NTNU

Norwegian University of
Science and Technology

Telemanipulation of NAO Robot Using Sixense STEM

Joint Control by External Analytic IK-Solver

Linn Danielsen Evjemo

Master of Science in Cybernetics and Robotics
Submission date: June 2016
Supervisor: Jan Tommy Gravdahl, ITK

Norwegian University of Science and Technology
Department of Engineering Cybernetics

Preface

This thesis is written as the concluding part of my master's degree in engineering cybernetics at the Department of Engineering Cybernetics at the Norwegian University of Science and Technology. The project was carried out in collaboration with SINTEF Fisheries and Aquaculture from January to June 2016.

Many thanks to my supervisor at NTNU, Jan Tommy Gravdahl, for his help, guidance, and encouraging words throughout this project. I would like to thank SINTEF Fisheries and Aquaculture for the opportunity to do this project, and Elling Ruud Øye and John-Reidar Mathiasen for their help and guidance. Thank you to Nikolaos Kofinas who took time to answer some of my questions about the functionality of his IK-solver, and thank you to Åsmund Pederson Hugo, who let me test his own IK-solver for the NAO-robot.

A special thanks to my family for all their patience, love and support, and for the hours spent giving me feedback on this thesis. And lastly, I would like to thank my friends at NTNU and Samfundet, who made these past five years unforgettable.

L.D.E.

Abstract

If it was possible to train robots to do new tasks instead of programming them directly, robots would be able to perform a greater variety of tasks that are usually carried out manually, both in industry and in society in general. This could for example be achieved by demonstrating the motion or task repeatedly through robot telemanipulation. In such a system, minimal latency is crucial in order for the robot control to feel natural. The focus of this project has been to find an effective way of performing telemanipulation of the arms of a NAO robot from Aldebaran Robotics using the motion-tracker system Sixense STEM. The motivation was to make the robot follow the movements of a hand-held motion tracker in as close to real-time as possible.

This Master thesis is a continuation of the work done in a summer project during the summer of 2015, and a project thesis written in the fall of 2015. This project has attempted using joint control methods to control the robot, as previous work has shown that Cartesian control methods for the NAO robot are very slow. An analytic *inverse kinematics* (IK) solver for the NAO robot developed by Nikolaos Kofinas at the University of Crete has been tested to transform the STEM-tracker's trajectory to the necessary joint angles. In the end it was unfortunately not possible to create a functioning system. An analytic IK-solver will only return solutions for feasible combinations of position and orientation. Because the robot's arm has fewer degrees of freedom than a human arm, it is impossible for the robot arm to follow the exact motions of a human arm. This meant that most of the tracker data sent to the IK-solver from the hand-held STEM-tracker was infeasible, and hence the IK-solver was not able to return valid solutions.

Some of the focus in this project has also been on determining whether or not the Sixense STEM system is suitable for this kind of work. The system used in this project is a BETA-version, and the commercial version of the system has not been released yet. When the system is fully developed, it will probably be a good choice of hardware, as it is very accurate when it works as it should. However, the system currently has many bugs and problems that are both limiting and time-consuming. The conclusion is therefore that the STEM-system is not suitable for this kind of project work at this stage.

Sammendrag

Dersom det var mulig å *trene* roboter til å utføre oppgaver framfor å programmere dem direkte, ville det vært mulig for roboter å utføre flere typer oppgaver enn de gjør i dag, både i industrien og i samfunnet generelt. Denne treningen kunne for eksempel vært gjort ved å demonstrere bevegelsen eller oppgaven for roboten ved hjelp av fjernstyring av robotens bevegelser. I et slik system er det avgjørende med minimal tidsforsinkelse dersom fjernstyringen skal føles naturlig. Målet i dette prosjektet har vært å finne en effektiv måte å fjernstyre armene til en NAO-robot fra Aldebaran Robotics ved hjelp av motion tracking-systemet Sixense STEM. Tanken var å få roboten til å følge bevegelsene til en håndholdt motion tracker i så nær sanntid som mulig.

Denne masteroppgaven er fortsettelsen på arbeid gjort i et studentprosjekt sommeren 2015, og i en prosjektoppgave gjennomført høsten 2015. I masterprosjektet er det forsøkt å bruke metoder for vinkelkontroll av NAO-roboten, da tidligere arbeid har vist at kartesisk posisjonskontroll er veldig treigt. En analytisk *inverskinematikk-solver* (IK) for NAO-roboten utviklet av Nikolaos Kofinas ved Universitetet på Kreta har blitt brukt for å forsøkte å transformere motion-trackerens bevegelser til de nødvendige vinkelverdiene. Til slutt viste det seg dessverre at det ikke var mulig å komme fram til et fungerende system. En analytisk IK-solver vil kun returnere løsninger for realiserbare kombinasjoner av posisjon og rotasjon. Fordi robotarmen har færre frihetsgrader enn en menneskearm, er det umulig for den å følge de nøyaktige bevegelsene til en håndholdt tracker. Dette betyr at de fleste bevegelsene som ble sendt fra motion trackeren til IK-solveren ikke var realiserbare, og det kunne dermed heller ikke bli returnert noen gyldig løsning.

Et delmål i dette prosjektet har også vært å evaluere hvorvidt STEM-systemet er egnet for bruk i denne typen prosjekt. Systemet som er brukt her er en BETA-versjon av STEM-systemet, som per i dag ikke har blitt sluppet for kommersiell bruk. Når systemet er ferdig utviklet, er det grunn til å tro at det vil være et godt valg for tracking-hardware i denne typen prosjekter, fordi det er veldig nøyaktig når det fungerer som det skal. Dessverre har systemet for øyeblikket mange problemer som er både tidkrevende og begrensende. Konklusjonen er derfor at STEM-systemet ikke er egnet for bruk i denne typen prosjekter på det nåværende tidspunkt.

Acronyms

API Application Programming Interface

DCM Device Communication Manager

DLL Dynamic Link Library

DOF Degrees of Freedom

DSP Digital Signal Processing

FK Forward Kinematics

HRR2030 Humanoid Robotics Roadmap 2030

IK Inverse Kinematics

MCU Micro Controller Unit

OS Operating System

SDK Software Development Kit

TCP Transmission Control Protocol

Contents

Chapter 1

Introduction and background

1.1 Introduction

The introduction is largely based on the introduction to my project thesis (Evjemo, 2016).

An increasing number of tasks at fish processing facilities and other parts of the aquaculture industry are performed automatically by industrial robots and machines. We see the same trend for most other industries. But many small and seemingly easy tasks still have to be performed manually. This can be because the task is too difficult or complex for today's technology, like doing the final sorting of fish that do not fit the standard measurements. But it can also be tasks that are simply so specific or isolated that it is not financially beneficial for the company to buy a machine or robot to do only this single task. These tasks are often repetitive and boring, and may lead to repetitive strain injuries for the human workers, as well as not being very inspiring or rewarding.

If it was possible to develop humanoid robots that could be *shown* and *taught* how to perform new tasks, instead of having to program them manually, the situation would change dramatically. This would make it possible for one single robot to perform a variety of tasks within the same factory. If one robot could learn through demonstration, it would also be a lot more economical for businesses to buy one. It would make it possible to teach the robot new tasks as the industry changed, and new challenges arose. And of course, this would make it unnecessary for

human workers to do tedious and repetitive tasks all day.

Big ideas have to start small. There is still a long way ahead until we get to a society where robots can do all the menial work for us. But we can start working with these self-taught robots in mind. The goal for this master's thesis it to try to create a highly effective way of performing telemanipulation of a humanoid NAO robot. If this can be achieved with minimal latency, it can be a starting point for training the robot to repeat tasks, and eventually learn how to perform them by itself. Minimal latency is important in order to obtain anthropomorphism: The ability to execute a specific function in a way that makes it appear close to human. This is necessary if we want the robot to follow our movements in a way that feels natural for us.

1.2 Background

This section is partially taken from the Background*-section in my project thesis (Evjemo, 2016).*

I was one of four students who during the summer of 2015 worked on creating basic telemanipulation of the arms of a NAO robot for SINTEF Fisheries and Aquaculture. This was part of a larger project at SINTEF called Humanoid Robotics Roadmap 2030 (HRR2030). By combining motion tracking with picture recognition, we were able to train the robot to pick up objects that were placed in front of it. We were also able to implement basic telemanipulation of the robot's arms, which was eventually intended to become part of the training process. The summer project did not get that far, so the actual training was performed in a different way, which is not relevant to this thesis.

However, the initial system was quite slow, which meant that the telemanipulation did not feel natural for the person controlling the robot. As a result, I continued working on the project during my project thesis in the fall of 2015. The focus of the project thesis was to identify the bottlenecks in the initial system when it comes to data flow and communication, and to map what limitations we face when trying to achieve minimal latency.

The conclusion of the project thesis was that the main reason for the system's latency was the methods used to control the arms of the robot. More specifically, the robot's internal *inverse kinematics* (IK) solver was too slow. As a result, the system from project thesis was still so slow that the control of the robot's arms did not feel natural. The robot was not able to execute the commands in anything close to real-time as long as the inverse kinematics computations were done simultaneously by the robot's controller. In this master's thesis, I will try to improve this by using alternative methods for robot control.

1.3 Problem description

The main goal for this project is to create a system for effective telemanipulation of the arm effectors of an Aldebaran NAO robot, using the motion tracker system Sixense STEM. To make the telemanipulation feel natural for the person controlling the robot, it is important to have a system with low latency. Many methods for controlling the NAO robot's actuators are included in the *software development kit* (SDK) from Aldebaran Robotics. Cartesian control methods from the SDK were tested in earlier work, but they turned out to be much too slow. These methods depend on the robot's controller to do the necessary calculations to get the robot arm's end effector to the desired position and orientation. The conclusion was therefore that the robot's internal IK-solver was the main delay.

In this master's thesis, the telemanipulation will be attempted using an external IK-solver, and joint control methods from the SDK. The system will use the analytic IK-solver for the NAO robot developed in the diploma thesis of Nikolaos Kofinas at the Technical University of Crete (Kofinas, 2012). The goal is to use the external IK-solver to compute the joint angles, and then send these directly to the robot. The IK-solver has to be tested to make sure that it works as expected. A system combining the tracker data from STEM, the analytic IK-solver, and the robot, must then be implemented. Part of the focus of the thesis will be to re-evaluate if the STEM-system is a suitable tool for telemanipulation of the robot, and to document any current problems with the system.

The final system will first be tested on a controllable 3D-model of the robot arm, before testing it on the physical robot. Lastly, the final system should be evaluated, before suggesting ways the system can be improved in further work.

Similar projects

This section is largely based on the Similar projects*-section in my project thesis (Evjemo, 2016).*

When doing research for both the project and master's thesis based on the work done during the summer of 2015, I found two similar projects. Both projects used NAO robots, and tried to make them follow human motions as closely as possible.

The first was a video showing the results of the bachelor's thesis of Jonas Koenemann at the University of Freiburg (Whole-body Imitation of Human Motions with a Nao Humanoid: Real-time teleoperation, 2011). His thesis focused on making a NAO robot imitate the whole-body motions of a human. He did this using a motion tracking suit which registered the movement of the major joints on his body. The joint angles were then sent directly to the robot. The system compensated for the differences in joint-length etc. by mapping each of the person's joints to the robot's joints. Koenemann's system seems to work well, but is very slow. In the film we can see that the robot is always a couple of seconds behind the movements of the person it is trying to imitate.

The second project was created by engineer and master's graduate Halit Bener Suay. His system only used a Kinect-camera in order to control the robot (Humanoid Robot Control and Interaction, 2010). He had also tried to make the control feel as natural as possible. From the video it looked like his system had less latency than that of Koenemann, even if there was clearly a small delay. I contacted Suay to ask about his project, and he gave me access to all the code for his system (Nao_rail, 2010). He told me that even though there seems to be very little delay between the human motions and the robot motions in the video, he considered his system to be quite slow. He said that there was a considerable delay, and that he therefore kept his movements as slow as possible when making the video, to try to minimize this. Suay had also used joint con-

trol methods in order to control the robot's effectors. The e-mail correspondence is included in appendix A.

Limitations

Even though the goal is to control both of the robot arms, the system created in this thesis only focuses on controlling the left arm effector. This choice was made partially in order to make testing easier, but also based on the fact that the STEM-system can be quite problematic to work with, so that it would be beneficial to only need one tracker at a time. The problems related to the STEM-system will be explained in section 1.4.1.

The system created in this thesis has not been tested on the physical NAO robot. This is because it was important to make sure that the robot control worked properly first, and not risk damaging the actual robot. Based on the work done during the summer and fall of 2015, I have learnt the hard way that the robot effectors can move unexpectedly if the control input is invalid in some way. Instead, the system is tested on a 3D-model created in LabVIEW and on a virtual NAO robot in Choreographe, see section 2.3.

1.4 Equipment

This section is largely based on section 1.5 of my project thesis (Evjemo, 2016).

1.4.1 Sixense STEM motion tracking system

In this project, a new, highly accurate motion tracker system called Sixense STEM is used to track the desired movement. The STEM-system is wireless, and consists of five motion trackers and a base station, shown in figure 1.1. The model used is a BETA version of the system, because the final version of the system has yet to be released.

The STEM-system is very accurate, and seemed like a good tool for performing telemanipulation of the arms of the NAO robot. This equipment was therefore acquired by SINTEF Fisheries and

Figure 1.1: **Sixense STEM:** Here we see the STEM-system used in this project. The system has two hand-held controllers, and three *packs*. Picture from (Sixense STEM, 2014).

Aquaculture in the spring of 2015 in preparations for the student summer project the same year.

Figure 1.2: **STEM-controller:** The controller's electromagnetic tracker is placed as shown in red. Image from (Kickstarter: Sixense STEM, 2014).

According to the developers, Sixense STEM has the lowest latency of any wireless consumer

motion control system (STEM System, 2014). The version of the system used in this project has hand-held controllers with additional functionality like joysticks and buttons, and three *packs*, which are only for motion tracking. Because controllers and packs are mainly being used for basic motion tracking in this project, they will all from hereon out mostly be referred to just as *trackers*. When using the trackers, it is important to keep in mind where actual electromagnetic tracker is placed in the different trackers, as shown in figure 1.2 and 1.3. This is where the centre of rotation will be, as well as where the Cartesian position is registered.

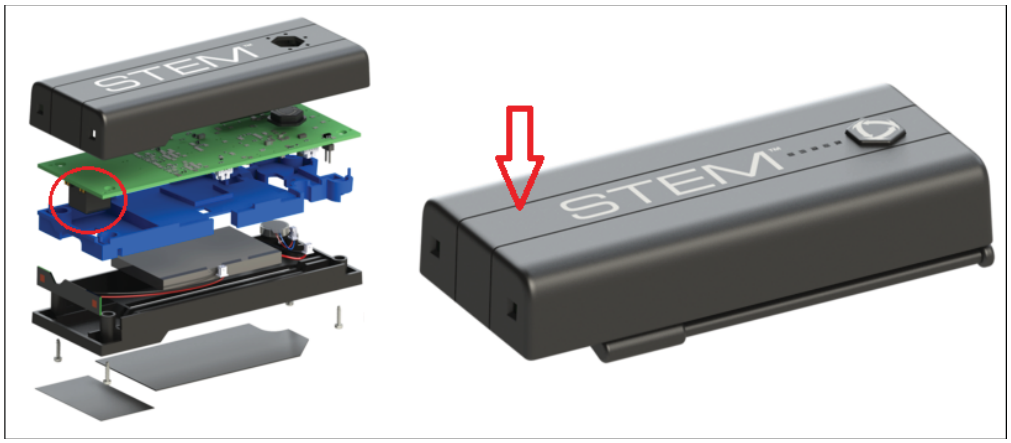

Figure 1.3: **STEM-pack:** The pack's electromagnetic tracker is placed as shown in red. Image from (08/01/2014 - STEM Pack Details, 2014).

When turning on the STEM-system, the base station has to be connected to the computer, and a Sixense application of some kind has to be running on the computer. This will be explained further in section 3.2.2.

During the work on my project thesis during the fall of 2015, I experienced a lot of problems with the STEM-trackers. They would sometimes disconnect, or refuse to connect to the base station at all. The system was sensitive to metal and WiFi, and would sometimes register tracker data that was obviously wrong. Because of these weaknesses, the conclusion in the project thesis was that the STEM-system in its current state was unsuited for use in any project where it had to be combined with different hardware.

However, several new firmware updates came in the two months between the end of the lab

work on my project thesis and the beginning of my Master thesis. The system seemed to have become a lot more stable, and the controllers would no longer disconnect as frequently as before. One of my advisers at SINTEF had also tested the system's accuracy after the latest firmware updates, and the conclusion was that the trackers would give out very accurate tracker data as long as they were used withing a radius of 1.5 meters from the system's base station. This did not seem like it should be a problem, at least not for a small scale system like the one I wanted to create.

SINTEF still looked for alternative tracking hardware to be used for the continuing work on this project. PlayStation Move was one alternative, with its thoroughly tested and stable motion tracker controllers. But it was discarded because PlayStation Move does not yet have stable compatibility with PC (Framework for PlayStation Move on PC, 2015), and might therefore create more problems than it would solve due to challenges with the implementation. They were also in contact with the project Advanced Realtime Tracking, which has developed a motion tracking system that can collect joint data for the entire human body through trackers placed directly on the joints of the human controller (Advanced Realtime Tracking, 2016). But this system was also discarded, because the price tag was a bit stiff, especially when considering that there was an alternative, and already familiar, system in place.

So it was decided that the STEM-system would be used for this project as well, and that one of the objectives of the project would be to evaluate whether or not the latest firmware updates had made it a more suitable tracking tool for this kind of work. This was based on a combination of the fact that there did not seem to be any obvious choices for alternative tracking hardware, the system's improved functionality, and my own familiarity with the system.

1.4.2 NAO - humanoid robot

In this project the goal is to control the arms of a 58 cm tall humanoid NAO robot from Aldebaran Robotics. The robot is designed to be interactive and social, and can be used both as an educational tool, and as a toy for both developers and kids.

Figure 1.4: **NAO:** The small, humanoid robot is developed by Aldebaran Robotics.

The NAO robot used in this system is of the model H25, which is an upgrade from the original full-body H21-model. The difference between the two is that the H25-model has implemented more functionality in the arms: both a rotation wrist, and the ability to close the hand (Effector Chain Definitions, 2012).

The thesis this project is based on, written by N. Kofinas, works with a robot of the H21-model. This means that the developed inverse and forwards kinematics do not take into account that the robot has a rotation wrist. Kofinas has later finished the code to account for the additional

four degrees of freedom (DOFs) so the script is up to date and suited for use on the robot model that is used in this thesis.

The robot has 25 DOFs altogether (How does NAO work, 2012). It has two cameras which enables it to register its surroundings, and touch sensors to register obstacles or touches from people around it. Directional microphones enable the robot to hear voices and sounds in its surroundings, and to identify which direction the sound is coming from. In this project, all of these functionalities will pretty much be disregarded. The focus will be on the robot's ability to move its effectors to a given position in space.

The robot was acquired by SINTEF Fisheries and Aquaculture for use in the student summer project in 2015. The goal of the summer project was to combine robot control and machine learning in order to *teach* the robot how to perform a simple task, like picking up an object from a table. NAO seemed like a good tool for this kind of test-project, as it quite functional and not too expensive. For more extensive work in the same area it will be necessary to invest in a more robust robot.

1.4.3 Kofinas' IK-solver

As explained in the *Background* section, the work with telemanipulation of NAO robot so far had shown that the robot's internal, numerical IK-solver was very slow. In order to perform effective telemanipulation with low latency, it was necessary to control the robot using angular methods. In order to do so, an external IK-solver was needed. In the early stages of the work on the project thesis, the idea was that I would create such an IK-solver myself. However, after some research it became clear that somebody else had saved me the work.

In 2012, Nikolaos Kofinas created the full, analytic IK-solver for a NAO H21-robot while he was a student at the University of Crete, Greece (Kofinas, 2012). In addition to his own thesis, Kofinas and some of his professors published a scientific article on the same subject (Kofinas, Orfanoudakis, Lagoudakis, 2013). The IK-solver itself was implemented in C++, and all the files made available to the public through GitHub (Team Kouretes, 2014). Later, the code was ex-

panded to include functionality for the NAO H25-model as well. The functionality of Kofinas' IK-solver will be explained further in section 3.3.1.

1.5 Structure of the report

The rest of this report is structured as follows. Chapter 2 will start by introducing the choice of programming languages and software for this project, as well as the system structure. Some of the choices regarding movement frames and control methods for the robot will be explained, before taking a closer look at the alternatives for simulating the control system on a virtual robot. The different alternatives for communication with the STEM-system will also be introduced. Chapter 2 ends with some necessary theory related to forward and inverse kinematics, different ways of describing rotations, and frame transformations.

In the beginning of chapter 3, the most relevant control methods for the NAO robot will be introduced. It will be explained how the tracker data was collected from Sixense STEM, and there will be a brief look into how the IK-solver works, as well as how it was tested. Some of the tracker data collected from the STEM-system had to be processed before it could be used with the rest of the system, and how this was done will be explained here. A large part of chapter 3 is devoted to presenting the work and testing done while trying to get the IK-solver to work with data from the STEM-tracker in the complete system. There were some technical problems with the STEM-system, which will be presented at the end of chapter 3. These problems are not given much attention in other parts of this thesis. This is partly because the problems were less prominent in this project compared to previous work, and partly because focusing on these issues continuously would draw attention away from the general progress of the project.

In chapter 4 it will be discussed why the final system did not work as expected. The difference in joint control methods compared to Cartesian control will also be evaluated. The suitability for the STEM-system for use in this kind of project will be discussed, and some of the system's improvements and new weaknesses will be highlighted. Some of the more general choices regarding the system structure will also be discussed.

In chapter 5, the conclusion of this project is presented. Even though I did not get the analytic IK-solver to work with the rest of the system, there are many modifications that should be tested at a later point that might help create a functional system. The recommendations and suggestions for further work is therefore included. Suggestions for alternative tracking approaches are presented, and DCM-programming is re-introduced as an alternative approach.

Chapter 2

Method

2.1 Software and system structure

Just like for the project thesis leading up to this master's thesis, many of the choices made for the system structure depended on each other. One example is that the kind of data that had to be extracted from the STEM-trackers, depended on the methods that were chosen to control the robot. However, it was easier to make decisions this time around, because it was already clear which methods that would be best suited for controlling the robot, as explained in the introduction. This meant that almost all other decisions could be done based on what was needed for controlling the robot in the way that I wanted, as will be explained more thoroughly in this section.

2.1.1 Choosing programming languages

Deciding which programming languages to use ended up being quite easy. Some decisions were made based on experiences from the summer project and project thesis, while others were made based on my own level of programming skills in the different languages.

Python

This section is based on section 2 . 1 . 1 of my project thesis (Evjemo, 2016).

During the work on the project thesis in the fall of 2015, it became clear that it was more beneficial to control the robot by writing scripts in a standard programming language than to use the Choreographe-program which came with the robot. I will not discuss why in this report, but the main issue was that the block-based control methods in Choreographe were thought not to be flexible enough. For more details, see my project report (Evjemo, 2016).

When wanting to write scripts in a standard programming language, it was necessary to find the language most suitable for the desired functionality of the system. The operating system that runs on the NAO robot is called NAOqi. Aldebaran Robotics has also released a Software Development Kit (SDK) to all users' disposal, which allows the developers to control NAO on a more basic level than the included Choreographe program (Cool Tools, 2016). The SDK is compatible with several programming languages, as shown in figure 2.1.

Programming Languages	Bindings running on		Choregraphe support	
	Computer	Robot	Build Apps	Edit code
Python	✔	✔	✔	✔
C++	✔	✔	⊘	⊘
Java	✔	⊘	⊘	⊘
JavaScript	✔	✔	✔	⊘

✔	OK
⊘	Not available

Figure 2.1: **Programming languages:** The NAO robot has the following compatibility with some of the most common programming languages. Python seems to have the best compatibility. Table from (Programming, 2015).

As will be explained later in this section, some of the programming on the IK-solver in this system would have to be done using C++. Still, I had far more experience using Python, both in general work and directly for controlling the NAO robot. Python therefore seemed like a safe choice for this project as well. Even though Choreographe was not used for implementing robot control methods in this project, it was useful that Choreographe and Python were compatible.

This left the possibility for combining the Python-scripts with block-diagram control in Choreographe if that were to become necessary later. In section 2.3.2 it will be explained how the virtual robot in Choreographe proved useful for testing, even if the plan was to steer clear of Choreographe all-together.

LabVIEW

During the student project in the summer of 2015, we were introduced to the programming language LabVIEW. This program has a very user-friendly GUI, as well as block diagrams and compatibility with MatLab-scripts. When starting out with the project thesis in the fall of 2015, the hypothesis was that this program might be the reason why the system was too slow to perform natural-feeling control of the robot (Evjemo, 2016). However, at the end of the project, the conclusion was that the main issue was the robot's internal IK-solver.

For my master's project, I therefore decided that LabVIEW was a good choice for implementing much of the system's functionality. This was also based on the fact that the engineers working at SINTEF Fisheries and Aquaculture are using this program in many of their projects, which should mean that it is quite suitable for projects where latency is an issue.

In addition, LabVIEW has a module called MathScript RT, which makes it possible to include basic `.m`-files to the block-diagram environment (Working with .m Files in LabVIEW, 2014). MathScript is a programming language with syntax similar to MatLab which contains more than 800 basic built-in functions (Antonacci and Morrell, 2009). LabVIEW also has a MatLab-block, which works in a similar way, but also includes more of the functions from MatLab. In order to use the MatLab-block, one must have a licenced version of MatLab installed on the computer. Using either MathScript RT or the MatLab-block in LabVIEW would allow for some calculations of for instance rotation matrices or Euler angles to be done directly in LabVIEW. The calculations in question will be presented in section 2.5.2.

C++

The IK-solver created by N. Kofinas was to be the basis for this master's thesis. In addition to his thesis (Kofinas, 2012), he had made all of his scripts available through GitHub as part of the Kouretes Robocup Team (Kouretes Team, 2014). As will be explained in section 3.3, the complete IK-solver was written in C++. Therefore it was to be expected that some programming had to be done in C++, at least to modify how the IK-solver received input and returned the output. LabVIEW has no easy way to communicate with C++, but it seemed likely that it would be necessary to find a way for the programs to communicate, like we did for Python and LabVIEW in the summer project. It was either that, or to redo the IK-solver in a compatible programming language, or directly in LabVIEW. This subject is discussed further in section 3.3.3.

MatLab - for checking calculations

It would be necessary to implement quite a lot of matrix calculations and transformations in this system. It would therefore be very helpful to have some way to check that these functions and transformations did in fact return the expected values. Luckily, MatLab's Robotics System Toolbox includes a lot of this functionality. Here are some of the most useful functions. The explanation for how they work are found in the MathWorks Documentation (MathWorks, 2016).

- **eul2rotm/rotm2eul:** Converts Euler angles to an orthonormal rotation matrix and vice versa. The Euler angles are given in the order 'ZYX'.

- **quat2rotm/rotm2quat:** Converts quaternions to an orthonormal rotation matrix and vice versa. The quaternions are given in the order q_ω, q_x, q_y, q_z.

- **eul2quat/quat2eul:** Converts Euler angles to quaternions and vice versa. The Euler angles are given in the order 'ZYX', and the quaternions are given in the order q_ω, q_x, q_y, q_z.

In addition, the methods for finding the inverse and determinant of a given matrix were used frequently to check the output of the implemented functions.

2.1.2 Structural changes in the system

Parts of this section are based on section 2.1.2 of my project thesis (Evjemo, 2016).

This master's thesis is in a sense the third attempt to implement telemanipulation of the NAO robot as effectively way as possible. Underway there has been some changes in the system structure, and it was only to be expected that the same would happen this time.

In our original program, the run-time was about 100 ms, which is not even close to what a natural-feeling telemanipulation system should be aiming for. Latency of about 10 ms or less would be ideal to make the telemanipulation of the robot to feel as natural as possible. In the summer project we were also using the LabVIEW and Python-scripts. Because LabVIEW is not compatible with Python, we originally had to use TCP-connections to send information between the different parts of the system. The communication flow in the old program is roughly summed up in figure 2.2.

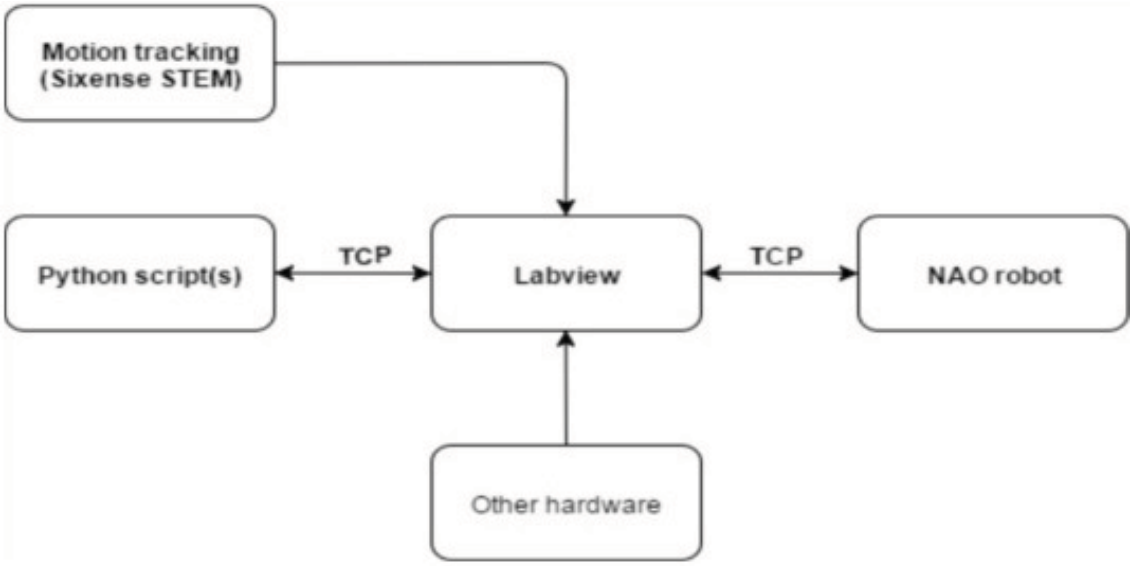

Figure 2.2: **Old program:** In the original program, LabVIEW was used as a centre of communication. This meant that it was necessary to have TCP-connections both between the Python-script and LabVIEW, and between LabVIEW and the NAO robot.

