylinder(Tnew_LEFT, scale_left, '
    FaceColor', [1 0 0], 'EdgeColor', 'none'); %red tube
    cylinder_LEFT_red=CollisionModel(base_LEFT_red);
    cylinder_LEFT_red.plot;
else
    cylinder_LEFT.plot;
end

if C_right==1
    base_RIGHT_red = Cylinder(Tnew_RIGHT, scale_right, '
    FaceColor', [1 0 0], 'EdgeColor', 'none'); %red tube
    cylinder_RIGHT_red=CollisionModel(base_RIGHT_red);
    cylinder_LEFT_red.plot; %Plot the red tube
```

```
else
  cylinder_RIGHT.plot; %Plot the green tube which does not
   crash
end
```

Listing A.1: SafetyApplication.m

A.3 C++ Alignment application

A.3.1 C++ Main source file

The source code below is for processing and finding the transformation matrix the right cylinder. The left cylinder is left out to save pages. The whole code is found in the digital appendix.

```

 /*
- Author: Simen Hagen Bredvold
- Master's Thesis NTNU IPK 2016

This code was developed for the purpose of aligning two cylindrical object to a
    known position.
The following code does the following:
- Capture point cloud from the environment
- Filter the captured environment and storing the two cylindrical object in each
    point cloud
- Three different methods for aligning a cylindrical object:
1) SAC-IA as initial guess for ICP.
2) RANSAC together with Search Method.
3) Only search method for translation
- Visualization of wanted and current point cloud

The source code consist of:
1) MultiPictureMain.cpp: where the classes and main is defined
2) Functions.h: header file which includes all liabaries and declear all functions
    needed.
3) Functions.cpp:  Function definitions needed.

Environment:
-Point Cloud Library
-Kinect for Windows SDK v2.0
-Visual Studio Community 2013
-C++ Standard Library
```

```
-Eigen library for linear algebra: matrices, vectors, numerical solvers, and
    related algorithms.
*/

#define _SCL_SECURE_NO_WARNINGS
#define _CRT_SECURE_NO_WARNINGS

#include "Functions.h"
#include "kinect2_grabber.h"

//Object decleration
pcl::NormalEstimation<PointT, pcl::Normal> ne_right;
pcl::SACSegmentationFromNormals<PointT, pcl::Normal> seg_right;
pcl::ExtractIndices<PointT> extract_right;
pcl::ExtractIndices<pcl::Normal> extract_normals_right;
pcl::search::KdTree<PointT>::Ptr tree_right(new pcl::search::KdTree<PointT>());
pcl::search::KdTree<PointT>::Ptr search_local_feature_right(new pcl::search::KdTree
    <PointT>());
pcl::SampleConsensusInitialAlignment<pcl::PointXYZ, pcl::PointXYZ, pcl::
    FPFHSignature33> sac_ia_right;
pcl::FPFHEstimation<pcl::PointXYZ, pcl::Normal, pcl::FPFHSignature33>
    fpfh_est_right;
pcl::IterativeClosestPoint<pcl::PointXYZ, pcl::PointXYZ> icp_RIGHT;
pcl::PCDWriter writer;

//Algorithm parameters
float LengthFromCamera(0.6);
float TubeRadius(0.084);
int iterations(150);

using namespace std;

//Class for grabbing point clouds and filtering them
class ViewGrapAndFilter
{
public:
```

```
ViewGrapAndFilter() : viewer("PCL Viewer"){
  frames_saved = 0;
  save_one = false;
}

void GetFilteredPointCloud(const pcl::PointCloud<pcl::PointXYZ>::Ptr cloudfiltered_LEFT,
  const pcl::PointCloud<pcl::PointXYZ>::Ptr cloudfiltered_RIGHT){
  //To get the clouds found in this class
  *cloudfiltered_LEFT = cloud_from_camera_filtered_LEFT;
  *cloudfiltered_RIGHT = cloud_from_camera_filtered_RIGHT;
}

void Grab(const pcl::PointCloud<pcl::PointXYZ>::ConstPtr &cloud){
  //Grabs the scene and adds it to cloud_a.
  if (!viewer.wasStopped()) {
    viewer.showCloud(cloud);

    if (save_one) {
      save_one = false;

      cloud_a = cloud_a + *cloud;
    }
  }
}

void FilterCapturedPC(){
  //Point clouds, model coefficients, normals and Indices storage
  pcl::PointCloud<pcl::PointXYZ>::Ptr cloud_inn(new pcl::PointCloud<pcl::PointXYZ>);
  pcl::PointCloud<pcl::PointXYZ>::Ptr cloud_passthroughz(new pcl::PointCloud<pcl::PointXYZ>);
  pcl::PointCloud<pcl::PointXYZ>::Ptr cloud_passthroughx(new pcl::PointCloud<pcl::PointXYZ>);
  pcl::PointCloud<pcl::PointXYZ>::Ptr cloud_passthroughy(new pcl::PointCloud<pcl::PointXYZ>);
```

```
    pcl::PointCloud<pcl::PointXYZ>::Ptr cloud_passthroughxyz(new pcl::PointCloud<
    pcl::PointXYZ>);
    pcl::PointCloud<pcl::PointXYZ>::Ptr cloud_downsampled(new pcl::PointCloud<pcl::
    PointXYZ>);
    pcl::PointCloud<pcl::PointXYZ>::Ptr cloud_negativ_x(new pcl::PointCloud<pcl::
    PointXYZ>);
    pcl::PointCloud<pcl::PointXYZ>::Ptr cloud_positiv_x(new pcl::PointCloud<pcl::
    PointXYZ>);
    pcl::PointCloud<PointT>::Ptr cloud_right(new pcl::PointCloud<PointT>);
    pcl::PointCloud<pcl::Normal>::Ptr cloud_normals_right(new pcl::PointCloud<pcl::
    Normal>);
    pcl::ModelCoefficients::Ptr  coefficients_cylinder_right(new pcl::
    ModelCoefficients);
    pcl::PointIndices::Ptr  inliers_cylinder_right(new pcl::PointIndices);

    //Get point cloud captured
    *cloud_inn = cloud_a;
    //Pass thorugh filter. Removes every point not lying within the boundaries of x
    ,y and z.
    PassthroughFilter(cloud_inn, cloud_passthroughz, 0.6, 2, 'z');
    PassthroughFilter(cloud_passthroughz, cloud_passthroughy, -0.7, 0.7, 'y');
    PassthroughFilter(cloud_passthroughy, cloud_passthroughx, -1.4, 1.4, 'x');

    //downsampling voxel grid:
    DownsamplingFilter(cloud_passthroughx, cloud_downsampled, 0.008f); //actually
    0.008

    //Split the point cloud into two. Right and left tube.
    //---------------LEFT - RIGHT cloud splitting start----------
    // Go through every point and store all point with negative and postive value
    of
    // x in different point clouds
    float X;
    for (size_t i = 0; i < cloud_downsampled->points.size(); ++i){
      X = cloud_downsampled->points[i].x;
      if (X < 0)
```

```
        {
          pcl::PointXYZ basic_point_negativ;
          basic_point_negativ.x = X;
          basic_point_negativ.y = cloud_downsampled->points[i].y;
          basic_point_negativ.z = cloud_downsampled->points[i].z;
          cloud_negativ_x->points.push_back(basic_point_negativ);
        }
        else
        {
          pcl::PointXYZ basic_point_positiv;
          basic_point_positiv.x = X;
          basic_point_positiv.y = cloud_downsampled->points[i].y;
          basic_point_positiv.z = cloud_downsampled->points[i].z;
          cloud_positiv_x->points.push_back(basic_point_positiv);
        }
      }

      cloud_negativ_x->width = (int)cloud_negativ_x->points.size();
      cloud_negativ_x->height = 1;
      cloud_positiv_x->width = (int)cloud_positiv_x->points.size();
      cloud_positiv_x->height = 1;
      cloud_right = cloud_positiv_x;
      cloud_left = cloud_negativ_x;

      //----------------LEFT - RIGHT cloud splitting end-------------------

      //------------------------- RANSAC RIGHT - START ---------------------
      // Estimate point normals
      ne_right.setSearchMethod(tree_right);
      ne_right.setInputCloud(cloud_right);
      ne_right.setKSearch(50);
      ne_right.compute(*cloud_normals_right);

      // RANSAC for cylindrical object
      seg_right.setOptimizeCoefficients(true);
      seg_right.setModelType(pcl::SACMODEL_CYLINDER);
```

```
    seg_right.setMethodType(pcl::SAC_RANSAC);
    seg_right.setNormalDistanceWeight(0.1);
    seg_right.setMaxIterations(10000);
    seg_right.setDistanceThreshold(0.05);
    seg_right.setRadiusLimits(0, 0.09);
    seg_right.setInputCloud(cloud_right);
    seg_right.setInputNormals(cloud_normals_right);

    // Obtain the cylinder inliers and coefficients
    seg_right.segment(*inliers_cylinder_right, *coefficients_cylinder_right);
    std::cerr << "Cylinder coefficients: " << *coefficients_cylinder_right << std::
    endl;

    // Write the cylinder inliers to disk
    extract_right.setInputCloud(cloud_right);
    extract_right.setIndices(inliers_cylinder_right);
    extract_right.setNegative(false);
    pcl::PointCloud<PointT>::Ptr cloud_cylinder_right(new pcl::PointCloud<PointT>()
    );
    extract_right.filter(*cloud_cylinder_right);

    if (cloud_cylinder_right->points.empty())
      std::cerr << "Can't find the right cylindrical component." << std::endl;
    else
    {
      std::cerr << "PointCloud representing the left cylindrical component: "
        << cloud_cylinder_right->points.size() << " data points." << std::endl;
    }
    //------------------------ RANSAC RIGHT - END --------------------------------

    //----------------------REMOVE BACK SIDE - START ------------------------------
    // WANT TO REMOVE ALL POINTS THAT REPRESENT THE BACK SIDE OF THE
    // TUBE BECAUSE THEY DISTURB ICP and SAC-IA ALIGNMENT.
    pcl::PointXYZ minPt_right, maxPt_right;
    pcl::getMinMax3D(*cloud_cylinder_right, minPt_right, maxPt_right);
    float ZvalueFilter_RIGHT;
```

```
    float zlimit_RIGHT = minPt_right.z + 0.084; // the distance along the z-axis to the center of the cylinder
    pcl::PointCloud<PointT>::Ptr cloud_cylinder_right_filtered(new pcl::PointCloud<PointT>());
    cout << "the value of z-right is " << zlimit_RIGHT << endl;

