



**NTNU – Trondheim**  
Norwegian University of  
Science and Technology

# Remote Operations of IRB140 with Oculus Rift

**Lars Tore Rørlien Carlsen**  
**Philip Røst Wehinger**

Master of Science in Cybernetics and Robotics  
Submission date: June 2015  
Supervisor: Tor Engebret Onshus, ITK

Norwegian University of Science and Technology  
Department of Engineering Cybernetics



# Contents

<b>1 Acknowledgements</b>	<b>viii</b>
<b>2 Summary and Conclusion</b>	<b>1</b>
2.1 Summary in Norwegian . . . . .	1
2.2 Summary . . . . .	2
2.3 Conclusion . . . . .	3
<b>3 Introduction</b>	<b>4</b>
3.1 History and Motivation . . . . .	4
3.2 Introduction of Previous Work . . . . .	5
3.2.1 Overview of the Concept . . . . .	5
3.2.2 Scope of Previous Work . . . . .	6
3.2.3 Choice of Equipment . . . . .	6
3.2.4 Oculus Rift . . . . .	6
3.2.5 ABB RobotStudio . . . . .	8
3.2.6 The Cameras . . . . .	9
3.2.7 C++ and the Oculus SDK . . . . .	10
3.3 Introduction . . . . .	11
3.3.1 Task Description . . . . .	11
3.3.2 Concept Overview . . . . .	11
3.3.3 The FlexPendant . . . . .	12
3.3.4 The IRC5 Controller . . . . .	13
3.3.5 External Computer . . . . .	16
3.3.6 The Robot Manipulator, IRB140 . . . . .	16
3.3.7 C++ and Qt . . . . .	18
3.3.8 RAPID, RobotStudio . . . . .	19
3.3.9 Ovrvision . . . . .	19
3.3.10 Camera Mount . . . . .	22
3.3.11 Directional Sound . . . . .	22
3.3.12 Safety Precautions . . . . .	25
<b>4 Theory</b>	<b>27</b>
4.1 Calibration of the Robot . . . . .	27
4.2 Network . . . . .	27
4.3 Stereo sound . . . . .	28
4.4 Software Overview . . . . .	29
4.4.1 Camera Communication, Ovrvision SDK . . . . .	29
4.4.2 Software Description of RAPID Server . . . . .	30
4.4.3 OculusRobot, Client Side Software . . . . .	31
4.4.4 Software Communication and Synchronization Description .	35
4.5 The Camera Specifications . . . . .	37
4.5.1 Image Transformation . . . . .	37
4.5.2 Minimum Focus Distance and Stereo Distance . . . . .	38
4.6 Linear Interpolation . . . . .	39

4.7	Performance Definitions . . . . .	40
4.7.1	Area Between Curves . . . . .	41
4.7.2	Finding the Delay . . . . .	42
4.8	Coordinate Systems . . . . .	43
4.8.1	Tool Center Point . . . . .	43
4.8.2	Base Frame . . . . .	43
4.8.3	Wrist Frame . . . . .	44
4.8.4	Tool Frame . . . . .	44
4.8.5	Coordinate Transformation From Oculus Rift to Robot TCP	45
4.9	Performance Parameters . . . . .	48
4.9.1	zonedata . . . . .	48
4.9.2	Future Estimation with OVR SDK . . . . .	49
4.9.3	Prefetch Time . . . . .	50
4.9.4	Path Resolution . . . . .	50
4.9.5	WaitTime . . . . .	50
4.9.6	MoveAbsJ and MoveL . . . . .	51
4.9.7	Process Update Time . . . . .	51
4.9.8	Queue Time . . . . .	51
4.9.9	Event Preset Time . . . . .	52
4.10	Multithreading . . . . .	52
4.11	The Rotation Matrix . . . . .	52
4.12	Forward Kinematics . . . . .	53
4.13	Inverse Kinematics . . . . .	55
4.13.1	Finding $\theta_1, \theta_2$ and $\theta_3$ , Geometric Approach . . . . .	55
4.13.2	Finding $\theta_4, \theta_5$ and $\theta_6$ , Algebraic Approach . . . . .	59
4.14	Singularities . . . . .	61
4.15	Sources of Delay . . . . .	63
<b>5</b>	<b>Literary Search</b>	<b>64</b>
5.1	Inverse Kinematics Literature . . . . .	64
5.2	Programming with Qt4 . . . . .	64
5.3	RobotStudio Manuals . . . . .	64
<b>6</b>	<b>Results</b>	<b>66</b>
6.1	Sources of Delay . . . . .	66
6.2	Offset Measurement . . . . .	68
6.3	Simulation vs the physical IRB140 . . . . .	69
6.4	tracking ability with different parameters . . . . .	70
6.4.1	Future Estimation with OVR SDK . . . . .	70
6.4.2	zonedata . . . . .	73
6.4.3	Prefetch Time . . . . .	80
6.4.4	Path Resolution . . . . .	82
6.4.5	Process Update Time . . . . .	85
6.4.6	Queue Time . . . . .	87
6.4.7	Move Joint Algorithm . . . . .	89
6.5	User Experience . . . . .	91

6.6	Parameter Adjustment Summary . . . . .	92
6.7	Sound . . . . .	93
6.8	Image Quality . . . . .	95
<b>7</b>	<b>Discussion</b>	<b>98</b>
7.1	Importance of Analyzing Measurements Versus How the System Feels . . . . .	98
7.2	Fine Tuning of System Parameters . . . . .	98
7.3	Total System Quality . . . . .	98
7.4	Consistency of Output Data from Standard Input in the Simulator	98
7.5	Vibrations and Subsequent Problems . . . . .	99
7.6	Drawbacks of Computing the Inverse Kinematics on the Client . . . . .	100
7.7	Possible Applications . . . . .	100
7.7.1	Remote Operations . . . . .	100
7.7.2	Demonstrational Purposes . . . . .	100
7.8	Latency . . . . .	101
7.9	RobotStudio and RAPID for Real-Time Tracking . . . . .	101
7.10	Choosing an Inverse Kinematics Algorithm . . . . .	102
<b>8</b>	<b>Further Work</b>	<b>103</b>
8.1	Custom Robotic Arm . . . . .	103
8.2	Different Robot Communication Environments . . . . .	103
8.3	Wireless transfer . . . . .	103
8.4	True 3D Sound . . . . .	104
8.5	Adaptive noise cancelling filter in the microphone . . . . .	104
8.6	New VR Headset . . . . .	104
8.7	Singularity Avoidance . . . . .	105
8.8	Better Cameras . . . . .	105
<b>Appendices</b>	<b>109</b>	
<b>A</b>	<b>Transformation Matrices</b>	<b>109</b>
<b>B</b>	<b>User Manual</b>	<b>111</b>
B.1	How To Calibrate the Robot . . . . .	111
B.2	Editing System Parameters In RobotStudio . . . . .	115
B.3	How to Run the Application . . . . .	117
B.3.1	RAPID Server Application . . . . .	117
B.3.2	C++ Client Application . . . . .	117
B.3.3	Running the system . . . . .	118
B.4	How To Navigate the CD . . . . .	120
<b>C</b>	<b>Code Samples</b>	<b>121</b>
C.1	C++ Qt . . . . .	121
C.1.1	main . . . . .	121
C.1.2	TCP connect and run-thread . . . . .	122

C.1.3	Oculus RiftHandler data extract	124
C.2	Inverse Kinematics	125
C.3	RAPID	127
C.3.1	TCPconnection thread	127
C.3.2	RobotMotion thread	130

## List of Figures

1	System overveiw . . . . .	5
2	A more precise description showing how the setup was when using the simulator. . . . .	6
3	A picture of the Oculus Rift with and without the case. In the picture without the case you can see the led lamps that provides positional tracking. The camera reads the pattern of the leds, and based on this information it calculates the position of the headset.	7
4	The Oculus Rift only has positional tracking while the operator remains inside the field of view of the IR-camera. . . . .	8
5	An image of the RobotStudio interface. . . . .	9
6	The adjustable mount made to hold the two Mobius ActionCams at interpupillary distance. . . . .	10
7	Illustration of the system setup. . . . .	12
8	The ABB FlexPendant . . . . .	13
9	The irc5 controller . . . . .	14
10	The control panel on the controller. . . . .	15
11	An image of the multipurpose robot, the IRB140. . . . .	17
12	A graphical representation of the range of the robotic arm. . . . .	18
13	The intended use of Ovrvision is to bring the real world into the rift screen. . . . .	20
14	The camera sensor chip inside the shell of the Ovrvision. . . . .	20
15	The UI of the Ovrvision calibration tool. . . . .	21
16	Due to low compatibility between the cameras and the tool plate on the robot it was attached with duct tape. . . . .	22
17	A minijack and USB connection is available from the bottom of the microphone. . . . .	24
18	The backside of the microphone shows the gain and the four modes which are available: stereo, cardioid, omnidirectional and bidirectional. . . . .	24
19	The security plexiglass surrounding the robot. . . . .	25
20	The door to enter the robot area. . . . .	26
21	Diagram of Network Communication . . . . .	28
22	The concept of stereo sound . . . . .	29
23	Class diagram for the robot server application. . . . .	30
24	Flow diagram of the RAPID server application. . . . .	31
25	The graphical user interface used to communicate with the robot . . . . .	32
26	The output if everything is connected correctly . . . . .	33
27	Class diagram of the C++/Qt software with dependencies . . . . .	34
28	Flowchart of the GUI options. . . . .	35
29	This sequence diagram shows the communication between the different parts during normal operation. . . . .	36
30	The asynchronous communication represented in another way, where the red line represents a border where what lies to the left of the line is asynchronous from whats on the right side of the line. . . . .	37

31	Left: Barrel distortion. Right: Pincushion distortion. Both picture relative to a parallel aligned picture. . . . .	38
32	The focus distance of the Ovrvision. . . . .	39
33	The figure shows the concept of how to generate missing data points with interpolation . . . . .	40
34	The area between the Oculus Rift plot and robot plot are marked in yellow . . . . .	41
35	illustration of the robots coordinate frames [20]. . . . .	44
36	This figure showes how the two coordinate systems are rotated with respect to each other . . . . .	45
37	This figure shows how the initial position of the Oculus Rift ( $OOx, OOy, OOz$ ) is defined. . . . .	46
38	This figure shows the offset the TCP has with respect to the origin of the robot coordinate system in x-direction and z-direction. . . . .	47
39	This figure shows the offset the TCP has with respect to the origin of the robot coordinate system in y-direction. . . . .	47
40	The origin of the base frame. . . . .	48
41	This figure illustrates the principle with <i>zonedata</i> . . . . .	49
42	illustration of the <i>zonedata</i> [6] . . . . .	49
43	Image of the <i>Production Window</i> used for multithreading in automatic mode. . . . .	52
44	Kinematic parameters of the ABB IRB 140 and frame assignment. . . . .	54
45	Kinematic decoupling displaying the elbow of the manipulator. . . . .	56
46	The projection of the WCP onto the $x_0 - y_0$ plane. . . . .	57
47	A projection of the plane formed by link2 and link3. . . . .	59
48	Symbolic representation of the IRB 140 . . . . .	60
49	Spherical wrist singularity[33]. . . . .	62
50	Illustration of the elbow singularity[33]. . . . .	62
51	Simulation of robot response to Oculus Rift with system parameters set to standard. . . . .	67
52	Scatterplot of the <i>MoveL</i> runtime. . . . .	68
53	This figure showes the plottet robot trajectory and the simulation trajectory with identical input . . . . .	69
54	$40ms$ vs the directly measured data . . . . .	70
55	$95ms$ vs the directly measured data when the operator moves the head at normal speed . . . . .	71
56	$95ms$ vs the directly measured data when the operator moves the head at very high speed . . . . .	72
57	Estimation for multiple different input parameters . . . . .	72
58	$95ms$ estimation vs $0ms$ ms estimation where the estimated data is time shifted to make the data overlapp . . . . .	73
59	the plots of all axis with $zonedata = z1$ and $WaitTime = 0.1$ . . . . .	75
60	There is a significant time difference between the signals in this plots . . . . .	76
61	the plots of all axis with $zonedata = z10$ and $WaitTime = 0.1$ . . . . .	77
62	the plots of all axis with $zonedata = z20$ and $WaitTime = 0.1$ . . . . .	78

63	the plots of all axis with <i>zonedata</i> = <i>fine</i> and <i>WaitTime</i> = 0 . . . . .	79
64	The Latency can be read from this plot to be about 0.3 seconds . . . . .	80
65	It is hard to read any differences from this case to the default case with <i>prefetchtime</i> = 0.1 . . . . .	81
66	It is hard to read any differences from this case to the default case with <i>prefetchtime</i> = 0.1 and the <i>prefetchtime</i> = 2 case . . . . .	82
67	. . . . .	84
68	It is hard to read any differences from this case to the default case with <i>prefetchtime</i> = 0.1 and the <i>prefetchtime</i> = 2 case . . . . .	85
69	This figure shows the plot where the <i>processupdatetime</i> = 1.93 . . . . .	86
70	This figure shows the plot where the <i>queuetime</i> = 0.05 . . . . .	88
71	This figure shows the plot where the <i>queuetime</i> = 0.29 . . . . .	89
72	It can be seen that even when the <i>MoveAbsJ</i> algorithm is used, a delay occurs about the same as the one when <i>MoveL</i> with <i>zonedata</i> = <i>zFine</i> is utilized. . . . .	90
73	The delay here is more significant than with <i>MoveAbsJ</i> , but for <i>zonedata</i> = <i>zFine</i> , there is no significant difference in delay be- tween the two algorithms. . . . .	91
74	microphone setup with no vibration protection between the robot and the microphone . . . . .	94
75	micrphone setup with a protective foam between the microphone and the robot . . . . .	95
76	Showing the camera image in normal lighting conditions, as can be seen the settings of the camera here is acceptable. . . . .	97
77	Showing the camera pointing towards a light source, the settings is not optimal for this, and the light gets intense in the picture. . . . .	97
78	An image of the robot bolted to the floor. . . . .	99
79	Image of calibration point for the first joint. . . . .	111
80	Image of calibration point for the second joint. . . . .	112
81	Image of calibration point for the third joint. . . . .	112
82	Image of calibration point for the fourth joint. . . . .	113
83	Image of calibration point for the fifth joint. . . . .	113
84	Image of calibration point for the sixth joint. . . . .	114
85	Print screen showing the location of the Controller and RAPID tab. . . . .	115
86	Print screen showing the location where to change system param- eters. . . . .	116

## **1 Acknowledgements**

The authors would like to extend our thanks to our supervisor Tor E. Onshus for guiding us through the work. The authors would also like to thank Knut Reklev for his participation in getting the IRB140 up and running, and for being available. Thanks to Åsmund Røst Wien from Liverpool Institute of Performing Arts for consulting us with sound technology. The authors will also thank the mechanical workshop at the faculty for engineering cybernetics for their contribution in the Robotic Lab.

## 2 Summary and Conclusion

### 2.1 Summary in Norwegian

Denne artikkelen gir et forslag på hvordan man kan lage et oppsett som gjør at brukeren kan projeksere tilstedevarelsen til en annen lokasjon. Artikkelen legger frem et konsept på en ny måte man kan utføre fjernstyrte operasjoner på. Denne metoden gjør at brukeren får en økt følelse av nærvær, og kan derfor gjøre bedre og tryggere valg under operasjoner. Målet er å sammenkoble bevegelsen til operatørens hode til en robotarm med 6 frihetsgrader. Denne strømmer deretter video fra robotarmen tilbake til en stereoskopisk skjerm festet til operatørens hode. Andre mål er å inkludere strømming av lyd, øke videokvaliteten, studere invers kinematikk og forbedre den helhetlige ytelsen til systemet. Systemet kjører på ABB plattformen RobotStudio™.

Hardwaren som brukes for å gi operatøren video er Oculus Rift™, dette er et headset som brukes i virtuell virkelighet segmentet. Programvare ble skrevet for å hente data fra Oculus Rift og sende dataen videre til robotarmen. Denne programvaren inkluderer logging av data, TCP/IP tilkobling, et enkelt brukergrensesnitt og inkluderingen av flere tråder, dette gjelder både i programmeringsspråket C++ og RAPID. Åpen kildekode ble brukt for å håndtere kameramodulen, Ovrvision™. Dette ble brukt for å få sendt bildestrømmen fra roboten til Oculus Rift sin skjerm. Strømmingen av lyd ble gjort ved å inkludere Blue Yeti™ stereo- kondensator-mikrofoner i systemet. IRB140 ble brukt som robot arm ettersom dette var den tilgjengelige robotarmen på universitetet. I tillegg til hardware- og programvareutvikling ble alternative fremgangsmåter for å forbedre systemet foreslått. Dette inkluderer studier av hvordan man kan gjøre invers kinematikk for å transformere posisjonen fra hodet til de ulike vinklene i robotarmen sine ledd.

For å prøve og forbedre systemet, ble det gjort endringer på systemets parametere. Resultatet ble evaluert ved både fysisk testing og ved og analysere den loggede dataen. Noe av testingen ble gjort kun i simulator ettersom tester viste at dette korrelerte godt med den fysiske robotten. Fremtidige estimater av hodets posisjon ble gjort for å redusere den totale forsinkelsen. Estimater 40ms inn i fremtiden ble gjort og dette gav et resultat som var et godt kompromiss mellom nøyaktighet og forsinkelse. Reduseringen i forsinkelse ble også gjort ved å endre systemparametere og ved å bruke ulike innstillinger i bevegelsesalgoritmen. Den totale systemforsinkelsen endte i verste fall opp på omrent 0.3 sekunder etter den endelige iterasjonen med testing og forbedring. Gjennomsnittlig var forsinkelsen omrent på 0.2 – 0.25 sekunder, mens den i beste fall lå på omrent 0.1 sekund.

## 2.2 Summary

This article poses a suggestion on how to create a setup which projects the presence of the user to another location. The article gives a proof of concept of a new way of doing remote operations, giving the operator a better feel of presence and thus making the operator able to make safer and better decisions. The main goal is to link the movement of the human head to the movement of a 6 degrees of freedom robotic arm and streaming stereo video from the robotic arm back to a stereoscopic screen mounted to the operators head. Additional objectives is to include streaming of sound, improving the video quality, studying inverse kinematics and to improve how well the system performs as a whole on the ABB RobotStudio™platform.

The virtual reality hardware that was chosen was the Oculus Rift™. The software was written to extract data from the Oculus Rift and pass it on to the robotic arm. This software includes logging of data, TCP/IP connection, a simple GUI and multiple threads to handle the data flow both in the programming language C++ and RAPID. An open source software to handle a camera module, the Ovrvision™, was studied and modified. This was to get a camera feed from the robot to the Oculus Rift screen. The sound streaming system was included by using the Blue Yeti™stereo condenser microphone. The IRB140 was chosen as the robot manipulator as this was the robot arm available at the university. In addition to the hardware and software testing and development, alternative ways of improving the system is also suggested. This includes studies on how to do inverse kinematics to translate the Oculus position into joint angles of the robot.

In order to try to improve the system, adjustments on system parameters was done and the results was evaluated by both physical testing and analyzing of the logged data. Some of the testing was done only in the simulator as tests showed that this correlated well with the physical robot. Future estimation of the position of the head was included in order to reduce overall latency. Predictions 40ms into the future gave a result that compromised the need for low latency and accuracy. Reduction of latency was also done by changing system parameters and using other settings in the move algorithm, the total system latency was reduced to about 0.3 seconds, this being the worst case, in the final iteration of testing and improving. The average case was at about 0.2 – 0.25 seconds and best case of about 0.1 seconds.

## 2.3 Conclusion

As factories, power plants and offshore operations becomes more reliant on an autonomous workforce the need for human presence also changes. The main reason for human intervention would be to perform maintenance and inspection, however, the environments and locations might not be suitable for humans to physically enter. Thus the need for remote robotic operations with the use of virtual or augmented reality arises as it provides the human presence without the need to be there in person. This serves as the foundation for the main goal of the thesis, which is to create a complete working proof of concept operational on a robotic manipulator.

The most important feature of an augmented reality system is low latency. If the latency is too high the operator will not get a feeling of presence, and might experience motion sickness. Although tuning of system parameters proved beneficial most of the reduction in latency stemmed from using the *zonedata fine*. Contrary to early predictions that fluid motion was more important than rapid response, it turned out that the user experience improves with reduced latency even at the cost of more stuttering. In order to further reduce the latency a future estimation algorithm proved beneficial as it could reduce the latency by 40 to 90 ms with the only downside of insignificant accuracy loss. This gives an acceptable average case where the delay is barely noticeable. This combined with the addition of stereo sound makes the system give the operator an almost complete feeling of presence.

The concept has shown potential for further development, especially in the field of optimizing the inverse kinematics solver. However, this would require access to lower level control as the current RobotStudio algorithms are not fast enough to gain anything from it. In the future the concept could also expand with the addition of two extra robot manipulators to simulate hands. If this was accomplished then it would enable complete human presence and the possibilities for the application would only be limited by the imagination of those who applied it.

### **3 Introduction**

This paper is based on the work done in the project report written by the authors of this thesis in the fall of 2014. The introduction will therefore first focus on the previous work that was done during the fall, before it introduces the work that was done with the current thesis in the spring of 2015.

#### **3.1 History and Motivation**

Virtual reality is not a new invention, and has previously had varying degrees of success. This time, however, the technology has risen to new heights with the innovative solutions brought on by the Oculus Rift team. Prominent technology companies, such as Valve and Sony, are investing billions into the technology, and all the bricks of the technology are finally there to really make it a success. All the attention it has had in the media, and after testing the Oculus Rift, see Section 3.2.4, made the authors curious on how to combine this technology with robotics. This brought up the idea to use the Oculus Rift as a tool in remote operations. Controlling a robotic arm with its sensors, and using its screen to display whatever the robotic arm sees in full 3D. The potential for innovation in this field motivated the authors to make a proof of concept of how such a system could work.

In the paper "Control of Robot Arm Through Oculus Rift" [2] which this thesis builds upon, research was made on what components to use. How to set up the simulation environment with RobotStudio as well as discussing some simulations results is also described. This task motivated us to proceed with the work.

This leads to the natural goal of this master thesis in making the system work on the physical Robot. It was also important to test if it was able to provide good real time tracking with the ABB ecosystem, which is generally made for pre-planned trajectories. Another goal was to improve the software both to optimize it, make it easier to run tests on and make more sophisticated communication. In addition to this, the authors wanted to measure the performance of the tracking both analyzing it and testing the user experience. To test how the user experience changed when introducing bidirectional sound was also some of the motivation to proceed with the project.

## 3.2 Introduction of Previous Work

This section will introduce the previous work done on this project. The previous work will be on the CD attached to this thesis[2]. The main contribution being the project work by the authors. Although most of the previous system was created anew, it still remains relevant to understanding the current system. T

### 3.2.1 Overview of the Concept

The goal of the work that was done in the previous report was to study the possibility of having a 6 degrees of freedom robotic arm do real-time tracking of a system with sensors measuring 6 degrees of freedom, e.g. the Oculus Rift. The idea is to make the robotic arm track the head of the operator while streaming 3D stereoscopic video and eventually directional sound back to the operator. See Fig. 1 and 2 for an illustration of the concept. The positional vector from Eq. 1 is continuously sent from the headset to the robotic arm, and the camera module sends a video feed back to the headset.

$$x = [x \ y \ z \ \alpha \ \beta \ \gamma]^T \quad (1)$$

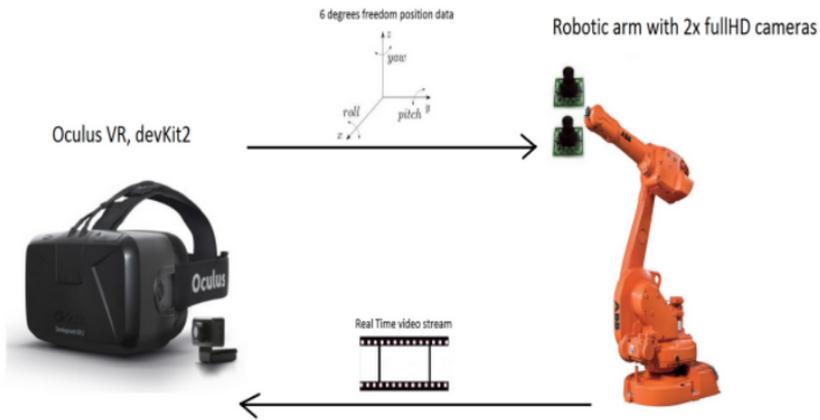


Figure 1: System overveiw.

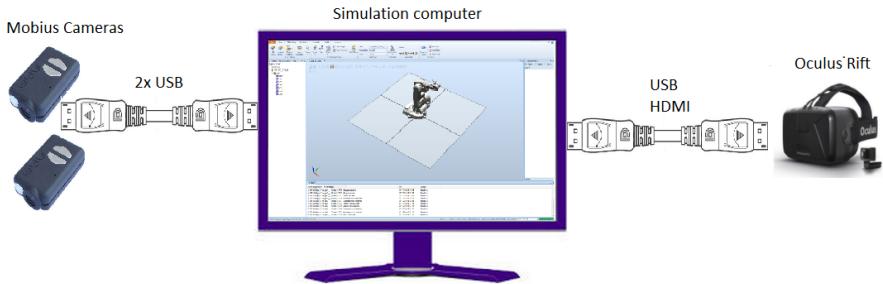


Figure 2: A more precise description showing how the setup was when using the simulator.

### 3.2.2 Scope of Previous Work

In the work done during the project work the robotic motion was only simulated, and not tried on an actual robotic manipulator, however, it was designed to work on the IRB140. Due to this the system as a whole could not be tested as the camera module was never attached to the robotic arm.

### 3.2.3 Choice of Equipment

Based on the equipment available at NTNU it was natural to choose the ABB IRB140 robotic arm and to do the simulations in the ABB provided software environment called RobotStudio which the institute provides free licences for.

At the start of the previous project the Oculus Rift DK2 was the only suitable option for a virtual reality headset. It has a large developer community and the software is fairly well documented. In addition it is relatively stable even though it is only a development kit and an early prototype of a virtual reality headset.

### 3.2.4 Oculus Rift

Oculus Rift is a new technology expected to have a profound impact on the future. It is called a Virtual Reality headset, which can be referred to as an immersive multimedia computer simulated experience. The version of the headset that was used for the project was the second development kit (Oculus Rift, DK2). This headset includes a full tracking system, both translation and rotation. The consumer version of the headset will be released in Q1 2016.

The 6 degrees of freedom (6DOF) tracking is done with the help of a gyroscope tracking its orientation, and a camera which tracks the position in space of the rift, see Fig. 4 and Fig. 3 for an explanation of how the position tracking works. The excellent tracking performance of the rift makes it a natural way to control

objects in a 3D environment. Conventional devices used for giving orientational input, such as a mouse, often has the limitation of 2DOF, while traditional controllers for computer games has 4DOF. To control a 6DOF robotic arm with a conventional controller can be hard and unintuitive, but when using the Oculus Rift to control it, the experience becomes more intuitive and requires less training to use accurately.

Because the company behind Oculus Rift focuses on gaming in a simulated environment, they put a lot of effort into making a good first person experience with low latency on both the tracking and the image. They have included a low persistence display which reduces motion judder when moving. The headset also covers 100 degrees field of view. This makes the operator feel that all he can see when wearing the headset, is the virtual reality.

The technical specifications of the rift is a data rate of 1000Hz from the internal sensors and a 60Hz update rate for the positional tracking. The headset is connected to the computer with a HDMI cable and two USB ports, one for the camera and one for the sensor data and power to the headset. The screen resolution is 960x1080 per eye, this is equivalent to a full HD display split into two parts.

As previously mentioned the concept of this paper is not to display a virtual reality in the Oculus Rift, but the reality from somewhere else. To learn more about the Oculus Rift it is recommended to watch the introduction video provided at their website[4].



Figure 3: A picture of the Oculus Rift with and without the case. In the picture without the case you can see the led lamps that provides positional tracking. The camera reads the pattern of the leds, and based on this information it calculates the position of the headset.

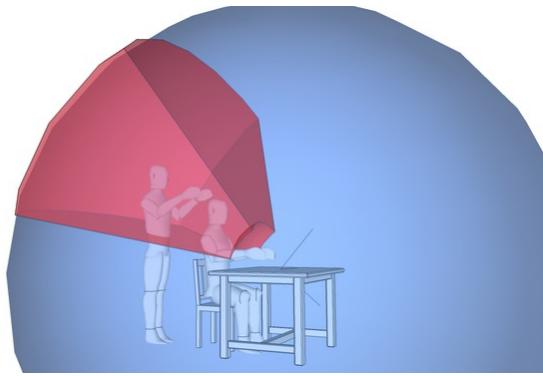


Figure 4: The Oculus Rift only has positional tracking while the operator remains inside the field of view of the IR-camera.

### 3.2.5 ABB RobotStudio

This is the IDE belonging to the ABB robot programming ecosystem. This is a virtual environment that simulates the physics of the physical robot, and is designed for testing and debugging the software before uploading to the physical robot. The program language used to program in RobotStudio is the RAPID programming language. This is a high level programming language dedicated to programming of the ABB portfolio of robotic arms. An example of the RobotStudio user interface can be seen in Fig. 5.

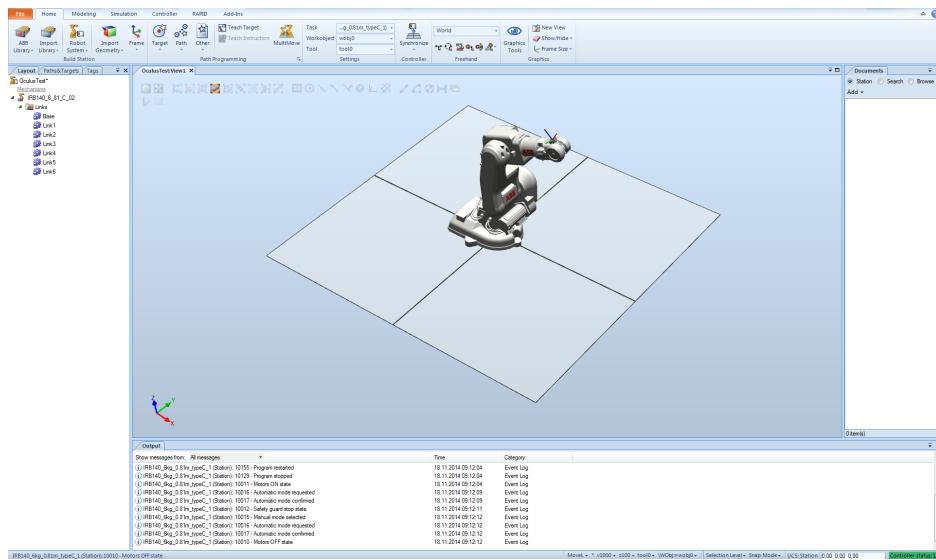


Figure 5: An image of the RobotStudio interface.

### 3.2.6 The Cameras

The cameras used in the project was of the type Mobius ActionCam, see Fig. 6 for an image of the camera module. These cameras provided a possibility to easily adjust resolution and other parameters through software to improve the image quality of the Oculus Rift. In order to get stereoscopic view from the cameras, a mount with an adjustable distance between the cameras was made. The reason why these cameras were not used in this project is due to the lack of support for the Oculus Rift, as well as a latency in the video stream which was larger than acceptable.

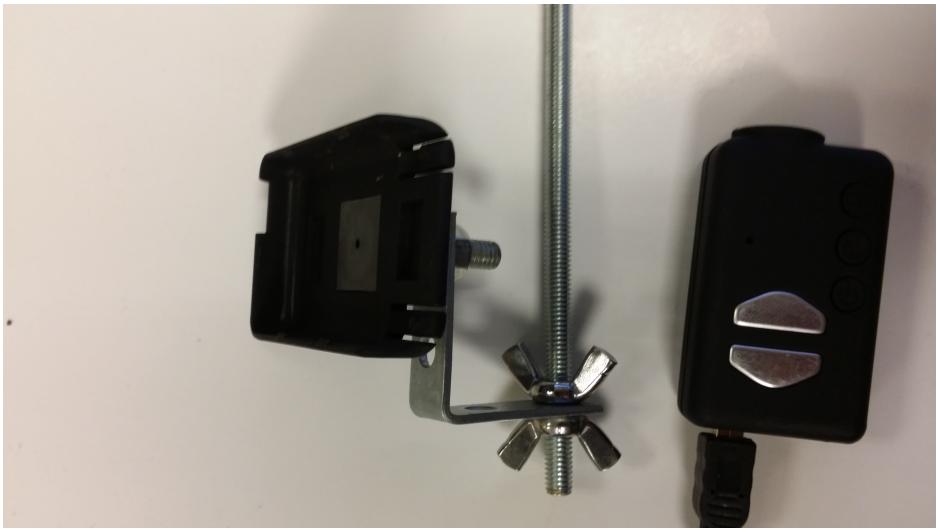


Figure 6: The adjustable mount made to hold the two Mobius ActionCams at interpupillary distance.