The system we had created in LabVIEW during the summer project was quite complex, and

included code from Python-scripts, additional MatLab-scripts, and block-diagrams. Because of the complex structure, the variety of scripts used in the system, and the additional TCP-connections, I suspected that the LabVIEW-program was the main reason why the original system so slow. The system created while working on the project thesis therefore looped around LabVIEW, and aimed to control NAO directly using scripts in a compatible programming language. The communication flow for the project thesis is shown in figure 2.3. As explained in section 1.2, the work in the project thesis led to the conclusion that LabVIEW was not the problem after all.

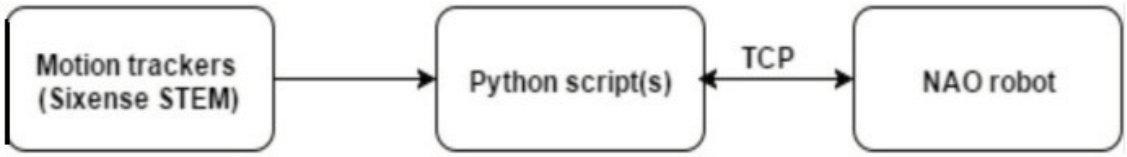

Figure 2.3: **Second attempt:** In the program created in the project thesis in the fall of 2015, all communication between the different hardware happened directly through one or several Python-scripts.

In the newest system created in this master's project, LabVIEW was re-introduced, as explained in section 2.1.1. Communication with C++-scripts was also necessary, and Python-scripts would still be used for the actual control of the robot.

However, it was difficult to say exactly how the system flow would turn out. The decisions related to communication and general methods will be discussed further in section 2.3 and 2.4. Still, the main system structure should be something like the structure shown in figure 2.4. Data would be collected from STEM, and sent to the IK-solver via one of the methods discussed later in this chapter. The IK-solver should then give the necessary joints as output. This data would be used to control the robot with joint control methods from the SDK in Python, or tested on a 3D-model of the robot created in LabVIEW. It was at this point unclear exactly how the IK-solver would communicate with the rest of the system, so this will be discussed later in the report. The structure in figure 2.4 was just a sketch to get an idea of the desired system flow.

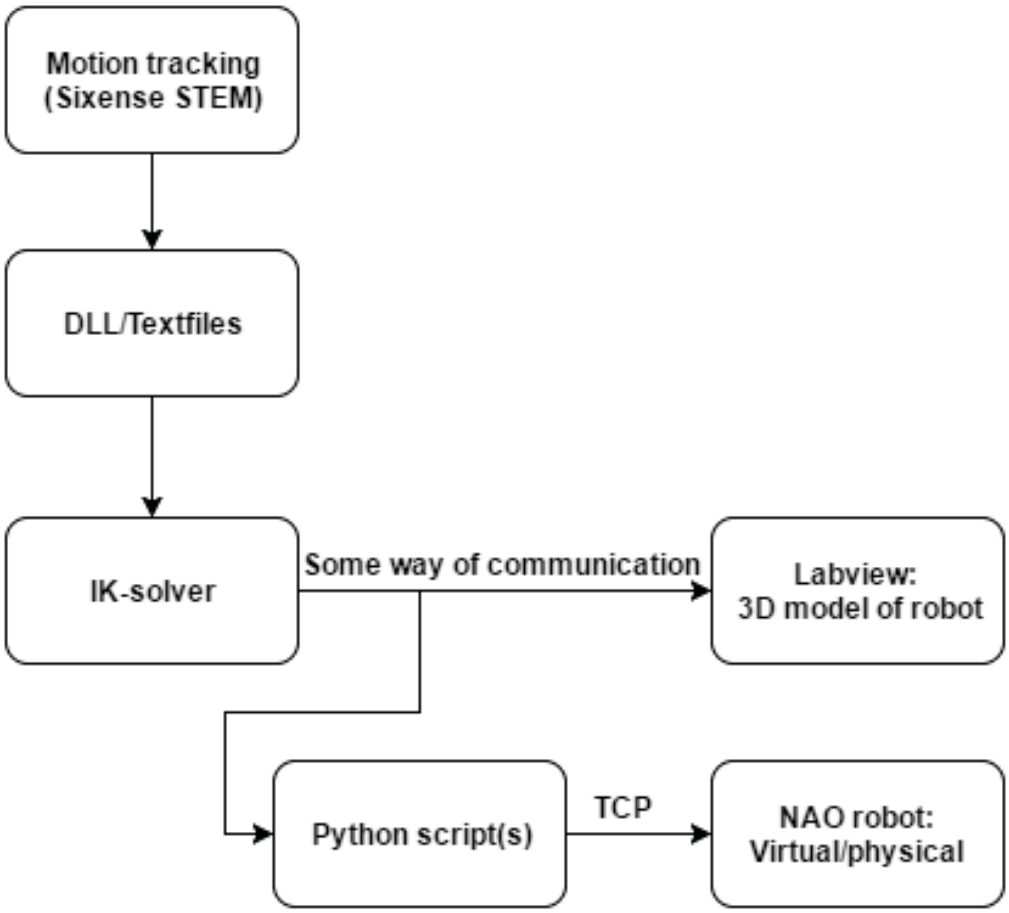

Figure 2.4: **New system structure:** In this new system, data from the STEM-system would be collected using DLL-files or text-files. The data would then be sent to the IK-solver, which would hopefully return the angles necessary to realize the movement. These angles would then either be sent to LabVIEW for testing, or to a Python-script to control the actual robot.

2.2 Controlling the robot

The goal of this project was to test the system directly on the robot, so as to see if the joint control methods could actually allow close to real-time control of the robot. As will be explained later in the report, getting as far as to actually test the system on the physical robot turned out to be more difficult than expected. Still, the whole system is built with control of the physical robot in mind. It is therefore necessary to explain how the control methods work, and present some of the many different methods for controlling the robot's motions included in the SDK (Python SDK, 2015). In this section the choice of start position and working frame for the robot will also be explained.

2.2.1 Choosing control methods

Some parts of this section are based on section 2.2.2 of my project thesis (Evjemo, 2016).

Just like when working on the project thesis, it was necessary to consider the fact that there is an alternative way of controlling the robot: Instead of using the methods from the SDK, it is possible to control the robot's effectors directly through the *Device Communication Manager* (DCM). However, according to the documentation for NAO, this way of controlling the robot is very complicated, and can lead to disaster if you do not know what you are doing (DCM - Introduction, 2012). The conclusion in the project report was that when it comes to controlling the robot in real-time, the main bottleneck was the robot's internal IK-solver. This is why the main goal of this master's thesis was to test the joint control methods from the NAOqi SDK. Control using the DCM directly will therefore not be considered in this thesis.

Instead, the focus was on methods from the NAOqi SDK. An arm movement can be described either as angular curves of the joints, or as a spatial trajectory (Morasso, 1981). As explained in the introduction, the focus for this thesis was to test angular control of the robot, because the Cartesian methods were much too slow. However, because the control was to be done using only a hand-held STEM-tracker, the system would have no information about how the rest of the human arm holding the tracker was oriented in space. This was the main thing separating this project from the two other telemanipulation projects I had looked into, as mentioned in section 1.2.

If my system had some way of tracking the whole arm for the person controlling the robot, it should be quite straight-forward to find the necessary angles between each joint, and send these directly to NAO. But this system was supposed to follow the trajectory of a hand-held STEM-tracker. This meant that the system had to rely on Kofinas' analytical IK-solver to find the joint angles necessary to reach the positions and orientations registered by the STEM-tracker, as explained in section 3.3. These angles could then be sent to joint methods from the NAOqi SDK, which is explained further in section 3.1.2.

2.2.2 Movement frame

This subsection is largely based on section 2.2.1 of my project thesis (Evjemo, 2016).

As will be explained in section 3.1, there are different ways of controlling an effector. But all of the control methods have to relate to a chosen *movement frame*. The NAO robot has initially three different frames it can move relative to. All coordinates are given relative to the origin, but the position of the origin differs for each coordinate system.

- **Torso:** The origin is placed in the centre of NAO's torso, like shown in the left part of figure 2.5. This is the most stable frame, as this point is never inaccurate relative to the robot's actuators. This movement frame will move along with the robot's torso, both when he moves around, and when he leans or sits down.

- **Robot:** The origin is placed along the z-axis, on the ground between NAO's feet, as shown in the right part of figure 2.5. This frame, like the torso frame, keeps the x-axis pointing straight forward, and moves with the robot.

- **World:** The origin is placed along the z-axis on the ground between NAO's feet, like for the robot frame, as shown in figure 2.6. However, for the world frame, the origin is left behind when NAO walks, both its position and orientation.

According to the documentation from Aldebaran, the *World* frame is most useful for calculations which require an external, absolute frame of reference (Cartesian Control 2-1, 2015). One of the long-term goals of this project is to place the robot and the trackers in a larger, global coordinate system, which will be discussed in section 5.2.4. Therefore, this frame had originally seemed like the obvious choice.

However, while working on the project thesis, the *World* frame was found to be quite inaccurate, because NAO's internal gyros did not handle slipping and inaccurate movements very well, which is discussed further in my project report (Evjemo, 2016). Since the Sixense STEM system was available, it seemed that the most accurate results could be acquired using the *Torso* frame:

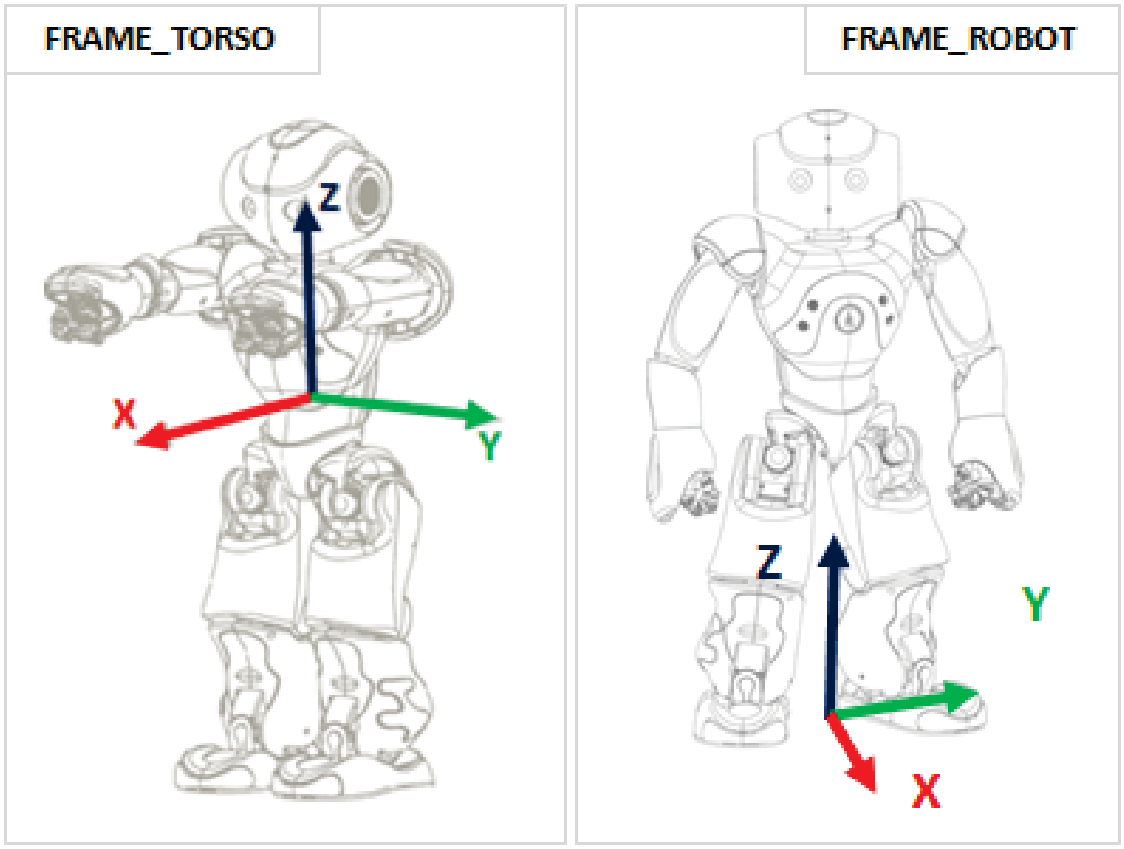

Figure 2.5: **Choosing the frame:** This is Nao's *torso* and *robot* frame. The torso fram is what we will be using in this project. Picture from (Cartesian Control 2-1, 2015).

By transforming the coordinates from the STEM system's coordinates over to local *Torso* coordinates before sending them to NAO, the movement control could be very accurate. This meant quite a few transformations of coordinate systems, as shown in section 3.4.4. This approach was used successfully both in the summer project and in the work on my project thesis, and the *Torso*-frame was therefore kept for the work on my master's thesis as well.

2.2.3 Initial position

This subsection is largely based on section 2.2.4 *in my project thesis (Evjemo, 2016).*

When the robot starts up, he stands up and goes to *Autonomous life-mode* (Understanding Autonomous Life, 2015). This is a mode where several of the robots actuators are in constant movement, making the robot sway slightly from side to side. Some of the sensors are also at work, so

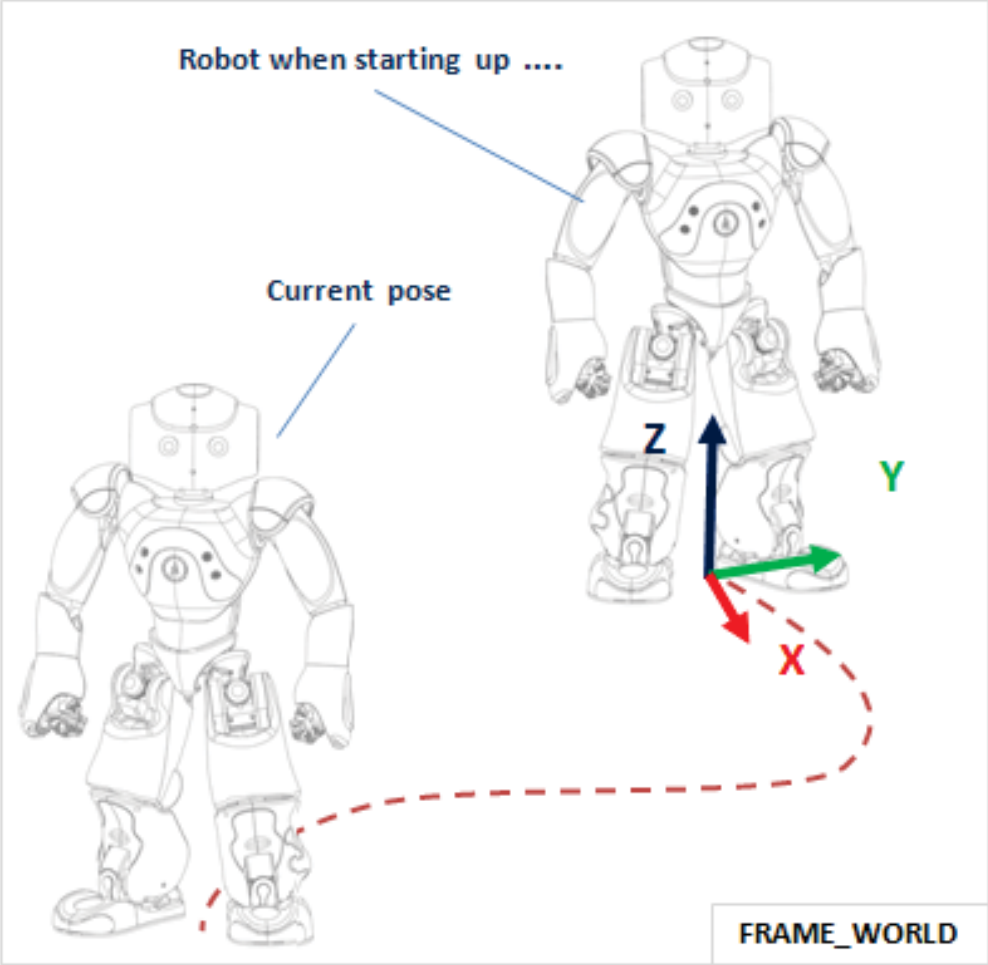

Figure 2.6: **Choosing the frame:** This is Nao's *world* and *robot* frame. Theoretically, this should be able to track NAO's complete movement from startup. Picture taken from (Cartesian Control 2-1, 2015).

that NAO can turn his head towards sounds etc. To be able to move NAO's arm effectors as accurately as possible, it was necessary to keep the robot in a position and state where he kept completely still.

When NAO is in his *initial position*, he has both hands slightly in front of him and out to the sides, like seen in figure 2.7. The robot's body is kept completely still, but it is possible to move the effectors by using functions from the NAOqi SDK. This start-position was chosen because it was a quite steady position when considering the robot's balance, at the same time as it allowed the robot to move his hands freely in all directions.

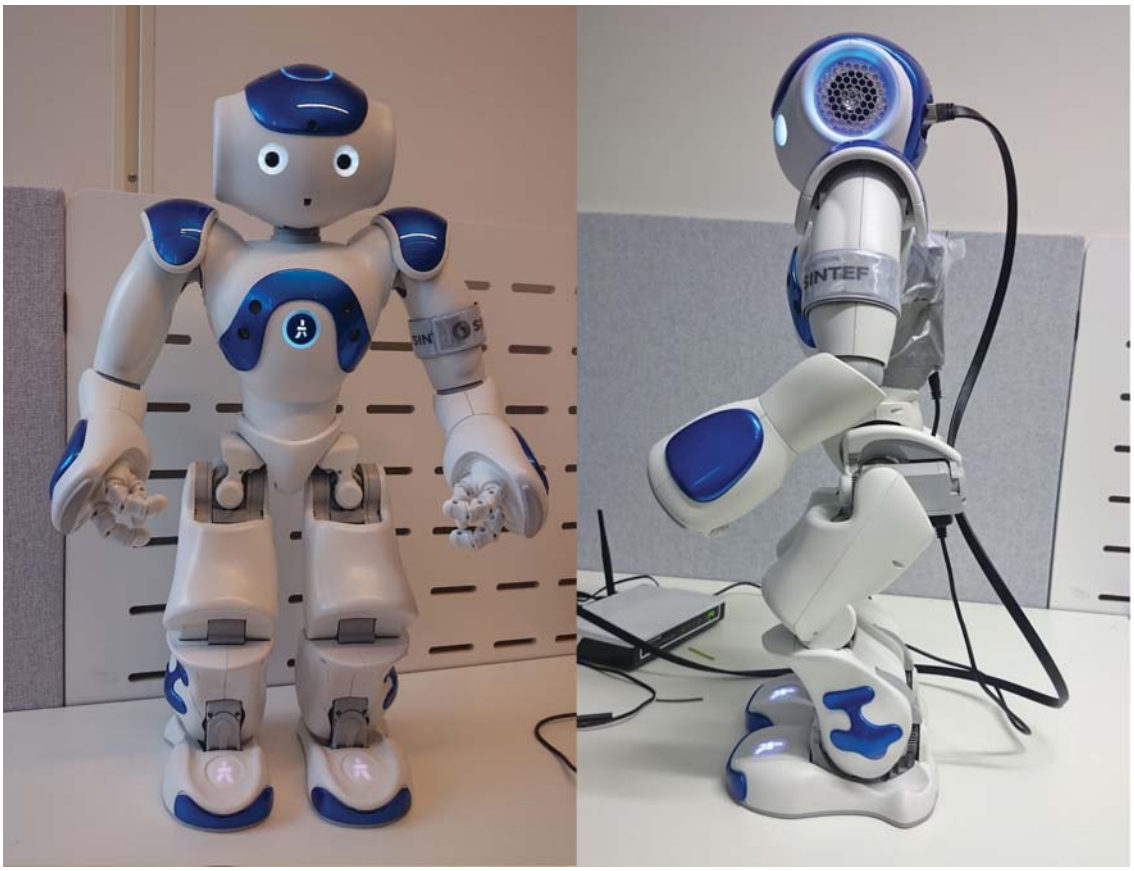

Figure 2.7: **StandInit:** This is the pre-set position *StandInit*, which is a good start position for controlling NAO's arms.

When working with the actual robot, and not just a model, it was important to keep in mind that NAO's engines might overheat due to the strain of keeping his joints locked in the same position over time. When the robot was kept in *autonomous life*-mode, the engines would not get warm as quickly, which made it evident that the slight wagging helped keeping the engines in NAO's effectors from overheating. It is also worth noting that in NAO's initial position, the robot legs are locked in a position where the knees are not hyper-extended. This is more challenging for the engines than if the legs were completely straight, because they must support the robot's weight against gravity (Kuo, 2007).

The plan was therefore to make NAO sit down and rest between all test runs, so that the engines could cool down. When controlling the robot's actuators, it was also important to make sure that the robot did not tip out of balance, as the system does not have any safety mechanism to

override the control-input if that should happen (Cartesian Control, 2015).

Data from the robot's *initial position* was used when testing the IK-solver, as described in section 3.3. The data for the position and orientation of NAO hands when the robot was in *initial position* were collected through functions from ALMotion, which are explained in more detail in section 3.1.2. This made it possible to test the IK-solver for a position and orientation which had a known, real solution. The initial position and orientation were also used to combine the STEM-tracker movements with corresponding movements on the robot's arm effector, as explained in section 2.5.3 and section 3.4.4.

2.2.4 A closer look at the arm effector

NAO's arm effectors has 5 DOFs, distributed as shown in figure 2.8: 2 DOFs in the ball-in-socket joint in the shoulder, 2 in the ball-in-socket joint in the elbow, and 1 in the rotating wrist. How the five DOFs are distributed is also shown in figure 3.16, which also illustrates better which axis each joint rotates about. The fact that the robot arm only has 5 DOFs means that the robot's arms are simplified compared to a human arm, which has 7 DOFs (Cruse, 1986).

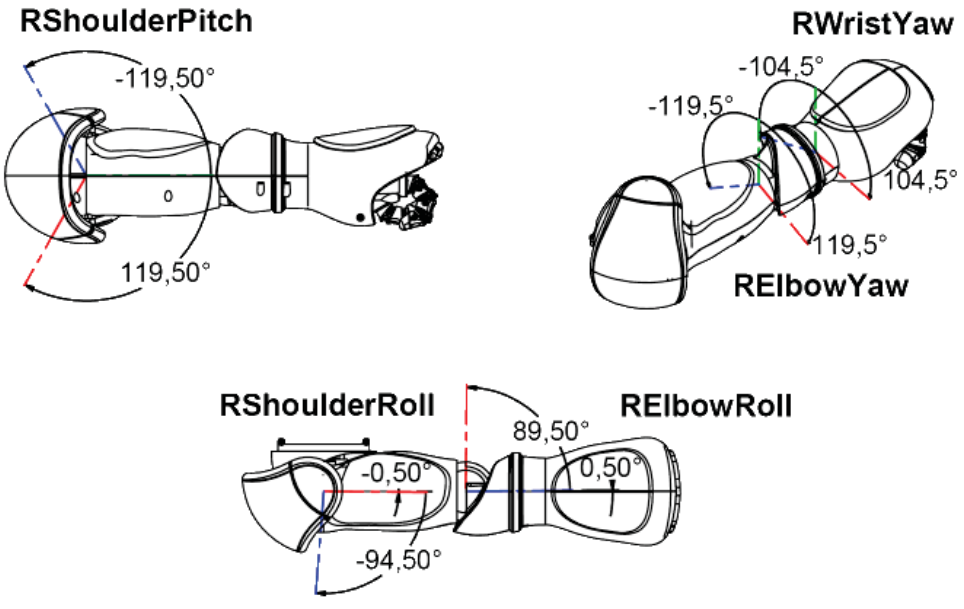

Figure 2.8: **Arm joints:** Here we can see how the five DOFs in NAO's right arm is distributed. Picture from (NAO Technical Guide: H25 - Joints - V3.2, 2012).

The most noticeable difference between a human arm and NAO's arm, is that the wrist only has 1 DOF. This means that the orientation of the hand effector is always locked to the orientation of the underarm, except for the ability to rotate about the axis going through the underarm. This means that the robot's arm does not have the same ability as the human arm to reach one given position and orientation in a seemingly endless number of angle orientations for the remaining angles in the arm.

Just consider this: If you place your hand flat on a table, you will still be able to move the rest of your arm around, because the wrist can move quite freely from the underarm. For the robot, keeping the hand in this position will mean that the entire arm is more restricted. However, there is still redundancy in the robot's arm effector, meaning that there are several ways to position each joint in order for the end effector to reach one desired position. This makes it very challenging to do the inverse kinematics, because it is necessary to create constraints that enables the robot to choose one way of moving to a given point over another (Cruse, 1986).

When using the methods from the NAOqi SDK, which will be presented in section 3.1, it is necessary to be aware of where the sensors that make up the end effector of NAO's arm chains are located. As shown in figure 2.9, the sensors are located in the center of the robot's hands. The goal is to make this point in the center of the hand follow the movements of the STEM-tracker, both in position and orientation.

A manipulator's *workspace* is the total volume in which the end effector can move when considering all possible movements. The workspace is constrained both by the mechanical constraints of the joints, and by the geometry of the manipulator (Spong, Hutchinson, and Vidyasagar, 2006, p. 6). The experience from working with the robot in the summer project and the project thesis, was that the mobility of the arm effector gave NAO a quite large workspace around its own torso. When controlling the arm effector with the STEM-controller, it would still be important to keep the movements small enough to stay within the robot's workspace.

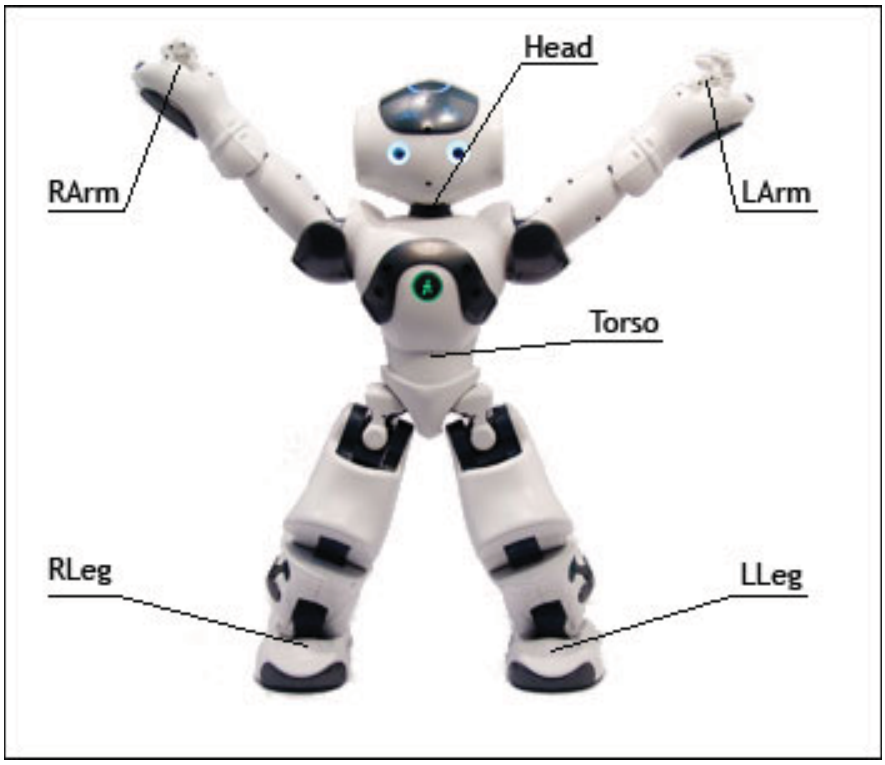

Figure 2.9: **Control points:** The sensors that make up the control point for NAO's effectors. As shown, the sensor in NAO's arm is located in the centre of the hand (Cartesian Control, 2012).

2.3 Simulating the robot

Based on my previous work with the NAO robot, I had learnt the hard way that the robot might behave quite surprisingly if the control input was not thoroughly tested. For example, when we were working on the summer project, we once experienced that the robot hyperextended its ankle joints and "jumped" backwards when we were really only trying to make the robot close its hand... This did of course happen because the control input was not tested before it was run on the physical robot. Luckily, the robot was not damaged in that incident, but it became clear that it would be a good idea to create some kind of simulated 3D-model to test the system on before trying angle control on the robot itself.

In addition, testing the program on a simulated model of the robot would make it easier to make sure that the system worked properly *before* introducing another possible error source in the form of more hardware. Working with hardware will always mean more uncertainty than working with theoretical models, and the STEM-system was already creating enough problems,

which will be explained further in section 3.8.