    //Remove every point having a larger z-value then zlimit_RIGHT and zlimit_LEFT.
    //RIGHT SIDE:
    for (size_t i = 0; i < cloud_cylinder_right->points.size(); ++i){
      ZvalueFilter_RIGHT = (cloud_cylinder_right->points[i].z);
      if (ZvalueFilter_RIGHT<zlimit_RIGHT)
      {
        pcl::PointXYZ basic_point_right;
        basic_point_right.x = cloud_cylinder_right->points[i].x;
        basic_point_right.y = cloud_cylinder_right->points[i].y;
        basic_point_right.z = cloud_cylinder_right->points[i].z;
        cloud_cylinder_right_filtered->points.push_back(basic_point_right);
      }
    }
    cloud_cylinder_right_filtered->width = (int)cloud_cylinder_right_filtered->points.size();
    cloud_cylinder_right_filtered->height = 1;


    //----------------------------REMOVE BACK SIDE OF TYPE - END ----------------------

    //--------------------------Make the surface smoother - start----------------------
    // Use the function defined in functions.cpp using moving least surface to smoothen surface
    MLS(cloud_cylinder_right_filtered, cloud_cylinder_right_filtered, 0.03);
    //---------------------------Make the surface smoother - end ----------------------

    //Store the two filtered in the private point cloud variable cloud_from_camera_filtered_LEFT
    // and cloud_from_camera_filtered_RIGHT
    cloud_from_camera_filtered_RIGHT = *cloud_cylinder_right_filtered;
```

```
    }

  void RunControl(){
    pcl::Grabber* grabber = new pcl::Kinect2Grabber();
    boost::function<void(const pcl::PointCloud<pcl::PointXYZ>::ConstPtr&)> f =
      boost::bind(&ViewGrapAndFilter::Grab, this, _1);
    grabber->registerCallback(f);
    grabber->start();
    // capture 10 point cloud from the scene and add them together.
    bool FilterStarter = false;
    bool PictureStarter = true;
    while (!viewer.wasStopped()){
      if (PictureStarter)
      {
        for (size_t i = 0; i < 10; i++)
        {
          cout << "Saving frame " << frames_saved << ".\n";
          frames_saved++;
          save_one = true;
          Sleep(500);
        }
        Sleep(3000);
      }
      if (FilterStarter)
      {
        FilterCapturedPC();
        Sleep(3000);
        viewer.~CloudViewer();
        break;
      }
      FilterStarter = true;
      PictureStarter = false;
      grabber->stop();
    }
  }
  pcl::visualization::CloudViewer viewer;
```

```
private:
  int frames_saved;
  bool save_one;
  pcl::PointCloud<pcl::PointXYZ> cloud_a;
  pcl::PointCloud<pcl::PointXYZ> cloud_from_camera_filtered_LEFT;
  pcl::PointCloud<pcl::PointXYZ> cloud_from_camera_filtered_RIGHT;
};

template <typename PointType>
//Class for  alignment:
class AlignmentViewerStream
{
  typedef pcl::PointCloud<PointType> PointCloud;
  typedef typename PointCloud::ConstPtr ConstPtr;
public:
  AlignmentViewerStream(pcl::Grabber& grabber)
    : viewer(new pcl::visualization::PCLVisualizer("Point Cloud Viewer"))
    , grabber(grabber)
  {
  }
  //Method for passing in the filtered point clouds
  void PassClouds(const pcl::PointCloud<pcl::PointXYZ>::Ptr scene_in_filtered_LEFT,
    const pcl::PointCloud<pcl::PointXYZ>::Ptr scene_in_filtered_RIGHT){
    scene_filtered_LEFT = *scene_in_filtered_LEFT;
    scene_filtered_RIGHT = *scene_in_filtered_RIGHT;
  };

  //Method for ICP algorithm and SearchMethod for correction
  void ICPalgorithm(){
    //PointClouds:

    pcl::PointCloud<pcl::Normal>::Ptr cloud_normals_Feature_LEFT(new pcl::PointCloud<pcl::Normal>);
    pcl::PointCloud<pcl::PointXYZ>::Ptr scene_ready_RIGHT(new pcl::PointCloud<pcl::PointXYZ>);
    pcl::PointCloud<pcl::PointXYZ>::Ptr SAC_IA_RIGHT(new pcl::PointCloud<pcl::
```

```
PointXYZ>);
pcl::PointCloud<pcl::Normal>::Ptr cloud_normals_Feature_RIGHT(new pcl::
PointCloud<pcl::Normal>);
pcl::PointCloud<pcl::FPFHSignature33>::Ptr features_RIGHT(new pcl::PointCloud <
pcl::FPFHSignature33>);
pcl::PointCloud<pcl::PointXYZ>::Ptr model_icp_RIGHT(new pcl::PointCloud<pcl::
PointXYZ>);
pcl::PointCloud<pcl::PointXYZ>::Ptr basic_cloud_ptr_RIGHT(new pcl::PointCloud<
pcl::PointXYZ>);
pcl::PointCloud<pcl::Normal>::Ptr model_normals_Feature_RIGHT(new pcl::
PointCloud<pcl::Normal>);
pcl::PointCloud<pcl::FPFHSignature33>::Ptr model_features_RIGHT(new pcl::
PointCloud <pcl::FPFHSignature33>);

//Tell the user which algorithm that have been chosen
cout << " This is the SAC-IA and ICP algorithm" << endl;

//Get the point cloud captured by the GrabAndFilter which is stored as private
variables in this class
*scene_ready_LEFT = scene_filtered_LEFT;
*scene_ready_RIGHT = scene_filtered_RIGHT;

//----------Create target model for tube left and right - START -----------

// Create a point cloud which represents where the tube should be and the
target for ICP AND SAC-IA
//For simplicity the cloud is modeled along the x-axis and is later transformed
 to the wanted position.
//For the right robot:
for (float x(-0.3); x <= 0; x += 0.005)
{
  for (float angle(135); angle <= 225; angle += 5.0)
  {
    pcl::PointXYZ basic_point;
    basic_point.x = x;
    basic_point.y = TubeRadius*sinf(pcl::deg2rad(angle));
```

```
        basic_point.z = TubeRadius*cosf(pcl::deg2rad(angle));
        basic_cloud_ptr_RIGHT->points.push_back(basic_point);
      }
    }
    basic_cloud_ptr_RIGHT->width = (int)basic_cloud_ptr_RIGHT->points.size();
    basic_cloud_ptr_RIGHT->height = 1;
    //-----------------Create target model for tube left and right - END ----------------

        // Transform the target model to have a center axis parallel to the y axis of the robot.
    //------------Locate the target point cloud at wanted position - START ----------
    ///RIGHT:
    Eigen::Matrix4f transform_target_right = Eigen::Matrix4f::Identity();
    transform_target_right(0, 0) = 0.9989;
    transform_target_right(0, 1) = -0.0215;
    transform_target_right(0, 2) = 0.0418;
    transform_target_right(0, 3) = LengthFromCamera;
    transform_target_right(1, 0) = 0.0222;
    transform_target_right(1, 1) = 0.9997;
    transform_target_right(1, 2) = -0.0116;
    transform_target_right(2, 0) = -0.0416;
    transform_target_right(2, 1) = 0.0125;
    transform_target_right(2, 2) = 0.9991;
    // Executing the transformation
    pcl::transformPointCloud(*basic_cloud_ptr_RIGHT, *model_icp_RIGHT, transform_target_right);
    //-------------------Locate the target point cloud at wanted position - END ----------------

    //-------------------SAC-IA ALIGNMENT RIGHT SIDE START--------------------------------
    // Estimate point normals for cylinder and target model
    ne_right.setSearchMethod(tree_right);
    ne_right.setInputCloud(scene_ready_RIGHT);
    ne_right.setKSearch(50);
    ne_right.compute(*cloud_normals_Feature_RIGHT);
```

```

    ne_right.setSearchMethod(tree_right);
    ne_right.setInputCloud(model_icp_RIGHT);
    ne_right.setKSearch(50);
    ne_right.compute(*model_normals_Feature_RIGHT);

    //compute local features for left side cylinder and target model
    //cylinder:
    fpfh_est_right.setInputCloud(scene_ready_RIGHT);
    fpfh_est_right.setInputNormals(cloud_normals_Feature_RIGHT);
    fpfh_est_right.setSearchMethod(search_local_feature_right);
    fpfh_est_right.setRadiusSearch(0.02f);
    fpfh_est_right.compute(*features_RIGHT);
    //target:
    fpfh_est_right.setInputCloud(model_icp_RIGHT);
    fpfh_est_right.setInputNormals(model_normals_Feature_RIGHT);
    fpfh_est_right.setSearchMethod(search_local_feature_right);
    fpfh_est_right.setRadiusSearch(0.02f);
    fpfh_est_right.compute(*model_features_RIGHT);

    //SAC-IA for the RIGHT side:
    sac_ia_right.setInputSource(scene_ready_RIGHT);
    sac_ia_right.setSourceFeatures(features_RIGHT);
    sac_ia_right.setInputTarget(model_icp_RIGHT);
    sac_ia_right.setTargetFeatures(model_features_RIGHT);
    sac_ia_right.setMaximumIterations(500);
    sac_ia_right.align(*SAC_IA_RIGHT);
    Eigen::Matrix4f transformation_SAC_IA_RIGHT = sac_ia_right.
    getFinalTransformation();
    //------------------------SAC-IA ALIGNMENT left SIDE END--------------------

    //------------------------ICP alignment RIGHT side START --------------------
    //ICP for right side
    icp_RIGHT.setMaximumIterations(iterations);
    icp_RIGHT.setInputSource(SAC_IA_RIGHT);
    icp_RIGHT.setInputTarget(model_icp_RIGHT);
```

```
    icp_RIGHT.align(*SAC_IA_RIGHT);
    icp_RIGHT.setMaximumIterations(1000);
      //----------------ICP alignment right side END ---------------
    //Check if the ICP score
    Eigen::Matrix4f transformation_ICP_RIGHT = Eigen::Matrix4f::Identity();
    if (icp_RIGHT.hasConverged())
    {
      std::cout << "\nICP for the right side has converged, score is " << icp_RIGHT.getFitnessScore() << std::endl;
      std::cout << "\nICP transformation " << iterations << " : model_icp -> cloud_in" << std::endl;
      transformation_ICP_RIGHT = icp_RIGHT.getFinalTransformation();
      print4x4Matrix(transformation_ICP_RIGHT);
    }
    else
    {
      PCL_ERROR("\nICP for one of the sides have not converged.\n");
    }


    //The transformation matrix obtained by the SAC-IA and the ICP is:
    Eigen::Matrix4f transformation_SAC_IA_ICP_RIGHT = Eigen::Matrix4f::Identity();
    transformation_SAC_IA_ICP_RIGHT = transformation_ICP_RIGHT*transformation_SAC_IA_RIGHT;

