



# Robotsveising med korreksjon fra 3D-kamera

**Eirik Bjørndal Njåstad**

Master i produktutvikling og produksjon

Innlevert: juni 2015

Hovedveileder: Olav Egeland, IPK

Norges teknisk-naturvitenskapelige universitet  
Institutt for produksjons- og kvalitetsteknikk



---

## MASTEROPPGAVE VÅREN 2015

**Eirik Njåstad**

**Tittel: Robotsveising med korreksjon fra 3D-kamera**

**Tittel (engelsk): Robotic welding with correction from 3D camera**

**Oppgavens tekst:**

Sveising er et viktig anvendelsesområde for industriroboter. Det er nyttig å bruk offline programmering av sveiseoperasjoner ved bruk av 3D grafikkssystemer. I denne oppgaven skal dette studeres, og robotsveising skal programmeres og simuleres for typiske sveiseoperasjoner.

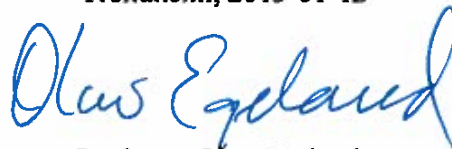
1. Gi en kort oversikt over metoder for programmering av robotsveising
2. Beskriv robotsyn ved bruk av 3D kamera.
3. Forklar hvordan et 3D-kamera kan brukes til korrigering av programmerte baner for sveiseroboter.
4. Programmer og simuler en sveiseoperasjon som er relevant for thrusterproduksjon i KUKA.Sim.
5. Implementer sveiseoperasjonen med korrigering fra 3D-kamera i robotlaboratoriet ved instituttet.

**Oppgave utlevert: 2015-01-12**

**Innlevering: 2015-06-10**

Utført ved Institutt for produksjons- og kvalitetsteknikk

Trondheim, 2015-01-12



Professor Olav Egeland  
Faglærer





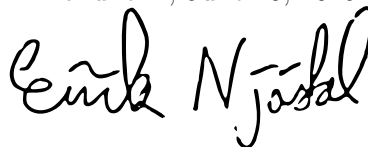
## Preface

This is the concluding Master's thesis of the study programme in Mechanical Engineering at NTNU. The work was carried out between January and June 2015.

As an apprentice for craft certificate in a manufacturing company, I was, among other things, assigned the task of programming a welding robot. This was the origin of my interest in the robotics field, and I remember spending much time back then pondering over how such robotic systems could be further improved.

Several years of studies later, it was fulfilling to get the opportunity to complete my degree by a contribution that might be used for improving robotic systems. A computer vision course and many good discussions resulted in the topic "Robotic welding with correction from a 3D camera" for this final Master's thesis.

Trondheim, June 10, 2015

A handwritten signature in black ink, reading "Eirik Njåstad". The signature is written in a cursive, flowing style.

Eirik Bjørndal Njåstad



## Acknowledgment

I would like to express my gratitude to my supervisor Professor Olav Egeland for the useful comments, remarks and engagement through the learning process of this master thesis. I am also very grateful for the opportunity he has given me to work on such a topic and be able to implement the system on actual robots. Furthermore, I would like to thank PhD candidate Lars Tingelstad for his cooperation and help along the way. His involvement in the work has been very helpful in selecting the right tools and software for the experiment. He also did a lot of work on the setup of the interface between the welding machine and robot controller.

The workshop employees at the lab have done a very good job making the parts that have been welded together. Thank you for this important contribution.

I would like to thank my friends and fellow students for a very pleasant and motivating working environment. And finally I would like to thank my loved ones, who have supported me throughout the entire process.

E.B.N.



## Summary

Robotic welding systems are widely used in manufacturing of many kinds. This thesis looks into the possibility of integrating a 3D camera into a robotic welding system, with the intention of improving the programming process for welding robots. The approach is to utilize data from the 3D camera for correcting offline programmed welding paths. The results were implemented in a robotic welding system demonstrator at the robot laboratory at the *Department of Production and Quality Engineering (IPK), NTNU*.

Computer vision for 3D data is a large part of the thesis. The 3D data acquisition, processing, and the different camera parameters are described. Various algorithms used for estimating the pose of the component to be welded are explained, and their performance is evaluated through a series of tests. The robustness and quality of the achieved results were of particular interest, and the time needed for estimating the component pose has also been evaluated for the different algorithms. Based on test results, the *Nonlinear Iterative Closest Point* algorithm was considered most suitable for this application.

The 3D camera was interfaced to the robot controller through a developed software application which estimates the correct object pose from a model of the object and 3D data from the camera. The corrected pose is communicated to the robot controller via a client-to-server connection over a computer network. A graphical representation of the information flow in the system is made, showing all steps from a CAD model and 3D data to the robotic welding process.

The results were demonstrated by programming the robot to perform welding on a section of a thruster tunnel for ships, and then estimate corrections for the welding paths based on data from the Kinect™ sensor from Microsoft, a consumer grade 3D camera. The demonstration illustrates how the developed system is able to detect and correct the welding paths for both translated and rotated objects. There are however some variations in the output from the object pose estimation. This causes inaccurate correction of the welding paths. By collecting 3D data for a longer time period and filtering the data, it was possible to reduce these variations but not eliminate them.

The resulting welding paths are presented as plots where the offline programmed, the camera

corrected, and optimal welding paths are shown.

## Sammendrag

Robotisert sveising benyttes for tilvirkning av mange slags produkter. I denne masteroppgaven har muligheten for å integrere et 3D kamera i et sveiserobotsystem blitt vurdert, med sikte på å forbedre programmeringsprosessen for sveiseroboter. Data fra et 3D kamera ble her anvendt for korreksjon av offline programmerte baner for sveiseroboter. Dette ble demonstrert ved å implementere metoden i robotlaboratoriet ved Institutt for produksjons- og kvalitetsteknikk, NTNU.

Beskrivelser av datasyn ved bruk av 3D data utgjør en stor del av innholdet i denne oppgaven. Ervervelse av 3D data, prosessering, og de forskjellige kameraparametrene er noe av det som er beskrevet. Et utvalg algoritmer som benyttes for å estimere posituren til komponenten som skal sveises forklares, og ytelsen til disse algoritmene er evaluert gjennom en rekke tester. Det var av interesse å undersøke hvor robust algoritmene er i ulike situasjoner, og å undersøke kvaliteten på estimatet fra de ulike algoritmene. Et annet vurderingskriterie var hvor lang tid som behøvdtes for å estimere komponentens positur. Testresultatene viste at algoritmen *Nonlinear Iterative Closest Point* var best egnet for bruk i dette systemet.

3D kameraet ble knyttet til sveiserobotens styreenhet gjennom et program som ble spesielt utviklet for dette prosjektet. Programmet beregner komponentens positur ved hjelp av en modell av komponenten og 3D data fra kameraet. Denne korrigerte posituren formidles til robotens styreenhet via en klient til tjener-tilkobling over et standard datanettverk. Et diagram som viser all informasjonsflyt i systemet har blitt utarbeidet. Dette diagrammet viser alle steg i prosessen, fra CAD modell og 3D data til en korrigeret sveiseprosess.

Resultatene ble demonstrert ved å programmere sveiseroboten til å sveise en seksjon av en thrustertunnel for skip. Data fra et Microsoft Kinect™ kamera benyttes så til å estimere korreksjoner av sveisebanene. Demonstrasjonen illustrerer hvordan systemet som er utviklet er i stand til å detektere og korrigere sveisebanene for objekter som er flyttet på både ved rettlinjert bevegelse og ved rotasjon. Det oppsto imidlertid noen variasjoner ved estimering av objektets positur, som igjen førte til unøyaktige korreksjoner av sveisebanene. Det var mulig å redusere disse variasjonene ved å samle 3D data i et litt lenger tidsrom og å implementere et filter.

Robotens sveisebaner er illustrert som grafer hvor offline programmerte, kamerakorrigerede, og optimale sveisebaner er vist.



## Glossary and Acronyms

**CAD** Computer-aided design is the use of computer systems to assist in the creation, modification, analysis, documentation, or optimization of a design.

**CMOS/CCD** Charge-Coupled Devices (CCD) or Complementary Metal–Oxide–Semiconductors (CMOS) are sensors that detect and convey the information that constitutes an image, i.e., an image sensor.

**CW-ToF** Continuous-Wave Time-of-Flight cameras are 3D imaging sensors which consist of a pixel array together with an active modulated light source.

**GMAW** Gas metal arc welding is a welding process in which an electric arc forms between a consumable wire electrode and the workpiece metal, causing them to melt and join.

**GUI** A graphical user interface or GUI is a type of interface that allows users to interact with electronic devices through graphical icons and visual elements, as opposed to text-based interfaces.

**ICP** Iterative Closest Point is an algorithm employed to minimize the difference between two clouds of points, often used for aligning point clouds.

**KR C5 Controller** The standard KUKA robot controller used for control of the KR5 robot.

**MAG** Metal Active Gas welding is a subtype of GMAW welding. A large area of application is for manufacturing of steel structures of thin and medium thick plates.

**MLS** Moving Least Squares is a method of reconstructing continuous functions from a set of unorganized point samples by calculating a weighted least squares measure.

**PCL** The Point Cloud Library is an open-source library of algorithms for point cloud processing tasks and 3D geometry processing.

**Pixel** In digital imaging, a pixel (picture element) is a physical point in an image.

**PLC** A Programmable Logic Controller is a digital computer typically used for automation of industrial processes.

**Point Cloud** A point cloud is a set of points in a coordinate system. In a three-dimensional coordinate system, these points are usually defined by X, Y, and Z coordinates, and often represents the surface of an object.

**RANSAC** RANdom SAmples Consensus is an iterative method for estimating a mathematical model from a data set that contains outliers.

**RGB** A three number color model, in which red, green, and blue light are added together in various ways to reproduce a broad array of colors.

**RSI** Robot Sensor Interface used for external sensor input by KUKA robots.

**SAC-IA** SAmples Consensus Initial Alignment is a method for aligning point clouds.

**SDK** A Software Development Kit is a set of software development tools that allows for creating applications for a certain system or platform.

**SVD** Singular Value Decomposition is a factorization of a real or complex matrix in linear algebra.

**TCP** The Tool Center Point is a point in relation to which robot positioning could be defined.

**ToF** Time of Flight describes a methods that measure the time that it takes for an object or wave to travel a distance through a medium.

**Voxel** A voxel represents a value on a regular grid in three-dimensional space. Voxel is a portmanteau for "volume" and "pixel".

**VRC** Virtual Robot Controller. A simulated version of the KR C4 robot controller used for programming KUKA robots.

# Contents

Acknowledgment . . . . .	iii
Summary . . . . .	v
Sammendrag . . . . .	vii
Glossary and Acronyms . . . . .	ix
List of Figures . . . . .	xv
<b>1 Introduction</b>	<b>1</b>
1.1 Background . . . . .	1
1.2 Objectives . . . . .	2
1.3 Approach . . . . .	2
1.4 Structure of the Report . . . . .	5
<b>2 Robot Programming</b>	<b>7</b>
2.1 Introduction . . . . .	7
2.2 Offline Robot Programming . . . . .	8
<b>3 Computer Vision</b>	<b>15</b>
3.1 Introduction . . . . .	15
3.2 Point Cloud Acquisition . . . . .	16

3.2.1	Depth images and Point Clouds	16
3.2.2	Time-of-Flight Cameras	17
3.2.3	Camera Calibration	22
3.3	Point Cloud Processing	26
3.3.1	Down-Sampling	27
3.3.2	Statistical Outlier Removal	27
3.3.3	Moving Least Squares Smoothing	28
3.3.4	Estimating Surface Normals	29
3.4	Object Alignment	30
3.4.1	Iterative Closest Point	31
3.4.2	Sample Consensus Initial Alignment	34
3.4.3	Algorithm Performance	35
3.4.4	Representing the Estimation Results	38
<b>4</b>	<b>Practical Setup</b>	<b>41</b>
4.1	Robot Cell	41
4.2	Information Flow	43
4.2.1	Offline Programming of Welding Sequence	44
4.2.2	Robot Program	46
4.2.3	Point Cloud Capturing	46
4.2.4	Representing the CAD model	47
4.2.5	Object Alignment and Calculations	48
4.2.6	Visualizations	48
4.2.7	TCP/IP Communication	48
4.2.8	Fronius TransSteel to Robot Interface	51

<b>5 Results</b>	<b>53</b>
5.1 Simple Welding Joints . . . . .	53
5.1.1 Case I: No Object Alignment . . . . .	54
5.1.2 Case II: 2D Object Alignment . . . . .	55
5.1.3 Case III: 3D Object Alignment . . . . .	56
5.1.4 Case IV: 3D Object Alignment from Several Point Clouds . . . . .	56
5.2 System Performance . . . . .	57
5.3 Thruster Tunnel Welding . . . . .	60
5.4 Video . . . . .	62
<b>6 Concluding Remarks</b>	<b>63</b>
6.1 Discussion . . . . .	63
6.2 Conclusion . . . . .	65
6.3 Recommendations for Further Work . . . . .	66
<b>Bibliography</b>	<b>68</b>
<b>A Source Code</b>	<b>75</b>
A.1 Main C++ application . . . . .	76
A.2 Communication Client . . . . .	91
<b>B Digital Appendix</b>	<b>95</b>



# List of Figures

2.1	The Offline Programming Concept. . . . .	9
2.2	Offline Programming Process Flow. . . . .	11
3.1	A sample Point Cloud. . . . .	17
3.2	ToF operating principle. . . . .	18
3.3	Multi-frequency technique. . . . .	20
3.4	Microsoft Kinect for Xbox One. . . . .	21
3.5	Amplitude image of checkerboard. . . . .	23
3.6	3D position of the ToF camera. . . . .	23
3.7	Coordinate frame relations. . . . .	25
3.8	Voxel grid filtering. . . . .	28
3.9	Moving least squares smoothing. . . . .	29
3.10	Iterative Closest Point alignment of the Stanford 3D bunny. . . . .	32
3.11	Robustness and quality of alignments. . . . .	37
3.12	Running time for the various algorithms. . . . .	39
4.1	The robot cell at IPK, NTNU . . . . .	42
4.2	Robot and welding machine setup. . . . .	43

4.3	System information flow . . . . .	45
4.4	CAD model and point cloud representation. . . . .	47
4.5	Visualization of the Kinect data stream. . . . .	49
4.6	Effects of ICP. . . . .	49
5.1	Welding joints used for testing the system. . . . .	54
5.2	Welding paths without corrections. . . . .	55
5.3	Welding paths with 2D corrections. . . . .	56
5.4	Welding paths with 3D corrections. . . . .	57
5.5	Welding paths with 3D corrections from several point clouds. . . . .	58
5.6	Absolute error of pose estimations. . . . .	59
5.7	Object alignment of rotated component. . . . .	59
5.8	A model of the thruster tunnel. . . . .	61
5.9	Thruster tunnel section after welding. . . . .	62
6.1	Sketch of a Kinect 3D camera mounted on a KR 5 robot. . . . .	66



# Chapter 1

## Introduction

### 1.1 Background

Robotic welding is a major field of application for industrial robots. Using robots in welding processes helps to give industries a competitive advantage by increasing their productivity, quality and flexibility. It also reduces the discomfort and significant health hazards associated with manual welding. Most types of welding requires motion control, sensor integration, and coordination with an external welding power source. The industrial robot manipulators are thus an almost perfect match for the vast majority of welding processes. Currently, spot resistance welding is the most common robotic welding method, followed by arc welding which is spreading faster than the former [1].

The automotive industry has traditionally been a driving force and the largest consumer of welding robots. Besides further development in this segment, enterprises with significantly lower production volumes will within a few years benefit from robots as much as the the automotive industry does today [2]. For enterprises with small batch production such automation is largely a matter of investment costs and programming ability. With continuously changing production patterns, it is likely that the cost of reprogramming will exceed the investment costs by a large margin [3]. It is thus a great potential for improving the robot programming process, enabling cost-effective production of high quality welds at shorter cycle times.

Robotic welding assures high repeatability of the trajectories, as robots performs well in constant processes. Due to tolerances in the work piece geometries and other offsets, the real welding situation will vary from time to time. In contrast to manual welders, the robot is not able to modify the welding path continuously by itself. To make robotic welding systems more flexible in dealing with varying work pieces, external inputs such as computer vision are introduced.

Computer vision can be used for detecting and measuring the physical location and orientation of the work piece to be welded. This could improve the efficiency of welding processes by offline programmed robots, and the required level of competence for using a robotic welding system could potentially be lowered. This will allow many more enterprises to benefit from robotic welding systems.

## 1.2 Objectives

The main objectives of this Master's project are

1. To provide a brief overview of methods for programming of welding robots.
2. To describe robotic vision by means of 3D cameras.
3. Explaining how a 3D camera can be used for correcting the programmed paths of welding robots.
4. To program and simulate a welding operation applicable in thruster production by using the *KUKA.Sim* software application.
5. Implementation of the welding operation with correction from 3D camera in the robot laboratory at the *Department of Production and Quality Engineering (IPK), NTNU*.

## 1.3 Approach

The objectives in this project has been of both theoretical and practical nature. Literature and articles have been studied in order to answer the theoretical aspects, but also in preparation for

and as a support when solving the practical challenges at hand.

The undersigned participated in the basic robot course at KUKA college in Göteborg. This was an important introduction to the KUKA control system and manual robot programming via the KR C4 controller.

### **Objective 1**

The overview of methods for programming welding robots is given in Chapter 2. Offline Programming of robots has been subject to the most detailed explanation as this method was applied for the developed system. A brief state of the art analysis is performed, where the typical workflow of today's offline programming systems is explained. A list of the available offline robot programming software is also provided.

### **Objective 2**

Robotic vision by means of 3D cameras is described in Chapter 3. In this project, a Microsoft Kinect™ which contains a *time-of-flight* camera was used for sensor input. Describing the operational principle of this type of camera has therefore been given priority. The structure of 3D images is also explained, along with descriptions of common processing steps required for using 3D data in robotic vision.

### **Objective 3**

This project has culminated in a practical demonstrator where a 3D camera is used for correcting the programmed paths of a welding robot. The practical setup described in Chapter 4 is an example of how such a system can be configured. Various approaches for correcting robot paths from 3D data have been explained in Chapter 3.

Development and implementation of the program that calculates the necessary corrections based on data from the 3D camera has been the largest and most demanding part of this project.

C++ is the programming language used in the development, with the additional Point Cloud Library used for implementation of most algorithms used. Microsoft Visual Studio 2013 has been used for developing the program.

Two object aligning algorithms, three versions of one of them, have been tested and the performance evaluated for use in correction of welding robot paths.

#### **Objective 4**

A graphical model has been created in *KUKA.Sim*. Mounting location and orientation of the welding robot manipulator and equipment are the same as in the robot cell at *Department of Production and Quality Engineering (IPK), NTNU*. Welding operations applicable in thruster production have been programmed and simulated by using the created model.

Robotic movements in *KUKA.Sim* are programmed in almost the same way as on the KR C4 controller. This makes the tool great for testing of robot positions and trajectories before implementation on the robot. It is also a good way of testing different welding path correction scenarios, where the welding operations is corrected by using a simulated 3D camera.

#### **Objective 5**

The programmed welding operation was implemented in the demonstrator at the robot laboratory at IPK. In the demonstrator, a CAD model of a thruster tunnel section is used for aligning the offline programmed welding operation to the physical object orientation and location. The location is determined by data from a Kinect 3D camera. The corrected pose is communicated to the robot controller via a *client-to-server* connection over Ethernet. System performance has been evaluated in Chapter 5.

#### **Literature**

Textbooks on computer vision, robotics, welding robots, and programming topics have been used to a large extent during the project. The main books are:

- *Introductory Techniques for 3-D Computer Vision* by Emanuele Trucco and Alessandro Verri [4].
- *Computer Vision: A Modern Approach (2nd Edition)* by David A. Forsyth and Jean Ponce [5].
- *Robotics, Vision and Control* by Peter Corke [6].
- *Welding Robots: Technology, System Issues and Application* by J. Norberto Pires, Altino Loureiro, and Gunnar Bölmsjö [7].
- *Absolute C++* by Walter Savitch [8].

In addition, the KUKA manual "*Operating and Programming Instructions for System Integrators*" and other documentation for the KUKA KR C4 controller, Fronius welding machine, and Ether-Cat has been used. Relevant scientific articles was also studied, especially on the *time-of-flight* cameras and object alignment subjects.

## 1.4 Structure of the Report

The report is focused on the development of this exact demonstrator, with a broader focus where this is found relevant. The experiment setup, depth data acquisition, object alignment, transformation estimation, and information flow is explained for this exact experiment. Most features will also function in other systems without further customization, but the total arrangement is customized for this exact equipment.

- In Chapter 2, a brief overview of robot programming is provided, with emphasis on offline robot programming. A survey of available offline programming software is also included.
- Chapter 3 describes the computer vision and depth data, with camera calibration, object alignment and transformation calculations.

- Chapter 4 presents the practical setup of the demonstrator, and explains the information flow and necessary elements in a system for correcting the programmed paths of welding robots.
- In Chapter 5, the results from the experiments are presented, along with some suggestions for improving the system performance.
- Chapter 6, summarize the work with a discussion, conclusion and recommendations for future work and implementations.
- Source code of the program calculating necessary corrections and of the communication client can be studied in the appendix.
- The digital appendix includes all source code written for this project. A video showing the system used for welding a thruster tunnel section is also found here.

# Chapter 2

## Robot Programming

### 2.1 Introduction

Robot programming is the process of determining a robot's interaction with its environment, usually in terms of generating a set of robot instructions and poses that will accomplish the desired task. Robot programming is often characterized as tedious and time-consuming, with a lack of intuitive tools for interaction. Many small and medium-sized enterprises (SMEs) are not using robots in their facilities because the configuration and programming process of this type of equipment is time-consuming and requires workers with a high level of expertise in the field [9]. Thus, there is a great potential for improvements in programming that enable cost-effective production of high quality welds at shorter cycle times.

The American Occupational Safety and Health Administration (OSHA) defines three ways of programming industrial robots: Lead-through programming, Walk-through programming, and Offline programming [10].

1. **Lead-Through Programming** or Teaching is a method involving the use of a proprietary teach pendant to physically guide the robot through the desired sequence of events. The programming is performed by trained personnel working within the robot's working envelope.

2. **Walk-Through Programming** or Teaching is similar to the lead-through method, but is characterized by the programmer interacting physically with the robot, guiding it through the desired positions within the working envelope. While programming, the robot controller is recording the coordinate values on a fixed time basis.
3. **Offline Programming** is the process of creating a robot program without using a real robot. CAD systems are used to model the particular robot, workpiece, tool, and workspace. The models are then used to perform the robot tasks and path planning, and to automatically generate programs that are downloaded to and executed by the robot controller. Before the program can be executed, it is usually necessary to perform verification and small changes to the program. This part of the process is called program *touch-up* and is typically performed as Lead-Through or Walk-Through programming.

If the Lead-Through or Walk-Through programming method is used, it is clear that the robot can not be used for production while programming. These programming methods can be classified as online programming. Offline programming is the only method where the robot can work uninterrupted while it is being remotely programmed for new tasks. This facilitates using robots for single piece manufacturing, in contrast to batch production which is a common application for robots today. It is thus of interest to improve the method further.

## 2.2 Offline Robot Programming

Offline programming shifts the tedious work of programming the robot from the operator in the workshop to the engineer in the office. The concept of offline programming is shown in Figure 2.1. By this method, it is beneficial to develop and configure a virtual model of the robot cell, providing the ability to simulate the process while programming. The various software solutions for offline robot programming largely follows the same key steps of creating the model and programming the robot [11]. These steps are summarized in Figure 2.2.

It is usually required to have 3D CAD models of the workpiece, all fixtures in the cell, and of the particular robot manipulator to be used. Products usually have a CAD model, it can be more



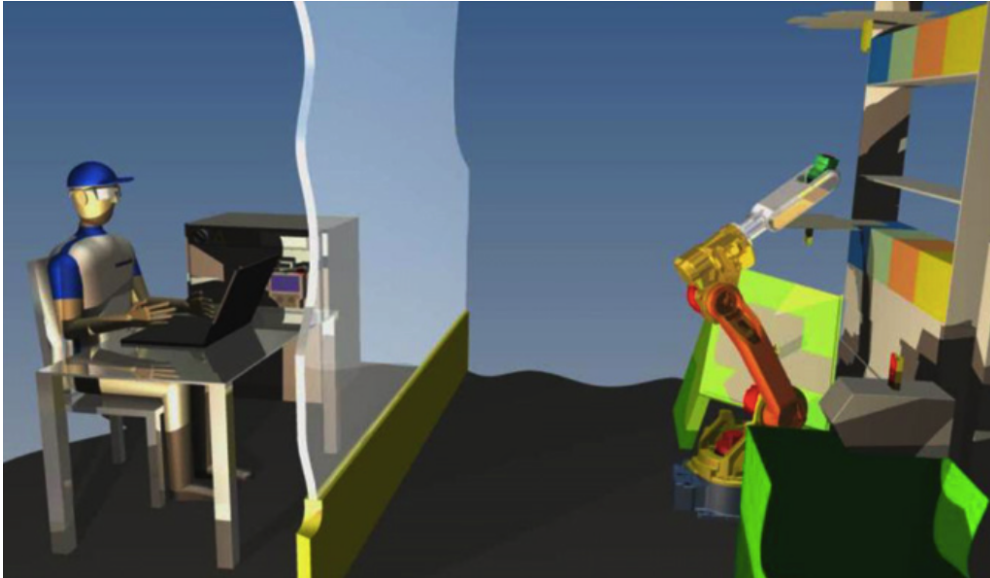


Figure 2.1: The Offline Programming Concept [9].

challenging to model fixtures in the cell. Models of robots are for the most part included in the software packages, or can be purchased as extensions. The wide range of different CAD formats being used leads to a frequent need for converting to a compatible format. Conversion between different types of CAD files is usually a straightforward task [12].