2.3.1 LabVIEW model

When trying to decide on a simulation tool for the 3D-model of the robot, programs like `V-rep` and `Autocad` were considered as alternatives. I had no previous experience with 3D-modelling, so I consulted with my advisers at SINTEF to find the most suitable program. My system was using the frame *Torso* for the robot, which mapped all movements relative to the robot's own torso, which will be further explained in section 2.2.2. It was therefore only necessary to create a simple model of the robot's torso with the same DOFs as NAO. Based on the advice from my supervisors at SINTEF, I ended up using the functionality for 3D-modelling included in LabVIEW. Even though I was not familiar with this part of the program, it seemed easier to learn a new part of a program I had some experience with than to start from scratch.

It is also worth mentioning that some of the most time consuming problems I experienced when working with my project thesis (Evjemo, 2016), was trying to make different programs and hardware communicate with each other. Because the plan for the master's project was to implement most of the system functionality in LabVIEW, it seemed safest to also include the 3D-model in LabVIEW in order to avoid an additional problem related to software communication. Even though LabVIEW does not necessarily have as much of the functionalities regarding the visual design of the 3D-model that can be found in more advanced 3D-modelling programs, the conclusion was that this did not matter. The important thing for my project was to have a way to test the code and my program other than the physical NAO robot, and a simplified stickman would suffice.

The 3D-model created is shown in figure 2.10. It is a simple model of the robot's torso and arms, which has the same DOFs in the arm effectors as the H25-model of the NAO robot. When implementing the model, the different angles were set with controls in the actual range for each joint, based on the specifications from the scripts and original thesis (Kofinas, 2012), and the documentation(H25 - Links, 2015). In figure 3.9, it is shown more clearly how the 3D-model ends up in the same position as the physical robot when the angles are set to be identical.

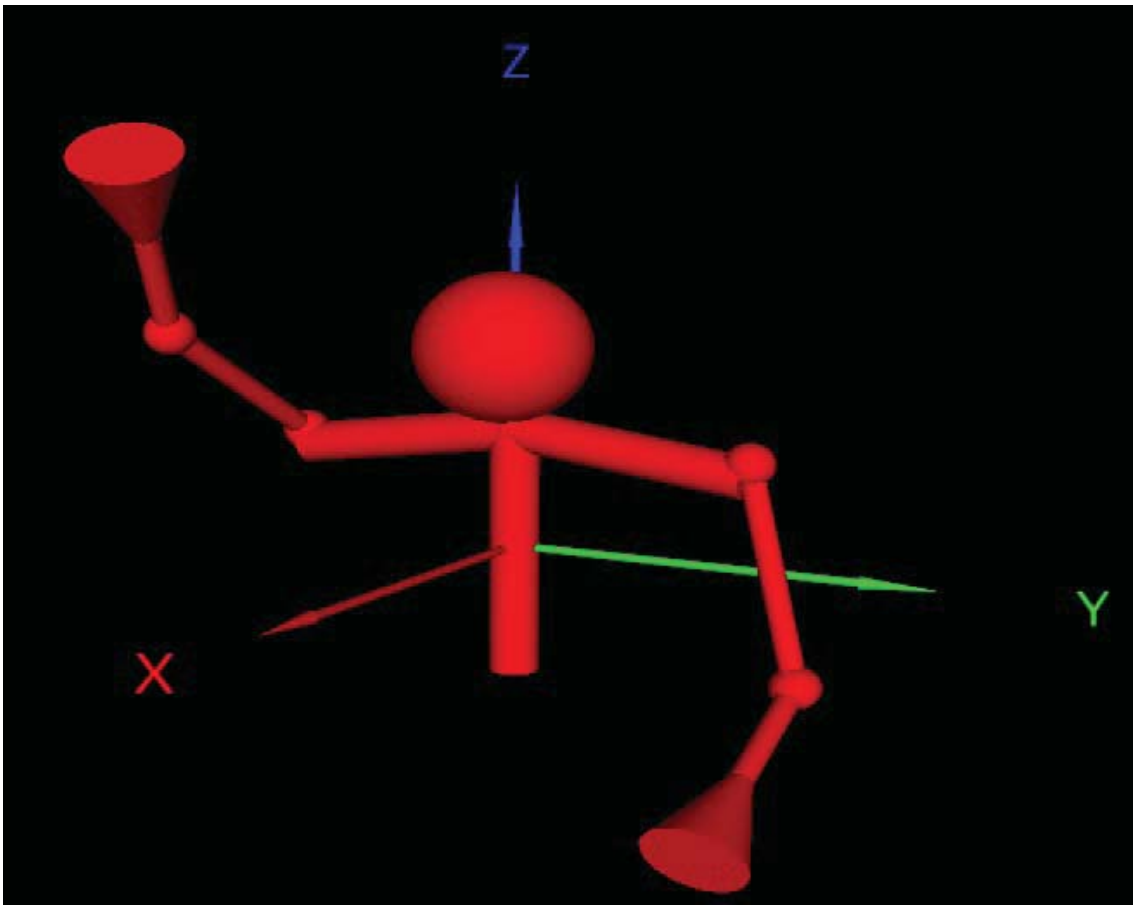

Figure 2.10: **Robot model:** This is the 3D-model created in LabVIEW to perform tests of the system. The model is similar to NAO's torso, and the arms have the same DOFs, so that the different joints moves just like NAO's joints.

The 3D-model had originally no latency implemented, because it was necessary to check how fast the LabVIEW-program was able to interpret the angle output written to file, and actuate the movement on the model. When the system worked as it should, the plan was to implement latency to be able to test how much latency one could have before the control stopped feeling natural. I will get back to this in section 3.9.

When creating the 3D-model in LabVIEW, it was important to be aware of which joints depended on each other, and in what order. For instance, if the elbow yaw had been independent of the elbow roll, the elbow yaw and wrist yaw would always be rotations about the same axes. However, this was not the case, and the 3D-model had to take this into account. The fact that the elbow

yaw was not independent from the elbow roll was also the reason why this system could not use the IK-solver developed by 5^{th} year student Åsmund Pederson Hugo, see section 3.6.

When creating the model, the angles were set via manual controls. When these were thoroughly tested, and it was certain that the 3D-model's mobility matched the NAO robot's, they were replaced with input from the IK-solver. When the IK-solver returned valid results, the five angles representing the arm effector in question were written to file. The LabVIEW-program with the 3D-model would then read these angles from file, and send them directly to the model's joints. The first test-run for this program was to use the collected data for the orientation and position of the arm effectors during NAO's `initial pose` for the robot. This data was run through the IK-solver, and the corresponding angle solution written to file. The resulting pose is shown in figure 3.9, and it is clear that this corresponds with the actual robot's pose, shown in the same figure.

2.3.2 Virtual NAO robot in Choreographe

In addition to the simulated robot created in LabVIEW, I realized that the program Choreographe that came with the robot also had its own, virtual robot model. Even though Choreographe was not used in the system in general, this model proved useful for testing the Python-scripts that were to control the physical robot. The simulated robot created in LabVIEW would be helpful for testing the validity of the angle input from the IK-solver. However, it could not be used for testing the methods from the NAOqi SDK, which were used to actually control the robot. These methods will be explained further in section 3.1.2.

The virtual robot in Choreographe is shown in figure 2.11, and looks exactly like the physical robot. It has the same abilities as the real robot, and can be controlled by the same methods, either through Choreographe's control methods, or though scripts in Python or C++. This is explained further in section 2.2. However, it is worth noting that the simulated robot is not subjected to gravity (Simulated Robots, 2015). This means that the virtual robot cannot fall down because it loses balance, unlike the real robot. Because the model was only meant for testing of small arm movements, this should not be a problem.

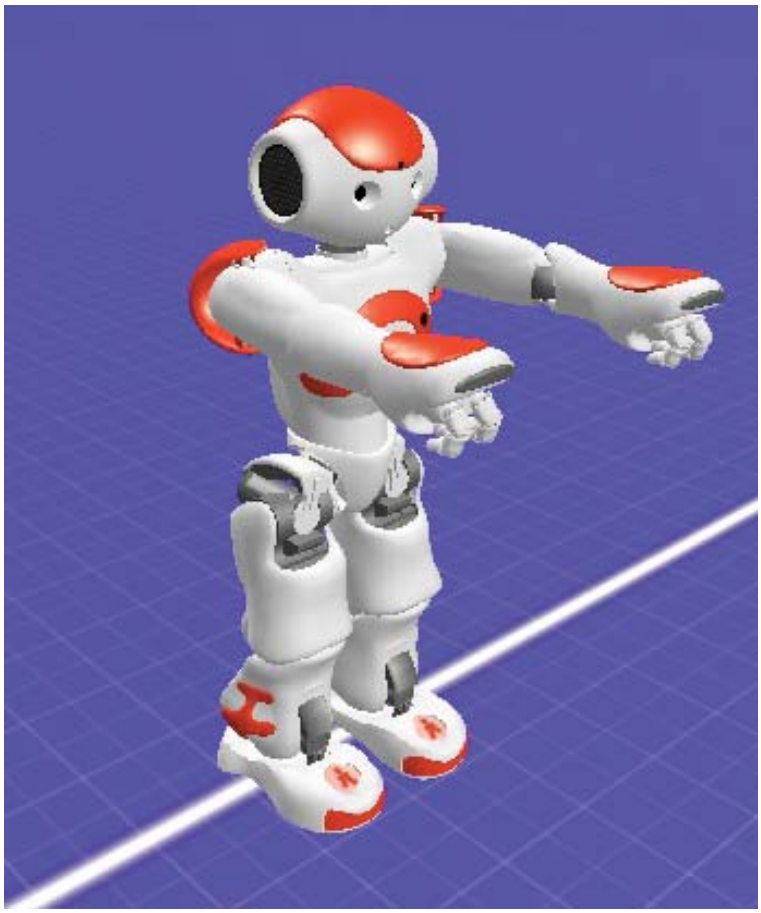

Figure 2.11: **Virtual NAO:** This is the virtual robot in Choreographe. All methods from the NAOqi SDK (see section 2.2) could be tested on this model, and it would behave almost exactly like the real robot.

2.4 Communicating with STEM

One of the challenges while working on the previous projects, was to get hold of the tracker data from the STEM-system. As explained in section 1.4.1, the 5-controller STEM-system used in this project is only a BETA-version, and the access to documentation and coding examples quite limited. This meant that simple things like extracting the tracker data became a challenge, and something that had to be solved in different ways depending on the complete system structure. Because the system-design would be different for the master's project than for the previous projects, as explained in section 2.1.2, it was once again necessary to rethink the approach for collecting tracker data. There were several different approaches that had to be considered.

2.4.1 Old solutions using DLL-files

During the summer project, the system structure made it necessary to send tracking data from the STEM-trackers to LabVIEW. This was because all the robot control was done via LabVIEW, and the tracking data was sent to the Python-scripts via TCP-connections, see section 2.1.2. The communication problem was then solved by using a DLL-file created in C++ combined with a DLL-block in LabVIEW.

When the system structure was changed during the work on the project thesis, it was necessary to send the tracking data from STEM to Python. This was because LabVIEW was not used in the project thesis, and all of the control was done directly through Python-scripts. The problem was then solved using the foreign functions library *ctypes* in Python (Stack overflow, 2008), which allowed the Python-scripts to read DLL-files.

2.4.2 New potential solution using text-files

In the late stages of the project thesis, Sixense Entertainment released a firmware update for STEM which also included some coding examples. This made it easier to create an effective way of reading data from the trackers, and send them to other programs or hardware. One of my advisers at SINTEF Fisheries and Aquaculture created a simple program in LabVIEW which communicated with STEM using text-files.

The basic functionality is that the position and orientation of each tracker is registered, and written to file. In addition, some information about the two controllers' joysticks and buttons are registered, and written to the same file. The positions are written as x-, y-, and z-coordinates, given in the STEM-systems own coordinate system, see section 3.4.3. The functionality of the program is explained further in section 3.2.1.

2.4.3 Specifications for the new system

For this master's project, it was necessary to extract the tracking data from the STEM-controllers, and somehow send the data describing position and orientation to the IK-solver. The IK-solver

should then return a solution containing the corresponding angles to realize the movement, as will be explained further in section 3.3. It was therefore necessary to re-evaluate if any of the old solutions to the communication problem would work for the new system structure. This will be discussed further in section 3.4.3.

2.5 Theory

In this theory section the principles behind forward and inverse kinematics will be explained, as these are the foundations for robot control. I will also present different ways of representing rotations and Cartesian movements mathematically, which all have different advantages that had to be considered when designing the system. How any given position and rotation can be transformed from one frame to another will also be explained. Lastly I will present the principles of polar decomposition, which has to be used in order to create valid, normalized input for the IK-solver.

2.5.1 Forward and inverse kinematics

A robot actuator, like the arm of the NAO robot, is composed of a set of links. These links are connected through joints, and the number of directions you can move these joints determines the actuator's total number of DOFs. A joint's movement can either be revolute, which allows relative rotation between two links, or prismatic, which allows a linear relative motion between two links (Spong, Hutchinson, and Vidyasagar, 2006, p. 4). Because all of NAO's joints are revolute, this thesis will not go into more detail regarding prismatic joints.

The forward kinematics problem is to determine the position and orientation of the end effector of an actuator when you know the joint angle values of the individual joint variables (Spong, Hutchinson, and Vidyasagar, 2006, p. 76). The joint variables are in this case the angles between the links. Forward kinematics can be used to determine the position and orientation for each and every link and joint in an effector, but in this thesis the focus will be only on the end effector of NAO's arm. A forward kinematics problem will always have *one* solution.

The inverse kinematics problem is the opposite of the forward kinematics problem, and quite a lot more challenging: It determines the joint variable values necessary to make the end effector end up in a desired position and orientation in Cartesian space (Spong, Hutchinson, and Vidyasagar, 2006, p. 93). In addition, any solution to the inverse kinematics problem depends on the engineering of the specific actuator, not only the mathematics. For example, a mathematical solution might not be realizable if the given joint angles cannot rotate a full 360°. Therefore, an IK-solver must take this into account, and check if each mathematical solution is compatible with the physical limitations of the actuator.

The equations necessary to solve the inverse kinematics problem are usually complicated nonlinear functions of the effector's joint variables. A solution can be found through an analytic, closed-form approach, which means that you find an explicit relationship between the joint variables and the position and orientation of the end effector. Alternatively, it is possible to use a numeric, iterative method to find a solution. The analytic, closed-form solution is preferable because it is much quicker, and because it allows you to implement rules for choosing among several solutions, should that be the case (Spong, Hutchinson, and Vidyasagar, 2006, p. 95). As explained earlier, the NAO robot uses a numeric IK-solver, which seems to be very slow (Evjemo, 2016). The hope is that an external, analytic IK-solver will make the system more effective.

Because the forward and inverse kinematics solver used in this thesis has been developed by others (Kofinas, Orfanoudakis, Lagoudakis, 2013), I will not go into further detail regarding how the forward kinematics and inverse kinematics calculations are done. It is still important to keep in mind that when doing the forward kinematics, a given set of values for the different joints will always result in one and only one position and orientation of the end effector. For inverse kinematics, however, a given position and orientation might have zero, one, or several solutions for the combination of joint values in the effector. How probable it is that there can be several solutions, depends on the number of DOFs and the geometry of the effector. For more information about forward and inverse kinematics, see chapter 3 in *Robot Modelling and Control* (Spong, Hutchinson, and Vidyasagar, 2006).

2.5.2 Describing rotation and Cartesian position

Rotations are registered relative to axes in a coordinate system with a known orientation, and this coordinate system is called the *frame*. A given rotation is described differently depending on the frame where the rotation happens, which will be explained further in section 2.5.4. A rotation can be described by *Euler angles*, *quaternions*, or a *rotation matrix*.

Euler angles

A very common way of representing a rotation with respect to a given frame, are *Euler angles*. This is also perhaps the most intuitive method, because it is quite easy to visualize the rotation. A rotation relative to a given coordinate system, or *frame*, is described by three successive rotations about the three axes (Spong, Hutchinson, and Vidyasagar, 2006, p. 53-54). The most common is that the three Euler angles ϕ, θ, and ψ describe a rotation about the z-axis, y-axis, and then the z-axis again, respectively. Euler angles can also describe rotations about different combinations about the three axes. If the rotation happens about all three axes, they are often called e_x, e_y, and e_z. As long as one knows how the Euler angles are defined for the system in question, these approaches are equally correct.

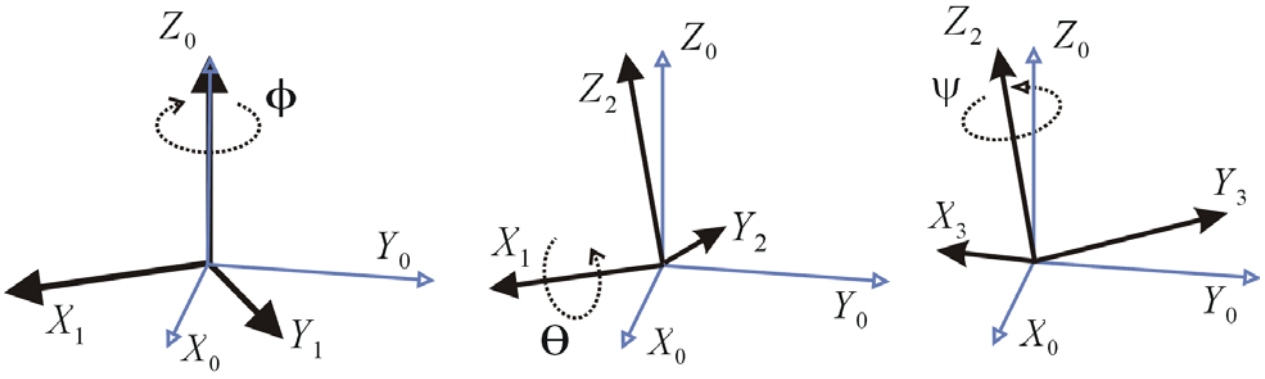

Figure 2.12: **Current frame:** This figure shows the Euler angle rotations happening about the current frame. When rotating about the current frame, only two of the three axes will change orientation. *Picture from Euler Angles, 2016.*

Rotations might happen about the current frame, or about a global, fixed frame (Spong, Hutchinson, and Vidyasagar, 2006, p. 53-54). Figure 2.12 shows rotations about the *current* frame. The

blue frame represents the initial, fixed frame, but the rotations are happening about the axes of the black coordinate system. With this kind of rotation, the axis about which the rotation happens will not change orientation.

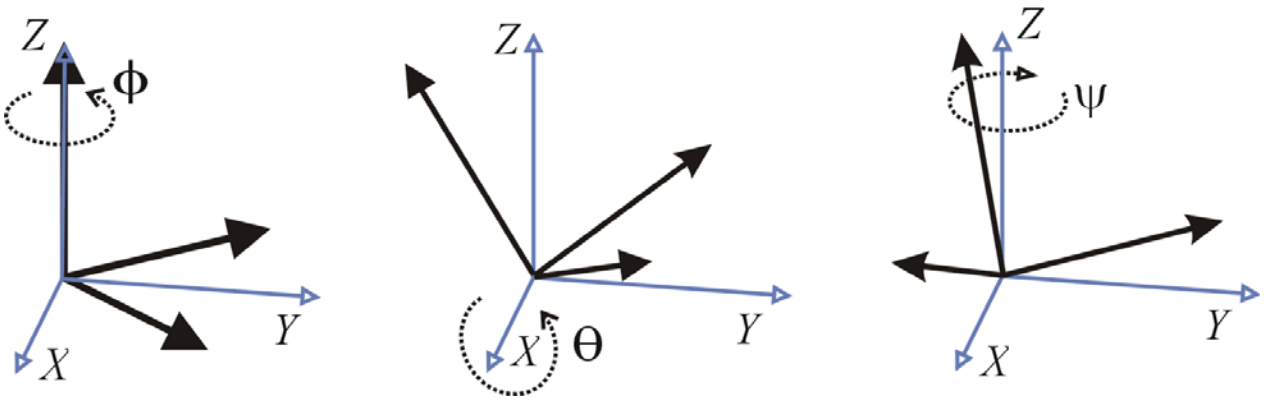

Figure 2.13: **Fixed frame:** This figure shows the Euler angle rotations happening about the same, fixed coordinate system shown in blue. This means that all of the axes might change orientation in one, single rotation. *Picture from Euler Angles, 2016.*

Another way of doing a rotation is to rotate an object or frame about a fixed frame, like shown in figure 2.13. The black coordinate system rotates in its current orientation about the fixed, blue frame. As the figure illustrated, this means that all three axes might change orientation at once.

Rotation matrix

Each rotation can be represented by a rotation matrix, which is a 3×3 matrix describing a rotation around the x-, y- and z-axis in a given frame. According to Spong, Hutchinson, and Vidyasagar, a rotation matrix R_n^{n-1} is a matrix whose column vectors are the coordinates of the unit vectors along the axes of the frame n expressed relative to frame n (Spong, Hutchinson and Vidyasagar, 2006, p. 39). Because unit axes are mutually orthogonal, the transpose of a rotation matrix R is equal to its inverse, and when working with right-hand coordinate frames, the determinant is also equal to +1 (Spong, Hutchinson and Vidyasagar, 2006, p. 40). These kinds of $n \times n$ matrices are called the *Spherical Orthogonal* group, and is denoted by the symbol $SO(n)$ (Spong, Hutchinson and Vidyasagar, 2006, p. 40). In general, a rotation matrix $R \in SO(n)$ has the following properties (Spong, Hutchinson and Vidyasagar, 2006, p. 41):

- $R^T = R^{-1} \in SO(n)$
- The columns are mutually orthogonal. This means that the rows are also mutually orthogonal.
- Each column is a unit vector. This means that each row is also a unit vector. In other words: For each row or column with elements $r_1, r_2, and r_3, \sqrt{r_1^2 + r_2^2 + r_3^2} = 1$.
- The determinant of R is equal to 1.

Because the column vectors of a rotation matrix are of unit length and mutually orthogonal (their inner product is zero), the matrix is also orthogonal. This means that the determinant of a rotation matrix must always be 1 (Spong, Hutchinson and Vidyasagar, 2006, p. 40). Having a determinant equal to 1, leads to the transpose and the inverse of the matrix being identical. The product of an orthogonal matrix and its transpose will always equal the identity matrix: $R^T \cdot R = R \cdot R^T = I$. A rotation matrix representing a movement can be written as follows:

$$\mathrm{R} = \begin{pmatrix} r_{00} & r_{01} & r_{02} \\ r_{10} & r_{11} & r_{12} \\ r_{20} & r_{21} & r_{22} \end{pmatrix}$$

A rotation matrix can be used to describe a rotation to both a fixed and a current frame, as will be explained further in section 2.5.3. The rotation matrices describing a single rotation θ about the x-, y-, or z- axis, are defined as:

$$R_{x,\theta} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos(\theta) & -\sin(\theta) \\ 0 & \sin(\theta) & \cos(\theta) \end{pmatrix}, R_{y,\theta} \begin{pmatrix} \cos(\theta) & 0 & \sin(\theta) \\ 0 & 1 & 0 \\ -\sin(\theta) & 0 & \cos(\theta) \end{pmatrix}, R_{z,\theta} \begin{pmatrix} \cos(\theta) & -\sin(\theta) & 0 \\ \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

When combining these rotation matrices, one can describe more complex rotations of frames. It is important to note that the order in which these, or any other, matrices are multiplied, is

not arbitrary. One rotation matrix can represent several combinations of Euler angles, because several different combinations of rotation can lead to the same, final orientation. This will not be explained further in this thesis, but is important to remember when converting between different ways of representing rotation. For more details, see my project thesis section 2.4 (Evjemo, 2016) or the article *Decomposing and Composing a 3x3 Rotation Matrix* (Ho, 2011).

Quaternions

In addition to Euler angles or a rotation matrix, a given rotation in a 3-dimensional space can also be represented by *quaternions*. This representation was first introduced by the the Irish mathematician William Rowan Hamilton in 1843 (Egeland and Gravdahl, 2002). The quaternions were written:

$$\mathbf{q} = q_\omega + i\,q_x + j\,q_y + k\,q_z \tag{2.1}$$

Where i, j, and k are all imaginary units satisfying (Egeland and Gravdahl, 2002, 232):

$$i^2 = j^2 = k^2 = i\,j\,k = -1 \tag{2.2}$$

$$i\,j = -j\,i = k, j\,k = -k\,j = i, k\,i = -i\,k = j \tag{2.3}$$

Unit-length is a characteristic of the quaternions, but when using numerical methods, it is sometimes necessary to to small corrections to make sure that the quaternions stay normalized, as will be explained further in section 3.5.6. Checking if the quaternions registered from the STEM tracker were already normalized could be done by making sure that the relation presented in equation 2.4 was true (Using Quaternion to Perform 3D rotations, 2011):

$$\sqrt{q_x^2 + q_y^2 + q_z^2 + q_w^2} = 1 \tag{2.4}$$

To have normalized quaternions is important when finding the rotation matrix based on the quaternions, which will prove useful in this project. If the quaternions are normalized, the rotation matrix can be represented as shown in figure 2.14. If they are not normalized, this can be done by dividing q_w, q_x, q_y, and q_z by $\sqrt{q_x^2 + q_y^2 + q_z^2 + q_w^2}$ (Using Quaternion to Perform 3D rotations, 2011).

It is worth noting that the four quaternions are sometimes given in the order q_ω, q_x, q_y, q_z, other times in the order q_x, q_y, q_z, q_ω. it is therefore important to check in which order the quaternions are given for a given system, as will be discussed further in section 3.4.2.

$1 - 2*qy^2 - 2*qz^2$	$2*qx*qy - 2*qz*qw$	$2*qx*qz + 2*qy*qw$
$2*qx*qy + 2*qz*qw$	$1 - 2*qx^2 - 2*qz^2$	$2*qy*qz - 2*qx*qw$
$2*qx*qz - 2*qy*qw$	$2*qy*qz + 2*qx*qw$	$1 - 2*qx^2 - 2*qy^2$

Figure 2.14: **Creating the rotation matrix:** When knowing the normalized quaternions, the ration matrix can be created using this corrolation. Picture taken from (Baker, 2016).

Transformation matrix

To describe both rotation and Cartesian position, one can use a transformation matrix. This matrix consists of the rotation matrix and Cartesian coordinates that describe the transformation from one frame to another (Spong, Hutchinson, and Vidyasagar, 2006, p. 73-74). For example will a rotation matrix describing the rotation R_n^0 and the Cartesian translation o_n^0 from the initial frame 0 to frame n, be described as:

$$T = \begin{pmatrix} \mathrm{R}_n^0 & \mathrm{o}_n^0 \\ 0 & 1 \end{pmatrix} = \begin{pmatrix} r_{00} & r_{01} & r_{02} & x \\ r_{10} & r_{11} & r_{12} & y \\ r_{20} & r_{21} & r_{22} & z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Like for the rotation matrix, it is possible to find partial transformations, for example the transformation T_1^0 from frame 0 to 1, and the transformation T_n^1 from frame 1 to n. When calculating partial transformations, it is necessary to extract the rotation matrix and translation separately. How to calculate the partial movements is explained in section 2.5.3.

2.5.3 Calculating partial movements

In order to make the arm effectors follow the trajectory and rotation of the STEM system, it was necessary to look at the position and rotation of both the tracker and the end effector (NAO's hand) at the moment we started the control program. From there, it was necessary to find the actual movements of the tracker from the point when our program started running. In other words: It was necessary to find the change in position and orientation of the tracker from the beginning of the current tracking session and up until the present time.

It was also necessary to consider the initial position and orientation of the robot arm. By combining this with the tracker data from STEM, it would be possible to find the new, desired position and orientation of the arm effector. Because the robot control was based on joint control methods in this project, this data would be sent to the IK-solver, as will be further explained in section 3.1.1. The IK-solver should then return the angles necessary to realize this movement, which finally would be sent to the robot, allowing it to move its arm accordingly.

Cartesian position

First, it was necessary to find the actual movement of the STEM-tracker from the beginning of the current session and up until the moment of the calculation. For the Cartesian position coordinates, this was quite straight forward. By subtracting the initial value from the new value, we get the actual movement:

$$pos_{move} = pos_{current} - pos_{init} \tag{2.5}$$

The process is a bit more complex for rotations because of how the frames are defined in our system.

Rotation

To find the change in rotation, it is possible to use rotation matrices. Basic rotation matrix calculations lead to this mathematical relationship (Spong, Hutchinson and Vidyasagar, 2006, p. 49):

$$R_2^0 = R_1^0 \cdot R_2^1 \tag{2.6}$$

Equation 2.6 tells us that the total rotation of an object, R_2^0, is equal to the product of the initial rotation R_1^0 and the actual change in rotation R_2^1. In our case, the R_1^0-matrix represents the rotation matrix we get the moment we initialize the tracked object. The current rotation is represented by the R_2^0-matrix. The change in rotation from the moment we began the tracking session is therefore represented by the R_2^1-matrix. This can be found by doing some basic matrix multiplications:

$$(R_1^0)^T \cdot R_2^0 = (R_1^0)^T \cdot R_1^0 \cdot R_2^1 \tag{2.7}$$

Because $(R_n)^T \cdot R_n = R_n \cdot (R_n)^T = I$, we end up with the following relationship between the current rotation matrix, the initial rotation matrix, and the rotation matrix that represents the movement we have performed:

$$R_2^1 = (R_1^0)^T \cdot R_2^0 \tag{2.8}$$

2.5.4 Frame transformations

Later in this report it will be explained that some of the rotation matrices have to be transformed because data is collected in two different coordinate systems, or *frames*. A rotation can be described in different frames by multiplying the initial rotation matrix with rotations about the three axes necessary to get to the new, desired frame (Spong, Hutchinson and Vidyasagar, 2006,

p. 50-51). An example are the two frames shown in figure 2.15: The coordinate system on the right is really only the coordinate system on the left that has been rotated +90° about its x-axis.