    //--------------SearchMethod right side for ICP algorithm START--------------
    scene_ready_RIGHT = SAC_IA_RIGHT;
    float Xvalue_RIGHT = 0;
    float ValueYY_RIGHT = 0;
    float ValueZZ_RIGHT = 0;
    SearchMethod_RIGHT(scene_ready_RIGHT, Xvalue_RIGHT, ValueYY_RIGHT, ValueZZ_RIGHT);
    Eigen::Matrix4f transformation_matrix_searchMethod_SAC_IA_ICP_RIGHT = Eigen::Matrix4f::Identity();
    float translateX_RIGHT = -Xvalue_RIGHT;
    float translateY_RIGHT = -ValueYY_RIGHT;
```

```
    float translateZ_RIGHT = -ValueZZ_RIGHT;
    transformation_matrix_searchMethod_SAC_IA_ICP_RIGHT = transformation_SAC_IA_ICP_RIGHT;
    transformation_matrix_searchMethod_SAC_IA_ICP_RIGHT(0, 3) = transformation_SAC_IA_ICP_RIGHT(0, 3) + translateX_RIGHT;
    transformation_matrix_searchMethod_SAC_IA_ICP_RIGHT(1, 3) = transformation_SAC_IA_ICP_RIGHT(1, 3) + translateY_RIGHT;
    transformation_matrix_searchMethod_SAC_IA_ICP_RIGHT(2, 3) = transformation_SAC_IA_ICP_RIGHT(2, 3) + translateZ_RIGHT;
    //--------------SearchMethod right side for ICP algorithm end----------------

    //-------Translation and Rotation in the robot coordinate system right START---------
    Eigen::Matrix4f Tcameratorobot_right = Eigen::Matrix4f::Identity();
    //Transformation matrix which mapes from the camera frame to the robot frame.
    Eigen::Matrix4f Trobot_right = Eigen::Matrix4f::Identity();
    //Transformation matrix which mappes from the camera frame to the robot frame.
    GetRightRobotToCameraMatrix(Tcameratorobot_right);
    Trobot_right = Tcameratorobot_right*transformation_matrix_searchMethod_SAC_IA_ICP_RIGHT;
    write4x4Matrix_RIGHT(Trobot_right); //Write the transformation matrix to file
    //--------Translation and Rotation in the robot coordinate system  right END---------

    //----Translate the cloud with the values found by SearchMethod for visualization right start---
    Eigen::Affine3f transform_right(Eigen::Affine3f::Identity());
    transform_right.translation() << translateX_RIGHT, translateY_RIGHT, translateZ_RIGHT;
    pcl::transformPointCloud(*scene_ready_RIGHT, *scene_ready_RIGHT, transform_right);
    writer.write(" Final aligned right side SACIA ICP SEARCHMETHOD.pcd", *scene_ready_RIGHT, false);
    //-----Translate the cloud with the values found by searchMethod for visualization right end ----

```

```
    //store the aligned cloud.
    scene_alignedRun_RIGHT = *scene_ready_RIGHT;

}
//Method for RANSAC algorithm and SearchMethod for translation
void SearchForEndOfTubeandRANSAC(){
    pcl::PointCloud<pcl::PointXYZ>::Ptr scene_filtered_search_RIGHT(new pcl::PointCloud<pcl::PointXYZ>);
    pcl::PointCloud<pcl::PointXYZ>::Ptr scene_ready_RIGHT(new pcl::PointCloud<pcl::PointXYZ>);
    pcl::PointCloud<PointT>::Ptr cloud_right(new pcl::PointCloud<PointT>);
    pcl::PointCloud<pcl::Normal>::Ptr cloud_normals_right(new pcl::PointCloud<pcl::Normal>);
    pcl::ModelCoefficients::Ptr coefficients_cylinder_right(new pcl::ModelCoefficients);
    pcl::PointIndices::Ptr inliers_cylinder_right(new pcl::PointIndices);

    //-------Translate the tube captured with the values found above END -------
    //Do the same with for right side:
    //--------------SEARCH METHOD RIGHT SIDE - START--------------------
    *scene_filtered_search_RIGHT = scene_filtered_RIGHT;
    cloud_right = scene_filtered_search_RIGHT;

    //------------Find centroid of the cloud and translate  it to the origin of the camera RIGHT START----------
    Eigen::Vector4f centroid_RIGHT(Eigen::Vector4f::Zero());
    pcl::compute3DCentroid(*scene_filtered_search_RIGHT, centroid_RIGHT);
    Eigen::Affine3f transform_centroid_right(Eigen::Affine3f::Identity());
    transform_centroid_right.translation() << -centroid_RIGHT(0), -centroid_RIGHT(1), -centroid_RIGHT(2);
    pcl::transformPointCloud(*scene_filtered_search_RIGHT, *scene_filtered_search_RIGHT, transform_centroid_right);
    //--------------Find centroid of the cloud and translate  it to the origin of the camera RIGHT END-----

    //---------FIND ORIENTATION OF CYLINDER AND ALIGN IT WITH A KNOWN FRAME START
```

```
RIGHT----------------
//-------------Use RANSAC to find orientation of right cylinder start ----------------------------
// Estimate point normals
ne_right.setSearchMethod(tree_right);
ne_right.setInputCloud(cloud_right);
ne_right.setKSearch(50);
ne_right.compute(*cloud_normals_right);

// Create the segmentation object for cylinder segmentation and set all the parameters
seg_right.setOptimizeCoefficients(true);
seg_right.setModelType(pcl::SACMODEL_CYLINDER);
seg_right.setMethodType(pcl::SAC_RANSAC);
seg_right.setNormalDistanceWeight(0.1);
seg_right.setMaxIterations(10000);
seg_right.setDistanceThreshold(0.05);
seg_right.setRadiusLimits(0.08, 0.09);
seg_right.setInputCloud(cloud_right);
seg_right.setInputNormals(cloud_normals_right);

// Obtain the cylinder inliers and coefficients
seg_right.segment(*inliers_cylinder_right, *coefficients_cylinder_right);
std::cerr << "Cylinder coefficients: " << *coefficients_cylinder_right << std::endl;

// Write the cylinder inliers to disk
extract_right.setInputCloud(cloud_right);
extract_right.setIndices(inliers_cylinder_right);
extract_right.setNegative(false);
pcl::PointCloud<PointT>::Ptr cloud_cylinder_right(new pcl::PointCloud<PointT>());
extract_right.filter(*cloud_cylinder_right);
//--------Use RANSAC to find orientation of right cylinder start ----------

//-------------Define orientation RIGHT START -------------------
```

```
  // The direction vector using the coefficients from RANSAC
  Eigen::Vector3f vector_orientation_right(Eigen::Vector3f::Zero());
  vector_orientation_right[0] = coefficients_cylinder_right->values[3];
  vector_orientation_right[1] = coefficients_cylinder_right->values[4];
  vector_orientation_right[2] = coefficients_cylinder_right->values[5];

  // Any non-zero vector which is not parallel to vector_orientation_right
  Eigen::Vector3f vector_Y_right(Eigen::Vector3f::Zero());
  vector_Y_right[0] = 0;
  vector_Y_right[1] = 1;
  vector_Y_right[2] = 0;

  if (vector_Y_right.dot(vector_orientation_right)==0)
  {
    vector_Y_right[0] = 1;
    vector_Y_right[1] = 0;
    vector_Y_right[2] = 0;
  }

  Eigen::Vector3f crossproductAxis1_right(Eigen::Vector3f::Zero());
  Eigen::Vector3f crossproductAxis2_right(Eigen::Vector3f::Zero());
  Eigen::Vector3f crossproductNew_right(Eigen::Vector3f::Zero());

  // Define a coordinate by finding two vectors which are perpendicular to
  vector_orientation_right and each other.
  crossproductAxis1_right = vector_Y_right.cross(vector_orientation_right);
  crossproductAxis2_right = vector_orientation_right.cross(
  crossproductAxis1_right);
  crossproductNew_right = crossproductAxis2_right.cross(vector_orientation_right)
  ;

  Eigen::Matrix4f rotmatrix_right_coordinatesystem = Eigen::Matrix4f::Identity();

  rotmatrix_right_coordinatesystem(0, 0) = vector_orientation_right[0];
  rotmatrix_right_coordinatesystem(0, 1) = vector_orientation_right[1];
```

```
    rotmatrix_right_coordinatesystem(0, 2) = vector_orientation_right[2];
    rotmatrix_right_coordinatesystem(1, 0) = crossproductAxis2_right[0];
    rotmatrix_right_coordinatesystem(1, 1) = crossproductAxis2_right[1];
    rotmatrix_right_coordinatesystem(1, 2) = crossproductAxis2_right[2];
    rotmatrix_right_coordinatesystem(2, 0) = crossproductNew_right[0];
    rotmatrix_right_coordinatesystem(2, 1) = crossproductNew_right[1];
    rotmatrix_right_coordinatesystem(2, 2) = crossproductNew_right[2];

    // Executing the transformation
    pcl::PointCloud<pcl::PointXYZ>::Ptr rotated_cloud_RIGHT(new pcl::PointCloud<pcl
    ::PointXYZ>());
    pcl::transformPointCloud(*cloud_right, *rotated_cloud_RIGHT,
    rotmatrix_right_coordinatesystem);

    Eigen::Matrix3f Alignmentrotation_right;

    Alignmentrotation_right(0, 0) = vector_orientation_right[0];
    Alignmentrotation_right(0, 1) = vector_orientation_right[1];
    Alignmentrotation_right(0, 2) = vector_orientation_right[2];
    Alignmentrotation_right(1, 0) = crossproductAxis2_right[0];
    Alignmentrotation_right(1, 1) = crossproductAxis2_right[1];
    Alignmentrotation_right(1, 2) = crossproductAxis2_right[2];
    Alignmentrotation_right(2, 0) = crossproductAxis1_right[0];
    Alignmentrotation_right(2, 1) = crossproductAxis1_right[1];
    Alignmentrotation_right(2, 2) = crossproductAxis1_right[2];

    // The wanted frame
    Eigen::Matrix3f rotmatrix_right_target_correction;
    rotmatrix_right_target_correction(0, 0) = 0.9989;
    rotmatrix_right_target_correction(0, 1) = -0.0215;
    rotmatrix_right_target_correction(0, 2) = 0.0418;
    rotmatrix_right_target_correction(1, 0) = 0.0222;
    rotmatrix_right_target_correction(1, 1) = 0.9997;
    rotmatrix_right_target_correction(1, 2) = -0.0116;
    rotmatrix_right_target_correction(2, 0) = -0.0416;
    rotmatrix_right_target_correction(2, 1) = 0.0125;
```

```
    rotmatrix_right_target_correction(2, 2) = 0.9991;

    Eigen::Matrix3f AlignmentRotation_correction_right;
    AlignmentRotation_correction_right = rotmatrix_right_target_correction*AlignmentRotation_left;
    scene_ready_RIGHT = rotated_cloud_RIGHT;
    //--------------------------Define orientation RIGHT END ------------------------------------------
    //-----------------FIND ORIENTATION OF CYLINDER AND ALIGN IT WITH A KNOWN FRAME END RIGHT----------


    //------------------Search Method to obtain translation right start --------------------
    float Xvalue_RIGHT = 0, ValueYY_RIGHT = 0, ValueZZ_RIGHT = 0;
    SearchMethod_RIGHT(scene_ready_RIGHT, Xvalue_RIGHT, ValueYY_RIGHT, ValueZZ_RIGHT);
    cout << "The right values found in searchMethod are: x" << Xvalue_RIGHT << "y: " << ValueYY_RIGHT << "z: " << ValueZZ_RIGHT << endl;
    //--------------Search Method to obtain translation right end ------------------------

    //------------------Find M_edge for the tilted right tube START----------------------------
    // Because the robot will rotate the cylinder about the end of the tube, while
    //  the cloud cylinder is rotated about the camera frame.
    // To adjust for this the inverse of the rotation matrix is multiplied with
    // the values found in SearchMethod and added the centroid translation.
    // This gives the position of the wanted point on the edge of the tube in a tilted orientation.