When an adequate model of the workspace and components are built, the programming can commence. Clicking and dragging the computer mouse replaces jogging with the teach pendant in the physical welding cell. Specification of robot positions are generally either done manually by point and click, or automatically by the software, which generates them when the desired welding paths are highlighted. Assisting robot positions such as home position, approach points, and retreat points can also be specified.

The inverse kinematics of industrial articulated robots have multiple solutions in Cartesian space. One of these solutions, the robot configuration, is usually selected manually because most of the offline robot programming tools can not automatically select an optimal solution [11].

Programming of welding robots also means selection of welding parameters such as current, voltage, and travel speed. The post-processing stage includes adding these and other necessary I/O control signals to the program, as well as necessary fine adjustments of the robot paths. If

using generic software, the program must be converted into the language of the specific robot type.

The robot motions can be simulated and tested while programming and when finished. The programmed instructions is used for emulating robot motions and to determine whether each movement can be successfully executed. Possible collisions, areas associated with problems for achieving the correct welding angle, and cycle times can be detected by the software. The leading software solutions also checks whether there are any violations of the robot joint limits, in terms of displacement or velocity.

After programming and offline testing is completed, the program is exported to the physical robot controller. This can be done by using a memory stick or by transmitting over Ethernet. It is usually necessary to do a manual touch-up of the points created by offline programming before the program can be executed in production mode. By current methods, the robot programs is about 75 percent completed before manual touch-up [7]. This is just one way to calibrate the program towards the real the robot cell. Other methods includes using a set of calibration points within the cell, or compensating for the discrepancies through the use of sensors on the real robot.

## **Available Software**

Simulation and programming software is a fundamental tool for robot and system designers, manufacturing companies, students of the field, and other users. This has motivated the development of a variety of software. The range of software environments originate from various academic research, robot manufacturers, and other developers.

Table 2.1 gives an overview of the available software for offline programming of robots. The table shows that nearly all major robot manufacturers provides extensions of their offline programming software aimed at arc welding.

Most robot manufacturers provide proprietary software for offline programming that includes accurate CAD models of their own robots. There are however software vendors offering solutions that allow simultaneous programming and provides accurate CAD models of robots from

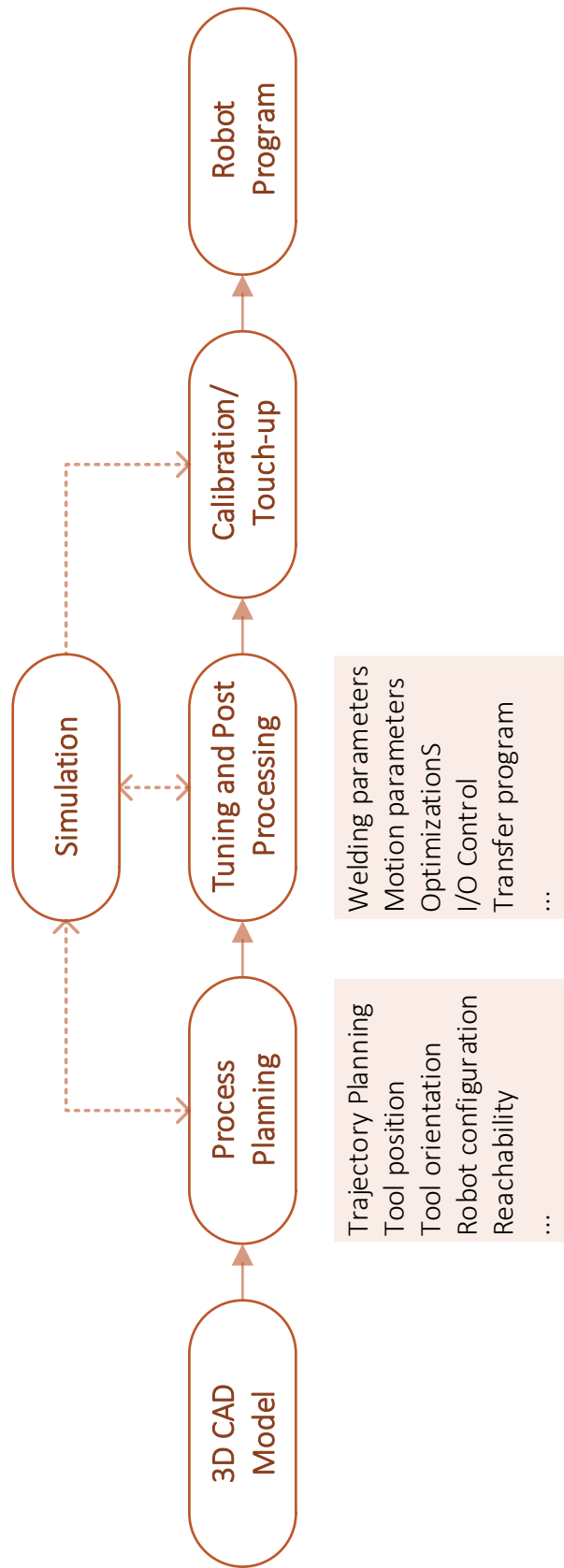


Figure 2.2: The various steps in the offline programming process of a welding robot.

numerous manufacturers. The manufacturer-specific software offers perhaps the finest control over the programming, and often have more adapted functionality.

The majority of the programming software features cycle time calculations, import of 3D CAD files, 3D simulation with collision detection, and a component and robot library of a varying extent. Among the robot manufacturer-specific software, it is popular to provide more accurate simulations by connecting to a Virtual Robot Controller (VRC). This controller usually contains a virtual version of the teach pedant associated with the particular robot make.

<b>Software</b>	<b>Provider</b>	<b>Utilities for Arc Welding</b>
<i>OLP software by robot manufacturers</i>		
AX On Desk	Nachi	Yes
DTPS-G3	Panasonic	Yes
K-ROSET, KCONG	Kawasaki	Yes
KUKA-Sim, CAMrob	KUKA	Yes
MELFA WORKS	Mitsubishi Electric	No
MotoSim EG VRC	Motoman (Yaskawa)	Yes
Roboguide (WeldPRO)	Fanuc	Yes
RobotStudio	ABB	Yes
Stäubli Robotics Suite	Stäubli	No
Wincaps III	Denso	No
<i>Generic OLP software</i>		
CimStation Robotics	Applied Computing & Engineering	Yes
Delfoi ARC, AutoARC	Delfoi	Yes
DELMIA Robotics Arc Welding (ARW)	Dassault Systèmes	Yes
EASY-ROB	EASY-ROB	No
FAMOS robotic	Carat Robotic Innovation	No
Robcad	Technomatix, Siemens PLM	Yes
Robotmaster	Jabez Technologies Inc.	No
RobotWorks, Robotics Interface for Solidworks	Compucraft Ltd	Yes
Workspace 5	WAT solutions	No

Table 2.1: Available Offline Programming Software (OLP). Most robot manufacturers provides their own programming solution. In addition, there are several other developers providing software for OLP.



# Chapter 3

## Computer Vision

### 3.1 Introduction

Humans are extremely good at identifying objects and estimating their pose in the scene. By using the visual input, useful decisions can be made about real physical objects and their surroundings from a distance. Vision guided robotics is a topic of continued interest. A lot of potential opens up in industrial processes by allowing for robots to see. Vision sensors are thus increasingly being utilized in industrial robotic systems.

There is a wide range of RGB cameras available at low cost. However, developing robust and efficient vision guided robotic systems can be a challenging task to accomplish when using 2D images that inherently only captures a projection of the 3D world. The recent arrival of Time-of-Flight (ToF) depth cameras and the even more recent introduction of the Microsoft Kinect™ depth camera (*Kinect* in the sequel) has made 3D data streams at video rate widely available. This enables new methods for helping the robot making useful decisions.

The focus in this project is on the use of depth data from a Kinect for correcting offline programmed welding paths. After acquisition and some processing, the depth data is input to an algorithm that estimates the 3D object pose, before the necessary corrections of the welding paths are calculated. The various steps are explained in the following sections.

The computer vision methods applied are mainly based on the features of Point Cloud Library [13] (PCL) and Peter Corke's *Robotics, Vision and Control* [6].

## 3.2 Point Cloud Acquisition

Point clouds can be acquired from various hardware sensors such as stereo RGB cameras, 3D scanners, or time-of-flight cameras, or be synthetically generated by a computer program. Section 3.2.1 briefly explains how depth data is represented. The technology behind the time-of-flight cameras and the Kinect is described in Section 3.2.2.

### 3.2.1 Depth images and Point Clouds

*Depth images are a special class of digital images where each pixel expresses the distance between a known reference frame and a visible point in the scene. Therefore, a depth image reproduces the 3D structure of a scene, and is best thought of as a sampled surface [4].* The direct encoding of surfaces makes depth images more useful for measuring distances than common intensity RGB and grayscale images. This is because in intensity images, the pixel values are only indirectly related to the surface geometry through optical and geometrical properties as well as lighting conditions. Depth images are also referred to as range images, depth maps,  $xyz$  maps, surface profiles, 2.5D images, and point clouds. An example of a depth image is shown in Figure 3.1.

There are two basic ways of representing depth images. The organized form is a matrix of depth values of points along the directions of the  $x, y$  image axes. The alternative is in its simplest form a list of 3D coordinates in a given reference frame, where no specific order is required. The unorganized representation is referred to as a point cloud. In this project the depth images have been represented by point clouds with the RGB color value of each 3D point included.

Depth images can be acquired from a variety of different sensors. A distinction is made between active and passive sensors. The active type is projecting energy (e.g. a pattern of light) on the scene in order to determine its 3D structure, while the passive type rely only on intensity images of the scene [4]. The time-of-flight camera used in this project is of the active type.



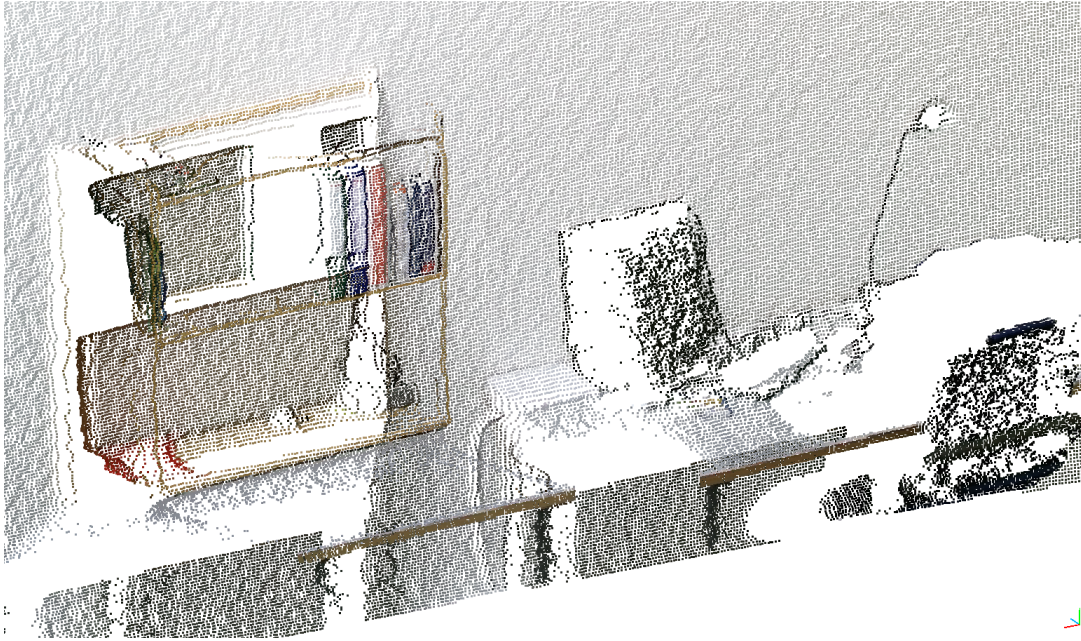


Figure 3.1: A point cloud representation of an office desk and a bookshelf. This unfiltered point cloud is acquired from the Kinect 3D camera with color information included for each of the 217.088 points.

### 3.2.2 Time-of-Flight Cameras

Optical time-of-flight (ToF) cameras measure the depth of a scene by illuminating the scene with a modulated light source, and observing the reflected light. The modulated light is typically emitted from a solid-state laser or a LED operating in the near-infrared range ( $\approx 850$  nm) [14], [15]. ToF methods can be classified in two subcategories: 1) Pulsed Modulation and 2) Continuous Wave Modulation. Most commercial ToF solutions are based on the latter method [16], and it is hence described in most detail here.

Pulsed Modulation ToF methods measure the distance to objects by measuring the absolute time a light pulse needs to travel from a source into the scene and back, after reflection. This can be achieved by integrating photoelectrons from the reflected light, or by the stopwatch technique, where a fast counter is used for measuring round-trip time of the single light pulse. The latter method is commonly implemented using single-photon avalanche diodes (SPADs) [17], and this technique gives a direct measurement of ToF with low influence of background illumination [18]. A drawback of the technique is its dependence on fast electronics, since achieving 1 millimeter accuracy requires timing a pulse of 6.6 picoseconds in duration [14]. Moreover, on-

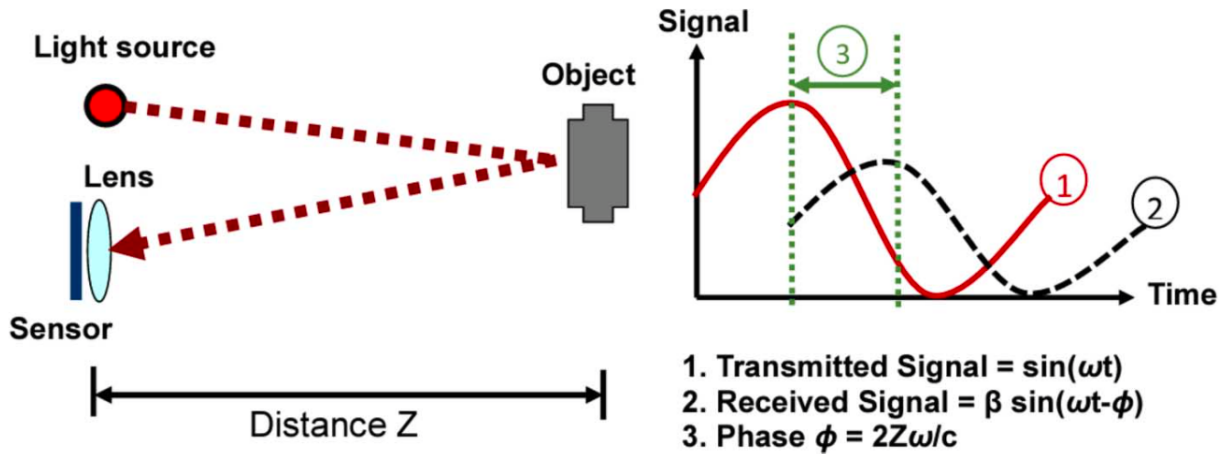


Figure 3.2: ToF operating principle [17].

chip converters required for this approach can use a significant pixel area, which can limit pixel array sizes [17].

In Continuous Wave Modulation ToF (CW-ToF) methods, the phase difference between the modulated emitted light wave and the detected wave reflected off the scene is measured. This is illustrated in Figure 3.2. The modulation is typically done by varying the frequency and amplitude of sinusoidal or square waves. Square wave modulation is more common because it can be easily realized using digital circuits [16]. The detected wave after reflection is phase shifted proportional to the distance of the reflecting object. Specialized CMOS/CCD sensors are used for integrating the photo-generated charges over a relatively long period of time, and the distance to the object is computed from the integrated signal. The depth resolution increases with the light source modulation frequency.

Calculation of the distance to a object when using CW-ToF methods is done by cross-correlating the received signal with the emitted signal. If the emitted signal is sinusoidal with modulation frequency  $\omega = 2\pi f$ , it can be described as

$$g(t) = \cos(\omega t)$$

and the received signal after reflection from object surface as

$$s(t) = b + a \cos(\omega t + \phi)$$

where  $b$  is a constant bias and  $a$  is the wave amplitude.  $\phi = 2z\omega/c$  is the phase shift with the distance  $z$  to the object, where  $c$  is the speed of light in air. Cross-correlation of both signals with the offset  $\tau$  leads to

$$c(\tau) = s * g = \int_{-\infty}^{\infty} s(t) \cdot g(t + \tau) dt$$

which can be simplified to

$$c(\tau) = \frac{a}{2} \cos(\omega\tau + \phi) + b$$

The CW-ToF methods takes multiple samples per measurement, with each sample phase-stepped by the offset  $\tau$  for a total of four samples

$$A_i = c\left(\frac{i \cdot \pi}{2}\right), i = 0, \dots, 3$$

The phase shift and amplitude can be directly obtained:

$$\phi = \arctan 2(A_3 - A_1, A_0 - A_2) \quad (3.1)$$

$$a = \frac{1}{2} \sqrt{(A_3 - A_1)^2 + (A_0 - A_2)^2} \quad (3.2)$$

Likewise the distance to the object:

$$d = \frac{c}{4\pi\omega} \phi \quad (3.3)$$

The distance measurement accuracy is influenced by the reflected pixel intensity (amplitude), reflected pixel offset, and how well the ToF sensor separates and collects the photoelectrons. The reflected intensity is a function of the optical power, while the reflected offset is a function of the ambient light and residual system offset. High amplitude, high modulation frequency and high modulation contrast will increase accuracy, while high offset can lead to saturation

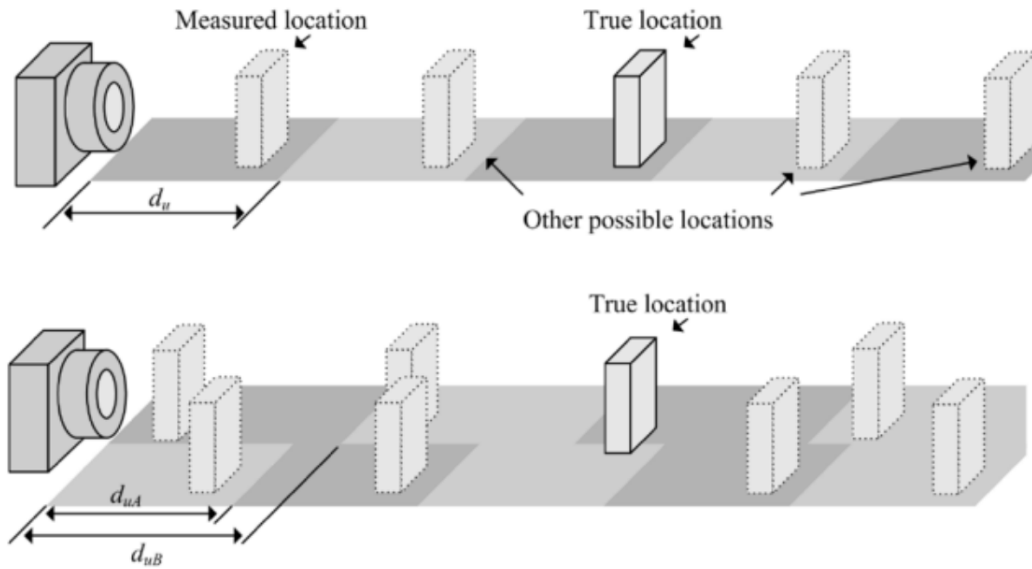


Figure 3.3: Multi-frequency technique [14].

and reduce the overall accuracy [14]. In practice, the CW-ToF methods requires integration over time to reduce noise. Long integration times reduces the possible frame rates, and can cause motion blur.

The fact that the CW measurement is based on phase, which wraps around every  $2\pi$ , causes each distance measurement to have an aliasing distance. Advanced ToF systems deploy multi-frequency techniques to avoid this ambiguity, where each modulation frequency will have a different aliasing distance, see Figure 3.3. The true object location is the one where the different frequencies coincide.

### Microsoft Kinect (tm)

Introduced in November 2013, the Kinect for Xbox One (Figure 3.4) is the second generation of sensor input devices developed for the Microsoft Xbox video game console systems. With the Kinect, state of the art depth sensing technology is available at a low cost. This version of Kinect consists of a 512 x 424 pixel wide-angle ToF camera, a high definition (1920 x 1080) RGB camera, and a four microphone array operating at 48 kHz.

The ToF camera is of the Continuous Wave Modulation type, with  $70^\circ$  horizontal and  $60^\circ$  vertical

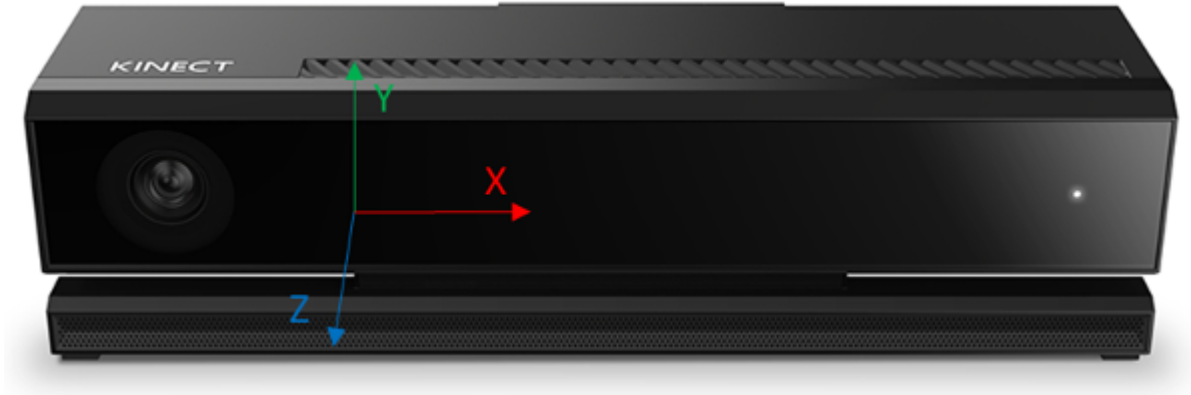


Figure 3.4: Microsoft Kinect(tm) for Xbox One Systems. The origin and axes of the camera space coordinate system are marked [19].

field of view. It has a very unique sensor where the pixel array is divided into a top and bottom half each driven by separate clock drivers. After several milliseconds of exposure, the two total amounts of photons recorded by the two halves are compared. This process of indirectly measuring the light wave time of flight allows for each pixel to independently measure distance, which results in a sensor that offers high degree of depth precision.

The modulation frequency and relative phase of each pixel clock, the light source modulation signal, as well as the gain for each pixel column are all programmable [17]. This functionality, combined with its high pixel resolution, allows for high-dynamic-range (>64 dB) operation at distances between 0.8 - 4.2 meters. Its depth uncertainty is below 0.5 % of the range, and the accuracy error is below 1 %. Higher accuracy can be achieved by taking the average of many frames at a near distance from the sensor. The Kinect has the highest pixel resolution with the smallest pixel pitch ( $10 \mu\text{m} \times 10 \mu\text{m}$ ) among all currently published ToF sensors (January 2015) [17].

A Microsoft Windows-compatible version of the Kinect, Kinect for Windows v2, was released in mid-2014 along with the Kinect for Windows software development kit (SDK) 2.0 [19]. It differs from the Kinect for Xbox One in that it includes an external power adapter and a USB 3.0 connection. This version is intended for developing Kinect-enabled software for Windows 8 and Windows 8.1, and is the version implemented in this work.

### 3.2.3 Camera Calibration

Since commercial ToF cameras use standard optics and their image formation can be modeled by perspective projection, the ToF camera calibration is strongly reminiscent of standard camera calibration. Instead of standard color (or gray-level) images, the amplitude image is used for ToF camera calibration. The geometrical camera calibration is of greatest interest here, where the relationship between sensor pixels and relative points in the scene are estimated. When using the pinhole camera model, this relationship is called a perspective projection and is described in terms of a set of physical parameters, subdivided in intrinsic and extrinsic camera parameters.

Calibration of the Kinect ToF camera in this project is performed by using the Camera Calibration Toolbox for MATLAB<sup>®</sup> [20], which is an implementation of Z. Zhang's [21] method from 1999. This method takes as input a series of pictures of a checkerboard in various positions and orientations. After the checkerboard image corners have been extracted by manual tagging, the toolbox calculates all camera parameters of interest by minimizing the reprojection error. Figure 3.5 shows the amplitude image of a checkerboard taken by the Kinect ToF camera with the corners detected. The detected location and orientation of the checkerboard in various poses are shown in Figure 3.6.

#### Intrinsic Parameters

The intrinsic camera parameters describe internal optical, geometric, and digital characteristics of the camera, such as the focal length of the lens and position of the camera principal point. Real cameras also have some radial and tangential distortions due to the lens curvature. The focal length  $(f_x, f_y)$  and camera principal point  $(c_x, c_y)$  forms the camera matrix. By calibration of the Kinect ToF camera, the camera matrix was found to be:

$$A = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 361.7 & 0 & 253.8 \\ 0 & 362.2 & 202.7 \\ 0 & 0 & 1 \end{bmatrix} \quad (3.4)$$

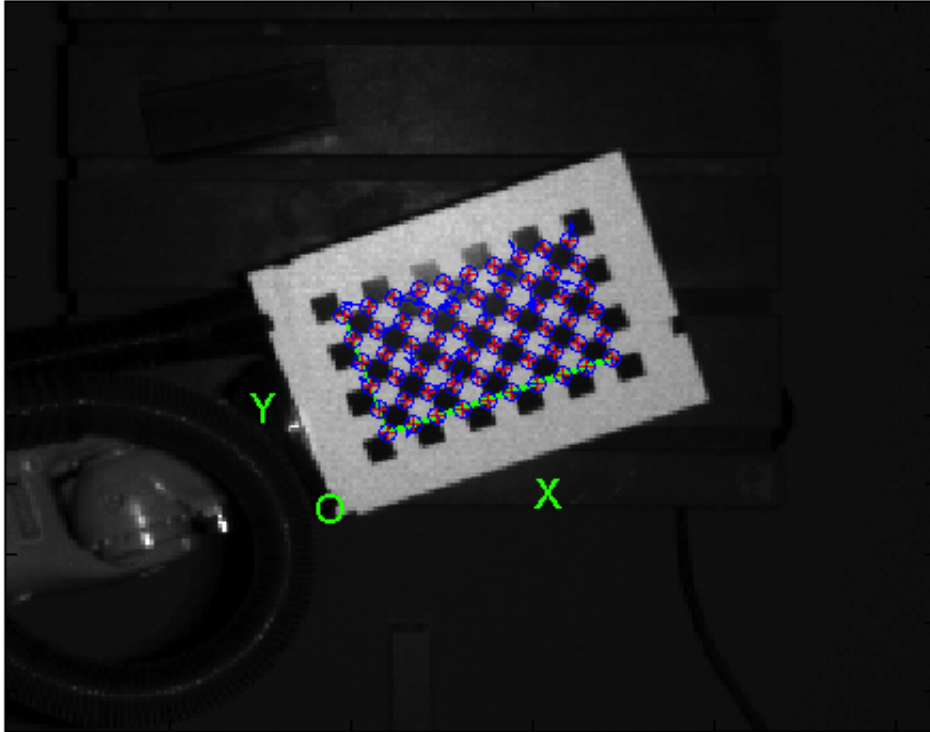


Figure 3.5: Example amplitude image of a checkerboard acquired by a Kinect ToF camera. The detected corners are marked " $\oplus$ ".