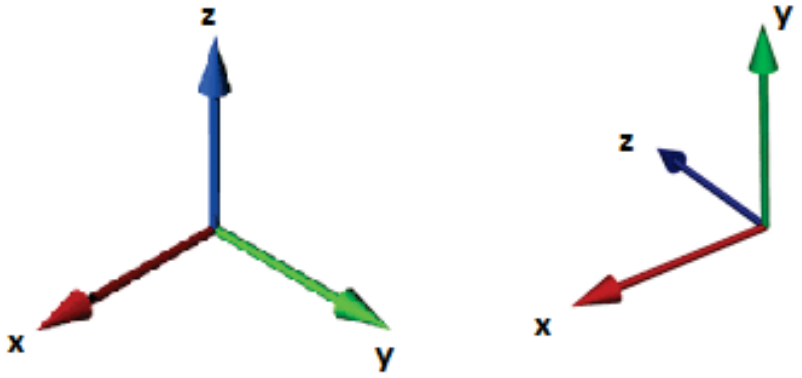

Figure 2.15: **Axis rotations:** If the coordinate system to the left is rotated 90° about the x-axis, the result will be the coordinate system to the right. In the same way, a rotation matrix describing a rotation in the right coordinate system, can be described in the left coordinate system by multiplying it with the corresponding rotation matrix R_x.

In the same way, a rotation described in the frame to the left can be described in the frame to the right if we "rotate it" +90° about its x-axis. To rotate the with the angle θ in 3D about the three axes, we use the three rotation matrices $R_{x,\theta}$, $R_{y,\theta}$, and $R_{z,\theta}$, shown in section 2.5.2.

The rotation can either be performed relative to the *current* frame, or with respect to the *world* frame, which is a global coordinate system defined for the given system. If the rotation is performed with respect to the current frame, the respective rotation matrices are *post-multiplied* to the initial rotation matrix. Otherwise, the rotation matrices are *pre-multiplied* to the initial rotation matrix (Spong, Hutchinson and Vidyasagar, 2006, p. 50-51).

The frames in the system in which the coordinate transformations will occur, have the same, general orientation. The only difference is that the axes are defined in different directions, but along the same lines, just like the two frames in figure 2.15. Therefore, changing the Cartesian position coordinates will be as easy as swapping them around and changing their sign. This will be explained further in section 3.4.3.

2.5.5 Polar decomposition

When working with real square matrices that have to be orthogonal, it is possible to use *polar decomposition*. What this methods says is that every real square matrix A can be factored into:

$$A = QS$$

where Q is orthogonal, and S is symmetric positive semi-definite (Strang, 1988, p. 445). There are several methods for factoring matrices into orthogonal matrices, but what makes polar decomposition very useful is that it guarantees that the orthogonal matrix Q is the *closest* orthogonal matrix to the matrix A (Keller, 1975).

The matrix S is given by

$$S = \sqrt{A^* A}$$

where A^* is the *conjugate transpose*. This thesis will only work with real matrices, and for real matrices the conjugate transpose A^* is equal to the transpose A^T (Strang, 1988, p.293). This means that the closes orthogonal matrix Q to a square matrix A, is any orthogonal matrix Q which is the result of the polar decomposition $A = Q\sqrt{A^T A}$ (Keller, 1975). How to find the square root(s) of a matrix will not be explained here, because this will not be used in the project work.

Chapter 3

Results

3.1 Joint control methods for NAO

In the project report, the focus was on Cartesian methods of control, and the conclusion was that these methods were too slow for achieving anything close to real-time control of the robot (Evjemo, 2016). As explained in section 2.2, control of the robot's arm effectors would therefore be done using *joint control methods* in this master's thesis.

3.1.1 Animation methods - why they were not used

This section is based on section 3.1.1 of my project thesis (Evjemo, 2016).

Like for Cartesian methods, the documentation for the robot states that there are two ways of controlling an effector (Joint Control API, 2015). One way is through *animation methods*, which are functions that are *time fixed* and *blocking*. *Time fixed* means that these methods move one or multiple joints to a target angle within specified *time trajectories*.

Like the Cartesian animation methods, the animation methods for joint control can be useful if you want the robot's arm to move steadily between a given series of orientations. However, if the animation methods were to be used, it would be necessary not only to register the change in movement of the STEM-trackers, but also the time it took the tracker to move from point to

point. This seemed like unnecessary work, as the goal was really only to make NAO's arm effectors follow the STEM-trackers as closely as possible.

Even though the time-step-condition is unnecessary, it would have been possible to implement. The thing that made it clear that animation methods should be avoided altogether, whether using Cartesian or joint control, was the second characteristic of these methods: The fact that they are *blocking*. Controlling NAO with blocking methods imply that the next instruction will only be executed after the end of the previous command (NAOqi Framework, 2012). In other words: It would make it impossible to control both hands at the same time. The robot would then have to take turns between executing commands for the two hands, which would likely lead to oscillating movements due to an uneven distribution of force.

An example of a blocking call is the *opening* and *closing* command for NAO's hands. This functionality was implemented in the original summer project in order for the robot to pick up an object. It was impossible for NAO to open the hand and move the arm at the same time. This meant that every time we wanted to make NAO close his hand, this required a full stop for about three seconds for all other movements. This could also to large leaps in the angle input for NAO's joints, should the controller forget to stop moving the STEM-tracker. Controlling the joints with methods that demand that you only perform one command at a time was therefore something that should be avoided.

3.1.2 Reactive methods

Parts of this section is taken from section 3 . 1 . 2 of my project thesis (Evjemo, 2016).

The other way to use joint control to move one of the robot's effectors, was by using *reactive methods*. These methods are non-blocking, which enables the robot to do several things at once, like moving both arm-effectors independently (Joint Control API, 2015). There were a few different methods for performing joint control included in the NAOqi SDK.

ALMotionProxy::setAngles

This method takes in the parameters *names*, *angles*, and *fractionofmaxspeed*, which are explained as follows:

- **Names:** The name or names of joints or chains that should move, for instance "HeadYaw" or "ElbowRoll".
- **Angles:** One or more angles corresponding to a joint or chain. The angle(s) must be given in radians.
- **FractionOfMaxSpeed:** A set value between 0 and 1 to describe how fast NAO's joint(s) should move to the given angle(s), relative to maximum speed. If 0, nothing will happen, while if this is set to 1, NAO will try to move the joint(s) to the given angle(s) at full speed. It can be wise to set this to 0.8 or 0.9, because maximum speed might make the robot unstable, or lead to oscillations in the effector because the joint reaches the desired angle so fast that it is unable to stop in time.

When running the IK-solver, the output should be the five angles for the robot's arm effector necessary in order to make the sensor in the hand reach a desired position and orientation. To realize this movement should therefore be as simple as to run the function `ALMotionProxy::setAngles` with the five angles in the arm effector as input, along with the values in radians, and a fraction of max speed.

ALMotionProxy::changeAngles

The NAOqi SDK also contains the function `ALMotionProxy::changeAngles`, which works similarly to the function `ALMotionProxy::setAngles`: It has the same three inputs, and is also a method for joint control. The difference is that this method does not *set* the joints in the angles that are given as input, but rather changes the joint-angles by this amount. In other words: Each angle-input is added to the existing joint-angle.

This might initially seem like a good method to use in this project. After all, the goal is to make NAO's arm effector follow *change* in position and orientation of the STEM-tracker. Therefore,

only registering the change in movement , and sending is as input to `ALMotionProxy::changeAngles` would perhaps seem like the most logical choice.

The problem is that in order to use joint control methods, it is necessary to get all of the angles for the effector that should realize the movement. In order to get these angles, the desired position and orientation need to be run through an IK-solver. Because the IK-solver is designed to give solutions for positions and orientations relative to its *torso*-frame (see section 2.2.2), it will return the full angles necessary to realize the movement. This means that in order to use a function which only looked at the *change* in the angle value, it would be necessary to subtract the new angle solution from the existing angle. This seems like it would be *more* work, not less. However, it was possible that this function could be faster than `ALMotionProxy::setAngles`, so it was considered worth testing if time allowed it.

ALMotionProxy::getPosition

Even though the system created in this project is based on joint control methods, it was necessary to use a Cartesian method to find the position and orientation of the robot's effectors at the beginning of the program (Cartesian Control API, 2015). As explained in section 2.2.3, the robot control would be performed while the robot was in the *initial position*. Therefore, it was only necessary to collect this information once, and use the collected data for position and orientation as constants in our calculations.

This method takes in the parameters *name*, *frame*, and *useSensorValues*, which are explained as follows:

- **Name:** The name or names of a joint or chain sensor, for instance "RHand".
- **Frame:** The desired frame for the coordinates sent to the robot. 0 for *Torso*, 1 for *World*, and 2 for *Robot*. As mentioned in section 2.2.2, the *Torso*-frame is always used in this project.
- **UseSensorValues:** Determines whether or not to use the sensors to find the position and orientation values. If true, the sensor values will be used to determine the position.

The function returns the position and Euler angle orientation of the sensor, given in meters and radians. The position and orientation for NAO's hand (sensor input "RHand" and "LHand") are shown in figure 3.4.

It would also have been quite easy to modify the program so that this function was simply run once at the beginning of each session, and the positions and orientations saved in some way. This would also make it possible to alter the robot's position when performing the arm control. The reason why this was not implemented from the beginning was to avoid an additional complication or problem area. If there was time, the program would be altered to include this functionality, but initially it was more practical to just use the constant data.

ALMotionProxy::getTransform

As will be explained later in the report, it turned out to be quite challenging to get valid output from the IK-solver. To make sure that the error was not caused by inaccuracy in the data for the initial position and orientation of NAO's arm effectors, the method `ALMotioProxy::getTransform` was also used. This function took in the parameters *name*, *frame*, and *useSensorValues*, which are explained in the previous subsection.

The only difference between `ALMotionProxy::getPosition` and `ALMotionProxy::getTransform`, is that the latter returns a vector of 16 floats representing the 16 values in the transformations matrix. In other words: both functions returns the Cartesian position, but one function returns the orientation in Euler angles, while the other returns the orientation in the form of a rotation matrix. See section 2.5.2 for a reminder of what data the transformation matrix describes.

3.2 Receiving tracker data from STEM

As explained in section 1.4.1, the Sixense STEM-system is in the BETA-testing phase, and not finalized for commercial use. Therefore, there is only a limited amount of documentation available, and many challenges with using the system had to be discovered and solved along the way.

In section 2.4 it was explained that receiving the tracker data from the STEM-system was one of these challenges. In section 3.2.1, I will discuss the limitations of the different solutions, and how this was to be solved in this project. In addition, the STEM-system and the NAO robot have defined their respective coordinate systems differently. In section 3.4.3 I will explain how this was solved, and how it had to be compensated for in the scripts.

3.2.1 Old approaches

As explained briefly in section 2.4, communication with Sixense STEM has been a challenge for as long as I have been using the system. While working with the system, different approaches have been used to solve the communication problem. For the work done in this master's thesis, it was important that the data collected from the STEM-trackers could, after some modifications, eventually be sent to the IK-solver.

The solution used in the summer project sent the data from the STEM-trackers directly to LabVIEW. If this approach were to be used in the master's project as well, the C++ based IK-solver and LabVIEW would have to be able to communicate somehow. LabVIEW can only interpret C or C++ scripts if they are compiled into a DLL file or a `.out` file (Using C/C++ Models, 2014). This approach might therefore mean that the IK-solver somehow would have to be included as a DLL-file as well, at least if the IK-solver was to communicate directly with LabVIEW.

The experience from working with the project thesis in the fall of 2015 had made it clear that creating DLL files from code written in C++ is both time consuming and challenging. The C++ in the IK-solver would first have to be wrapped to create regular C-code, before finally creating DLL-files. This would be very challenging in itself, but an additional problem would be my lack of depth-understanding of the scripts I would be working with. Therefore, a different method for using C++ scripts together with LabVIEW would be preferred.

The same challeng occurred for the solution used in the project thesis: If the communication was to happen through Python, it would still be necessary to create a DLL-version of the IK-solver. This seemed both challenging ant time-consuming, and I hoped to avoid this if possi-

ble. The most promising solution initially seemed to be the system created by Elling Ruud Øye, which collected the tracker data, and wrote it to file. This system was examined closer, to see if it provided all necessary data and functionality.

3.2.2 Øye's tracker-data program

Because the tracker-data program was created by one of my supervisors at SINTEF, he was availible to answer questions about the program's functionality and output. The focus was therefore on understanding what information this program could provide, rather than spending time understanding exactly how it worked. The program consisted of several scripts and files, but according to Øye, there was only one file that needed my attention: the executable file `sxCoreTest.exe`.

In order to connect the STEM-tracker to the base station, the `Sixense System Test`-program that came with the hardware had to be run on the computer. This was necessary in order to allow the trackers to connect to the base station, and to communicate with the computer. When asking one of the main developers, Alejandro Diaz, about why this was necessary (Redmine - Sixense, 2015), he replied:

"Just to make sure, are you running a Sixense application (SxTest)? Without a Sixense enabled application running, there is nothing for the devices to communicate with. There is no OS level driver/service for Sixense devices yet, so without a Sixense enabled application running, there is no one to manage the connection, update LED states, etc."

So, after running the `Sixense System Test`-program, the trackers would then be connected to the base station (at least when not experiencing any of the problems listed in section 3.8). After the trackers had connected, it was necessary to close the `Sixense System Test`-program before running the `.exe`-file, so that this program and Øye's program would not compete for connection to the trackers. I will not go into detail on how this worked, because the important thing for my thesis is what data this program returned.

While `sxCoreTest.exe` was running, tracking data from each of the five trackers would be written to text files named `device0.txt` -`device4.txt`. The data registered in one of these files is shown in figure 3.1.

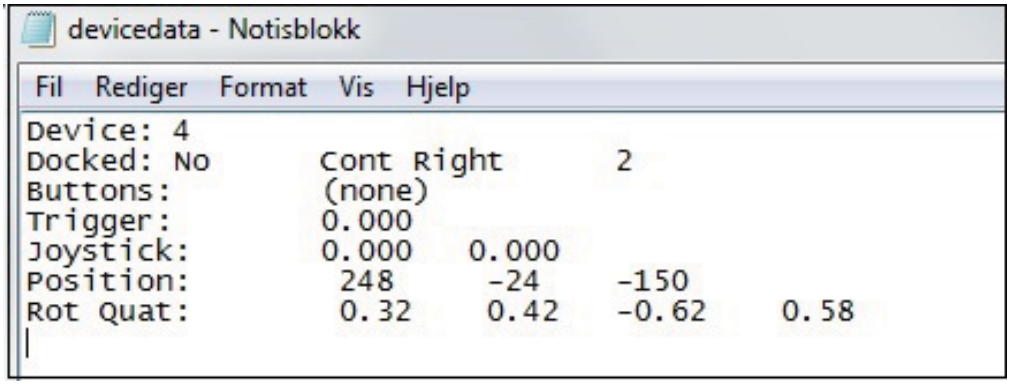

Figure 3.1: **Tracker data:** The following data from the given STEM-tracker was continuously written to a designated text file.

The different data shown in the text file is explained as follows:

- **Device:** Ranged 0 to 4 based on which order they were connected to the base station. In other words, this number could change from time to time, and was not a good indicator for which of the trackers this data belonged to.
- **Docked:** *Yes*/*No* tells whether or not the given tracker is placed in its docking station on the STEM-base station. Next is the name of the tracker, in this case the right controller, or *Cont Right*. The number at the end of this line is a constant number that tells us which tracker this is. The trackers are numbered 1-5 in the following order: Left controller, right controller, left pack, middle pack, right pack.
- **Buttons:** The two larger controllers have several buttons in addition to the tracking, because the system is meant for games. The buttons currently pressed are listed here, but this functionality is not relevant for this thesis.
- **Trigger:** The two larger controllers each have a trigger which gives out a value between 0.0 and 1.0 depending on how much the trigger button is pressed in. If unpressed the value is 0.0, and if pressed all the way in the value is 1.0.

- **Joystick:** The two larger controllers have joysticks, because the system is meant to be used for games. The two values returned are the x- and y coordinates for the joystick's position. In x-direction the value is -1.0 to 1.0 from left to right. In y-direction the value is -1.0 to 1.0 from down to up.

- **Position:** Returns the exact position of the magnetic tracker inside the given controller realtive to the magnetic tracker in the base station. The values returned are the x-, y-, and z-coordinates given in millimeters. The coordinates are given in the STEM-system's own coordinate system, and had to be transformed in order to be used for controlling the robot, as explained in section 3.4.3.

- **Rot Quat:** Returns the four quaternions representing the current rotation of the STEM-tracker. The order of the quaternions are q_ω, q_z, q_y, and q_x (this is discussed in section 3.4.2).

Data from the buttons and trigger will not be used in this project, but it is worth noting that this could easily be used to include functionality for opening and closing NAO's hand etc. The data is available - all that has to be done is to combine it with methods from the SDK. As explained in section 3.3.1, the only data the IK-solver needed was the position and rotation. During the summer project, the rotation matrix for the STEM-tracker was retrieved directly from the STEM-system. It *is* possible to get the matrix directly, but Øye's system did not include this functionality: it only returned the rotation in quaternions. Because the conversion from quaternions to rotation matrix seemed pretty straight forward, it did not seem necessary to ask Øye to help change his data collecting system.

On the previous projects, there had been some problems with identifying what data came from which tracker, as explained in section 3.2.3 in my project thesis (Evjemo, 2016). The number in each file identifying which of the trackers the data is from, solved this problem.

As long as the executable file was running, the text files would be updated continuously for all connected trackers. These coordinates and quaternions could then be read by for example

a LabVIEW-program or a C++-script, and used to do additional calculations. In addition, the tracking data was displayed in a command window, as shown in figure 3.2.

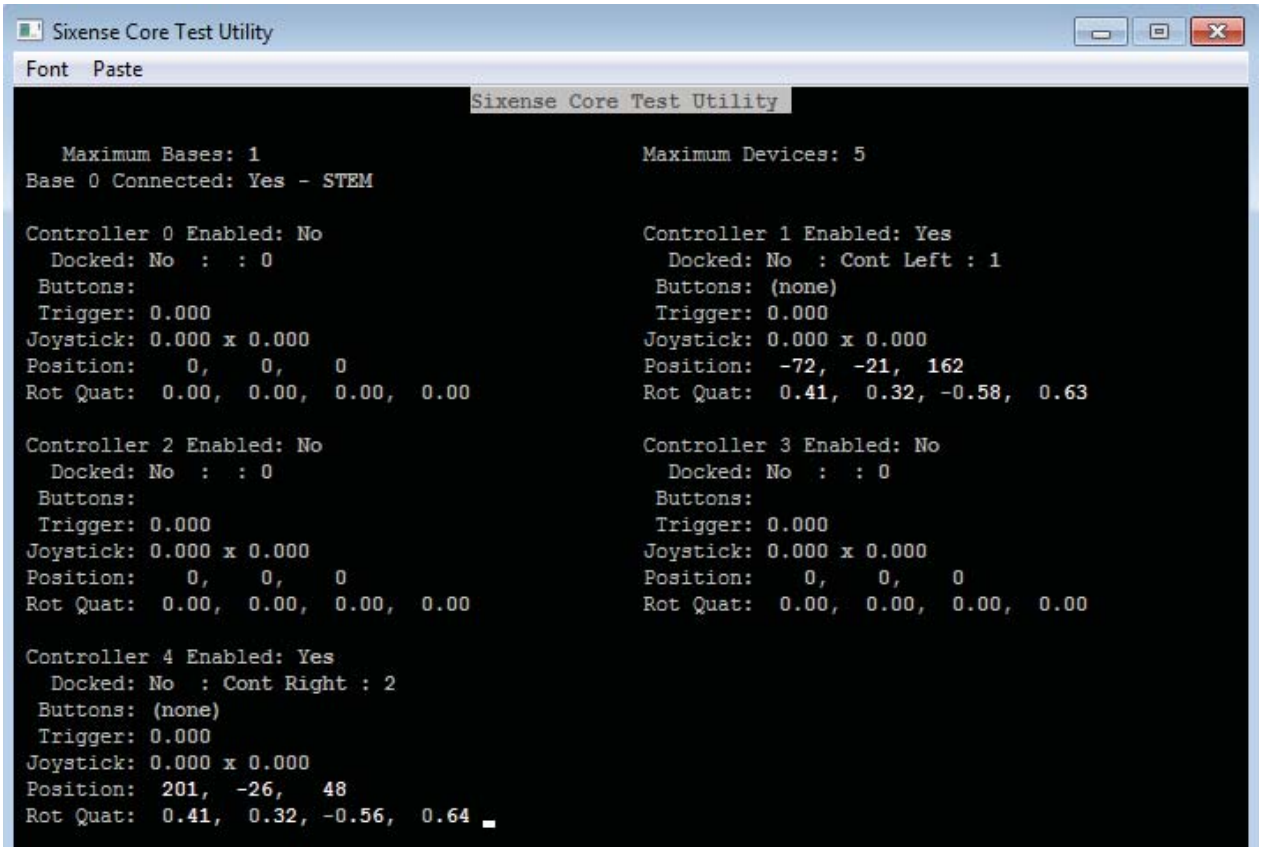

Figure 3.2: **Program running:** This window shows the tracker data for all five trackers. In this case, only two of the trackers are connected. The other three trackers would not connect because of the problems discussed in section 3.8

3.2.3 Advantages and limitations

Because the tracker-data program returned quaternions instead of Euler angles or the rotation matrix, this had to be implemented in our system. As explained in section 3.3.1, the IK-solver needed the transformation matrix representing the desired position and orientation of the end effector, in our case the hands. Therefore, it was necessary to find the rotation matrix from the Euler angles. The equations for this are presented in section 2.5.2.

This way of collecting data from the STEM-system seemed very accurate when comparing the registered data with the movements of the trackers. The registered tracker-data displayed in the

command window was updated in what felt like real-time, so it seemed like the *write to file*-approach had very low latency. In addition, the trackers would not disconnect and act funny very often, unlike when I was working on the project thesis. This would change during the course of the project, which will be discussed further in section 3.8.

If the IK-solver could read the data from the text files, it should be able to use this data to perform the necessary inverse kinematics and return the angles for the joints in the arm effector necessary to make the end effector end up in the desired position and orientation. How to access the angles returned by the IK-solver will be discussed further in section 3.5.5.

3.3 The inverse kinematics solver

The foundation for the work done in this master's thesis, is the analytic IK-solver for the NAO robot developed by N. Kofinas in 2012. The hope was that by combining the tracker data from STEM with his IK-solver, it would be possible to make the robot move its arms along the same trajectory as the trackers, and that this could also be done in something close to real-time. First, it was necessary to get some basic understanding of how the IK-solver worked, and to test that it worked properly when given valid input. It would also be necessary to find some way to include the IK-solver in the rest of the system.

3.3.1 A brief introduction to how Kofinas' system works

As will be discussed further in section 3.3.3, the C++-scripts that made up the inverse kinematics solver created by N. Kofinas were quite complicated. However, fully comprehending the complexity of the code was not necessary in order to use the scripts in this project. The most important thing to understand was how the input and output of the IK-solver worked, as this really determined the design of the rest of the system.

The IK-solver needs input in the form of a transformation matrix, which consists of a rotation represented by a rotation matrix, and Cartesian position coordinates. The position coordinates must be given in millimeters, and in NAO's *Torso*-frame, which is described in section 2.2.2.

Kofina's system also includes a FK-solver, which takes a given effector and its respective angles as input. The FK-solver returns a transformation matrix describing the position and orientation of the end effector when the joints are set to these angles.

The `main.cpp` file for Kofina's system is used as a foundation for how the IK-solver and FK-solver have been used in this project. This script starts with setting all angles of the five effectors, before running the FK-solver. The final position and orientation of the end of these five effectors are then returned in five 2D-arrays named output1-output5. These five arrays are transformation matrices describing the current position and orientation of NAO's five effectors: Left Hand, Right Hand,Left Leg, Right Leg and Head.

To get solutions from the IK-solver, it should only be necessary to set the transformation matrix corresponding with the desired effector to values describing the desired point and orientation, and run the program. If there is a solution, the angle solution will returned in an array called `results`. If not, this array will be empty, and the text "No valid solution" will be displayed in the command window.

In this project, the IK-solver was used by building the Makefile in the Visual Studio Command Prompt. This created an `exe` file, which could be run in the command window. It is worth mentioning that because I was not able to edit the Makefile, the main.cpp-file had to be named the same throughout the process. In other words: The backup files could be named `mainedit.cpp` etc., but the file currently in use had to be named `main.cpp`.

3.3.2 Testing the existing scripts

As will be explained in section 3.3.3, the simplest approach for combining the IK-solver in the C++ files with LabVIEW, was thought to be re-creating the IK-solver in another programming language. But no matter if the IK-solver was to be implemented in a new programming language or not, it would always be necessary to compare the output from potential new IK-solver files with the output from the old C++-files. And if the original files were to be used, it would still

be necessary to check if the results from this IK-solver made sense.

The inverse kinematics in the C++ files created by N. Kofinas had of course been checked by other professors at his university, and was approved by the IEEE. It therefore seemed safe to assume that they worked as they should. Still, some testing should be done to make sure that the IK-solver was using the same movement frame for NAO, if the output made any sense, and simply to familiarize with the system.

As mentioned in section 3.3.1, the original `main.cpp`-file created by N. Kofinas was a simple test program for both inverse and the forwards kinematics calculations.

- First, all of the angles were set manually.
- Then, the FK-solver was run, and the x-, y-, and z-coordinates for the point of the end effector was returned.
- Lastly, the IK-solver was run with the transformation matrix given by the forward kinematics solver as input.

This was an easy way to test if the calculations done by the system were in fact correct. If so, the angles returned by the IK-solver should be equal to the angles set in the beginning of the script.

When testing this program, the only focus was on the inverse and forwards kinematics for the arm effectors, because these were the effectors I hoped to control. The arm effectors each have five DOFs, distributed as explained in section 2.2.4. This meant that five angles had to be set before the FK-solver was run, and the IK-solver should return the same five angles.

The inverse kinematics solver seemed to work well for both the right and the left arm. When running the test program, it returned approximately the same angles. The variations were less than 1 µm, and could be considered irrelevant, as the control of the robot will not be that accurate anyway.

However, one exception was discovered. If the initial joint angles were set so that that the elbow yaw and wrist yaw were parallel, i.e. the arm was completely straight, the angles returned by the inverse kinematics solver would not be identical to the initial angles. However, the difference between the two angles remained the same.

One example of this is shown in figure 3.3, where the initial values for elbow yaw and wrist yaw are 0.3 and 0, respectively. However, when running the position and orientation through the IK-solver, the return values were 1.97 radians for elbow yaw, and -1.67 radians for wrist yaw. This meant that the angles had been altered dramatically. But, as we can see, the difference between the two yaw angles were the same. The reason for this is that when the arm is kept straight, like in figure 2.11, the end effector will end up with exactly the same orientation for the two different angle combinations. In fact, as long as the elbow yaw and wrist yaw are rotating around the same axis, any solution for the two yaw-angles is valid, as long as the difference remains the same. It is also worth mentioning that the new values for elbow yaw and wrist yaw were both within the valid bounds shown in figure 2.8: 1.97 radians is approximately 112°, and -1.67 radians is approximately 96°.

Several tests were run to make sure that the IK-solver did in fact return values equal to the initial values in cases where the arm was not straight, and it worked every time.

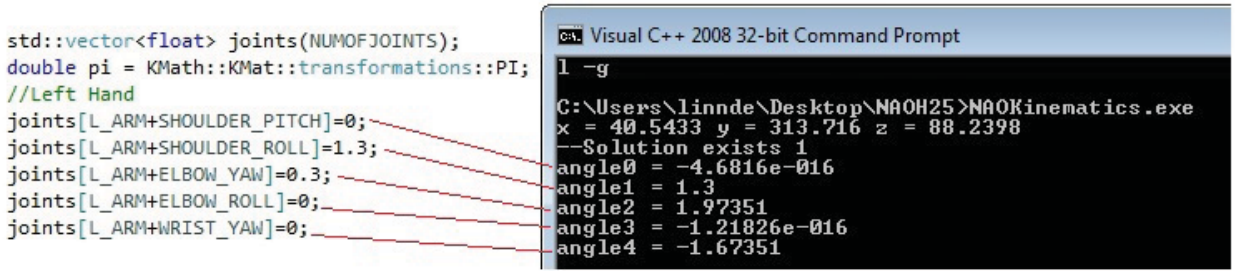

Figure 3.3: **Change in angles:** Here we see that the elbow yaw and wrist yaw has changed values. However, the difference between the two stays the same, so the end effector would still be positioned with the desired orientation.

In addition to testing the IK-solver with data directly from the FK-solver, it was also tested with input for the left arm effector retrieved from NAO using output from the function ALMotion-

Proxy::getPosition when NAO was in its initial position. This will be explained in section 3.5.2.

3.3.3 Using original scripts or not

This project uses the IK-solver that was developed in N. Kofinas thesis (Kofinas, 2012), and the code for the IK-solver was implemented in C++ as part of his thesis. It is possible to implement MatLab-code directly in a LabVIEW-program, but LabVIEW does not have compatibility with C++ scripts.