    Eigen::Vector4f wanted_RIGHT;
    Eigen::Vector4f rotated_RIGHT(Xvalue_RIGHT, ValueYY_RIGHT, ValueZZ_RIGHT, 0);
    Eigen::Matrix4f inverse_RIGHT = Eigen::Matrix4f::Identity();

    // Find the M_edge mark to account for translation of edge when rotating. See report for further information.
    inverse_RIGHT = rotmatrix_right_coordinatesystem.inverse();
    // M_edge in the camera frame:
    wanted_RIGHT = inverse_RIGHT*rotated_RIGHT;
```

```
    wanted_RIGHT(0) = wanted_RIGHT(0) + centroid_RIGHT(0);
    wanted_RIGHT(1) = wanted_RIGHT(1) + centroid_RIGHT(1);
    wanted_RIGHT(2) = wanted_RIGHT(2) + centroid_RIGHT(2);

    //The correction value to place the tube at wanted position.
    float translateX_RIGHT = -wanted_RIGHT(0);
    float translateY_RIGHT = -wanted_RIGHT(1);
    float translateZ_RIGHT = (LengthFromCamera - TubeRadius) - wanted_RIGHT(2);

    rotmatrix_right_coordinatesystem(0, 3) = translateX_RIGHT;
    rotmatrix_right_coordinatesystem(1, 3) = translateY_RIGHT;
    rotmatrix_right_coordinatesystem(2, 3) = translateZ_RIGHT;

    //----------Translation and Rotation in the robot coordinate system RIGHT START
    -------
    Eigen::Matrix4f Tcameratorobot_right = Eigen::Matrix4f::Identity();
    GetRightRobotToCameraMatrix(Tcameratorobot_right);

    Eigen::Matrix3f Rcameratorobot_right;
    Rcameratorobot_right(0, 0) = -0.0416;
    Rcameratorobot_right(0, 1) = 0.0125;
    Rcameratorobot_right(0, 2) = 0.9991;
    Rcameratorobot_right(1, 0) = 0.9989;
    Rcameratorobot_right(1, 1) = -0.0215;
    Rcameratorobot_right(1, 2) = 0.0418;
    Rcameratorobot_right(2, 0) = 0.0222;
    Rcameratorobot_right(2, 1) = 0.9997;
    Rcameratorobot_right(2, 2) = -0.0116;

    Eigen::Vector4f TanslationInCamera_right(translateX_RIGHT, translateY_RIGHT,
    translateZ_RIGHT, 0); // Translation correction in camera frame
    Eigen::Vector4f TanslationInRobot_right;
    TanslationInRobot_right = Tcameratorobot_right*TanslationInCamera_right; //
    Translation correction in robot frame

    Eigen::Matrix3f RotationInRobot_right;
```

```
    RotationInRobot_right = Rcameratorobot_right*AlignmentRotation_correction_right; //Rotation corrections for robot

    cout << "In the right robot coordinate system det translation equals: \n" << "x " <<
      TanslationInRobot_right(0) << "y " << TanslationInRobot_right(1) << "z " << TanslationInRobot_right(2) << endl;

    //final transformation matrix which is used to correct the end effector of the robot:

    Tcameratorobot_right(0, 0) = Rcameratorobot_right(0, 0);
    Tcameratorobot_right(0, 1) = Rcameratorobot_right(0, 1);
    Tcameratorobot_right(0, 2) = Rcameratorobot_right(0, 2);
    Tcameratorobot_right(0, 3) = TanslationInRobot_right(0);
    Tcameratorobot_right(1, 0) = Rcameratorobot_right(1, 0);
    Tcameratorobot_right(1, 1) = Rcameratorobot_right(1, 1);
    Tcameratorobot_right(1, 2) = Rcameratorobot_right(1, 2);
    Tcameratorobot_right(1, 3) = TanslationInRobot_right(1);
    Tcameratorobot_right(2, 0) = Rcameratorobot_right(2, 0);
    Tcameratorobot_right(2, 1) = Rcameratorobot_right(2, 1);
    Tcameratorobot_right(2, 2) = Rcameratorobot_right(2, 2);
    Tcameratorobot_right(2, 3) = TanslationInRobot_right(2);
    Tcameratorobot_right(3, 0) = 0;
    Tcameratorobot_right(3, 1) = 0;
    Tcameratorobot_right(3, 2) = 0;
    Tcameratorobot_right(3, 3) = 1;

    write4x4Matrix_RIGHT(Tcameratorobot_right);
    //--------Translation and Rotation in the robot coordinate system RIGHT END--------

    //For visualization with values not equal to the one given to the robot:

    //-------Translate the centroid back to the original position right start--------
```

```
Eigen::Affine3f transform_centroid_back_right(Eigen::Affine3f::Identity());
transform_centroid_back_right.translation() << centroid_RIGHT(0),
centroid_RIGHT(1), centroid_RIGHT(2);
pcl::transformPointCloud(*scene_ready_RIGHT, *scene_ready_RIGHT,
transform_centroid_back_right);

//---------Translate the centroid back to the original position right end
-----------

//--------Translate the tube captured with the values found above START
----------
Xvalue_RIGHT = Xvalue_RIGHT + centroid_RIGHT(0);
ValueYY_RIGHT = ValueYY_RIGHT + centroid_RIGHT(1);
ValueZZ_RIGHT = ValueZZ_RIGHT + centroid_RIGHT(2);
float translateXfake_RIGHT = -Xvalue_RIGHT;
float translateYfake_RIGHT = -ValueYY_RIGHT;
float translateZfake_RIGHT = (LengthFromCamera - TubeRadius) - ValueZZ_RIGHT;

Eigen::Affine3f transform_SearchMethod_RIGHT = Eigen::Affine3f::Identity();

// Define the translation
transform_SearchMethod_RIGHT.translation() << translateXfake_RIGHT,
translateYfake_RIGHT, translateZfake_RIGHT;

// Executing the transformation
pcl::PointCloud<pcl::PointXYZ>::Ptr transformed_cloud_RIGHT(new pcl::PointCloud
<pcl::PointXYZ>());
pcl::transformPointCloud(*scene_ready_RIGHT, *transformed_cloud_RIGHT,
transform_SearchMethod_RIGHT);

writer.write<pcl::PointXYZ>("5 rotert og translert RIGHT.pcd", *
transformed_cloud_RIGHT, false);

//--------Translate the tube captured with the values found above end ------

scene_alignedRun_RIGHT = *transformed_cloud_RIGHT;
```

```
}
//Method for only SearchMethod
void OnlySearchMethod(){

  //PointClouds for the environment:

  pcl::PointCloud<pcl::PointXYZ>::Ptr scene_ready_RIGHT(new pcl::PointCloud<pcl::
  PointXYZ>);




  //Get the point cloud captured by the GrabAndFilter
  *scene_ready_RIGHT = scene_filtered_RIGHT;

  //---------------- SEARCH METHOD RIGHT SIDE - START-----------------------------
  float Xvalue_RIGHT = 0;
  float ValueYY_RIGHT = 0;
  float ValueZZ_RIGHT = 0;
  //Start SearchMethod for left tube
  SearchMethod_RIGHT(scene_ready_RIGHT, Xvalue_RIGHT, ValueYY_RIGHT,
  ValueZZ_RIGHT);
  //---------------- SEARCH METHOD RIGHT SIDE - END--------------------------------

  //----------------Translate the right tube captured with the values found above
  START --------------------
  float translateX_RIGHT = -Xvalue_RIGHT;
  float translateY_RIGHT = -ValueYY_RIGHT;
  float translateZ_RIGHT = (LengthFromCamera - TubeRadius) - ValueZZ_RIGHT;
  Eigen::Affine3f transform_SearchMethod_RIGHT = Eigen::Affine3f::Identity();
  // Define the translation
  transform_SearchMethod_RIGHT.translation() << translateX_RIGHT,
  translateY_RIGHT, translateZ_RIGHT;
  // Executing the transformation
  pcl::PointCloud<pcl::PointXYZ>::Ptr transformed_cloud_RIGHT(new pcl::PointCloud
```

```
<pcl::PointXYZ>());
pcl::transformPointCloud(*scene_ready_RIGHT, *transformed_cloud_RIGHT, transform_SearchMethod_RIGHT);
//----------------Translate the right tube captured with the values found above END -----------------------

//--------------Find the correction translation in the robot frame and write it to file right START --------------
Eigen::Vector4f TanslationInCamera_right(translateX_RIGHT, translateY_RIGHT, translateZ_RIGHT, 0);
Eigen::Vector4f TanslationInRobot_right;
Eigen::Matrix4f Tcameratorobot_right = Eigen::Matrix4f::Identity();
GetRightRobotToCameraMatrix(Tcameratorobot_right);
TanslationInRobot_right = Tcameratorobot_right*TanslationInCamera_right; // The translation given to the robot
Eigen::Matrix4f transformation_matrix_searchMethod_RIGHT = Eigen::Matrix4f::Identity();
//There is no rotation using this method so the rotation matrix in the transformation matrix is just an identity matrix.
transformation_matrix_searchMethod_RIGHT(0, 0) = 1;
transformation_matrix_searchMethod_RIGHT(0, 1) = 0;
transformation_matrix_searchMethod_RIGHT(0, 2) = 0;
transformation_matrix_searchMethod_RIGHT(0, 3) = TanslationInRobot_right(0);
transformation_matrix_searchMethod_RIGHT(1, 0) = 0;
transformation_matrix_searchMethod_RIGHT(1, 1) = 1;
transformation_matrix_searchMethod_RIGHT(1, 2) = 0;
transformation_matrix_searchMethod_RIGHT(1, 3) = TanslationInRobot_right(1);
transformation_matrix_searchMethod_RIGHT(2, 0) = 0;
transformation_matrix_searchMethod_RIGHT(2, 1) = 0;
transformation_matrix_searchMethod_RIGHT(2, 2) = 1;
transformation_matrix_searchMethod_RIGHT(2, 3) = TanslationInRobot_right(2);
transformation_matrix_searchMethod_RIGHT(3, 0) = 0;
transformation_matrix_searchMethod_RIGHT(3, 1) = 0;
transformation_matrix_searchMethod_RIGHT(3, 2) = 0;
transformation_matrix_searchMethod_RIGHT(3, 3) = 1;
write4x4Matrix_RIGHT(transformation_matrix_searchMethod_RIGHT);
```