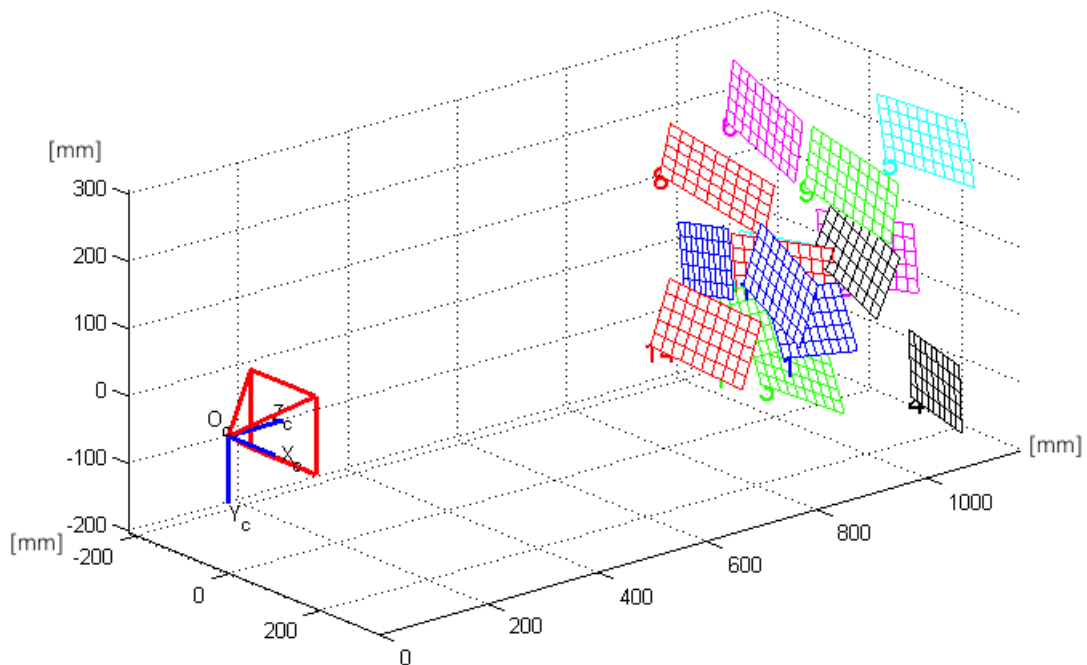


Figure 3.6: 3D position of the Kinect ToF camera with respect to the calibration checkerboards.



The calibration toolbox also calculates radial and tangential distortion coefficients, using a distortion model called "Plump Bob" (radial polynomial + "thin prism"). This model was introduced by Brown [22] in 1966. The radial distortion arises from the symmetry of a photographic lens, while the tangential distortion is due to imperfect centering of the lens components and other manufacturing defects. Distortion parameters are presented as a vector of five elements. Radial factors are denoted with  $k$  and tangential with  $p$ . Distortion vector for the Kinect ToF camera was found to be:

$$k_c = \left[ k_{(1)} \quad k_{(2)} \quad p_{(1)} \quad p_{(2)} \quad k_{(3)} \right] = \begin{bmatrix} 0.06646 \\ -0.02316 \\ -0.00224 \\ -0.00370 \\ 0.00000 \end{bmatrix}^T \quad (3.5)$$

The distortion parameters turned out to be of little importance in order to achieve good measurements, thus the Kinect ToF camera appeared to be well calibrated out of the box. It also seemed to be most distortions at the image edges, which had little significance in this project where the middle part was mostly used.

### Extrinsic Parameters

The extrinsic parameters indicate the external position and orientation of the camera in the 3D world or in relation to a known coordinate frame. In this project the camera has been fixed with its field of view from above over a worktable, on which the robot is welding the component parts together. Figure 3.7 shows the various coordinate frames and the relations between them.

The Kinect ToF camera is calibrated in relation to a coordinate frame originating from the corner of the worktable.  ${}^C\xi_B$  is the transformation from the camera coordinate frame to the worktable



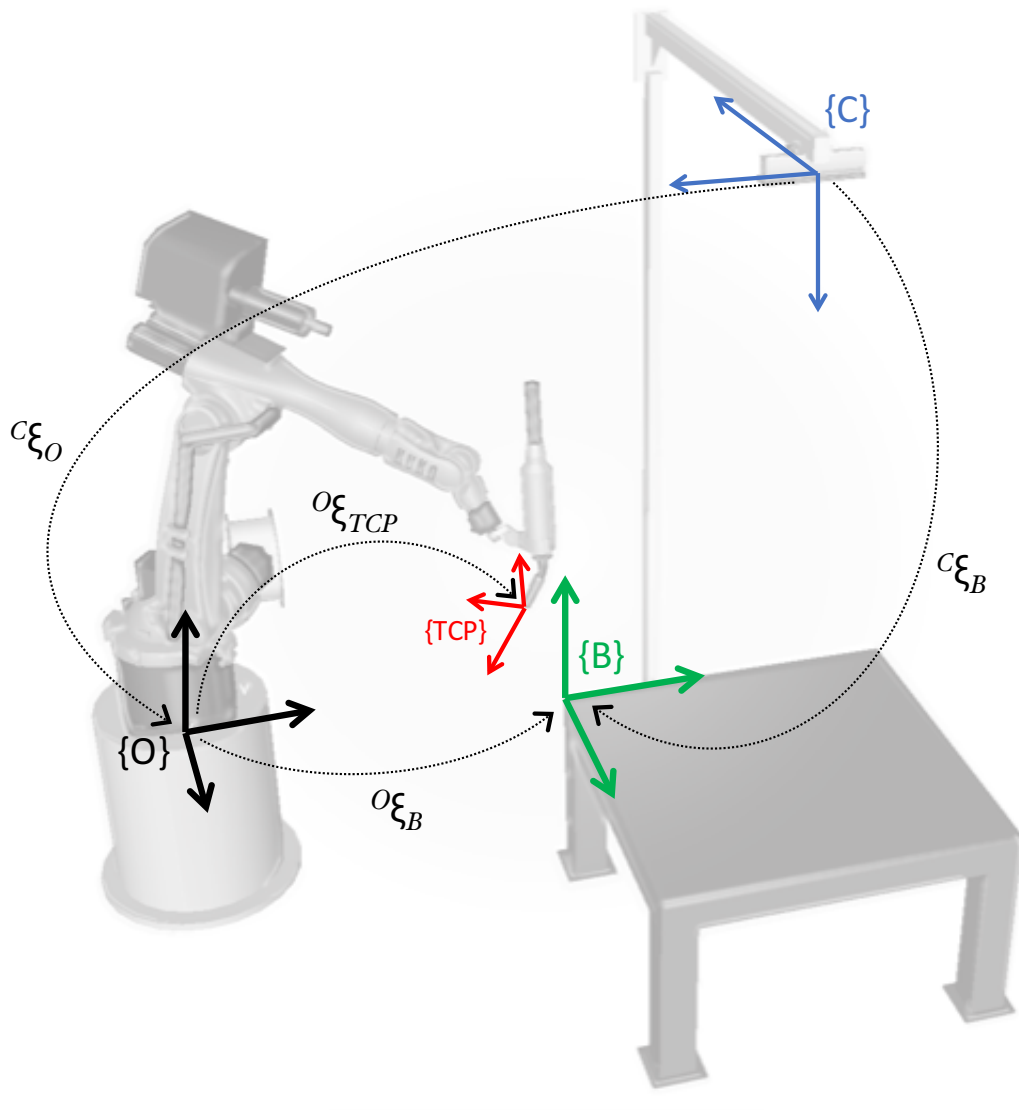


Figure 3.7: The relation between camera (C), world/robot (O), worktable (B), and *Tool Center Point* (TCP) coordinate frames.

coordinate frame. By using the Calibration Toolbox, this transformation was found to be:

$${}^C\xi_B = \begin{bmatrix} R & t \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} -0.217295 & 0.975797 & -0.024544 & 97.813596 \\ 0.976069 & 0.217000 & -0.014146 & -291.230500 \\ -0.008478 & -0.027031 & -0.999599 & 1391.410813 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (3.6)$$

The transformation  ${}^O\xi_B$  is extracted from the KUKA controller after defining the coordinate frame at the worktable as a *Robot Base Frame* in the controller. This transformation is in the robot controller defined by  $X, Y, Z, A, B, C$  values, but can be described in the form of a square rigid transformation matrix:

$${}^O\xi_B = \begin{bmatrix} R & t \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} -0.2074 & -0.9783 & 0 & 702.0100 \\ 0.9783 & -0.2074 & 0 & 1449.0900 \\ 0 & 0 & 1 & -17.2500 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (3.7)$$

The pose of the robot relative to the fixed camera is thus given by

$${}^C\xi_O = {}^C\xi_B \oplus ({}^O\xi_B)^{-1}. \quad (3.8)$$

### 3.3 Point Cloud Processing

Point Cloud Processing involves steps such as point cloud preprocessing and 3D feature estimation. The purpose of this processing is to enhance the quality of the data obtained in the point cloud acquisition step. When acquired, the point clouds are often degraded due to distortion and noise in the camera system. Another problem is the massive amount of data captured in each point cloud, which can greatly reduce the effect of recognition and alignment algorithms. Down-sampling, smoothing, and estimating local surface geometry are all examples of operations necessary to make the data in point clouds more meaningful.

### 3.3.1 Down-Sampling

In order to reduce the computation time and improve the efficiency of subsequent algorithms, it is necessary to reduce the overall size of the data set. A naive approach for reducing the density of point clouds is to *down-sample* by randomly selecting points. This can be done by using *Algorithm A* from Vitter [23], which is random sampling with uniform probability.

To preserve more of the information while reducing overall size of the data set, a *voxelized* grid approach can be used rather than random sampling. The term *voxel* is short for "volume pixel", which can be thought of as a tiny box in three-dimensional space. In the voxelized grid approach, a 3D voxel grid is created over the input point cloud data. Each voxel then represents a group of points that are close enough to each other to be approximated (i.e., *down-sampled*) by a single point. This point is the arithmetic mean of the group of points, thus their centroid. The approach is illustrated in Figure 3.8.

An advantage of the *voxelized* grid method is that the filter is easily parameterized by choosing the size of the voxels, making it easy to align the density of point clouds from different sources. The centroid is chosen instead of the center of the voxel because it represents the real environment more accurately. This is a bit slower than approximating with the center of the voxel, but it represents the underlying surface more accurately.

### 3.3.2 Statistical Outlier Removal

Acquired point clouds typically have varying point densities. Measurement errors and noise can also lead to sparse outliers, which could corrupt and degrade the results from alignment algorithms. By performing a statistical analysis on each point's neighborhood, and reject those which do not meet a certain criteria, some of these problems can be solved.

The outlier removal performed in this project is based on computing the average distance each point has to its nearest  $k$  neighbors. A distance threshold is then determined based on the mean and standard deviation of all the point-to-neighbors distances. Finally, all of the points are classified as *inlier* or outlier according to the threshold, and the outliers are rejected.

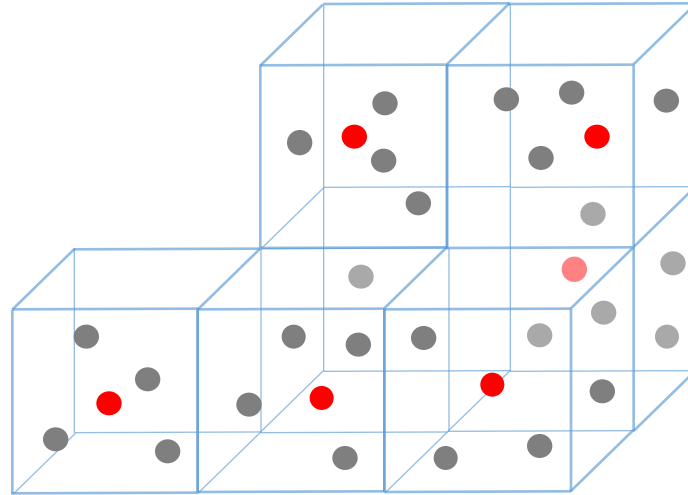


Figure 3.8: Voxel grid filtering. The initial points (*grey dots*) in each voxel with sides  $r$  are approximated and replaced by their centroid (*red dots*).

### 3.3.3 Moving Least Squares Smoothing

The acquired point cloud can be thought of as a series of implicitly defined surfaces. Although the point cloud is down-sampled and outliers are removed, these surfaces may still contain imperfections and errors. These errors mainly originate from noise intrinsic to the camera and its interaction with the surfaces being acquired [24]. By locally approximating the surface with polynomials using moving least squares (MLS), it is possible to achieve a smoother and more accurate representation.

The MLS method was proposed by Lancaster and Salkauskas in 1981 [25] for smoothing and interpolating irregularly distributed data. A MLS surface provides an interpolating surface for a given point cloud by fitting higher order bivariate polynomials to each point neighborhood locally [26]. MLS is initialized with a weighted least squares formulation for a single point. It is then *moved* over the entire domain, computing and evaluating a weighted least squares fit for each point. A more detailed description of MLS may be found in [27] and is beyond the scope of this thesis. Figure 3.9 shows an acquired point cloud before and after MLS smoothing.

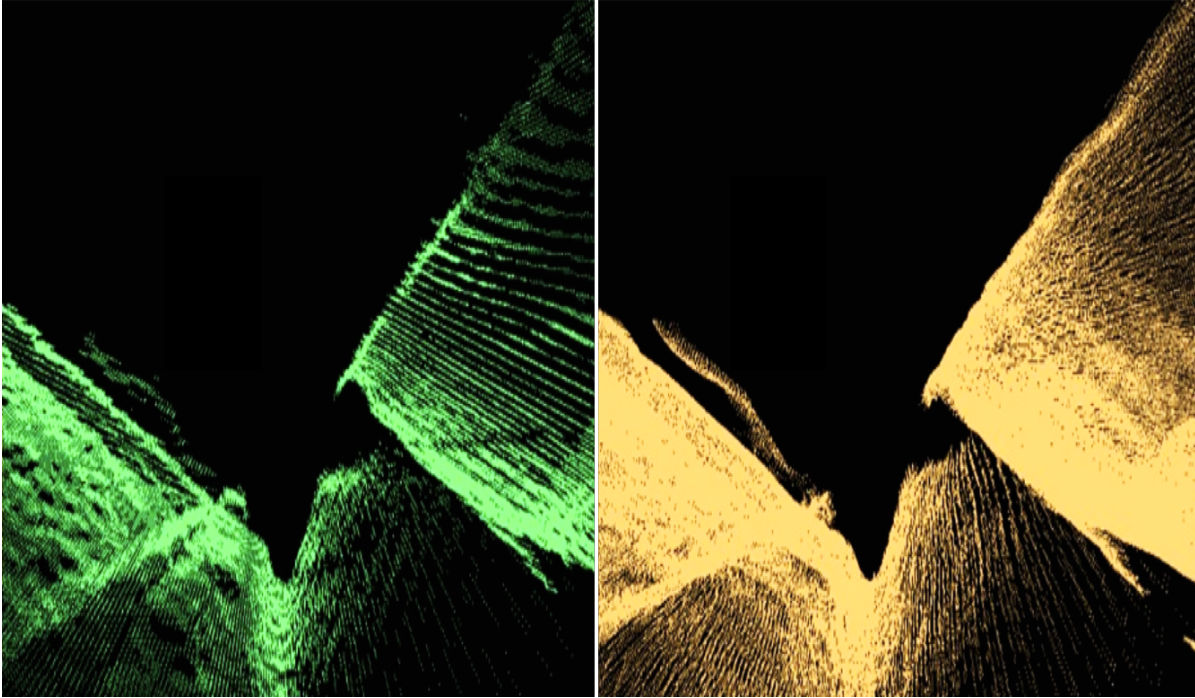


Figure 3.9: The effect of moving least squares (MLS) smoothing. The left side shows a unfiltered point cloud where the surface micro structure is revealed. The right side shows the same point cloud after MLS smoothing.

### 3.3.4 Estimating Surface Normals

Some of the algorithms evaluated in this work require information about the normal at each of the points in the point cloud. The local surface around a query point can be represented by a estimation of a local feature using the neighboring points. An important problem in describing the geometry of the surface is to first infer its orientation in a coordinate system, that is, estimate its normal [26]. Surface normals contains a lot of information about the underlying surface and are heavily used in many areas such as rendering, making visibility computations, answering inside-outside queries, etc. [28]

The simplest method for estimating surface normals is based on the first order 3D plane fitting as proposed by [29]. Determining the normal to a point on an estimated surface can be approximated by the normal of a plane tangent to the surface, thus becoming a least-square plane fitting estimation for the neighboring points. The tangent plane is represented by a point  $\mathbf{x}$  and a normal vector  $\vec{\mathbf{n}}$ . The query point is denoted  $\mathbf{p}_i$ , the neighboring points  $P^k$ . The distance between a point  $\mathbf{p}_i \in P^k$  and the plane is defined as  $d_i = (\mathbf{p}_i - \mathbf{x}) \cdot \vec{\mathbf{n}}$ . The  $\mathbf{x}$  and  $\vec{\mathbf{n}}$  values are

computed in a least-square sense so that  $d_i = 0$ . If

$$\mathbf{x} = \bar{\mathbf{p}} = \frac{1}{k} \cdot \sum_{i=1}^k \mathbf{p}_i \quad (3.9)$$

is the centroid of  $P^k$ , the solution for  $\vec{\mathbf{n}}$  is found by analyzing the eigenvectors and eigenvalues of the covariance matrix  $C \in \mathbb{R}^{3 \times 3}$  of  $P^k$ , expressed as:

$$C = \frac{1}{k} \sum_{i=1}^k \xi_i \cdot (\mathbf{p}_i - \bar{\mathbf{p}}) \cdot (\mathbf{p}_i - \bar{\mathbf{p}})^T, \quad C \cdot \vec{v}_j = \lambda_j \cdot \vec{v}_j, \quad j \in \{0, 1, 2\} \quad (3.10)$$

The term  $\xi_i$  is representing a weight for  $\mathbf{p}_i$ , and is usually equal to 1. The eigenvalues of  $C$  are real numbers  $\lambda_j \in \mathbb{R}$ , and the eigenvectors  $\vec{v}_j$  form an orthogonal frame, corresponding to the principal components of  $P^k$ . If  $0 \leq \lambda_0 \leq \lambda_1 \leq \lambda_2$ , the eigenvector  $\vec{v}_0$  corresponding to the smallest eigenvalue  $\lambda_0$  is approximating  $+\vec{\mathbf{n}}$  or  $-\vec{\mathbf{n}}$ .

### 3.4 Object Alignment

Object Alignment is the problem of finding a transformation that takes one object pose to another. The goal is to align the objects so that they both share the same coordinate system. In the most straightforward form, the transformation between the objects consists of rotation, translation, and perhaps scale. For point clouds, object alignment means to find correct point correspondences in a given dataset, and estimating transformations that can rotate and translate each individual dataset into a consistent coordinate framework.

The various methods for aligning point clouds can be divided into two main approaches [5]. One is to search for the right transformation by estimating correspondences, then estimating a transformation given a correspondence, and repeating. Such approaches can be classified as local optimization methods, and in this category the most popular method is indubitably the *Iterative Closest Point* (ICP) algorithm [30]. A global alternative for alignment optimization is based on searching for small groups of points that correspond, and then use them to estimate the transformation. *Sample Consensus Initial Alignment* (SAC-IA) is an example of the latter.

Development of an offline programmed robotic welding sequence includes establishing the

pose of a simulated representation (i.e., a CAD model) of the object to be welded. This pose usually provides a good initial guess of the true physical object pose, and serves as a good starting point for object aligning by local approaches. The initial alignment is relatively easy to obtain for applications described in this work. ICP, which is considered a fine tuning alignment method [13], then emerges as an appealing alternative for object alignment. ICP is thus subject to a more thorough evaluation than SAC-IA, which provides a second point of comparison.

*Iterative Closest Point* and *Sample Consensus Initial Alignment* are explained in Section 3.4.1 and Section 3.4.2 respectively. Due to numerous proposed versions, three different ICP approaches are evaluated.

### 3.4.1 Iterative Closest Point

One of the most popular methods for aligning unorganized point clouds is the Iterative Closest Point (ICP) algorithm, which aims to find the best transformation between two point clouds by minimizing the distance between corresponding entities. ICP starts with a *target* and a *model* point cloud, and usually an initial guess for the rigid-body transformation between them. The algorithm then iteratively tries to find the optimal transformation by generating pairs of corresponding points and minimizing an error metric. The correspondences are thus continuously reconsidered as the solution comes closer to the error metric local minimum. Since the *model* cloud points usually are matched to their nearest 3D point in the *target* cloud in a linear way, ICP can be considered a brute force method. An example of the ICP method is shown in Figure 3.10. The ICP algorithm was introduced by Chen and Medioni [33] and Besl and McKay [34] in the early 90s, but many variations on the basic ICP concept have later been introduced. Most of the approaches can be summarized in six steps [35]:

1. **Selecting** a set of points in one of the clouds.
2. **Matching** these points to samples in the other cloud.
3. **Weighting** the corresponding point pairs.
4. **Rejecting** certain pairs.

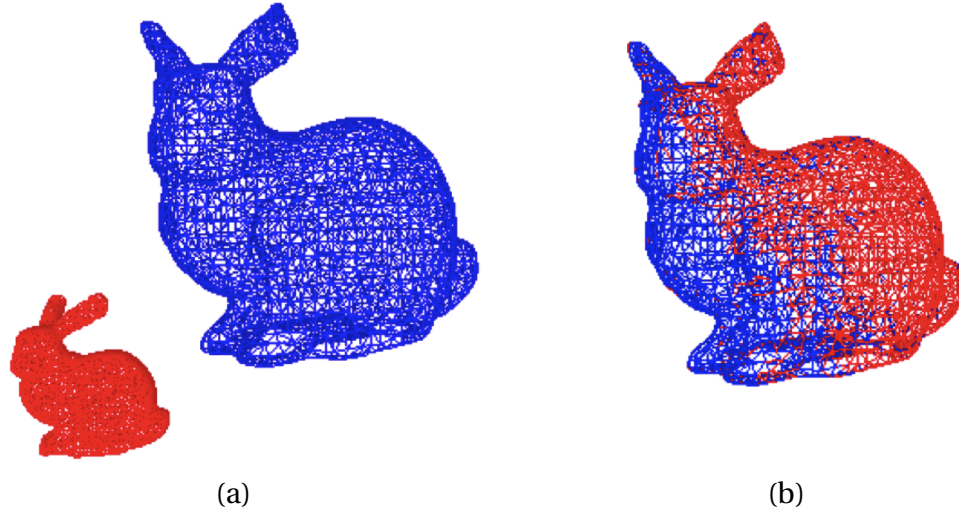


Figure 3.10: Iterative Closest Point alignment for the 3D bunny data set originating from Stanford 3D scanning repository [31]. The scene is shown in red, and the model set is shown in blue. (a) Initial configuration. (b) Result of the ICP algorithm. Figure from [32]

5. **Assigning** an error metric based on the point pairs.

6. **Minimizing** the error metric.

The standard ICP algorithm aims to find the transformation consisting of a rotation  $\mathbf{R}$  and a translation  $\mathbf{t}$  which minimizes a cost function  $E(\mathbf{R}, \mathbf{t})$ . Given two independent point clouds,  $M$  (model cloud) and  $D$  (target or data cloud) describing the same shape, the associated cost function is as follows:

$$E(\mathbf{R}, \mathbf{t}) = \sum_{i=1}^{|M|} \sum_{j=1}^{|D|} w_{i,j} \|\mathbf{m}_i - (\mathbf{R}\mathbf{d}_j + \mathbf{t})\|^2. \quad (3.11)$$

Where  $w_{i,j}$  is 1 if the  $i$ -th point of  $M$  describes the same 3D point as the  $j$ -th point of  $T$ , and 0 otherwise [36]. The various points in the model cloud is denoted  $\mathbf{m}_i$  ( $i = 1, \dots, |M|$ ), and  $\mathbf{d}_j$  ( $i = 1, \dots, |T|$ ) denotes the points in the target cloud [37].

The optimal transformation  $E(\mathbf{R}, \mathbf{t})$  has to be computed for every iteration. Since the point correspondence matrix can be represented by a vector containing the point pairs, the cost function (Eq. 3.11) can be reduced to

$$E(\mathbf{R}, \mathbf{t}) \propto \frac{1}{N} \sum_{i=1}^N \|\mathbf{m}_i - (\mathbf{R}\mathbf{d}_i + \mathbf{t})\|^2, \quad (3.12)$$



where  $N = \sum_{i=1}^{|M|} \sum_{j=1}^{|D|} w_{i,j}$ .