### 3.2.7 C++ and the Oculus SDK

During the initial phase of the software design the only available SDK for the Oculus Rift was written in C++. Later on the community has made other SDK e.g in python. There was no Linux driver for the Oculus at the time of this report, this is something that has been made later by the developers at Oculus inc. The Oculus software developer kit(SDK) has a manual[7] to explain the functions and how to build a basic application.

Due to the camera setup that was used it was necessary to create a custom viewer for the video stream. For such applications OpenCV and OpenGL provides a framework that is well documented and user friendly in the C++ version. When combined with the fact that C++ is suitable for real-time systems it remained a solid choice even after SDKs for other languages were developed.

### **3.3 Introduction**

#### **3.3.1 Task Description**

This thesis will build upon the project, written Fall 2014, which researched the implementability of the concept. As this project consisted of only simulations the main task of this thesis is to implement the system on an IRB140 as well as reducing the overall latency of the system. The following list sums up the main goals.

- Creating a communication network between the external computer and the robot controller.
- Analyze the sources of the latency.
- Reduce the latency of the total system.
- Improve user experience.
- Test the effect stereo sound has on the total user experience.

#### **3.3.2 Concept Overview**

In the section explaining earlier work there is a picture showing the simulation setup 2. This setup was extended in this thesis to include the physical robot and the Robot controller. Fig. 7 explains this setup. The setup consists of six main parts. The Flex pendant (1), the controller (2) the robot (3), an external computer (4), the Ovrvision camera (5) and the Blue Yeti™microphone (6). All these components communicates in different manners. The communication which had to be handled directly was the communication between the robot controller and the laptop. This will be explained more in detail in the technical part of the paper.

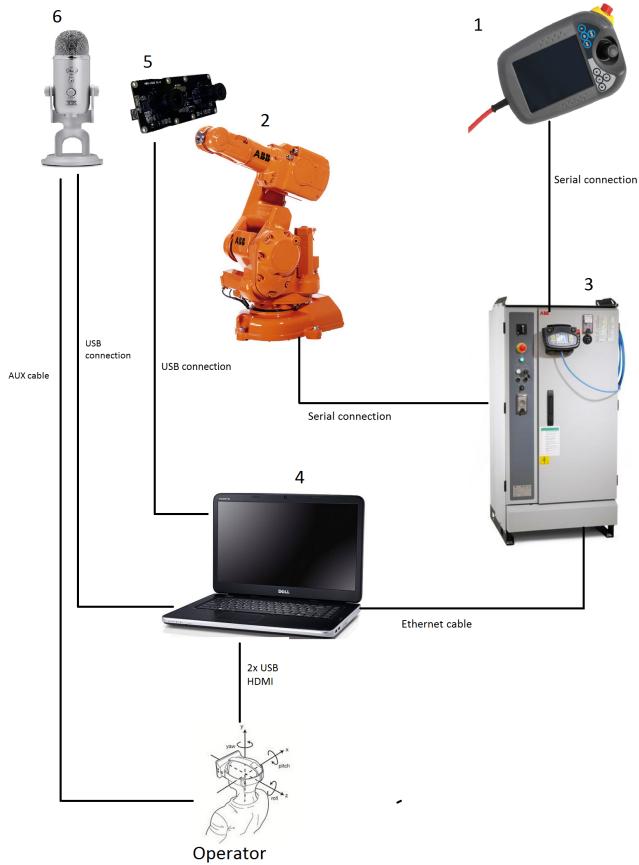


Figure 7: Illustration of the system setup.

### 3.3.3 The FlexPendant

The FlexPendant is the user interface(UI) of the robot controller. The program is uploaded directly to the robot controller from the external computer with a Transmission Control Protocol/Internet Protocol(TCP/IP) connection or serial connection. This can be then viewed as well as executed from the FlexPendant. It also has a built in editor for making programs, however, it is recommended to use this for only small programs. The FlexPendant operates with two different UIs based on the current operational mode it is in, namely the program editor and the production window. A convenient feature is that while running the program it is possible to see the program flow on the FlexPendant as well as write output for debugging purposes. The screen utilizes touch screen technology, however, it is designed to be used within industrial facilities which means it has high

durability with the cost of reduced responsiveness.

The robot controller is running two OS, a Unix core and a Windows core. There is a manual[39] following the FlexPendant which could be read to get more details about the computer inside it and how to use it.



Figure 8: The ABB FlexPendant

When the robot is set to manual mode, see Section 3.3.4, it is possible to use a joystick to control the joints of the robot. There are multiple ways to do this, both with controlling the individual joints, or by using linear movement. The joystick has 3 degrees of freedom, in addition to the traditional joystick, it is also possible to rotate the joystick to control the 3rd axis. In joint control mode, which is the optimal way to jog all 6 axis of the robot, one can switch between a mode where you control the first three joints, the elbow, and the outermost remaining joints, the spherical wrist.

### 3.3.4 The IRC5 Controller

The FlexPendant is linked to the robot controller, see Fig. 9. It runs a OS called RobotWare OS<sup>TM</sup>which is designed by ABB. The controller is designed for industrial purposes and therefore safety is a high priority of the system. This could be a disadvantage for this project where performance, speed and response time are the most important features, and a complex safety systems could lead to a reduced performance. It is also mainly designed for preplanned trajectories as this is how it is used in industrial applications. This is also a challenge, and to find out if our project is even possible to do on this controller is one of the things

to be studied in this thesis.



Figure 9: The irc5 controller

The ABB robotics ecosystem is a closed system where the end user does not have access to the lower layers of the software. The software gives the end user access to certain control parameters, but the exact details of the control algorithms and other functions remains hidden. There are also several features that require specific licenses. This proved to be challenging, however, the basic tools provided enough support to create workarounds.

The IRC5 controller has three modes it runs in. In Fig. 10 the input device at

the bottom is the one that is used to switch between these modes. To do this, a key is needed, this is a safety procedure as the robot can operate at considerable velocities and can be dangerous when operating in automatic mode.

- **Manual Mode:**

In this mode, the speed of the robotic arm is reduced to a safe level. In addition it will automatically stop if it impacts with some external object, and can only move if the dead man's switch is pressed. In manual mode it is possible to use the joystick on the FlexPendant to move the robot to a desired position, even though with very reduced speed. The amount you move the joystick controls the speed of the robot.

- **Manual Full speed mode:**

This mode does the same as the manual mode except that the robot is allowed to move in full speed in this mode.

- **Automatic mode:**

In this mode the robot is only allowed to move by running a pre-programmed script. This mean you cannot run the robot with the joystick in this mode. The robot should always be tested in manual mode before testing it in Automatic mode for safety purposes.



Figure 10: The control panel on the controller.

The IRC5 controller includes a battery. This battery is there in order to store the previous state of the robot when the robot is shut down or during a power loss. This battery usually lasts two years during normal usage, but if the robot is rarely used it will empty quicker as the battery charges when the power of the controller is on. If the battery is empty one can leave the robot on for a day or two to charge it.

### 3.3.5 External Computer

There are some requirements for the external computer when using the Oculus Rift[9]. These are mainly aimed at using Oculus Rift for games and not for remote operations, thus they are a bit lower in this case as it does not require any heavy rendering. There is however a hard real-time problem regarding the number of frames that are displayed. This is due to the fact that each frame that is dropped will be felt on a much more intense level when compared to framedrops on a monitor. The official requirements for running the Oculus Rift is:

- NVIDIA GTX 970 / AMD 290 equivalent or greater.
- Intel i5-4590 equivalent or greater.
- 8GB+ RAM
- Windows 7 SP1 or newer
- 2x USB 3.0 ports
- HDMI 1.3 video output supporting a 297MHz clock via a direct output architecture

These requirements can be lowered somewhat with respect to the graphic card and processor. Another reason for having a reasonably powerful external computer is that RobotStudio is a very demanding program which will decrease in performance if the specifications are too low. The specifications of the external computer used in this project is described below.

- Processor: Intel Core i5 (4th Gen) 4300U / 1.9 GHz Dual-Core
- RAM: 8 GB ( 2 x 4 GB ) DDR3L SDRAM
- Graphics card: NVIDIA GeForce GT 720M - 2 GB GDDR5 SDRAM
- Inputs: VGA, HDMI, 2 x USB 3.0, USB 2.0, LAN

### 3.3.6 The Robot Manipulator, IRB140

The current system is designed to work on the ABB IRB140, however, it is a trivial task to port it to other ABB robots. See Fig. 11 for an image of the IRB140 robot.



Figure 11: An image of the multipurpose robot, the IRB140.

This is one of the smaller robots in the ABB robot portfolio. It is described as a compact and powerful industrial robot[8]. It is a six axis multipurpose robot that can handle payloads of up to 6kg and a total reach of 810mm. Because the range of the operator is constrained by the view of the camera cone and the fact that the operator would be seated when wearing the Oculus Rift, this range is sufficient for the application. See Tab. 1 for a chart of the angular velocities and range of the joints and Fig. 12 for an illustration of its reachable and dexterous workspace.

Table 1: The maximum angular velocity and the range of the joints[8].

	Axis 1	Axis 2	Axis 3	Axis 5	Axis 6
Range [degrees]	360	200	280	400	230
Maximum velocity [degrees/s]	150	160	170	320	400
					460

The variation in speeds of the different joint is due to the different momentum of each joint. Axis 1 has to carry much more weight when moving in comparison to e.g axis 6, this makes the possible angular movement speed much higher. Translational movement of the operators head when seated is relatively low, this will also be possible to see in the results, and thus are the speed of the elbow joints sufficient. The maximum rotational speed of the human head is often estimated to be about 600 °/s. This speed is extreme and it is not necessary for the robot to be able to track such fast motions as this represents extreme cases, and not

movement in our everyday life.

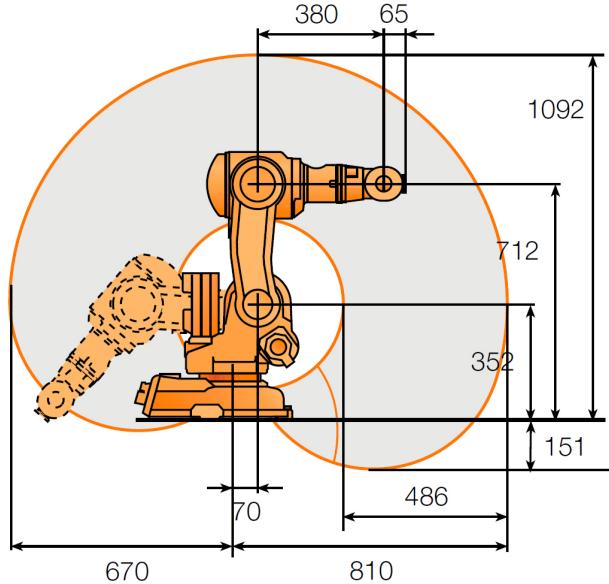


Figure 12: A graphical representation of the range of the robotic arm.

### 3.3.7 C++ and Qt

To communicate with the controller, C++ with Qt<sup>TM</sup> was used as the development tools. The programming was done in Visual Studio. Qt is a cross platform application framework developed by Nokia to make applications. It works like an upgrade to the C++ programming language with libraries including convenient classes and functions. In addition to this Qt has fast and easy libraries to develop graphical user interface (GUI). The ability to make a small and easy GUI was helpful when it came to testing the application of this paper. To easily switch input, source, output source, and enable/disable logging of data.

In the former version of the software the communication with the simulator was done using a shared file. When migrating to communication with the physical robot it was necessary to use a TCP/IP connection. That also introduced the need for multithreading on both the client side on the external computer, and on the server side on the robot controller. Qt has support for multithreading and includes classes which makes introducing a multithreaded program flow less complicated than with the STD libraries.

Keeping the opportunity open to include sound transfer from microphones to a headset via the software was important in order to introduce sound to the total

system. In Qt there are some samples on how to take sound-input from USB and send the stream out through the mini jack output on the computer in the Qt multimedia library. However, a more mechanical solution was chosen instead due to a limited amount of USB ports as well as the cost of proper microphones.

In the previous version of the software OpenCV was included in the application for the video streaming. As a new custom camera module was bought for this project, some open source programs followed written in C++. This software worked as intended and only needed some tuning in order to work for our application, more on this in Section 3.3.9. All of these advantages made C++ and Qt a natural choice to develop with.

### 3.3.8 RAPID, RobotStudio

As the robotic programming platform, ABB RobotStudio and RAPID was the simulation environment and programming language used during the entire project. RobotStudio features an interface designed for realistic testing of the ABB robots. Since the behaviour of the simulation was almost identical to the robotic manipulator it served as a platform to benchmark the performance of the different parameters with respect to the standard setup without having to always operate in the laboratory. In comparison to earlier usage the new features of multithreading, called tasks in RAPID, as well creating a network with TCP/IP were looked into. It is also possible to tune some of the system parameters to tune the performance of the robot which is a part of the studies.

Recently ABB launched a new version of RobotStudio and RobotWare which introduces several bug fixes as well as a feature called Externally Guided Motion(EGM). Unfortunately the software the robot controller used in this project is outdated and does not support this.

### 3.3.9 Ovrvision

Ovrvision[10] is a custom built camera module provided by a Japanese start-up company that aims to bring augmented reality to the Oculus Rift screen, see Fig. 13 and 14. It is specifically designed for the Oculus Rift, and also comes with a SDK to develop applications with the camera module.

In the previous camera setup, two Mobius ActionCams was bought to make the viewer in the Oculus Rift. A custom viewer was made in order to render the picture to the headset screen, however, it proved difficult to improve the quality to acceptable levels. There is a lot of research behind making satisfactory stereoscopic vision, and that is the why a custom hardware with open source software was chosen.



Figure 13: The intended use of Ovrvision is to bring the real world into the rift screen.

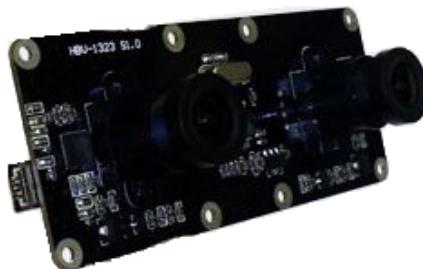


Figure 14: The camera sensor chip inside the shell of the Ovrvision.

Experimentation proved that the image resolution was not the most important thing when considering image quality. The Mobius ActionCams were both full HD quality. Other properties were far more important than the pixel density in order to get a pleasant experience, some adjustments were tested with the old setup, e.g to adjust the distance between the two frames rendered to the rift. The quantitative analysis on these adjustments was already done by Ovrvision.

On the Ovrvision website there is a developer tab. This brings up a StartUp Guide to start programming C++ in Visual Studio and a C++ reference[11]. This reference provides all functions needed to use the cameras and the Oculus Rift together. When downloading the SDK[13] there was also some example programs, one of which was called Ovrvision to Rift. Some experimentation and minor adjustments were done to the source code, but mainly this program provided what was necessary when coming to rendering a transformed image from the Ovrvision into the Oculus Rift.

The barrel and pincushion distortion is something that will be written more about in the theory section. The main point is that there has to be done an image transformation on the video feed when sending it to the rift. This was not done in the previous work which was some of the reason for the reduced quality when using the old camera setup. The decision behind not implementing it was based on the results that showed it introduced enough latency to warrant being left out. The Ovrvision solves this problem by doing this transformation with lenses on the cameras instead of software. This removes the delay of this transformation completely. The lenses used for this on the Ovrvision is called fish eye lenses and performs a pincushion distortion on the image that is passed through to the sensor.

It was necessary to perform a calibration of the cameras and the firmware to make the images overlap correctly. This was done using a calibration tool provided by Ovrvision. The calibration tool was provided with the SDK [13] and was run from there. The software is intuitive and there is also an instruction provided by the developers [14]. The approach used to calibrate the camera is to take 25 pictures of a pattern from different angles on the screen that shows when running the software, as can be seen in Fig. 15, and the cameras calibrates automatically. It is outside the scope of this paper to explain the theory behind this calibration. There is also an option to rotate the lenses of the Ovrvision to focus it so it adjusts to the operators eyes.

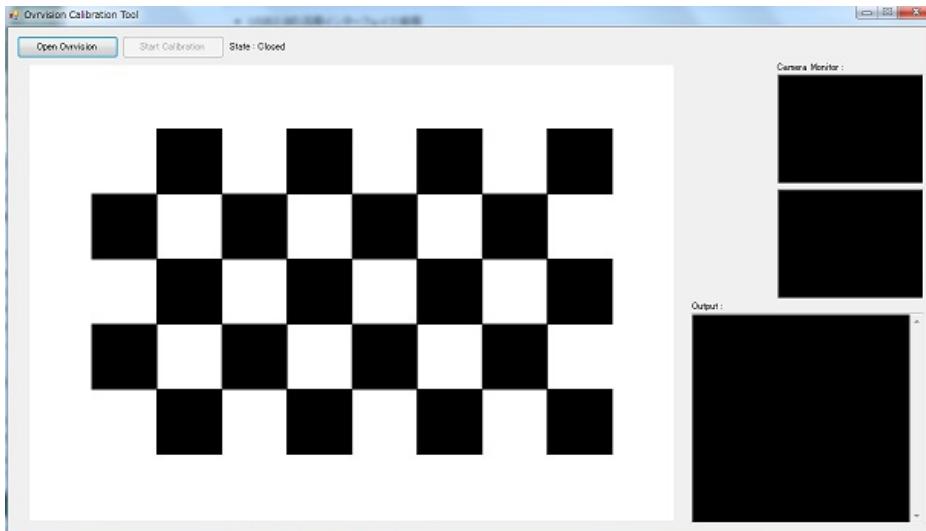


Figure 15: The UI of the Ovrvision calibration tool.

Technical specifications:

- USB 2.0

- Fish-eye lenses corresponding to Oculus Rift lens
- 60fps
- Resolution 1280x480
- 50ms latency

### 3.3.10 Camera Mount

By default it would be hard to attach a camera setup to the robot. This is why there is attached a metal plate at the outermost joint of the robot. This is a multipurpose mount and has previously been used to attached a gripper to the robotic arm. In this setup the mount was used to attach the camera system described in the previous section. Experience when testing the system showed that small inaccuracies in the camera placement on the robotic arm is negligible, see Fig. 16 for how it was attached.



Figure 16: Due to low compatibility between the cameras and the tool plate on the robot it was attached with duct tape.

### 3.3.11 Directional Sound

Initially there was only the input provided by the video stream given to the operator. However, the user experience seemed incomplete and the lack of sound proved to be a hindrance. Thus a simple solution was added to test the effects of directional sound to the system.

- Two active USB microphones with Qt multimedia:

This would be the most elegant solution, however there proved to be problems with using a USB HUB and QtMultimedia. This would have been necessary as there is not enough inputs on the computer to connect both the Oculus and the microphones. With this setup the sound handler could be run in a separate thread in the already existing software and stream stereo sound out through the mini jack output on the computer.

- Two XLR microphones and a mixer:

In this setup two Shure pg58<sup>TM</sup>vocal microphones and a xenyx 1002 mixer<sup>TM</sup>with two XLR inputs was tested. The two microphones was plugged into the mixer and a Twin RCA Phono to 3.5mm Mini Jack Stereo female was used to send the two sound signals to each ear in a headset. If the two microphones then is set to point in two separate directions this could give some of the wanted effect. However vocal microphones is not very good for this purpose. They mainly catches sound in a cardioid right in front of them and has very little range of sound capturing. This means that they only picked up sound that is very close to the microphones. The way to go would be to use condenser microphones used to record symphonic orchestras and choirs, these microphones are capable of recording sounds from a great range.

- Blue Yeti:

The final solution was using the Blue Yeti microphone[15]. This is a high quality USB microphone which poses good sound quality and multiple modes. It is a array of condenser microphones arranged in a pattern and has a built in amplifier.

In Fig. 17 one can see the bottom side of the microphone with its connections. It has a USB connection to connect the microphone to the computer and power the active components. In addition it has a mini jack output to connect a headset directly to the microphone. The yeti poses built in filters to give a decent sound output in this port. Another possible solution using this microphone would be to send the sound to the computer with the USB port and then use the mini jack output on the computer to transfer the sound. This would however require a dedicated sound driver to avoid delay as e.g ASIO4ALL<sup>TM</sup>[17], and even with this dedicated driver it poses higher delay then with the direct output on the microphone. This solution was also unrealistic because of the problems with using a USB hub.



Figure 17: A minijack and USB connection is available from the bottom of the microphone.

In Fig. 18 one can see the back side of the Blue Yeti microphone. Here there are two buttons, one controlling the gain of the microphone and the other one controlling the different modes to set the microphone in. The available modes are stereo, cardioid, omnidirectional and bidirectional. There are two of these modes that where relevant to us in order to have the possibility to detect where the sound is coming from. That is bidirectional and stereo mode. Unfortunately the bidirectional mode does not output a distinguished signal for left and right. The stereo mode does this and makes it possible to notice where the sound is coming from. The results of this will be discussed in the results section.

The best result for a sound system would probably be to have true 3D sound, like the 3DMicPro from Mitra [29].



Figure 18: The backside of the microphone shows the gain and the four modes which are available: stereo, cardioid, omnidirectional and bidirectional.

### 3.3.12 Safety Precautions

Due to the force that the robot can exert there needs to be some safety precautions. This is due to the fact that when running at maximum velocity it can cause injuries.

After the robot had been rigorously tested in the low speed manual mode it was switched to automatic mode, however, in order for the operations to be considered safe, additional security had to be added. The reason extra security had to be added was to avoid people from entering the workspace of the robot during operations. The first of these precautions were the added screen around the robot, see Fig. 19. Thus the only way to interact with the robot was to go through a door on the side, see Fig. 20. The entirety of the wall was placed on a distance greater than the reachable workspace of the robot making it impossible for the robot to break it.



Figure 19: The security plexiglass surrounding the robot.

Another thing that was important in order to avoid dangerous situations, and reduce the risk of damage to the robot, was to test changes in the software in manual mode first. If there appeared to be an unexpected bug in the software the robot would move at reduced speed and not do any damage. Even small changes in the software can cause the robot to behave unexpectedly. This is why this should be done consequently each time before the program is run in automatic mode.

A precaution that should have been taken, but was unfortunately not implemented was to connect the door to the emergency stop switch. This would make it so that if the door was opened the robot actuators would turn off and thereby avoiding any accidents.



Figure 20: The door to enter the robot area.

## 4 Theory

### 4.1 Calibration of the Robot

The manipulator requires two forms of calibration. The first is due to the fact that the Serial Measurement Board(SMB) battery unit no longer is capable of recharging. This causes it to lose the information held by the revolution counters. This causes the manipulator to lose track of its current position with respect to the origin. The reason for this is that the cabinet for the robot controller goes without electricity for long periods of time, which puts a large load on the internal battery. On how to calibrate the robot, see Appendix B.1.

The robot also requires a manual calibration, or reset, after each program execution. This is due to how RAPID stores the variables as they are not deleted after a program is done executing. This concerns variables for global use and not variables defined in functions. This causes problems when the robot is trying to move due to variables being stored as they were at the end of the last execution. Initially it was thought that a simple setup function would solve this, but it turned out it was insufficient to the task as the origin of the Tool Center Point(TCP) was based on the start position of the robot, and not the zero values of the individual joints. Thus a stand alone dedicated reset program had to be run after each execution to ensure nominal behaviour.

### 4.2 Network

The network for the system is set up as a TCP/IP network with the robot controller serving as a server, while the C++ application is acting as a client. See Fig. 21 for a simplified model of the network. TCP/IP protocol was chosen in order to keep the integrity of the messages. The reason that the integrity of the messages are of the utmost importance is that a single missing character from the string can cause the server to crash. This is mainly due to the algorithm that checks if the new coordinates that are being processed are within the reachable workspace of the manipulator. If the coordinates are off then this algorithm will fail to execute, due to the sum of the quaternions no longer being 1, and terminate the program.

Another reason for choosing TCP/IP over UDP is that the RAPID programming language has better support for socket messaging than it has for connectionless transmission. This is amplified by the fact that the robot controller only has the LAN connection option.

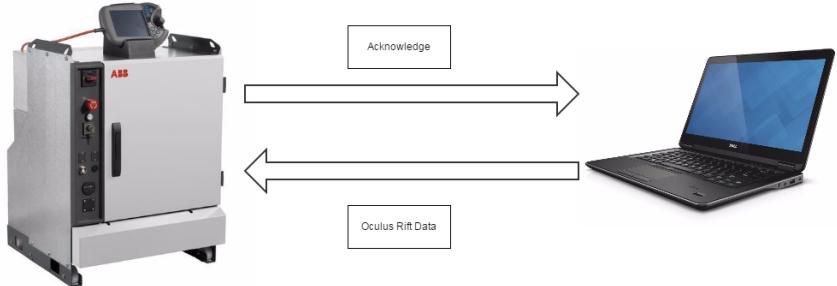


Figure 21: Diagram of Network Communication

A limiting factor in the design of the communication protocol is that the maximum string length in RobotStudio is 80 characters. Normally this might not be a problem, but RAPID reads all data from the buffer and puts it in a string. This limits the amount of data that is possible to send with each package by a significant amount since the position and orientation data can use almost the whole length. This has the unfortunate effect of preventing the implementation of buffer characters or start characters to ensure package integrity. Even though the string might not always be full, the remaining characters are used as a safety margin to make sure there is no overflow.

A way to implement the safety system to make sure that the messages does not overflow would be to send several small packages instead of one large, however, this has the disadvantage of being much slower as it introduces extra latency and was thus deemed to be an inferior solution.

### 4.3 Stereo sound

This report will not go in depth of this topic, but as the effect of sound turned out to have a positive impact on the total experience, the concept of stereo sound will be mentioned. In the microphone head of the Blue Yeti microphone there are multiple condenser microphones. Two of them are used to create the stereo effect by recording different areas. This is illustrated in Fig. 22. By streaming sound from the red and the black area to the corresponding ear of the operator, the operator will be able to determine the source of the sound.

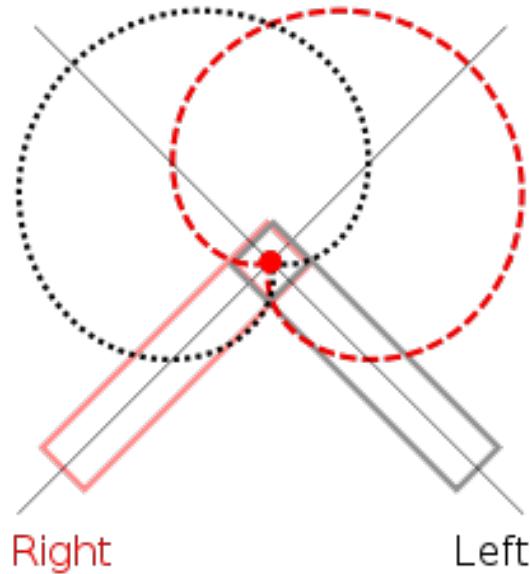


Figure 22: The concept of stereo sound

## 4.4 Software Overview

In the following sections firstly the different software components will be presented, in the last section, the communications between the components will be explained.

### 4.4.1 Camera Communication, Ovrvision SDK

The Ovrvision has a well established developer reference guide[21] and a SDK [22] with working examples. The example in the SDK called *examplevs2010orlater* was the one that was used. The functions used to tune the camera values was the following: The first try was to set the *OvPSQuality* in order to see if this affected the quality of the camera output:

```
void GetCamImage(unsigned char* pImageBuf, OVR::Cameye eye,
OvPSQuality processing)
```

Because the camera showed tendencies to be a little to bright, the following parameters where also tested. All controlling the brightness of the image:

```
void SetExposure(int value)
void SetBrightness(int value)
void SetGamma(int value)
```

The camera module sends data via an USB cable to the application which runs on the computer.

#### 4.4.2 Software Description of RAPID Server

The direct communication with the robot is done through a single command, *MoveL*, which solves the inverse kinematic problem and insures a linear path. However, in order to optimize the performance the application runs a series of tests each time to see if a move is necessary or legal. The program flow is described in Fig. 24. The main components of the tests consists of *LookForChange* and *isReachable*. If the change is too small the reduction of precision will make it so that the algorithm will perceive it to not have changed, and thus a move action will not be needed. The second algorithm guarantees that all move actions will remain within the reachable workspace of the robot. If the the command makes the robot go outside the reachable workspace, the robot will stop and start again as soon as a legal command occurs. If this function is not included the program will terminate if such a command is sent to the robot.

The RAPID server application is divided into two threads and their modules can be viewed in Fig. 23. In RobotStudio all variables are limited to the scope defined by the modules, which means the only way to share information between the threads is to use the RAPID version of global variables, *PERS*.

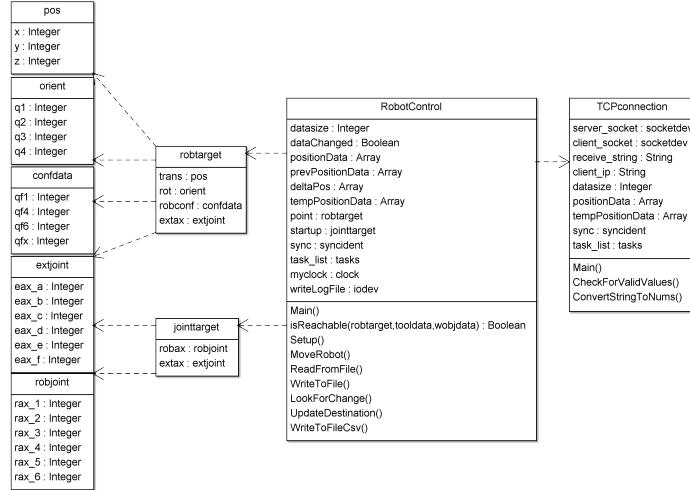


Figure 23: Class diagram for the robot server application.

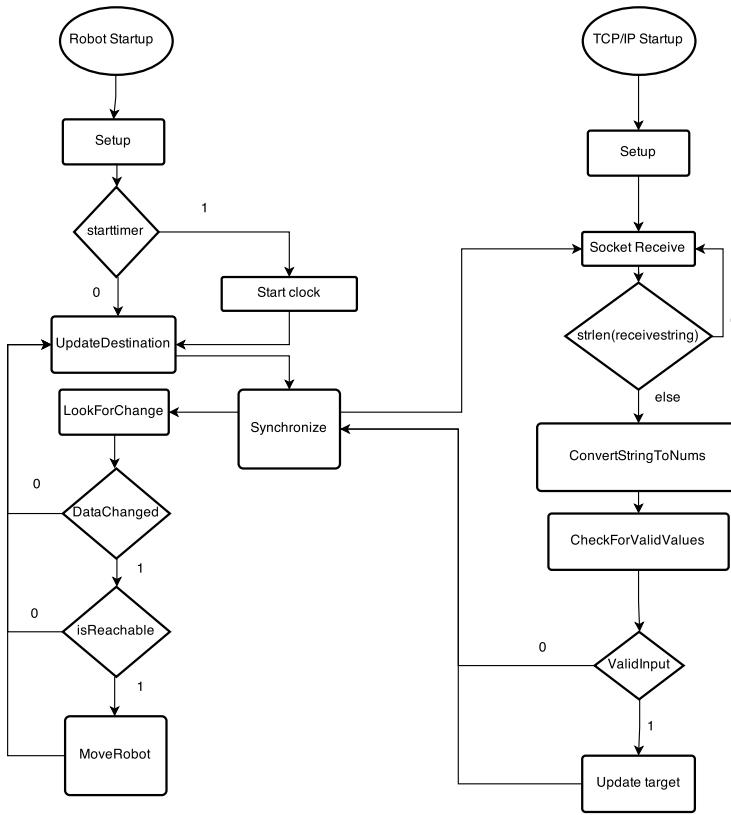


Figure 24: Flow diagram of the RAPID server application.

#### 4.4.3 OculusRobot, Client Side Software

To explain the C++ client application it is natural to start with getting an overview of the system by studying the UI. It is a simple UI made to simplify the process of continuously switching between the robot and simulation, different input sources and to enable/disable logging.

As can be seen in Fig. 25 the user interface consists of two parts. One of the parts enables switching between the output. This enables the user to set the output to the robot, to the simulation on the computer you are currently on or send the data to a machine with some other IP than the two default values, e.g another controller or another computer.

The second part of the GUI makes the user able to choose input source. If the

Input From File is enabled, the input to the output source will be logged data in a file in the program folder called *Oculus RiftValues*. If the *Input From Oculus* is chosen, the input will be from the Oculus Rift, an error message will occur if the computer cannot find the Oculus, i.e it has not been connected correctly.

The check box simply enables logging of the output data to a file the user can give a custom name. The file will be saved in the program folder. This is the logged data from the Oculus Rift, the log from the robot will be saved in the *HOME:* folder of RobotStudio. The logging procedure runs in a separate thread in order to log all the data points from the Oculus Rift.