```
    //------------Find the correction translation in the robot frame and write it to file LEFT END ---------------

    scene_alignedRun_RIGHT = *transformed_cloud_RIGHT;
}

void  runAlignment(char* argv[]){

    //Check which of the alignment methods that has been chosen:
    if (strcmp(argv[1], "i") == 0)
    {
      cout << "SAC-IA and ICP are going to align" << endl;
      ICPalgorithm();
    }

    if (strcmp(argv[1], "r") == 0)
    {
      cout << "RANSAC and SearchMethod are going to align" << endl;
      SearchForEndOfTubeandRANSAC();

    }
    if (strcmp(argv[1], "h") == 0)
    {
      cout << "Only SearchMethod for translation" << endl;
      OnlySearchMethod();
    }

    //------Write bool to file to tell the java-script the tube is found  start -----

    ofstream myfile;
    myfile.open("startJAVA.txt");
    myfile << "TRUE \n";
    myfile.close();

    //----Write bool to file to tell the java-script the tube is found  end -----
```

```

//Visualization of a live stream where the wanted and actual point clouds are.
//----------------------------Visualization START----------------------

boost::function<void(const ConstPtr&) > callback = boost::bind(&AlignmentViewerStream::cloud_callback, this, _1);
boost::signals2::connection connection = grabber.registerCallback(callback);

grabber.start();
//The point clouds added to the visualization
pcl::PointCloud<pcl::PointXYZ>::Ptr cloud_transformed_viz_right(new pcl::PointCloud<pcl::PointXYZ>);
pcl::PointCloud<pcl::PointXYZ>::Ptr cloud_original_viz_right(new pcl::PointCloud<pcl::PointXYZ>);
*cloud_transformed_viz_right = scene_alignedRun_RIGHT;
*cloud_original_viz_right = scene_filtered_LEFT;

//IF RANSAC is chosen add a coordinate system on M_edge with the orientation of the tubes
if (strcmp(argv[1], "r") == 0)
{
  // Add the coordinate system for the captured cylinder using RANSAC.
  Eigen::Affine3f tt_right;
  tt_right = Eigen::Translation3f(wanted_RIGHT(0), wanted_RIGHT(1), wanted_RIGHT(2))*Eigen::AngleAxisf(ea_robot_right(0), Eigen::Vector3f::UnitX())*Eigen::AngleAxisf(ea_robot_right(1), Eigen::Vector3f::UnitY())*Eigen::AngleAxisf(ea_robot_right(2), Eigen::Vector3f::UnitZ());
  viewer->addCoordinateSystem(1, tt_right);
    }

// Add the point cloud to the viewer and pass the color handler
//Where the tubes actually are:
pcl::visualization::PointCloudColorHandlerCustom<pcl::PointXYZ> source_cloud_color_handler_RIGHT(cloud_original_viz_right, 255, 255, 0);
viewer->addPointCloud(cloud_original_viz_right, source_cloud_color_handler_RIGHT, "Where the right actually is");
```

```
    viewer->setPointCloudRenderingProperties(pcl::visualization::PCL_VISUALIZER_POINT_SIZE, 4, "Where the right actually is");

    //Where the tubes are transformed to:
    pcl::visualization::PointCloudColorHandlerCustom<pcl::PointXYZ> source_cloud_color_handler_transformed_cloud_RIGHT(cloud_transformed_viz_right, 255, 255, 0);
    viewer->addPointCloud(cloud_transformed_viz_right, source_cloud_color_handler_transformed_cloud_RIGHT, "Where the right tube is transported");
    viewer->setPointCloudRenderingProperties(pcl::visualization::PCL_VISUALIZER_POINT_SIZE, 2, "Where the right tube is transported");

    //Stream the visualization:
    while (!viewer->wasStopped()){
      viewer->spinOnce();
      viewer->setCameraPosition(-0.0684716, 1.77067, -2.51146, -0.193301, 0.992061, -0.659233, 0.0155019, 0.921378, 0.388357); //Camera position

      ConstPtr cloud;

      if (mutex.try_lock()){
        buffer.swap(cloud);
        mutex.unlock();
      }

      if (cloud){
        if (!viewer->updatePointCloud(cloud, "Cloud")){
          viewer->addPointCloud(cloud, "Cloud");
          viewer->resetCameraViewpoint("Cloud");
        }
      }

      if (GetKeyState(VK_ESCAPE) < 0){
        break;
      }
```

```
    }

    grabber.stop();

    if (connection.connected()){
      connection.disconnect();
    }
  }


private:
  void cloud_callback(const ConstPtr& cloud)
  {
    boost::mutex::scoped_lock lock(mutex);
    buffer = cloud;
  }
  Eigen::Vector4f wanted_RIGHT;
  Eigen::Vector3f ea_robot_right;

  boost::shared_ptr<pcl::visualization::PCLVisualizer> viewer;
  pcl::Grabber& grabber;
  boost::mutex mutex;
  ConstPtr buffer;

  pcl::PointCloud<pcl::PointXYZ> scene_alignedRun_RIGHT;
  pcl::PointCloud<pcl::PointXYZ> scene_filtered_RIGHT;

};

pcl::PointCloud<pcl::PointXYZ >::Ptr scene_filtered_RIGHT_main(new pcl::PointCloud<
    pcl::PointXYZ>);

float vectorOrientation[3];

int main(int argc, char* argv[])
{
```

```
    ofstream myfile;
    myfile.open("startJAVA.txt");
    myfile << "FALSE \n";
    myfile.close();

    //Grab point cloud and filter it
    ViewGrapAndFilter v;
    v.RunControl();

    //Get the point clouds from ViewGrapAndFilter and store them in
      scene_filtered_LEFT_main and  scene_filtered_RIGHT_main
    v.GetFilteredPointCloud(scene_filtered_LEFT_main, scene_filtered_RIGHT_main);

    boost::shared_ptr<pcl::Grabber> grabber = boost::make_shared<pcl::Kinect2Grabber
      >();
    AlignmentViewerStream<pcl::PointXYZRGB> viewer(*grabber);

    //Pass the point clouds to  AlignmentViewerStream object viewer
    viewer.PassClouds(scene_filtered_LEFT_main, scene_filtered_RIGHT_main);
    viewer.runAlignment(argv);
    return 0;
}
// That's it!
```

Listing A.2: Main.cpp

A.3.2 C++ Functions source file

```

/*
- Author: Simen Hagen Bredvold
- Master Thesis NTNU IPK 2016

This Source File holds all the functions used.
*/

#define _SCL_SECURE_NO_WARNINGS
#define _CRT_SECURE_NO_WARNINGS

#include "Functions.h"

//Downsamlping function definition:
void DownsamplingFilter(const pcl::PointCloud<pcl::PointXYZ>::ConstPtr&
    cloudtobefiltered, const pcl::PointCloud<pcl::PointXYZ>::Ptr cloudfiltered,
    const float gridsize){

  pcl::VoxelGrid<pcl::PointXYZ> sor;
  sor.setInputCloud(cloudtobefiltered);
  sor.setLeafSize(gridsize, gridsize, gridsize+0.05f); // It is added 0.05m in the
    z-axis because of variation in depth values. T
  sor.filter(*cloudfiltered);
}
//Passthrough function definition:
void PassthroughFilter(const pcl::PointCloud<pcl::PointXYZ>::ConstPtr&
    cloudtobefiltered, const pcl::PointCloud<pcl::PointXYZ>::Ptr cloudfiltered,
    const float minrange, const float maxrange, char xyz){
  pcl::PassThrough<pcl::PointXYZ> pass;
  pass.setInputCloud(cloudtobefiltered);
  if (xyz=='x')
  {
    pass.setFilterFieldName("x");
  }
```

```
  if (xyz=='y')
  {
    pass.setFilterFieldName("y");
  }
  if (xyz == 'z')
  {
    pass.setFilterFieldName("z");
  }
  pass.setFilterLimits(minrange, maxrange);
  pass.filter(*cloudfiltered);
}
// Move least surface smoothning algorithm
void MLS(const pcl::PointCloud<pcl::PointXYZ>::ConstPtr& cloudtobefiltered, const
    pcl::PointCloud<pcl::PointXYZ>::Ptr cloudfiltered, const float SearchRadius){

  // Create a KD-Tree
  pcl::search::KdTree<pcl::PointXYZ>::Ptr tree(new pcl::search::KdTree<pcl::
    PointXYZ>);

  // Output has the PointNormal type in order to store the normals calculated by
    MLS
  pcl::PointCloud<pcl::PointNormal> scene_mls_points;

  // Init object (second point type is for the normals, even if unused)
  pcl::MovingLeastSquares<pcl::PointXYZ, pcl::PointNormal> mls;
  mls.setComputeNormals(true);

  // Set parameters
  mls.setInputCloud(cloudtobefiltered);
  mls.setPolynomialFit(true);
  mls.setSearchMethod(tree);
  mls.setSearchRadius(SearchRadius);

  // Reconstruct
  mls.process(scene_mls_points);

```

```
  // Save output
  pcl::io::savePCDFile("tube_mls.pcd", scene_mls_points);
  Sleep(5000);
  pcl::io::loadPCDFile("tube_mls.pcd", *cloudfiltered);
}
void
print4x4Matrix(const Eigen::Matrix4f & matrix)
{
  printf("Rotation matrix :\n");
  printf("    | %6.3f %6.3f %6.3f | \n", matrix(0, 0), matrix(0, 1), matrix(0, 2));
  printf("R = | %6.3f %6.3f %6.3f | \n", matrix(1, 0), matrix(1, 1), matrix(1, 2));
  printf("    | %6.3f %6.3f %6.3f | \n", matrix(2, 0), matrix(2, 1), matrix(2, 2));
  printf("Translation vector :\n");
  printf("t = < %6.3f, %6.3f, %6.3f >\n\n", matrix(0, 3), matrix(1, 3), matrix(2,
    3));
}

void write4x4Matrix_RIGHT(const Eigen::Matrix4f & matrix)
{
  ofstream fout("transformationmatrix_RIGHT.txt"); // writes matrix with the given
    name
  /*
  fout << "Rotation matrix :\n";
  fout << "    |" << matrix(0, 0) << matrix(0, 1) << matrix(0, 2) << "| \n";
  fout << "R = |" << matrix(1, 0) << matrix(1, 1) << matrix(1, 2) << "| \n";
  fout << "    |" << matrix(2, 0) << matrix(2, 1) << matrix(2, 2) << "| \n", matrix
    (2, 0), matrix(2, 1), matrix(2, 2);
  fout << "Translation vector :\n";
  fout << "t = <" << matrix(0, 3) << matrix(1, 3) << matrix(2, 3) << ">\n\n\n";
  */
  std::locale mylocale("");   // get global locale
  fout.imbue(mylocale);
  fout << matrix(0, 0) << " " << matrix(0, 1) << " " << matrix(0, 2) << " " <<
    matrix(0, 3) * 1000 << "\n";
  fout << matrix(1, 0) << " " << matrix(1, 1) << " " << matrix(1, 2) << " " <<
    matrix(1, 3) * 1000 << "\n";
```