ICP is applicable when a rough initial alignment is available, due to its susceptibility to local minima. The algorithm is also time-consuming when applied to large datasets, and it generally needs many iteration steps until convergence is reached. Various efforts to reduce the extent of these problems have led to different ICP versions. In this work, three of these contributions have been tested and evaluated for use in object alignment.

### ICP by Singular Value Decomposition

A simple but robust version of the ICP algorithm is based on the work of Arun et al [38] from 1987, where the transformation is estimated by Singular Value Decomposition (SVD). It is shown by [36] that the error function (Eq. 3.12) for this ICP version becomes:

$$E(\mathbf{R}, \mathbf{t}) \propto \sum_{i=1}^N \left\| \mathbf{m}'_i - \mathbf{R} \mathbf{d}'_i \right\|^2 \quad \text{with} \quad \mathbf{t} = \mathbf{c}_m - \mathbf{R} \mathbf{c}_t \quad (3.13)$$

The centroids of the points,

$$\mathbf{c}_m = \frac{1}{N} \sum_{i=1}^N \mathbf{m}_i \quad \text{and} \quad \mathbf{c}_t = \frac{1}{N} \sum_{i=1}^N \mathbf{d}_j, \quad (3.14)$$

is used to decouple the calculation of the rotation  $\mathbf{R}$  from the translation  $\mathbf{t}$ .

The algorithm minimizes the error function until the difference between the previous transformation and the current estimated transformation is smaller than an user imposed value. It is also possible to terminate the algorithm by setting a threshold for how small the sum of Euclidean squared errors should be, or by setting the maximum number of iterations the algorithm can run.

### ICP with Normals

In the standard ICP algorithm as described above, each point in a cloud is paired with the closest point in another cloud with a point-to-point error metric. Another approach is to use a point-to-plane error metric, where the sum of squared distance between a point and a tangent plane at its

corresponding point is minimized. The point-to-point error metric has a closed-form solution, while the point-to-plane metric can be solved by using nonlinear least squares methods, such as the Levenberg-Marquardt method [39]. The point-to-point ICP is generally iterating faster than the point-to-plane version, but [35] observed significantly better convergence in the latter, making the version worthy of being tested in this project.

In this version of the ICP algorithm, the error metric is minimized along surface normals, and is thus taking advantage of more information at each point. This improvement can be implemented by changing the inner part of the cost function (Eq. 3.12) to

$$E(\mathbf{R}, \mathbf{t}) \propto \frac{1}{N} \sum_{i=1}^N \left\| \eta_i (\mathbf{m}_i - (\mathbf{R}\mathbf{d}_i + \mathbf{t})) \right\|^2, \quad (3.15)$$

where  $\eta_i$  is the surface normal at  $\mathbf{m}_i$ .

The algorithm termination criteria corresponds to those of the standard ICP version.

### 3.4.2 Sample Consensus Initial Alignment

The *Sample Consensus Initial Alignment* (SAC-IA) algorithm estimate transformations between point clouds by using geometric point features. The method was proposed by Rusu et al [30], and is a variant of the *Random Sample Consensus* (RANSAC) algorithm [40]. SAC-IA uses *Fast Point Feature Histograms* (FPFH) as descriptors, which is computed using the estimated surface normals  $n_i = (n_x, n_y, n_z)$ .

The main steps of the algorithm are as follows:

1. Randomly selection of point samples from the model dataset  $M$ , at a minimum distance between points.
2. For each sample, find the most similar samples in the target dataset  $D$  by comparing point features.
3. In case of multiple matches, randomly choose one for each sample.

4. Compute the transformation between sample points in  $M$  and their correspondences in  $D$ , and evaluate a registration error metric.

The main steps are repeated for a predefined number of iterations, before the best transformation is chosen and refined by using the Levenberg-Marquardt algorithm.

The accuracy of the SAC-IA alignment depends on point cloud characteristics and the quality of the computed descriptors. Robust descriptors should lead to good transformation estimates, descriptors with poor quality may cause unpredictable results.

### **Fast Point Feature Histograms**

Geometrical properties of point cloud points at different surfaces can produce unique signatures when represented by histograms. These signatures provide a good basis for comparisons in alignment methods like SAC-IA.

FPFH descriptors are pose-invariant local features which represent the underlying surface properties for all the elements composing a point cloud. These features form a full description of a point cloud, and can thus be used for several tasks, like aligning a model point cloud to a target cloud. FPFH descriptors are computed for each point in the given point cloud, and are generated by comparing the estimated surface normals of a specific point with the normals of the points within a user-defined search radius. The relation between each point in the surface is a triplet of angles,  $\{\alpha, \phi, \theta\}$ . The angles spans between the point normals and  $d$ , the Euclidean distance between points in the surface.

### **3.4.3 Algorithm Performance**

Three variants of ICP and SAC-IA have been tested in order to find the best object aligning algorithm for this project. The robustness and quality of the achieved alignment was of particular interest. Time needed for estimating a transformation has also been evaluated for the different algorithms.

Two objects of different complexity have been studied when evaluating the algorithms. The first object is a simple rectangular steel tube, the second and more complex one is a corner-like object made up of three thick steel plates. The complex object is shown in Figure 5.1.

For effective comparison of the algorithms robustness and achieved alignment quality, an error metric called *fitness score* is calculated for all object alignments. The fitness score represents the distance error between the aligned clouds after the registration process, and is found by calculating the sum of squared distances from source points to the corresponding target points. It is desirable to minimize the fitness score, as a low distance error value represents a closer pointwise alignment. The robustness can be evaluated by studying the fitness score variation. A robust algorithm has only minor variations in the achieved fitness scores.

Evaluation of the algorithms have been performed by translating and rotating the components in small steps. For every step, the fitness score have been calculated and elapsed time measured. Calculations have been carried out on a computer running Windows 8.1 with a 3.4 Ghz Intel i7 processor. The translations and rotations have been conducted from various starting points on the worktable. The physical setup is explained in more detail in Chapter 4.

Figure 3.11 shows the fitness score for translations in  $X$  and  $Y$  directions, and rotation about the  $Z$  axis for the two objects. When evaluating the robustness and accuracy of the algorithms, it was observed that the *standard ICP* and *nonlinear ICP* appeared to have highest accuracy for translations and rotations of the complex object. This was seen especially for translations up to about 130 mm, and rotations between  $-30$  and  $30$  degrees. *SAC-IA* and *ICP with normals* turned out to perform better for the less complex object, and by the *SAC-IA* method the object could be adequately aligned from significantly longer distances. More accurate alignments from relatively short distances were given higher priority in this work, and from that perspective the *nonlinear ICP* version was considered the most useful method, with results slightly better than the *standard ICP*. An interesting observation is that all methods seem to perform better for translations in  $X$  than  $Y$  direction. This might be a result of the specific setup for the tests carried out here.

In Figure 3.12 the elapsed alignment times of the four methods are compared for the two objects. *SAC-IA* is clearly the slowest by running for more than a second in most cases. *ICP with normals*

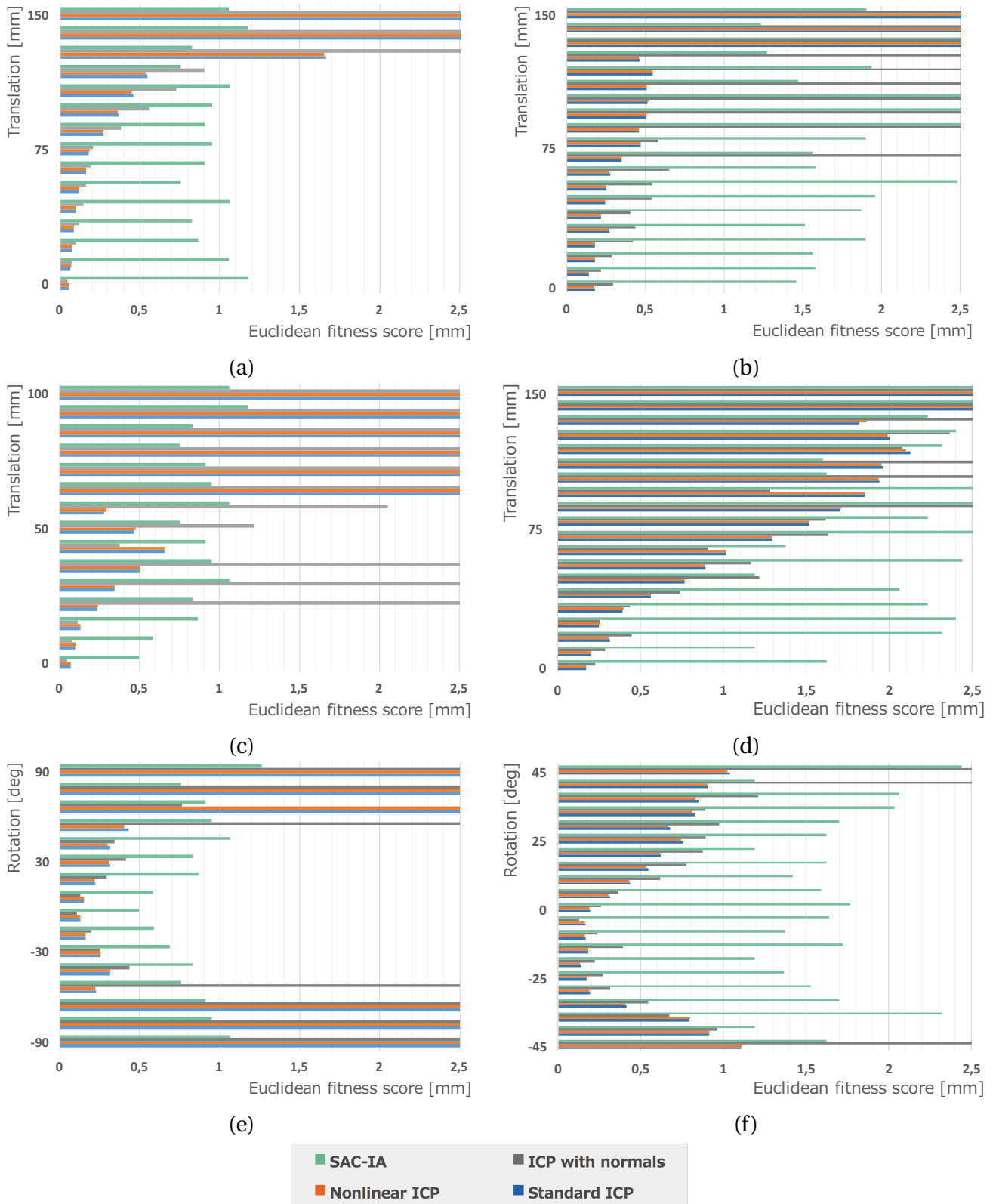


Figure 3.11: The achieved fitness score for the transformation estimations. In (a), (c), and (e), transformation of a simple rectangular steel tube is estimated. (b), (d), and (f) is estimations for a more complex object. Low values represent better estimation, bars running off the chart represents a failed transformation estimation. In (a) and (b), a X translation is performed. In (c) and (d), a Y translation, and a rotation about the Z axis is performed in (e) and (f).

is also a relatively slow method, requiring 689 milliseconds on average for all alignments. *Standard ICP* is the fastest method at 201 milliseconds on average, with *nonlinear ICP* just behind at 256 milliseconds on average. Faster runtimes can be achieved by further preprocessing and down-sampling of the point clouds. With limited computational resources, these results are a trade-off between accuracy and speed.

These tests show that SAC-IA is a good method for achieving a rough and initial alignment from relatively long distances, but is outperformed by the *standard ICP* and *nonlinear ICP* methods when it comes to quality and robustness of the alignment. The tests also show that *nonlinear ICP* performs slightly better than *standard ICP* in terms of alignment robustness, although the former requires a bit more time. For use in aligning offline programmed welding paths, *nonlinear ICP* will probably be the best solution given that a good initial alignment is available. *Nonlinear ICP* was therefore used in the practical experiments described in Chapter 5.

It must be emphasized that these are just brief tests of the algorithm performances. More thorough evaluations and different conditions might get different results.

### 3.4.4 Representing the Estimation Results

The resulting transformation estimation from the object alignment has the form of a square rigid transformation matrix:

$$T = \begin{bmatrix} R & t \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} r_{11} & r_{12} & r_{13} & p_x \\ r_{21} & r_{22} & r_{23} & p_y \\ r_{31} & r_{32} & r_{33} & p_z \\ 0 & 0 & 0 & 1 \end{bmatrix}. \quad (3.16)$$

In this project, the commands are sent to the robot as cartesian corrections. The KUKA KR5 robot system describes orientations by Tait-Bryan angles, using the *Yaw Pitch Roll (ZYX)* composite rotation convention. The notation is as follows:

- $A$  represents rotation about the  $Z$ -axis.

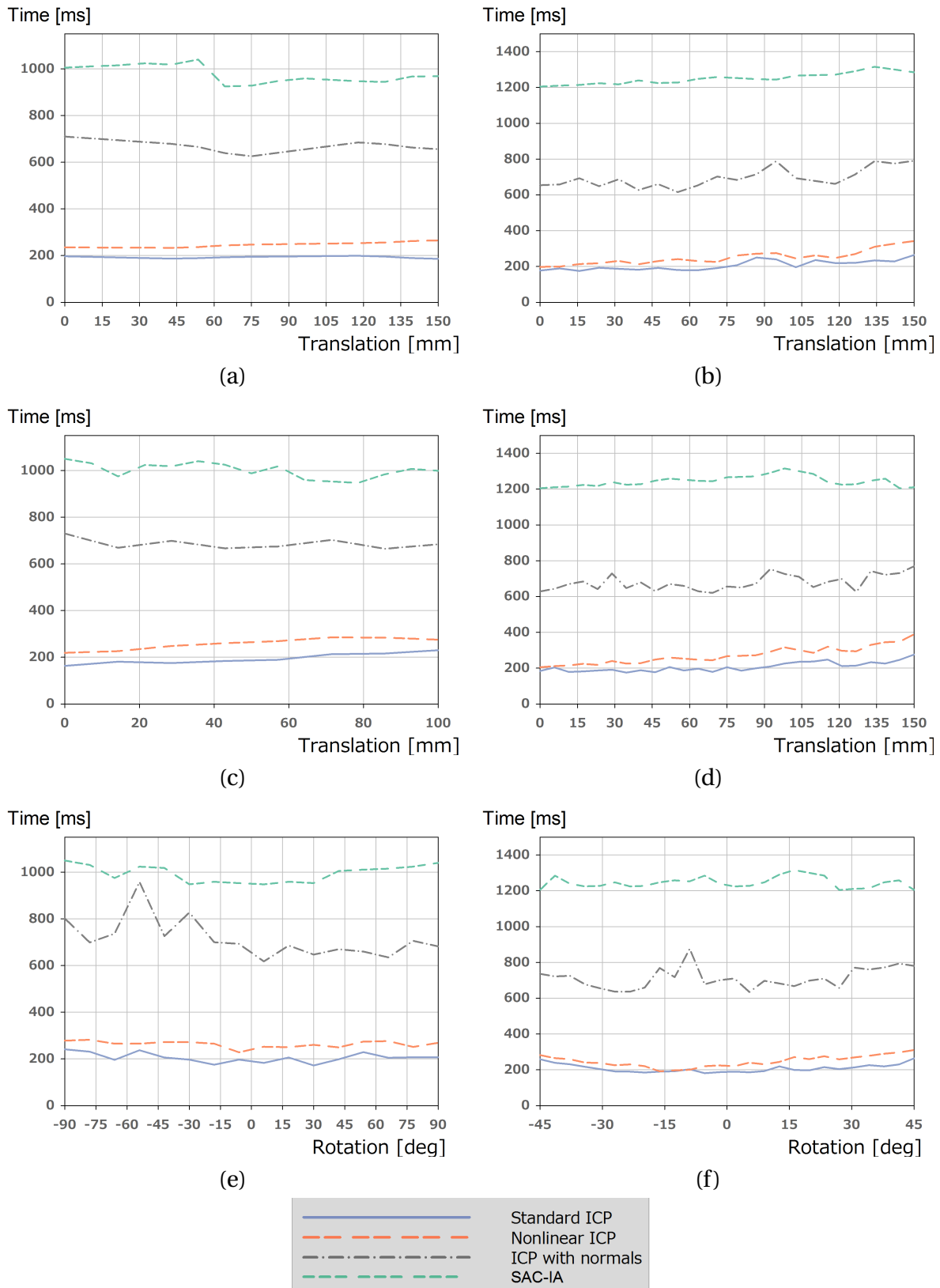


Figure 3.12: Running time for the various algorithms. In (a), (c), and (e), running times of a simple rectangular steel tube is measured. (b), (d), and (f) is the same measures for a more complex object. In (a) and (b), a X translation is performed. In (c) and (d), a Y translation, and a rotation about the Z axis is performed in (e) and (f)

- $B$  represents rotation about the  $Y$ -axis.
- $C$  represents rotation about the  $X$ -axis.

The rotation matrix  $R$  from Eq. 3.16 is thus composed by:

$$R = \begin{bmatrix} \cos(A) \cos(B) & \cos(A) \sin(B) \sin(C) - \sin(A) \cos(C) & \cos(A) \sin(B) \cos(C) + \sin(A) \sin(C) \\ \sin(A) \cos(B) & \sin(A) \sin(B) \sin(C) + \cos(A) \cos(C) & \sin(A) \sin(B) \cos(C) - \cos(A) \sin(C) \\ -\sin(B) & \cos(B) \sin(C) & \cos(B) \cos(C) \end{bmatrix}$$

The  $ABC$  angles can be extracted from the estimated transformation in Eq. 3.16:

$$\text{If } (r_{11} = r_{21} = 0 \Leftrightarrow \cos B = 0), \text{ then } \begin{cases} A = 0, \\ B = \frac{\pi}{2}, \\ C = \tan_2^{-1}(r_{12}, r_{22}) \end{cases}$$

$$\text{Else, then } \begin{cases} A = \tan_2^{-1}(-r_{31}, \sqrt{r_{11}^2 + r_{21}^2}), \\ B = \tan_2^{-1}(r_{12}, r_{11}), \\ C = \tan_2^{-1}(r_{32}, r_{33}) \end{cases}$$



# Chapter 4

## Practical Setup

### 4.1 Robot Cell

The *KUKA KR 5* robot used for the welding operations is part of a larger cell containing four KUKA robots in total. Besides the *KR 5* robot, the cell consists of a *KR 16* and two *KR 120* robots. All of them are equipped with their own *KUKA KR C4* robot controller. Safety features like emergency stop and door switch connects to a common safety PLC over *ProfiNET*.

Figure 4.1 illustrates the cell layout for this project. The illustration is created by using the *3DAutomate* software by VisualComponents and *KUKA.Sim*.

A *Fronius TransSteel 5000* welding machine is also part of the system. It is physically attached to the *KR 5* robot manipulator and communicates with the *KR C4* robot controller. Figure 4.2 shows how robot and welding machine are physically connected.

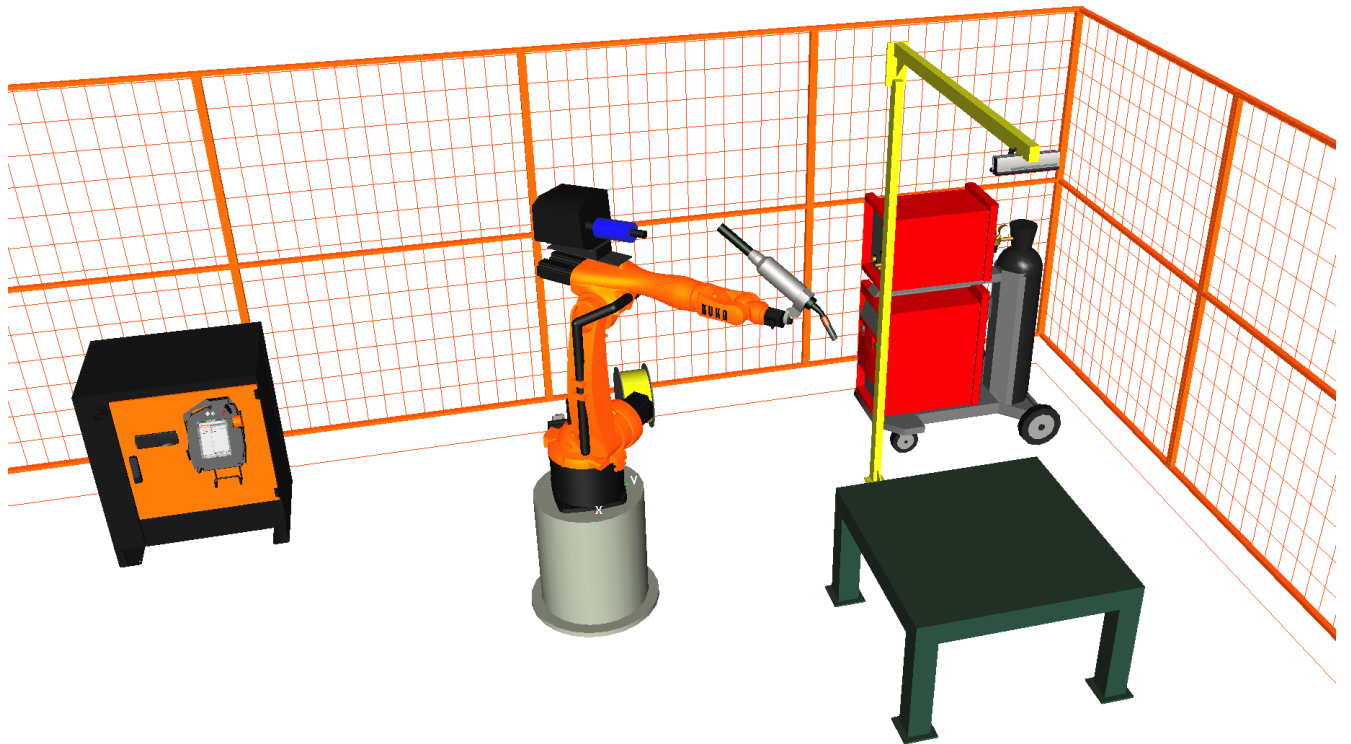


Figure 4.1: Graphical model of the test cell layout at Department of Production and Quality Engineering, NTNU. The Kinect is mounted over the worktable, looking down at the table.

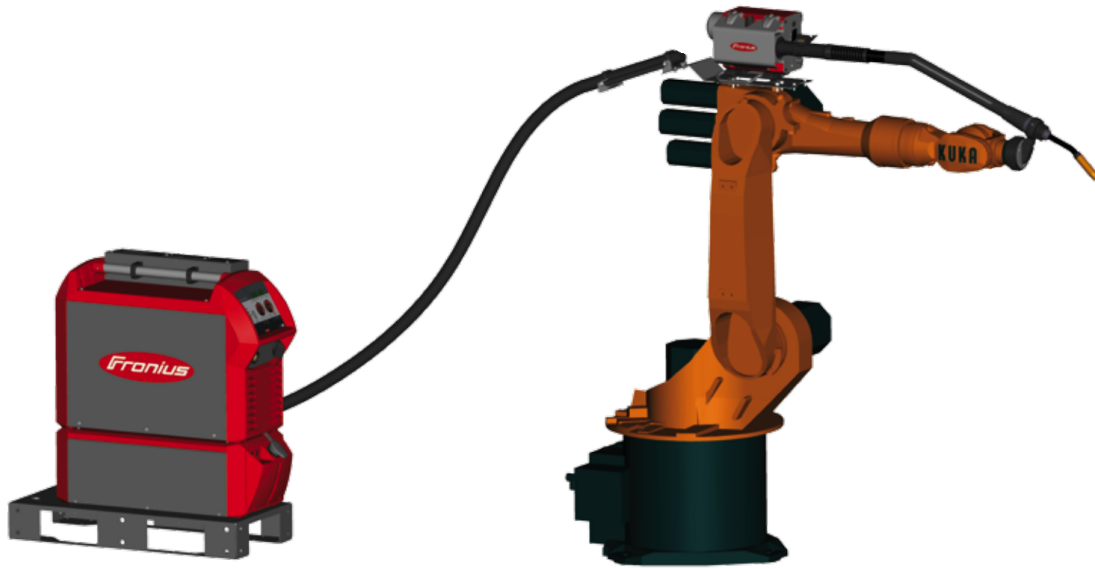


Figure 4.2: A model showing how the KUKA KR 5 robot is setup and connected to a Fronius TransSteel 5000 welding machine. Figure from [41].

The welding machine is with the current configuration set up for *metal active gas* (MAG) welding. The welding rod thickness is 1.0 mm, and the power source supports welding with currents up to 500 A [41]. Setup and testing of the welding equipment was carried out in the first period of the project.

## 4.2 Information Flow

The system combines data from the depth camera with prior knowledge from a CAD model and an offline programmed welding sequence to determine where the robot will weld. Existing KUKA infrastructure has been the basis for the distribution of data in the system. In addition, a multithreaded C++ program was developed in *Microsoft Visual Studio 2013*. The C++ program serves as an interface between the Kinect and the robot, and handles the different algorithms and calculations. Source code for this program is provided in Appendix A. Requirements for the interface to work are a computer running Windows 8.1 (due to Kinect driver support) and a working installation of *Point Cloud Library* (PCL). By installing the prebuilt binaries from the

PCL website all other dependencies are automatically included and installed.

Figure 4.3 gives an overview of the information flow in the system. The purple box contains the C++ program where the various sub-components and information flow between them are shown. The CAD model is part of the information flow both through offline programming of the welding sequence and as input to the object adjustment process. The desired translation and rotation of welding sequences are sent to the robot controller as  $X', Y', Z', A', B', C'$  values.