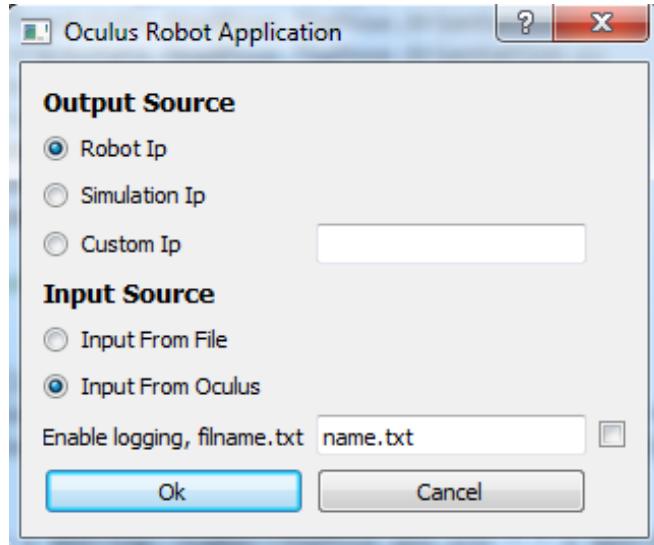


Figure 25: The graphical user interface used to communicate with the robot

The command window, see Fig. 26, shows how the output screen looks like if the programs are started correctly and simulation is set as output and Oculus is set as input. If the input is from a log file, the Oculus does not have to be connected for the program to function properly.

```
Velg C:\Users\bjornph\Desktop\oculusRobot3.0\Win32\Debug\oculusRobot3.exe
Debug: Performance timer Vista flag = 1
Debug: [NetClient] Connected to a server running version 1.1.0 <my version=1.0.0>
Debug: Setting up display shim
Debug: Failed to open file: C:\Users\bjornph\AppData\Local\Oculus\Devices.json
Debug: Using profile philip
Debug: [SharedMemory] Creating factory
Debug: [SharedMemory] Creating shared memory region: OVRObject00 > 1184 bytes
Oculus Detected

connection to Simulation set
input from oculus set
IP is set to "127.0.0.1"
TCP Connection Success!

in TCP send thread
```

Figure 26: The output if everything is connected correctly

The OculusRobot program runs two main threads. One thread for logging data, and the other for transmitting packages with TCP/IP to the output source. Both these threads calls the function

```
trackstate = ovrHmd_GetTrackingState(HMD, ovr_GetTimeInSeconds() + predictonTime);
```

in order to get the last state from the Oculus. This function is documented as a thread safe function in the OculusSDK reference[24].

To go a little more in depth of the functioning of the software, a class diagram with dependencies is presented in Fig. 27.

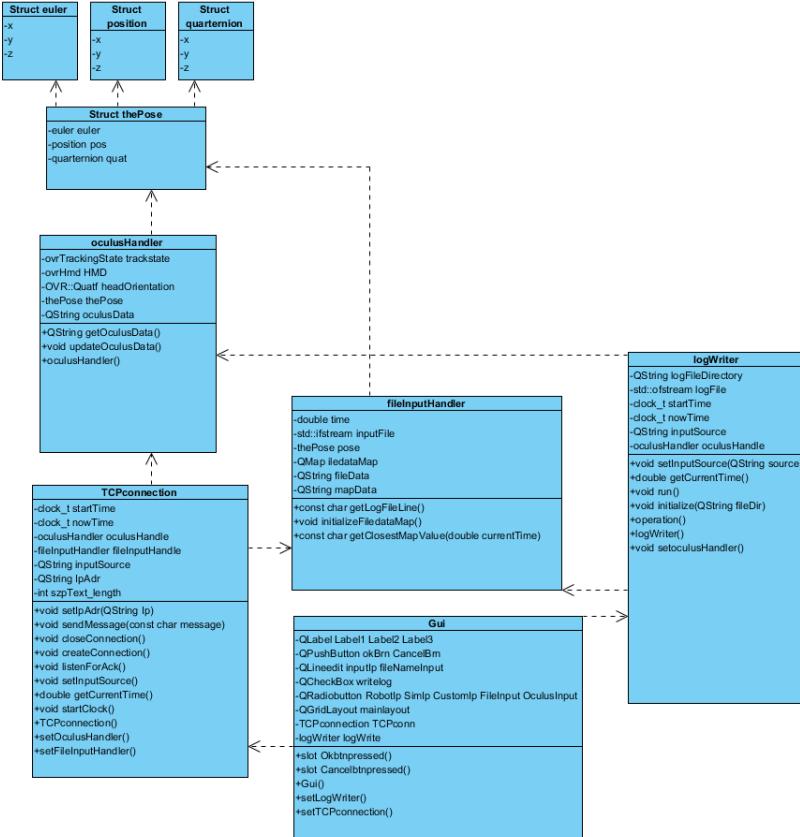


Figure 27: Class diagram of the C++/Qt software with dependencies

The flowchart diagram, see Fig. 28, shows the UI and what happens when the different options is enabled/disabled in the user interface. It gives an illustration of how the client software works and what software modules and threads that are enabled when different inputs in the GUI are given.

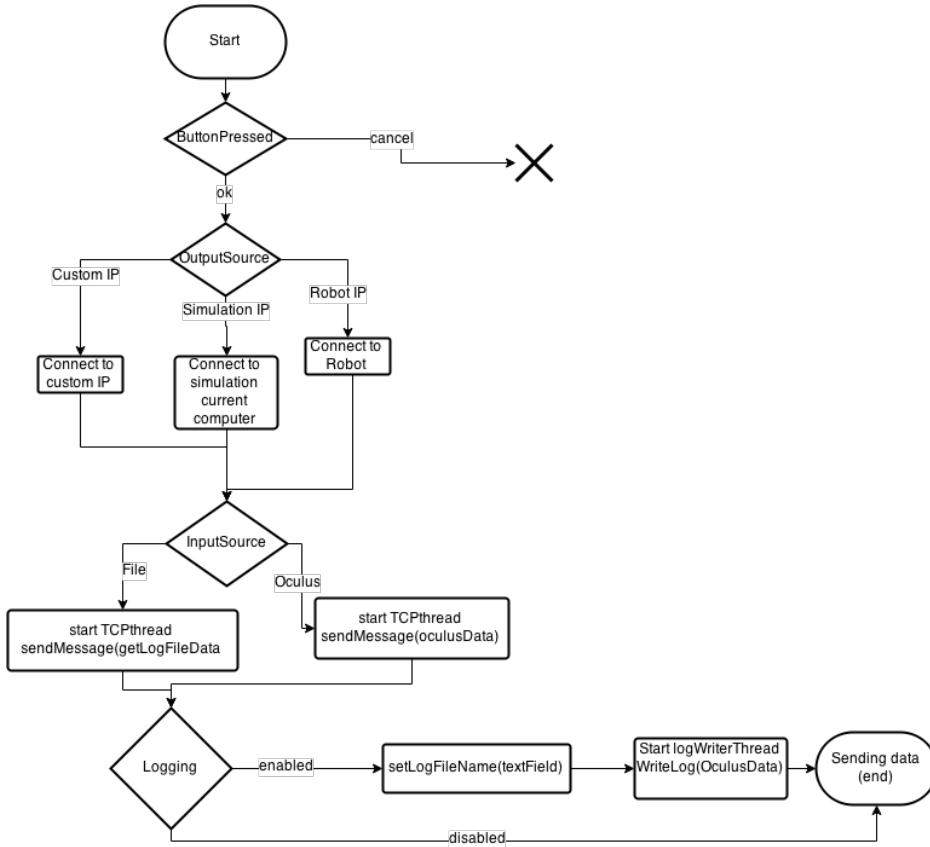


Figure 28: Flowchart of the GUI options.

When input source from file is enabled in the GUI, the system gets data from a logging file which have logged data from the Oculus Rift with an update rate of 30Hz. However, this is not synchronous with the TCP data transfer thread, which update rate could vary and is inconsistent. This is solved in the following manner: All the data from the input file is saved into a *QMap* data structure which contains a key which is the time the data was logged, and an item which contains the a position structure, *thePose*. When the application starts transmitting data to RobotStudio, a clock is started simultaneously in RobotStudio and the C++ program. This time is then used as a key each time data is passed from the QMap to RobotStudio, if the time key does not match any values in the map, the closest time key in the map will be chosen.

#### 4.4.4 Software Communication and Synchronization Description

The following sequence diagram, see Fig. 29 and 30, gives an explanation of how the system communicates. It is split into three parts: C++ computer, Robot

Controller and Robot Motions. This is because the program flow in RAPID is such that the program pointer in the robot controller and the robot not necessarily operates in the same place. The code runs on and the robot does so also. Some of the reason for this is because one cannot foresee how long it takes for the robot to finish a move command. If new move commands appear before the robot is finished with its last one, the move command is put into a first in first out(FIFO) queue, and the robot can get the last move value out of this queue when its done with its motion. This FIFO list is continuously being updated with values from the client.

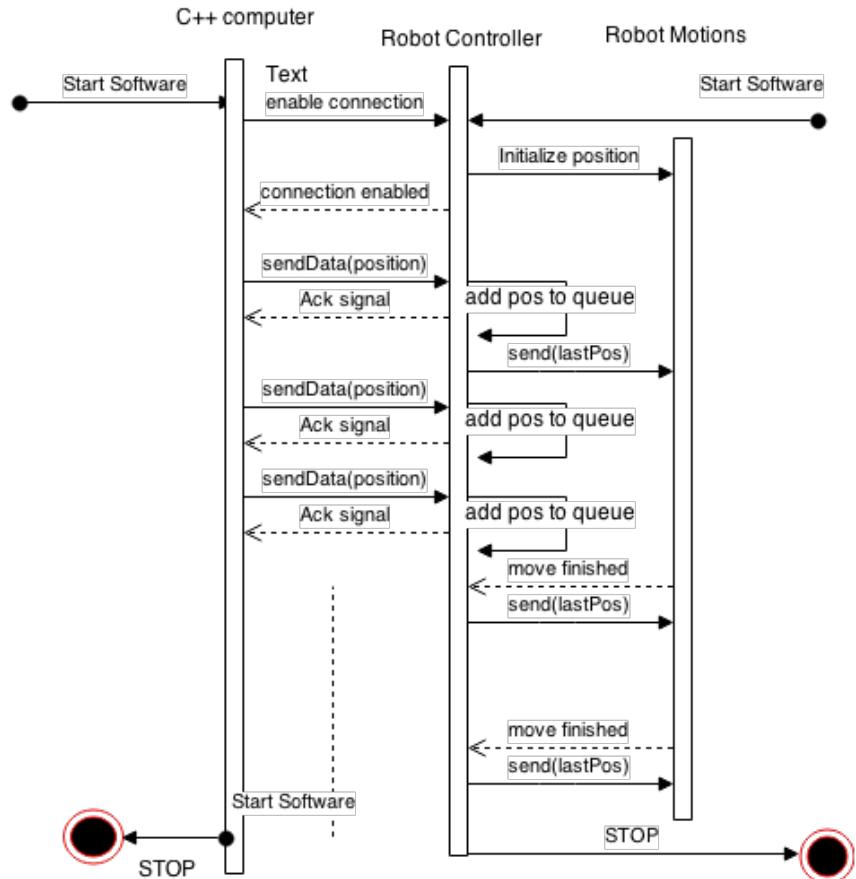


Figure 29: This sequence diagram shows the communication between the different parts during normal operation.

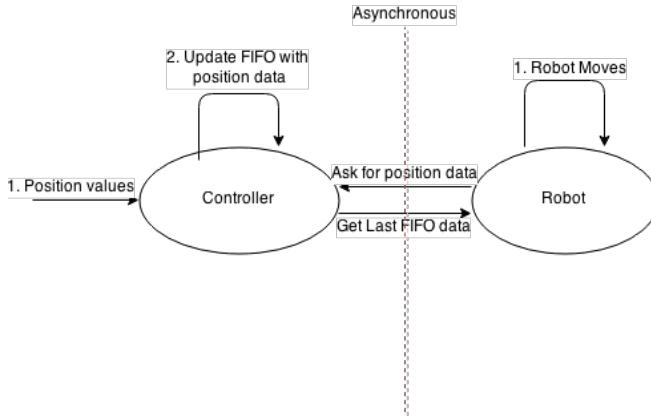


Figure 30: The asynchronous communication represented in another way, where the red line represents a border where what lies to the left of the line is asynchronous from what's on the right side of the line.

## 4.5 The Camera Specifications

### 4.5.1 Image Transformation

The Lenses of the Oculus Rift performs something called a Pincushion distortion on the picture before it reaches the eye of the operator. This is the inverse process of a barrel distortion, see Fig. 31. The transformation is done with the following formula,  $r_d$  gives the distance of each pixel from the center of distortion in the distorted image, and  $r_u$  the distance in the undistorted image.  $k$  is a distortion parameter given by the physics of the lens.

$$r_d = r_u(1 + kr_d^2) \quad (2)$$

The equation shows that the magnification decreases with the distance from the center. This causes each image point to move radially towards the center of the image, and thus cause the barrel distortion. The pincushion distortion is simply the inverse process of the barrel distortion[31].

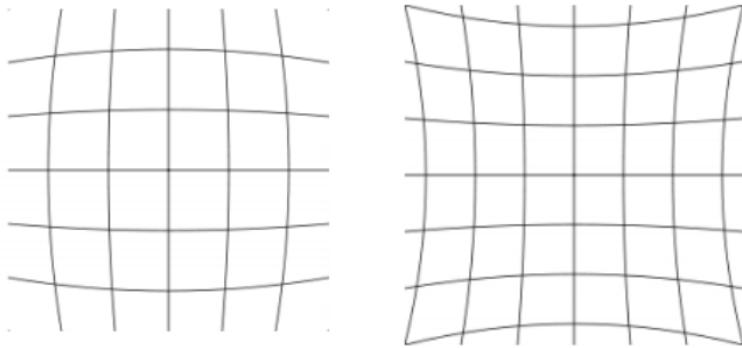
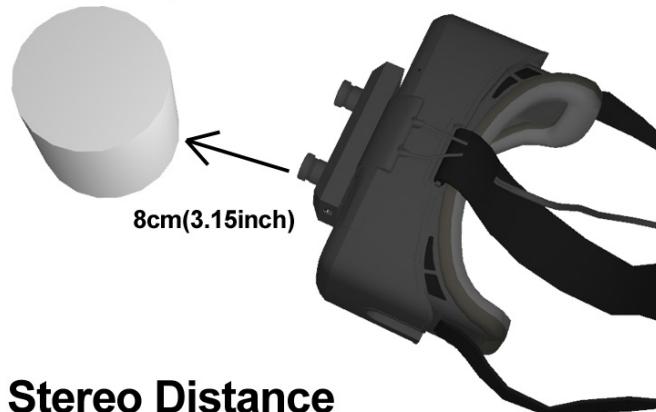


Figure 31: Left: Barrel distortion. Right: Pincushion distortion. Both picture relative to a parallel aligned picture.

#### 4.5.2 Minimum Focus Distance and Stereo Distance

An important constant of the cameras is the shortest distance it is able to focus at. The Ovrvision has a focus distance of 8cm while the human eye has 1.7cm [23]. A related concept which is integral is the stereo distance, that is the distance the cameras overlap and are able to capture the scene from two different angles to create stereo vision and 3D. The Ovrvision has a stereo distance of 30cm.

## Minimum Focusing Distance



## Minimum Stereo Distance

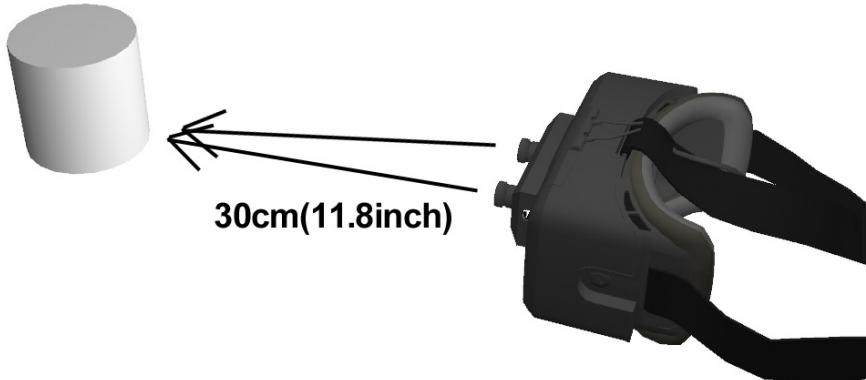


Figure 32: The focus distance of the Ovrvision.

### 4.6 Linear Interpolation

Because the RobotStudio software and the Oculus Software runs on two asynchronous machines, the samples from the two does not fall on the same moments in time. In order to compare the signals, linear interpolation was used to generate the missing data points in the plots from the robot, see Fig. 33 for a graphical explanation. To perform the linear interpolation excel was used. If  $(x_1, y_1)$  and  $(x_2, y_2)$  are two known values, then the value  $y$  for some point  $x$  can be found with the following formula[12]:

$$y = y_1 + (x - x_1) \frac{y_2 - y_1}{x_2 - x_1} \quad (3)$$

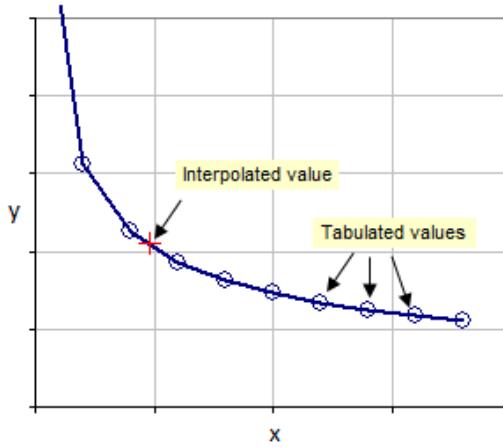


Figure 33: The figure shows the concept of how to generate missing data points with interpolation

In this case there are data set from the log file, defined as  $L[t_s]$  where  $t_s$  is all the sampling values of the log file  $\{t_{s1}, t_{s2}, \dots, t_{sp}\}$ . There are also a data set from the robot, defined as  $R[t_r]$  where  $t_r$  is all the sampling values from the robot  $\{t_{r1}, t_{r2}, \dots, t_{rq}\}$ . In order to analyze the behaviour of the graphs, linear interpolation was used to extend  $R[t_r]$  to include all the time instances of  $t_s$  in  $R$  to make a new dataset  $R * [t_s]$ . The pseudo code for the algorithm to generate the new dataset  $R * [t_s]$  from  $R[t_r]$  and  $t_s$  looks as follows:

```

FOR i=1 TO p
    deltaX = NearestLargest_r(t_s[i]) - NearestSmallest_s(t_s[i]);
    deltaY = NearestLargestR(t_s[i]) - NearestSmallestR(t_s[i]);
    Slope = deltaY/deltaX;
    R*[t_si] = NearestSmallestR(t_s[i]) +
        (t_s[i]-NearestLargest_r(t_s[i]))*Slope;
ENDFOR

```

## 4.7 Performance Definitions

There are mainly two ways to compare and evaluate the performance of the system. The first is the physical test with testing the complete system and get feedback from the operator on how the system feels, and the second is the quantitative analysis approach studying the output data sets. In order to be able to compare the results in an analytical way it is necessary for the robot and simulation to be able to take input from a data set. This is in order to get a repetitive experiment where the results are comparable. There are different ways of comparing the results, one is to study the graphs visually in order to

conclude performance, the other is to define different parameters which could say something about tracking ability and the average time delay. The visual approach can sometimes be better in the evaluation, as it is hard to define parameters that says something about the trade-offs between accuracy and latency, which is a recurring dilemma. Section 4.7.1 and 4.7.2 will define the performance parameters that are utilized.

#### 4.7.1 Area Between Curves

This is one of the methods used to have a measurement of the tracking performance. The idea is that the area between the plot from the Oculus and the plot from the robot gives an indication of how well it tracks. Both accuracy and delay will be indicated by this parameter.

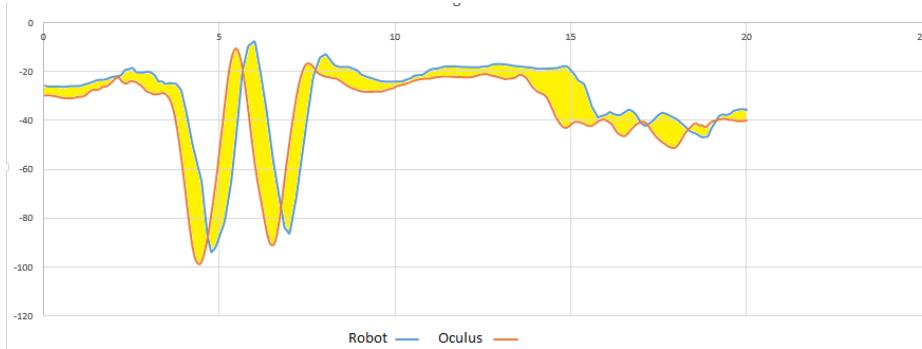


Figure 34: The area between the Oculus Rift plot and robot plot are marked in yellow

This parameter is defined as following:  $V_o[t]$  is a  $6 \times 1$  vector function containing the orientation and position of the Oculus at all time instances.  $V_r[t]$  is a  $6 \times 1$  vector function containing the orientation of the transformed TCP coordinate system of the robotic arm, interpolated to match the time scale of  $V_o[t]$ , see Section 4.8.5.  $\alpha = roll, \phi = pitch, \varphi = yaw$

$$V_o[t_i] = [x_o[t_i], y_o[t_i], z_o[t_i], \alpha_o[t_i], \phi_o[t_i], \varphi_o[t_i]] \quad (4)$$

$$V_r[t_i] = [x_r[t_i], y_r[t_i], z_r[t_i], \alpha_r[t_i], \phi_r[t_i], \varphi_r[t_i]] \quad (5)$$

Using the Trapezoidal rule [32] the difference vector  $\delta = [\delta_1, \delta_2, \delta_3, \delta_4, \delta_5, \delta_6]$  is defined. This vector contains the area between the curves of the vector functions. The numerical integral is trivial to do because the linear interpolation has been done to  $V_r[t]$ . The step size of  $V_o[t]$  and  $V_r[t]$  is defined as  $h = t_{n+1} - t_n$ .

$$\delta = \int_0^t |V_o(t) - V_r(t)| dt \approx \frac{h}{2} \sum_{i=1}^N |V_o[t_{i+1}] - V_r[t_{i+1}]| + |V_o[t_i] - V_r[t_i]| \quad (6)$$

Finally the performance parameter  $\Psi$  is defined as the sum of the components of  $\delta$ :

$$\Psi = \sum_{i=1}^6 \delta_i \quad (7)$$

If there is perfect tracking  $\Psi$  will become zero. If there is a big time delay, inaccuracy in the tracking or both, this integral will have a value grater than zero and give us a relative indicator of the performance of the tracking.

#### 4.7.2 Finding the Delay

There is also an advantage to have a value for the time delay between the two signals. In order to measure this, the Matlab function *finddelay*[16] is used. This function can be found in the *Signal Processing Toolbox* of Matlab. This function uses the *xcorr* function to find the cross-correlation between each pair of signals at all possible lags. The signals do not need to be exact copies of each other, but need sufficiently correlation between them in order to get a meaningful value. The value returned gives an indication of the delay between the two correlated signals. This coefficient is defined by summing up the delay coefficient of each vector component in the orientation. This value is given the symbol  $\tau$ .

$$V_o[t_i] = [x_o[t_i], y_o[t_i], z_o[t_i], \alpha_o[t_i], \phi_o[t_i], \varphi_o[t_i]] \quad (8)$$

$$V_r[t_i] = [x_r[t_i], y_r[t_i], z_r[t_i], \alpha_r[t_i], \phi_r[t_i], \varphi_r[t_i]] \quad (9)$$

$$\tau = \tau_x + \tau_y + \tau_z + \tau_\alpha + \tau_\phi + \tau_\varphi \quad (10)$$

Where

$$\begin{aligned} \tau_x &= \text{finddelay}(x_o, x_r) \\ \tau_y &= \text{finddelay}(y_o, y_r) \\ \tau_z &= \text{finddelay}(z_o, z_r) \\ \tau_\alpha &= \text{finddelay}(\alpha_o, \alpha_r) \\ \tau_\phi &= \text{finddelay}(\phi_o, \phi_r) \\ \tau_\varphi &= \text{finddelay}(\varphi_o, \varphi_r) \end{aligned}$$

When using  $D = \text{finddelay}(X, Y)$ , where  $X$  and  $Y$  are vectors containing the two data sets, and  $X$  serves as the reference vector. If  $Y$  is delayed with respect to  $X$ ,  $D$  is positive.

## 4.8 Coordinate Systems

### 4.8.1 Tool Center Point

The tool center point (TCP) defines the center of a given tool. All movement that the robot will do will be relative to the TCP, which means that it is that point that will enter through the different point targets for the robot movement. In most application the TCP will be defined on the active tool, e.g. the center of a gripper or on the end of a pointer. However, it is also possible to have a stationary TCP which means that all movement will be relative to that point in space, e.g. a stationary tool.

### 4.8.2 Base Frame

The base frame defines the coordinate system which has its origin at the intersection between the first axis and the robots mounting surface. It is defined so that the z-axis will be perpendicular to the surface, the x-axis straight forward and the y-axis points to the left, see Fig. 35.

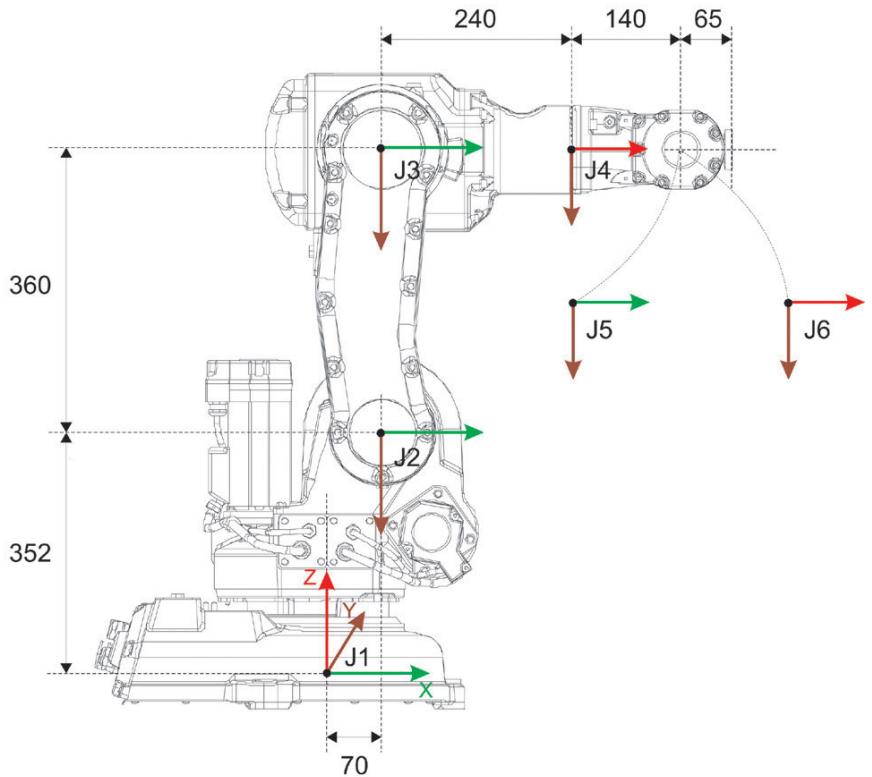


Figure 35: illustration of the robots coordinate frames [20].

#### 4.8.3 Wrist Frame

The wrist frame is defined to be fixed at the center of the mounting flange. The z-axis of this frame coincides with the sixth axis of the manipulator. This frame is also the same as *tool0* in the program, which is the default tool frame, see Fig. 35 and J6.

#### 4.8.4 Tool Frame

To define a new TCP a tool frame is required. The tool frame is the coordinate system for the tool and is defined to have its origin at the TCP. It is possible to obtain information about the direction of the manipulators movement from the tool frame. The tool frame is defined by *tool0* in this application, which means

the tool frame is located at the tool flange.

#### 4.8.5 Coordinate Transformation From Oculus Rift to Robot TCP

The Oculus Rift frame and the robot frame are both rotated with respect to each other, in order to visualize this rotation one can study Fig. 36. In addition to this rotation between the coordinate frames there is also an offset bound to the two coordinate frames.

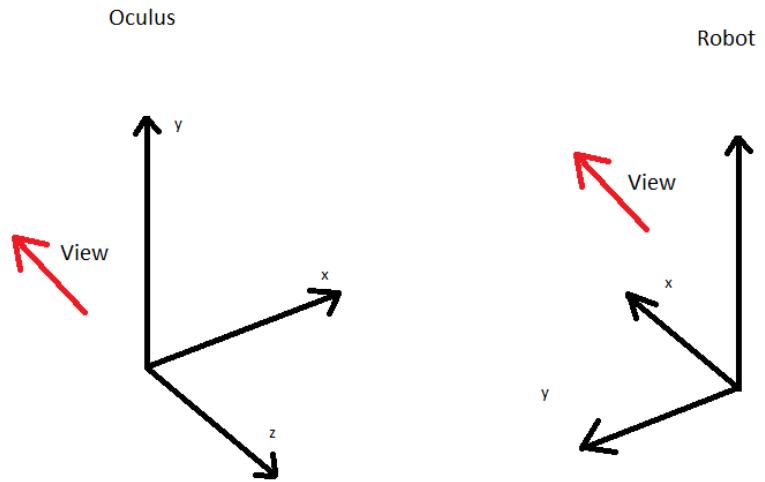


Figure 36: This figure shows how the two coordinate systems are rotated with respect to each other

In order to define the offset one needs to consider when adding the offset when plotting, some parameters are defined:

The values used are the measured values from the data set used when plotting the results. The symbols defining the offset of the Oculus Rift are as follows in  $x$ ,  $y$  and  $z$ -direction.

$$OOx \quad (11)$$

$$OOy \quad (12)$$

$$OOz \quad (13)$$

This offset is the Oculus initial position with respect to the origin. The reason why no offset is needed for angles is because there is a one to one relation between the angle in the Oculus framework and the robot. But that is not the case for translatory movements. The initial position of the Oculus Rift changes for each

time it is used due to the fact that the operator is positioned slightly different from one time to the next. This is why the change in position but the absolute angle is sent. Fig. 37 shows the initial position of the Oculus with respect to the origin.

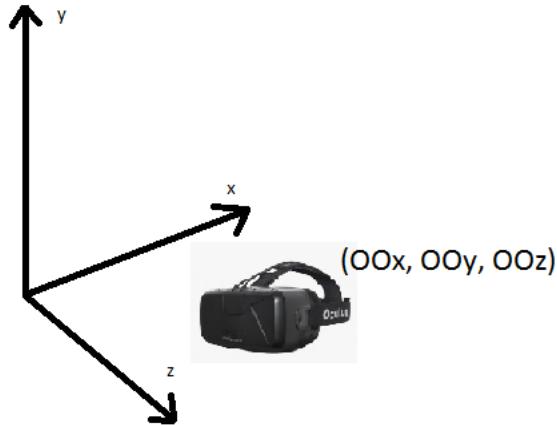


Figure 37: This figure shows how the initial position of the Oculus Rift ( $OOx, OOy, OOz$ ) is defined.

The same offset occurs on the robot side. The robots position is defined as the distance from the base frame. Fig. 38 and 39 shows this offset. The symbols used when defining this offset are the following:

$$ROx \quad (14)$$

$$ROy \quad (15)$$

$$ROz \quad (16)$$

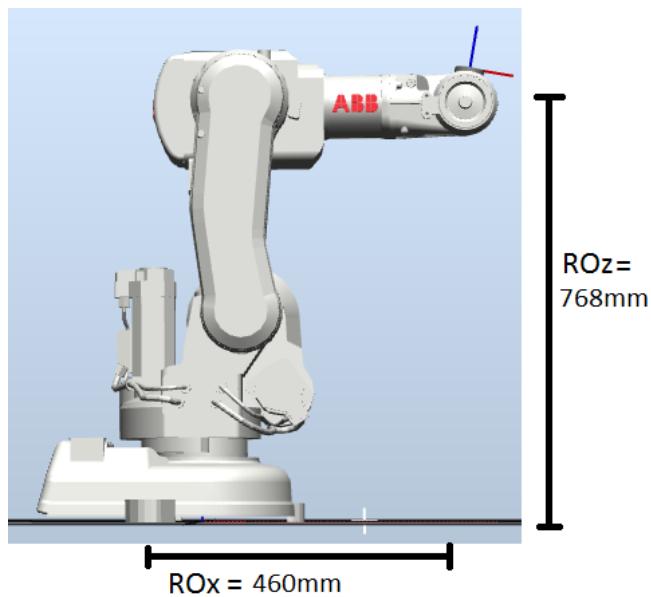


Figure 38: This figure shows the offset the TCP has with respect to the origin of the robot coordinate system in x-direction and z-direction.