```
  fout << matrix(2, 0) << " " << matrix(2, 1) << " " << matrix(2, 2) << " " << matrix(2, 3) * 1000 << "\n";
  fout << 0.000000 << " " << 0.000000 << " " << 0.000000 << " " << 1.000000 << "\n";
}

void SearchMethod_RIGHT(const pcl::PointCloud<pcl::PointXYZ>::ConstPtr& Searchmethodcloud, float& x, float& y, float& z){

  //-----------Compute normals start -----------------------------------
  // Estimate point normals
  pcl::NormalEstimation<PointT, pcl::Normal> ne;
  pcl::search::KdTree<PointT>::Ptr tree(new pcl::search::KdTree<PointT>());
  ne.setSearchMethod(tree);
  ne.setInputCloud(Searchmethodcloud);
  ne.setKSearch(50);
  pcl::PointCloud<pcl::Normal>::Ptr cloud_normals(new pcl::PointCloud<pcl::Normal>);
  ne.compute(*cloud_normals);
  //--------------------Compute normals end ------------------------


  //---------------search for the end of the tube start--------------------------
  //Start with a large negative x value and large z-value for the search
  float minX = -1000;
  float minZ = 100000;
  float minY;

  float currentX, currentY, currentZ;
  for (size_t i = 0; i < Searchmethodcloud->points.size(); ++i){

    currentX = Searchmethodcloud->points[i].x; //gets the x-value for point i
    currentY = Searchmethodcloud->points[i].y;//gets the y-value for point i
    currentZ = Searchmethodcloud->points[i].z;//gets the z-value for point i

    if (currentX >= minX) // If the current x-value is larger then the stored minX
```

```
    {
      minX = currentX;
      if (currentZ < minZ)// If the current z-value is larger then the stored minZ
      {
        minZ = currentZ;
        minY = currentY;

      }
    }
}

//---------------search for the end of the tube END--------------------------

//-------------Search method dividing into interval START----------------------

float treshold = 0.005; // distance between each interval
float searchX, searchY, searchZ, searchZnormal;
float  holdX, holdY, holdZ, holdZnormal;
holdZ = 1000;
float scoreArray[101][5];
int counter = 0;
int score = 1;
minX = minX + treshold;
bool FirstOver10=FALSE;
float SumX=0;
float Xvalue = 0.0;

// Create 100 intervals and for each of the intervals find the point with
// lowest z-value, its normal vector, x-value, y-value and how many point inside the given interval
for (float start = minX; start > (minX - 0.5); start = start - treshold)
{

  for (size_t i = 0; i < Searchmethodcloud->points.size(); ++i)
  {
    searchX = Searchmethodcloud->points[i].x;//gets the x-value for point i
```

```
      searchY = Searchmethodcloud->points[i].y;//gets the y-value for point i
      searchZ = Searchmethodcloud->points[i].z;//gets the z-value for point i
      searchZnormal = cloud_normals->points[i].normal_z; //gets normal vector in z-direction

      if (searchX<start && searchX>start - treshold) //filters point not inside the interval
      {
        score = score + 1; //Counts how many points are inside the interval

        if (holdZ > searchZ) //store the closest z-value
        {
          holdX = searchX;
          holdZ = searchZ;
          holdY = searchY;
          holdZnormal = searchZnormal;
        }

      }

    }
    // Find the sum of the x values inside the first interval containing more points than 10.
    if ((score>10) && (FirstOver10==FALSE))
    {
      for (size_t i = 0; i < Searchmethodcloud->points.size(); ++i){

        searchX = Searchmethodcloud->points[i].x;//gets the x-value for point i
        if (searchX<start && searchX>start - treshold) //filters point not inside the interval
        {
          SumX = SumX + searchX;

        }
      }
      FirstOver10 = TRUE; // To never enter this loop again
```

```
      Xvalue = SumX / score; // The mean value of the x-values
    }


    scoreArray[counter][0] = start; // X-value for the point chosen
    scoreArray[counter][1] = holdY; // y-value for the point chosen
    scoreArray[counter][2] = holdZ; // z-value for the point chosen
    scoreArray[counter][3] = holdZnormal; // z-normal for the point chosen
    scoreArray[counter][4] = score; //number of point inside a interval
    holdZ = 1000; //reset variable
    score = 1;//reset variable
    counter = counter + 1;
  }

  // Check which intervals that contain bore than 10 points and have normal
  // vectors above 0.99. Calulates the mean of the 8 first value. Check for
    outliers;
  counter = 0;
  float Values[2][8];
  float sumY = 0.0, sumZ = 0.0;
  float Yvalues[8], Zvalues[8];
  for (size_t i = 0; i < 101; i++)
  {
    //filter away all intervals not having over 10 point in its interval and a
    normal vector value below 0.99
    if ((scoreArray[i][4]>10) && (abs(scoreArray[i][3]))>0.99)
    {
      sumY = sumY + scoreArray[i][1];
      sumZ = sumZ + scoreArray[i][2];
      Values[0][counter] = scoreArray[i][1]; // y value
      Values[1][counter] = scoreArray[i][2]; //z value

      Yvalues[counter] = scoreArray[i][1]; // y value for z-score
      Zvalues[counter] = scoreArray[i][2]; //z value for z-score
      counter = counter + 1;

```

```
      if (counter >7) // Breaks the loop if 8 intervals are obtained.
      {
        break;
      }
    }
  }


  //---------------Modified z-score for outlier detection START-----------------
  //Find the median of the 8 stored y z values in Values_RIGHT:
  // Sort the values by size.
  std::sort(Yvalues, Yvalues + 8);
  std::sort(Zvalues, Zvalues + 8);
  float medianY, medianZ;
  medianY = (Yvalues[3] + Yvalues[4]) / 2.0; // the median of 8 is the sum of the
    third and fourth value devided on 2
  medianZ = (Zvalues[3] + Zvalues[4]) / 2.0;

  float AbsDeviationY[8];
  float AbsDeviationZ[8];
  // Compute the absolute deviation about medianY and medianZ.
  for (size_t c = 0; c < 8; c++)
  {
    AbsDeviationY[c] = abs(Yvalues[c] - medianY);
    AbsDeviationZ[c] = abs(Zvalues[c] - medianZ);
  }
  std::sort(AbsDeviationY, AbsDeviationY + 8);
  std::sort(AbsDeviationZ, AbsDeviationZ + 8);
  float srtAbsDevY = (AbsDeviationY[3] + AbsDeviationY[4]) / 2.0;
  float srtAbsDevZ = (AbsDeviationZ[3] + AbsDeviationZ[4]) / 2.0;

  float ZscoreY[8];
  float ZscoreZ[8];

  //compute the z-scores
  for (size_t z = 0; z < 8; z++)
```

```
  {
    ZscoreY[z] = (0.6745*(Yvalues[z] - medianY)) / srtAbsDevY;
    ZscoreZ[z] = (0.6745*(Zvalues[z] - medianZ)) / srtAbsDevZ;
  }

  //calculate the mean for the point not having a z-score above 3.5
  float sumY_Zscore = 0.0;
  float sumZ_Zscore = 0.0;
  int passedY = 0;
  int passedZ = 0;
  for (size_t score = 0; score < 8; score++)
  {
    if (ZscoreY[score]<3.5)
    {
      sumY_Zscore = sumY_Zscore + Yvalues[score];
      passedY = passedY + 1;
    }
    if (ZscoreZ[score]<3.5)
    {
      sumZ_Zscore = sumZ_Zscore + Zvalues[score];
      passedZ = passedZ + 1;
    }

  }
  float meanY_Zscore = sumY_Zscore / passedY;
  float meanZ_Zscore = sumZ_Zscore / passedZ;

  //-----------------Modified z-score for outlier detection END---------------------

  //return the value
  x = Xvalue;
  y = meanY_Zscore;
  z = meanZ_Zscore;
}

void GetLeftRobotToCameraMatrix(Eigen::Matrix4f& matrix){
```

```
  matrix(0, 0) = -0.0443;
  matrix(0, 1) = 0.0230;
  matrix(0, 2) = 0.9987;
  matrix(0, 3) = 0;
  matrix(1, 0) = 0.9988;
  matrix(1, 1) = -0.0184;
  matrix(1, 2) = 0.0447;
  matrix(1, 3) = 0;
  matrix(2, 0) = 0.0144;
  matrix(2, 1) = 0.9996;
  matrix(2, 2) = -0.0224;
  matrix(2, 3) = 0;
  matrix(3, 0) = 0;
  matrix(3, 1) = 0;
  matrix(3, 2) = 0;
  matrix(3, 3) = 1;
}

void GetRightRobotToCameraMatrix(Eigen::Matrix4f& matrix){
  matrix(0, 0) = -0.0416;
  matrix(0, 1) = 0.0125;
  matrix(0, 2) = 0.9991;
  matrix(0, 3) = 0;
  matrix(1, 0) = 0.9989;
  matrix(1, 1) = -0.0215;
  matrix(1, 2) = 0.0418;
  matrix(1, 3) = 0;
  matrix(2, 0) = 0.0222;
  matrix(2, 1) = 0.9997;
  matrix(2, 2) = -0.0116;
  matrix(2, 3) = 0;
  matrix(3, 0) = 0;
  matrix(3, 1) = 0;
  matrix(3, 2) = 0;
  matrix(3, 3) = 1;
}
```

Listing A.3: Functions.cpp

A.3.3 C++ Functions header file