### 4.2.1 Offline Programming of Welding Sequence

*KUKA.Sim* is used for both building a 3D model of the robot cell and for developing welding sequences. When developing these sequences it is required to have a CAD model of the component to be welded and the model of the physical robot cell. *KUKA.Sim* has the ability to import CAD models from a variety of different 3D computer-aided design applications. In this project, *AutoCAD* from Autodesk has been used.

When the welding sequences are developed it is necessary to perform accurate and realistic simulations of the welding operations. By using the Virtual Robot Controller (VRC) *KUKA.OfficeLite* from KUKA, the programming and optimizations can be made on a system that is nearly identical to the system software for the KR C4 robot controller. The robot application program sequences is executed in real time by the VRC, making cycle time optimizations possible.

*KUKA.OfficeLite* is also a necessity in order to perform post-processing of weld sequences and generating programs for the robot to execute.

The simulated 3D position and orientation of the component to be welded is required input parameters for the object alignment. It represents the system's initial guess for the pose of the physical component, and any deviations are measured relative to this pose. The input parameters are represented in terms of  $X, Y, Z, A, B, C$  values.

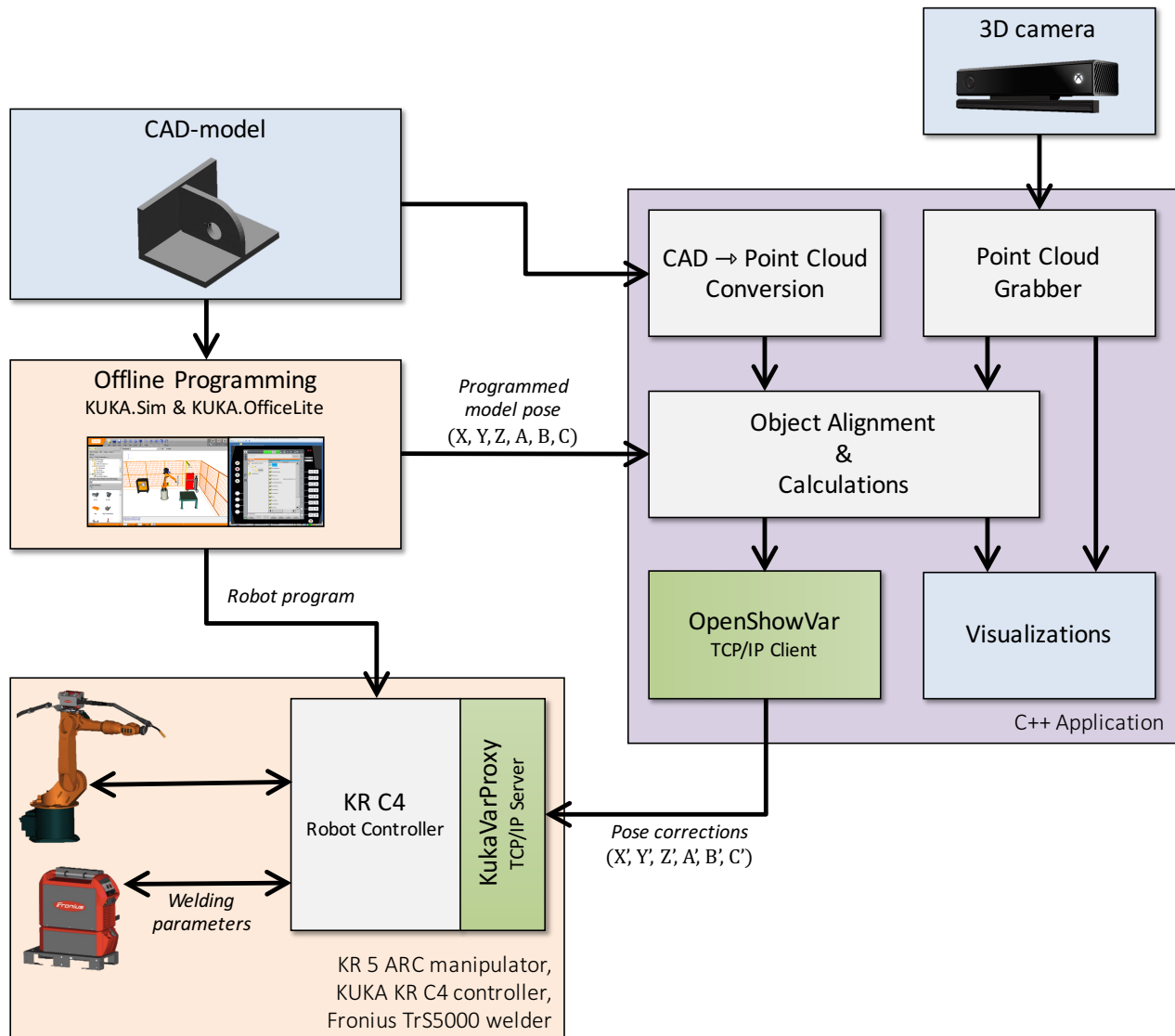


Figure 4.3: Information flow of the developed system. A CAD model and data from the Kinect 3D camera is the basis for all activities. The developed C++ application estimates a corrected component pose, and communicates the pose to the KRC4 robot controller. Offline programmed welding paths are then performed by the robot, based on the corrected component pose.

### 4.2.2 Robot Program

The programmed robot welding sequence is the main output from the offline programming process, containing all information necessary for the robot to perform the desired welding operations. In addition to information about planned positions, velocities, and accelerations, the sequence also includes welding parameters such as current and weaving data for each weld.

The 3D points of the offline programmed robot welding sequence originates from a coordinate frame located on the worktable. If the system detects deviations from the programmed and simulated component pose, the sequence points are transformed into the new location and orientation. This transformation is performed by the robot controller at program runtime, by transforming the origin of the coordinate frame. The new pose is represented in terms of  $X, Y, Z, A, B, C$  values and comes from the C++ program.

### 4.2.3 Point Cloud Capturing

The depth stream from the Kinect 3D camera is running at 30 frames per second. This stream is captured and converted to point clouds in the C++ application by using the standard driver provided by Microsoft and the Point Cloud Library framework. The Kinect often require a few seconds to start and to calibrate itself to give accurate depth measurements, so the depth stream is started with the C++ program and continues for as long as the program runs.

Capturing and converting the depth stream into point clouds are performed in real time in a separate thread. The point cloud is also transformed from Kinect's coordinate frame to the coordinate frame located on the worktable.

It is possible to retrieve two kinds of point clouds from the Kinect, with or without color information for each point. If just the depth information is retrieved, the point cloud is stored in the form *PointXYZ*. The alternative is *PointXYZRGBA*, which will give depth data aligned to RGB data from the high definition RGB camera on the Kinect.



Figure 4.4: A CAD model of the KUKA KR5 robot and its resulting point cloud. The CAD model is here converted to 50 000 points.

#### 4.2.4 Representing the CAD model

A 3D CAD model and information about its pose in the offline programming environment is required in order to do object alignments. It is preferable to import a CAD model file directly without having to convert it into another file format or perform other processing. On the other hand, the object aligning algorithms can only evaluate point clouds. The solution is to import the file as a CAD model and then convert it to a point cloud by sampling a specified number of points on all surfaces of the model. This is done by applying a variety of built in methods in PCL, as well as methods and data structures from the *Visualization Toolkit* [42] (VTK). Figure 4.4 illustrates a CAD model represented by the initial mesh and its corresponding point cloud.

The developed C++ application has the ability to import CAD files of *STereoLithography* (STL) format in *ASCII* representation. The STL format is also known as *Standard Tessellation Language* and is widely used for rapid prototyping, 3D printing and computer-aided manufacturing [43]. It is also a file format supported by most CAD drawing programs, including *AutoCad* [12] from Autodesk, which was used to create models and drawings for this project.

### 4.2.5 Object Alignment and Calculations

Object alignment and calculation of the desired corrections of object pose are the core functions in the developed C++ program. Point Cloud Library has several methods and algorithm implementations for aligning objects. As described in Chapter 3.4.3, the Iterative Closest Point (ICP) algorithm has been used for the alignment in this project. More specifically, the *nonlinear* implementation of ICP with Levenberg-Marquardt optimization backend. If the algorithm has converged towards a possible solution, it returns a  $4 \times 4$  transformation matrix. This matrix must be converted to a representation that the robot controller can utilize when correcting the pose, i.e.,  $X, Y, Z, A, B, C$  values.

### 4.2.6 Visualizations

Good visualizations are necessary to assess the results of the object alignment, as ICP can converge to false matches for some challenging poses. Visualizations are also needed for verifying proper representation of the imported CAD model. Point Cloud Library comes with its own visualization library, based on VTK [42]. This library allows for interactive rendering of the data stream from Kinect, the imported CAD model, and the results of the alignment algorithms. Figure 4.5 shows how the data from Kinect is visualized together with the CAD model. Figure 4.6 shows a captured point cloud and the point cloud representation of the CAD model before and after alignment.

### 4.2.7 TCP/IP Communication

Several alternatives were considered for passing information between the KR C4 robot controller and the computer hosting the Kinect and C++ program. Supplementary software packages provided by KUKA such as *KUKA.RobotSensorInterface* (RSI) or *KUKA.Ethernet KRL XML* [44] makes it possible to externally influence the program execution. However, these methods are intended for communicating with sensors in a direct manner, and were considered too complex and time-consuming to implement for this application.



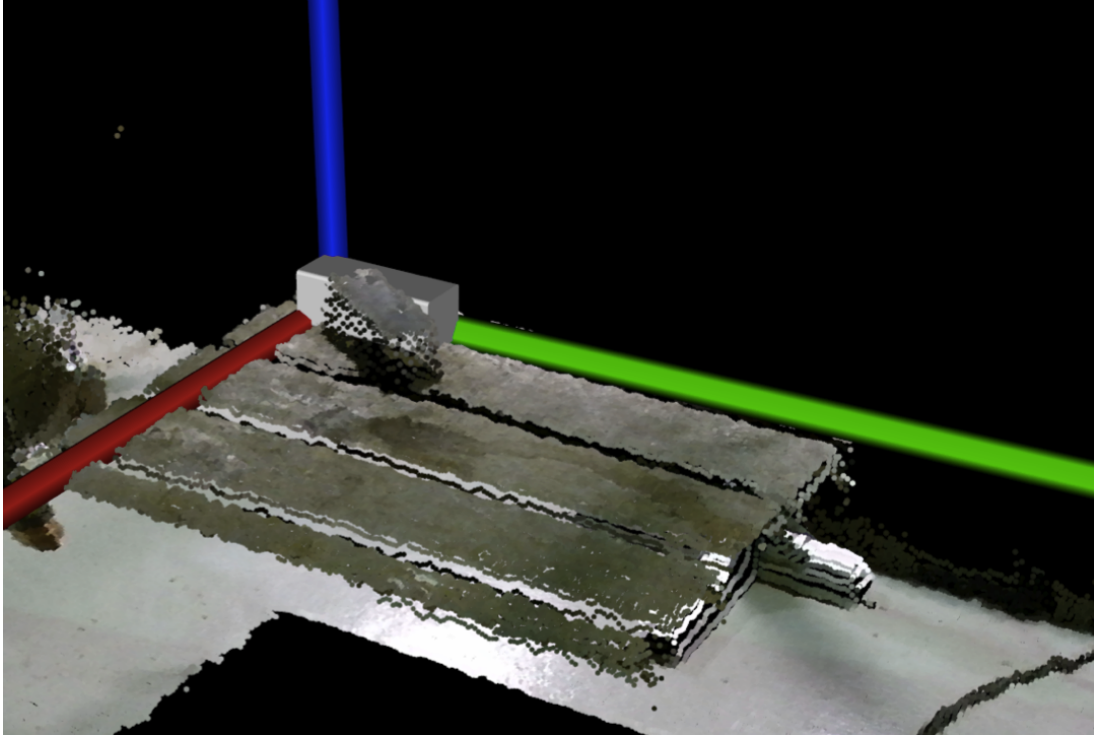


Figure 4.5: The implemented visualization of the Kinect depth data stream with a CAD model inserted. The calibrated camera coordinate frame origin is marked by the 3D origin marker in red ( $X$ ), green ( $Z$ ), and blue ( $Y$ ).

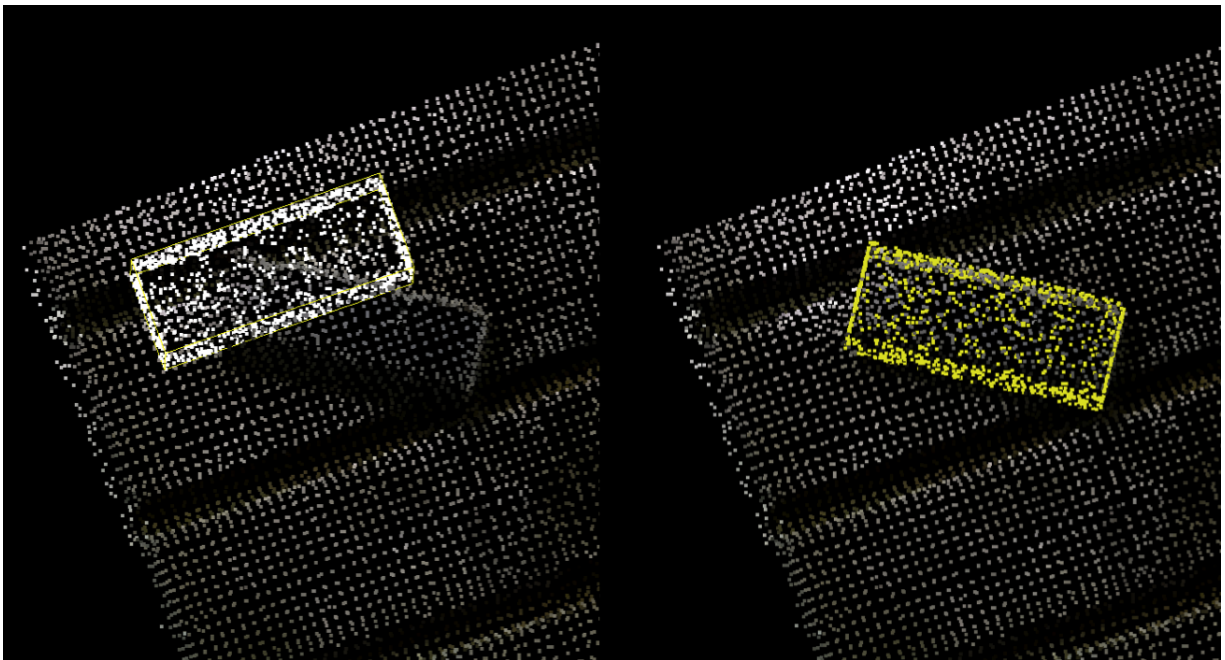


Figure 4.6: Object alignment by Iterative Closest Point (ICP). The left side shows the acquired point cloud and the point cloud representation of the CAD model. The right side shows how ICP aligns the CAD model to the acquired data.

The cross-platform communication interface *OpenShowVar* is an open-source alternative that makes it possible to read and write to all global variables of the KR C4 robot controller. The interface is based on a server-client architecture and communicates over *TCP/IP*. *OpenShowVar* is the client side and the server side is *KukaVarProxy*, a software application installed in the windows backbone of the KR C4 controller. *KukaVarProxy* can serve up to 10 clients simultaneously [45] and is implemented through *KUKA CrossCom*, an internal software interface of the robot controller.

*OpenShowVar* is designed as a standalone GUI application. In order to implement it in the C++ application developed in this project, it became necessary to completely rewrite it as a pure communication client. The result is a simplified and platform independent C++ client for *KukaVarProxy* based on the *Boost C++ Library* [46]. The *Boost* library is also one of the PCL dependencies. The client is written as a simple C++ header file, "*BoostClientCross.h*", and is inspired by a corresponding program written in Java, *JOpenShowVar* [45]. Source code for the client is provided in Appendix A.

The communication client allows for interaction with the realtime control process of the robot through *KukaVarProxy* and *KUKA CrossCom*. Operations that can be performed includes selection or cancellation of a specific program, renaming program files, detecting errors and faults, saving programs, resetting I/O drivers, and reading and writing variables [45]. The latter operation is the primary application area for this project. Access to a variable is obtained by specifying whether to make a read or write operation, the variable name, and optionally any value to be written to the variable.

In this project, the communication is based on updating user-defined global variables in the robot controller. The variables are defined with data type *REAL*, an approximation of a real number [44]. A separate variable is defined for each transformation parameter.

### **4.2.8 Fronius TransSteel to Robot Interface**

PhD candidate Lars Tingelstad and the undersigned have established the communication between the KR C4 robot controller and the *Fronius TransSteel 5000* welding machine. The Ethernet-based fieldbus system *EtherCAT* from Beckhoff Automation is used.

The robot-to-welder interface allows for programming of the welding machine directly in the robot program sequence. Welding parameters such as welding current, torch speed, weaving patterns, and pre- and post-flow of the shielding gas can thus be set and adjusted from the robot programming system.



# Chapter 5

## Results

The system has been evaluated by performing welding operations at various component positions and orientations. The first step was to test two simple welding operations. A *fillet weld* was performed in both horizontal and vertical position for various poses and transformation estimations. The next step was to conduct a case study where the system is used for welding together parts of a thruster tunnel.

### 5.1 Simple Welding Joints

To evaluate the potential of the system in a manner that facilitates good comparisons, a series of simple welding tests were performed. The main objective was to test the capabilities of the object alignment process for ordinary welding operations.

The tests have been conducted for two different welding joints, *fillet weld* in horizontal position and *fillet weld* in vertical position. In Figure 5.1, the two welding paths are marked on a sample component. The horizontal welding joint is performed by following an edge through a 90 degree corner, and therefore runs in both directions of the  $XY$  plane. The vertical welding joint is carried out almost straight down along the  $Z$  axis.

In all tests, the performance have been evaluated by comparing the translated offline programmed welding path to an optimized welding path. The optimized path was made by manual correc-

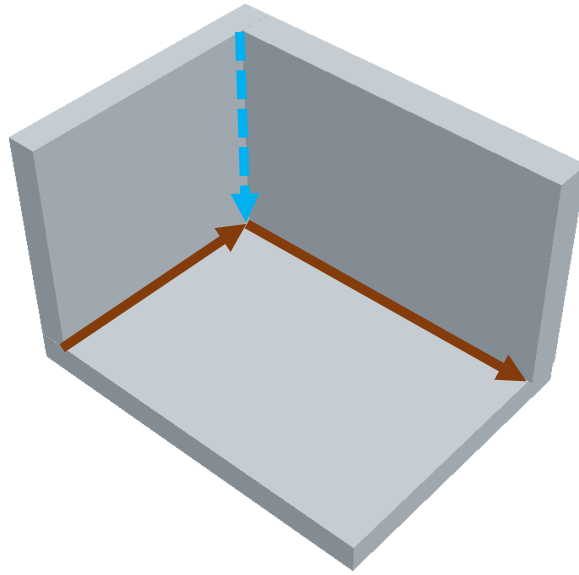


Figure 5.1: The simple welding joints used for testing system performance. *Fillet weld* in horizontal position is illustrated by brown arrows, *fillet weld* in vertical position by a blue arrow.

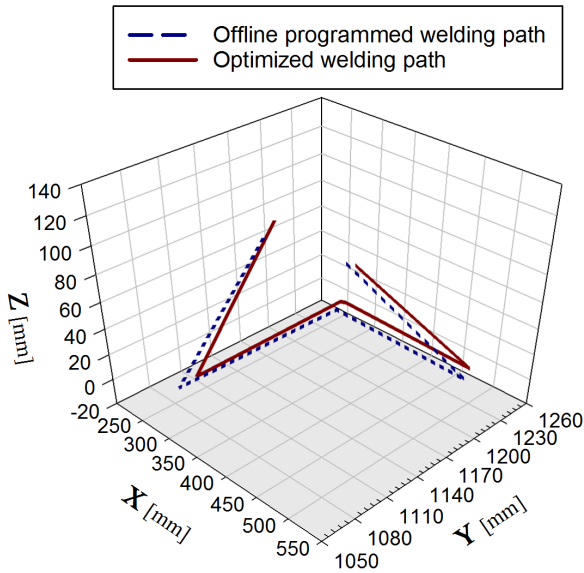
tion of the programmed welding paths. Thus, the measured deviations are found by using the robot end-effector, i.e., the tip of the welding electrode.

### 5.1.1 Case I: No Object Alignment

As a reference for the comparisons, the system was first tested without corrections from the Kinect camera. This corresponds to normal offline robot programming, where welding sequences are programmed based on pre-measured or assumed values for the component location and orientation.

The results achieved from this method were for the most part an unfinished welding program. The precision obtained were not good enough to perform welding directly, it was necessary to adjust the program by updating the programmed points. Programmed and adjusted robot trajectories for the two welding sequences are shown in Figure 5.2.

### Fillet Weld in Horizontal Position



### Fillet Weld in Vertical Position

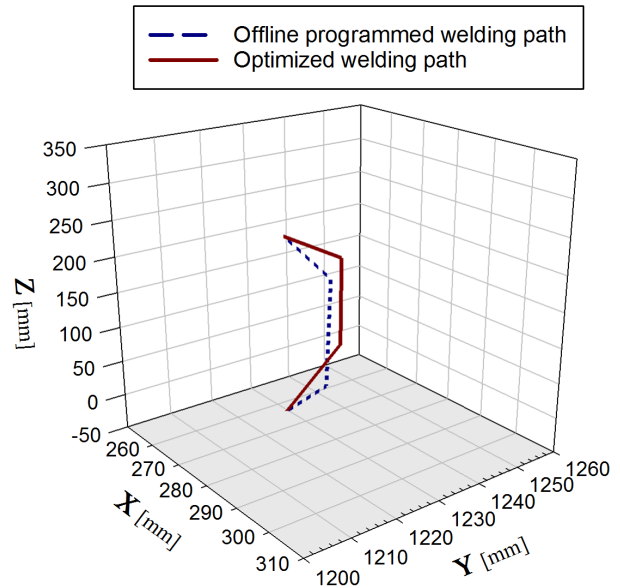


Figure 5.2: The offline programmed welding paths achieved when corrections from the 3D camera are not used.

#### 5.1.2 Case II: 2D Object Alignment

The first test including object alignment was performed with limitations in the estimation of the object pose. The Kinect camera is placed over the worktable with its field of view oriented towards the table. Because the camera  $Z$  axis is perpendicular to the table surface, a simple 2D rigid transformation  $(X, Y, R_z)$  could be estimated as a first step.

When comparing the reference test without corrections with the results from the 2D object alignment, the latter showed clearly improved welding paths. As shown in Figure 5.3, the adjusted trajectories are close to those of the desired welding paths. The accuracy in  $Z$  direction is equivalent of that achieved in the reference test. Correcting the vertical position of the welding joints are thus necessary.

This limited test method proved to be a good basis for testing and adjusting the point cloud processing algorithms, and to test the camera calibration.

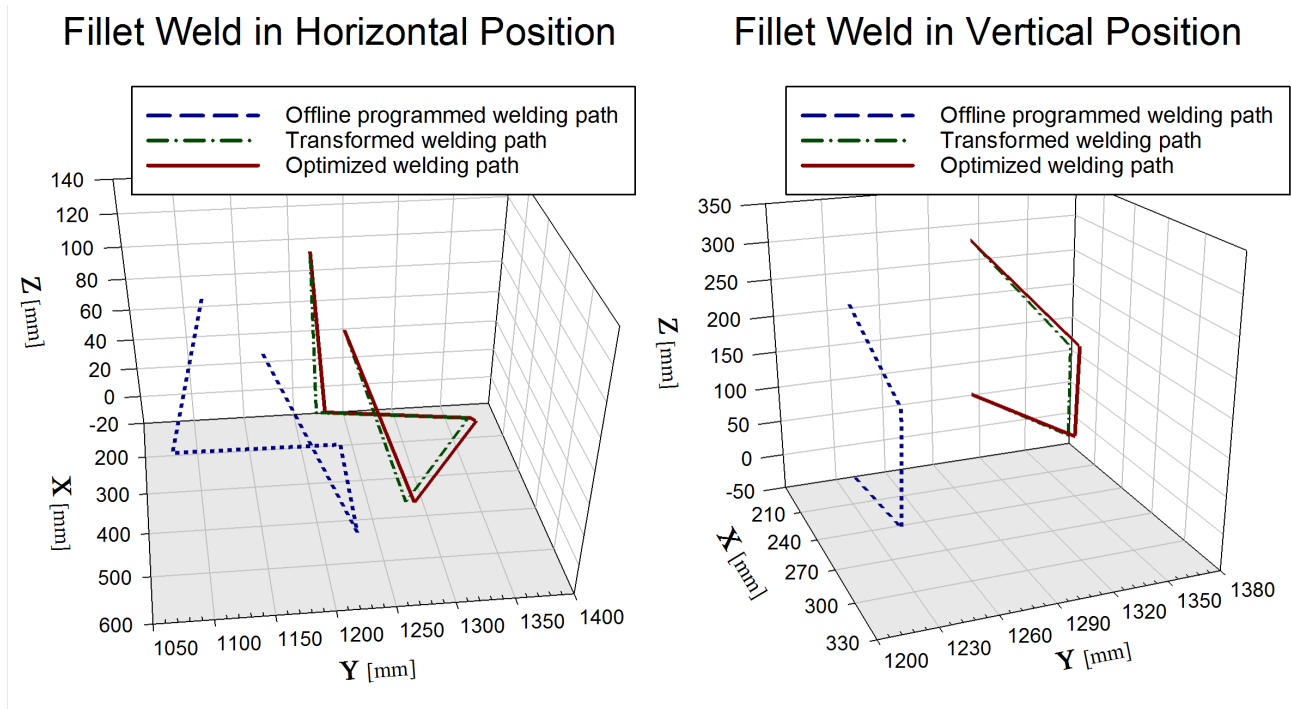


Figure 5.3: The offline programmed, transformed, and optimized welding paths achieved by using only 2D ( $X, Y, R_z$ ) corrections.