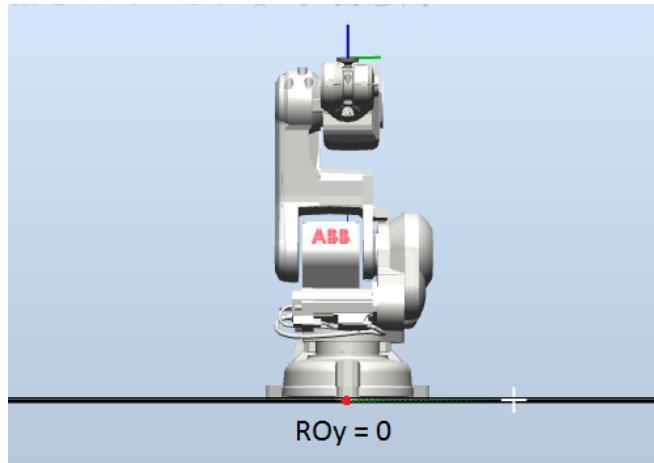


Figure 39: This figure shows the offset the TCP has with respect to the origin of the robot coordinate system in y-direction.

These values measures the distance from origin of the base frame which can be seen in Fig. 40.

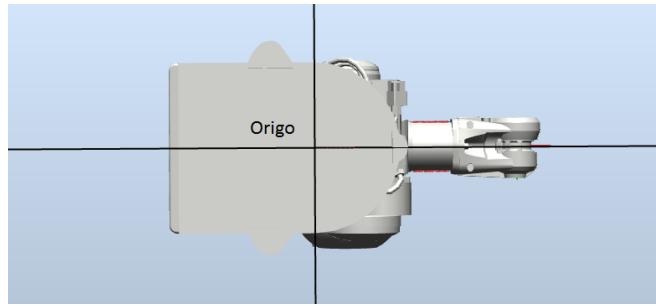


Figure 40: The origin of the base frame.

The final transformation that one needs to perform on the robot coordinate system in order to align the plots can be calculated by studying Fig. 36  $Rx*$  refers to the transformed value of the robot coordinate system while  $Rx$  is the old value.

$$Rx* = -Rx + ROy + OOx \quad (17)$$

$$Ry* = Ry + OOy - ROz \quad (18)$$

$$Rz* = -Rz + OOz + ROx \quad (19)$$

## 4.9 Performance Parameters

### 4.9.1 zonedata

There are two ways that a point can be terminated, that is in either a stop point, or a fly-by point, see Fig. 42 for a graphical explanation of the fly-by points. The stop point means that the TCP must stop at the assigned target, it is possible to make custom *stoppointdata* to address what accuracy it must have.

Fly-by points on the other hand is defined by *zonedata*, which defines how close to the pre-programmed point the tool must be before it can start to change its path to the next point. This has the added bonus of never stopping the motion as it is constantly changing its path on the fly. As can be seen in Fig. 41 the points  $p40$  and  $p30$  are fly-by point where the tool does not need to hit the exact point, but it can have a smoother transition towards the next point compared to the fine points  $p10$  and  $p30$ .

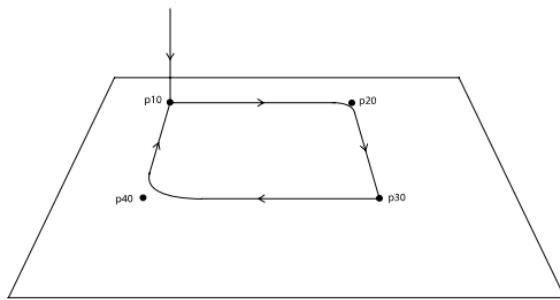


Figure 41: This figure illustrates the principle with *zonedata*.

A fly-by point has two different zones for each target. These are divided into the zone for the TCP path and the extended zone. The TCP path describes the zone for which the tool can start to reoriented. The tool will be oriented in such a fashion that when it leaves this zone it will be oriented in the same way as if the point was defined by a stop point and not a fly-by point. The extended zone defines the outer shell of the *zonedata* which defines the zone where a corner path will be generated when breached.

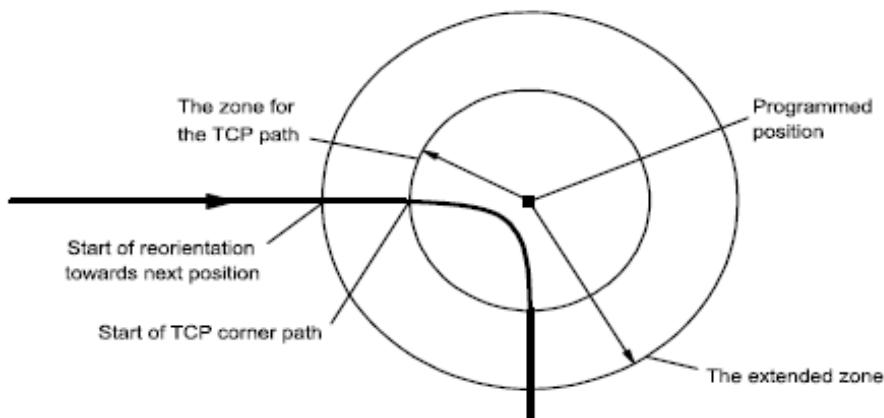


Figure 42: illustration of the *zonedata*[6]

#### 4.9.2 Future Estimation with OVR SDK

The OVR SDK poses a function to estimate future head poses using the built in function:

```
trackstate = ovrHmd_GetTrackingState(HMD,0.0);
```

This function takes two input arguments. The first, HMD, which is the head mount display object one needs to initialize to access the Oculus data. The second input argument gives the time horizon for the estimation. A input of current time or earlier(e.g 0.0) returns the current state of the HMD (the current position of the Oculus Rift) and a input of currenttime+0.03 will return a estimate of the state of the HMD 0.03 seconds into the future.

This blogpost from one of the Oculus cooperators [19] suggests that it is possible to predict head tracking for 20ms-40ms for a human head. This number could possibly go slightly higher for a operator moving in a slow and steady movement.

#### 4.9.3 Prefetch Time

Prefetch time is a parameter that is possible to adjust as a RobotStudio system parameter. It affects the point in time at which the controller starts to plan a motion that goes through a corner zone. If this time is to short, the corner zone becomes a fine point and the error "corner path failure is generated". This is a recurring problem when using a *zonedata* that is different from *fine*. Increasing the *prefetch time* will result in a higher CPU load. It will also result in less corner path failures, and it might be more optimal for the arm to have a continuous movement through corner zones rather than stopping at each point. However, it can be problematic for the CPU load if the corner zones are placed to close together, and therefore create higher latency. The *prefetch time* value can be set somewhere between 0 and 10 seconds.

#### 4.9.4 Path Resolution

This parameter corresponds to the distance between the sampling points. In our application there is a low distance which means that this parameter probably also should be set low. A high *path resolution* will lead to a decreased resolution because it corresponds to increasing the distance between points. Decreasing this parameter will also lead to a higher CPU load. According to the manual[18] tuning this parameter might increase the performance if the program has multi-threading with computationally high demands, a high number of simultaneously controlled axes or if the distance between closely programmed points is decreased. As the program has all of these suggestions it is worthwhile to try to optimize this parameter.

#### 4.9.5 WaitTime

The while-loop located in the *Main* procedure of the move thread needs a wait timer between each move instruction if *zonedata*  $\neq$  *fine* is used. This wait time can be adjusted to increase the number of commands sent to the controller, however, if the *WaitTime* parameter is set too low when fly-by points are used, an error message called "*Correct regain impossible*" occurs. This error message

occurs due to too many close points with corner zones. It has been experimentally proven that to ensure that this is avoided a minimum *WaitTime* of 0.1 seconds must be in place when using fly-by points. If the *zonedata* is set to *fine* then his added latency can be avoided.

A simplified look at the main loop and the *WaitTime* command looks as follows:

```

WHILE TRUE DO
    UpdateDestination;
    LookForChange;
    IF dataChanged AND isReachable(point, tool0, wobj0) THEN
        MoveRobot;
        WriteToFile;
    ENDIF
    WaitTime 0.1;
ENDWHILE

```

#### 4.9.6 MoveAbsJ and MoveL

The RobotStudio algorithm used for movement is the *MoveL* algorithm (Move Linear). This algorithm solves the inverse kinematic problem and gives commands to each joint based on the input position and orientation of the TCP. The possibility of doing the inverse kinematics on the client and only send joint movement was discussed. By using *MoveAbsJ* (Move Joint) which is just a simple servo command to each joint one could find out if this method had potential to run any faster. If there is to be any value of using an inverse kinematics solver on the client side then the latencies must satisfy Eq. 20.

$$\tau_{MoveAbsJ} + \tau_{InverseKinematics} + \tau_{communication} < \tau_{MoveL} \quad (20)$$

#### 4.9.7 Process Update Time

This variable gives an indication of how often the process calculates the path information. This information is used for path following. Decreasing the *process update time* increases the CPU load but would also improve the accuracy. When the robot is moving slowly it is not crucial to adjust this parameter as the controller has time to calculate during motion. In our application where the points are closely together and often exposed to rapid movements this parameter would be crucial to adjust.

#### 4.9.8 Queue Time

This is a parameter that makes the system more tolerant to uneven loads on the CPU and indicates how long instructions should stay in the queue. The *queue time* will be set to the closest even multiple of the dynamic resolution which is a system parameter that should not be changed which is set to 1 by default. When increasing the *queue time* the robot reacts more slowly which could potentially have a large impact on the system performance[41].

#### 4.9.9 Event Preset Time

Event Preset Time is a parameter that delays the robot by a set amount of time. This should therefore be set to zero in any case for our application.

#### 4.10 Multithreading

Multithreading allows an application to execute several operations at the same time. The main reason that multithreading is used in this application is that it increases responsiveness when the program needs to wait for input. For a single thread program the program flow will stop while it waits for the input, while on a multithreading system only a single thread has to wait for the input as the other threads continue their execution. This makes it ideal to use when having to work with networks as it can have a thread dedicated to this. The first drawback of multithreading is that a single thread can cause the whole program to crash if it encounters an error. The second drawback is the need for synchronization to maintain an intuitive program flow as well as avoiding race conditions. Fig. 43 shows an example of how the FlexPendant GUI operates with several tasks.

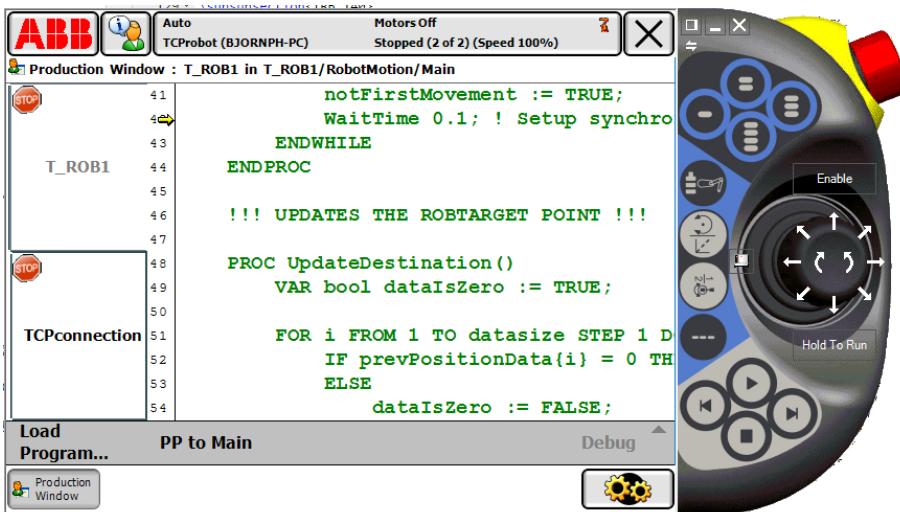


Figure 43: Image of the *Production Window* used for multithreading in automatic mode.

#### 4.11 The Rotation Matrix

To perform rotations on points in Euclidean space a rotation matrix is utilized. What the rotation matrix accomplishes is that it specifies the orientation of one frame with respect to another. In order to get the coordinate vectors of the first frame with respect to frame 0 in the three dimensional space a 3x3 rotation

matrix is required.

$$_1^0R = \begin{bmatrix} 0 & x & 0 \\ 0 & y & 0 \\ 0 & z & 1 \end{bmatrix} \quad (21)$$

The relation between several frames is given by Eq. 22

$$_i^0R = (_1^0R) \cdot \dots \cdot (^{i-1}_iR) \quad (22)$$

To perform a rotation about of the axes of a coordinate system an elemental rotation will be used. This gives three different basic rotations which will rotate a vector by theta around either x,y or z, see Eq. 23 - 25.

$$R_x(\theta) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos(\theta) & -\sin(\theta) \\ 0 & \sin(\theta) & \cos(\theta) \end{bmatrix} \quad (23)$$

$$R_y(\theta) = \begin{bmatrix} \cos(\theta) & 0 & \sin(\theta) \\ 0 & 1 & 0 \\ -\sin(\theta) & 0 & \cos(\theta) \end{bmatrix} \quad (24)$$

$$R_z(\theta) = \begin{bmatrix} \cos(\theta) & -\sin(\theta) & 0 \\ \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad (25)$$

## 4.12 Forward Kinematics

In robotics, forward kinematics refers to the process that computes the position and orientation by using kinematic equations and given values for the joint angles of the manipulator. In order to get the end-effector vector a transformation matrix is applied. The transformation matrix is defined by the joint and link matrices. One way to find the transformation matrix is to apply the Denavit-Hartenberg convention, or in this case, the improved Denavit-Hartenberg convention which is defined by Eq. 28.

$$c_i = \cos(\theta_i) \quad (26)$$

$$s_i = \sin(\theta_i) \quad (27)$$

$$_i^{i-1}T = \begin{bmatrix} c_i & -s_i & 0 & a_{i-1} \\ s_i c \alpha_{i-1} & c_i c \alpha_{i-1} & -s \alpha_{i-a} & -s \alpha_{i-a} d_i \\ s_i s \alpha_{i-1} & c_i s \alpha_{i-1} & c \alpha_{i-1} & c \alpha_{i-1} d_i \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (28)$$

From Fig. 44 we can derive the improved Denavit-Hartenberg parameters, used by Craig[3], see Tab. 2.

Table 2: Improved Denavit-Hartenberg parameters.

i	$\alpha_{i-1}$	$a_{i-1}$	$d_i$	$\theta_i$
1	0	0	$d_1$	$\theta_1$
2	90	$a_1$	0	$\theta_2$
3	0	$a_2$	0	$\theta_3$
4	90	0	$d_4$	$\theta_4$
5	-90	0	0	$\theta_5$
6	90	0	0	$\theta_6$

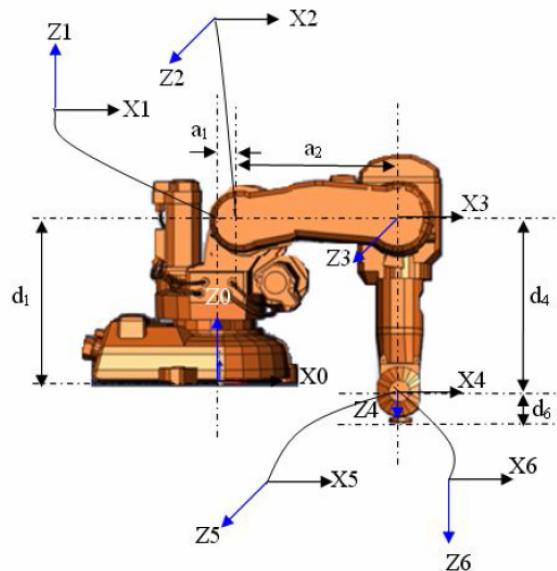


Figure 44: Kinematic parameters of the ABB IRB 140 and frame assignment.

Using Eq. 28 we can derive the six transformation matrices needed to locate the

end link.

$${}^0_1T = \begin{bmatrix} c_1 & -s_1 & 0 & 0 \\ s_1 & c_1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}, {}^1_2T = \begin{bmatrix} c_2 & -s_2 & 0 & a_1 \\ 0 & 0 & -1 & 0 \\ s_2 & c_2 & 0 & 1 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (29)$$

$${}^2_3T = \begin{bmatrix} c_3 & -s_3 & 0 & a_2 \\ s_3 & c_3 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}, {}^3_4T = \begin{bmatrix} c_4 & -s_4 & 0 & 0 \\ 0 & 0 & -1 & -d_4 \\ s_4 & c_4 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (30)$$

$${}^4_5T = \begin{bmatrix} c_5 & -s_5 & 0 & 0 \\ 0 & 0 & -1 & -d_4 \\ s_4 & c_4 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}, {}^5_6T = \begin{bmatrix} c_6 & -s_6 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ s_6 & c_6 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (31)$$

Using Eq. 29-31 we can find that the transformation matrix (Eq. 33) that locates the end effector with respect to the base frame. The intermediate calculations can be seen in Appendix A.

$${}^0_6T = {}^1_0T \cdot {}^2_1T \cdot {}^3_2T \cdot {}^4_3T \cdot {}^5_4T \cdot {}^6_5T \quad (32)$$

$${}^0_6T = \begin{bmatrix} r_{11} & r_{12} & r_{13} & P_{xw} \\ r_{21} & r_{22} & r_{23} & P_{yw} \\ r_{31} & r_{32} & r_{33} & P_{zw} \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (33)$$

## 4.13 Inverse Kinematics

Inverse kinematics describes solutions on how to find joint angles corresponding to the desired orientation and position of the *end-effector*. There are several different approaches to solve the inverse kinematics problem, this paper chooses to experiment with the partial geometric and algebraic method. This method is restricted by the fact that it solves for a maximum of 6 DOF (degrees of freedom). The ABB IRB 140 has only six joints, meaning it is within the constraints of the algorithm.

### 4.13.1 Finding $\theta_1$ , $\theta_2$ and $\theta_3$ , Geometric Approach

Pieper's condition[1] is the sufficient condition for solving a six-axis problem. The kinematic manipulator must have three consecutive revolute axes that intersect at a common point. To solve the inverse kinematic problem it is normal to use kinematic decoupling to split up the problem into two smaller sub-problems (See Spong[33]). You can then solve for the position and orientation separately. Since position is only defined by the first three joint angles it is possible to find them

through a geometric approach. In order to make it simpler the robot can be modelled as an elbow manipulator for the time being, see Fig. 45.

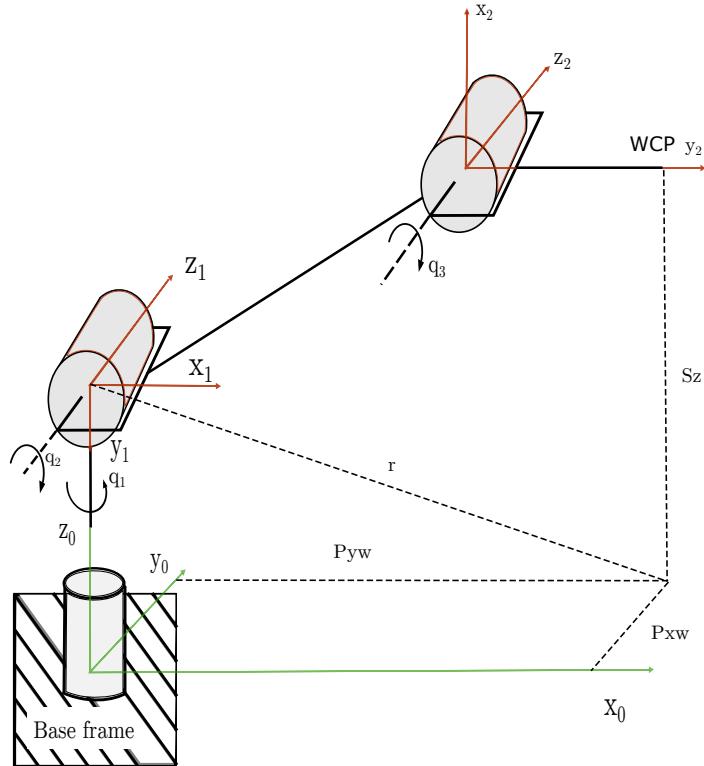


Figure 45: Kinematic decoupling displaying the elbow of the manipulator.

The wrist center point (WCP) is defined by  $P_w$  which consists of  $P_{xw}, P_{yw}$  and  $P_{zw}$ . In order to find  $\theta_1$  the projection of the wrist center onto the  $x_0-y_0$  plane is used to find it through geometry.

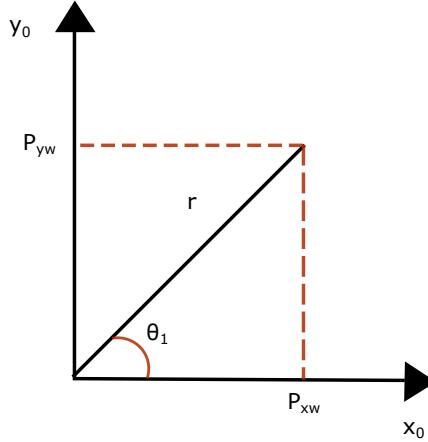


Figure 46: The projection of the WCP onto the  $x_0 - y_0$  plane.

Using Fig. 46 we can see the trigonometric relation for  $\theta_1$ .

$$\tan(\theta_1) = \frac{P_{yw}}{P_{xw}} \quad (34)$$

$$\theta_1 = \tan^{-1} \left( \frac{P_{yw}}{P_{xw}} \right) = \text{atan2}(P_{yw}, P_{xw}) \quad (35)$$

Or

$$\theta_1 = \pi + \text{atan2}(P_{yw}, P_{xw}) = \text{atan2}(-P_{yw}, -P_{xw}) \quad (36)$$

Note that *atan2* describes a function to solve arc tangent with two arguments.

In order to find  $\theta_2$  and  $\theta_3$  a projection of the plane formed by the second and the third link is utilized, see Fig. 47. To be able to find  $\theta_3$  it is needed to use the law of cosines, and the shoulder offset defined by the D-H parameters,  $a_1$  and  $d_1$  are introduced.

$$\cos(\theta_3) = \frac{r^2 + s^2 - a_2^2 - d_4^2}{2a_2d_4}$$

$$\cos(\theta_3) = \frac{(P_{xw} + a_1 \cos(\theta_1))^2 + (P_{yw} + a_1 \sin(\theta_1))^2 + (P_{zw} - d_1)^2 - a_2^2 - d_4^2}{2a_2d_4}$$

$$\cos(\theta_3) = D$$

The simplest way to get  $\theta_3$  would be to take the arccosine of  $D$ , however, this solution does not take into account the "elbow up" and "elbow down" scenarios. In order to recover both of these solutions it is better if it is transformed to another trigonometric function.

$$\sin^2(\theta_3) + \cos^2(\theta_3) = 1 \quad (37)$$

$$\sin^2(\theta_3) + D^2 = 1 \quad (38)$$

$$\sin(\theta_3) = \pm\sqrt{1 - D^2} \quad (39)$$

Thus by using this relation it is possible to get a better solution for  $\theta_3$ .

$$\tan(\theta_3) = \frac{\sin(\theta_3)}{\cos(\theta_3)} = \frac{\pm\sqrt{1 - D^2}}{D} \quad (40)$$

$$\theta_3 = \tan^{-1}\left(\frac{\pm\sqrt{1 - D^2}}{D}\right) \quad (41)$$

$$\theta_3 = \text{atan2}(\pm\sqrt{1 - D^2}, D) \quad (42)$$

The "elbow-up" and "elbow-down" scenarios are recovered by choosing the positive or negative sign respectively.

By continuing to follow the approach set out by Spong[33] we can find  $\theta_2$  in a similar fashion.

$$\theta_2 = \text{atan2}(s, r) - \text{atan}(d_4 \sin(\theta_3), a_2 + d_4 \cos(\theta_3)) \quad (43)$$

$$\begin{aligned} \theta_2 = & \text{atan2}\left(P_{zw} - d1, \sqrt{(P_{xw} + a_1 \cos(\theta_1))^2 + (P_{yw} + a_1 \sin(\theta_1))^2}\right) - \\ & \text{atan2}\left(d_4 \sin(\theta_3), a_2 + d_4 \cos(\theta_3)\right) \end{aligned} \quad (44)$$

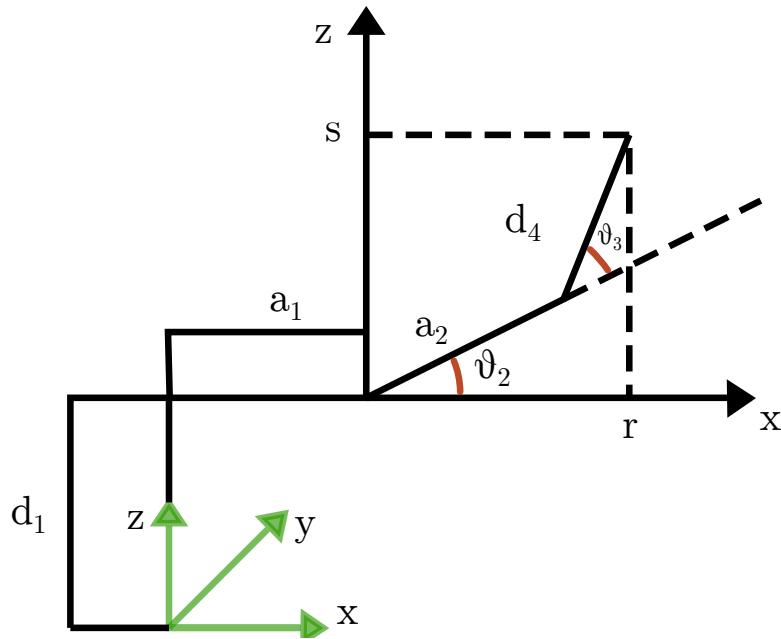


Figure 47: A projection of the plane formed by link2 and link3.

#### 4.13.2 Finding $\theta_4, \theta_5$ and $\theta_6$ , Algebraic Approach

The second sub-problem of finding the inverse kinematics is getting the remaining three joint angles corresponding to the orientation of the TCP. Due to the last three joint angles representing a spherical wrist it is possible to look at this as finding a set of Euler angles corresponding to a given rotation matrix, R. Looking at Fig. 48 it can be seen that all changes in orientation can be represented as a combination of elemental rotations given by  $R_z(\alpha)R_x(\beta)R_z(\gamma)$ .

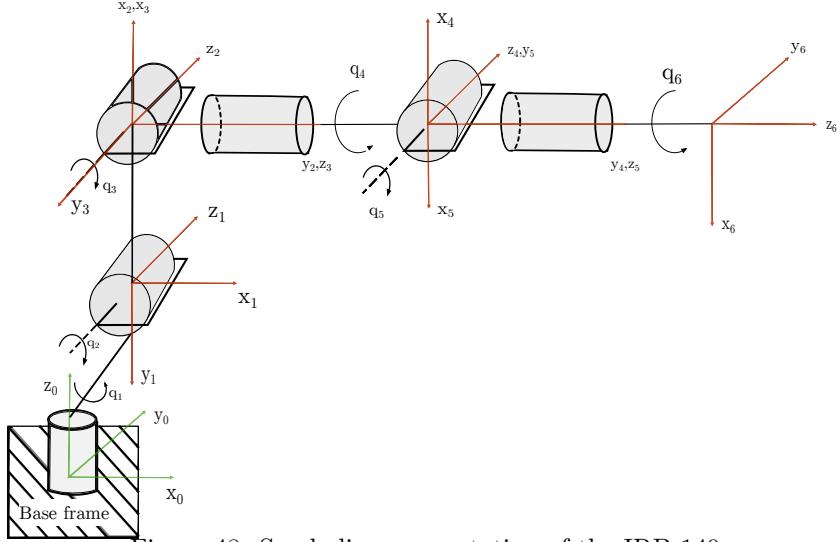


Figure 48: Symbolic representation of the IRB 140

The equation needed to compute the last remaining joint angles is straightforward from the definition of the rotation matrices.

$$R = {}_0^3R {}_3^6R \quad (45)$$

$${}_3^6R = {}_0^3R^{-1}R \quad (46)$$

$R$  is known as the rotation matrix of the TCP, while  ${}_0^3R$  can be found as  $\theta_1$ ,  $\theta_2$  and  $\theta_3$  has been found.  ${}_3^6R$  on the other hand is a matrix which is a function of  $\theta_4$ ,  $\theta_5$  and  $\theta_6$ . Since both  ${}_0^3R$  and  $R$  is known then it can be expressed as in Eq. 47.

$$({}_0^3R)^{-1}R = \begin{bmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \\ b_{31} & b_{32} & b_{33} \end{bmatrix} = B \quad (47)$$

In order to find  ${}_3^6R$  Eq. 22 is used.

$${}_3^6R = {}_0^3R \cdot {}_3^4R \cdot {}_4^5R \cdot {}_5^6R = \begin{bmatrix} c_4c_5c_6 - s_4s_6 & -c_4c_5s_6 - s_4c_6 & c_4s_6 \\ s_4c_5c_6 + c_4s_6 & -s_4c_5s_6 + c_4c_6 & s_4s_5 \\ s_5c_6 & s_5s_6 & c_5 \end{bmatrix} \quad (48)$$

$$\begin{bmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \\ b_{31} & b_{32} & b_{33} \end{bmatrix} = \begin{bmatrix} c_4c_5c_6 - s_4s_6 & -c_4c_5s_6 - s_4c_6 & c_4s_6 \\ s_4c_5c_6 + c_4s_6 & -s_4c_5s_6 + c_4c_6 & s_4s_5 \\ s_5c_6 & s_5s_6 & c_5 \end{bmatrix} \quad (49)$$

Thus there are 9 equations with 9 unknown values containing the three unknown

joint angles.

$$b_{11} = c_4 c_5 c_6 - s_4 s_6 \quad (50)$$

$$b_{12} = -c_4 c_5 s_6 - s_4 c_6 \quad (51)$$

$$b_{13} = c_4 s_5 \quad (52)$$

$$b_{21} = s_4 c_5 c_6 + c_4 s_6 \quad (53)$$

$$b_{22} = -s_4 c_5 s_6 + c_4 c_6 \quad (54)$$

$$b_{23} = s_4 s_5 \quad (55)$$

$$b_{31} = -s_5 c_6 \quad (56)$$

$$b_{32} = s_5 s_6 \quad (57)$$

$$b_{33} = c_5 \quad (58)$$

Now using these equations the solution for  $\theta_4$ ,  $\theta_5$  and  $\theta_6$  can be found.

$$\tan(\theta_4) = \frac{b_{23}}{b_{13}} \rightarrow \theta_4 = \text{atan2}(b_{23}, b_{13}) \quad (59)$$

$$\tan(\theta_6) = -\frac{b_{32}}{b_{31}} \rightarrow \theta_6 = \text{atan2}(b_{32}, -b_{31}) \quad (60)$$

$$\tan(\theta_5) = \frac{b_{13}}{b_{33} c_4} \rightarrow \theta_5 = \text{atan2}(b_{13}, b_{33} c_4) \quad (61)$$

## 4.14 Singularities

A singularity represents loss of rank in the system Jacobian which is defined by Eq. 62 and can be derived from the forward kinematics. There are several reasons for why singular configurations are unwanted. As explained by Spong[33], the most noteworthy is that it can cause unbounded velocities or make it so that the inverse kinematics problem has an infinite amount of solutions or none. This can cause problems for this application as it is hard to calculate further trajectories to escape the proximity to a singularity.

$$\dot{\mathbf{X}} = J(\mathbf{q})\dot{\mathbf{q}} \quad (62)$$

For a manipulator with a spherical wrist there are mainly two types of singularities, namely wrist and arm singularities. These singularities corresponds to the motion of either the wrist or the arm. The most common singularities are those where two axes are collinear. These also include the singularities at the boundary of the reachable workspace. This explains why the robot does not start with all the joint axes at zero degrees, see Fig. 49 for an illustration of the spherical wrist singularity. The arm singularity is defined by the TCP crossing axis 1, see Fig. 50 for a graphical explanation. This singularity does not occur often due to the position it would require the operator to be in.

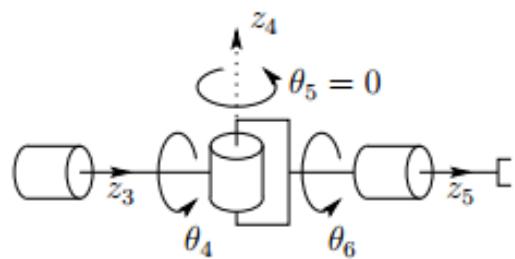


Figure 49: Spherical wrist singularity[33].

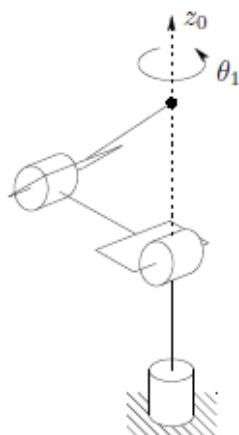


Figure 50: Illustration of the elbow singularity[33].