```
/*
- Author: Simen Hagen Bredvold
- Master's Thesis NTNU IPK 2016

This Header File holds functions declarations and libraries used.
*/

  #include <iostream>
  #include <string>
  #include <sstream>
  #include <tchar.h>
  #include <math.h>
  #include <cmath>
    #include <algorithm>
  #include <pcl/io/pcd_io.h>
  #include <pcl/io/ply_io.h>
  #include <pcl/point_types.h>
  #include <pcl/filters/voxel_grid.h>
  #include <pcl/filters/statistical_outlier_removal.h>
  #include <pcl/filters/passthrough.h>
  #include <pcl/registration/icp.h>
  #include <pcl/visualization/pcl_visualizer.h>
  #include <pcl/visualization/cloud_viewer.h>
  #include <pcl/kdtree/kdtree_flann.h>
  #include <pcl/kdtree/impl/kdtree_flann.hpp>
  #include <pcl/surface/mls.h>
  #include <pcl/keypoints/uniform_sampling.h>
  #include <pcl/point_cloud.h>
  #include <pcl/common/io.h>
  #include <pcl/correspondence.h>
  #include <pcl/features/normal_3d_omp.h>
  #include <pcl/features/shot_omp.h>
  #include <pcl/features/board.h>
  #include <pcl/recognition/cg/hough_3d.h>
```

```
  #include <pcl/recognition/cg/geometric_consistency.h>
  #include <pcl/common/transforms.h>
  #include <pcl/console/parse.h>
  #include <Eigen/StdVector>
  #include <pcl/keypoints/sift_keypoint.h>
  #include <pcl/features/normal_3d.h>
  #include <pcl/features/pfh.h>
  #include <Eigen/SVD>
  #include <pcl/common/transformation_from_correspondences.h>
  #include <pcl/registration/ia_ransac.h>
  #include <pcl/registration/correspondence_rejection_sample_consensus.h>
  #include <pcl/registration/correspondence_rejection_one_to_one.h>
  #include <pcl/keypoints/harris_3d.h>
  #include <pcl/ModelCoefficients.h>
  #include <pcl/filters/extract_indices.h>
  #include <pcl/segmentation/sac_segmentation.h>
  #include <pcl/sample_consensus/model_types.h>
  #include <pcl/sample_consensus/method_types.h>
  #include <float.h>
  #include <boost/thread/thread.hpp>
  #include <pcl/common/common_headers.h>
  #include <pcl/features/fpfh.h>

//short handings
typedef pcl::PointXYZ PointT;
typedef pcl::PointXYZ PointType;
typedef pcl::Normal NormalType;
typedef pcl::ReferenceFrame RFType;
typedef pcl::SHOT352 DescriptorType;

//Downsample decleration:
void DownsamplingFilter(const pcl::PointCloud<pcl::PointXYZ>::ConstPtr&
    cloudtobefiltered, const pcl::PointCloud<pcl::PointXYZ>::Ptr cloudfiltered,
    const float gridsize);

//Passthrough decleration:
```

```
void PassthroughFilter(const pcl::PointCloud<pcl::PointXYZ>::ConstPtr&
    cloudtobefiltered, const pcl::PointCloud<pcl::PointXYZ>::Ptr cloudfiltered,
    const float minrange, const float maxrange, char xyz);


//MLS filter decleration:
void MLS(const pcl::PointCloud<pcl::PointXYZ>::ConstPtr& cloudtobefiltered, const
    pcl::PointCloud<pcl::PointXYZ>::Ptr cloudfiltered, const float SearchRadius);

//SearchMethod for right robot decleration:
void SearchMethod_RIGHT(const pcl::PointCloud<pcl::PointXYZ>::ConstPtr&
    Searchmethodcloud, float& x, float& y, float& z); //& pass by reference

//Prints to screen matrix
void print4x4Matrix(const Eigen::Matrix4f & matrix);
//Write 4x4 matrix to file for both left and right
void write4x4Matrix_RIGHT(const Eigen::Matrix4f & matrix);

//The transformation matrix between robot and camera for right robo
void GetRightRobotToCameraMatrix(Eigen::Matrix4f&  matrix);
```

Listing A.4: Functions.h

A.4 Java code

A.4.1 GUI Source code

```

// Author: Simen HAgen Bredvold
// Master's Thesis NTNU IPK 2016

// For fully run the robots and solutuin in auto mode this
    code is // run to sequence robot programs,
// start Matlab and C++ application, read/write to robot,
    ask if
// the new poses are OK or not, start/stop welding. This is
    done by // calling the class RobotConnection and its
    subclasses RobotKR120 // and RobotKR240.

        //Signal Matlab Safety application to start:
        try {
            String str = "TRUE";
            File newTextFile = new File("MatlabStarter.txt");

            FileWriter fw = new FileWriter(newTextFile);
            fw.write(str);
            fw.close();
        }
        catch (IOException iox) {
            iox.printStackTrace();
        }

        //Close the grippers for both robots.
```

```
        PLCConnection.closeGripper120();
        PLCConnection.closeGripper240();
        // pick up left tube
        System.out.println("Pick up the tubes");
        KR120.startPickUp120();
        try{
             Thread.sleep(1000);
        }
        catch(Exception e){
        }
        KR240.startPickUp240();
        try{
             Thread.sleep(1000);
        }
        catch(Exception e){
        }

      // wait for the two pick up programs to finished
        while(KR120.PickUpRunning()&& KR240.PickUpRunning()){
            try{
                Thread.sleep(500);
            }
            catch(Exception e){
            }
        }
        //open the grippers to grab the tubes.
        System.out.println("Open the Grippers");
         PLCConnection.openGripper120();
         PLCConnection.openGripper240();

```

```
        //to find the length of the tube. After pick up read the pose and the z-values.
        double[] pose_left,pose_right;
        pose_left=KR120.readPose_LEFT();
        pose_right=KR240.readPose_RIGHT();

        //Tell matlab that the tubes are picked up and the length of the tubes.
        try {
            double z_left=pose_left[2];
            double z_right=pose_right[2];
            String str = "TRUE" + z_left + z_right;
            File newTextFile = new File("MatlabTubePickedUp.txt");
            FileWriter matlabobject = new FileWriter(newTextFile);
            matlabobject.write(str);
            matlabobject.close();

       } catch (IOException iox) {
            //do stuff with exception
            iox.printStackTrace();
       }


       //Move the tubes infront of the camera
       System.out.println("Place the tubes infront of the camera");
       KR120.startSetCamera120();
       try{
```

```
                Thread.sleep(1000);
            }
            catch(Exception e){
            }

            KR240.startSetCamera240();
            try{
                Thread.sleep(1000);
            }
            catch(Exception e){
            }
            //wait for the set infront of camera programs to
        finish
            while(KR120.SetCameraRunning()&&
        KR240.SetCameraRunning()){
                try{
                    Thread.sleep(500);
                }
                catch(Exception e){
                }
            }
           // start the MultiPicture.exe file which is the c++
        application for
           // pose correction of tubes.
            try{
            Process process = new
        ProcessBuilder("C:\\Users\\simen_000\\Desktop\\Skole\\Master\\C++
          for innlevering
        10mai\\Win32Project1\\x64\\Debug\\MultiPicture.exe","h").start();

```

```
            Thread.sleep(5000);
            boolean end=false;
            int i =1;
            System.out.println("The C++ application is
    running...");
           //While loop which runs until the C++ application
    is finished.
            while (!end){

                   Thread.sleep(1000);

                   Path filePath =
    Paths.get("C:\\Users\\simen_000\\Desktop\\NetBeansProjectsmededit\\Ja
                   Scanner input = new Scanner(filePath);
                   while(input.hasNext()) {
                           String word = input.next();
                           if(word.equals("TRUE")){
                               end=true;
                               System.out.println("The
    application is finished");
                           } //if end

                   } //while end

                 } //while end

            } // outter try end
            catch(Exception e){
          }

```

```
        //Tell matlab safety program to start.
        try {
            String str = "TRUE";
            File newTextFile = new
    File("MatlabSafetyProgram.txt");

            FileWriter fw = new FileWriter(newTextFile);
            fw.write(str);
            fw.close();

        } catch (IOException iox) {
            //do stuff with exception
            iox.printStackTrace();
        }

        //Ask operator if the new poses are ok!

         int choice = JOptionPane.showOptionDialog(null,
        "Are the new poses OK?",
         "Matlab Safety Program",
        JOptionPane.YES_NO_OPTION,
         JOptionPane.QUESTION_MESSAGE,
        null, null, null);

        // interpret the user's choice
        if (choice == JOptionPane.NO_OPTION)
        {
            System.exit(0);
        }

```

```
        // Now read the transformation matrix from the application above.
        // Then write the new position to the variable in the KR controller
        // called "NEW_POINT" which is stored in config.dat in KR120 LEFT robot.
        KR240.writeEEposRIGHT();
        KR120.writeEEposLEFT();

        // Set UpgratePosition (KR variable) to true to start the robot program
        // SETPOSITION which moves the robot to the new position.

        System.out.println("Move the tubes to the wanted position");
        KR120.startSETPOSITION();
        try{
            Thread.sleep(1000);
        }
        catch(Exception e){

        }
        while(KR120.SETPOSITIONRunning()){
            try{
                Thread.sleep(500);
            }
            catch(Exception e){

            }
```

```
        }

        KR240.startSETPOSITIONright();
        try{
             Thread.sleep(1000);
        }
        catch(Exception e){

        }
        while(KR240.SETPOSITIONRunningRIGHT()){
            try{
                Thread.sleep(500);
            }
            catch(Exception e){

            }
        }

        // Start the tack welding program for the KR16 robot.
        KR16.startFive();
         try{
             Thread.sleep(1000);
        }
        catch(Exception e){
        }

         int i = 0;
        while(KR16.fiveRunning()){

            boolean weld_sh = false;
```

```
            while(i < 3){
                if(KR16.isWelding() && !weld_sh){
                    KR5.startWelding();
                    System.out.println("Started welding sequence: "+(i+1));
                    weld_sh = true;
                }

                if(!KR16.isWelding() && weld_sh){
                    KR5.stopWelding();
                    System.out.println("Stopped welding");
                    weld_sh = false;
                    i++;
                }
            }
        }
```

Listing A.5: GUI Source code for controlling the process

A.4.2 Class RobotConnection for reading/writing to robot

```

// Author: Simen HAgen Bredvold
// Master's Thesis NTNU IPK 2016

// This class declares the methods for connecting with the robot.
// Also, methods for reading/writing to system global variables and
// methods for reading the transformation matrix from the C++
// application and method obtain the RPY-angles.

package controlsystem;

import java.io.*;
import java.io.File;
import java.io.FileNotFoundException;
import java.util.ArrayList;
import java.util.Scanner;
import java.nio.file.Path;
import java.nio.file.Paths;
import java.util.*;
import java.util.Arrays;
import java.util.List;
import java.lang.*;
import java.io.IOException;
import java.net.UnknownHostException;
import no.hials.crosscom.CrossComClient;
import no.hials.crosscom.KRL.KRLBool;
```