On N. Kofinas' GitHub, both C++ and MATLAB files for the IK-solver were available. Because LabVIEW is compatible with `.m`-files, see section 2.1.1, I wanted to use the MATLAB code as a basis for the new implementation of the inverse kinematics. However, when contacted, N. Kofinas explained that the MATLAB-code that was available on GitHub was not finished, and could not be used in its current state.

After consulting with my supervisors at SINTEF Fisheries and Aquaculture, who have a lot of experience with LabVIEW and its compatibilities, the next plan was to try to re-create the IK-solver in LabVIEW. This could either be done using block diagrams, or by using LabVIEW MathScript or MatLab-blocks. Because the IK-solver is mostly basic math operations on matrices and vectors, the functionality of MathScript or MatLab-blocks should be enough.

As mentioned in section 3.3.1, a lot of time went into getting the old IK-solver to run. Making decisions based on what would be least time consuming therefore became more important for the rest of the project. After studying the scripts thoroughly, I decided against trying to implement the code in a different programming language. It was thought to be too time consuming, especially when considering that there was no guarantee that the new system would work properly. If this was a wise decision or not will be discussed further in section 4.4.1.

3.3.4 Early correspondence with N. Kofinas

To make some parts of the code clear, N. Kofinas was contacted through e-mail, see appendix B. He explained that the inverse kinematics solver naturally depended on being given a valid set of parameters in order to return a solution. This meant that if it was impossible for the arm effector to reach a desired point, the IK-solver would *not* return the closest feasible solution. Initially the idea was that the possibility of getting no valid solution from the IK-solver would make it be necessary to implement some exception handling in case the desired position was not in the feasible area. However, this turned out to be quite a big deal, and much more severe for the functionality of the system than first expected, as will be explained further in section 4.1.

N. Kofinas also explained that when given valid input, it was very unlikely that a given position and orientation of the end effector could have several feasible solutions for the joint angles in the arm effector. Still, if the IK-solver was to return multiple solutions, a possible way to handle this would be to simply compare the solutions with the current joint angles, and choose the solution closest to the current robot pose.

As will be explained further in section 3.5, the problems I experienced when trying to use the IK-solver in the full system caused me to try to contact Kofinas again. At the time of submitting this Master thesis, he has not replied. However, it eventually became clear why it was so difficult to get the analytic IK-solver to return valid solutions for the STEM tracker's trajectory, and this will be discussed in section 4.1. I also wanted to ask Kofinas why it was unlikely that the IK-solver would return multiple solutions. As was mentioned in section 3.3.2, certain combinations of position and orientation should potentially have infinite solutions due to parallel joint axes. Perhaps the IK-solver only returns the first solution it finds, or perhaps it is so sensitive that it only considers one of the solutions to be accurate enough. This thesis will not answer these questions, because the system never became functional enough for this to be a concern.

3.4 Processing the tracked data

The data collected from the STEM-tracker had to be processed before it could be sent to the IK-solver. It had to be combined with the data describing the robot effector's current position and orientation, it was necessary to compensate for differences in frames, and the data had to be transformed into a form that was compatible with the rest of the system.

3.4.1 Getting initial position and orientation of robot arm

This section is partially based on section 3.1.2 of my project thesis (Evjemo, 2016).

As explained in section 2.2.3, the NAO robot's "initial position" was chosen to be the starting point for all movement of the robot arms. This position would be used both for testing done on the actual NAO robot, and on simulated robots used for testing.

In order to be able to account for the starting position of NAO's hands, it was necessary to collect the data describing the initial position and orientation of NAO's hands.

Finding the initial position and orientation of the arms could be done using the method `ALMotionProxy::getPosition` from the NAO'robots SDK (Cartesian Control API, 2015). Given an effector and a movement frame, this function will return the Cartesian position and Euler angle orientation of the end effector, like seen in the output in figure 3.4. The Euler angles returned here are based on a rotation about the fixed frame, more specifically NAO's *Torso* frame, which was explained in section 2.2.2. In order to use the initial rotation in further calculations, it was necessary to convert the Euler angles to a rotation matrix, as will be explained later in this section.

The only thing separating the coordinates for the right and the left hand, is the y-coordinates. Other than that, the coordinates are almost exactly mirrored, as shown in figure 3.4. The same goes for the orientation of the robot hands, which are identical except for opposite orientation about the x-axis. As shown in figure 2.5, the robot's x-axis is defined straight forwards, and fig-

ure 1.4.2 showing the initial position confirms that the arms have this opposite rotation around this axis. In the edited `main.cpp`-file, these initial values were set manually in the script based on the position and orientation in figure 3.4.

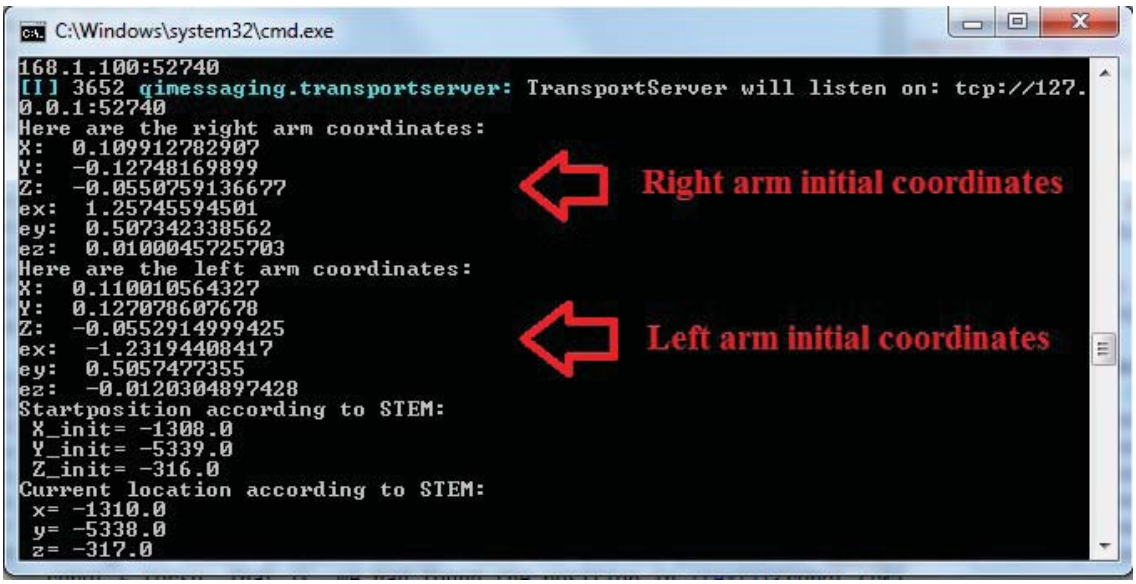

Figure 3.4: **Initial values:** Here we can read what the coordinates for NAO's right and left arm is, given in the *Torso* frame. Image taken from (Evjemo, 2016)

After first using the `ALMotionProxy::getPosition`-function to get the initial Cartesian coordinates and Euler angles for the robot arms, it became clear that it would be much more useful to get the rotation matrix directly. This would of course shorten the script's execution time, but more importantly: It would mean one less possible source of error. Therefore, the function `ALMotionProxy::getTransform` was used instead: Given an effector name and a movement frame, this function will return the complete transform matrix of the given effector, see section 3.1.2.

By adding the movement and rotation of the tracker together with the initial position and rotation of NAO's arms, like explained In section 2.5.3, i was possible to find the Cartesian coordinates and rotation matrix describing the desired point for the end effector to end up. How to do this is explained in section 3.4.4.

3.4.2 Some necessary transformations

When combining different hardware and software with individual specifications regarding units, frames etc., there were a lot of transformations and calculations that had to be done. This section sums up some of the modifications that had to be done on different kinds of data before it was compatible with the rest of the system.

Euler angles to rotation matrix

As shown in figure 3.4 and explained in section 2.2.3, the initial position and orientation of the arm effectors could be found using the function `ALMotionProxy::getPosition`. This function would return the rotation given in Euler angles relative to the robot's *Torso* frame. As explained in section 2.5.3, calculations on partial rotations are possible when using rotation matrices. Because of differences in frames, it would be easier to have the initial orientation of NAO's arm effectors represented as a rotation matrix rather than in Euler angles. Using rotation matrices also removes some uncertainty because an Euler angle rotation can be described as only one rotation matrix, while one rotation matrix can describe several combinations of Euler angle rotations. This is explained in section 2.5.2.

When knowing the initial Euler angle rotation, the rotation matrix could be found using the MatLab-function `eul2rotm`, see section 2.1.1. It was important to remember that the function `eul2rotm` assumes that the Euler-vector has the Euler angles in the order z, y and x. The rotation matrix gotten from MatLab could then be included in the script as constants.

Including the initial position and orientation as constants seemed like a possible approach since the idea was to always start the robot in the same initial position, as described in section 2.2.3. Still, it would have been more ideal to extract the initial position of the arm effectors at the beginning of the program. As mentioned earlier in this section, a different function from the NAOqi SDK was used to get the initial rotation matrix directly, skipping the MatLab-step. Using this function would make it even easier to include this in the script, rather than setting position and orientation as constants.

Quaternions to rotation matrix

In section 3.2.2 it was explained that the position and orientation of the STEM-trackers were written to file, and that they were represented in millimeters and quaternions. It is possible to do partial rotation calculations using quaternions, but I had worked more with rotation matrices than with quaternions, and consequently felt more comfortable with this way of representing rotation. I therefore chose to convert the rotations described in quaternions to rotation matrices, using the formulas shown in figure 2.14.

When using quaternions, it is always necessary to check if they are given in the order q_ω, q_x, q_y, q_z or in the order q_x, q_y, q_z, q_ω, as explained in section 2.5.2. To check this for the STEM-system, the following approach was used:

- First, the STEM-tracker was placed in the orientation which relative to STEM's frame represents a rotation of zero. To find this position, the Sixense STEM application that came with the system was used.
- Then, MatLab was used to see how the quaternions looked for a rotation equal to zero. That is: Euler angles e_x=e_y=e_z=0, or a rotation matrix equal to the identity matrix, R=I.
- Lastly, the quaternions from the STEM-tracker with zero rotation were compared to the results from MatLab.

The testing showed that when the rotation is zero, the quaternions have these values: $q_\omega = 1$, $q_x = q_y = q_z = 0$. The output from the STEM-tracker was approximately " 1 0 0 0 ", which gave this order of the quaternions: q_ω, q_x, q_y, q_z.

In order to double-check that the order of the quaternions was registered correctly, two examples from Sixense' own documentation were used (Sixense SDK Overview, 2012).

According to the documentation from Sixense, the point of zero rotation for the flat trackers, or *packs*, was when they were lying flat on the table with the LEDs facing the ceiling, pointing in

Figure 3.5: **Checking rotation matrices:** According to the documentation from Sixense, these two rotations of the tracker relative to the base station, should result in the two rotation matrices R_a and R_b (Sixense SDK Overview, 2012).

the same direction as the base station. This is shown to the left in figure 3.5. Being the point of zero rotation, the rotation matrix created from the quaternions in this position should result in the identity matrix, illustrated in matrix R_a.

$$R_a = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}, R_b = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & -1 \\ 0 & 1 & 0 \end{pmatrix}$$

The documentation also states that if the tracker is placed aiming at the ceiling, like shown to the right in figure 3.5, then the resulting rotation matrix should be similar to matrix R_b. Unfortunately, the C++ script did *not* result in these two matrices when processing the quaternions for the tracker in this position. This meant that the order of the quaternions were not correct after all.

As explained in section 3.2.2, the rotation matrix was collected directly from the STEM-tracker during the summer project and the work on my project thesis. One thing that led to some confusion back then, was that the rotation matrix was indexed differently than what is the norm, which is explained in detail in section 3.2.2 in my project report (Evjemo, 2016). According to

documentation from Sixense, the rotation matrix is stored in column order, each of the columns representing one of the three transformed unit axes (Sixense SDK Overview, 2012).

$$R_{norm} = \begin{pmatrix} r00 & r01 & r02 \\ r10 & r11 & r12 \\ r20 & r21 & r22 \end{pmatrix}, R_{STEM} = \begin{pmatrix} r00 & r10 & r20 \\ r01 & r11 & r21 \\ r02 & r12 & r22 \end{pmatrix}$$

This meant that the rotation matrix collected directly from the STEM-tracker in earlier work, was composed so that it actually was the transpose of a rotation matrix following the norm. The next idea for achieving the correct rotation matrix from the four quaternions, was therefore that the quaternions might be given in an order, or calculated in a way which made the resulting rotation matrix have the same composition. The order of the elements in the matrix was therefore changed, and tested both for the order q_ω, q_x, q_y, q_z and the order q_x, q_y, q_z, q_ω. However, when placing the pack in the orientations illustrated in figure 3.5, none of these matrices corresponded to the matrix R_a or R_b.

The order of the quaternions was eventually determined by testing the quaternion values when the tracker was rotated 180° around each of the three axes, always starting in the "zero-rotation" position shown to the left in figure 3.5. When the rotation was zero, the q_ω would be 1 or -1, while the rest of the quaternions were 0. When rotating the controller 180° about one of the three other axes, the quaternions corresponding to this axis would be 1 or -1, while the others would be zero. The quaternion output was checked with MatLab, to see which of the quaternions this rotation corresponded with.

This last approach showed that the order of the quaternions was in fact q_ω, q_z, q_y, q_x.

Scaling of translation

The tracked Cartesian position for the STEM-trackers was given in millimeters. When using the function `ALMotionProxy::getPosition`, the position of NAO's effector was given in meters.

The IK-solver needed the input in millimeters. It was therefore necessary change the unit of the position of NAO's effector from meters to millimeters, simply by multiplying the value by 1000.

3.4.3 Compensating for differences in coordinate systems

This section is partly based on section 3.2.2 of my project thesis (Evjemo, 2016).

The STEM-system has defined its coordinate system, differently from the NAO robot. For NAO, the x-axis points forwards, the y-axis points to the left, and the z-axis points upwards, as seen in figure 2.5. For the STEM-system, the x-axis points to the right, the y-axis points upwards, and the z-axis points backwards, like seen in figure 3.6. Because the IK-solver uses the NAO robot's coordinate system, the tracked data received from STEM had to be transformed into NAO's coordinate system. In other words, it was necessary to implement that:

- The STEM-system's x-axis was the NAO robot's negative y-axis
- The STEM-system's y-axis was the NAO robot's z-axis
- The STEM-system's z-axis was the NAO robot's negative x-axis

Fixing the Cartesian coordinates was quite basic, and could be done simply by changing which of the Cartesian values read from file was connected to which coordinate, as shown in figure 3.7. However, it was also necessary to transform the rotation matrix R_2^1 before it could be sent to the IK-solver.

Like for the Cartesian coordinates, this meant changing the output of the rotation matrix so that it fitted the robot's coordinate system. If the system had used Euler angles, this could have been done by just changing the output according to the axes, like for the Cartesian position. But because the IK-solver needed the rotation matrix, as explained in section 3.3.1, the process was a bit more complicated.

As explained in section 3.2.2, the tracker data describing the orientation of the STEM-tracker

Figure 3.6: **STEM coordinates:** The coordinate system that the STEM-trackers follow are defined like shown in this figure, which is differently from the robot's coordinate system. Picture taken from (Evjemo, 2016)

```
double initx = -atof(initpostokens[3].c_str()); //NAO x = STEM -z
double inity = -atof(initpostokens[1].c_str()); //NAO y = STEM -x
double initz = atof(initpostokens[2].c_str()); //NAO z = STEM y
```

Figure 3.7: **Changing the axes:** When registering the Cartesian position of the STEM-conroller, the axes are changes to fit NAO's coordinate system

.

was registered in quaternions. Because changing frames for rotation matrices is quite straight froward, the first step was therefore to transform the rotation from quaternions to the corresponding rotation matrix. Next, the registerred rotation had to be transformed from STEM'c coordinate system to NAO's coordinate system.

At first, this was done by rotating the STEM-coordinate system 90° around the y-axis, and then

-90° around the x-axis. The rotation matrices for these rotations are given as follows (Spong, Hutchinson and Vidyasagar, 2006):

$$R_y = \begin{pmatrix} \cos(90°) & 0 & \sin(90°) \\ 0 & 1 & 0 \\ -\sin(90°) & 0 & \cos(90°) \end{pmatrix} = \begin{pmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ -1 & 0 & 0 \end{pmatrix}$$

$$R_x = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos(-90°) & -\sin(-90°) \\ 0 & \sin(-90°) & \cos(-90°) \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & -1 & 0 \end{pmatrix}$$

When registering the rotation matrix R_1^0 from STEM, the transformation to the robot's coordinate system was done by multiplying it with these to rotations, as seen in equation 3.1

$$R_1^0 = R_{STEM} \cdot R_y \cdot R_x \tag{3.1}$$

The rotation matrices R_y and R_x were post-multiplied, because the rotations are about the current axes, not the axes in the world coordinate system, as explained in section 2.5.4.

However, testing showed that this approach really only worked if the STEM-tracker's initial orientation corresponded to what the STEM-system considered "zero" rotation relative to the base station (see section 3.7.2). My assumption had been that the tracked orientation were always given relative to the STEM-system's base station, but this turned out to be wrong. The reason was that the registered orientation was given relative to each tracker's current orientation. In other words: the frame in which the rotation was registered, followed the rotation of the STEM-tracker continuously. This meant that is was necessary to compensate for the STEM-tracker's initial orientation relative to the base station as well, not only the difference between the frames of the base station and the robot.

While this initially seemed like it would only create a more complicated system, this also made it possible to trick the system into registering the orientation in the robot's frame from the beginning. How this was done will be explained further in section 3.7.2.

3.4.4 Real movement for robot

When the change in position of the STEM-tracker was registered, this had to be combined with data for the position and orientation of NAO's arm effector. The initial rotation of NAO's arm effector was described by the rotation matrix R^0_{1NAO}, while the rotation of the STEM controllers after starting the program was described by the rotation matrix R^1_{2STEM}. Because the goal was to make NAO's arms follow the movement of the STEM-trackers, the movements after starting the program should be identical:

$$R^1_{2NAO} = R^1_{2STEM} \tag{3.2}$$

The new rotation matrix for NAO's arm, R^1_{2NAO}, was therefore found by multiplying the two rotation matrices together (Spong, Hutchinson Vidyasagar, 2006):

$$R^0_{2NAO} = R^0_{1NAO} \cdot R^1_{2STEM} \tag{3.3}$$

The goal was to get the final transformation matrix T^0_{2NAO} describing the full, current position and orientation, and send this into the IK-solver created by N. Kofinas. To do this, it was necessary to find the new position as well.

As explained in section2.5.3, this was done by adding the change in the Cartesian coordinates of the STEM-tracker to the initial position of the effector:

$$newpos_{NAO} = initpos_{NAO} + (newpos_{STEM} - initpos_{STEM}) \tag{3.4}$$

$newpos^0_{NAO}$ is the desired Cartesian position of the effector. By combining this Cartesian position with the rotation matrix R^0_{2NAO} representing the desired orientation, it was possible to get the transformation matrix T^0_{2NAO} describing what the robot's effector should do. The final transformation matrix was then sent into the IK-solver, as explained in section 3.3.1 and section 3.3.2.

3.5 Trying to get valid output from the IK-solver

The methods and transformations necessary to crate the transformation matrix describing the desired position and orientation for the robot's arm effector were implemented directly in the `main.cpp` file of the IK-solver. As explained in section 3.3.1, the reason was that the IK-solver was used by running an `exe`-file created when building the scripts. In this section I will present some of the tests that were run on the system and the scripts to try and get valid output from the IK-solver. Some of the choices made when editing the `main.cpp` file are also explained. All the testing was done for NAO's left arm effector.

3.5.1 Initial testing: movement between two points

It turned out to be extremely difficult to get valid output from the IK-solver. Testing was done by writing the initial position and orientation of a STEM-tracker to a file called `deviceX0.txt`. Then, the tracker was moved slightly, and the program run again, this time writing the value to a different text file called `deviceX1.txt`. A more detailed description on how this was done is included in section 3.5.3. With the tracker data for two different points, the change in position and orientation between these two points could be calculated using the methods explained in section 3.4.

However, when the new transformation matrix for NAO's arm effector was sent as input to the IK-solver, all it returned was "No valid solutions". As explained in section 3.3.2, the IK-solver had been tested to check that it worked the way it should as long as the input was valid. Still, during testing I never seemed to get a valid solution when trying to run the IK-solver for a movement of the STEM-tracker, no matter how small or simple the movement was.

Because the IK-solver had not only been created as a thesis by Kofinas, but had also been published as a scientific article by Kofinas and several professors (Kofinas, Orfanoudakis, and Lagoudakis, 2013), it seemed safe to assume that it should work properly when given valid parameters. Therefore, it seemed very probable that the error was somewhere in my own code. So a series of different tests were performed to try to identify the problem. These are described in the following sections.

3.5.2 Checking for type errors

As explained in section 3.3.1, I did not feel that I fully understood how the IK-solver scripts worked. Therefore, my initial though was that the problem could be re-setting the values in the transformation matrix which was sent into the IK-solver. Unlike Python, which had been used for most of the programming when writing my project report (Evjemo, 2016), C++ is a type-sensitive language, meaning that any mix-ups between floats, ints, or doubles etc. could be critical.

Because of this, I tried setting the transformation matrix to values retrieved from NAO using output from the function ALMotionProxy::getPosition when NAO was in its initial position. As explained in section 3.1.2, this function returns the position and Euler angle rotation of a given effector. The output that was used is showed in figure 3.4. The Cartesian position only had to be scaled down from meters to millimeters. Using the `eul2rotm`-function in MatLab (see section 2.1.1), I was able to get the rotation matrix describing the same rotation as the Euler angles. The rotation matrix values were used to replace the old values in the transformation matrix.

This time, the IK-solver did return a valid solution in the form of five angles, as shown in figure 3.8. To make sure that this really corresponded with NAO's initial position, they were tested on the 3D-model of the robot in LabVIEW. The left arm of the 3D-model ended up as shown in figure 3.9, which convinced me that the IK-solver worked properly when given valid input. At a later stage, this was also tested on the virtual robot in LabVIEW.

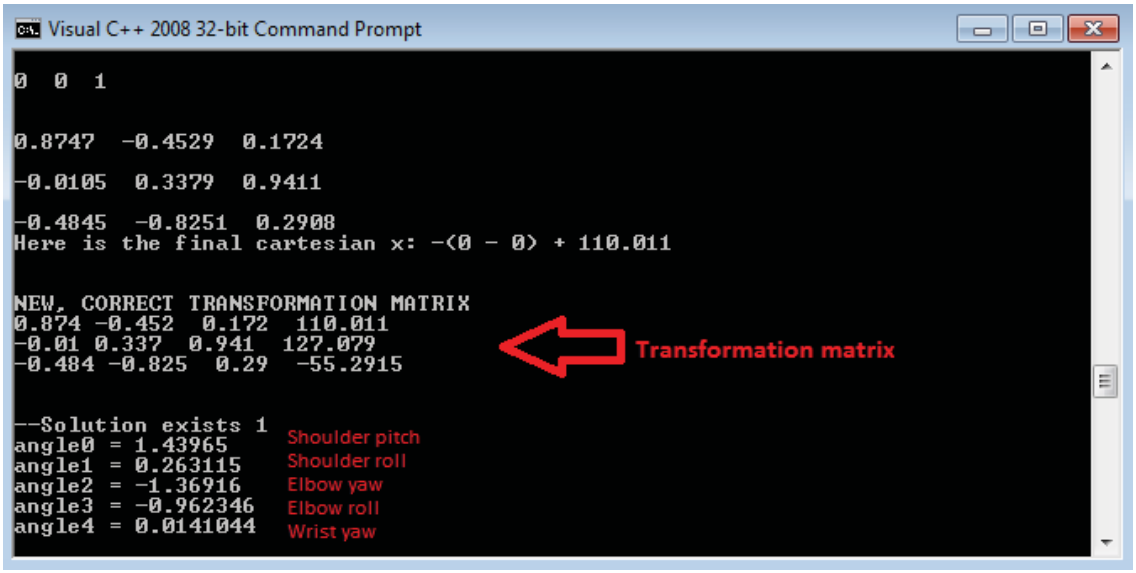

Figure 3.8: **Valid solution:** When manually setting the transformation matrix in the C++-script to match the transformation matrix for NAO's left hand when being in the *Initial Position*, the IK-solver returned the output shown above for the five angles in NAO's left arm effector.

From the results illustrated in figure 3.9, it seemed clear that the position of the left arm matched the position of the left arm on the actual robot when in the *Initial position*.

The conclusion of this test was that the problems with getting valid output from the IK-solver were not caused by wrong types, or other errors related to changing the values in the transformation matrix.

3.5.3 General troubleshooting

The next step was to double-check if the problem was caused by some mistake in the edited scripts related to units or simple calculations. It would have been much harder to discover errors in the system if the troubleshooting was done when the C++-scripts were combined with LabVIEW etc, so in a sense it was a good thing having to do these tests at an early stage.

One example was double-checking the calculations of the changes in the tracker's position and

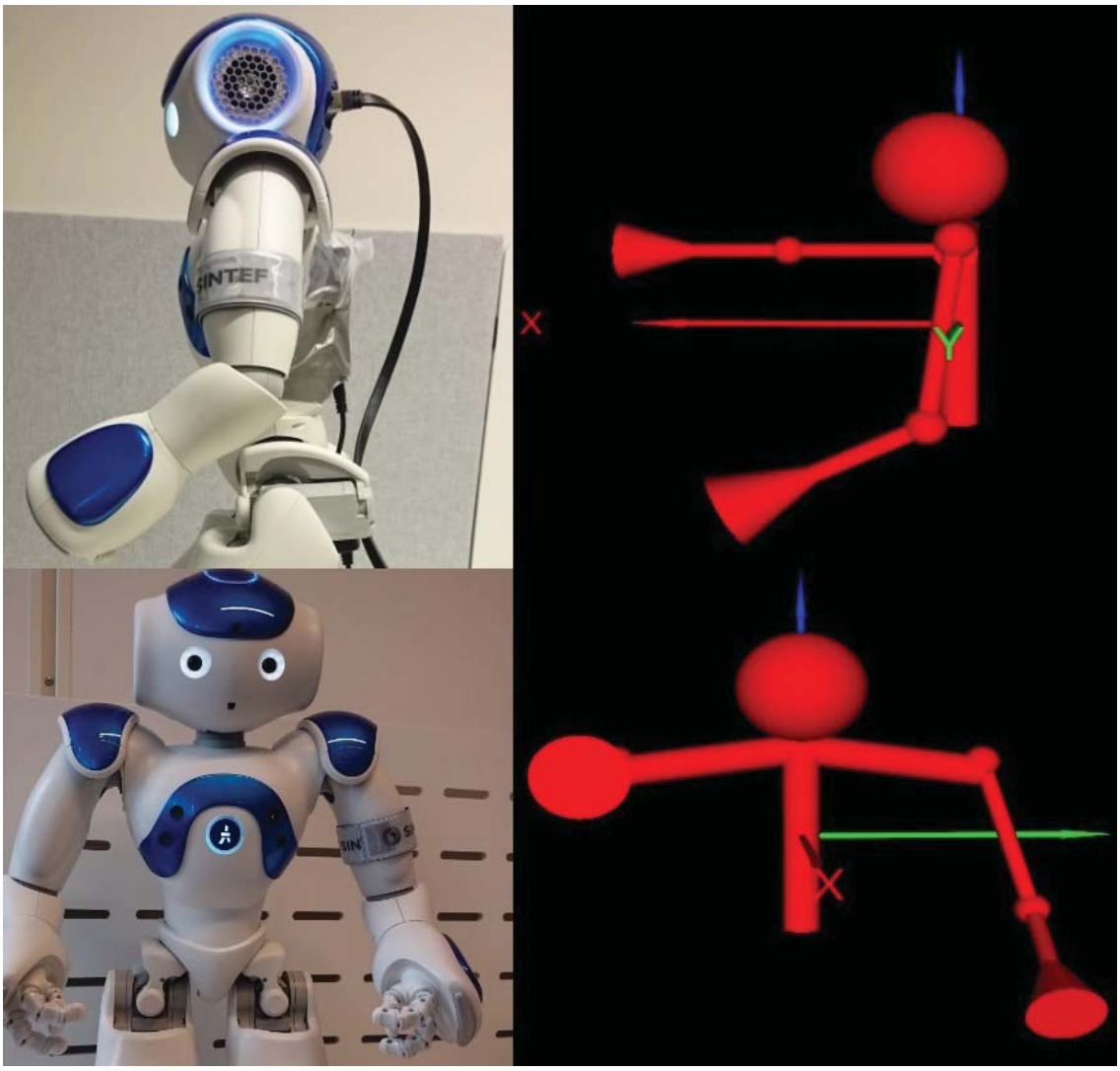

Figure 3.9: **Matching angles:** When setting the five angles on the left arm of the simple 3D-model on the right, the final position and orientation of the arm matched the actual robot in its *Initial position*. As mentioned in section 3.5, testing was only done on the left arm of the robot.

orientation, and see if the calculated movement made any sense when comparing it to the movement in the real world.

A simple test was run to check if the inverse kinematics solver in C++ did in fact return results that made sense. The program that wrote the tracker data to file was run. Two backup-versions of the given text file was then saved at two different times, with the tracker being moved slightly in between. The two backup files are shown in figure 3.10, where `device30` was saved first, and `device31` was saved after the tracker was moved.