### 5.1.3 Case III: 3D Object Alignment

In this case, a full 3D transformation ( $X, Y, Z, R_x, R_y, R_z$ ) is estimated. The offline programmed and resulting transformed welding paths are illustrated in Figure 5.4. The transformation estimations are in this case close to the optimized welding paths, although some deviations can be observed.

Compared to the 2D object alignment, the full 3D alignment shows slightly inferior results for estimations in  $X$  and  $Y$  directions. However, the full 3D alignment performs much better in  $Z$  direction and for rotations.

### 5.1.4 Case IV: 3D Object Alignment from Several Point Clouds

It was observed that the object alignment differed slightly for consecutive estimations of stationary objects. A solution based on using the pointwise median value of several point clouds was attempted in order to stabilize the results. A good result which did not take too long were



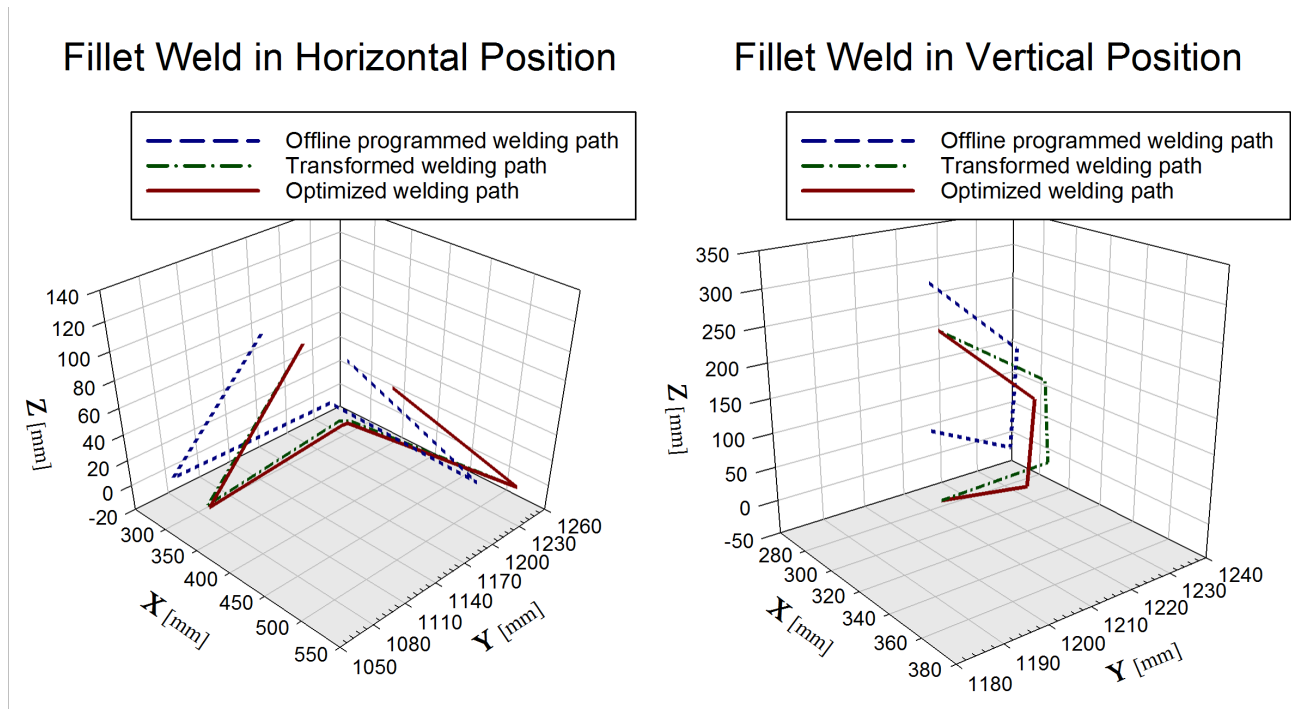


Figure 5.4: The offline programmed, transformed, and optimized welding paths achieved by using 3D ( $X, Y, Z, R_x, R_y, R_z$ ) corrections.

observed by combining from 7 to 12 point clouds. This solution proved to be relative efficient, but requires approximately 2 - 5 seconds longer than what was observed in Case III.

The offline programmed and resulting transformed welding paths for this method are illustrated in Figure 5.5. As in the previous case, a full 3D transformation ( $X, Y, Z, R_x, R_y, R_z$ ) is estimated.

Compared to the previous object alignments, this solution performs similar to the 2D object alignment for estimations in  $X$  and  $Y$  directions. It is also slightly better than the 3D object alignment in Case III for estimations in  $Z$  direction and for rotations. Of the achieved transformation estimations, the results from this solution are closest to the optimized welding paths.

## 5.2 System Performance

In order to better reflect the quality of the results, the absolute errors for the simple welding joints in Case I to IV (Section 5.1) are shown in Figure 5.6. For the final solution in Case IV, a mean absolute error of approximately 2.43 mm with a maximum of approximately 5.70 mm was

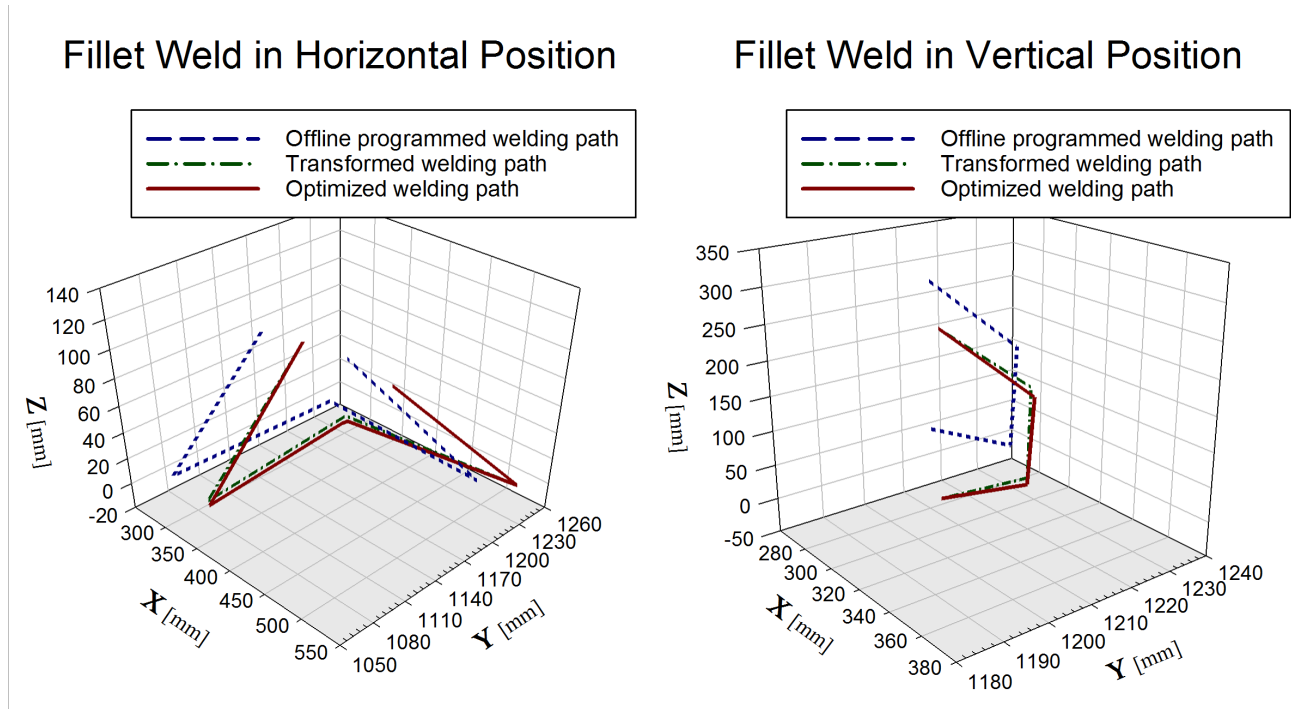


Figure 5.5: The offline programmed, transformed, and optimized welding paths achieved by using 3D corrections ( $X, Y, Z, R_x, R_y, R_z$ ) from 10 point clouds.

achieved. This is not ideal, but it is an acceptable deviation for many welding applications and promising for future work.

The objects to be aligned in this setup are located on a work table, a surface which is relatively straight and even along the  $X$  and  $Y$  directions relative to the camera coordinate system. Consequently, it has for common object poses not been estimated significant rotation of objects around the  $X$  and  $Y$  axes ( $R_x, R_y$ ). Various object poses with such rotations have been tested, the results corresponded to transformation estimates for object poses without said rotation. An example of a transformation estimate of an object rotated about all axes are shown in Figure 5.7.

It can sometimes take a while to grab point clouds from the stream. This is probably because of the limited computing power in this setup. The Kinect camera generates vast data streams, and the applied computer fulfills the minimum hardware requirements only by a small margin.

The implemented communication interface is fast. Tests have shown that the robot controller variables is updated in a few milliseconds. Updating 6 variables ( $X, Y, Z, R_x, R_y, R_z$ ) usually takes less than 20 ms. By using variable arrays, this process could probably be performed even faster.

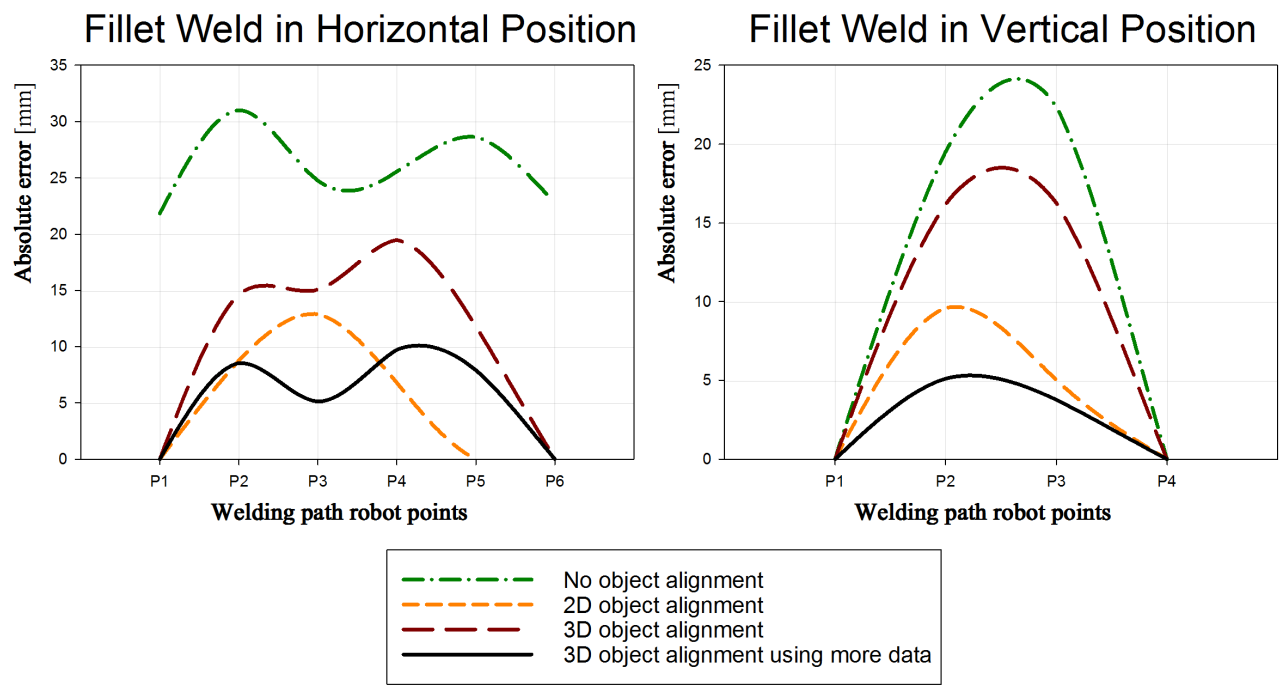


Figure 5.6: Absolute error observed for the simple welding joint tests. The errors were found by comparing the offline programmed and transformed welding paths to an optimized welding path. The optimized path was made by manual correction of the programmed welding paths.

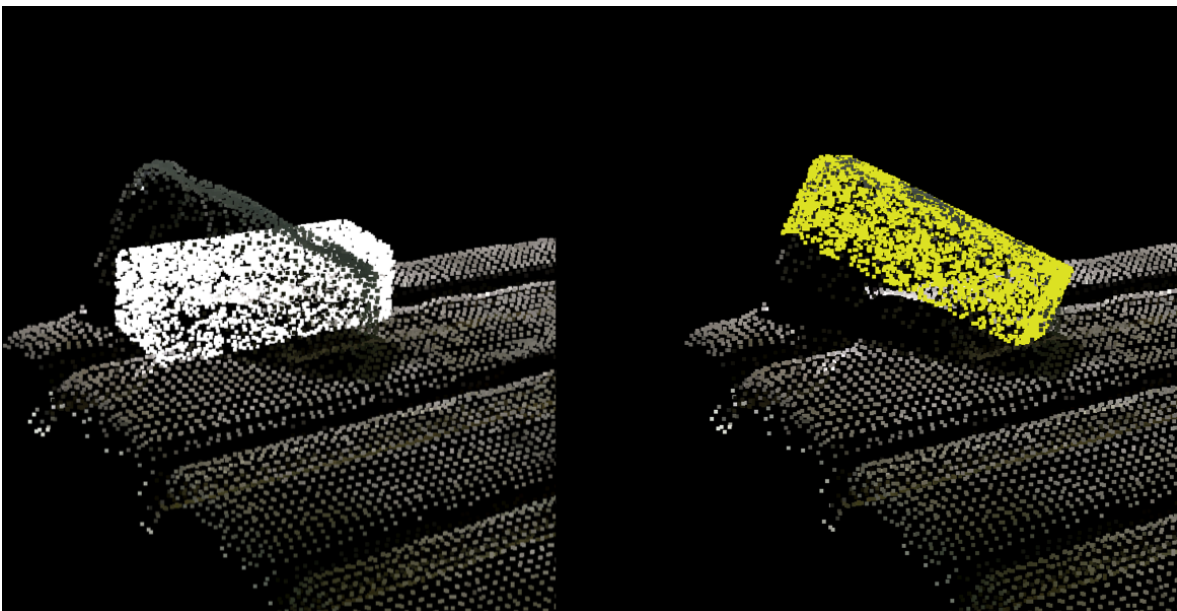


Figure 5.7: Object alignment of component rotated about  $X$ ,  $Y$ , and  $Z$  axes ( $R_x, R_y, R_z$ ). The left image shows the initial model placement (white) and point cloud data from the Kinect. In the right image, a successful object alignment has been performed and the resulting pose is shown in yellow.

It is thus a promising interface for future use in a more real-time system.

### 5.3 Thruster Tunnel Welding

The demonstrator setup has also been tested under conditions similar to what is found in industrial applications. Robotic welding is an important process in the fabrication of components for the shipbuilding industry. By using a section of a thruster tunnel, the system performance has been examined for manufacturing actual ship components. The relation between the complete thruster tunnel and extracted section is shown in Figure 5.8. The section consists of three steel plates perpendicular to one another.

The thruster section was joined together by tack welding before it was fully welded. All welds were programmed offline and performed by the robot. The component pose was changed both before and after tack welding, with the new transformation estimated for each pose.

The experiments performed in the demonstrator have shown that the system functionality is as expected;

- the point cloud processing finds adequate representations of the object of interest
- the 3D object alignment procedure calculates an updated object pose
- the updated pose is sent to the robot control system
- the robot follows the joints of the object accordingly, and with adequate accuracy.

The final result of the welding process is shown in Figure 5.9. The tack welding was performed with a welding current of about 160 ampere. When fully welding the component, the current was about 230 ampere.

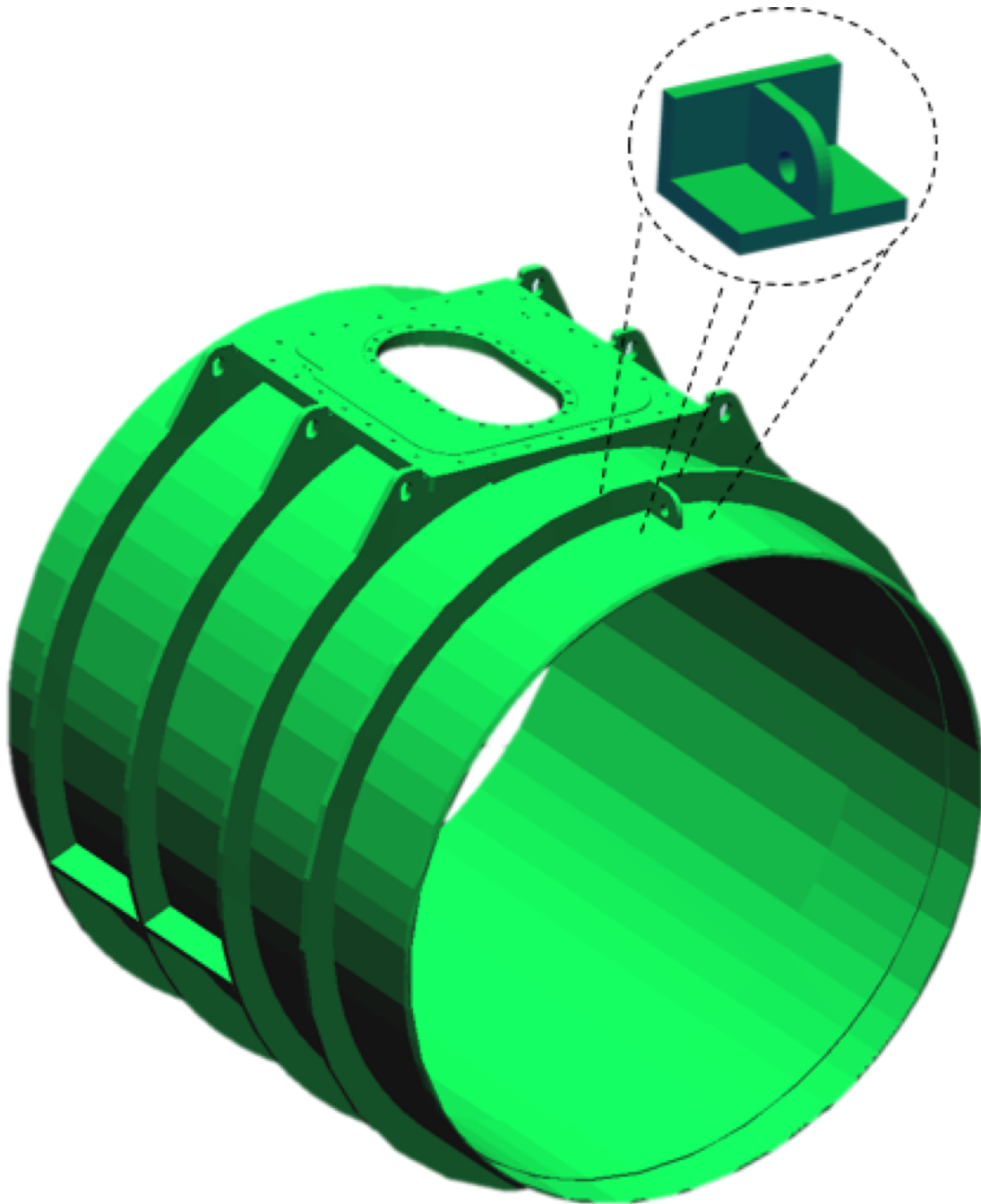


Figure 5.8: A model of the thruster tunnel. The tunnel section used for system testing is shown in a magnified view.



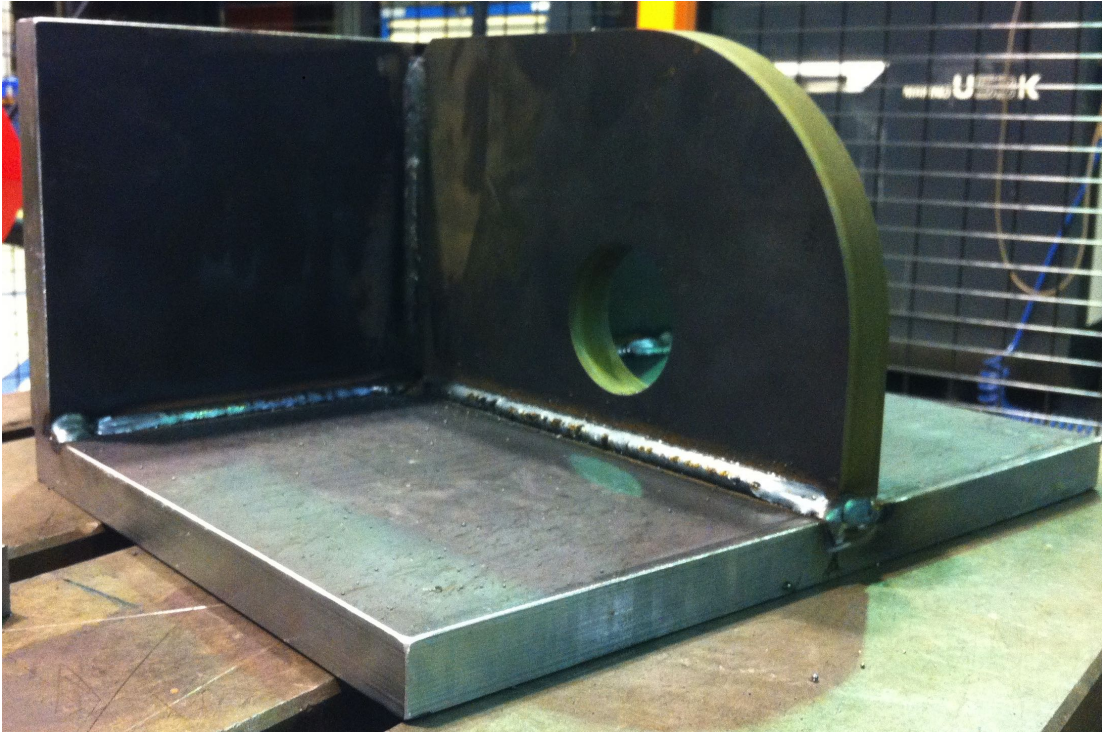


Figure 5.9: Thruster tunnel section after welding. The welding parameters applied (Current, waving parameters) are not optimal, but the precision of the welding paths is acceptable.

## 5.4 Video

A video showing the object alignment and welding of the thruster tunnel section has been produced and placed in the digital appendix. The video begins by explaining the physical setup in the robot cell. The following section shows how the object aligning algorithm performs for various scenarios. This part is a screen recording from the computer, and it displays the graphical interface of the developed C++ application.

Part three of the video shows how the offline robot programming and welding simulations is carried out. This is also a screen recording of a computer, but this time *KUKA.Sim* and *KUKA.OfficeLite* are demonstrated.

The final part of the video is a recording of the final welding process. Individual parts of the thruster tunnel are first assembled by spot welding. The component pose is then changed before the thruster tunnel part is fully welded.

# Chapter 6

## Concluding Remarks

### 6.1 Discussion

During the work in this project, it has been noted that planning of experiments ahead in time had a valuable impact on the flow of the experiments that were carried out. Some of the problems which came up during the experiments were of such a nature that we were unable to affect in advance, and what were initially thought to be small obstacles has often turned out to need more attention and effort than planned. Some of the noted difficulties were related to the delivery of necessary equipment for the welding machinery. In addition, some problems were encountered during the image processing stage. The Kinect camera generates vast data streams, and deciding on processing steps that make the data manageable while preserving the essential information turned out to be time consuming.

Starting out with a barely used robot manipulator and an unused welding machine, and ending up with a working robotic welding system, has been an interesting journey. The objective of creating a working system was always the main focus, and has taken up most of the time. The lack of background knowledge in necessary concepts and software like Microsoft Visual Studio, Point Cloud Library, C++ and Point Clouds has been a challenge from the start-up. The first part of the project was almost exclusively spent getting to know the software and programming interfaces.

Implementing the client-server communication between the robot controller and the developed point cloud acquisition and object alignment application turned out to be a major task. Several alternatives were considered and examined before OpenShowVar was chosen, which had to be rewritten in C++ in order to be implemented. With more background knowledge in C++ programming, this would probably not be such a demanding task to perform.

Perhaps the biggest advantage of using a 3D camera and CAD model for correcting robot programs is the versatility. This type of system can very easily be reprogrammed for new welding operations, or it can even be used for other industrial processes such as material handling (pick and place).

If the developed system were to be used for welding large and complex components, it could be necessary to estimate the object pose several times throughout the welding operation. The object alignment algorithm is in its current form probably not performing fast enough for use in such a near real-time system. Some solutions have been proposed, and with the adequate time it would be very interesting to implement these in order to achieve higher speed of the pose estimation. A faster system could potentially also be used for visual servoing and other autonomous systems.

The comprehensive practical part has seized most of the available time. The theory section has consequently been limited to some background information and explaining the work performed. In addition to the objectives, a brief comparison between various object aligning algorithms has been performed. With the initial problem formulations fulfilled, the undersigned believes that the project is successfully completed.



## 6.2 Conclusion

In this thesis, a 3D computer vision solution was developed and used for improving the process of offline programming a welding robot. The results were demonstrated by programming and welding a section of a thruster tunnel for ships with corrections from a consumer grade 3D camera. The system is able to detect and correct the welding paths for both translated and rotated objects.

The results show small variations in the corrected object pose estimation. This appeared to originate from random variations in the depth data from the Kinect 3D camera, and led to inaccurate object pose estimations which were not very robust. A solution based on using the point-wise median value of several point clouds for estimating the object pose proved to be relative efficient. For this solution, a mean absolute error of approximately 2.43 mm with a maximum of approximately 5.70 mm was achieved. Whether this is a sufficiently accurate solution will differ from operation to operation, but for the tasks studied in this project it resulted in adequate welds.