## 4.15 Sources of Delay

The most important part of the user experience is the delay between the head movement and the visual feedback. This will be defined as total system latency (TSL), see Eq. 63, and gives an indication towards how good the system is operating. It is generally accepted by the international community that a latency of 60 ms[19] is the maximum threshold for an optimal experience in virtual reality, however, it has been mostly applied towards pure software application and not physical systems. This makes the constraint not very realistic in the case of real time control of robots with the current system.

$$TSL = latency_{rapid} + latency_{c++} + latency_{robot} + latency_{video} \quad (63)$$

Beneath the TSL there is also latency for the substructures which are the robot, the RAPID server, the C++ client and the video handler. During the project much of the TSL resided with the camera setup. This was due to firmware not being optimized for FPV usage, and could add as much as 200 ms to the TSL. By researching into more suitable camera modules the current setup has a worst case latency at 40 ms. Without this improvement reaching the goal of a reasonable latency would get almost impossible. The other lesser substructure is the communication between the client and the server. As long as the structural integrity of the packages that are sent and the synchronization within the programs perform as intended the only latency in the networking should be the latency induced by distance.

In the RAPID server application the latency should be confined to the movement algorithm, the synchronization between the tasks and perhaps the log file writer. There are some other functionality, but their complexity is low and should not require much of the CPU.

The robot itself contributes to the TSL due to transmission delay and slow response of some joints. Simple testing with manually controlling the robot through the FlexPendant joystick proves that the transmission delay between the controller and the robot is not negligible. The specifics of this delay is not known to the public, but it is possible to spot with quantitative analysis. As can be seen in section 3.3.6 that the speed of the first three joints are much lower than that of the wrist joints.

The client application does not run any computationally complex algorithms and can therefore be excluded as a big source of latency. The Oculus sensors with update rate of 60Hz for the position tracking and 1000Hz for orientation tracking does not pose significant delay.

## 5 Literary Search

Due to the fact that this application was only recently made possible due to the advances of Oculus Rift DK2 there is a very limited amount of similar projects that has been published. Thus only partial problems could be solved by searching for relevant litterature, while the developer forums of ABB and Oculus Rift were helpfull in the other cases.

### 5.1 Inverse Kinematics Literature

A topic that was researched in this thesis was the inverse kinematics problem. In order to choose an appropriate solver and understand it some research into the field was required. One of the reasons this topic was approached was due to the mediocre runtime of the RAPID solver. This made it so that the criteria was real-time inverse kinematics, this is mostly been studied in order to get better computer animation. Several papers describe various algorithms which can solve the problem numerically[36] [34]. The reason for this is that the most effective way to solve the inverse kinematics problem for a 6 DOF manipulator is analytical rather than numerical, and is therefore somewhat more trivial. Several books cover this area, but Spong[33] and Craig[3] were found to most usefull.

### 5.2 Programming with Qt4

Qt4 was used in making the GUI and multithreading in the software. To read up on Qt programming with C++ the following book was used [37]. The book provided good examples on how to use the Qt framework, how to set up multiple threads and how to create the graphical user interface.

### 5.3 RobotStudio Manuals

The RobotStudio manuals were essential when using the ABB robot programming ecosystem. In addition to the forum, these documents was the only source to learn about the ecosystem. The forum could sometimes provide answers to some questions, but the documents provided from ABB was without doubt the most important source to knowledge about the system.

The RAPID instruction manual [40] was the reference manual used the most when coding in rapid and setting up the connection with an external computer to the robot.

When the IRC5 controller with the FlexPendant was used, the IRC5 Manual [39] was an important source to knowledge about how these pieces of hardware functioned and how to use the system. The manual also explains what input you can give the system through pushbuttons and dataports, and how the system would respond to those inputs.

When finding different system parameters and reading on how they could affect the system the manual [41] was the one used.

In the debugging process many poorly explained error messages was given by the system. Often one could find more details about the system parameters in the manual about troubleshooting [38].

## 6 Results

### 6.1 Sources of Delay

Optimizing the performance of the system requires knowledge about the locations of the latency. Looking at Fig. 51 it can be observed that when the robot successfully tracks the movements of the Oculus Rift it has a maximum delay of  $500ms$  when operating with system parameters set to standard and *zonedata fine*.

A key note before presenting the results is that some of the delay is caused by operations outside of the optimal workspace for the robot and can be considered worst case. Out of the approximately  $500ms$  latency from the Oculus to the robot at the worst case scenario, about  $420ms$  has been identified. Most functions of the applications can be considered zero contributors to the TSL (Total System Latency) due to low complexity and load. The RAPID command for moving the robot, *MoveL* proved to be the part contributing the most to delay as it could run as slow as  $209ms$  as seen in Tab. 3.

Table 3: Tests were done with 10 000 samples.

Latency / Source	MoveRobot	Communication	Synchronization wait time
Average [ms]	123	0.85	76
Min [ms]	61	<sup>1</sup> 0	0
Max [ms]	209	5	206

By looking at the scatterplot of the *MoveL* runtime the worst runtimes correlates well to the fast movements shown in Fig. 51. Another result to notice is that the synchronization *WaitTime*, which is defined by the time that the *TCPconnection* thread must wait for synchronization, also has a direct correlation to the *MoveL* runtime. This causes an additional source of latency as it increases the time it takes to accept the next package by that amount. Thus when receiving and acknowledging the motion will already be behind that amount.

---

<sup>1</sup>Lack of resolution causing minimum to be zero.

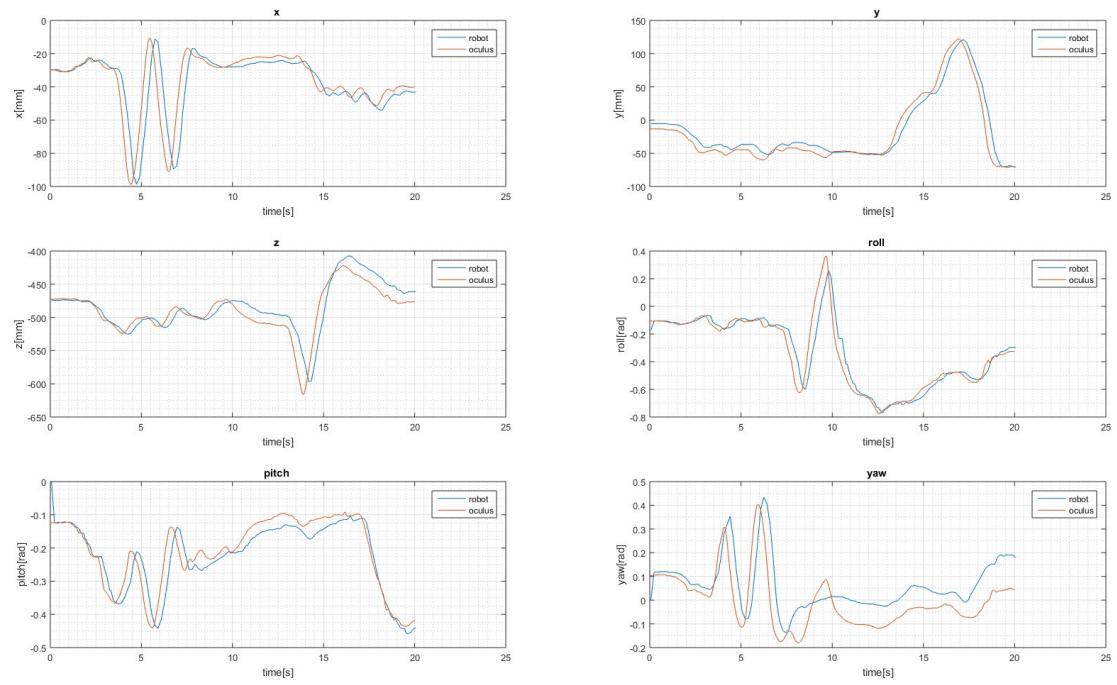


Figure 51: Simulation of robot response to Oculus Rift with system parameters set to standard.

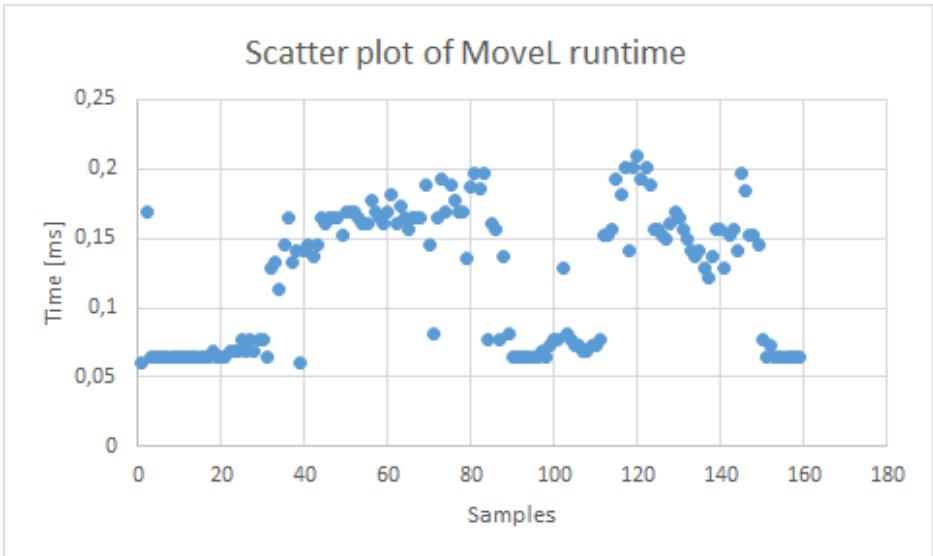


Figure 52: Scatterplot of the *MoveL* runtime.

After summarizing the sources of latency there is still approximately 80 ms that is unaccounted for in the worst case scenario which probably comes from slow respons time of the robot.

## 6.2 Offset Measurement

The total offset between the robot coordinate system and the Oculus Rift in  $x, y, z$  direction is defined as follows:

$$Rx* = -Rx + ROy + OOx \quad (64)$$

$$Ry* = Ry + OOy - ROz \quad (65)$$

$$Rz* = -Rz + OOz + ROx \quad (66)$$

The value for  $ROx$ ,  $ROy$  and  $ROz$  (the Oculus Rift offset in  $x, y, z$  direction) will vary from time to time when you use the Oculus Rift as input, because the Oculus Rift will be placed at slightly different initial positions each time the user puts it on. However, with the dataset as input, the Oculus Rift starting point is the same each time. In addition the robot always has the same initial point as long as it is reset between each time the program starts. This makes us able to calculate the total offset for the standard input dataset with the formula presented in the theory section.

The offset values for the Oculus Rift can be read from the input dataset [30]:

$$OOx = -30,004 \quad (67)$$

$$OOy = -13,2617 \quad (68)$$

$$OOz = -472,657 \quad (69)$$

The offset values of the robot can be read in the simulation in RobotStudio after the robot is initialized, see Fig. 38 and 39:

$$ROx = 460.27 \quad (70)$$

$$ROy = 0 \quad (71)$$

$$ROz = 768.52 \quad (72)$$

This results in the following offsets:

$$Rx* = -Rx + 0 + -30,004 = -Rx - 30,004 \quad (73)$$

$$Ry* = Ry + -13,2617 - 768.52 = -781.78 \quad (74)$$

$$Rz* = -Rz - 472,657 + 460.27 = -12.4 \quad (75)$$

When plotting the raw data from the Oculus and from the Robot, these are the exact transformation that needs to be done to the data from the robot to make it fit the data from the Oculus Rift.

### 6.3 Simulation vs the physical IRB140

When testing the different performance parameters it is convenient to be able to do this in the simulator. It is important that the simulation environment fits well with the physical environment. That is why Fig. 53 shows the response of the robot and the response of the simulator in y-direction with the same input. We can see that there is a good overlap between these, indicating that the simulation is a good enough representation for our testing. It is also interesting to note that the latency is the same for the robot and for the simulation.

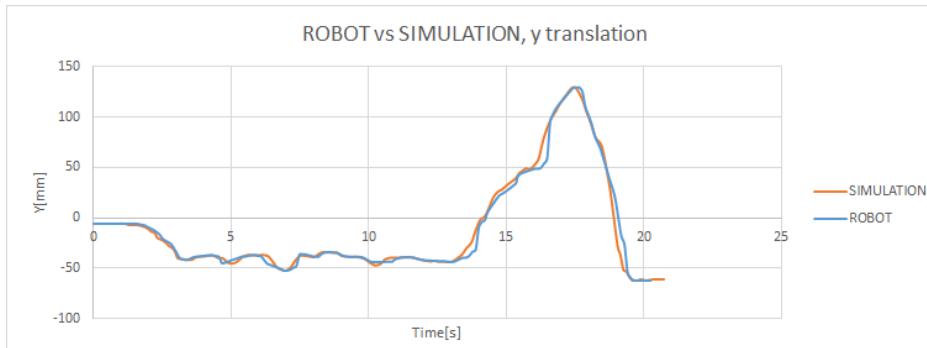


Figure 53: This figure shows the plot of robot trajectory and the simulation trajectory with identical input

## 6.4 tracking ability with different parameters

### 6.4.1 Future Estimation with OVR SDK

In this section the results of the future state estimator that is implemented in the OVR SDK will be presented.

```
ovrTrackingState predictedState = ovrHmd_GetTrackingState(hmd,  
ovr_GetTimeInSeconds() + predictionValue);
```

The *ovrTrackingState* struct is a data structure storing all the information about the current state of the Oculus Rift. The *predictionValue* can be set in the range [0.0, 0.095] which is the time in seconds it is estimated into the future. This can be used to reduce the latency of the final application.

The same input dataset was used in all the plots in this section. Only the *x-motion* (from side to side) is used in the plots for the sake of simplicity, but plots from the other axis showed the same basic behavior. The first plot shows the measured data at the current time and an estimate 40ms into the future. In the interval [0, 7] seconds, the test person moves the head in normal natural motions as a human normally moves the head. from [7, 10] seconds the test person moves the head as fast as possible back and forth.

Fig. 54 shows a measurement of an estimation 40ms into the future. As we can see from the graphs it tracks well from [0, 7] seconds and also does a good job in the interval [7, 10].

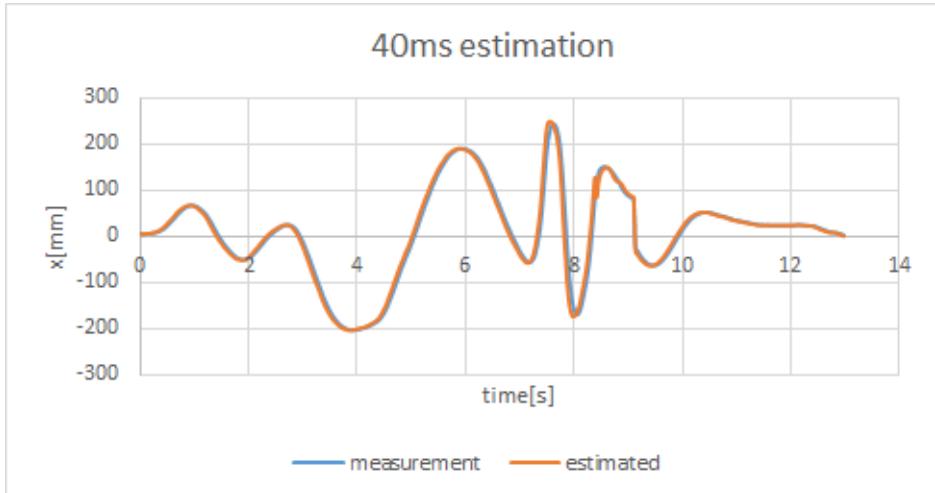


Figure 54: 40ms vs the directly measured data

In Fig. 55 the estimation for the maximum case is studied. That is the estimation parameter set to 95ms. In the first plot the first part of the graph [0, 7] is studied. In this part the operator moves at normal speed. As we can see from the plots, the estimator works well and is able to make a good estimation 95ms into the future.

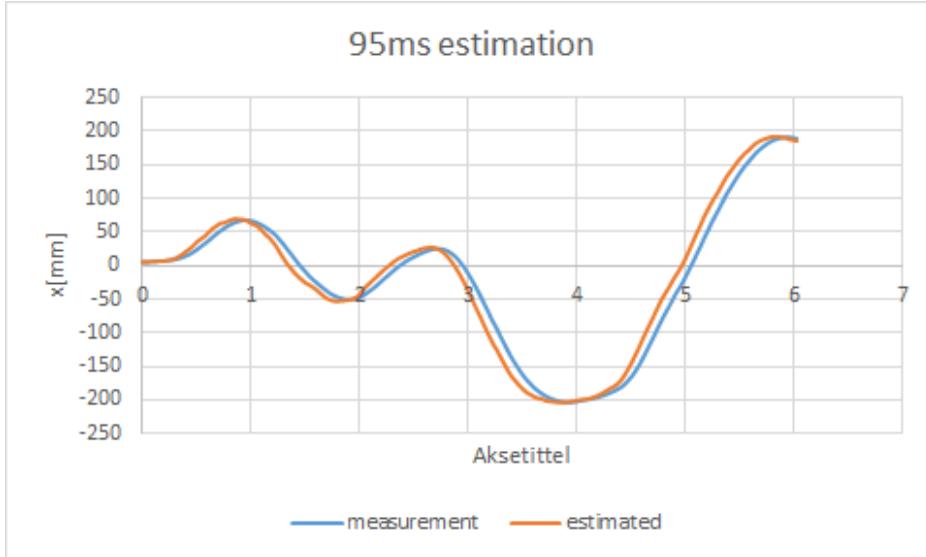


Figure 55: 95ms vs the directly measured data when the operator moves the head at normal speed

In Fig. 56 the estimation for the maximum case in the part where the operator moves "unnaturally" fast is studied. That is in the [7, 10]s part of the graph. As we can from the plot see the estimator performs poorly and misses with nearly 10cm at the worst case. In addition it has some unwanted oscillations at about 8.2 seconds.

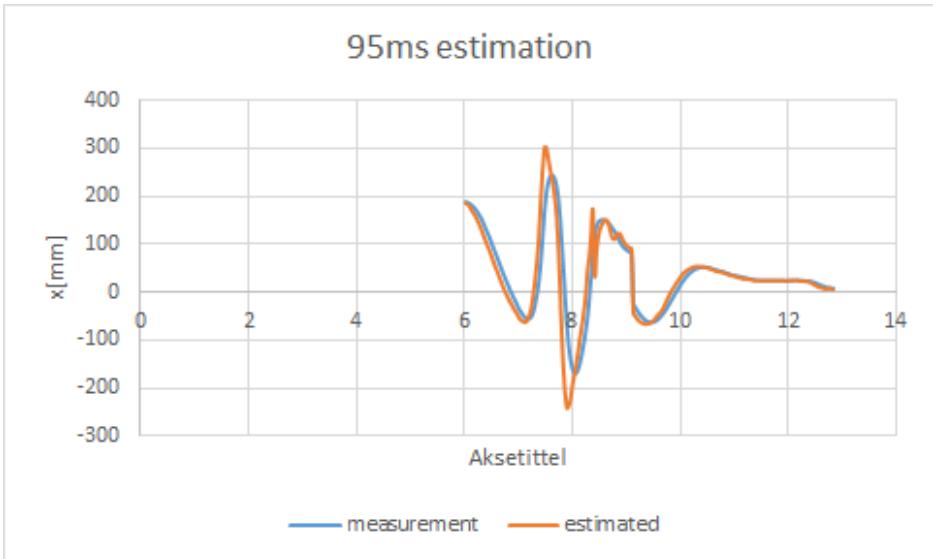


Figure 56: 95ms vs the directly measured data when the operator moves the head at very high speed

In Fig. 57 we study the estimation in the [7, 10] case for different input parameters. We can see that the 20ms estimation manages to make an accurate estimation even in this extreme case, and can see how the estimation performs worse and worse the longer into the future it tries to estimate.

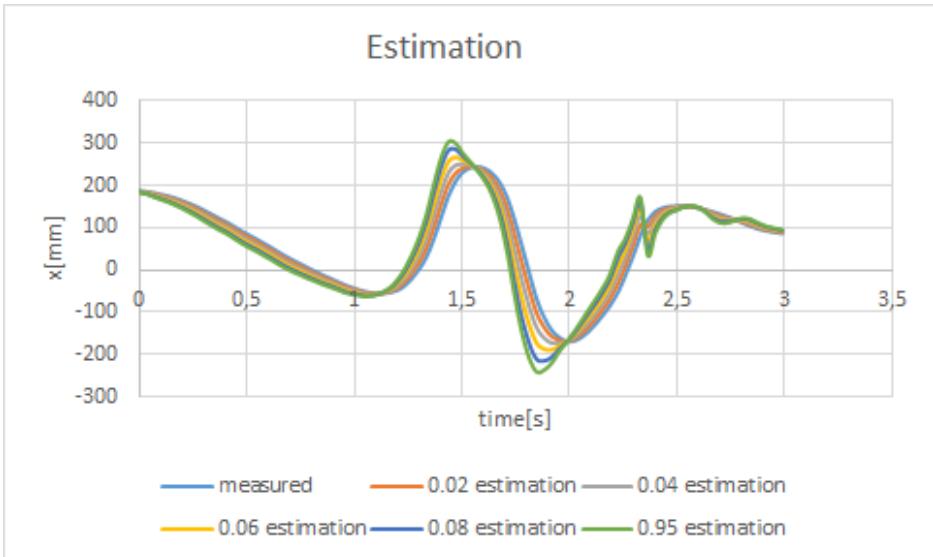


Figure 57: Estimation for multiple different input parameters

It is hard to conclude anything from this data without trying the application to see if the tradeoff between accuracy and delay time gives the operator a better experience. When concluding from these datasets it is reasonable to exclude the case where the operator moved the head at high speed. The Oculus Rift is made for computer games, and in some games it can be crucial to throw the head from one side to the other. For our application however, these extreme movements are not considered crucial at this point in order to prove the concept. When studying the case with  $95ms$  estimation for normal motions we can see that the estimation tracks really well. There are maximum a few cm difference at the extreme points from the actual movements. This is perhaps easier to see in the following plot (Fig. 58) where the estimation has been shifted  $95ms$  in order to make the graphs overlap each other

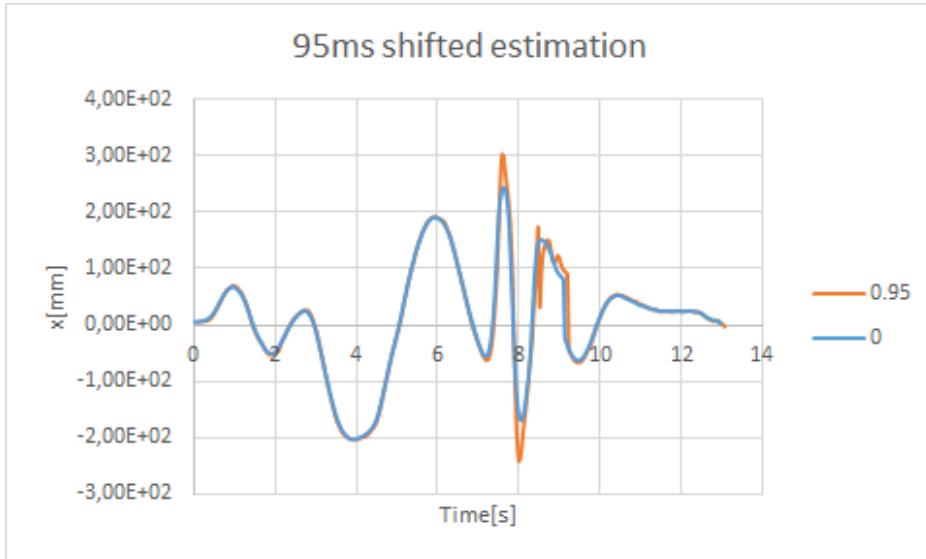


Figure 58:  $95ms$  estimation vs  $0ms$  ms estimation where the estimated data is time shifted to make the data overlapp

The tracking seems acceptable with the maximum future estimation parameter of  $95ms$  at normal head motions. A guess can be made that this is something that will improve the experience for the operator as it will reduce the latency felt by the operator by  $95ms$ . This must however be tested with the complete system in order to make a conclusion as it may make the experience to inaccurate.

#### 6.4.2 zonedata

The system was tested with different *zonedata*. The system performance parameter  $\Psi$  (See Eq. 7) will be given in the time interval  $[0, 20]$  seconds for all the measurement. In the default case, the following values are set for the different

parameters:

- zonedata: z1
- prediciton interval: 0.0
- prefetch time: 0.1
- path resolution: 1
- WaitTime in moverobot thread: 0.1
- process update time: 0.048384
- queue time: 0.193536

## Zonedata: z1

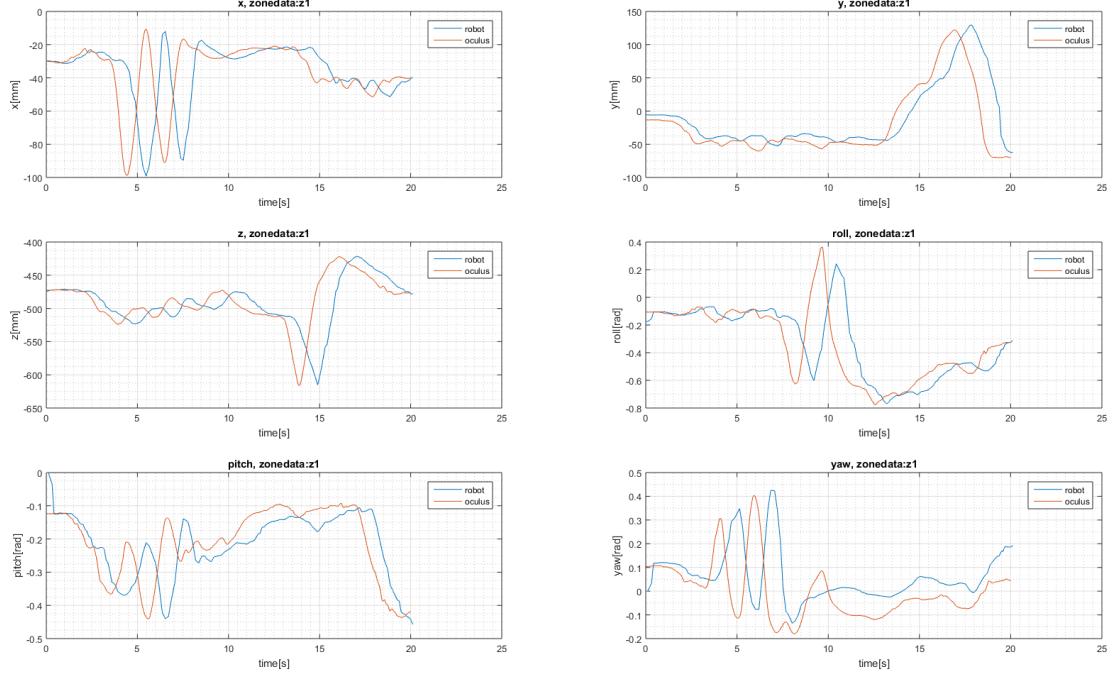


Figure 59: the plots of all axis with  $zonedata = z1$  and  $WaitTime = 0.1$

$Zonedata = z1$  gives the following performance parameters:

$$\Psi = 1137$$

$$\tau = 113$$

As can be see from the plots in Fig. 59 there is quite a bit of delay. In addition to this there is also some quite clear inaccuracies. We can see that in the rapid *x-motions* in the beginning, the robot is in opposite phase of the Oculus Rift, which causes the operator to become confused and dizzy. The relatively big  $\Psi$  and  $\tau$ , compared to some of the other plots later on, indicates this inaccurate behaviour with the high latency. A sample of the latency can be read from Fig. 60.

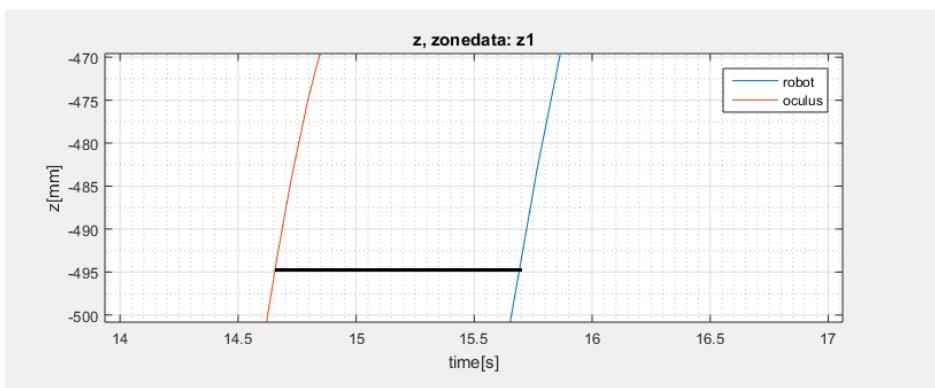


Figure 60: There is a significant time difference between the signals in this plots

## Zonedata: z10

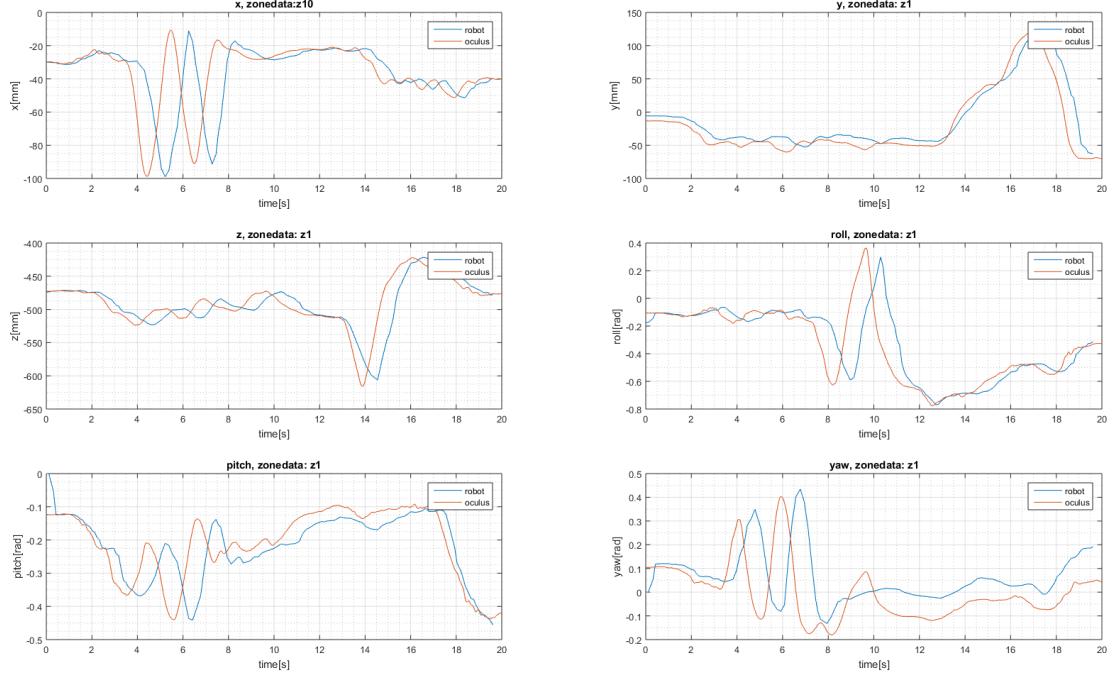


Figure 61: the plots of all axis with  $zonedata = z10$  and  $WaitTime = 0.1$

$Zonedata = z10$  gives the following performance parameters:

$$\Psi = 1029$$

$$\tau = 104$$

When  $zonedata = z10$  is used, a tracking that is a little better than with  $zonedata = z1$  is achieved. The plots are presented in Fig. 61. This could be because of more fluent motions where the robotic arm takes more shortcuts between the points, possibly leading to more inaccurate motion, but less time delay. This compromise leads to a smaller  $\Psi$ . The variations are however small, and it is not possible to draw safe conclusions based on this result.

## Zonedata: z20

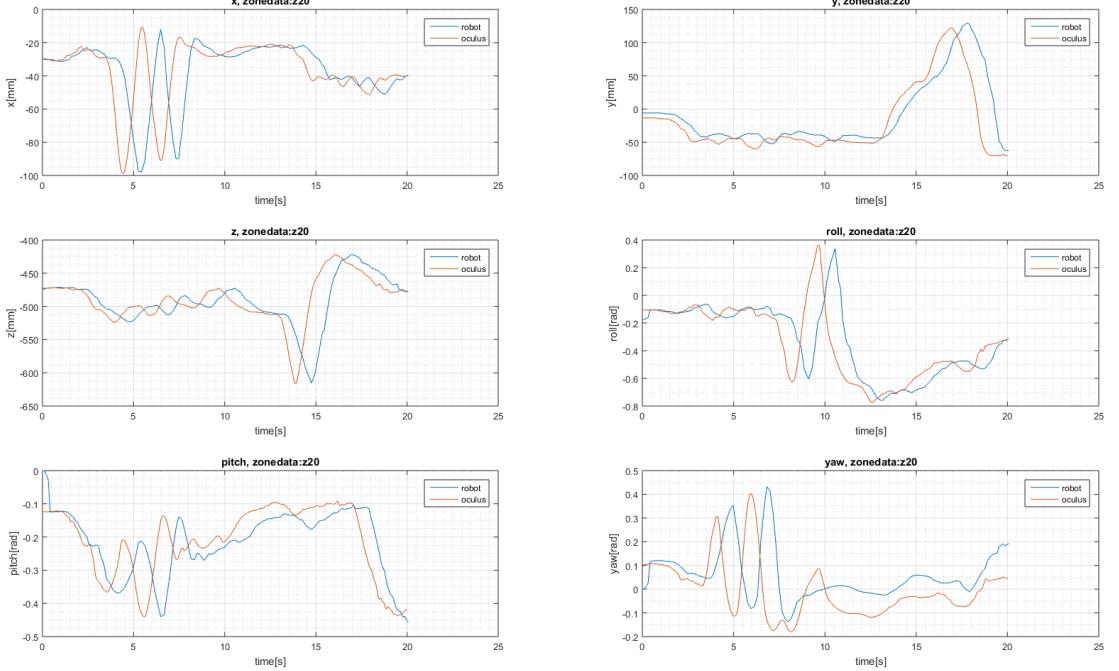


Figure 62: the plots of all axis with  $zonedata = z20$  and  $WaitTime = 0.1$

$Zonedata = z20$  gives the following performance parameters:

$$\Psi = 1071$$

$$\tau = 114$$

When the  $zonedata$  is adjusted to high, like in Fig. 62 with  $zonedata = z20$ , the performance decreases slightly, however, the difference read from the performance parameters are very small. The planning of the trajectories takes time and the calculations makes the time delay high as well as the inaccuracy is high.