```java
import no.hials.crosscom.KRL.structs.KRLFrame;
import no.hials.crosscom.KRL.KRLReal;
import no.hials.crosscom.KRL.structs.KRLPos;
import no.hials.crosscom.KRL.KRLVariable;
import no.hials.crosscom.KRL.structs.KRLE6Pos;

public class RobotConnection {
    private CrossComClient connection;
    private String ipAddress;
    private int port;
    public RobotConnection(){
    }
    public RobotConnection(String ipAddress, int port){
        this.port = port;
        this.ipAddress = ipAddress;
        connect();
    }

    //Reads bool values
    public boolean readBoolean(KRLBool bool){
        try{
            this.connection.readVariable(bool);
        }
        catch(Exception e){
            System.out.println("Error writing bool to
  Robot");
        }
        return bool.getValue();
    }
    //Writes bool values
```

```
    public void writeBoolean(KRLBool bool){
        try{
            this.connection.writeVariable(bool);
        }
        catch(Exception e){
            System.out.println("Error writing bool to
   Robot");
        }
    }

// reads position x,y,z and orientation A,B,C from robot
    public double[] readFrame(String frameName){
        KRLFrame frame = new KRLFrame(frameName);
        try{
            this.connection.readVariable(frame);
        }
        catch(Exception e){
            System.out.println("Error reading frame from
   Robot");
        }
        //Return the pose
        double
   []CurrentValue={frame.getX(),frame.getY(),frame.getZ(),frame.getA(),f
        return CurrentValue;
    }

    //Read the transformation matrix created in the c++
   application.
    public double[]  readTransformation(String textfile){
        try{
```

```
        Path filePath = Paths.get(textfile);
        Scanner scanner = new Scanner(filePath);
        List<Double> integers = new ArrayList<>();
        while (scanner.hasNext()) {

            if (scanner.hasNextDouble()) {
                integers.add(scanner.nextDouble());}
            else {
                scanner.next();}
            }

        double[] T = new double[integers.size()];

        for (int i = 0; i < T.length; i++) {
                T[i] = integers.get(i);
                 }
        //returns an array of type double.
        return T;
        }

        catch(IOException ioe){
            System.out.println("Error reading the file");
            double[] K={0,0,0};
            return K;
            }
    }
    //Calculation of the RPY angles from the transformation
   matrix
    public double[]  RPYanglesCalculation(double
   []TransformationMatrix){
```

```
    //calculate the RPY angles:
        double A,B,C;

   A=Math.atan2(TransformationMatrix[4],TransformationMatrix[0]);

   B=Math.atan2(-TransformationMatrix[8],Math.sqrt(Math.pow(Transformati
                                                   +
   Math.pow(TransformationMatrix[9],2)));

   C=Math.atan2(TransformationMatrix[9],TransformationMatrix[10]);
        //The correction translation
        double[]
   CorrectionValues={TransformationMatrix[3],TransformationMatrix[7],

   TransformationMatrix[11],A,B,C};
        return CorrectionValues;
    }

    //Input: Current pose and correction pose for robot.
    //Writes the new pose to to robot.
    public void writeFrame(String frameName,double
   []CurrentValue,double []NewPose,String WhichRobot){
        double X,Y,Z,A,B,C;

        if (WhichRobot.equals("LEFT")) {
        X=CurrentValue[0]+NewPose[0];
        Y=CurrentValue[1]+NewPose[1];
        Z=CurrentValue[2]+NewPose[2];
        A=NewPose[3];
        B=NewPose[4];
```

```
        C=NewPose[5];

        KRLFrame newframe = new KRLFrame(frameName);
        newframe.setXToZ(X,Y,Z);
        newframe.setAToC(A,B,C);
        try{
            this.connection.writeVariable(newframe);
        }
        catch(Exception e){
            System.out.println("Error reading frame from the left Robot");
        }
    }

        if((WhichRobot.equals("RIGHT"))){

        X=CurrentValue[0]+NewPose[0];
        Y=CurrentValue[1]+NewPose[1];
        Z=CurrentValue[2]+NewPose[2];
        A=NewPose[3];
        B=NewPose[4];
        C=NewPose[5];

        //write to robot
        KRLFrame newframe = new KRLFrame(frameName);
        newframe.setXToZ(X,Y-5,Z); //take away 5mm to let the force control put them togehter
        newframe.setAToC(A,B,C);
        try{
            this.connection.writeVariable(newframe);
```

```
        }
        catch(Exception e){
            System.out.println("Error reading frame from the
   right Robot");
             }
        }
    }
```

Listing A.6: RobotConnection.java

A.4.3 RobotKR120, a subclass of RobotConnection, which declare methods used to control the the left robot

```

// Author: Simen HAgen Bredvold
// Master's Thesis NTNU IPK 2016

// Subclass of RobotConnection. This subclass declares the methods // called by the process control in GUI.java to control the left //robot. The methods calls the methods from RobotConnection to //function.

public class RobotKR120 extends RobotConnection {
    // The system variable files declared in $config.dat file.
     private KRLBool KR120UpdatePosition = new KRLBool("UpdatePosition");
     private KRLBool KR120startPickUp = new KRLBool("startPickUp");
     private KRLBool KR120startSetCamera= new KRLBool("startSetCamera");
     private KRLBool KR120startGetPosition = new KRLBool("startGETPOSITION");

     public RobotKR120(String ipAddress, int port){
         super(ipAddress, port);
         KR120UpdatePosition.setValue(false);
         KR120startPickUp.setValue(false);
         KR120startSetCamera.setValue(false);
         KR120startGetPosition.setValue(false);
```

```
    }

    //read the current pose from the robot
    public double[] readPose_LEFT(){
         double[]CurrentValue_LEFT=readFrame("POINT_IN_SPACE_LEFT");
         return CurrentValue_LEFT;
     }

    //Method which does the following:
    // 1)reads transformation matrix
    // 2)Calculates the correction angles
    // 3)Reads current robot pose from "POINT_IN_SPACE_LEFT" which is stored  in the robot
    // 4) Writes the new pose to "NEW_POINT_LEFT" stored  in the robot.

      public void writeEEposLEFT(){
        double[]Tmatrix_left=readTransformation("C:\\Users\\simen_000\\Desktop\\Net
        double[]NewValue_LEFT=RPYanglesCalculation(Tmatrix_left);
        double[]CurrentValue_LEFT=readFrame("POINT_IN_SPACE_LEFT");
        writeFrame("NEW_POINT_LEFT",CurrentValue_LEFT,NewValue_LEFT,"LEFT");
    }

 // Starts the robot program called SETPOSITION which reads "NEW_POINT_LEFT"
```

```
// and moves to this pose.
    public void startSETPOSITION(){
        KR120UpdatePosition.setValue(true);
        writeBoolean(KR120UpdatePosition);
    }

    public boolean SETPOSITIONRunning(){
        readBoolean(KR120UpdatePosition);
        return KR120UpdatePosition.getValue();
    }

    // The following is for starting the robot program for pick up of tube
     public void startPickUp120(){
        KR120startPickUp.setValue(true);
        writeBoolean(KR120startPickUp);
    }

    public boolean PickUpRunning(){
        readBoolean(KR120startPickUp);
        return KR120startPickUp.getValue();
    }

    public void startSetCamera120(){
        KR120startSetCamera.setValue(true);
        writeBoolean(KR120startSetCamera);
    }

    public boolean SetCameraRunning(){
        readBoolean(KR120startSetCamera);
```

```
        return KR120startSetCamera.getValue();
    }
}
```

Listing A.7: RobotKR120.java

A.4.4 RobotKR240, a subclass of RobotConnection, which declare methods used to control the the right robot

```java


// Author: Simen HAgen Bredvold
// Master's Thesis NTNU IPK 2016

// Subclass of RobotConnection. This subclass declares the
   methods // called by the process control in GUI.java to
   control the right  //robot. The methods calls the methods
   from RobotConnection to //function.

public class RobotKR240 extends RobotConnection {
    private KRLBool KR240startPickUp = new
   KRLBool("startPickUp");
    private KRLBool KR240startSetCamera= new
   KRLBool("startSetCamera");
    private KRLBool KR240UpdatePositionRight = new
   KRLBool("UpdatePositionRight");

    public RobotKR240(String ipAddress, int port){
        super(ipAddress, port);

        // Set all KRLBool FALSE
        KR240UpdatePositionRight.setValue(false);
        KR240startPickUp.setValue(false);
        KR240startSetCamera.setValue(false);
    }
    //read the current pose from the robot
```

```
    public double[] readPose_RIGHT(){
        double
[]CurrentValue_RIGHT=readFrame("POINT_IN_SPACE_RIGHT");
        return CurrentValue_RIGHT;
    }

  //Function which does the following:
  // 1)reads transformation matrix
  // 2)Calculates the correction angles
  // 3)Reads current robot pose from
"POINT_IN_SPACE_RIGHT"  which is stored  in the robot
  // 4) Writes the new pose to "NEW_POINT_RIGHT" stored
in the robot.

   public void writeEEposRIGHT(){
       double
[]Tmatrix_right=readTransformation("C:\\Users\\simen_000\\Desktop\\Ne
       double
[]NewValue_RIGHT=RPYanglesCalculation(Tmatrix_right);
       double
[]CurrentValue_RIGHT=readFrame("POINT_IN_SPACE_RIGHT");

writeFrame("NEW_POINT_RIGHT",CurrentValue_RIGHT,NewValue_RIGHT,"RIGHT
  }

  public void startSETPOSITIONright(){
       KR240UpdatePositionRight.setValue(true);
       writeBoolean(KR240UpdatePositionRight);
  }

```

```
    public boolean SETPOSITIONRunningRIGHT(){
        readBoolean(KR240UpdatePositionRight);
        return KR240UpdatePositionRight.getValue();
    }

    public void startPickUp240(){
        KR240startPickUp.setValue(true);
        writeBoolean(KR240startPickUp);
    }

    public boolean PickUpRunning(){
        readBoolean(KR240startPickUp);
        return KR240startPickUp.getValue();
    }

    public void startSetCamera240(){
        KR240startSetCamera.setValue(true);
        writeBoolean(KR240startSetCamera);
    }

    public boolean SetCameraRunning(){
        readBoolean(KR240startSetCamera);
        return KR240startSetCamera.getValue();
    }
}
```

Listing A.8: RobotKR240.java

Appendix B

Digital Appendix

A .zip file is included as digital appendix. This contains:

- A video "Masters Simen Hagen Bredvold Welding with pose correction.avi" showing the alignment using SAC-IA and ICP, RANSAC with Search Method and only Search Method.
- Source code for the C++ application. Needs PCL and its 3third party libraries to be build and run.
- KUKA Robot files for handling of the tubes.
- $Config.dat file for right and left robot.
- Source code for the "ControlSystem" project developed in Java for communication with the robots.
- Matlab files to run the Safety application.