The speed of the object pose estimation is not very high, but it is believed to be more than adequate for correcting offline programmed welding paths. Compared to manual correction of the welding paths, the amount of time saved is noticeable and could facilitate single piece manufacturing. It is possible to further improve the speed of the pose estimation, and potential solutions have been proposed.

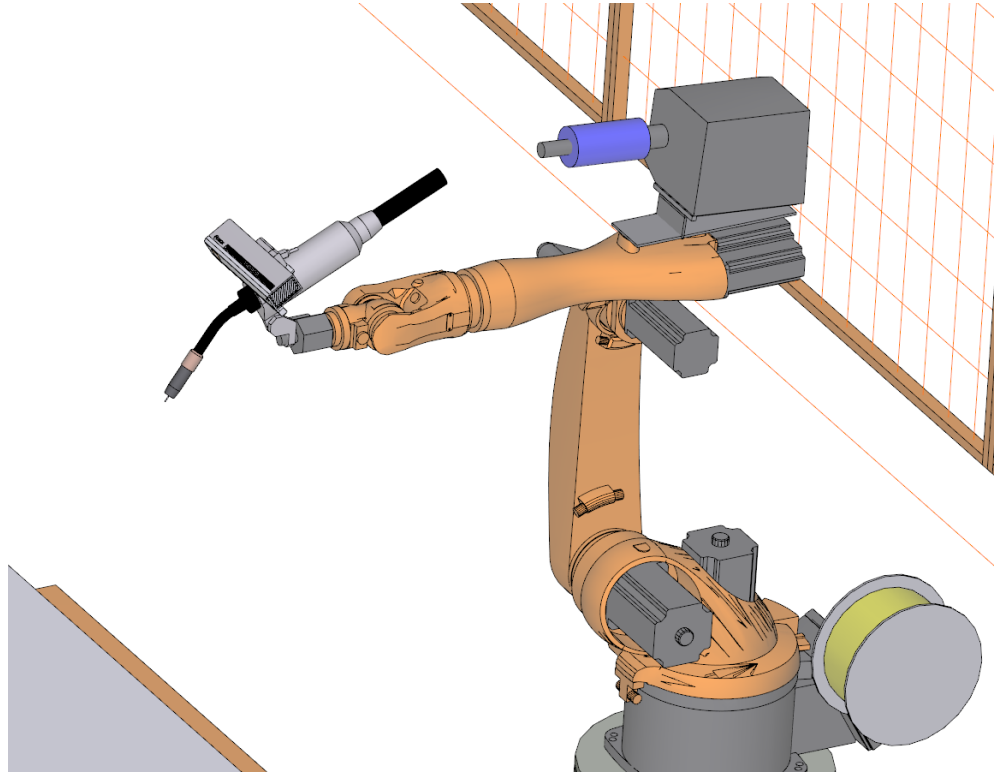


Figure 6.1: Sketch illustrating how a Kinect 3D camera could be mounted on a KR5 welding robot manipulator.

### 6.3 Recommendations for Further Work

It is yet to be tested how the object pose estimation performs if the 3D camera view is partially blocked, e.g. by the robot arm. Different camera positions and orientations should also be tested in order to evaluate the system robustness. Mounting the Kinect 3D camera on the robot arm itself or on a dedicated robot manipulator would allow for estimating object pose from different point of views, and would make it possible to perform welding on significantly larger objects. A graphical illustration of how this system might look is given in Figure 6.1.

In order to improve the system without updating the hardware, it is necessary to reduce the search area of the object aligning algorithms. A possible solution could be to use segmentation methods for estimating the region of interest. A segmentation algorithm based on colorimetric similarity and spatial proximity proposed by Zhan et al [47] could be useful in this work.

The CAD model is in the current system converted to a point cloud directly, with all visible and invisible surfaces represented by points. When comparing the point cloud acquired from the 3D

camera with the point cloud converted from the CAD model, the points converted from model features invisible to the camera can contribute to degrading the object pose estimation. A better way to represent the model would be to only convert the surfaces visible from the 3D camera's point of view. This could be achieved by using the initial guess for object pose and a simulated camera, and then sample points on the CAD model by ray tracing.

Physical components are rarely a proportionally perfect version of the CAD model it originates from. Deformations, lengths and angle deviations should be taken into account in the object pose estimations. A possible solution could be to divide the object into smaller parts, and estimate poses at a lower level. Introducing object scale as a factor to estimate would also be beneficial.



# Bibliography

- [1] I. Karabegović, E. Karabegović, S. Pašić, and S. Isić, “Worldwide trend of the industrial robot applications in the welding processes,” *International Journal of Engineering and Technology*, vol. 12, no. 01, 2012.
- [2] R. D. Schraft and C. Meyer, “The need for an intuitive teaching method for small and medium enterprises,” in *2006 ISR Robotik*, (Munich, Germany), May 2006.
- [3] B. Akan, *Human Robot Interaction Solutions for Intuitive Industrial Robot Programming*. Västerås: Mälardalen University, 2012.
- [4] E. Trucco and A. Verri, *Introductory techniques for 3-D computer vision*, vol. 201. Prentice Hall Englewood Cliffs, 1998.
- [5] D. A. Forsyth and J. Ponce, *Computer Vision: A Modern Approach 2nd Ed.* Pearson Education, Inc., 2 ed., 2012.
- [6] P. Corke, *Robotics, vision and control: fundamental algorithms in MATLAB*, vol. 73. Springer Science & Business Media, 2011.
- [7] J. N. Pires, A. Loureiro, and G. Böllmsjo, *Welding robots: technology, system issues and application*. Springer Science & Business Media, 2006.
- [8] W. J. Savitch, *Absolute C++*. Pearson Education, 2006.
- [9] P. Neto and N. Mendes, “Direct off-line robot programming via a common cad package.,” *Robotics and Autonomous Systems*, vol. 61, no. 8, pp. 896–910, 2013.

- [10] OSHA, *OSHA directive of 1987, Part II-E*. Occupational Safety and Health Administration, OSHA, 1987.
- [11] Z. Pan, J. Polden, N. Larkin, S. V. Duin, and J. Norrish, "Recent progress on programming methods for industrial robots," in *Robotics (ISR), 2010 41st International Symposium on and 2010 6th German Conference on Robotics (ROBOTIK)*, June 2010.
- [12] Autodesk, *Autodesk AutoCAD*. Autodesk, 1982-2015.
- [13] R. B. Rusu and S. Cousins, "3d is here: Point cloud library (pcl).," in *ICRA*, IEEE, 2011.
- [14] L. Li, "Time-of-flight camera—an introduction," *Technical White Paper*, May, 2014.
- [15] F. Brunet, "Contributions to parametric image registration and 3d surface reconstruction," *University of Auvergne*, 2010.
- [16] C. Dal Mutto, P. Zanuttigh, and G. M. Cortelazzo, *Time-of-flight cameras and microsoft Kinect™*. Springer Science & Business Media, 2012.
- [17] A. Payne, A. Daniel, A. Mehta, B. Thompson, C. S. Bamji, D. Snow, H. Oshima, L. Prather, M. Fenton, L. Kordus, *et al.*, "7.6 a 512 × 424 cmos 3d time-of-flight image sensor with multi-frequency photo-demodulation up to 130mhz and 2gs/s adc," in *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2014 IEEE International*, pp. 134–135, IEEE, 2014.
- [18] V. Castaneda and N. Navab, "Time-of-flight and kinect imaging," *Kinect Programming for Computer Vision*, 2011.
- [19] Microsoft, *Kinect for Windows SDK*. Microsoft, October 2014.
- [20] J.-Y. Bouguet, "Camera calibration toolbox for matlab," 2004.
- [21] Z. Zhang, "Flexible camera calibration by viewing a plane from unknown orientations," in *Computer Vision, 1999. The Proceedings of the Seventh IEEE International Conference on*, vol. 1, pp. 666–673, IEEE, 1999.
- [22] D. C. Brown, "Decentering distortion of lenses," *Photometric Engineering*, vol. 32, no. 3, pp. 444–462, 1966.

- [23] J. S. Vitter, "Faster methods for random sampling," *Communications of the ACM*, vol. 27, no. 7, pp. 703–718, 1984.
- [24] M. Alexa, J. Behr, D. Cohen-Or, S. Fleishman, D. Levin, and C. T. Silva, "Computing and rendering point set surfaces," *Visualization and Computer Graphics, IEEE Transactions on*, vol. 9, no. 1, pp. 3–15, 2003.
- [25] P. Lancaster and K. Salkauskas, "Surfaces generated by moving least squares methods," *Mathematics of computation*, vol. 37, no. 155, pp. 141–158, 1981.
- [26] R. B. Rusu, "Semantic 3d object maps for everyday manipulation in human living environments," *KI-Künstliche Intelligenz*, vol. 24, no. 4, pp. 345–348, 2010.
- [27] P. Lancaster and K. Salkauskas, *Curve and surface fitting*. Academic press, 1986.
- [28] N. J. Mitra, A. Nguyen, and L. Guibas, "Estimating surface normals in noisy point cloud data," *International Journal of Computational Geometry & Applications*, vol. 14, no. 04n05, pp. 261–276, 2004.
- [29] J. Berkmann and T. Caelli, "Computation of surface geometry and segmentation using covariance techniques," *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, vol. 16, no. 11, pp. 1114–1116, 1994.
- [30] R. B. Rusu, N. Blodow, and M. Beetz, "Fast point feature histograms (fpfh) for 3d registration," in *Robotics and Automation, 2009. ICRA'09. IEEE International Conference on*, pp. 3212–3217, IEEE, 2009.
- [31] M. Levoy, J. Gerth, B. Curless, and K. Pull, "The stanford 3d scanning repository," [www.graphics.stanford.edu/data/3dscanrep](http://www.graphics.stanford.edu/data/3dscanrep), 2005.
- [32] L. Ding, Y. Peng, C. Shen, and Z. Hu, "Affine registration for multidimensional point sets under the framework of lie group," *Journal of Electronic Imaging*, vol. 22, no. 1, pp. 013022–013022, 2013.

- [33] Y. Chen and G. Medioni, "Object modeling by registration of multiple range images," in *Robotics and Automation, 1991. Proceedings., 1991 IEEE International Conference*, pp. 2724–2729, IEEE, 1991.
- [34] P. J. Besl and N. D. McKay, "Method for registration of 3-d shapes," in *Robotics-DL tentative*, pp. 586–606, International Society for Optics and Photonics, 1992.
- [35] S. Rusinkiewicz and M. Levoy, "Efficient variants of the icp algorithm," in *3-D Digital Imaging and Modeling, 2001. Proceedings. Third International Conference on*, pp. 145–152, IEEE, 2001.
- [36] A. Nuchter, K. Lingemann, and J. Hertzberg, "Extracting drivable surfaces in outdoor 6d slam," *VDI BERICHTE*, vol. 1956, p. 189, 2006.
- [37] Z. Zhang, "Iterative point matching for registration of free-form curves and surfaces," *International journal of computer vision*, vol. 13, no. 2, pp. 119–152, 1994.
- [38] K. S. Arun, T. S. Huang, and S. D. Blostein, "Least-squares fitting of two 3-d point sets," *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, no. 5, pp. 698–700, 1987.
- [39] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery, "Numerical recipes in c: the art of scientific computing," *Cambridge University Press, Cambridge, MA*, vol. 131, pp. 243–262, 1992.
- [40] M. A. Fischler and R. C. Bolles, "Random sample consensus: a paradigm for model fitting with applications to image analysis and automated cartography," *Communications of the ACM*, vol. 24, no. 6, pp. 381–395, 1981.
- [41] F. GMBH, *Fronius TransSteel Robotics*. Fronius GMBH, 2015.
- [42] W. J. Schroeder, B. Lorensen, and K. Martin, *The visualization toolkit*. Kitware, 2004.
- [43] C. L. CHUA, K. LIM, and C. R. Prototyping, *Principles and Applications*. World Scientific, 2003.
- [44] K. R. GmbH, *KUKA System Software 8.2*. KUKA Roboter GmbH, Zugspitzstraße 140, D-86165 Augsburg, Germany, kss 8.2 si v4 en ed., July 2012.



- [45] F. Sanfilippo, L. I. Hatledal, H. Zhang, M. Fago, and K. Y. Pettersen, “Jopenshowvar: an open-source cross-platform communication interface to kuka robots,” in *Proc. of the IEEE International Conference on Information and Automation (ICIA), Hailar, China*, pp. 1154–1159, 2014.
- [46] Boost, “Boost c++ libraries,” <http://www.boost.org>, 2015.
- [47] Q. Zhan, Y. Liang, and Y. Xiao, “Color-based segmentation of point clouds,” *Int Arch Photogrammetry, Remote Sens Spat Inf Sci*, vol. 38, pp. 248–252, 2009.



# **Appendix A**

## **Source Code**

## A.1 Main C++ application

Listing A.1: Source code of main.cpp.

```
1  /*
2  Source code for Point cloud capturing, CAD-to-Point Cloud conversion,
3  object pose estimation, and communication with KUKA robot controller.
4
5  Author: Eirik B. Njaastad.
6  NTNU 2015
7  */
8  #include "stdafx.h"
9  #include <ctime>
10 #include <iostream>
11 #include <fstream>
12 #include "BoostClientCross.h" // For communication with the KUKA robot controller
13 #include "kinect2_grabber.h" // Grabber for the Kinect, made by Tsukasa SUGIURA (MIT License)
14 #include <boost/interprocess/sync/scoped_lock.hpp>
15 #include <Eigen/Core>
16 #include <pcl/io/pcd_io.h>
17 #include <pcl/io/vtk_lib_io.h>
18 #include <pcl/point_types.h>
19 #include <pcl/common/common.h>
20 #include <pcl/filters/passthrough.h>
21 #include <pcl/filters/voxel_grid.h>
22 #include <pcl/filters/median_filter.h>
23 #include <pcl/filters/statistical_outlier_removal.h>
24 #include <pcl/filters/extract_indices.h>
25 #include <pcl/features/fpfh.h>
26 #include <pcl/features/normal_3d.h>
27 #include <pcl/features/vfh.h>
28 #include <pcl/features/normal_3d_omp.h>
29 #include <pcl/features/fpfh_omp.h>
30 #include <pcl/kdtree/kdtree_flann.h>
31 #include <pcl/octree/octree.h>
32 #include <pcl/registration/icp.h>
33 #include <pcl/registration/icp_nl.h>
34 #include <pcl/registration/transformation_estimation_2D.h>
35 #include <pcl/registration/ia_ransac.h>
36 #include <pcl/surface/mls.h>
37 #include <pcl/surface/vtk_smoothing/vtk_utils.h>
38 #include <pcl/visualization/pcl_visualizer.h>
39 #include <pcl/visualization/histogram_visualizer.h>
40 #include <vtkTriangle.h>
```

```
41
42 // Conversion from radians to degrees:
43 #define RADTODEG(rad) (rad * (180/M_PI))
44
45 using namespace std;
46 using namespace pcl;
47
48 typedef PointXYZRGB PointT;
49 typedef PointCloud<PointT> PointCloudT;
50 typedef pcl::PointCloud<pcl::Normal> SurfaceNormalsT;
51 typedef pcl::search::KdTree<pcl::PointXYZRGB> SearchMethodT;
52
53 // Booleans for keeping track of what has happened and not:
54 bool saveTargetCloud(false);
55 bool loadCADmodel(false);
56 bool runICP(false);
57 bool ICPRunning(false);
58 bool targetCloudPresent(false);
59 bool socketConnected(false);
60 bool commRob(false);
61 bool firstRun(true);
62 bool cloudNormalsPresent(false);
63
64 // Settings for the robot controller communication:
65 BoostClientCross boostclientcross;
66 string ip = "192.168.251.5";
67 string port = "7000";
68
69 // Name of variables in the robot controller to update:
70 vector<unsigned char> translX = { 'T', 'R', 'A', 'N', 'S', 'L', 'X' };
71 vector<unsigned char> translY = { 'T', 'R', 'A', 'N', 'S', 'L', 'Y' };
72 vector<unsigned char> translZ = { 'T', 'R', 'A', 'N', 'S', 'L', 'Z' };
73 vector<unsigned char> rotateA = { 'R', 'O', 'T', 'A', 'T', 'E', 'A' };
74 vector<unsigned char> rotateB = { 'T', 'R', 'A', 'N', 'S', 'L', 'B' };
75 vector<unsigned char> rotateC = { 'T', 'R', 'A', 'N', 'S', 'L', 'C' };
76
77 // Name of the CAD model to import:
78 string cadName = "CADmodel.stl";
79 PolygonMesh mesh;
80
81 // The two viewports in the ICP viewer
82 int v1(0);
83 int v2(1);
84
85 // Creating pointers for the point clouds:
```

```

86 PointCloudT::Ptr cloud_model(new PointCloudT);
87 PointCloudT::Ptr cloud_target(new PointCloudT);
88 PointCloudT::Ptr cloud_ICP(new PointCloudT);
89 // Temporary point clouds for filtering and transformations:
90 PointCloudT::Ptr downsamplingCloud(new PointCloudT);
91 PointCloudT::Ptr outlierRemovedCloud(new PointCloudT);
92 PointCloudT::Ptr cloud_transformed(new PointCloudT);
93 // Point Cloud for holding the CAD mesh:
94 PointCloud<PointXYZ>::Ptr pointCloudFromMESH(new PointCloud<PointXYZ>);
95 // Matrices for holding the camera calibration values and estimated transformation:
96 Eigen::Matrix4f calibrationMatrix = Eigen::Matrix4f::Identity();
97 Eigen::Matrix4f transformationMatrix = Eigen::Matrix4f::Identity();
98
99 class SimpleViewer{
100 private:
101     visualization::PCLVisualizer* KinectStreamVisualizer;
102     visualization::PCLVisualizer* ICPVisualizer;
103
104     boost::mutex cloud_mutex_;
105     PointCloudT::ConstPtr cloud_;
106     bool recieved_first_;
107
108 public:
109     void run();
110     void IterativeClosestPointfunction();
111     void OutlierRemovalFilter(const PointCloudT::ConstPtr& inCloud, int nNeighbors);
112     void DownsamplingFilter(const PointCloudT::ConstPtr& inCloud, const float voxel_grid_size);
113     void cloudCB(const PointCloudT::ConstPtr& cloud);
114     void updatePointCloud(const PointCloudT::ConstPtr& cloud);
115     void print4x4Matrix(const Eigen::Matrix4d & matrix);
116 };
117
118 // Function for estimating the surface normals for points:
119 PointCloud<PointNormal>::Ptr estimateSurfaceNormals(const PointCloud<PointNormal>::ConstPtr& input_cloud, float
    radius)
120 {
121     PointCloud<PointNormal>::Ptr cloud_normals(new PointCloud<PointNormal>);
122     copyPointCloud(*input_cloud, *cloud_normals); // Make a copy to work on
123
124     NormalEstimation<PointNormal, PointNormal> normal_estimation;
125
126     // Create an empty kdtree representation, and pass it to the normal estimation object.
127     // Its content will be filled inside the object, based on the given input dataset.
128     pcl::search::KdTree<PointNormal>::Ptr tree(new pcl::search::KdTree<PointNormal>());
129     normal_estimation.setSearchMethod(tree);

```

```

130     // Set search radius for neighboring points:
131     normal_estimation.setRadiusSearch(radius);
132     normal_estimation.setInputCloud(input_cloud);
133     // Computing the normals:
134     normal_estimation.compute(*cloud_normals);
135     return (cloud_normals);
136 }
137
138 // Function performing the Iterative Closest Point algorithm
139 void SimpleViewer::IterativeClosestPointfunction() {
140
141     //
142     // This function contains 3 versions of ICP,
143     // only the nonlinear version is currently activated
144     //
145
146     boost::posix_time::seconds workTime(3);
147     ICPRunning = true;
148     string modelCloudname_ICP = "modelCloud_ICP";
149
150     // Optional 2D transformation estimation:
151     //registration::TransformationEstimation2D<PointT, PointT>::Ptr est;
152     //estNorm.reset(new registration::TransformationEstimation2D<PointT, PointT>);
153
154     //
155     // STANDARD ICP
156     //
157     /*
158     cout << "\n\nStarting standard ICP \n";
159     IterativeClosestPoint<PointT, PointT> icp;
160     // Optional 2D transformation estimation:
161     //icp.setTransformationEstimation(est);
162     icp.setInputSource(cloud_model);
163     icp.setInputTarget(cloud_target);
164     //icp.setEuclideanFitnessEpsilon(1e-8); // optional convergence criteria
165     icp.align(*cloud_model);
166
167     if (icp.hasConverged())
168     {
169         cout << "\nStandard ICP converged:" << icp.hasConverged() <<
170             " with the score: " << icp.getFitnessScore() << endl;
171         transformationMatrix = icp.getFinalTransformation().cast<double>();
172         SimpleViewer::print4x4Matrix(transformationMatrix);
173     }
174     else

```

```

175     {
176         PCL_ERROR("\nStandard ICP did NOT converge!");
177     }
178     // VISUALIZING STANDARD ICP:
179     visualization::PointCloudColorHandlerCustom<PointT> single_colorStandardICP(cloud_model, 200, 200, 0);
180     ICPVisualizer->updatePointCloud<PointT>(cloud_model, single_colorStandardICP, modelCloudname_ICP);
181     */
182
183     //
184     // NONLINEAR ICP
185     //
186
187     cout << "\n\n___Starting_NonLinear_ICP_\n";
188     IterativeClosestPointNonLinear<PointT, PointT> NLicp;
189     // Optional 2D transformation estimation:
190     //NLicp.setTransformationEstimation(est);
191     NLicp.setInputSource(cloud_model);
192     NLicp.setInputTarget(cloud_target);
193     NLicp.align(*cloud_model);
194
195     if (NLicp.hasConverged())
196     {
197         cout << "\nNonLinear_ICP_converged:" << NLicp.hasConverged() <<
198             "\n_with_the_score:_:" << NLicp.getFitnessScore() << endl;
199         transformationMatrix = NLicp.getFinalTransformation().cast<double>();
200         SimpleViewer::print4x4Matrix(transformationMatrix);
201     }
202     else
203     {
204         PCL_ERROR("\nNonLinear_ICP_did_NOT_converge!");
205     }
206     // Visualization of nonlinear ICP:
207     visualization::PointCloudColorHandlerCustom<PointT> single_colorNonlinearICP(cloud_model, 200, 200, 0);
208     ICPVisualizer->updatePointCloud<PointT>(cloud_model, single_colorNonlinearICP, modelCloudname_ICP);
209
210     //
211     // ICP WITH NORMALS
212     //
213     /*
214     cout << "\nEstimating target Surface Normals";
215     pcl::PointCloud<pcl::PointNormal>::Ptr normals_target;
216     pcl::PointCloud<pcl::PointNormal>::Ptr normals_model;
217     normals_target = estimateSurfaceNormals(cloud_targetICPwn, 0.005f);
218     cout << "\nEstimating model Surface Normals";
219     normals_model = estimateSurfaceNormals(cloud_modelICPwn, 0.005f);

```



```

220     // Visualizing the normals:
221     if (!cloudNormalsPresent){
222         ICPVisualizer->addPointCloudNormals<PointNormal>(normals_target, 10, 0.05, "normals", 1);
223         ICPVisualizer->addPointCloudNormals<PointNormal>(normals_model, 10, 0.05, "normals2", 1);
224         cloudNormalsPresent = true;
225     }
226     cout << "\n\nStarting ICP with normals\n";
227     IterativeClosestPointWithNormals<PointNormal, PointNormal> icpWN;
228     // Optional 2D transformation estimation:
229     //registration::TransformationEstimation2D<PointNormal, PointNormal>::Ptr estNorm;
230     //estNorm.reset(new registration::TransformationEstimation2D<PointNormal, PointNormal>);
231     //icpWN.setTransformationEstimation(estNorm);
232     icpWN.setInputSource(normals_model);
233     icpWN.setInputTarget(normals_target);
234     icpWN.align(*normals_model);
235     if (icpWN.hasConverged())
236     {
237         cout << "\nICP with normals converged:" << icpWN.hasConverged() <<
238             " with the score: " << icpWN.getFitnessScore() << endl;
239         transformationMatrix = icpWN.getFinalTransformation().cast<double>();
240         SimpleViewer::print4x4Matrix(transformationMatrix);
241     }
242     else
243     {
244         PCL_ERROR("\nICP with normals did NOT converge!");
245     }
246     // Visualization of ICP with normals:
247     visualization::PointCloudColorHandlerCustom<PointNormal> single_colorICPwn(normals_model, 0, 225, 0);
248     ICPVisualizer->updatePointCloud<PointNormal>(normals_model, single_colorICPwn, modelCloudname_ICP);
249     */
250     ICPRunning = false;
251     boost::this_thread::sleep(workTime);
252 }
253
254 // Filter for statistically removing outlier points:
255 void SimpleViewer::OutlierRemovalFilter(const PointCloudT::ConstPtr& inCloud, int nNeighbors){
256     StatisticalOutlierRemoval<PointT> sor; // Create the filtering object
257     sor.setInputCloud(inCloud);
258     sor.setMeanK(nNeighbors); // Sets the number of neighboring points
259     sor.setStddevMulThresh(0.15); // Sets a standard deviation threshold value
260     sor.filter(*outlierRemovedCloud);
261 }
262
263 // Filter for reducing (down-sampling) the number of points in a point cloud:
264 void SimpleViewer::DownsamplingFilter(const PointCloudT::ConstPtr& inCloud, const float voxel_grid_size){