## Zonedata: fine

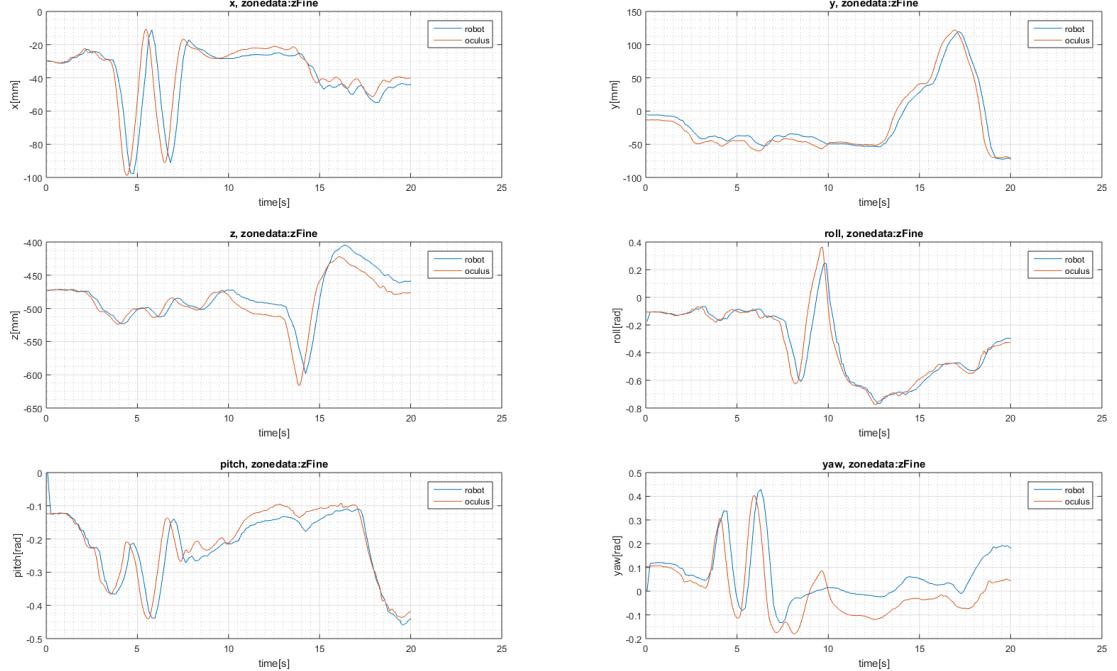


Figure 63: the plots of all axis with *zonedata* = *fine* and *WaitTime* = 0

*Zonedata* = *fine* gives the following performance parameters:

$$\Psi = 605$$

$$\tau = 29$$

In Fig. 63 *zonedata* is set to *fine*. This means that the robot passed through each point, not making shortcuts with zones. This requires less calculations and removes the need of a *WaitTime* in the *MoveRobot* thread. This reduces latency but can lead to a more choppy movement (more on this in Sec. 6.5). The increased performance with this setting can be seen from the graphs and the performance parameters. However, this setting still has some pretty serious inaccuracies in some of the complex motions patterns (e.g [8, 11] seconds).

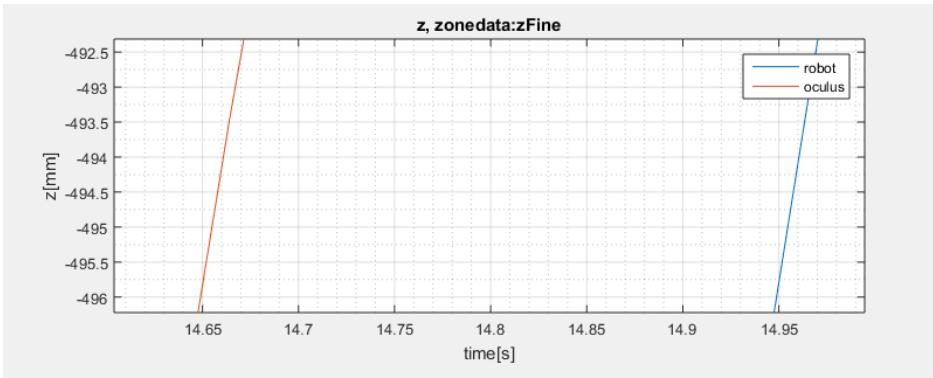


Figure 64: The Latency can be read from this plot to be about 0.3 seconds

When setting the *zonedata* to *fine*, the system is allowed to do a more rapid update of the robot position, which makes the system more responsive to changes. This can be seen from the latency sample in Fig. 64. In order to use fly-by points, *WaitTime* = 0.1 second must be set in the T\_ROB1 thread. If not RobotStudio gives the error message: "*Correct regain impossible*". This is due to too many subsequent close points requiring corner paths making it impossible to use *zonedata* ≠ *fine*. We can see from the performance parameters, and from the plots that the system performs better when *zonedata* = *fine*. In the coming sections, *zonedata* = *fine* will be tested as the default case, however, with some exceptions.

#### 6.4.3 Prefetch Time

The system is then tested with different prefetch times for *zonedata* = *z10* because this *zonedata* performed best in the previous test. Testing for *prefetchtime* = 0 is equivalent of testing the system with *zonedata* = *fine* and will not be tested. Prefetch time defines the resources put into calculating the *zonedata* trajectories, if this would lead to an increase in performance due to more accurate and rapid computations is not known. The reason this parameter is tested is to see if the system can perform better even though *zonedata* is used. The *prefetch time* can be set between [0, 10]:

- *zonedata*: *z10*
- predictiton interval: 0.0
- prefetch time: varying
- path resolution: 1
- *WaitTime* in moverobot thread: 1
- process update time: 0.048384

- queue time: 0.193536

## Prefetch Time: 2

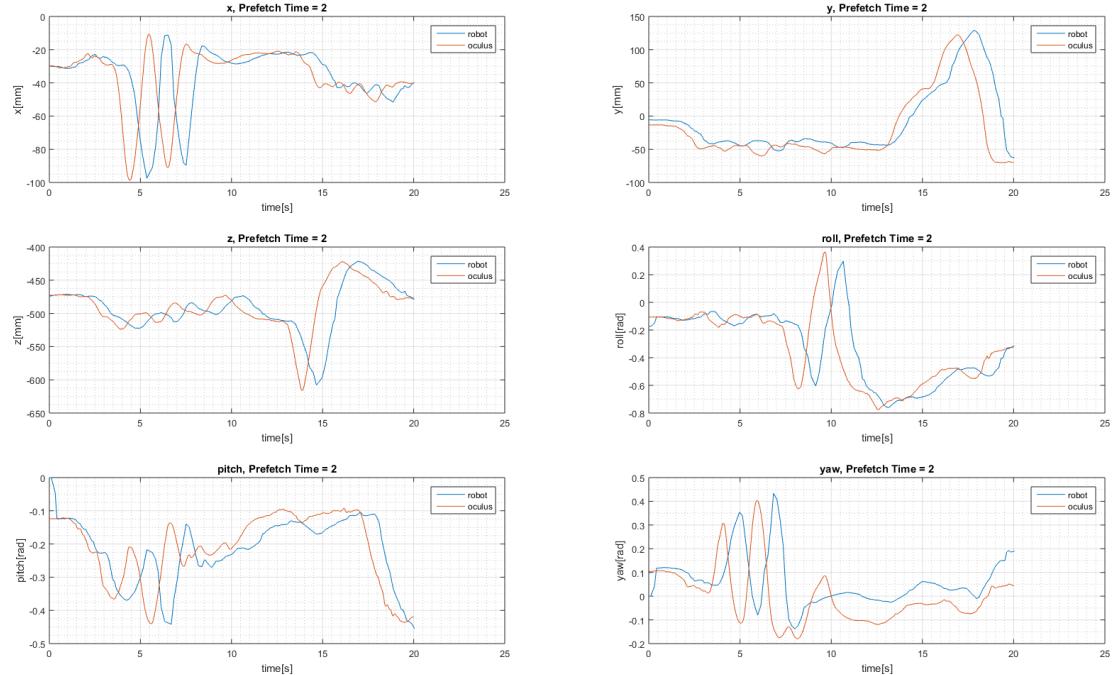


Figure 65: It is hard to read any differences from this case to the default case with *prefetchtime* = 0.1

*prefetchtime* = 2 gives the following performance parameters:

$$\Psi = 1106$$

$$\tau = 119$$

We can see from the performance parameters and from Fig. 65 that the increase in *prefetch time* does not give an improvement in the accuracy or time delay of the system. The variations from the standard case with *prefetchtime* = 0.1 is so small that it hard to say something accurate about the influence of this parameter.

## Prefetch Time: 10

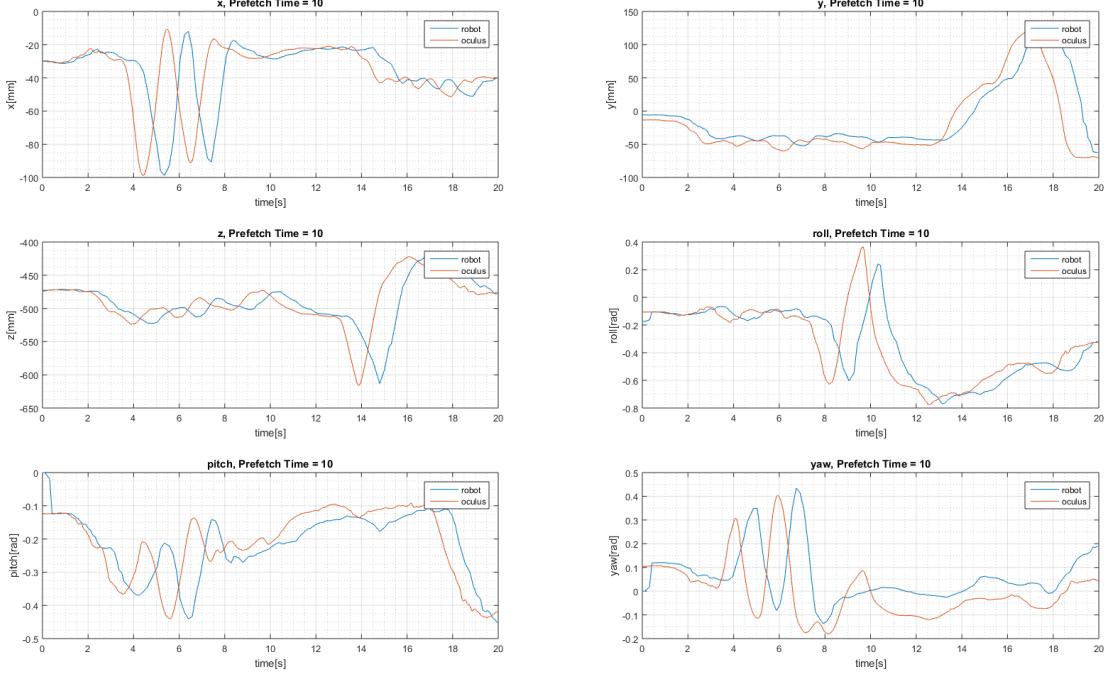


Figure 66: It is hard to read any differences from this case to the default case with *prefetchtime* = 0.1 and the *prefetchtime* = 2 case

*prefetchtime* = 10 gives the following performance parameters:

$$\Psi = 1064$$

$$\tau = 107$$

The same can be seen from these. The performance parameters and Fig. 66 shows that the influence of the *prefetch time* parameter is negligible on the system performance.

### 6.4.4 Path Resolution

The system is now tested with different parameters for *path resolution*. The *path resolution* for the default case where set to 1, so the system will now be tested with the minimum value 0.1667 to compare against the case where *pathresolution* = 1 and the max case where *pathresolution* = 6 in order to see if there will be any

difference in performance. To lower the *path resolution* could lead to a more accurate path, but lead to more computations and therefor increased latency.

- zonedata: fine
- predicton interval: 0.0
- prefetch time: 0.1
- path resolution: varying
- WaitTime in moverobot thread: 0
- process update time: 0.048384
- queue time: 0.193536

**path resolution: 0.1667** This gives the following error in RobotStudio: "50226: Motor reference error - Reduce load on main computer" which indicates that this low value gives to high CPU load.

**path resolution: 0.4** Both  $\text{pathresolution} = 0.2$  and  $\text{pathresolution} = 0.3$  gave the same error message,  $\text{pathresolution} = 0.4$  was the lowest value not giving the 50225 error message and was therefor tested.

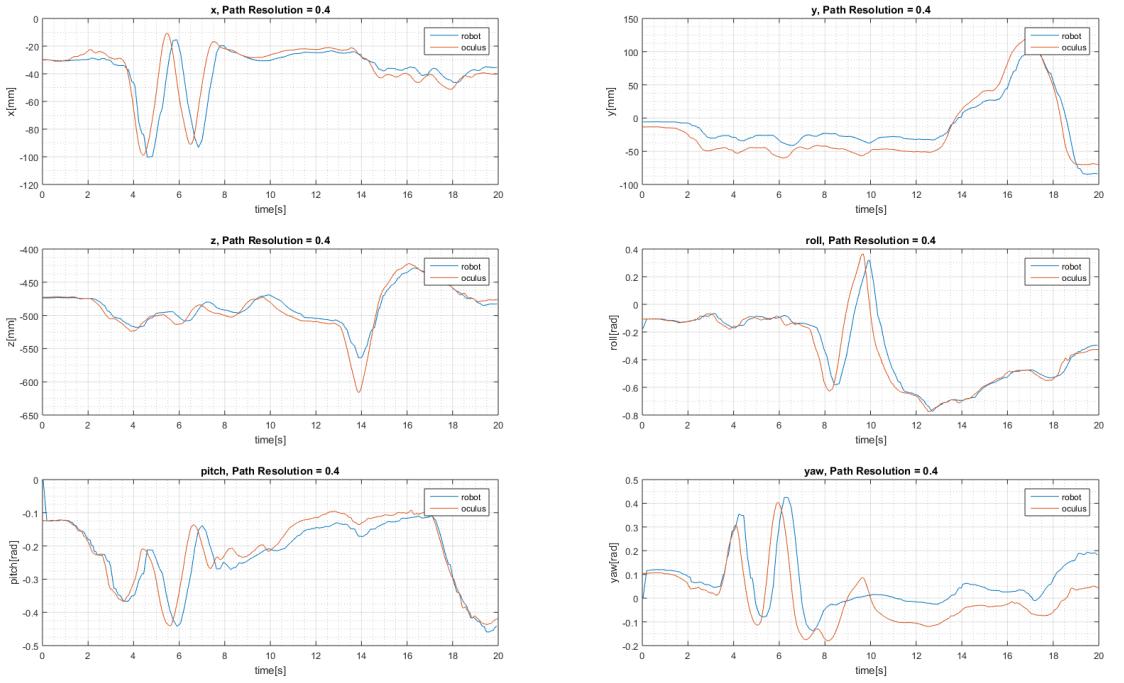


Figure 67

*pathresolution* = 0.4 gives the following performance parameters:

$$\Psi = 639$$

$$\tau = 30$$

As seen from the performance parameters and Fig. 67, setting the *path resolution* lower does not seem to affect the performance of the system to much.

## path resolution: 6

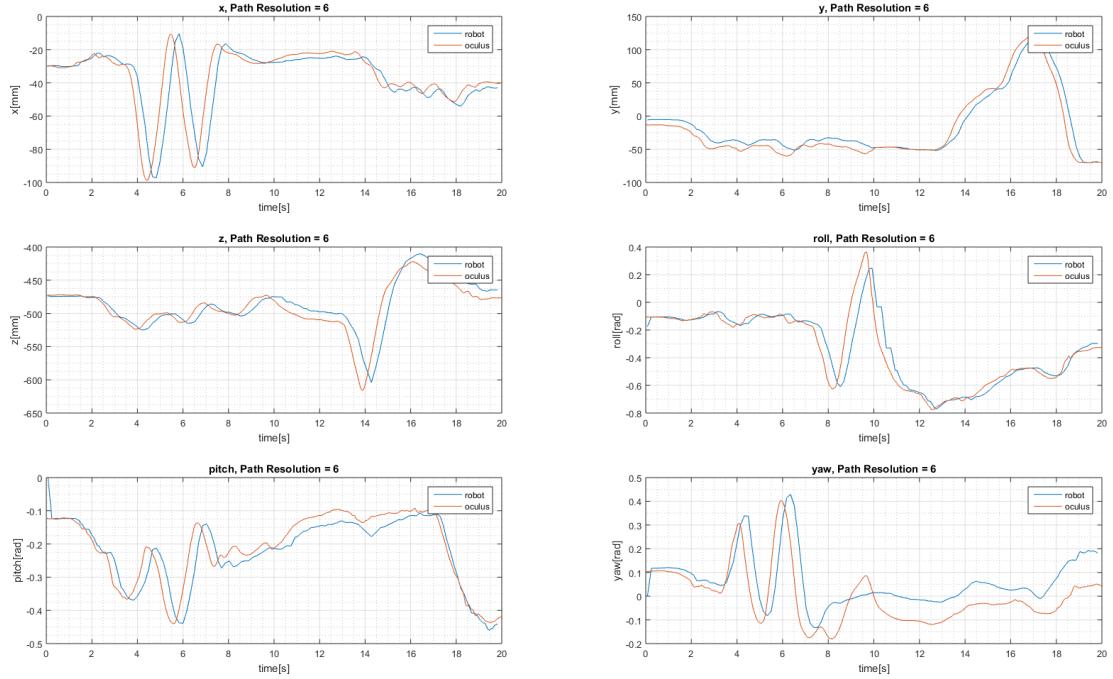


Figure 68: It is hard to read any differences from this case to the default case with  $\text{prefetchtime} = 0.1$  and the  $\text{prefetchtime} = 2$  case

$\text{pathresolution} = 6$  gives the following performance parameters:

$$\Psi = 600$$

$$\tau = 34$$

As seen from these performance parameters and Fig. 68 Using the highest extremal value of  $\text{path resolution}$  does not affect the system to a significant degree either.

### 6.4.5 Process Update Time

Process update time determines how often the process path information is calculated. Decreasing it increases accuracy and increases CPU load, increasing it decreases accuracy and decreases CPU load. This is why adjusting this parameter can go both ways. Decreasing it could make the system accurate, but slower

while increasing it could make the system more inaccurate, but faster. The value can be set between 0.012 and 1.93. Due to the low default value of 0.0483 only the upper extremal value of 1.93 will be tested in addition to the default value to see if this affects the performance.

- zonedata: fine
- predictiton interval: 0.0
- prefetch time: 0.1
- path resolution: 1
- WaitTime in moverobot thread: 0
- process update time: varying
- queue time: 0.193536

### process update time: 1.93

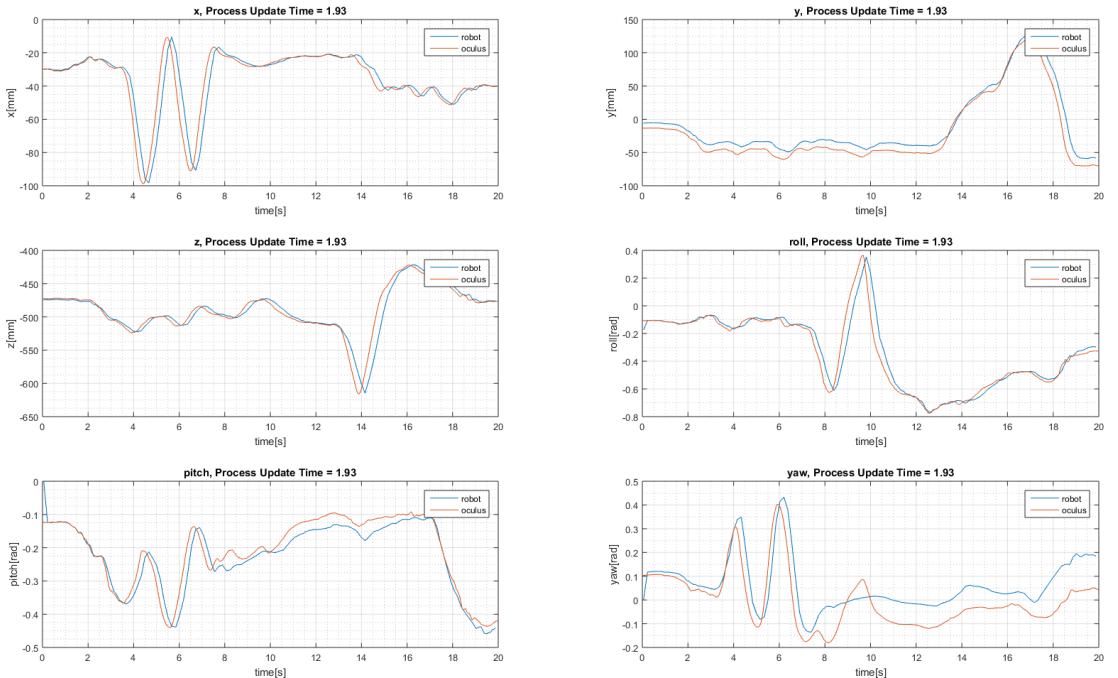


Figure 69: This figure shows the plot where the *processupdatetime* = 1.93

*processupdate time* = 6 gives the following performance parameters:

$$\Psi = 449.18$$

$$\tau = 20$$

Based on the performance parameters and Fig. 69 at this setting, it could seem like a high *process update time* have a positive effect on the performance of the system. This could be because of a lower CPU load and therefor faster computations.

#### 6.4.6 Queue Time

*Queue time* is a parameter that could make the system more responsive. The *queue time* can be set in the interval [0.004032, 0.290304]. A higher *queue time* will make the system more unresponsive but more tolerant to uneven CPU loads, a low *queue time* would make the system more responsive but less tolerant to uneven CPU loads. If this will have a noticeable affect on this system is now tested by testing the two extremal values of *queue time*, 0.004032 and 0.29:

- zonedata: fine
- prediciton interval: 0.0
- prefetch time: 0.0
- path resolution: 1
- WaitTime in moverobot thread: 0
- process update time: 0.048384
- queue time: varying

**queue time: 0.004032** This gives the error message "10014: System failure state"

**queue time: 0.05** This was the lowest value not giving the *10014* error message.

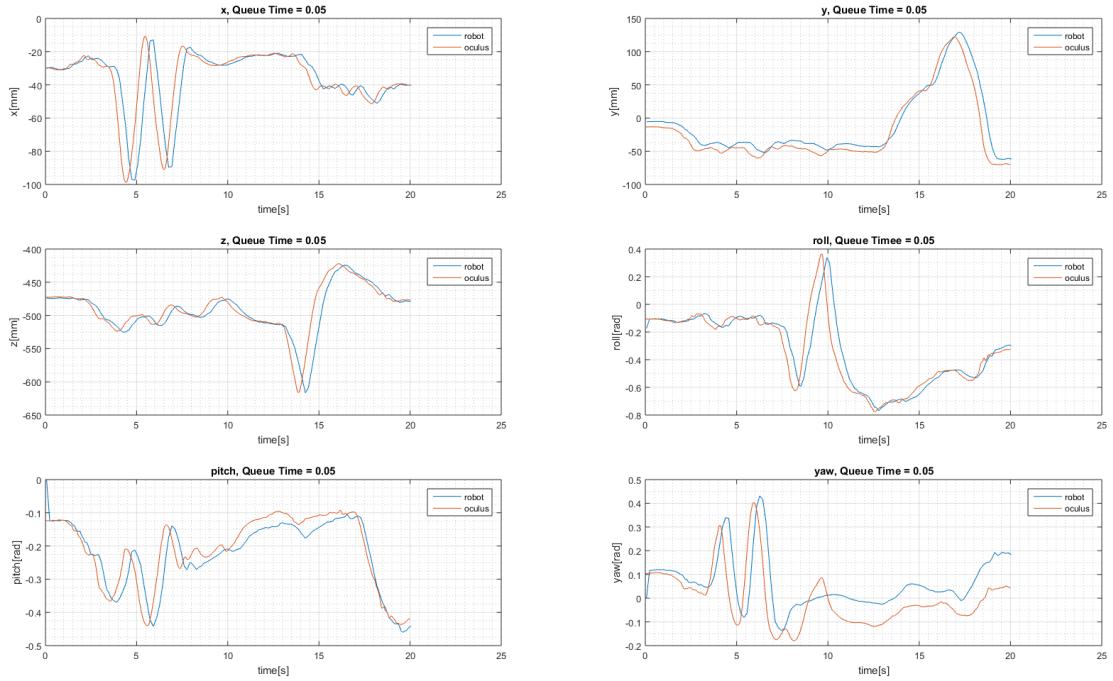


Figure 70: This figure shows the plot where the *queuetime* = 0.05

$$\Psi = 576$$

$$\tau = 40$$

**queue time: 0.29**

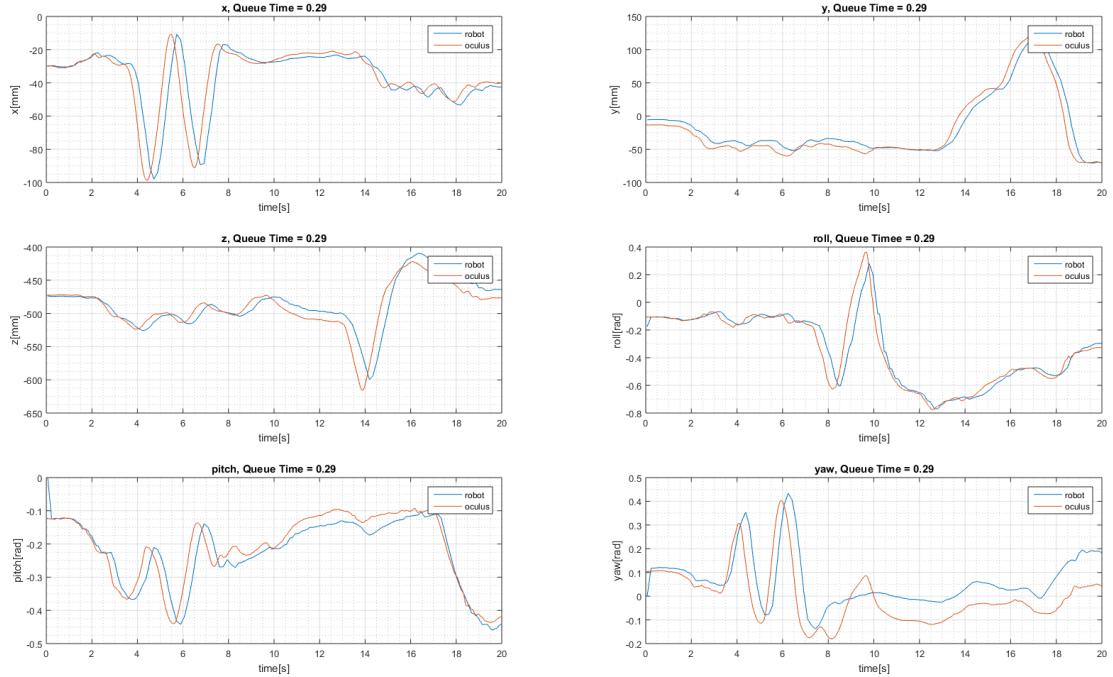


Figure 71: This figure shows the plot where the *queuetime* = 0.29

$$\Psi = 572$$

$$\tau = 29$$

It is not possible to make any conclusions based on these small variations in performance parameters. As can be seen, the change of *queue time* changes the result slightly, but does not have any significant effect on any of the performance parameters and it is not possible to spot any changes in Fig. 70 and 71.

#### 6.4.7 Move Joint Algorithm

In order to see if the latency of the system lay in the use of the calculations done in the *MoveL* algorithm, the simpler *MoveAbsJ* was tested. For this setup the

time delay was found to be  $\approx 0.43$  seconds by studying the Fig. 72. The test environment was the following:

- zonedata: z1
- predictiton interval: 0.0
- prefetch time: 0.0
- path resolution: 1
- WaitTime in moverobot thread: 0.1
- process update time: 0.048384
- queue time: 0.193536

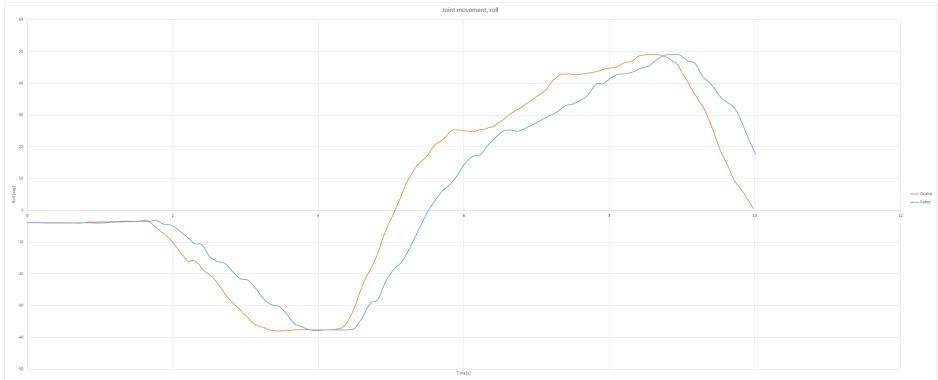


Figure 72: It can be seen that even when the *MoveAbsJ* algorithm is used, a delay occurs about the same as the one when *MoveL* with *zonedata = zFine* is utilized.

In comparison the *MoveL* algorithm gives the graph shown in Fig. 73, with the same test environment as the move joint algorithm. Because of the nature of *MoveAbsJ* and *MoveL* the same input list could not be used in the two plots, this is why the input is directly from the Oculus and therefore looks different. Both of the graphs shows roll motion. Here the time delay was found to be  $\approx 0.63$  seconds by studying the graphs, however, this will vary with the complexity of the input.

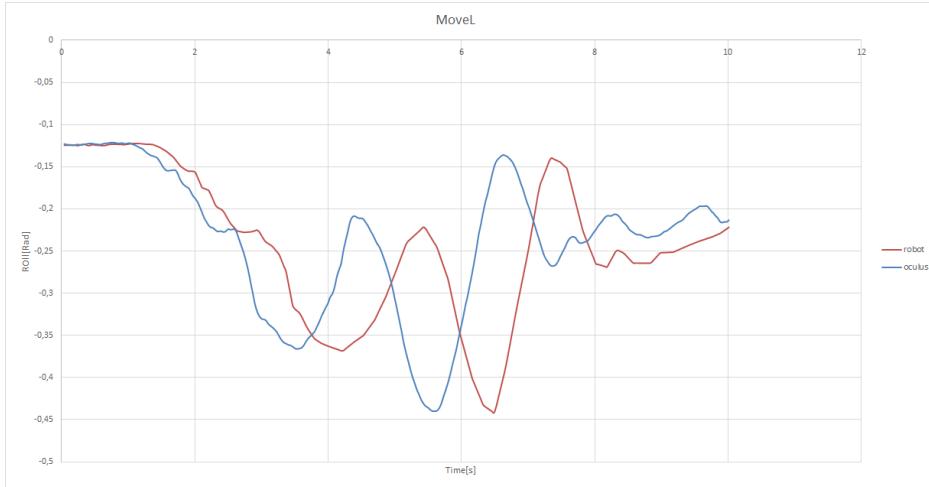


Figure 73: The delay here is more significant than with *MoveAbsJ*, but for *zonedata* = *zFine*, there is no significant difference in delay between the two algorithms.

## 6.5 User Experience

The testing of how the system was now important to determine what settings was the best to use. As the experience is dependent on many different variables (accuracy, stuttering, delay, video quality) one cannot conclude anything by just studying the system plots, the system also has to be tested.

First of all the system was tested with different *zonedata*.

- *zonedata* = *z10* gave a relatively smooth motion, but the delay was significant.
- *zonedata* = *z1* gave a more choppy experience than *z10*, but still with a significant delay
- *zonedata* = *zFine* gave a relatively choppy motion (about the level of *z1*) but a significantly more responsive feel with less delay.