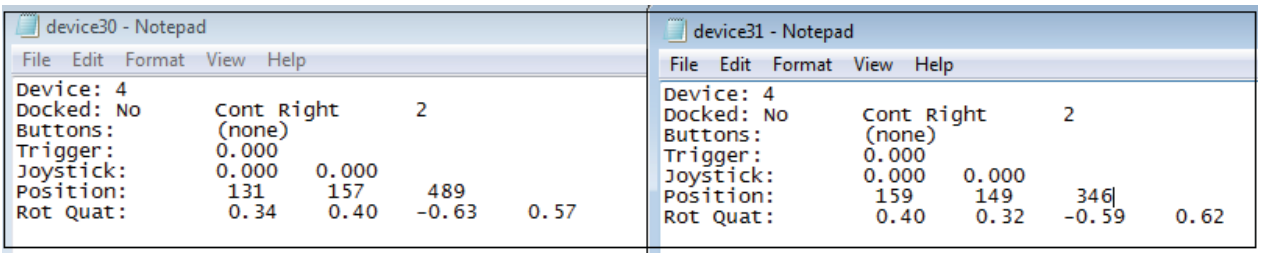

Figure 3.10: **Small changes:** Here we see the tracking data from the same tracker at two different points in time. It is clear that the rotation is almost identical, as the quaternions are quite similar, but the controller has moved a lot in the STEM-system's negative z-direction. This means that it has moved*forwards*, or in the NAO robot's positive x-directions, see section 3.4.3.

If the script worked as it should, the transformation matrix created at the end of the script should contain the new rotation matrix and Cartesian coordinates for NAO's arm effector if it began its movement in NAO's `initial position`, see section 3.4.2.

Checking if the rotation matrix made sense or not is a little complicated, and requires using the methods explained in section 2.5.3. Checking the Cartesian coordinates in the transformation matrix should on the other hand be relatively straight forward, so this was step one. The initial position of the effector the test was run on, NAO's left hand, is:

- x-coordinate: 110.01
- y-coordinate: 127.08
- z-coordinate: -55.29

When changing the tracked coordinates for the STEM-tracker to match NAO's coordinate system like described in section 3.4.3, the movement given in millimeters ends up being:

- In x-direction: -(346 - 489) = 143
- In y-direction: -(159 - 131) = -28
- In z-direction: 149 - 157 = -8

Added together, the new Cartesian coordinates for NAO's hand should be:

- x-coordinate: 110.01 + 143 = 253
- y-coordinate: 127.08 - 28 = 99
- z-coordinate: -55.29 - 8 = -63

If the scripts worked properly, these values should be found in the right-most column of the final rotation matrix. However, when running the scripts, the final transformation matrix ended up as shown in figure 3.11. This meant that the maths in the script was somehow wrong.

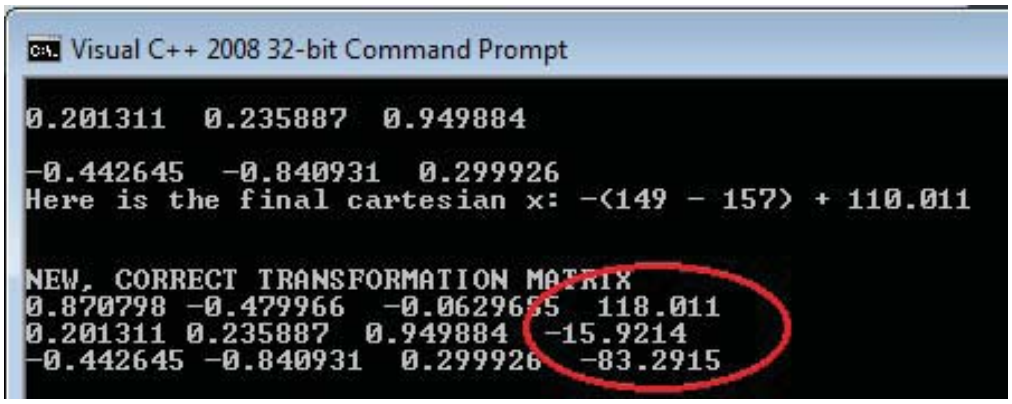

Figure 3.11: **Wrong output:** It is clear that the calculated Cartesian coordinates do not match what is known to be the correct answer. Somewhere in the program there must be some mathematical error.

In this case, the mistake was simply that the axes had been flipped one too many times, because the difference in coordinate systems had already compensated for earlier in the script.

Several tests like this were run in order to double check that the position and rotation values were saved correctly, and that calculations were done right. All details regarding this testing will not be included in this thesis, as most of them are quite elementary, though time-consuming.

3.5.4 Reducing accuracy

As explained in section 3.5.2, problems with variable types etc. when setting the values in the transformation matrix was not why the IK-solver would not return valid solutions. So far, the only way of getting valid solutions from the IK-solver, seemed to be sending in positions and orientations for the effector which were know to be valid because they were acquired by using the `ALMotion:getPosition`-function on the actual robot. If the IK-solver worked the way I hoped, it should be possible to get valid solutions for other desired positions and orientations for the arm effector as well.

The next potential problem investigated, was that the rotation matrix and Cartesian positions that were sent to the IK-solver might be *too specific*. In other words: There might be so many decimals that the exact position and orientation did not correspond to a valid solution, even though another point close-by might give a solution. Reducing the number of decimals was therefore tested, which meant sending a less accurate rotation matrix and position to the IK-solver. The number of decimals was reduced from 6 significant figures, to 3 decimals after the decimal point. For the Cartesian coordinates, this meant reducing the number of decimals by one or two decimals, as the old values had six significant figures.

This kind of change in the number of decimals would make the system less accurate, which could end up being a problem in itself. But it still had to be tested, as a slightly less accurate system seemed better than a system that did not work at all. Also, the reduction in accuracy would probably not be too noticeable in the end. After all, reducing the number of decimals from 5 to 2, only means reducing the accuracy of, for instance, the Cartesian position with one tenth of a millimeter. When considering that the person controlling the robot is unlikely to be able to control his own movements with that kind of accuracy in the first place, it seemed negligible. Still, possible implications would have to be discussed and evaluated further if reducing the number of decimals did in fact help.

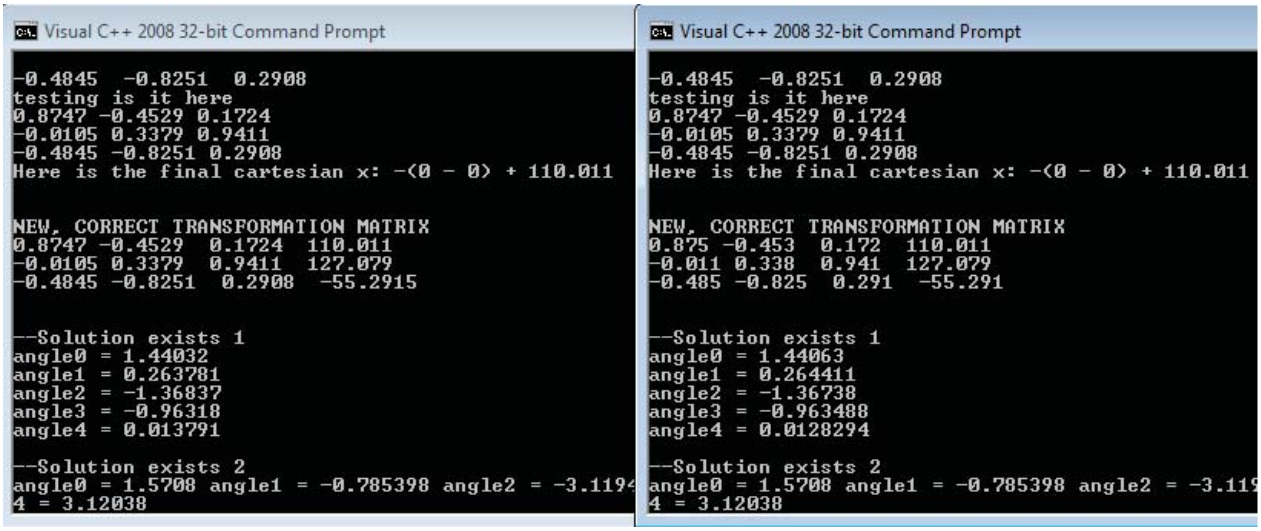

Figure 3.12: **Change in solution:** As shown in the command windows above, reducing the number of decimals for the values in the transformation matrix also changed the values of the angles returned by the IK-solver. The command window to the left is shows the original transformation matrix, while the scaled version is shown to the right.

Changing the number of decimals of the values in the transformation matrix on known, valid positions, had some effect on the solution returned by the IK-solver, as shown in figure 3.12. However, it did not seem to solve the problem, because any solution based on a movement made by the STEM.tracker would still return "No solution".

In fact, if the number of decimals was reduced to only two decimals after the decimal point, the IK-solver would give "No valid solution" even for the position and rotation taken directly from the robot, which I *knew* was supposed to have a solution. This made it necessary to reconsider the approach of reducing the number of decimals, at least for the values representing the rotation matrix. Making these values less accurate might lead to the rotation matrix no longer being full rank. Therefore, I abandoned this approach. Reducing the accuracy of the Cartesian position, on the other hand, proved useful on a later stage of the project. This will be explained further in section 3.7.3.

3.5.5 Continuous update of movement

As explained in section 3.5.2 and 3.5.4, neither setting the types in the transformation matrix nor the number of decimals for the values in the transformation matrix seemed to be the problem. However, as explained in section 3.5, the testing was done by finding the movement between two specific Cartesian positions and orientations. Therefore, it was possible that I had only been unfortunate with my choice of values. Even though about 15-20 different movements had been tested, this was still a relatively small set of points and orientations.

The next step was therefore to create a program which would continuously check for valid solutions for a desired position and orientations while the STEM-tracker moved. Because writing to and reading from text files was already used in different parts of this project, this method was used here as well. The new program should:

- Register and save the initial position and orientation values of the STEM-tracker, preferably to a text file.
- Save the current position and orientation values for the STEM-tracker. These values should also be written to a text file, but a different one than the initial values.
- Continuously update the position and orientation values for the STEM-tracker in close-to real-time. This could hopefully be done using a loop.

Since the IK-solver was used through the `exe`-file, as explained in section 3.3.1, it was natural to try and use this file directly in the new program. The hope was to not have to spend time learning a new method unless it became absolutely necessary.

A simple program called `IKtester.vi` was created in LabVIEW, and the block diagram part of this program is shown in figure 3.13. Before running `IKtester.vi`, Øye's program for writing the STEM-tracker data to file, `SixenseTest.exe` (see section 3.2.1), had to be run in the background, before `IKtester.vi` was run.

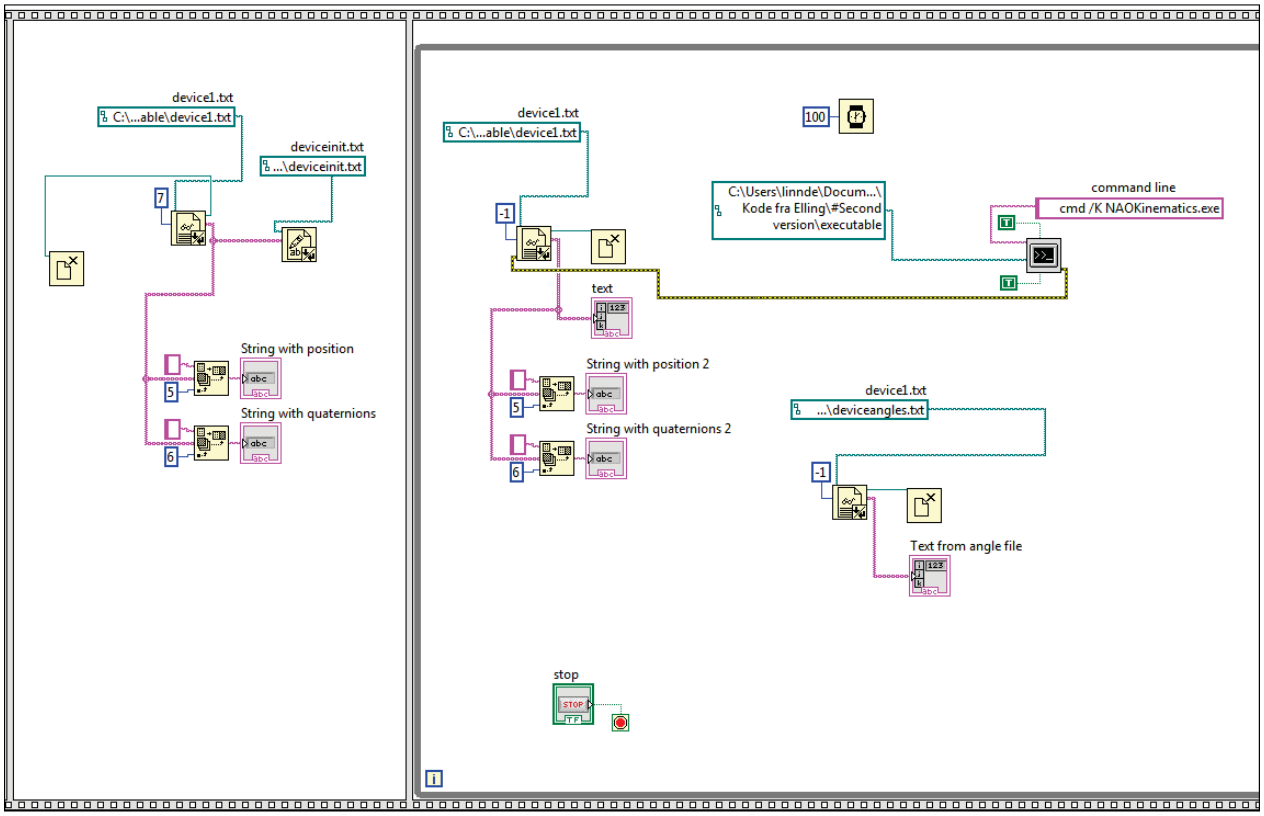

Figure 3.13: **Continuous IK-solver testing:** The `exe`-file is calculating the movement between the tracker data in `deviceinit.txt` with the data currently written in `device1.txt`. The data in the last file is continuously updated by Øye's program, see section 3.2.2.

This program would first read the tracker values from the text file corresponding to the STEM-tracker in use. See section 3.2.2 for details regarding which file corresponds to which tracker. These values were then copied to the text file `deviceinit.txt`. This was only done once, at the very beginning of the program, meaning that the text in `deviceinit.txt` remained the same for the entire session. This was necessary in order to find the actual movement of the STEM-tracker relative to where it started, as explained in seciton 2.5.3.

In the next part of the program, shown to the right in figure 3.13, the `NAOKinematics.exe`-file was run continuously within a while-loop. Because Øye's file `SixenseTest.exe` was still running in the background, the data written in `device1.txt` was updated continuously, see section 3.2.1. Simultaneously, some small changes were done in `main.cpp`. Another textfile,

```
vector<vector<float> > result;
result = nkin.inverseLeftHand(output1);
if(!result.empty()){
    cout << "--Solution exists 1" << endl;

    //If there is a solution, the corresponding angles are written
    //to the file named "deviceangles.txt"
    ofstream anglefile("deviceangles.txt");//

    for(int j=0; j<result[0].size(); j++){
        cout << "angle" << j << " = " << result[0][j] << endl;
        anglefile << result[0][j] << '\n';//
    }
    cout << endl;
    anglefile.close();//

}
else{
    cout << "No solution for left arm" << std::endl;
    cout << '\n' << endl;
    //If there is no solution, the line "No valid solution"
    //is written to "deviceangles.txt"
    ofstream anglefile("deviceangles.txt");//
    anglefile << "No valid solution" << '\n';//
    anglefile.close();//
}
```

Figure 3.14: **Angle solution:** The C++-script was edited so that if there was a valid solution, the five angles were written to a text file called `deviceangles.txt`. If there was no solution, the line "No valid solution" was written to this text file instead.

`deviceangles.txt`, was updated with these angles, as shown in figure 3.14. If there was no valid solution, the text *No valid solution* was written there instead.

For this program to work, it was only necessary to include the block with functionality for running the `NAOKinemtaics.exe`-file through a command window, which can be seen the top-right corner of figure 3.13. In addition, it was necessary to continuously check what was written in `deviceangles.txt`. Some functionality for continuously reading from the two files `device1.txt` were also included. This was necessary in order to see that the values were written to `device1.txt` fast enough, and that the values made sense. It was possible to see if `NAOKinematics.exe` gave valid solutions for any of the new positions and orientations simply by reading from the `deviceangles.txt`-file: If any valid solutions were generated, the text read from this file should be angle values.

AS it turned out, all that was ever returned was the text *No valid solutions*, so the IK-solver still made no sense. It is also worth mentioning that LabVIEW would sometimes crash during these tests, though not so often that it made it impossible to run the tests. However, LabVIEW did seem to crash more often when moving the STEM-controller to an infeasible point, so that the IK-solver would return "No solution".

3.5.6 Normalizations and orthogonality

To begin with, I had assumed that all quaternions that were collected from the STEM-trackers were already normalized. This had been tested by checking if equation 3.5 was true for a random sample of quaternions collected from the STEM-tracker.

$$\sqrt{q_x^2 + q_y^2 + q_z^2 + q_w^2} = 1 \tag{3.5}$$

Having normalized quaternions was necessary in order to create an orthogonal rotation matrix by using the equations described in figure 2.14, as these were valid if and only if the quaternions were normalized (Using Quaternion to per-form 3D rotations, 2011).

It still seemed necessary to double-check if *all* of the collected quaternions were normalized, and if the resulting rotation matrix were indeed orthogonal. The IK-solver could perhaps not return a valid solution because it was not able to handle a non-orthogonal rotation matrix as input. The C++ script was modified to check if $q_x^2 + q_y^2 + q_z^2 + q_w^2 = 1$ every time new quaternions were read from file. It was unnecessary to check for the square root, because $+\sqrt{1} = 1$.

It turned out that the quaternions were in fact not always normalized, which meant that $q_x^2 + q_y^2 + q_z^2 + q_w^2$ sometimes added up to a number which was a few hundredths larger than 1. It was only a very small deviation, but it was possible that this was the reason why the IK-solver did not work as it should. Luckily, normalizing quaternions is done quite easily by dividing each of the quaternions with whatever number you *do* get from the left side of equation 3.5. The script was

modified to always normalize the tracked quaternions, regardless of whether or not they were normalized already. This was OK because trying to normalize an already normalized quaternion will not change anything, as this would only mean dividing each quaternion by 1.

Creating a rotation matrix based on normalized quaternions, should ensure that the resulting rotation matrix was indeed orthogonal, as explained in section 2.5.2. Still, things can go wrong when doing numeric calculations. For example: $0.67 + 0.67 = 1.34$, which can be rounded off to 1.3. But if the addends are rounded off before they are added together, the result will be: $0.7 + 0.7 = 1.4$. These kinds of simplifications can result in small irregularities, and give a different result than what was expected. It was therefore necessary to check it the rotation matrix was indeed always orthogonal.

To start with, this was done my checking if the determinant was equal to 1, which turned out not to be the case. The determinant would almost always end up as a number slightly larger than 1, like *1.000000342*. This could seem like an insignificantly small deviation, but it was still possible that the IK-solver could not handle anything but absolute accuracy. For orthogonal matrices, the product of the matrix and the matrix' transpose equals the identity matrix, as explained in section 2.5.2. Testing showed that this was not the case for the rotation matrices generated in this system, which was not unexpected considering that the determinant was also inaccurate. The next step was therefore to find a way to get the *correct*, orthogonal rotation matrix.

The method of polar decomposition was considered, and is described in section 2.5.2. This method takes a non-orthogonal matrix, and gives us the closest orthogonal matrix. Because there are several steps to this method that demands matrix multiplications, it seemed quite comprehensive to implement in the script. Especially considering that this would have to be done every time a new rotation matrix was created from the quaternions received from the STEM-tracker.

Because implementing the polar decomposition method seemed both challenging and time-consuming, it was important to make sure that it was necessary. The idea was that the IK-solver

was very sensitive to non-orthogonal rotation matrices. If this was really the case, then the rotation matrix collected directly from the NAO robot in a valid posture (see section 3.3.2), which *did* get a valid solution from the IK-solver, should be orthogonal. However, this matrix' determinant was also slightly larger than 1, with a margin just as large as for the other matrices. This is shown in figure 3.15, where it is also shown that the product of the rotation matrix and the rotation matrix' transpose was not exactly equal to the identity matrix either.

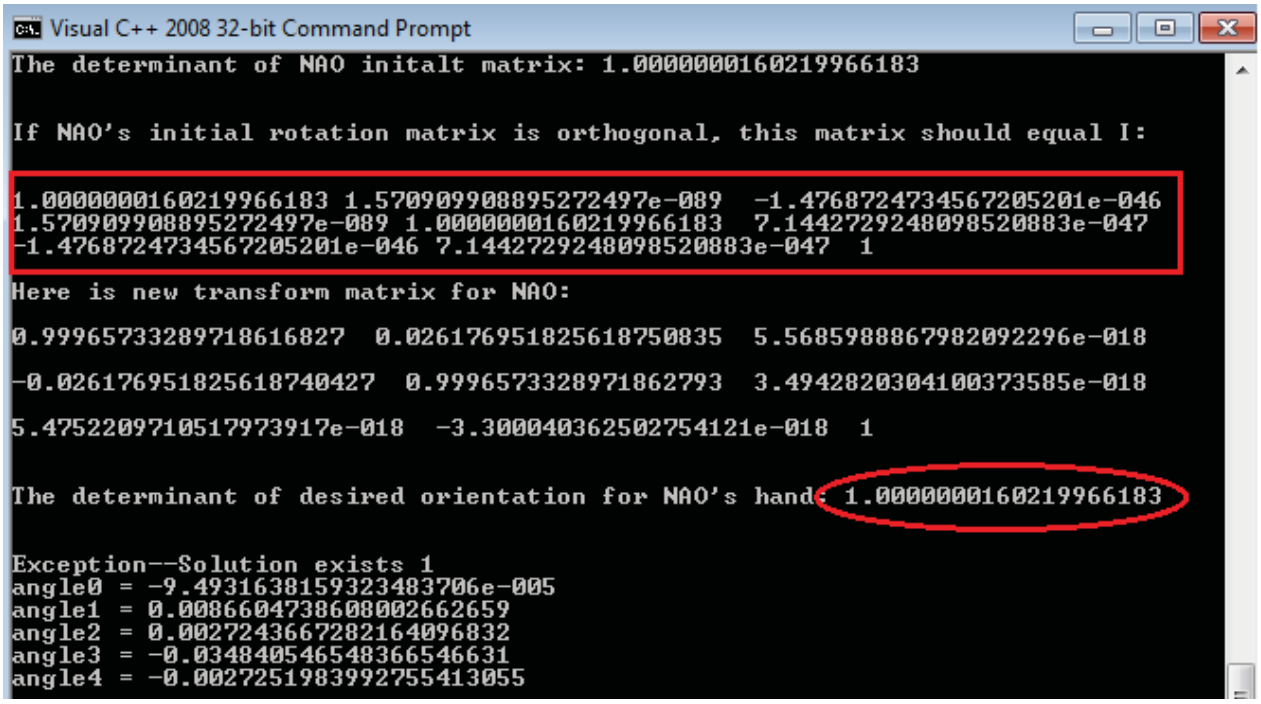

Figure 3.15: **Checking for orthogonality:** Here is the results when checking if the rotation matrix is orthogonal. The red square at the top shows that the product of the matrix R and R transpose does not equal the identity matrix, and further down it is shown that the determinant is in fact larger than 1. Still, the IK-solver returns a valid solution.

This was proof that the IK-solver could indeed handle slightly inaccurate rotation matrices. The polar decomposition method was therefore not implemented. Making sure that the rotation matrices were always orthogonal might be a good functionality in a fully-working system, since rotation matrices are supposed to be orthogonal. However, seeing as the system did not give any valid output at this point, it did not seem like a good idea to use time on implementing this functionality at this stage of the project.

3.5.7 Alternative approach: Do most calculations in MatLab

Because it seemed impossible to get any valid output from the IK-solver, I started wondering if the problem might be that the formulas created for matrix calculations in my edited `main.cpp` file were not correct. The output of these functions had been cross-checked with built-in methods in MatLab to check that they would in fact invert a matrix correctly etc. Even so, it started to seem plausible that errors made in the calculations had to be the problem.

An alternative LabVIEW-program was therefore created. As explained in section 2.1.1, LabVIEW has MatLab-blocks that allows you to use many MatLab-methods directly in LabVIEW. The new program would read the initial and current tracking data for the STEM-tracker directly from the two text files `deviceinit.txt` and device1.txt, just like the system described in section 3.5.5. However, this program would not convert the quaternions to rotation matrices using the formulas in figure 2.14, or do matrix multiplications based on formulas I had programmed myself. Instead, the quaternions would be sent to the MatLab-block, and all computations done using MatLab-methods. The MatLab-block would then return the final transformation matrix describing the desired point and orientation of NAO's arm effector.

The output from this new program ended up being almost exactly the same as the output of the C++ script. Some decimals had changed, which were to be expected when doing a series of calculations in two different programming languages, each with their own set of norms. When testing with values collected from the initial position of NAO's hand, the new transformation matrix would give close to the same solution as when using the transformation matrix from my C++ script. But it was still impossible to find a valid solution with a final transformation matrix based on the movements of the STEM-tracker.

This was of course also good news in the sense that it made me more confident that the functions I had implemented in C++ were correct. Still, the problem was not solved. Because the MatLab-based program in LabVIEW was slow, and would crash quite often, the calculations were again done directly in the C++ script after these series of tests.

3.6 Testing an alternative IK-solver

As explained in section 3.5, getting valid output from Kofinas' IK-solver proved to be very problematic. I tried discussing what the problem might be with the students I had worked with on the summer project in 2015, seeing as they also had experience with working with the NAO robot. As it turned out, one of them, Åsmund Pedersen Hugo at the institute of Marine Cybernetics, had developed an analytic IK-solver for the NAO robot in MatLab during his project thesis in the fall of 2015. I was not aware that he had developed this kind of system, or I might have considered using it from the beginning. MatLab is directly compatible with LabVIEW, as explained in section 2.1.1, which might have made the system structure less complicated.

Hugo had developed an analytic, full-body IK-solver for the NAO robot in MatLab. His solution had never been tested on the physical NAO robot, and Hugo informed me that some of the specifications he had used for joint length and joint angle limitations might not be accurate. Still, at this point I felt that it would be better to test an IK-solver on an experimental stage, than to be stuck with a thoroughly tested IK-solver that would not give any valid output. After all, I had both my own 3D-model in LabVIEW and the virtual robot described in section 2.3 to test the IK-solver on. Even if Hugo's IK-solver proved to be inaccurate, there was no risk of damaging the physical robot.

Unfortunately, when taking a closer look at Hugo's IK-solver, more specifically the IK-solver for the arm effectors, it became evident that this IK-solver would not work on the simulated robots in my project. The reason was that Hugo had made an assumption regarding the DOFs of the arm effector that did not match the physical robot. Hugo's IK-solver was based on the elbow yaw and wrist yaw of the arm effector always rotating about a parallel axis, namely the robot's underarm. In other words, it assumed that the elbow yaw was a rotation about the underarm. The joint angles representing the rotations of the elbow yaw and wrist yaw had therefore been combined, and the arm effector was considered to have only 4 DOFs instead of 5.

As explained in section 3.3.2, the elbow yaw and wrist yaw does in fact rotate about the same

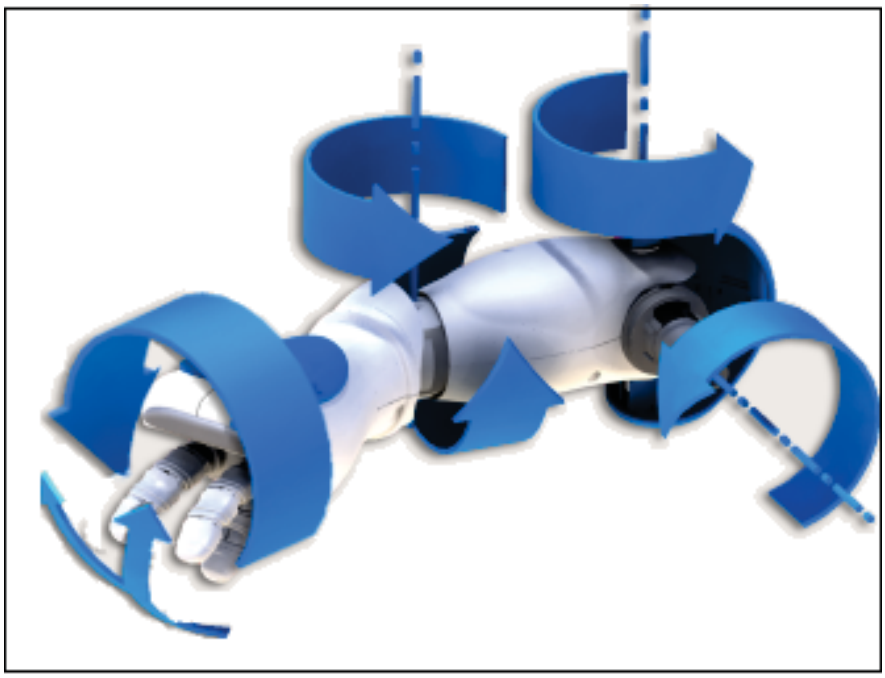

Figure 3.16: **Actual DOFs:** This figure shows the arm effectors 5 DOFs, and about which axis each of the joints rotates. As shown here, the elbow yaw is a rotation about the overarm, whereas the wrist yaw is a rotation about the underarm. Picture taken from Choreographe.

axis when the robot arm is kept straight. But, as shown in figure 3.16, the elbow yaw on NAO's arm effector is in fact a rotation about the overarm. This implies that when the arm is bent slightly, the two joints no longer rotate about the same axis. Because the alternative IK-solver was based on an arm effector with a different structure and different DOFs than the real one, I only did basic testing of it, never including it in my complete system. Hugo's solution seems both valid and functional, just not for a robot arm with the same DOFs as NAO.