```

```

265     pcl::VoxelGrid<PointT> vox_grid; // Using a voxel grid filter
266     vox_grid.setInputCloud(inCloud);
267     // The size of the voxels (Equal sided cubes in this case):
268     vox_grid.setLeafSize(voxel_grid_size, voxel_grid_size, voxel_grid_size);
269     PointCloudT::Ptr tempCloud(new PointCloudT);
270     vox_grid.filter(*tempCloud); // Performs the filtering
271     downsamplingCloud = tempCloud;
272 }
273
274 // This function runs continuously, updating the Kinect stream and controlling the other functions:
275 void SimpleViewer::updatePointCloud(const PointCloudT::ConstPtr& skyen) {
276
277     // Transforming the grabbed point cloud to the calibrated values:
278     pcl::transformPointCloud(*skyen, *cloud_transformed, calibrationMatrix);
279
280     // Setting names for point clouds added to the visualizer:
281     string streamCloudname = "streamCloud";
282     string modelCloudname = "modelCloud";
283     string modelCloudname_ICP = "modelCloud_ICP";
284     string targetCloudname = "targetCloud";
285     string targetCloudnameV2 = "targetCloudV2";
286
287     if (!recieved_first_) {
288         recieved_first_ = true;
289         return;
290     }
291
292     // Color handler for the current grabbed cloud:
293     visualization::PointCloudColorHandlerRGBField<PointT> rgb_s(cloud_transformed);
294     if (firstRun) {
295         // If first run, add the current grabbed cloud to the Kinect data stream visualizer:
296         KinectStreamVisualizer->addPointCloud<PointT>(cloud_transformed, rgb_s, streamCloudname, 0);
297         firstRun = false;
298     }
299
300     else {
301         // Update the current grabbed cloud in the Kinect data stream visualizer:
302         KinectStreamVisualizer->updatePointCloud<PointT>(cloud_transformed, rgb_s, streamCloudname);
303         // Saving the grabbed cloud as target cloud for the transformation estimation:
304         if (saveTargetCloud) {
305             *cloud_target = *cloud_transformed;
306
307             // Filtering the grabbed cloud:
308             SimpleViewer::DownsamplingFilter(cloud_target, 0.008f); //0.008
309             cout << "Voxel_Grid_filter_";

```

```

310         SimpleViewer::OutlierRemovalFilter(downsamplingCloud, 50);
311         cout << "Outliers_removal_";
312         SimpleViewer::DownsamplingFilter(outlierRemovedCloud, 0.004f); //0.005
313         cout << "Voxel_Grid_filter_";
314         *cloud_target = *downsamplingCloud;
315
316         // Moving least squares (MLS) filter:
317         pcl::search::KdTree<pcl::PointXYZRGB>::Ptr MLSTree(new pcl::search::KdTree<pcl::
            PointXYZRGB>); // Create a KD-Tree
318         PointCloud<PointXYZRGB> MLScld;
319         MLScld = *cloud_target;
320         // Init object (second point type is for the normals, even if unused)
321         pcl::MovingLeastSquares<pcl::PointXYZRGB, pcl::PointXYZRGB> mls;
322         // Set MLS parameters
323         mls.setInputCloud(cloud_target);
324         mls.setPolynomialFit(true);
325         mls.setSearchMethod(MLSTree);
326         mls.setSearchRadius(0.02);
327         // Reconstruct
328         mls.process(MLScld);
329         *cloud_target = MLScld;
330         cout << "Least_squares_smoothing_Done!";
331
332         // Add the filtered cloud to the ICP visualizer viewports:
333         if (!targetCloudPresent){
334             visualization::PointCloudColorHandlerRGBField<PointT> rgb_sveis(cloud_target);
335             ICPVisualizer->addPointCloud<PointT>(cloud_target, rgb_sveis, targetCloudname,
                v1);
336             ICPVisualizer->setPointCloudRenderingProperties(visualization::
                PCL_VISUALIZER_POINT_SIZE, 5, targetCloudname);
337             ICPVisualizer->addPointCloud<PointT>(cloud_target, rgb_sveis, targetCloudnameV2
                , v2);
338             ICPVisualizer->setPointCloudRenderingProperties(visualization::
                PCL_VISUALIZER_POINT_SIZE, 5, targetCloudnameV2);
339             targetCloudPresent = true;
340         }
341         visualization::PointCloudColorHandlerRGBField<PointT> rgb_extracted_target(cloud_target
            );
342         ICPVisualizer->updatePointCloud<PointT>(cloud_target, rgb_extracted_target,
            targetCloudname);
343         ICPVisualizer->updatePointCloud<PointT>(cloud_target, rgb_extracted_target,
            targetCloudnameV2);
344
345         // If ICP is used and model is moved, this resets the model cloud in ICP window
346         visualization::PointCloudColorHandlerRGBField<PointT> rgb_model_reset(cloud_model);

```

```

347         ICPVisualizer->updatePointCloud<PointT>(cloud_model, rgb_model_reset,
           modelCloudname_ICP);
348
349         saveTargetCloud = false;
350     }
351     // Loading the CAD model into the scene:
352     if (loadCADmodel){
353         // Load file
354         pcl::io::loadPolygonFileSTL(cadName, mesh);
355
356         // Scale the cloud, since the PCL visualizer uses meters and not millimeters as unit:
357         PointCloud<PointXYZ> temp_trans_cloud;
358         fromPCLPointCloud2(mesh.cloud, temp_trans_cloud);
359         Eigen::Matrix4f scaleCAD = Eigen::Matrix4f::Identity();
360         float scaleFactor = 0.001;
361         scaleCAD(0, 0) = scaleFactor;
362         scaleCAD(1, 1) = scaleFactor;
363         scaleCAD(2, 2) = scaleFactor;
364         pcl::transformPointCloud(temp_trans_cloud, temp_trans_cloud, scaleCAD);
365         toPCLPointCloud2(temp_trans_cloud, mesh.cloud);
366
367         // Add the scaled CAD model mesh to the Kinect stream visualizer:
368         KinectStreamVisualizer->addPolygonMesh(mesh);
369
370         // Sampling 5000 points on the CAD model:
371         vtkSmartPointer<vtkPolyData> meshVTK;
372         VTKUtils::convertToVTK(mesh, meshVTK);
373         uniform_sampling(meshVTK, 5000, *pointCloudFromMESH);
374
375         // Copies the CAD sampled cloud into the model cloud:
376         copyPointCloud(*pointCloudFromMESH, *cloud_model);
377
378         // Add the sampled point cloud to the ICP visualizer:
379         ICPVisualizer->addPointCloud<PointT>(cloud_model, modelCloudname, v1);
380         ICPVisualizer->setPointCloudRenderingProperties(visualization::
           PCL_VISUALIZER_POINT_SIZE, 5, modelCloudname);
381
382         loadCADmodel = false;
383     }
384     // This starts the ICP pose estimation:
385     if (runICP){
386         if (!ICPRunning){
387             boost::thread workerThread(&SimpleViewer::IterativeClosestPointfunction, this);
388             runICP = false;
389         }

```

```

390         else{
391             cout << "\nProcess_running,wait_a_minute..\n";
392             runICP = false;
393         }
394     }
395     // Communicating the updated pose to the robot controller:
396     if (commRob){
397         std::vector<unsigned char> formattedMessage, receivedMessage;
398         string newVarValueStr;
399
400         // Open the socket connection if not connected already:
401         if (!socketConnected){
402             boostclientcross.connectSocket(ip, port);
403             socketConnected = true;
404         }
405
406         // Communicating the updated XYZ-ABC values:
407         cout << "X-translation ,_";
408         newVarValueStr = boost::lexical_cast<string>(1000 * transformationMatrix(0, 3));
409         vector<unsigned char> newVarValueX(newVarValueStr.begin(), newVarValueStr.end());
410         formattedMessage = boostclientcross.formatWriteMsg(transIX, newVarValueX);
411         receivedMessage = boostclientcross.sendMsg(formattedMessage);
412
413         cout << "Y-translation ,_";
414         newVarValueStr = boost::lexical_cast<string>(1000 * transformationMatrix(1, 3));
415         vector<unsigned char> newVarValueY(newVarValueStr.begin(), newVarValueStr.end());
416         formattedMessage = boostclientcross.formatWriteMsg(transIY, newVarValueY);
417         receivedMessage = boostclientcross.sendMsg(formattedMessage);
418
419         cout << "Z-translation ,_";
420         newVarValueStr = boost::lexical_cast<string>(1000 * transformationMatrix(2, 3));
421         vector<unsigned char> newVarValueZ(newVarValueStr.begin(), newVarValueStr.end());
422         formattedMessage = boostclientcross.formatWriteMsg(transIZ, newVarValueZ);
423         receivedMessage = boostclientcross.sendMsg(formattedMessage);
424
425         cout << "A-rotation ,_";
426         double Aangle = atan2(transformationMatrix(1, 0), transformationMatrix(0, 0));
427         newVarValueStr = boost::lexical_cast<string>(RADIODEG(Aangle));
428         vector<unsigned char> newVarValueA(newVarValueStr.begin(), newVarValueStr.end());
429         formattedMessage = boostclientcross.formatWriteMsg(rotateA, newVarValueA);
430         receivedMessage = boostclientcross.sendMsg(formattedMessage);
431
432         cout << "B-rotation ,_";
433         double Bangle = atan2(-transformationMatrix(2, 0), transformationMatrix(0, 2));
434         newVarValueStr = boost::lexical_cast<string>(RADIODEG(Bangle));

```

```

435         vector<unsigned char> newVarValueB(newVarValueStr.begin(), newVarValueStr.end());
436         formattedMessage = boostclientcross.formatWriteMsg(rotateB, newVarValueB);
437         receivedMessage = boostclientcross.sendMsg(formattedMessage);
438
439         cout << "G-rotation, ";
440         double Cangle = atan2(transformationMatrix(2, 1), transformationMatrix(2, 2));
441         newVarValueStr = boost::lexical_cast<string>(RADIODEG(Cangle));
442         vector<unsigned char> newVarValueC(newVarValueStr.begin(), newVarValueStr.end());
443         formattedMessage = boostclientcross.formatWriteMsg(rotateC, newVarValueC);
444         receivedMessage = boostclientcross.sendMsg(formattedMessage);
445
446         cout << "\nRobot_variables_updated!" << endl;
447         commRob = false;
448     }
449     KinectStreamVisualizer->spinOnce(1, true);
450     ICPVisualizer->spinOnce(1, true);
451 }
452 }
453
454 // A function for printing out the alignment results:
455 void SimpleViewer::print4x4Matrix(const Eigen::Matrix4d & matrix){
456     printf("Rotation_matrix:\n");
457     printf("____|_%.3f_%.3f_%.3f_|_\n", matrix(0, 0), matrix(0, 1), matrix(0, 2));
458     printf("R_|_%.3f_%.3f_%.3f_|_\n", matrix(1, 0), matrix(1, 1), matrix(1, 2));
459     printf("____|_%.3f_%.3f_%.3f_|_\n", matrix(2, 0), matrix(2, 1), matrix(2, 2));
460     printf("Translation_vector:\n");
461     printf("t=<_%.3f,_%%.3f,_%%.3f>\n", matrix(0, 3), matrix(1, 3), matrix(2, 3));
462 }
463 // Function for registering keyboard inputs from user:
464 void keyboardEventOccurred(const visualization::KeyboardEvent& event, void* nothing){
465     if (event.getKeySym() == "n" && event.keyDown()){
466         saveTargetCloud = true;
467         cout << "\nKey_'n'_pressed, saving_a_target_cloud.\n";
468     }
469     if (event.getKeySym() == "k" && event.keyDown()){
470         loadCADmodel = true;
471         cout << "\nKey_'k'_pressed, loading_CAD_model_into_scene.\n";
472     }
473     if (event.getKeySym() == "m" && event.keyDown()){
474         runICP = true;
475         cout << "\nKey_'m'_pressed, starting_ICP.\n";
476     }
477     if (event.getKeySym() == "l" && event.keyDown()){
478         cout << "\nKey_'l'_pressed, sending_corrected_pose_to_robot";
479         commRob = true;

```

```

480     }
481 }
482
483 void SimpleViewer::run() {
484     // Creating the two visualizers
485     KinectStreamVisualizer = new visualization::PCLVisualizer("Kinect_Stream_Viewer");
486     ICPVisualizer = new visualization::PCLVisualizer("Point_Cloud_and_ICP_Viewer");
487
488     // Setting the viewer colors
489     float bckgr_gray_level = 1.0; // Black
490     float txt_gray_lvl = 1.0 - bckgr_gray_level;
491
492     // Add a coordinate system
493     KinectStreamVisualizer->addCoordinateSystem(1.0);
494
495     // Set the desired camera poses
496     KinectStreamVisualizer->setCameraPosition(2.47, 3.20, 1.80, -0.26, -0.31, 0.91, 0);
497     ICPVisualizer->setCameraPosition(2.47, 3.20, 1.80, -0.26, -0.31, 0.91, 0);
498
499     // Register keyboard callback
500     KinectStreamVisualizer->registerKeyboardCallback(keyboardEventOccurred, (void*)&KinectStreamVisualizer);
501     ICPVisualizer->registerKeyboardCallback(keyboardEventOccurred, (void*)&ICPVisualizer);
502
503     // Create two vertically separated viewports for ICPVisualizer
504     ICPVisualizer->createViewPort(0.0, 0.0, 0.5, 1.0, v1);
505     ICPVisualizer->createViewPort(0.5, 0.0, 1.0, 1.0, v2);
506
507     // Calibration values from camera calibration:
508     // Rotation matrix:
509     calibrationMatrix(0, 0) = -0.002569;
510     calibrationMatrix(0, 1) = 0.999933;
511     calibrationMatrix(0, 2) = 0.011321;
512     calibrationMatrix(1, 0) = 0.999570;
513     calibrationMatrix(1, 1) = 0.002898;
514     calibrationMatrix(1, 2) = -0.029180;
515     calibrationMatrix(2, 0) = -0.029211;
516     calibrationMatrix(2, 1) = 0.011241;
517     calibrationMatrix(2, 2) = -0.999510;
518     // Translation vector:
519     calibrationMatrix(0, 3) = 0.176108833 - 0.01326875;
520     calibrationMatrix(1, 3) = 0.260746314 + 0.04710675;
521     calibrationMatrix(2, 3) = 1.364965019;
522
523     while (!KinectStreamVisualizer->wasStopped()) {
524         if (cloud_ != NULL) {

```

```

525             boost::interprocess::scoped_lock<boost::mutex> lock (cloud_mutex_);
526             updatePointCloud (cloud_);
527             cloud_.reset ();
528         }
529         if (recieved_first_) KinectStreamVisualizer->spinOnce(30, false);
530     }
531 }
532
533 void SimpleViewer::cloudCB(const PointCloudT::ConstPtr& cloud){
534     boost::interprocess::scoped_lock<boost::mutex> lock (cloud_mutex_);
535     cloud_ = cloud;
536 }
537
538 //
539 // The following four functions are used for converting the CAD model,
540 // into a point cloud. They are adopted from PCL's mesh_sampling.cpp
541 //
542
543 // Generates random numbers in a specified range:
544 inline double uniform_deviante(int seed)
545 {
546     double ran = seed * (1.0 / (RAND_MAX + 1.0));
547     return ran;
548 }
549
550 // Draw a Sierpinski Triangle by plotting random points:
551 inline void randomPointTriangle(float a1, float a2, float a3, float b1, float b2, float b3, float c1, float c2,
552     float c3, Eigen::Vector4f& p)
553 {
554     float r1 = static_cast<float> (uniform_deviante(rand()));
555     float r2 = static_cast<float> (uniform_deviante(rand()));
556     float r1sqr = sqrtf(r1);
557     float OneMinR1Sqr = (1 - r1sqr);
558     float OneMinR2 = (1 - r2);
559     a1 *= OneMinR1Sqr;
560     a2 *= OneMinR1Sqr;
561     a3 *= OneMinR1Sqr;
562     b1 *= OneMinR2;
563     b2 *= OneMinR2;
564     b3 *= OneMinR2;
565     c1 = r1sqr * (r2 * c1 + b1) + a1;
566     c2 = r1sqr * (r2 * c2 + b2) + a2;
567     c3 = r1sqr * (r2 * c3 + b3) + a3;
568     p[0] = c1;
569     p[1] = c2;

```



```

569     p[2] = c3;
570     p[3] = 0;
571 }
572
573
574 inline void randPSurface(vtkPolyData * polydata, std::vector<double> * cumulativeAreas, double totalArea, Eigen
    ::Vector4f& p)
575 {
576     float r = static_cast<float>(uniform_deviate(rand()) * totalArea);
577
578     std::vector<double>::iterator low = std::lower_bound(cumulativeAreas->begin(), cumulativeAreas->end(),
        r);
579     vtkIdType el = vtkIdType(low - cumulativeAreas->begin());
580
581     double A[3], B[3], C[3];
582     vtkIdType npts = 0;
583     vtkIdType *ptIds = NULL;
584     polydata->GetCellPoints(el, npts, ptIds);
585     polydata->GetPoint(ptIds[0], A);
586     polydata->GetPoint(ptIds[1], B);
587     polydata->GetPoint(ptIds[2], C);
588     randomPointTriangle(float(A[0]), float(A[1]), float(A[2]),
589         float(B[0]), float(B[1]), float(B[2]),
590         float(C[0]), float(C[1]), float(C[2]), p);
591 }
592
593 // The main function for sampling points on a mesh surface:
594 void uniform_sampling(vtkSmartPointer<vtkPolyData> polydata, size_t n_samples, pcl::PointCloud<pcl::PointXYZ> &
    cloud_out)
595 {
596     polydata->BuildCells();
597     vtkSmartPointer<vtkCellArray> cells = polydata->GetPolys();
598
599     double p1[3], p2[3], p3[3], totalArea = 0;
600     std::vector<double> cumulativeAreas(cells->GetNumberOfCells(), 0);
601     size_t i = 0;
602     vtkIdType npts = 0, *ptIds = NULL;
603     for (cells->InitTraversal(); cells->GetNextCell(npts, ptIds); i++){
604         polydata->GetPoint(ptIds[0], p1);
605         polydata->GetPoint(ptIds[1], p2);
606         polydata->GetPoint(ptIds[2], p3);
607         totalArea += vtkTriangle::TriangleArea(p1, p2, p3);
608         cumulativeAreas[i] = totalArea;
609     }
610

```

```

611     cloud_out.points.resize(n_samples);
612     cloud_out.width = static_cast<pcl::uint32_t>(n_samples);
613     cloud_out.height = 1;
614
615     for (i = 0; i < n_samples; i++){
616         Eigen::Vector4f p;
617         randPSurface(polydata, &cumulativeAreas, totalArea, p);
618         cloud_out.points[i].x = p[0];
619         cloud_out.points[i].y = p[1];
620         cloud_out.points[i].z = p[2];
621     }
622 }
623
624 int _tmain(int argc, _TCHAR* argv[])
625 {
626     // Starting the viewer windows
627     SimpleViewer viewer;
628
629     // Connect to grabber
630     Grabber* grab = new Kinect2Grabber();
631
632     // Make callback function from member function
633     boost::function<void(const PointCloudT::ConstPtr&> callbackFunct =
634         boost::bind(&SimpleViewer::cloudCB, &viewer, _1);
635
636     // Connect callback function
637     grab->registerCallback(callbackFunct);
638
639     // Start receiving point clouds
640     grab->start();
641
642     // Check if grabber started successfully
643     if (!grab->isRunning()){
644         cout << "\nFailed_to_start_Kinect_v2\n";
645     }
646     else{
647         cout << "\nSuccessfully_started_Kinect_v2\n";
648     }
649
650     viewer.run();
651
652     boostclientcross.disconnectSocket(); // Disconnects from robot socket connection
653
654     return 0;
655 }

```

## A.2 Communication Client

Listing A.2: Source code of the robot communication client based on the Boost C++ library.

```

1  /*
2  Source code for the KUKA robot controller communication client.
3  Author: Eirik B. Njaastad.
4  NTNU 2015
5
6  Communicates with the KUKAVARPROXY server made by
7  Massimiliano Fago - massimiliano.fago@gmail.com
8  */
9
10 #ifndef BOOSTCLIENTCROSS
11 #define BOOSTCLIENTCROSS
12
13 #include <iostream>
14 #include <boost/array.hpp>
15 #include <boost/asio.hpp>
16 #include <boost/foreach.hpp>
17 #include <boost/lexical_cast.hpp>
18
19 boost::asio::io_service iosClientCross;
20 boost::asio::ip::tcp::socket socketClientCross(iosClientCross);
21 boost::system::error_code socketError;
22
23 class BOOSTCLIENTCROSS{
24 public:
25     // Function for opening a socket connection and initiate the server connection:
26     void connectSocket(std::string ipAddress, std::string portNumber){
27         socketClientCross.connect(
28             boost::asio::ip::tcp::endpoint(boost::asio::ip::address::from_string(ipAddress),
29             boost::lexical_cast<unsigned>(portNumber)));
30     }
31     // For writing a variable to the robot controller, the message to send must contain
32     // a variable name (varName) and a value to write (varValue).
33     std::vector<unsigned char> formatWriteMsg(std::vector<unsigned char> varName, std::vector<unsigned char>
34         > varValue){
35         std::vector<unsigned char> header, block;
36         int varNameLength, varValueLength, blockSize;
37         int messageId;
38         BYTE hbyte, lbyte, hbytemsg, lbytemsg;
39         varNameLength = varName.size();

```

```

40     varValueLength = varValue.size();
41     messageId = 05;
42
43     hbyte = (BYTE)((varNameLength >> 8) & 0xff00);
44     lbyte = (BYTE)(varNameLength & 0x00ff);
45
46     block.push_back((unsigned char)1);
47     block.push_back((unsigned char)hbyte);
48     block.push_back((unsigned char)lbyte);
49
50     for (int i = 0; i != varNameLength; ++i) {
51         block.push_back(varName[i]);
52     }
53
54     hbyte = (BYTE)((varValueLength >> 8) & 0xff00);
55     lbyte = (BYTE)(varValueLength & 0x00ff);
56
57     block.push_back((unsigned char)hbyte);
58     block.push_back((unsigned char)lbyte);
59
60     for (int i = 0; i != varValueLength; ++i) {
61         block.push_back(varValue[i]);
62     }
63
64     blockSize = block.size();
65     hbyte = (BYTE)((blockSize >> 8) & 0xff00);
66     lbyte = (BYTE)(blockSize & 0x00ff);
67
68     hbytemsg = (BYTE)((messageId >> 8) & 0xff00);
69     lbytemsg = (BYTE)(messageId & 0x00ff);
70
71     header.push_back((unsigned char)hbytemsg);
72     header.push_back((unsigned char)lbytemsg);
73     header.push_back((unsigned char)hbyte);
74     header.push_back((unsigned char)lbyte);
75
76     block.insert(block.begin(), header.begin(), header.end());
77     return block;
78 }
79 // For reading a variable from the robot controller, the message to send must contain
80 // the desired variable name (varName).
81 std::vector<unsigned char> formatReadMsg(std::vector<unsigned char> varName) {
82     std::vector<unsigned char> header, block;
83     int varNameLength, blockSize;
84     int messageId;

```

```

85         BYTE hbyte, lbyte, hbytemsg, lbytemsg;
86
87         varNameLength = varName.size();
88         messageId = 05;
89
90         hbyte = (BYTE)((varNameLength >> 8) & 0xff00);
91         lbyte = (BYTE)(varNameLength & 0x00ff);
92
93         block.push_back((unsigned char)0);
94         block.push_back((unsigned char)hbyte);
95         block.push_back((unsigned char)lbyte);
96
97         for (int i = 0; i != varNameLength; ++i) {
98             block.push_back(varName[i]);
99         }
100
101         blockSize = block.size();
102
103         hbyte = (BYTE)((blockSize >> 8) & 0xff00);
104         lbyte = (BYTE)(blockSize & 0x00ff);
105
106         hbytemsg = (BYTE)((messageId >> 8) & 0xff00);
107         lbytemsg = (BYTE)(messageId & 0x00ff);
108
109         header.push_back((unsigned char)hbytemsg);
110         header.push_back((unsigned char)lbytemsg);
111         header.push_back((unsigned char)hbyte);
112         header.push_back((unsigned char)lbyte);
113
114         block.insert(block.begin(), header.begin(), header.end());
115         return block;
116     }
117     // Send the formatted message and receive server response:
118     std::vector<unsigned char> sendMsg(std::vector<unsigned char> message){
119         // Send message:
120         const size_t bytes = boost::asio::write(socketClientCross, boost::asio::buffer(message));
121
122         // Read answer:
123         boost::array<unsigned char, 7> reheader;
124         size_t sendLen = socketClientCross.read_some(boost::asio::buffer(reheader), socketError); //
125             Header
126         int messageLength = reheader[3] - 6;
127         std::vector<unsigned char> recblock(messageLength);
128         size_t recLen = socketClientCross.read_some(boost::asio::buffer(recblock), socketError); //
129             Message

```

```
128
129     // Error handling:
130     if (socketError == boost::asio::error::eof)
131         std::cout << "Connection_closed_cleanly_by_peer" << std::endl;
132     else if (socketError)
133         throw boost::system::system_error(socketError); // Some other error.
134
135     // Print results (alternative):
136     // std::cout << "received: " << std::endl;
137     // for (int i = 0; i != recLen; ++i){
138     //     std::cout << recblock[i];
139     // }
140 }
141 // Function for terminating the socket and thus disconnect from server:
142 void disconnectSocket(){
143     socketClientCross.shutdown(boost::asio::ip::tcp::socket::shutdown_both, socketError);
144     socketClientCross.close();
145
146     // Error handling:
147     if (socketError)
148         throw boost::system::system_error(socketError);
149 }
150 };
151
152 #endif
```

# Appendix B

## Digital Appendix

A *.zip* file is included as digital appendix. This contains:

- The video *Kinect\_corrected\_welding\_demo.mp4* showing the welding of a thruster tunnel section at *Department of Production and Quality Engineering, NTNU*.
- Source code and a Windows build of the main developed C++ application in this project. The build requires a Kinect camera and Point Cloud Library 1.7.2 in order to run.
- Source code for the developed communication client, *BoostClientCross.h*.
- KUKA robot program files for welding, configured for cartesian pose corrections.
- KUKA WorkVisual configuration files for the EtherCat fieldbus used for setup of the robot-welding machine interface.
- Files for the KUKA.Sim robot cell layout.