In the continuation of the testing, the system was tested with *zFine*, as this gave the best result both by testing the system and by studying the performance parameters.

The next parameter that was tested was different prediction intervals.

- *predictoninterval* = 0.095: from studying the graphs in the testing of different prediction intervals, it was assumed that 0.095 would give a decent

estimate and reduce the latency so much that the tradeoff would be good. However, after testing the system, both the testers was clear that this high prediction interval made the system uncomortable to use because the system was drifting to much.

- $\text{predicitoninterval} = 0.040$ : from studying the graphs it is hard so notice any inaccuracies with this prediction. This also correlated well with how the system felt with this setting. It was impossible to notice any difference in accuracy from the 0 seconds prediciton, so the tradeoff in reduced latency is well worth it with this setting.

The system was then tested with the maximum *process update time* to see if this gave a better experience. The reason for this test was based in the performance parameters. The operator was however not able to sense any difference in performance from this adjustment.

In the documentation of *queue time*, there is a good indication that this parameter could have some impact on latency. This parameter will therefore be tested with the system even though there was no significant indication of influence from the plots. The two extremal values of the parameter will be tested in order to see if the operator can feel any difference

- queue time = 0.05
- queue time = 0.3

The operators testing the system could not sense any difference between these two settings.

## 6.6 Parameter Adjustment Summary

The most significant changes from the parameter adjustments came from changing the *zonedata* to *zonedata = fine*. In some of the parameters that was believed to have an impact (like *queue time*) no significant impact was found. The *prediction interval* clearly had some impact on the final solution. Because many of the parameter adjustments brings a tradeoff between accuracy and delay, the use of the performance parameters and studying of the graphs are limited, the best they can do is to give an indication of what setting is the best for the performance of the system. As there was some inconsistency in the measurement of the performance, conclusions cannot be based on small variations.

The indication from the performance parameters was a good indication for us to have with us when the test was done on a physical robot. To change the system parameters and restart the robot controller on the physical robot for each parameter change is a time consuming process, and to have good indications from the simulation helped us to make better decisions when testing on the physical robot.

The settings that proved to be closest to optimal after both testing the system and studying the performance parameters was the following:

- zonedata: zFine
- predicton interval: 0.040
- prefetch time: 0.0
- path resolution: 1
- WaitTime in moverobot thread: 0
- process update time: 0.048384
- queue time: 0.1

## 6.7 Sound

During the early testing there was no sound stream from the robot to the operator. However, it was later added a stereo microphone on the top of the cameras in order to be able to stream stereo sound to the operator. The result of getting both the vision input and sound input from the robot made a huge difference in the immersive experience. The whole experience became more realistic. It was very interesting how both the testers felt a lot more dizzy and "weird" after testing the system with sound and vision, compared to testing it only with vision.

Initially the yeti microphone was duck taped directly on the top of the camera. However, this made the sound from the vibrations from the robot to significant and reduced the quality of the experience. You can see the setup in Fig. 74.

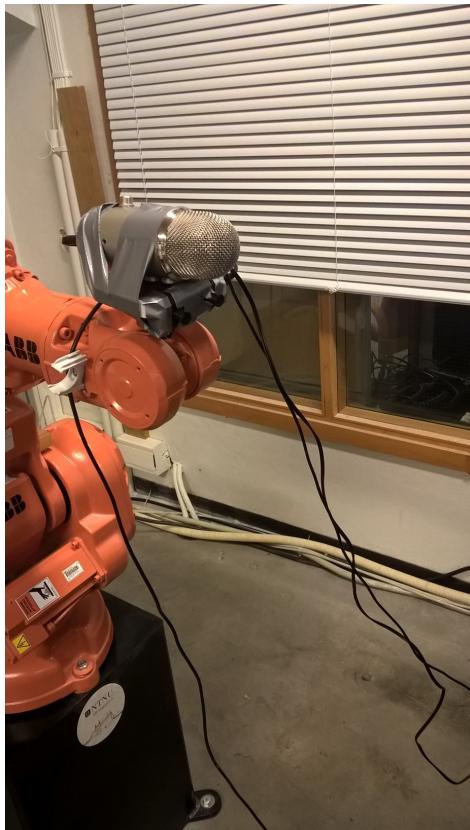


Figure 74: microphone setup with no vibration protection between the robot and the microphone

Because of the significant sound from the vibration of the robot, it was decided to add a protective foam between the robot and the microphone. This made most of the vibrational sound disappear and increased the quality drastically. There is still some vibrational sounds, but they are deeper and less significant. Fig. 75 shows the setup with the protective foam.

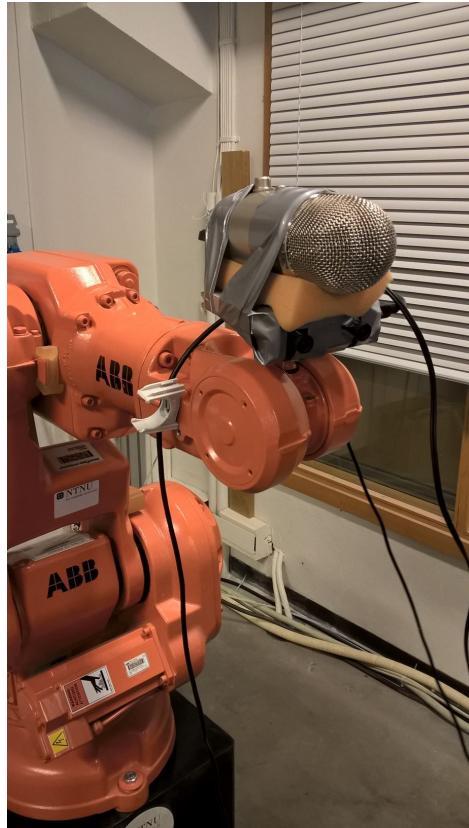


Figure 75: microphone setup with a protective foam between the microphone and the robot

To use a condenser microphone was a solid choice, as it is able to pick up sound from a long distance. This made it possible to stand far away from the robot and talk to the operator, giving a very realistic feeling for the operator. The stereo recording ability of the microphone also gave an important aspect in the feeling of realism, as the operator, as expected, was able to point out the direction to the sound source.

## 6.8 Image Quality

The image quality was greatly increased from the previous setup with the Mobius ActionCams to this setup with the Ovrvision. The added barrel distortion to the image rendered to the Oculus Rift resulted in a picture that was lined up correctly, and not distorted. The increased framerate made the picture appear more smoothly and not so choppy as it did with the Mobius ActionCams. In addition to this, the alignment of the pictures was much better in this setup than what it was with the Mobius ActionCams. Even with the adjustment opportu-

nities in the custom viewer made for the Mobius ActionCams, the picture from the Ovrvision had a much better overlap and gave the 3D view increased quality. The only disadvantage with the Ovrvision compared to the Mobius ActionCam setup was the reduced pixel density. This disadvantage was hard to notice as the changes of the other parameters improved total experience.

The brightness of the camera was also considered, and some different parameters where tested. The environment around the robot in the lab is difficult, with bright lights in the roof and dark sections around the robot. This forced us to do a compromise, and the best options turned out to be a relatively high brightness where the lamps in the roof occurred to bright, but the rest of the environment turned out in good, natural light. With a more advanced camera setup it would perhaps be possible to do real-time image handling, taking care of the variable light conditions.

Even though the camera quality was greatly improved with the Ovrvision, the most important aspect with the new camera rig was definitely the reduced latency, as the latency of the Mobius ActionCam is estimated to be  $100 - 200ms$ , the Ovrvision has a latency of about  $50ms$ .

Fig. 76 and 77 illustrates the difference between normal lighting and a bright light source respectively. Due to the software connecting the OVRvision to the Oculus Rift it was not possible to take a print screen, and therefore the image was taken through the lenses causing a reduction of quality.



Figure 76: Showing the camera image in normal lighting conditions, as can be seen the settings of the camera here is acceptable.

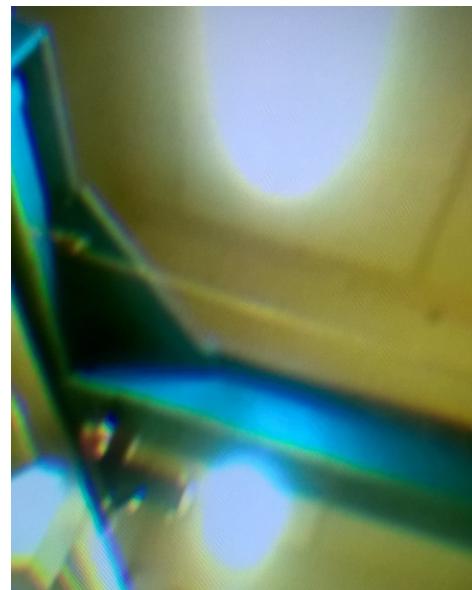


Figure 77: Showing the camera pointing towards a light source, the settings is not optimal for this, and the light gets intense in the picture.

## 7 Discussion

### 7.1 Importance of Analyzing Measurements Versus How the System Feels

In the work of analyzing how well the system performed some work was put into defining parameters that was possible to compare in order to find out under what circumstances the system performed best. The importance of these parameters turned out not to be as significant as first though. The most significant changes in performance was so clear that it could be felt when testing the system as well as from the performance parameters (e.g when setting *zonedata = fine*). However, neither the performance parameters or the operator testing the system was able to pick up the small impacts changing some of the system parameters may have caused. This is both due to small inaccuracies in the system parameters, see Sec. 7.2, and due to limited capability of human sensing. It could be that there exist some better ways to determine tracking performance in an analytical way that should be investigated if further developing the system.

### 7.2 Fine Tuning of System Parameters

The fine tuning of the system parameters was not as successfully as hoped for. The idea was that the tuning of these parameters would improve the system, and as the results shows, some of them did. However, the performance parameters were unable to pick up the small changes in the system, caused by fine tuning parameters, due to inconsistency in input to output, see Sec. 7.4. This is not necessarily important, as the tuning of the system parameters is clearly not where the main source of delay lies. However, if the system are to be improved in the future, the presented edition of RobotStudio is probably not a good choice as a developer platform. In that case, other approaches would be made to improve the system.

### 7.3 Total System Quality

The total experience of the system was above the expectations of the authors. Even though the system had some delay, this was hard to notice when actually testing the system. The camera quality is also of such a quality that the operator gets a clear overview of the surroundings. When using the whole system the operator indeed gets an "out-of-the-body" experience which could be a useful feature in remote operations in terms of getting a good overview and have a feeling of presence. In total, the system has turned out to be a promising concept.

### 7.4 Consistency of Output Data from Standard Input in the Simulator

Even though sending through the same input with the same settings in RobotStudio, some inconsistency in the performance parameters where noticed. This is

why it was hard to make conclusions based on the performance parameters when there were only small changes, even though they might had a small effect on the final result. This inconsistency can have multiple reasons, which are hard to evaluate without a deep understanding of how RobotStudio works (which is not accessible due to ABB copy rights). One of the reasons could however be inconsistency in the data transfer with TCP protocol. The data update rate is variable because the run time in the MoveRobot algorithm varies for different motion complexity. Small changes here could lead to different values being passed from the *QMap* containing the position values, which could lead to a slightly different result and slightly different performance parameters. In addition, background processes can steal resources and change the calculation time of the algorithms in RobotStudio. It is also speculated of the possibility that noise is added to the simulation in order to create a more realistic simulation environment and that this also leads to some inconsistency. There is no information about how the Robot Models are defined in the documentation from ABB.

## 7.5 Vibrations and Subsequent Problems

The IRB140 is a big and heavy robot. When the robot is tracking the Oculus Rift it does this by going through each point, and sometimes stopping at these points, this causes choppy motions which are not ideal. If the robot is to be placed on a mobile platform, which indeed would be necessary for some of the applications in remote operations, the robot will cause vibrations in whatever vehicle it is placed on. This could cause problems in navigation and stability of the vehicle it is placed upon. This problem must be solved by either having a big and stable vehicle where the vibrations are not relevant, having a smaller robotic arm and/or having higher update rate with smoother motions and fly-by-points. The vibration problem can bee seen directly (see the video[42]) from when the robot was placed on a metal board which was not attached to the floor, heavy vibrations occurred in the structure due to momentum from the robot on the metal board. In order to stabilize the robot it was bolted to the floor, see Fig. 78.



Figure 78: An image of the robot bolted to the floor.

## **7.6 Drawbacks of Computing the Inverse Kinematics on the Client**

Due to RAPID being a high-level programming language and the fact that it is not made to operate in a real-time environment it seems like it might be preferable to solve the inverse kinematics problem on the C++ client instead of on the RAPID server. The advantages would be that the algorithm would likely be faster as well as it would be possible to tune and optimize it whereas this is not possible to do in RAPID due to the algorithms being proprietary.

The drawback is the amount of communication that it would require as well as the low reduction of runtime that MoveAbsJ offers. Due to the Oculus Rift and the manipulator being in two separate coordinate systems and the only information available is the change in position, or the position of the Oculus Rift. This would require a constant stream of information from the server which would cause a latency which would be higher than the gain in run-time. It would also require another thread on the server side to track the movements of the robot and calculate the location of the new position. Although the strain on the controller would be lessened as it would not need to calculate the path itself it would not be beneficial from a performance perspective as the responsiveness of the system is drastically decreased by the amount synchronization required.

After considering the drawbacks and advantages of using a custom solver for the inverse kinematics it was deemed unnecessary. Another reason is the possibility of using fly-by points in RAPID as well as that the movements are ensured to be linear at the TCP.

## **7.7 Possible Applications**

### **7.7.1 Remote Operations**

This is probably the most relevant application for the system. Remote operations defines the range of operations where human presence is dangerous or impossible. With this system, the operator of a remotely operated system would get an overview that is impossible with the current systems available. The possibility of observing objects from multiple angles with full depth vision could prove useful in e.g sub sea remote operations. Other operations could be after accidents in nuclear power stations. If some hardware upgrades are made, the system at its current state already has the quality to use in such applications. The inclusion of sound localisation also gives the operator a better overview of the surroundings because more of the operators senses are stimulated. This could provide useful and make the operator make safer decisions.

### **7.7.2 Demonstrational Purposes**

The system could be used for demonstrating robotic arms and their performance. It could be used by robotic companies at fairs to demonstrate their robots in an

hands-on and intuitive way for the crowd. The system is intuitive to use for people with little knowledge about robotics because all the input the operator gives the system is through natural motions. The system could therefore be a portal for the general populace to learn more about robotics and its applications. It could be a motivator for young engineering students to test the system to see what is possible to do with their education.

## 7.8 Latency

As could be seen in Sec. 6.1 the worst case scenario will create a high amount of TSL. However, it is important to note that the movement that was represented was too fast, and as shown in Fig. 52 faster movement correlates to more latency. On the other hand the best case scenario and the runtimes around average are within the target area presented earlier. This was also done with standard settings, while as shown in the Results section the worst case delay could be reduced down to  $300ms$ . While this is still a major delay, it feels acceptable when using the application.

As mentioned earlier there was some latency for which the source could not be identified, however, it is speculated that this is due to the transmission delay between the robot controller and the robot as well as the time it takes for the robot to react. This can be seen when trying to jog the robot with the joystick on the FlexPendant. Another fact that is interesting is the robots ability to track when there are fast movements in more than one orientation. As can be seen in the Result section it loses the ability to track properly when there is a rapid change in both yaw and roll. By looking at the plots it can also be found that there is generally less latency when tracking the orientation in comparison to the position. This is due to the responsiveness of the respected joint as the elbow joints are slower than the wrist joints.

One of the most important new features is the addition of the future estimation from the Oculus Rift. This can reliably predict up to  $95ms$  into the future with decent accuracy, and up to  $40ms$  with good accuracy. As the accuracy is not that important it is a straight up reduction of latency leaving the average case of the latency well within the target area.

## 7.9 RobotStudio and RAPID for Real-Time Tracking

The most important facets of real-time tracking is latency and precision. In the RobotStudio algorithms the precision of the movement is accurate when using the *zonedata fine*. This could even be reduced to perhaps *z5* and still be precise enough if not for the delay it would bring.

The problem with the current iteration is that it does not allow any optimization of the movement algorithms related to the inverse kinematics. This would not be an issue if the joint move algorithm was faster, but until this is rectified there is

a low degree of potential improvement while using RobotStudio.

There is also the lack of length on the strings, which proved to be constricting when establishing robust communication. While this has less with precision and latency it still can have an impact as a more secure communication system would increase the integrity of the packages. All of the issues could be rectified in later versions of RAPID.

## 7.10 Choosing an Inverse Kinematics Algorithm

The inverse kinematics problem is usually solved by numerical methods such as the Jacobian transpose, the Jacobian pseudo inverse or the *Newton-Raphson method* due to their scalability[34]. On the other side numerical methods suffers from mediocre or bad run-times, as seen in Tolani et al. and Meredith[35], in comparison to the analytical algebraic methods. Most of the methods also suffer from numerical instability and have problems around the singularities.

The reason that analytical approaches are not often used is due to the fact that they don't scale beyond 6 DOF. If however, the system is a 3 DOF or 6 DOF system it is a more intuitive approach as well as it has faster run-time. The analytical approach is often divided into an algebraic and/or geometric method. The geometric method for finding the first three joint angles is practical to use since it is intuitive in the case of the *D-H parameter*  $\alpha_i$  is either 0 or  $\pm 90^\circ$ . Another upside with the analytical method is that it does not suffer from numerical instability.

Due to the main drawback of the analytical method is the lack of scalability it was deemed to be the most appropriate method for solving the inverse kinematics problem.

## 8 Further Work

### 8.1 Custom Robotic Arm

It is hard to imagine the current setup being useful in any practical cases when using the IRB140. It is a big and heavy robot with the ability to lift heavy loads up to *6kg*. It could also pose serious injury to humans nearby which leads to necessary safety procedures for any practical application. A small and lightweight, but still fast and accurate robotic arm would be ideal for the application this paper poses. Perhaps the technology is not there yet, but there are some better options on the market. The Japanese company Denso [25] makes smaller robotic arms with a portfolio of robots which focuses more on speed and accuracy rather than powerful engines to lift heavy payloads. To take one example from their portfolio the Denso VP-6242G is a robotic arm weighing 15kg capable of very rapid movements [26]. This robotic arm gets close to the lightweight and portability needed in order to place it on ROVs for remote operations.

### 8.2 Different Robot Communication Environments

In addition to a better robotic arm in general, to test different softwares for robot programming is essential. As concluded, the ABB ecosystem was not open enough to provide us with the customisation needed to perfect our application. There are options to try out, one of them to make a custom software and implement all the inverse kinematics and control on a computer and communicate directly with the robot. It seems like Denso also has software more suitable for this purpose with the b-Cap binary controller access protocol [27]. This software is more open and could provide the insight needed in order to answer some of the unanswered questions of this report, like the source of the inconsistency in the simulation when same input was used.

### 8.3 Wireless transfer

To make the whole system wireless is a natural next step in order to improve the setup. This could be done in multiple ways. The robot controller in itself can not be connected to the Internet, so an external computer must be connected to the controller to send commands to it. One way to do this would be to have a e.g a laptop connected to the robot controller, running all the software. This includes receiving position data from the Oculus with TCP connection from another computer, send the position data to the controller, receiving the video feed and sound stream from the robot and stream it to the computer connected to the Oculus. An even more elegant solution would perhaps be to connect the cameras and the microphone on the robot to a small computer attached to the robot, like a raspberry pie. The raspberry pie could use the built in gstreamer function to stream the video directly to the computer running the Oculus. This would make it possible to run the system without any cables attached to the setup on the robot. An energy source must in that case also be included on

the setup on the robot. Because it is necessary to hard wire a computer to the controller to communicate with it, another computer should be connected to the robot controller to transmit the Oculus values to the controller. This solution would make it possible to be in a other place and operate the system. This setup could however include more sources of latency which is crucial for the system.

## 8.4 True 3D Sound

The stereo microphone that was used to capture sound does not give a full 3D sound replica (binaural recording [28]). A newly released product called "Mitra 3D Mic Pro" [29] makes a good binaural sound replica with its relatively lightweight microphone with the help of software and signal processing. This makes the system able to replicate binaural sound without having a whole human torso to replicate the resonance from the body. This microphone creates a sound picture to make the experience of our concept better.

## 8.5 Adaptive noise cancelling filter in the microphone

When operating the robot it is in the current environment inevitably that noise occurs. It is noise from the controller and from the engines, in addition it is noise from the vibrations of the robot. In order to achieve better sound quality (which indeed turned out to be an important feature of the total system), some kind of noise cancelling could be done on the sound in the microphone in order to get better sound quality with lower noise.

## 8.6 New VR Headset

The development of new technology in the Virtual Reality market is tremendous. All the big technology companies out there are realizing that this will grow to be a billion dollar market in the next few years, and everyone wants a piece of it. To hold track of all the new technologies appearing every week is a challenge in itself. But a couple of techs are separating from the masses. Of course one of them is Oculus Rift, and in the beginning of May, they finally announced (after three years of developer kits) that they would release the consumer version of the Oculus Rift Q1 2016. At the moment this report was written, the technical specifications of this headset was not yet set, but it is likely to be an improvement of the current Oculus Rift DK2. To use this headset to create a more immerse experience for the application would be a natural next step for this project. In addition to the Oculus Rift, the companies HTC and Valve have a cooperation project in creating a VR-headset which will hit the shelves in the end of 2015. The technology is much the same, even though the SDK of this headset would be different than that of the Oculus Rift.

## 8.7 Singularity Avoidance

When the robot hits a singularity, or its close proximity, performance becomes significantly reduced or at worst it will stop the application from performing new move commands. The two types of singularities, arm and wrist singularities, each has a different way of avoidance. Wrist singularities occur if axis 5 and axis 6 are collinear. Thus avoiding the motions that would put axis 5 at the zero angle would solve this. This would also avoid the singularities which lie at the boundary of the reachable workspace as it would introduce a wrist singularity.

Avoiding the arm singularity can be done by using the RobotStudio algorithm *SingArea \ Wrist* which makes it possible to move through singularities at reduced speed. A better alternative would be to check if a movement would make the system Jacobian lose rank. This would be a computationally demanding algorithm, and would best be used in an external application.

## 8.8 Better Cameras

The last obvious improvement would be on the camera system. This is what brings the world to the headset screen and is one of the most important technologies in order to improve the user experience of the system. The company behind Ovrvision are providing a new Ovrvision pro with better specifications than the current version which would be a suitable upgrade for the system.

## References

- [1] Pieper, D.L., "The Kinematics of Manipulators Under Computer Control", memo. AIM72, Stanford Artificial Intelligence Laboratory, 1968.
- [2] Wehinger, P.R. and Carlsen, L.T., "Control of Robot Arm Through Oculus Rift" 2014, NTNU, page 80-90, 101-127.
- [3] Craig, J.J., "Introduction to Robotics, Mechanics and Control", 2nd Edition, Addison-Wesley, 1989.
- [4] <https://www.OculusRift.com/dk2/> 22.03.2015
- [5] <https://forums.RobotStudio.com/> 22.03.2015
- [6] CD:/RAPIDusermanual
- [7] <https://developer.OculusRift.com/documentation/> 22.03.2015
- [8] IRB 140 Industrial Robot Manual, ABB 16.04.2015
- [9] <https://www.OculusRift.com/blog/powering-the-rift/> 16.04.2015
- [10] <http://ovrvision.com/> 17.04.2015
- [11] [http://dev.ovrvision.com/doc\\_en/index.php?reference](http://dev.ovrvision.com/doc_en/index.php?reference) 17.04.2015
- [12] [http://en.wikipedia.org/wiki/Linear\\_interpolation](http://en.wikipedia.org/wiki/Linear_interpolation) 18.04.2015
- [13] [http://dev.ovrvision.com/doc\\_en/index.php?downloads](http://dev.ovrvision.com/doc_en/index.php?downloads) 21.04.2015
- [14] [http://dev.ovrvision.com/doc\\_en/index.php?startup\\_manual](http://dev.ovrvision.com/doc_en/index.php?startup_manual)  
21.04.2015
- [15] <http://www.bluemic.com/yeti/> 23.04.2015
- [16] <http://se.mathworks.com/help/comm/ref/finddelay.html> 23.04.2015
- [17] <http://wwwasio4all.com/> 24.04.2015
- [18] <http://developercenter.RobotStudio.com/BlobProxy/manuals/SysParametersTechRefManual/doc455.html> 30.04.2015
- [19] <https://www.oculus.com/blog/the-latent-power-of-prediction/>  
03.05.2015
- [20] [http://www.aass.oru.se/Research/Learning/drdrv\\_dir/abb\\_irb\\_140.html](http://www.aass.oru.se/Research/Learning/drdrv_dir/abb_irb_140.html) 10.05.2015
- [21] [http://dev.ovrvision.com/doc\\_en/index.php?reference#x5892156](http://dev.ovrvision.com/doc_en/index.php?reference#x5892156)  
10.05.2015
- [22] [http://dev.ovrvision.com/doc\\_en/index.php?downloads#sdk](http://dev.ovrvision.com/doc_en/index.php?downloads#sdk)  
10.05.2015

- [23] <http://hypertextbook.com/facts/2002/JuliaKhutoretskaya.shtml>  
12.05.2015
- [24] <https://developer.OculusRift.com/reference/libovr/> 13.05.2015
- [25] <http://densorobotics.com/world/> 13.05.2015
- [26] <https://www.densorobotics-europe.com/en/product/vp-6242g>  
13.05.2015
- [27] <https://www.densorobotics-europe.com/en/product/b-cap-113> 05.2015
- [28] [http://en.wikipedia.org/wiki/Binaural\\_recording](http://en.wikipedia.org/wiki/Binaural_recording) 16.05.2015
- [29] <http://www.3dmicpro.com/> 16.05.2015
- [30] refers to the Oculus Rift dataset on the CD 17.05.2015
- [31] <http://www.mathworks.com/matlabcentral/fileexchange/37980-barrel-and-pincushion-lens-distortion-correction> 17.05.2015
- [32] [http://en.wikipedia.org/wiki/Trapezoidal\\_rule](http://en.wikipedia.org/wiki/Trapezoidal_rule) 23.05.2015
- [33] M.W. Spong, Hutchinson and Vidyasagar, "Robots Dynamics And Control", 2nd Edition, 2004, page 83-97, 117-123.
- [34] D. Tolani, Goswami and Badler, "Real-Time Inverse Kinematics Techniques for Anthropomorphic Limbs", Academic Press, 2000.
- [35] S. R. Buss, "Introduction to Inverse Kinematics with Jacobian Transpose, Pseudoinverse and Damped Least Squares methods", University of California San Diego, 2009.
- [36] M. Meredith and Maddock, "Real-Time Inverse Kinematics: The Return of the Jacobian", University of Sheffield.
- [37] Jasmin Blanchette and Mark Summerfield, "C++ GUI Programming with Qt4", Second Edition, 2013, page 45-121, 339-356.
- [38] CD:\documents\troubleshootingIRC5.pdf
- [39] CD:\documents\IRC5andFlexPendant.pdf
- [40] CD:\documents\RobotStudioIntro.pdf
- [41] CD:\document\RAPIDdatatype.pdf
- [42] CD:\videos\vibrations.mp4
- [43] CD:\software\RobotStudio,RAPID\TCPProbot\_2.tsstn
- [44] CD:\software\objviewer\Ovrvision\_to\_rift.exe

[45] CD:\software

[46] CD:\software\OculusRiftRobot3.0\Win32\Debug

[47] CD:\videos\system\_rear.mp4

# Appendices

## A Transformation Matrices

There will be some short hand notation used which is defined in Eq. 76 - 79, and takes advantage of some trigonometric identities.

$$s_i = \sin(\theta_i) \quad (76)$$

$$c_i = \cos(\theta_i) \quad (77)$$

$$s_{23} = \sin(\theta_2 + \theta_3) = s_2 c_3 + c_2 s_3 \quad (78)$$

$$c_{23} = \cos(\theta_2 + \theta_3) = c_2 c_3 - s_2 s_3 \quad (79)$$

The intermediate calculations as well as the parameters of  ${}_0^6T$  are computed below.

$${}^2_0T = {}^1_0T \cdot {}^2_1T = \begin{bmatrix} c_1 c_2 & -c_1 s_2 & s_1 & c_1 a_1 \\ s_1 c_2 & -s_1 s_2 & -c_1 & s_1 a_1 \\ s_2 & c_2 & 0 & d_1 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$${}^3_0T = {}^2_0T \cdot {}^3_2T = \begin{bmatrix} c_1 c_{23} & -c_1 s_{23} & s_1 & c_1(c_2 a_2 + a_1) \\ s_1 c_{23} & -s_1 s_{23} & -c_1 & s_1(c_2 a_2 + a_1) \\ s_{23} & c_{23} & 0 & s_2 a_2 + d_1 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$${}^4_0T = {}^3_0T \cdot {}^4_3T = \begin{bmatrix} c_1 c_{23} c_4 + s_1 s_4 & -c_1 c_{23} s_4 + s_1 c_4 & c_1 s_{23} & c_1(s_{23} d_4 + c_2 a_2 + a_1) \\ s_1 c_{23} & -s_1 c_{23} s_4 - c_1 c_4 & s_1 s_{23} & s_1(s_{23} d_4 + c_2 a_2 + a_1) \\ s_{23} c_4 & -s_{23} s_4 & -c_{23} & -c_{23} d_4 + s_2 a_2 + d_1 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$${}^5_0T = {}^4_0T \cdot {}^5_4T = \begin{bmatrix} {}^5_0R & {}^5_0d \\ 0 & 1 \end{bmatrix}$$

$${}^5_0R = \begin{bmatrix} c_5(c_1 c_{23} c_4 + s_1 s_4) - s_5 c_1 s_{23} & -s_5(c_1 c_{23} c_4 + s_1 s_4) - c_1 s_{23} & -c_1 c_{23} s_4 + s_1 c_4 \\ c_5(s_1 c_{23} c_4 - c_1 s_4) - s_5 s_1 s_{23} & -s_5(s_1 c_{23} c_4 - c_1 s_4) - s_1 c_{23} c_5 & -s_1 c_{23} s_4 - c_1 c_4 \\ c_5(s_{23} c_4) + c_{23} c_5 & -s_{23} c_4 s_5 + c_{23} c_5 & -s_{23} s_4 \end{bmatrix}$$

$$d = \begin{bmatrix} c_1(s_{23} d_4 + c_2 a_2 + a_1) \\ s_1(s_{23} d_4 + c_2 a_2 + a_1) \\ -c_{23} d_4 + s_2 a_2 + d_1 \end{bmatrix}$$

$${}^6_0T = {}^5_0T \cdot {}^6_5T = \begin{bmatrix} r_{11} & r_{12} & r_{13} & P_{xw} \\ r_{21} & r_{22} & r_{23} & P_{yw} \\ r_{31} & r_{32} & r_{33} & P_{zw} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Where the indices are:

$$\begin{aligned}
r_{11} &= c_1 c_{23} (c_4 c_5 c_6 - s_4 s_6) - c_1 s_{23} s_5 c_6 + s_1 (s_4 c_5 c_6 + c_4 s_6) \\
r_{12} &= -c_1 c_{23} c_4 c_5 c_6 + c_1 s_{23} s_5 s_6 - s_1 s_4 c_5 c_6 \\
r_{13} &= c_1 c_{23} c_4 c_5 + c_1 s_{23} c_5 + s_1 s_4 s_5 \\
r_{21} &= s_1 c_{23} (c_4 c_5 c_6 - s_4 s_6) - s_1 s_{23} s_5 c_6 - c_1 (c_4 c_5 c_6 + c_5 s_6) \\
r_{22} &= -s_1 c_{23} c_4 c_5 c_6 + s_1 s_{23} s_5 s_6 + c_1 s_4 c_5 s_6 \\
r_{23} &= s_1 c_{23} c_4 c_5 + s_1 s_{23} c_5 - c_1 s_4 c_5 s_5 \\
r_{31} &= s_{23} (c_4 c_5 c_6 - s_4 s_6) + c_{23} s_5 c_6 \\
r_{32} &= -c_6 s_{23} c_4 c_5 - c_{23} s_5 s_6 \\
r_{33} &= s_{23} c_4 c_5 - c_{23} c_5 \\
P_{xw} &= c_1 s_{23} d_4 + c_1 c_2 a_2 + c_1 a_1 \\
P_{yw} &= s_1 s_{23} d_4 + s_1 c_2 a_2 + s_1 a_1 \\
P_{zw} &= -c_{23} d_4 + s_2 a_2 + d_1
\end{aligned}$$

## B User Manual

### B.1 How To Calibrate the Robot

As previously mentioned it might be necessary to manually recalibrate the robot after it has been turned off. This is done by manually jogging the different revolution joints into a position where a calibration mark on the link matches the calibration mark on the joint. This can be seen in FIG. 79 - 84.