3.7 Achieving valid output - a revelation

After what seemed like countless tests and modifications to the script, it was finally possible to get valid solutions from the IK-solver. However, this was only under very specific circumstances, which lead to some quite eye-opening realizations related to the methods used in this project. In this section it will be explained how these final tests were done, and how they lead to valid joint angle solutions from the IK-solver.

3.7.1 Changing NAO's initial position

After many attempts to make the IK-solver return a valid input, it seemed like the structure of the input was not the problem. For instance, the problem was not that the rotation matrix-part of the transformation matrix was not orthogonal, because the IK-solver could handle small irregularities, as explained in section 3.5.6. But it was possible that the combination of orientation and position that were sent to the IK-solver was infeasible.

The next step was to test a simple rotation movement of the robot hand, controlled by the STEM-tracker. The rotations that were sent to the IK-solver , were given relative to the robot's own coordinate system. In the `initial pose` used for the robot, the hand effector was kept in an orientation which made it impossible to realize a simple rotation about only one axis in the robot's frame. However, if the start-position of the robot was changed so that the arms were stretched out in front of the robot, the arms would be parallel with the robot's x-axis. This would mean that a rotation about the x-axis would be realizable by simply changing the value of elbow yaw or wrist yaw, as explained in section 3.3.2.

The position where NAO has his arms stretched out in front of him corresponds to all of the joint values in the arm effectors being zero (Poses, 2015). The pose is shown to the left in figure 2.5, and in figure 2.11. This was also one of the pre-programmed poses in the NAOqi SDK, which meant that the virtual robot could be sent to this position using methods in Choreographe. From there, it was possible to get the values for position and orientation of the hand by using `ALMotionProxy::getPosition` or `ALMotionProxy::getTransform`, just like for the initial position (Joint Control API, 2015). These values could then be included in the script as constants, making the IK-solver consider this as the robot arm's initial position and orientation.

However, sending a simple rotation of the STEM-tracker to the IK-solver would still not return a valid solution.

3.7.2 Simplifying the approach for rotation tracking

As explained in section 3.4.3, registering the orientation of the STEM-tracker turned out to be more complex than expected. In this section it is explained how the tracker data was collected in a way that made it directly compatible with the NAO robot's coordinate frame.

Instead of using the larger controllers, one of the flat STEM-trackers, or *packs*, were used for testing. As shown in figure 1.2 and 1.3, the electromagnetic tracker inside a controller is slightly tilted when the controller is placed on a flat surface, while this is not the case for the packs. Therefore, it was easier to test specific rotations about a single axis relative to the base station or the robot on a prismatic pack than on a controller. According to the documentation for the STEM-system, the orientation of this kind of tracker should be "zero" relative to the base station when placed in the position shown on the top left in figure 3.17 (Sixense Entertainment Inc., 2012).

When placed in this position, a rotation of 90° about each of the three axes was tested, and the rotation matrix that was generated for each movement was compared to the standard rotation matrices introduced in section 2.5.2. By studying the signs of the values in these matrices, which corresponded to the sines and cosines of the registered angles, it was possible to check how the current frame had defined each of the three axes, and if a specific rotation about each axis was considered to be positive or negative. These tests showed that the frame for the tracker in this orientation was actually oriented differently than for the base station, as shown to the top left of figure 3.17. In order to get to the same frame as the base station, the tracker had to be rotated 180° about the z-axis, which made the controller end up in the position shown to the upper right in the same figure.

As discussed in section 3.4.3, the rotations necessary to get from the frame of the base station to the frame of the robot, were a positive rotation of 90° about the current y-axis, followed by a rotation of −90° about the current x-axis. These rotations were performed directly on the tracker, as shown in the lower half of figure 3.17. When the telemanipulator held the trakcer in this final orientation while facing the base station at the beginning of a tracking session, the change in

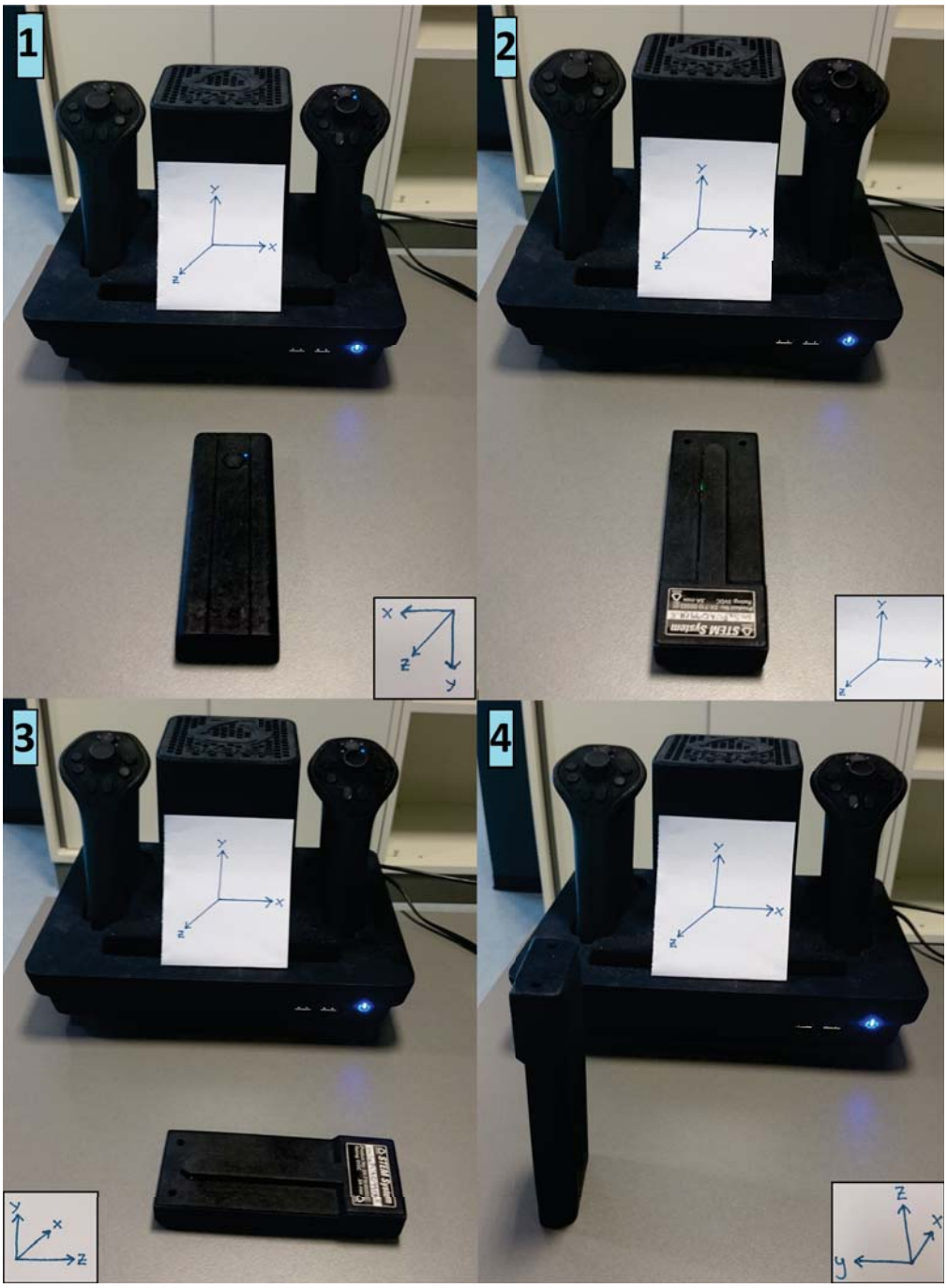

Figure 3.17: **Changing frame for the tracker:** Here are the four steps for changing the frame of the tracker to match the frame of the robot. The rotating frame of the tracker is shown in the bottom left corner of each step, while the frame of the base station remains fixed.

orientation would be registered in the robot's frame. For example, the small rotation shown in figure 3.19 would be registered as a positive rotation about the x-axis. This meant that no additional transformations were necessary for the rotation, other than combining it with the initial

rotation of the robot's hand, see section 3.4.4. The position was still tracked relative to the base station's frame, and had to be transformed by swapping the axes, as described in section 3.4.3.

The new approach for tracking the change in rotation made it easier to double-check that the registered orientation and the actual orientation were identical. This was not necessarily a very good way for controlling the robot in the final telemanipulation system, because using a tracker from an arbitrary start position would be more intuitive. But at this point, the priority was to see if it was possible to get the IK-solver to work at all. It is worth mentioning that the tests described earlier in this chapter were done with this new tracking approach as well, but the IK-solver would still refuse to return a valid solution for any of the movements that were tested.

To simplify the movements even more, the initial position of the robot was changed so that the arm effectors were pointing straight forward as described in section 3.7.1. This should mean that the small rotation about the robot's x-axis shown in figure 3.19 was feasible: Seeing as the arms pointed straight forward, parallel to the robot's x-axis, it should only require a small change in elbow yaw or wrist yaw, as mentioned in section 3.3.2.

When testing rotations of the STEM-tracker, it felt natural to assume that the centre of rotation was in the middle of the STEM-tracker. But, as shown figure 1.3, the electromagnetic tracker in the STEM-tracker is in fact placed at one end. If the goal was to get the STEM-tracker to rotate about its own axis, it was important to keep this in mind, and try to make the rotation happen about the point where the electromagnetic tracker was located.

But even when it was possible to check that the change in orientation registered by the STEM-tracker should be feasible, the IK-solver would refuse to return a valid solution.

3.7.3 Focus only on rotation

As explained in section 3.7.2, it was challenging to rotate the STEM-tracker exactly about the point where its electromagnetic tracker was located. This meant that there would be some change in Cartesian position, not only rotation. To check how large the change in Cartesian

Figure 3.18: **Slight rotation:** This shows a small rotation about .

position was, the difference between the initial and current position was written to screen.

The results are shown in figure 3.19, and it is clear that it is difficult for the telemanipulator to rotate the controller about a fixed point. The Cartesian position would change with a few millimeters even when trying to keep the tracker's center of rotation completely still. To see if the IK-solver would give a solution for a rotation only about the x-axis, the script was changed so that the change in position was ignored. This meant that the change in orientation of the tracker was combined with the initial rotation of NAO's hand, but the Cartesian position stayed fixed in the initial position.

To begin with, it was still not possible to get a valid output. But when I tried reducing the accuracy of the position to millimeters instead of meters as well, the IK-solver would finally return valid solutions. It would return solutions continuously as long as the movement was restricted to a simple rotation about the x-axis, so it was evident that ignoring the changes in position and keeping the rotation about the x-axis made the movement feasible. What this actually meant in the long run, will be discussed in section 4.1.

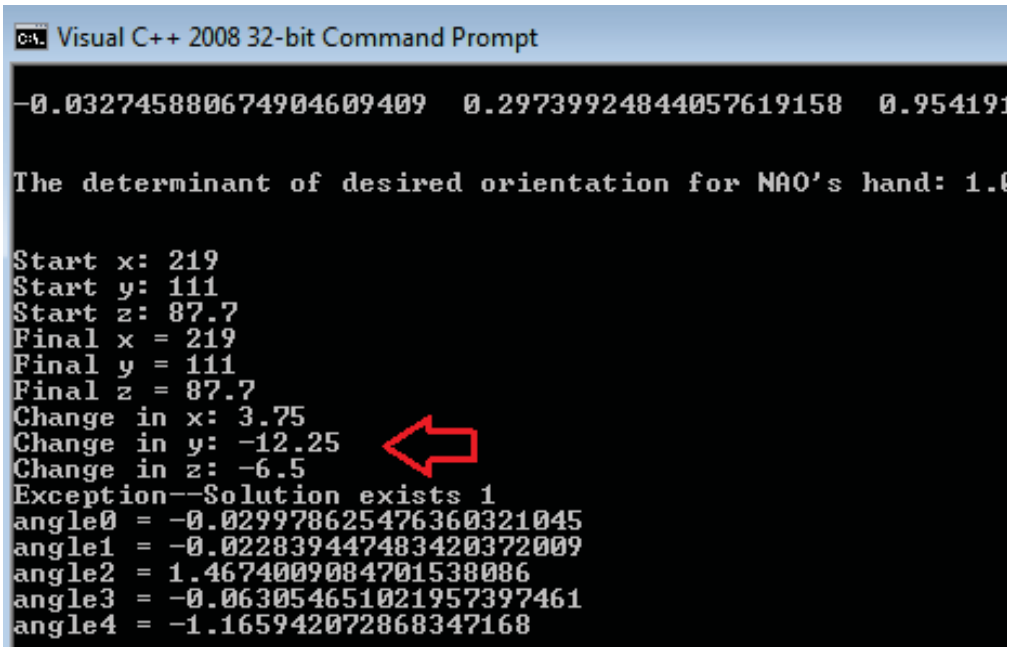

Figure 3.19: **Changes in position:** The output here shows the small, but significant, changes in position which are registered by the STEM-tracker when trying to keep the tracker's centre of rotation perfectly still. The sensitivity of the tracker, and the inaccuracy of the telemanipulator, makes the combination of position and orientation very difficult for the robot arm to follow.

3.8 Problems with STEM

Some parts of this section are based on section 3.3 of my project thesis (Evjemo, 2016).

As mentioned in section 1.4.1, the tracking system used in this project is a BETA-version of the 5-controller system Sixense STEM. When working with this system during the summer and fall of 2015, I encountered some quite time-consuming problems, and many of these issues were still there while working on the Master thesis. These issues have been a continuous problem, but they have not been given much attention in other parts of this thesis. This is partly because these problems were less prominent in this project compared to previous work because there were little or no direct robot control, and partly because focusing on these issues continuously, would draw attention away from the general progress of the project.

Some of the issues with the STEM-system were due to severe physical weaknesses in the STEM-system. These problems will hopefully be improved or removed before STEM is released to the

commercial market.

3.8.1 WiFi sensitivity

The Sixense STEM system is quite sensitive to WiFi-connections, so using it in the lab was difficult during the project thesis. The trackers would more often than not refuse to connect to the base station, which was indicated by that the LED lights in the front kept rotating. Sometimes only a few of the trackers would connect, and it seemed completely random which trackers would connect and not. Other times, none of the trackers would connect at all. The solutions suggested in the documentation (Diaz, 2015) would not help. For more details, see my project thesis section 3.3.1 (Evjemo, 2016).

In the summer project, these issues eventually improved when the robot was connected to the local network with a wire instead of using WiFi. This was challenging because this made it necessary to make sure that the NAO robot did not trip on any of the wires connecting the robot to the router, as we can see in figure 3.20. Even though the problems improved somewhat, the trackers would still disconnect quite frequently, which made the system very difficult to work with.

The difficulties with WiFi-sensitivity was not that big of an issue when working with the Master thesis as it had been in the previous projects. This was mainly because most of the work was done outside of the lab, away from both the physical robot and routers. In e-mail correspondence with the Sixense STEM-developers in the fall of 2015, their advice was to change the frequency of the routers nearby, because routers running on 2.4 GHz where known to cause interference with the system. The full e-mail correspondence is included in appendix C. This was not tested because the problems became less prominent with the latest firmware updates. It is still necessary to mention it, because if the complete system was to be tested on the physical robot, the WiFi weakness would be a limitation in the sense that the robot would have to be connected to the Internet through a wire, as shown in figure 3.20. It is not a crippling obstacle, but still quite limiting.

3.8.2 Metal sensitivity

The Sixense STEM system is incredibly sensitive to metal. If the controllers and packs were used too close to objects with metal surfaces, like chairs, desks or screens, the tracking failed completely. This was evident both when following the trackers with the STEM-system's API, and with Øye's Sixense application, which was described in section 3.2.2. For example, when a tracker was actually standing still, close to a metal surface, the tracking data would indicate that it bounced around, or that it was positioned in a different location.

If the entire base station was placed too close to metal, it would for the most part fail to connect with the computer at all, and trackers that managed to connect to the base station would disconnect again within seconds. The metal sensitivity was the reason why the whole lab-setup in the summer project was eventually changed to be made out of cardboard, as shown in section 3.20.

When the Sixense STEM-developers were contacted about the problem in the fall of 2015, they explained that the STEM-system was sensitive to metal because of the magnetic tracking. They explained that this was common for other tracking solutions as well, and that it was necessary to make environmental considerations when using it. They recommended to avoid any metal within a radius of 1 meter from the base station. The full e-mail correspondence is included in appendix C.

As mentioned in section 3.8.2, most of the testing was done outside of the lab while working on this Master thesis, so the metal sensitivity were less of an issue than it had been in previous projects. Still, if the base station was placed to close to the computer screen, or the trackers came too close to my office chair, they would disconnect or give out tracking data that made no sense. This was not too difficult to avoid, but it made it clear that combining the STEM-system with other kinds of hardware consisting of metal will be problematic.

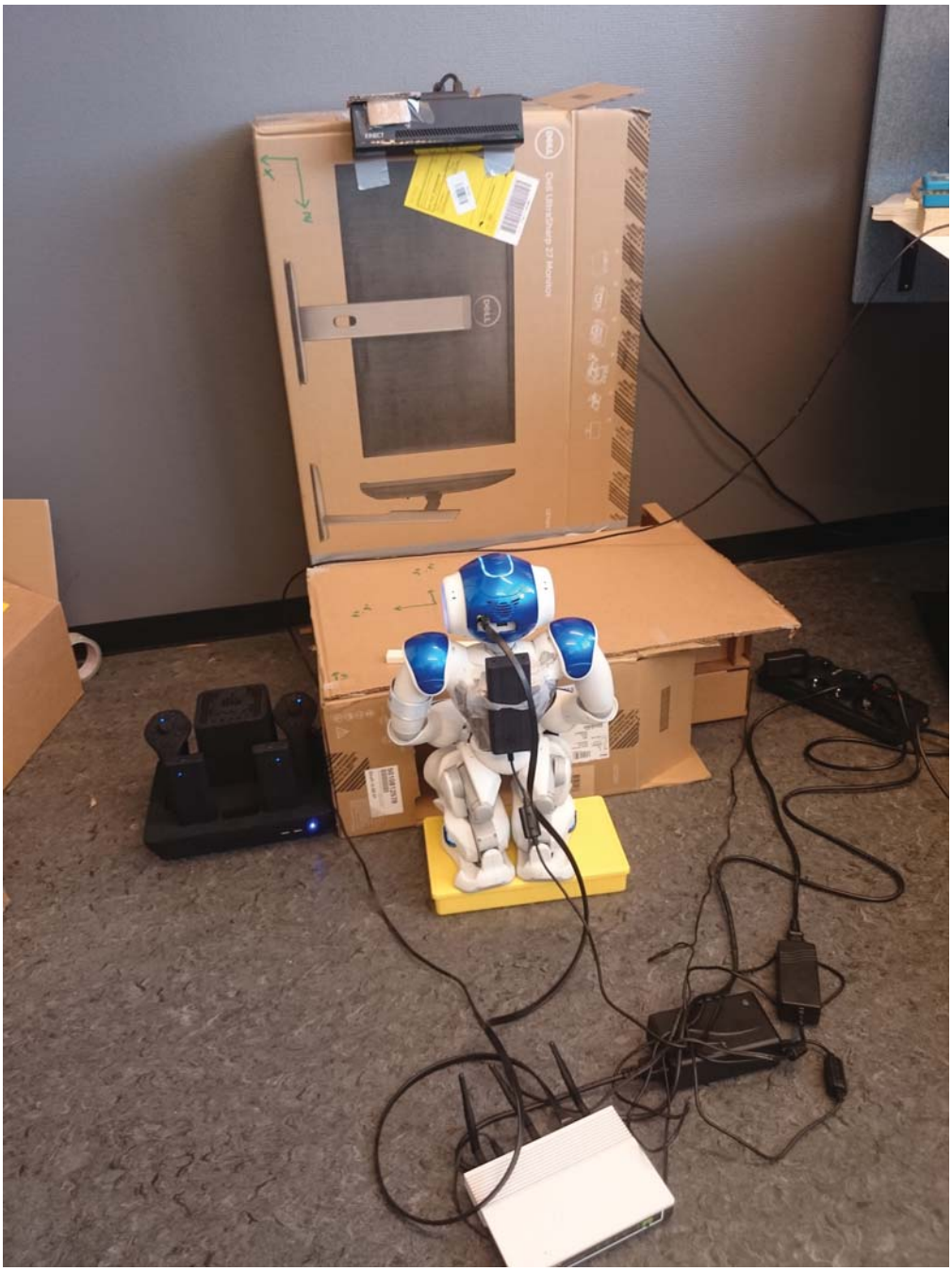

Figure 3.20: **Summer lab:** Here is the complete system from the summer project. The rig is made of cardboard because of Sixense STEM's metal sensitivity, and the robot is connected to the Internet through a cable because of the WiFi-sensitivity.

3.8.3 Lifeless trackers

A new problem that arose at the late stages of the work on my project assignment, was that one or several trackers would seem completely lifeless. The LED lights would stay dark, and it would

refuse to connect to the base station.

While working with the Master thesis, there have been very few periods of time where all five trackers have been functional at the same time. In the early stages of my work, four out of five trackers worked properly, while the middle *pack* was lifeless (see section 1.4.1). It did not help to restart the system or the computer. In fact, the tracker was "dead" for several weeks.

Then, suddenly, all five trackers worked for a short period of time. There had not been any updates of firmware, and I had done nothing differently. And only a short while after, the two other packs stopped working. So throughout the period that I have been working on this thesis, one or several trackers have always been lifeless for no apparent reason, before suddenly working again when I least expected it.

Luckily, my system could be tested using only one tracker at a time, so this was not a very big issue for the testing. It only meant that I occationally had to change which of the text files the IK-solver colelcted the data from, see section 3.2.2. However, as will be discussed further in section 5.2.4, a suggestion for further work would be to place the robot, the STEM-system, and other necessary hardware in the same coordinate frame using all five trackers. In that case, the problem with lifeless trackers would be very inconvenient, as I will get back to in section 4.3.

3.8.4 Issues that have been fixed or improved by firmware updates

Some of the STEM-problems that I experienced in previous projects, have been improved or fixed through firmware updates while I have been working with the Master thesis.

No more confusion related to axis directions

One of the problems I had experienced while working with the STEM-system that could potentially do most harm, was that STEM would sometimes register both directions of an axis to be positive. The values would rise in one direction, and approach zero when closing in on the base station's center, as it should. But when moving the tracker in the opposite direction, the position value for the given axis would start decreasing as normal, before suddenly becoming

positive again. This is explained further in my project thesis, section 3.3.4 (Evjemo, 2016). This error caused large problems when trying to control NAO, because it can lead to massive jumps in the given coordinates. Fortunately, the latest firmware updates seems to have gotten rid of this problem completely, and it has not been an issue while working on the Master thesis.

Fewer random disconnects

When working with the STEM-system in this project, it was also less common that the trackers would randomly disconnect. While working on the project thesis in the fall of 2015, it almost became a daily routine to have the trackers or the base station disconnect, and refuse to connect again until the computer and the entire system was rebooted. This would happen in the same environment as where I have been working on the Master's thesis. Fortunately, the issue has been improved tremendously by the latest firmware updates. Trackers would still disconnect at random if left outside of their docking stations, sometimes after only a minute, other times after a long period of time. However, they would almost always connect again without any problems as soon as they were placed back in their docking stations. This is, of course, not counting the semi-permanently lifeless tracker-problem described in section 3.8.3.

Chapter 4

Discussion

The goal of this project was to create a system allowing telemanipulation of the NAO robot's arm with minimal latency. It turns out that using an analytic IK-solver to make a robot arm with a limited number of DOFs follow the motions of a human arm, is much easier said than done. Even though this project did not result in a working system, it has lead to some important insight into both robot control in general, and to how a system like this could be improved further.

4.1 Combining STEM-control with analytic IK-solver

Throughout chapter 3, it has been explained how the system was tested in order to achieve valid solutions from the analytic IK-solver. The problem was not type errors when setting the values for the transformation matrix in the C++ script, nor was it basic mathematical mistakes made in the implementation. And the problem was not that the rotation matrix was not orthogonal, because the IK-solver could handle small inconsistencies like this. In section 3.7, it finally became clear that the system *did* work, but only under very specific circumstances. This allowed me to conclude that my implementation was correct, and that the challenge seemed to be with the combination of Cartesian position and orientation registered by the STEM-tracker. Therefore, the problem had to be related to limitations of the analytic IK-solver.

The analytic IK-solver has been thoroughly tested, both by myself and by others. It was also possible to get a valid solution from it, if only for a very specific rotational movement of the

STEM-tracker described in section 3.7. It was therefore necessary to try to understand why it did not return a valid solution for *all* movements. To understand this, I had to go back and consider what limitations the IK-solver had that made it consider almost all of the combinations of position and rotation that the STEM-tracker sent it non-feasible.

The analytic IK-solver was of course created specifically for the NAO robot's effectors. And, as explained in section 2.2.4, the arm effectors of the NAO robot does not have the same number of DOFs as a normal, human arm. Even though the mobility of the shoulder and elbow joint was quite similar to a human arm, the robot's wrist joint was very limited in comparison: The robot's wrist joint only has 1 DOF, allowing nothing but a simple rotation about the underarm. This meant that while a human is able to change the orientation of its hand quite freely from the position of the full arm, the robot-hand was locked to the orientation of its underarm.

As explained in section 2.2.4, the NAO robot's arm effectors seemed to have quite a large workspace. However, this was based only on which points in Cartesian space the robot arm could *reach*, not on what orientation the arm effector could have in each given point. The general workspace of a manipulator can be divided into the *reachable workspace* and the *dexterous workspace*. The first is, as the name indicates, all the points the end effector is able to reach. The dexterous workspace, on the other hand, is the subset of points that the manipulator can reach with an arbitrary orientation (Spong, Hutchinson, and Vidyasagar, 2006, p. 6). Because NAO's arm effectors have only 5 DOFs, the dexterous workspace is very limited.

When considering this, one can understand that it is not possible to send any combination of position and orientation from the STEM-tracker to the IK-solver, and expect a valid solution. For each Cartesian position the end effector is able to reach, there is an almost endless number of orientations that is out of bounds, and only a small subset of orientations that is realizable. This feels more intuitive if tested on your own, human arm: If you try "locking" your wrist so that you only allow for rotation about the underarm, the number of orientations you can realize with your hand in any, fixed position in Cartesian space is very limited.

Early on in the project, Kofinas had informed me that the IK-solver would *not* return the "nearest possible solution" if given a non-feasible combination of position and orientation (Appendix B). At the time this did not seem too problematic, only indicating that there would have to be some kind of exception-handling for such occasions. I now understand that this was not only a limitation in the IK-solver Kofinas had designed, but on the robot itself. This did in fact mean that the robot would have no way of following most of the movements registered by the STEM-tracker.

4.2 Cartesian vs. joint control

Because I was not able to create a functioning system for joint control of the NAO robot in this project, it is not possible to give an exact estimate of how effective joint control is compared to Cartesian control. However, the experiences from testing the two methods during both this project and earlier work, has shown that joint control seems more stable. In addition, the Cartesian methods are based on the robot's own numeric IK-solver. As mentioned in section 2.5.1, an analytic IK-solver is more effective than a numeric one. This is because an analytic approach is a direct calculation, while a numeric approach is an optimization problem. Therefore, it is almost certain that a system based on joint control will be most effective.

As explained in the introduction, no testing was done on the physical NAO robot in this project. However, some testing was done on the virtual robot in Choreographe. I had little experience with the virtual robot in Choreographe before this project, so it was difficult to tell whether or not it behaved enough like the real robot for the results to be valid. However, when running old code from the project thesis (Evjemo, 2016) on the virtual robot, the behaviour was exactly like the behaviour of the real robot. Even the scripts that had made the physical robot shake or become unstable, gave the same behaviour. Therefore, I choose to trust that this is the case the other way around as well, and that the results seen on the virtual robot in this project are transferable to the physical robot.

The Cartesian control methods seemed to have difficulties with keeping the robot stable if the desired position and orientation of the end effector were too far from the current position. The

robot arm would often be shaking, never coming to a full stop. The same would sometimes be the case for smaller movements: Instead of just moving the end effector to its new position and orientation, the robot arm would keep moving slightly, never really coming to a halt. Joint control methods, on the other hand, seemed to have no problem with big changes in angle values. Once the end effector had reached its new position, it would also stay completely still until being told to move to a new point.

To understand the robot's behaviour for the two control methods, it is necessary to go back to the basics of forward and inverse kinematics, which were explained in section 2.5.1. First of all, a inverse kinematics problem can have one solution, multiple solutions, or no solution at all. A forward kinematics problem on the other hand, will always have one and only one solution. Secondly, a numeric IK-solver like the one used by the NAO robot for Cartesian control, uses an iterative method to come to an optimized solution to the problem. This means that when the robot is told to move its effector to a given position and orientation, the numeric IK-solver might come up with a slightly different solution each time.

After all, the robot's controller will never give "no solution" as output, like the analytic IK-solver. Instead, it will try to find a feasible solution that is close to the desired position and orientation. For joint control, however, the robot's controller only has to set each of the joint angles to the given value. the end effector will therefore always end up exactly in the one position and orientation that is the solution to the given forward kinematics problem. This could explain why joint control methods are a more stable way of controlling the robot.