After this is completed the FlexPendant must be utilized to update the revolution counters. To do this, first start at the start up page for the FlexPendant and push the ABB logo in the top left corner. This will open up a control panel tab where the option *Calibration* can be found. Now a list of the mechanical units attached to the robot controller will appear, and it will also say whether they are calibrated or not. If it is not calibrated push that status to enter the manual calibration screen. This should give you an option to update the Revolution Counters.

After the robot is done calibrating it is necessary to check if the calibration was a success. This is done by writing a simple program that jogs all joints to the origin. Once the program is done executing it is possible to check the current value of the angles by pressing the ABB logo in the top left corner. In the option menu screen select *Jogging*. In this tab you can see the current angles of the robot, if they are all at zero degrees then the calibration was successful.



Figure 79: Image of calibration point for the first joint.



Figure 80: Image of calibration point for the second joint.

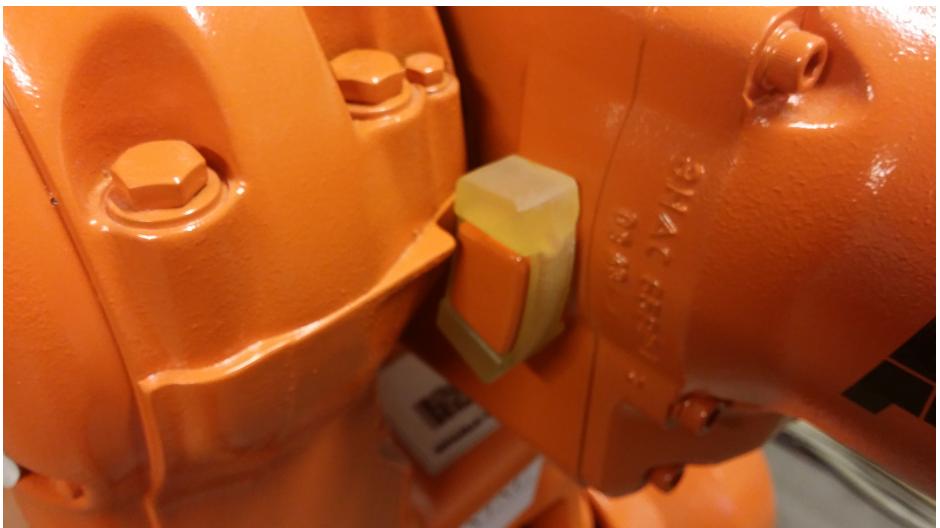


Figure 81: Image of calibration point for the third joint.



Figure 82: Image of calibration point for the fourth joint.



Figure 83: Image of calibration point for the fifth joint.

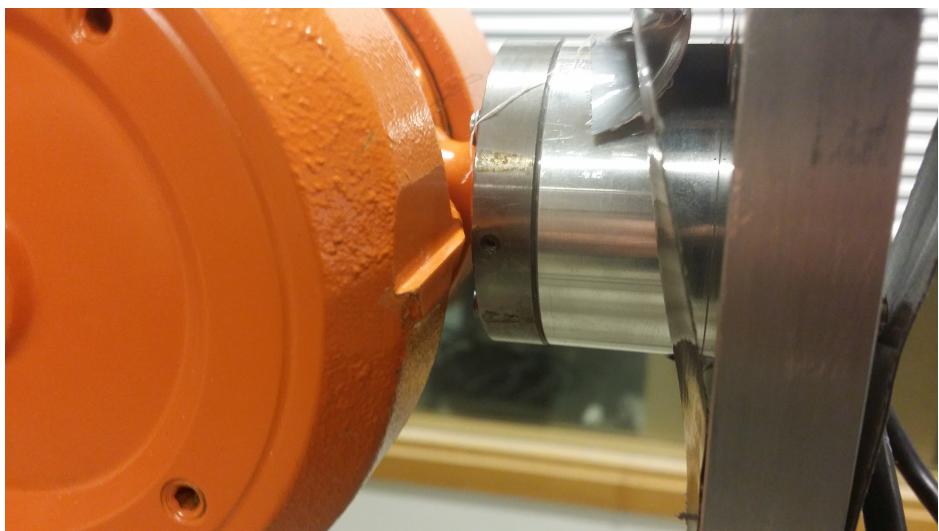


Figure 84: Image of calibration point for the sixth joint.

## B.2 Editing System Parameters In RobotStudio

In order to tune the performance of the robot it is necessary to change some system parameters. There are a large amount of parameters which can be changed, however, it is important to note that some parameters can only be changed if the robot controller has the correct options installed.

In order to change a parameter, first open up a station in RobotStudio which you would like to edit. Then enter either the Controller or RAPID tab, see Fig. 85. Afterwards there will be a menu on the left side where Configuration will appear beneath the name of the station. Most of the Parameters that needs to be tuned will be found under the topic Motion, see Fig. 86. In order for the changes to take affect a warm start the robot controller is necessary. Any information on the system parameters can be found in the help section: File → Help → System Parameters.

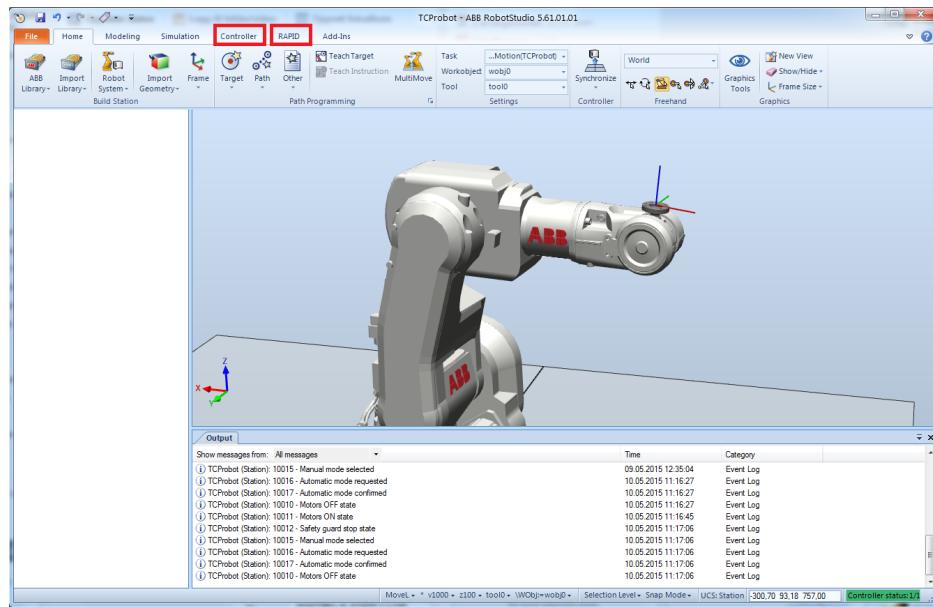


Figure 85: Print screen showing the location of the Controller and RAPID tab.

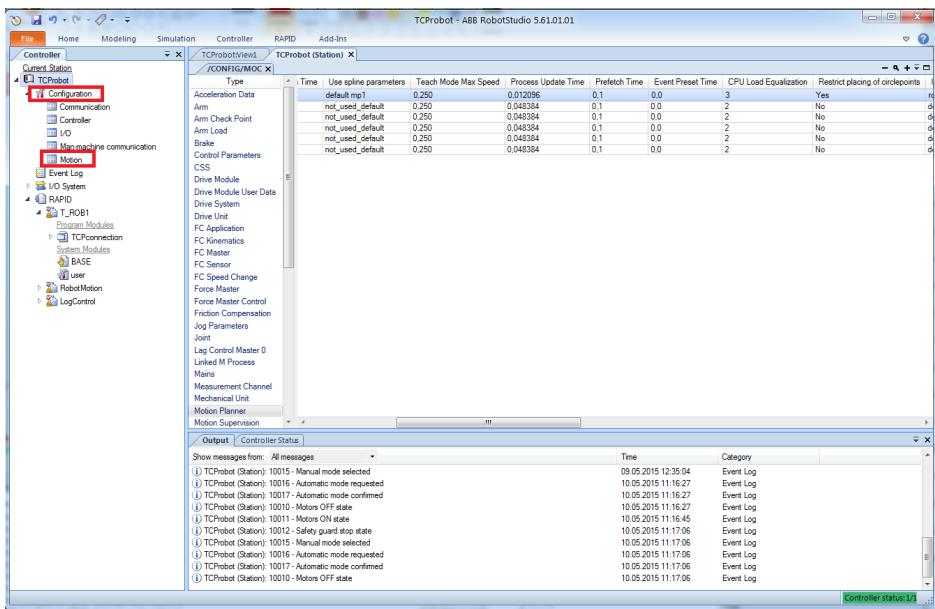


Figure 86: Print screen showing the location where to change system parameters.

## B.3 How to Run the Application

To run the application there are some prerequisites which have to be fulfilled.

- Computer with at least 3 USB ports, and a HDMI port.
- Oculus Rift DK2.
- Ovrvision stereo camera.
- Bidirectional capacitor microphone (optional).
- 2 USB cable extensions.
- An ethernet cable.

### B.3.1 RAPID Server Application

When this is achieved one must connect the robot controller to the computer with the Ethernet cable. The port is located underneath the USB port. Once they are connected it is possible to upload the RAPID server application from RobotStudio to the robot controller. This is done by first establishing a connection through RobotStudio by opening the *Controller* tab and clicking *Add Controller* followed by *One Click Connect*.

Since the application requires two tasks (threads) it is necessary to check if the robot controller has two tasks already or if it needs to be created. In the case that the second task is missing it can be created by opening either the *Controller* or the *RAPID* tab and open the *Configuration* folder. The file that is going to be edited is the *Controller* configuration file. Once this is opened, there is a *Task* subsection which gives the option to create a new task when right clicked. In the pop-up window that ensues there are three instances that needs to be edited. The first is the *task in foreground*, this must be set to the main task which is most often called *T\_ROB1*. The second is that the *Type* must be set to *NORMAL* or else the task cannot be edited. The last is that *MotionTask* must be set to *NO*. This is because there can only be one motion task per robot.

Once it has two tasks the programs can be uploaded to the controller by right-clicking the tasks. Before turning on the application on the FlexPendant make sure that that the controller is set to *Auto* for best performance. By pressing the *PP TO MAIN* option on the FlexPendant and then the play button the application should be running.

### B.3.2 C++ Client Application

If the C++ client application has to be built before usage and in order to do so there are some libraries that are required.

- libOVR 0.4.2

- QT:
  - Core
  - GUI
  - Test
  - Widgets
  - Threads

In addition the Oculus Configuration Utility version 1.4 is needed to communicate with the Oculus Rift. This needs to run as an background process in order to get data from the Oculus Rift.

### B.3.3 Running the system

#### 1. With the Robot:

Connect the Oculus Rift, the cameras and the microphone to the computer. Then, use an ethernet cable to connect the computer to the IRC5 controller. Start the robot and make sure it is set in "manual mode".

When running the software first start the TCPProbot\_2 script [43] in RobotStudio. Then go to the controller option in RobotStudio and press add controller with "one click connect". The IRC5 controller should now pop up in the window to the left. Upload the program to the controller and start the programs from the FlexPendant. After this application has been started one can either build and run the C++ application in a software IDE or run the executable that can be found on the CD[46].

Once the C++ application has started, it will open up a window. To start the system with the robot select *Robot IP* and *Input From Oculus* and press Ok to start the software. To exit the program terminate the program running on the flexendant and the C++ application.

To activate the cameras, start the program Ovrvision\_to\_rift [44]

After the program has been tested and verified in "manual mode", switch to "automatic mode" and repeat the procedure.

#### 2. In the simulation

Connect the Oculus Rift to the computer. Open the TCPProbot\_2 and press play in the simulation tab in RobotStudio. After this application has been started one can either build and run the C++ application in a software IDE or run the executable that can be found on the CD [46].

Once the C++ application has started, it will open up a window. To start the system in the simulation select *Simulation IP* and *Input From Oculus* and press Ok to start the software. To exit the program press stop in the simulation tab in RobotStudio and press cancel in the C++ application.

## B.4 How To Navigate the CD

This report comes with a CD which contains some files. The CD is devided into 6 folders. Some of these folders are refered to in the text, the path on the CD is in those cases given in the bibliography.

*The software folder* contains the software used to run the application. In addition it contains some of the software used when analyzing the data.

*The documents folder* contains documents that is important in order to understand RobotStudio. Some of these documents can also be found online.

*The logFiles folder* contains the logFiles from the different tests. In order to not have a total mess in this folder, only the most relevant logfiles have been included.

*The pictures folder* shows some of the pictures from the report, many of these pictures can also be found in this paper.

*The texFiles folder* contains the latex script generating this pdf

*The Videos folder* contains videos showing how the system works.

## C Code Samples

In this section some samples from the code which was found most relevant will be posted. The rest of the code can be found on the CD [45].

### C.1 C++ Qt

#### C.1.1 main

```
#pragma once
#include <QtWidgets/QApplication>
#include <QtCore/QCoreApplication>
#include <QtCore>
#include "Gui.h"
#include "oculusHandler.h"
#include "logWriter.h"
#include "TCPconnection.h"
#include "fileInputHandler.h"

int main(int argc, char *argv[])
{
    QApplication app(argc, argv);

    oculusHandler *oculusHandler = new oculusHandler;
    Gui *OculusRobotGui = new Gui;
    TCPconnection *TCPconn = new TCPconnection;
    logWriter *log = new logWriter;
    fileInputHandler *fileHandler = new fileInputHandler;

    log->setFileInputHandler(fileHandler);
    log->setOculusHandler(oculusHandler);
    TCPconn->setOculusHandler(oculusHandler);
    TCPconn->setFileInputHandler(fileHandler);
    OculusRobotGui->setLogWriter(log);
    OculusRobotGui->setTCPconnection(TCPconn);
    OculusRobotGui->show();

    return app.exec();
}
```

### C.1.2 TCP connect and run-thread

```

void TCPconnection :: createConnection () {
    std :: string IP = IpAdr . toStdString () ;
    std :: vector<char> writableChar (IP . begin () , IP . end ()) ;
    writableChar . push_back ('\0') ;

    qDebug () <<"IP_is_set_to" << QString :: fromStdString (IP) ;
    if (ConnectToHost (1025 , &writableChar [0])) {
        qDebug () <<"TCP_Connection_Success!" << endl ;
    } else {
        qDebug () <<"TCP_Connection_Failure!" << endl ;
    }
}

bool TCPconnection :: ConnectToHost (int PortNo , char* IPAddress)
{
    //Start up Winsock
    WSADATA wsadata ;

    int error = WSAStartup (0x0202 , &wsadata) ;

    //Error occured?
    if (error)
        return false ;

    //Did we get the right Winsock version?
    if (wsadata . wVersion != 0x0202)
    {
        WSACleanup () ; //Clean up Winsock
        return false ;
    }

    SOCKADDR_IN target ; //Socket address information

    target . sin_family = AF_INET ; // address family Internet
    target . sin_port = htons (PortNo) ; //Port to connect on
    target . sin_addr . s_addr = inet_addr (IPAddress) ; //Target IP

    s = socket (AF_INET , SOCK_STREAM , IPPROTO_TCP) ; //Create socket

    if (s == INVALID_SOCKET)
    {
        return false ; //Couldn't create the socket
    }

    //Try connecting...

    if (::connect (s , (SOCKADDR *) &target , sizeof (target)) ==
        SOCKET_ERROR)
    {
        return false ; //Couldn't connect
    }
    else
}

```

```

        return true; //Success
    }

void TCPconnection :: run(){
    qDebug() << "in_TCP_send_thread";

    while(1){
        if(inputSource == "file"){
            qDebug() << fileInputHandler->
                getClosestMapValue(getCurrentTime());
            sendMessage(fileInputHandler->
                getClosestMapValue(getCurrentTime()));
            listenForAck();
        }

        if(inputSource == "oculus"){
            qDebug() << oculusHandler->
                getOculusDataQString();
            sendMessage(oculusHandler->
                getOculusDataQString().toStdString().
                c_str());
            listenForAck();
        }
    }
}

void TCPconnection :: sendMessage(const char* content){
    szpText = content;
    szpText_length = strlen(szpText);

    send(s, szpText, szpText_length, 0);
}

```

### C.1.3 Oculus RiftHandler data extract

```
oculusHandler::oculusHandler () {
    ovr_Initialize ();
    HMD = ovrHmd_Create (0);
    if (!HMD) {
        qDebug () << "Oculus not detected" << endl;
    } else {
        qDebug () << "Oculus Detected" << endl;
    }
}

void oculusHandler::updateOculusData () {
    trackstate = ovrHmd_GetTrackingState (HMD,
        ovr_GetTimeInSeconds () + 0);

    //setting the data in thePose struct
    headOrientation = trackstate.HeadPose.ThePose.Orientation;
    headOrientation.GetEulerAngles <OVR::Axis_X, OVR::Axis_Y,
        OVR::Axis_Z> ( &thePose.euler.x, &thePose.euler.y, &
        thePose.euler.z );
    thePose.quat.x = trackstate.HeadPose.ThePose.Orientation.x;
    thePose.quat.y = trackstate.HeadPose.ThePose.Orientation.y;
    thePose.quat.z = trackstate.HeadPose.ThePose.Orientation.z;
    thePose.quat.w = trackstate.HeadPose.ThePose.Orientation.w;
    thePose.pos.x = 1000 * trackstate.HeadPose.ThePose.Position.x;
    thePose.pos.y = 1000 * trackstate.HeadPose.ThePose.Position.y;
    thePose.pos.z = 1000 * trackstate.HeadPose.ThePose.Position.z;
}
```

## C.2 Inverse Kinematics

```

Eigen::VectorXd calculateAngles1to3(Eigen::VectorXd ee, Eigen::VectorXd angles, forwardKinematics robot){

    double Pwx = ee(0);
    double Pwy = ee(1);
    double Pwz = ee(2);

    // Calculating theta_1
    angles(0) = atan2(Pwy,Pwx);

    cout << "Theta_1:" << endl;
    cout << angles(0) << endl;

    // Calculating theta_3
    double a1 = robot.geta1();
    double a2 = robot.geta2();

    double d1 = robot.getd1();
    double d4 = robot.getd4();

    double theta_1 = angles(0);

    double D = ( pow((Pwx+a1*cos(theta_1)),2) + pow((Pwy+a1*sin(theta_1)),2) + pow((Pwz-d1),2) - pow(a2,2) - pow(d4,2) ) / (2*a2*d4);

    cout << "D:" << endl;
    cout << D << endl;

    // There are two versions of theta_3 due to elbow down and
    // elbow up case

    cout << "Debug_information_part_1:" << endl;
    cout << Pwx+a1*cos(theta_1) << endl;

    cout << "Debug_information_part_2:" << endl;
    cout << Pwy+a1*sin(theta_1) << endl;

    cout << "Debug_information_part_3:" << endl;
    cout << Pwz-d1 << endl;

    double theta_3 = atan2(sqrt(1-pow(D,2)),D);

    cout << "theta_3:" << endl;
    cout << theta_3 << endl;
    cout << sqrt(1-pow(D,2)) << endl;
    cout << sqrt(1-pow(D,2))/D << endl;
    // double theta_3 = atan2(-sqrt(1-pow(D,2)),pow(D,2));
    // calculate theta_3

    angles(2) = theta_3;
}

```

```

//calculate theta_2

double theta_2 = atan2(Pwz-d1,sqrt(pow(Pwx+a1*cos(theta_1),
    ,2) + pow(Pwy+a1*sin(theta_1),2))) - atan2(d4*sin(
    theta_3),a2+d4*cos(theta_3));

angles(1) = theta_2;

cout << "Angles_1_to_3:" << endl;
cout << angles << endl;

return angles;
}

Eigen::VectorXd calculateAngles4to6(Eigen::VectorXd ee, Eigen:::
    VectorXd angles, forwardKinematics robot){

    Eigen::Matrix3d R = calculateRotationMatrix(ee);

    Eigen::Matrix3d R_0_3;

    double theta_1 = angles(0);
    double theta_2 = angles(1);
    double theta_3 = angles(2);

    R_0_3 << cos(theta_1)*cos(theta_2+theta_3), -cos(theta_1)*
        sin(theta_2*theta_3), sin(theta_1),
        sin(theta_1)*cos(theta_2*theta_3), -sin(
            theta_1)*sin(theta_2*theta_3), -cos(
                theta_1),
        sin(theta_2*theta_3), cos(theta_2*theta_3)
        , 0;

    Eigen::Matrix3d B = R_0_3.inverse()*R;

    cout << "Rotation_matrix_R_3_6:" << endl;
    cout << B << endl;
    double theta_4;
    double theta_5;
    double theta_6;

    theta_4 = atan2(B(1,2),B(0,2));
    theta_6 = atan2(B(2,1),-B(2,0));
    theta_5 = atan2(B(0,2),B(2,2)*cos(theta_4));

    angles(3) = theta_4;
    angles(4) = theta_5;
    angles(5) = theta_6;

return angles;
}

```

## C.3 RAPID

### C.3.1 TCPconnection thread

```
MODULE TCPconnection
  !!! CONSTANTS !!!
  CONST num datasize := 7;

  !!! PERSISTANTS !!!
  PERS num positionData{datasize};
  PERS tasks task_list{2} := [ ["T_ROB1"] , ["TCPconnection"] ];

  !!! VARIABLES !!!
  VAR socketdev server_socket;
  VAR socketdev client_socket;
  VAR string receive_string;
  VAR string client_ip;
  VAR num tempPositionData{datasize};
  VAR syncident sync;

  !!! DEBUG VARIABLES !!!
  VAR num testingNumber := 0;
  VAR string subString;
  VAR num pos_num;
  VAR bool pos_bool;
  VAR bool validInput;

  VAR num timerOne;
  VAR num timerTwo;
  VAR iodev writeFile;
  VAR clock myclock;

PROC Main()
  ! Used in debugging
  ! Open "HOME:\File:="synchdelay.txt", writeFile \Write;
  ! ClkStart myclock;

  SocketCreate server_socket; !Creating a server socket
  SocketBind server_socket, "127.0.0.1", 1025; !Bind server
  socket to this port and ip 192.168.125.1 127.0.0.1
  SocketListen server_socket; !Listen to incomming connections

  !Accepts incomming connection and stores the ip address in
  !client_ip
  SocketAccept server_socket, client_socket\ClientAddress:=
  client_ip;

FOR i FROM 1 TO datasize STEP 1 DO
  IF i=5 THEN
    positionData{i} := 1;
  ELSE
    positionData{i} := 0;
  ENDIF
ENDFOR
```

```

WHILE TRUE DO

    ! Receives a string message from the client and stores it
    ! in receive_string
    SocketReceive client_socket \Str := receive_string;

    IF StrLen(receive_string)=0 THEN
        !Do Nothing
    ELSE
        pos_bool := ConvertStringToNums(receive_string);
        validInput := CheckForValidValues();
        IF validInput = TRUE THEN
            FOR i FROM 1 TO datasize STEP 1 DO
                positionData{i} := tempPositionData{i};
            ENDFOR
            SocketSend client_socket \Str:="Ack";
        ELSE
            !Do not update position data due to invalid
            input
        ENDIF
        !timerOne := ClkRead(myclock \HighRes);

        WaitSyncTask sync, task_list;

        !timerTwo := ClkRead(myclock \HighRes)-timerOne;
        !Write writeFile ,""\Num:=timerTwo;
    ENDIF

    ENDWHILE
    ERROR
    RETRY;
    UNDO
    SocketClose server_socket;
    SocketClose client_socket;

ENDPROC

!! Converts wanted segments of the string into nums !!
FUNC bool ConvertStringToNums(string received_string)

    VAR num steps := datasize;
    VAR num pos_num := 0;
    VAR bool pos_bool := FALSE;
    VAR string substring;
    VAR string TEMP;
    VAR num strPos;
    VAR num string_length;

    string_length := StrLen(received_string);
    received_string := StrPart(received_string, 2, string_length
        -1);

    FOR i FROM 1 TO steps DO

```

```

        string_length := StrLen(received_string);
        strPos := StrFind(received_string , 1,"_");
        IF strPos < string_length THEN
            subString := Strpart(received_string ,1, strPos);
            TEMP := Strpart(received_string ,strPos+1,
                            string_length-strPos);
            received_string := TEMP;
        ELSE
            subString := received_string;
        ENDIF
        pos_bool := StrToVal(subString ,pos_num);
        tempPositionData{i} := pos_num;

    ENDFOR

    RETURN TRUE;
ENDFUNC

!! Makes certain that the input values are legitimate and not
!! garbage values !!
FUNC bool CheckForValidValues()
    FOR i FROM 1 TO datasize DO
        IF tempPositionData{i} > 200000 OR tempPositionData{i} <
           -200000 THEN
            RETURN FALSE;
        ELSE
            RETURN TRUE;
        ENDIF
    ENDFOR
ENDFUNC

ENDMODULE

```

### C.3.2 RobotMotion thread

```

MODULE RobotMotion

    !!! CONSTANTS !!!
    CONST num datasize := 7;

    !!! PERSISTANTS !!!
    PERS num positionData{datasize} := [0,0,0,0,0,0,1];
    PERS tasks task_list{2} := [{"T_ROB1"}, {"TCPconnection"}];

    !!! VARIABLES !!!
    VAR num prevPositionData{datasize};
    VAR num deltaPos{datasize};
    VAR num tempPositionData{datasize};
    VAR bool dataChanged := FALSE;
    VAR robtarget point;
    VAR jointtarget startup;
    VAR clock myclock;
    VAR bool starttimer;
    VAR syncident sync;

    !!! DEBUG !!!
    VAR iodev writeLogFile;
    VAR iodev writeFile;
    VAR num timerOne;
    VAR num timerTwo;

    !!! MAIN PROCEDURE !!!
    PROC Main()
        Setup;
        !Open "HOME:\File:=" robotData.csv , writeLogFile \Write;
        !Open "HOME:\File:=" moveAlgDelay.txt , writeFile \Write;
        starttimer := TRUE;

        WHILE TRUE DO

            UpdateDestination;
            WaitSyncTask sync, task_list;
            IF starttimer THEN
                ClkStart myclock;
                starttimer := FALSE;
            ENDIF;
            LookForChange;
            IF dataChanged AND isReachable(point, tool0, wobj0) THEN
                !timerOne := ClkRead(myclock \HighRes);
                MoveRobot;
                !timerTwo := ClkRead(myclock \HighRes)-timerOne;
                !Write writeFile,""\Num:=timerTwo;
                !WriteToFile;
                !WriteToFileCsv;

            ENDIF;
            notFirstMovement := TRUE;
        
```

```

        !WaitTime is used when utilizing fly-by points
        !WaitTime 0.1;
ENDWHILE
ENDPROC

!!! MOVEMENT ALGORITHM !!!

PROC MoveRobot()
    ConfJ \Off;
    ConfL \Off;

    MoveL point ,vmax, fine ,tool0\WObj:=wobj0 ;
ENDPROC

!!! UPDATES THE ROBTARGET POINT !!!

PROC UpdateDestination()
    VAR bool dataIsZero := TRUE;

    FOR i FROM 1 TO datasize STEP 1 DO
        IF prevPositionData{i} = 0 THEN
        ELSE
            dataIsZero := FALSE;
        ENDIF
    ENDFOR

    IF dataIsZero THEN
    ELSE
        FOR i FROM 1 TO datasize STEP 1 DO
            deltaPos{i} := positionData{i} - prevPositionData{i};
        ENDFOR
        point := Offs(point, -deltaPos{3}, -deltaPos{1},
                      deltaPos{2});
        point.rot.q1 := -positionData{5}; !5
        point.rot.q2 := -positionData{4}; !4
        point.rot.q3 := positionData{6}; !6
        point.rot.q4 := positionData{7}; !7
    ENDIF
    ENDPROC

!!! LOOKS FOR CHANGE IN THE POSITION DATA !!!

PROC LookForChange()
    IF positionData = prevPositionData THEN
        dataChanged := FALSE;
    ELSE
        dataChanged := TRUE;
        prevPositionData := positionData;
    ENDIF
ENDPROC

!!! SETUP PROCEDURE FOR INITIAL SETTINGS !!!

PROC Setup()

```

```

point := CRobT(\ Tool:=tool0\WObj:=wobj0);

FOR i FROM 1 TO datasize STEP 1 DO
    positionData{i} := 0;
    prevPositionData{i} := 0;
ENDFOR
prevPositionData := [0,0,0,0,0,0,0];
positionData := [0,0,0,0,0,1,0];
startup := [[0,0,0,0,-80,-180],[9E9,9E9,9E9,9E9,9E9,9E9]];
MoveAbsJ startup , v400,z1 , tool0 ;
ENDPROC

!!! READ FROM FILE, IS USED IN FTP COMMUNICATION !!!
PROC ReadFromFile()
    VAR bool dataOK :=TRUE;
    VAR iodev readFile;
    Open "HOME:"\File:="outputFile.txt" , readFile \Read;
    Rewind readFile;

    FOR i FROM 1 TO datasize STEP 1 DO
        tempPositionData{i} := ReadNum(readFile);
    ENDFOR

    Close readFile;

    FOR i FROM 1 TO datasize STEP 1 DO
        IF tempPositionData{i} > 200000 OR tempPositionData{i} <
            -200000 THEN
            dataOK := FALSE;
        ENDIF
    ENDFOR

    IF dataOK THEN
        FOR i FROM 1 TO datasize STEP 1 DO
            positionData{i} := tempPositionData{i};
        ENDFOR
    ENDIF
ENDPROC

!!! WRITES TO FILE !!!
PROC WriteToFile()
    VAR num timer;
    VAR robtarget curPos;

    curPos := CRobT(\ Tool:=tool0\WObj:=wobj0);
    timer := ClkRead(myclock);

    Write writeLogFile ,"\Num:="timer , \NoNewLine;
    Write writeLogFile ,"\_ ,\NoNewLine;

    Write writeLogFile ,"\Num:="curPos.trans.y-30,\NoNewLine;
    Write writeLogFile ,"\_ ,\NoNewLine;

```

```

Write writeLogFile ,”\Num:=curPos . trans . z -781.78 ,\NoNewLine;
Write writeLogFile ,”_ ,\NoNewLine;

Write writeLogFile ,”\Num:=-curPos . trans . x -12.4 ,\NoNewLine;
Write writeLogFile ,”_ ,\NoNewLine;

Write writeLogFile ,”\Num:=(PI*EulerZYX(\Y, curPos . rot ))
/180 ,\NoNewLine;
Write writeLogFile ,”_ ,\NoNewLine;

Write writeLogFile ,”\Num:=-PI+(PI*EulerZYX(\Z, curPos . rot ))
/180 ,\NoNewLine; !( PI*EulerZYX(\Y, curPos . rot ))/180 ,\
NoNewLine;
Write writeLogFile ,”_ ,\NoNewLine;

Write writeLogFile ,”\Num:=(PI*EulerZYX(\X, curPos . rot ))/180;
ENDPROC

!!! WRITES TCP COORDINATES TO FILE WITH CSV FORMAT !!!
PROC WriteToFileCsv()
  VAR num timer;
  VAR robtarget curPos;

  curPos := CRobT(\Tool:=tool0\WObj:=wobj0);
  timer := ClkRead(myclock);

  Write writeLogFile ,”\Num:=timer , \NoNewLine;
  Write writeLogFile ,”; ,\NoNewLine;

  Write writeLogFile ,”\Num:=curPos . trans . y -30 ,\NoNewLine;
  Write writeLogFile ,”; ,\NoNewLine;

  Write writeLogFile ,”\Num:=curPos . trans . z -781.78 ,\NoNewLine;
  Write writeLogFile ,”; ,\NoNewLine;

  Write writeLogFile ,”\Num:=-curPos . trans . x -12.4 ,\NoNewLine;
  Write writeLogFile ,”; ,\NoNewLine;

  Write writeLogFile ,”\Num:=(PI*EulerZYX(\Y, curPos . rot ))
/180 ,\NoNewLine;
  Write writeLogFile ,”; ,\NoNewLine;

  Write writeLogFile ,”\Num:=-PI+(PI*EulerZYX(\Z, curPos . rot ))
/180 ,\NoNewLine; !( PI*EulerZYX(\Y, curPos . rot ))/180 ,\
NoNewLine;
  Write writeLogFile ,”; ,\NoNewLine;

  Write writeLogFile ,”\Num:=(PI*EulerZYX(\X, curPos . rot ))/180;
ENDPROC

FUNC bool IsReachable(robtarget pReach, PERS tooldata ToolReach
, PERS wobjdata WobjReach)

```

```

! Check if specified robtarget can be reach with given tool and
! wobj.
!
! Output:
! Return TRUE if given robtarget is reachable with given tool and
! wobj
! otherwise return FALSE
!
! Parameters:
! pReach      - robtarget to be checked, if robot can reach this
! robtarget
! ToolReach   - tooldata to be used for possible movement
! WobjReach   - wobjdata to be used for possible movement

VAR bool bReachable;
VAR jointtarget jntReach;

bReachable := TRUE;

jntReach := CalcJointT(pReach, ToolReach\Wobj:=WobjReach);

RETURN bReachable;

ERROR
IF ERRNO = ERR_ROBLIMIT THEN
    bReachable := FALSE;
    TRYNEXT;
ENDIF
ENDFUNC

ENDMODULE

```