4.3 Evaluating the STEM system's practical use

In previous work there has been a lot of difficulties with the STEM-system, and the work on this master's thesis was no exception. There would still be random disconnects of the trackers, times when the base station would not connect with the computer, and of course problems related to metal and WiFi sensitivity. However, the problems were considerably less dominant than before. It was clear that the newest firmware updates had had some effect, which made it possible

to use the system for testing and control throughout the project. The physical problems related to metal in the system's surrounding had not improved, but they were fairly easy to avoid in this case. However, in larger projects involving a lot of different hardware, this would make the system quite useless. Hopefully, this will be improved before the commercial release of the tracking system.

One major issue in earlier work had been that the sign of the position coordinates would sometimes change. These random changes could potentially send massive jumps in position data to the robot's effector, which could be harmful to the physical robot. This problem was fixed now, which made testing on the physical robot a lot less risky - if the project had ever gotten that far. The fact that the stability had improved, with fewer random disconnects, also indicated that the work on making the system more user friendly was going in the right direction. As mentioned in section 3.8.1, the firmware updates had also made the system less sensitive to WiFi interference.

The main concern at this point was the new problem that had arisen during the work on the project thesis: the "permanently" dead trackers. Most of the other problems, like the base station not connecting to the computer, or the trackers losing contact with the base station, could be fixed by simply re-setting the computer and the STEM-system. This was of course somewhat time-consuming, but it was a quick-fix to most of the problems. The fact that some of the trackers would now stay dead for weeks at a time, makes the system less suitable for use in larger projects, where this kind of delay could be critical.

After seeing how the system has improved over time, it seems very probable that it will be suitable use in this kind of projects at a later stage. When it works as it should, the tracking is very accurate. The controllers also have many functionalities like joysticks and buttons, which makes the system even more attractive when working on larger projects. As mentioned in section 3.2.2, implementing opening/closing functionality for NAO's hands, or changing modes during testing, could be done relatively easily using the functionalities which the STEM-system provides in addition to tracking. Still, the STEM-system has some weaknesses that must to be improved or removed completely before it is ready to be used in projects that demands a fully functioning

tracking system.

4.4 More general system structure decisions

In this section some of the general choices for the system structure will be discussed to see if they were the wisest choice or not. This could help create an easier and more effective system in future work.

4.4.1 The decision to use the original C++-files

The decision to stick with the original C++ scripts for the IK-solver seems like a good choice. Even after working with the scripts for several months, I do not understand how they work down to a point where it would be an easy task to recreate the IK-solver in a different programming language. However, the method of running the executable file in LabVIEW described in section 3.5.5, would occasionally make LabVIEW crash. Therefore, it might be an idea to try and finish the MatLab-scripts that Kofinas has made available through GitHub. However, at this point, using the C++ scripts seems like a good enough solution, as long as LabVIEW crashing does not become a bigger problem.

4.4.2 Collecting data from the STEM-system

Out of the three choices for collecting tracker data discussed in section 2.4, this project went with the approach of using Øye's program to write data for each of the STEM-trackers to a designated text file. This system seemed to work fairly well, and was highly effective. However, it would have been possible to use DLL-files in LabVIEW as well. This solution was discarded because I thought at the time that using DLL-files would make it necessary for LabVIEW to get data directly from the IK-solver, which would require some way of direct communication between the C++ scripts and LabVIEW. As explained in section 3.2.1, this would be easier said than done, because LabVIEW is not compatible with C++. However, the work on the project made it clear that communication between LabVIEW and the IK-solver could have been solved in a similar way as for the current system if I had been using DLL-files.

A solution with DLL-files would require some way of transporting the information between the programs, like writing to and reading from file, which was used in the current system. Using TCP-connections might also be an alternative. Because using Øye's program makes it possible to get the tracker data directly to text files, which can be read by the IK-solver, this still seems like the easiest solution for the system at this point. But if restructuring the system had made it easier to use DLL-files, this solution would also be possible. Using DLL-files combined with Python could, for the same reasons, be used if more of the system was to be implemented directly in Python, but would also make it necessary to transport the data between the programs using text files or network connections. In other words: Discarding the DLL-solutions is still the wisest choice for this system structure. But in future work, the decision for how to collect data from the STEM-trackers must be based on what is more effective for the given system structure, because all three approaches can be used.

When working on the project thesis during the fall of 2015, the rotation of the STEM-trackers were collected in the form of a rotation matrix. Both the quaternions and the rotation matrix can be collected directly from the trackers. For more details on this, see section 2.3 of my project report (Evjemo, 2016). Øye's program collects the rotation in the form of quaternions instead of the rotation matrix. Because the IK-solver needed the rotation in the form of a transformation matrix, it would have been a better choice to get the rotation directly in the form of a rotation matrix.

To get the rotation matrix directly would save the system the work of transforming the quaternions to rotation matrices. When using DLL-files in earlier project, it was possible to choose what data to extract from the STEM-tracker. However, I was not able to make changes in Øye's system. In retrospect, it would have been best to ask Øye to help edit his program to return the rotation matrix from the start. This was not done because at the time, I thought it would be more trouble than it was worth. I did not consider how much work it might be to convert the quaternions to rotation matrices. As explained in section 3.4.2, it took time to determine the order of the quaternions. It was also necessary to make sure that they were normalized, and they

had to be converted to a rotation matrix.

Chapter 5

Conclusion and further work

5.1 Conclusion

In the end, I was not able to create a system that would allow for low-latency control of the arm effectors of a NAO robot. The physical limitations of the robot arm makes it impossible for the end effector to follow the exact movements of a tracker that moves like a human hand. If the robot is to follow the STEM-tracker, there must be some kind of compromise between following the position and following the orientation. It is therefore impossible to use only a straightforward analytic IK-solver to perform joint control of the arm effectors, as this will only return solutions for the very few, feasible combinations of position and orientation. How the system might be changed to make this possible, must be left for further work.

As for the STEM-system, the conclusion has to be, once again, that it is currently too unstable to be used in this kind of project work, even though it is very accurate and easy to use, and has improved a lot over the last year. Dealing with the bugs takes up too much time, and sometimes puts a halt to the testing altogether. Hopefully, future firmware updates will help make this BETA-version more stable. When the commercial version of the system is released to the public, it will hopefully have reached its full potential.

Based on the limited testing done in this project and experiences from previous work, it seems pretty clear that joint control is the most effective way of controlling the arm of the NAO robot.

It is both a faster and a more stable control method than Cartesian control, and seems like the most promising way to achieve low-latency telemanipulation of the robot arm. To create a system with joint control based on hand-held motion trackers, it could be possible to use an external IK-solver to do the maths, and then send the angles directly to the robot. This project has shown that this would require quite a lot of work. Another approach could be full-arm motion tracking, where the angles between the different joints in the human arm are measured directly. Because the STEM-system cannot be considered a suitable tracking tool at this point, one should keep this approach in mind when looking for alternative tracking solutions.

5.2 Suggestions for further work

This project did not succeed in creating a low latency system for telemanipulation of the NAO robot. But the new-found understanding of what did *not* work,brings with it ideas and suggestions for what could be done differently in future projects. In this section I will present my suggestions for future work, and my ideas for fixing the system, that there were unfortunately not enough time to test.

5.2.1 Modifying joint control with analytic IK-solver

The conclusion of the work in this project is that it is not possible to make the end effector of a robot arm with only 5 DOFs follow the movements of a human hand both in position and orientation. The human arm has 2 additional DOFs, which makes our dexterous workspace much larger than the dexterous workspace of the robot arm. In this section, some possible solutions to this problem will be presented. None of these ideas have been tested in this project, but are suggestions for further work.

Mapping the robot's workspace with orientation constraints

One way of creating a system using joint control methods together with a hand held tracker similar to STEM, could be to somehow map all of the feasible positions and orientations of the arm effector. If it was possible to create a table with all of these values, and then look up a given value each time, it could be possible to narrow it down to a feasible solution. This would almost

certainly demand the use of an effective search algorithm.

If the current position and orientation did not have a solution, then one could try iterating over "nearby" points in position and orientation to find a feasible point close by. This could for example be done by changing the position one millimeter at a time in one direction or the other, or changing the orientation by one degree around one of the axes. This would demand some kind of weighting between position and orientation to determine what solution the system should be looking for: Is it most important to keep the position, and alter the rotation, or is the orientation the most important thing? Should either position or rotation stay fixed, or are small changes in both parameters preferable? These are questions that need to be answered if this kind of solution is to be tested in a larger system.

Adding more degrees of freedom

Another way to solve the problem with an analytic IK-solver and a hand-held motion tracker, would be to introduce more DOFs. This could for example be done by allowing the robot to move forward and/or sideways. This can easily be implemented using methods from the SDK. The desired position would still be given in Cartesian coordinates relative to the robot's torso. This means that when the robot moved, the movement in the "global frame" would have to be added or subtracted from the Cartesian coordinates describing the position in the robot's *Torso* coordinates. This way, if given a position and orientation that were not feasible, the robot could move along its x- and y-axis until it hopefully ended up somewhere that made the desired position and orientation a feasible point.

This approach would be quite straight-forward to simulate, because it would only be necessary to iterate over the x- and y- values of the desired position to see if changing the value makes it easier to find a valid solution. If this solution was to be put to practical use, however, it might be more realistic to place the robot on a controllable, moving surface than to have it walk around. In order to stay in balance, the NAO robot's walk is very slow, and would therefore be a big interruption in a telemanipulation system that requires low latency. This approach of introducing more DOFs is still worth mentioning if the idea is to create a functioning system based on an

analytic IK-solver.

5.2.2 Tracking the joint angles directly

Parts of this section is based on section 5.1 in my project report (Evjemo, 2016).

Another way to use joint control would be to try and avoid an IK-solver altogether, and instead measure the angles between the joints directly. This could be done with the STEM-system, should its functionality improve. By fastening one tracker to the overarm, one to the underarm, and holding one tracker in the hand, all the information needed to send direct angles to the robot would be available. It would also be necessary to have one tracker fastened to the torso of the controller, and for the controller to keep his back straight. This way the system would have a point of reference for the straight position NAO's torso is kept in.

With the current STEM-system, with its five trackers, this would limit the telemanipulation to only one arm at a time. Still, it could be used for testing that would help determine how much faster joint control is compared to Cartesian control. Like mentioned in section 1.4.1, one of the other tracking systems that were considered for this project is based on direct joint control (Advanced Realtime Tracking, 2016). The ART Hybrid Suit has trackers that are fastened to all the major joints of the human body, measuring the change in position and orientation for each of them. This would, on a larger scale, perhaps allow telemanipulation of the full body of the NAO robot, but as for now, the focus is on producing a way of successfully controlling the arms.. It is important to note that it is necessary to limit the DOFs of the teleoperator's arm when using STEM, or any other hardware to measure the joint angles directly. It would therefore be necessary to somehow lock the wrist to stay parallel to the underarm. otherwise, the orientation of the hand of the teleoperator would make no sense when comparing it to the DOFs of the robot arm.

5.2.3 DCM-programming

Parts of this section is based on section 5.3 in my project thesis (Evjemo, 2016).

It is possible that using the DCM is the fastest way to control the robot, but it might also be extremely difficult, and possibly disastrous. The DCM is a link between the "upper level" software and the "lower level" software of the NAO robot (DCM - Introduction, 2012). The DCM is in charge of communication with all sensors, actuators and boards in the robot. The ALMotion module uses the DCM to send commands to the actuators. If it was possible to go around the ALMotion module, and communicate directly with the DCM, this would certainly be more effective.

However, Aldebaran strongly warns against meddling with the DCM unless you know *exactly* what you are doing. It therefore seems logical to try to develop a functioning system using the approaches described in section 5.2.1 and 5.2.2 before attempting to go over to DCM-control. If it is possible to find a way of controlling the robot using joint control methods from ALMotion which is effective enough for the telemanipulation to feel natural, it might be unnecessary to use DCM-control, even if it could result in a system with even lower latency.

5.2.4 Global frame and scaling of movements

Parts of this section is based on section 5.4 in my project thesis (Evjemo, 2016).

In this project, the system is based on the robot working relative to its *Torso* frame, meaning that it interprets all coordinates to be relative to its own torso's position. As explained in section r2.2.2, it can theoretically also use coordinates relative to its starting position, but this is very inaccurate in real-life. For further work, sensor fusion algorithms might be something to consider. This is a technique that allows a system to be robust by combining the data from several sensors to make sense of input that has been disturbed by the surroundings (Mathas, 2012).

Sensor fusion could be part of a system where all the equipment is placed in one global frame.

This could be achieved by using several tracking units from the system STEM-system - if it starts working properly, of course. By fastening one tracker directly on the robot, and on any other hardware included in the system, it would be possible to find the exact position of all the different hardware in the same, global frame. It would then be possible to get the robot to move its effectors, mainly its arms, towards a given point in this global coordinate system. This might even allow the robot to move around in the room, because all of the movements would be tracked accurately. By converting the tracked coordinates to be relative to the robot's torso, as has been done to some extent in this thesis, the robot could move its arm to a given point in the global system.

In further work with a functional system, it would also be a good idea to scale the movements of the telemanipulator down to the robot's size, to properly test if the movements feel and look natural when the robot performs them. This way, larger and more realistic movements can be tested without moving the tracker outside of the robot's quite limited workspace. In the long run, the goal is to make the robot follow a movement as closely as possible relative to its own body, which makes scaling down the movements essential. In a functional system where the movements are tested on a simulation of the robot, one should also try testing intentionally introducing time-delay in order to evaluate how big the system latency can become before the telemanipulation feels unnatural.

Appendix A

E-mail correspondence with Halit Bener Suay

Linn Danielsen Evjemo <linndevjemo@gmail.com> Mon, Sep 21, 2015 at 1:17 PM

To: benersuay@wpi.edu

Hello

name is Linn Danielsen Evjemo, and I am a 5th year student of cybernetics and robotics at the Norwegian University of Science and Technology. I am currently working on a project assignment where we are trying to control the arms of a NAO robot using a new, and very accurate motion tracking system callen Sixense STEM: *http://sixense.com/wireless*

One of the main problems of our project is to reduce latency enouch so that the control of NAO's arms feel natural for the person controlling him. I have looked into your project with telemanipulation of NAO using a 3D camera, and I see that you have very little delay. We have so far tried controlling everything though a program called Labview: We have written the code for controlling NAO in Python, which the communicates with Labview using a TCP-connection. Labview is connected to the motion tracking system, and sends coordinates for where NAO should move his arms using another TCPconnection, and the setPositions()-method.

I wondered if you could share how you communicate with/send coordinates to NAO in your system? I see that the github-link does not work anymore. if would be very helpful to get some input on a better

way to communicate with the robot than what we are using now. I really hope that you can help me.

Regards

Linn Danielsen Evjemo

5th year Cybernetics and robotcs

Norwegian University of Science and Technology

`Suay, Halit Bener <benersuay@wpi.edu> Mon, Sep 21, 2015 at 3:25 PM`

`To:  Linn Danielsen Evjemo <linndevjemo@gmail.com>`

Dear Linn,

Thank you for getting in touch with me. I would like to clarify a few things,

1. I've moved the code here /fair warning: super messy code with very little testing and it used to work with a very old version of ROS): https://github.com/benersuay/nao_rail/tree/fuerte-devel/nao_openni
2. Since I was using ROS, inherently the communication is over TCP/IP and yes, there is a noticeable amount of delay between the user's motions and the robot's imitation.
3. When I was controlling the robot, my movements are as slow as possible in order to make up for the delay.

I've had a few other people complaining about this problem (from different schools), and there may be a few things to do to mitigate the issue:

- If it is possible, try using the wired connection and see if that makes any difference. Nao's hardware is very limited in general (though I don't know which version you're using, the latest Naos may be better than the ones I have used).
- If possible, try to timestamp all the outputs of your system blocks and see where the biggest delay is occurring (you mentioned Labview, and I have no experience with it, however I know that it's being used for signal processing in general. For example if your system is roughly designed as:

$[cameras] - - > [PC : labviewDAQmethod] - - > [labviewfiltermethod] - - >$

$[otherlabviewfunctions] - - > [labviewtcpsocketoutput] - - > [nao:pythonsocketinput]$

Then try to measure how many msec every block takes to process each call. Of course, it all adds up and what you observe as a user is the total delay, however if one of the blocks is takings 10 times more than the others, you may re-think / re-design that block and improve the throughput of your system.

If I had any experience in LabView I would offer to read your code and help you with that but unfortunately I know almost nothing about it. Please let me know if there's anything more I can help you with. I check my emails very frequently.

Best,

Ben

Appendix B

E-mail correspondence with Nikolaos Kofinas

Linn Danielsen Evjemo <linndevjemo@gmail.com> Wed, Mar 9, 2016 at 2:08 PM

To: nikofinas@gmail.com

Hello

My name is Linn Danielsen Evjemo. I am a 5th year student in cybernetics and robotics at the Norwegian University of Science and Technology. I am currently writing my master's thesis, which is partly based on the inverse kinematics that you developed for the NAO robot in you own thesis. My goal is to use the inverse kinematics that you developed in order to control the arms of a NAO robot with two hand-held motion trackers. By transforming the registered position and orientation of the hand-held trackers, I hope to send the joint angles for the arm effector-chains directly to NAO, and thereby control NAO in close to real-time, at least a lot quicker than when using NAO's own IK-solver.

I would really appreciate it if you had time to answer a couple of quick questions regarding the c++ files that you have made available on GitHub. I just want to make sure that I have understood the code properly, as I still find it a bit difficult to read and interpret other people's code.

- Is the code that is currently available on GitHub "finished", in the sense that it can be used directly as it is? I ask because of the commented ares, like the "Under construction"part in main.cpp.
- I wish to send in a given position and orientation, and then get the angles in the arm effector chain to

make the hand end up there. Can your code, as it is now, be used for this?
- When there are several possible solutions, what method do you recommend for finding the "best" solution? Comparing the solution with the previous movements, to see what is "closest"? Or send all solutions through the forward kinematics, and compare the results?

If you have the time to answer me, I would greatly appreciate it. And if you are too busy, I completely understand, but it would be great if you could reply to my email and let me know.

Sincerely
Linn Danielsen Evjemo
5th year student in cybernetics and robotcis
Norwegian University of Science and Technology

`Nikos Kofinas <nikofinas@gmail.com> Wed, Mar 16, 2016 at 10:19 PM`
`To: Linn Danielsen Evjemo <linndevjemo@gmail.com>`

Dear Linn,
First of all I am really sorry for the late response.

- Is the code that is currently available on GitHub "finished", in the sense that it can be used directly as it is? I ask because of the commented ares, like the "Under construction"part in main.cpp.
The code is finished and can be used as is. I don't see any "under construction" parts in the main function, can you please give me a pointer to them?
- I wish to send in a given position and orientation, and then get the angles in the arm effector chain to make the hand end up there. Can your code, as it is now, be used for this?
Yes. But you need to give a valid set of parameters else you will not get any solution (e.g. you will not get the closest feasible solution).
- When there are several possible solutions, what method do you recommend for finding the "best" solution?Comparing the solution with the previous movements, to see what is "closest"? Or send all solutions through the forward kinematics, and compare the results?
It is unlikely that you will have multiple solutions. If you get multiple solutions then just use the one that

it is closest to the current robot pose.

Best regards,
Nikolaos Kofinas

Linn Danielsen Evjemo <linndevjemo@gmail.com> Thu, Mar 17, 2016 at 1:52 PM
To: Nikos Kofinas <nikofinas@gmail.com>

Hello
Thank you so much for your reply! It seems that the first time I downloaded the files, I somehow managed to download only the oldest versions. In the updated files there are no "under construction" parts, sorry about that :) I only have one more question: Is the MatLabsolution as complete as the C++ solution? I am not as familiar with MatLab as i am with C++, so I do not understand if all of the inverse kinematics are there? Sorry if that is a silly question, but I guess it is better do ask anyway. If the MatLabsolution is complete, it would be easier to use in my project than C++, because I am using LabVIEW, which is not compatible with C++.

Best regards
Linn Danielsen Evjemo

Nikos Kofinas <nikofinas@gmail.com> Fri, Mar 18, 2016 at 6:48 AM
To: Linn Danielsen Evjemo <linndevjemo@gmail.com>

Hey,
The matlab solution is incomplete. I lost the actual correct files and I never reimplemented it.

Cheers,
Nikos

Appendix C

E-mail correspondence with STEM-developer

Linn Danielsen Evjemo <linndevjemo@gmail.com> Mon, Aug 31, 2015 at 5:56 AM

To: Steve Braman <steve.b@sixense.com>

Cc: John Reidar Mathiassen <John.Reidar.Mathiassen@sintef.no>, Elling Ruud Øye <Elling.Ruud.Oye@sintef.no>

Hello Steve,

The STEM system has worked well in general, and is very accurate. However, we have had some quite time consuming difficulties, mostly because it took us a while to realize exactly what the problems were.

It seems like our STEM system is easily disturbed by WiFi. We tried using the STEM system in the same room as a NAO robot from Aldebaran Robotics, which was connected to a local network via WiFi. The controllers and packs would more often than not refuse to connect to the base station. Sometimes only a few of the controllers and packs would connect, other times none of them. When the controllers and packs connected, it was also just a matter of time before they all disconnected again, and returned to the state with rotating lights. When we eventually connected the Nao robot to the local network with a wire instead, the system worked much better.

In addition, the STEM system is very(!) sensitive to metal. When we tried using the controllers and packs too close to metal (chairs, screens etc), the tracked movement in the Sixense Test program showed that

the motion tracking failed completely. When we placed the base station too close to metal, it often failed to connect with the computer at all. Controllers and packs that managed to connect to the base station would disconnect again within seconds, and go to the state where the leds kept flashing. This happened if we placed the base station as far as 2 meters from a small metal frame (about 1.5x0.5 m2) which was part of our lab. When we removed the metal frame, the STEM system worked fine. Both the WiFi problem and the metal sensitivity is very limiting, and makes it problematic to integrate Sixense STEM in a larger project that includes different kinds of hardware. At this point we are able to work around these problems, but that might become more challenging las our project progresses. Has anybody else had similar problems?

Sincerely
Linn Danielsen Evjemo

Steve Braman <steve.b@sixense.com> Wed, Sep 9, 2015 at 3:02 AM
To: Linn Danielsen Evjemo <linndevjemo@gmail.com>
Cc: John Reidar Mathiassen <John.Reidar.Mathiassen@sintef.no>, Elling Ruud Øye <Elling.Ruud.Oye@sintef.no>

Hey all,
I apologize for the delayed response. I am happy to hear that the system performs well and hope I can help address the issues you are experiencing, As far as the Wifi issue, yes some routers that are running 2.4GHz wifi have been known to cause interference. Can you switch to running 5GHz wifi?

Similarly to other tracking solutions, there are environmental considerations that need to be made for magnetic tracking. The environment should be as free of metal as much as possible especially near the base station and between the base station and user. We recommend keeping metal out of the magnetic field which would be 1 meter put from the base station. If you are still experiencing these issues or have further questions please let me know.

Thanks,

Steve Braman

Designer/Developer Support Sixense Studios

Bibliography

Aldebaran Robotics, 2012a. Cartesian control. Accessed: 2016-06-06.
URL http://doc.aldebaran.com/1-14/naoqi/motion/control-cartesian.html

Aldebaran Robotics, 2012b. Dcm - introduction. Accessed: 2016-05-21.
URL http://doc.aldebaran.com/1-14/naoqi/sensors/dcm/introduction.html

Aldebaran Robotics, 2012c. Nao technical guide: H25 - joints - v3.2. Accessed: 2016-03-31.
URL http://doc.aldebaran.com/2-1/family/nao_h25/joints_h25_v32.html#h25-joints-v32

Aldebaran Robotics, 2012d. Naoqi framework. Accessed: 2015-11-20.
URL http://doc.aldebaran.com/1-14/dev/naoqi/index.html

Aldebaran Robotics, 2015a. Cartesian control 2-1. Accessed 2015-11-15.
URL http://doc.aldebaran.com/2-1/naoqi/motion/control-cartesian.html

Aldebaran Robotics, 2015b. Cartesian control api. Accessed: 2016-06-02.
URL http://doc.aldebaran.com/2-1/naoqi/motion/control-cartesian-api.html#ALMotionProxy::getPosition__ssCR.iCR.bCR

Aldebaran Robotics, 2015c. H25 - links. Accessed: 2016-06-01.
URL http://doc.aldebaran.com/2-1/family/robots/links_robot_v32.html

Aldebaran Robotics, 2015d. Joint control api. Accessed: 2015-05-21.
URL http://doc.aldebaran.com/2-1/naoqi/motion/control-joint-api.html

Aldebaran Robotics, 2015e. Poses. Accessed: 2016-06-09.
URL http://doc.aldebaran.com/2-1/dev/python/examples/motion/poses.html

Aldebaran Robotics, 2015f. Programming. Accessed: 2016-06-01.
URL http://doc.aldebaran.com/2-1/dev/programming_index.html

Aldebaran Robotics, 2015g. Python sdk. Accessed: 2016-02-24.
URL http://doc.aldebaran.com/2-1/dev/python/index.html

Aldebaran Robotics, 2015h. Simulated robots. Accessed: 2016-06-01.
URL http://doc.aldebaran.com/2-1/dev/tools/robot-simulation.html

Aldebaran Robotics, 2015i. Understanding autonomous life settings. Accessed 2015-12-04.
URL http://doc.aldebaran.com/2-1/nao/nao_life.html

Aldebaran Robotics, 2016a. Cool tools. Accessed 2016-06-01.
URL https://www.ald.softbankrobotics.com/en/cool-robots/cool-tools

Aldebaran Robotics, 2016b. Joints information. Part of complete, updated NAO documentation. Accessed: 04.06.2016.
URL http://ii.tudelft.nl/naodoc/site_en/reddoc/hardware/joints-names.html

Anthony Antonacci, D. M., 2009. Introduction to labview mathscript. Accessed: 2016-03-07.
URL http://home.hit.no/~hansha/documents/labview/training/LabVIEW%20MathScript/Background/Introduction%20to%20LabVIEW%20MathScript.pdf

ART, 2016. Advanced realtime tracking. Accessed: 2016-01-24.
URL http://www.ar-tracking.com/technology/

Baker, M. J., 2016. Maths - conversion quaternion to matrix. Accessed: 2016-04-21.
URL http://www.euclideanspace.com/maths/geometry/rotations/conversions/quaternionToMatrix/

CProgramming.com/"Confuted", 1997-2011. Using quaternion to perform 3d rotations. Accessed 2016-04-10.
URL http://www.cprogramming.com/tutorial/3d/quaternions.html

Cruse, H., 1986. Constraints for joint angle control of the human arm. Biological Cybernetics (54), 123–132.

De Xu, Carlos A Acosta Calderon, J. Q. G. H. H., 2013. An analysis of the inverse kinematics for a 5-dof manipulator. International journal of Automation and Computing 2 (2), 114–124.

Diaz, A., January 2015. STEM System Devkit Manual. Sixense Entertainment Inc., gpd-002 Edition, effective date: 2015-08-05.

Google Code, 2015. Framework for playstation move on pc. Accessed 2015-01-24.
URL https://code.google.com/archive/p/moveframework/

Ho, N., 2011. Decomposing and composing a 3x3 rotation matrix. Accessed: 2015-06-19.
URL http://nghiaho.com/?page_id=846

Keller, J. B., 1975. Closest unitary, orthogonal and hermitian operators to a given operator. Mathematics Magazine 48 (4), 192–197.

Koenemann, J., 2011. Whole-body imitation of human motions with a nao humanoid: Real-time teleoperation. Youtube-video, results of bacherlor's thesis, https://www.youtube.com/watch?v=dC16A6u8WA8. Accessed: 2016-06-02.

Kofinas, N., 2012. Forward and inverse kinematics for the nao humanoid robot. Master's thesis, Technical University of Crete, Chania, Greece.

Kuo, A. D., June 2007. Choosing your steps carefully: Trade-offs between economy and versatility in dynamic walking bipedal robots. IEEE Robotics & Automation Magazine (14), 18–29.

Mark W. Spong, Seth Hutchinson, M. V., 2006. Robot Modelling and Control. John Wiley & Sons, Inc.

Mathas, C., 2012. Sensor fusion: The basics. Accessed: 2016-06-18.
URL http://www.digikey.com/en/articles/techzone/2012/apr/sensor-fusion-the-basics

Mathworks, 2016. Robotics system toolbox. Accessed: 2016-06-08.
URL http://se.mathworks.com/help/robotics/ref/eul2rotm.html

Morasso, P., 1981. Spatial control of arm movements. Experimental brain Research (42), 223–227.

N. Kofinas, E. Orfanoudakis, M. G. L., 2013. Complete analytical inverse kinematics for nao. 13th International Conference on Autonomous Robot Systems (Robotica).

National Instruments, 2014. Working with .m files in labview. Accessed: 2016-06-01. Full name: Working with .m Files in LabVIEW for Text-Based Signal Processing, Analysis, and Math.
URL http://www.ni.com/white-paper/4854/en/

National Instruments, August 2014. Using c/c++ models(model interface toolkit). Accessed: 2016-03-28.
URL http://zone.ni.com/reference/en-XX/help/374160B-01/vsmithelp/mit_model_from_c/

Redmine - Sixense, 2015. Trouble connecting controllers and packs to base station. Closed forum for BETA-system users Accessed: 22.05.2016.
URL http://redmine.sixense.com/issues/213

Sixense Entertainment, Inc., 2012. Sixense sdk overview. Redmine.Sixense.com, internal pages for BETA-users. Closed forum.Accessed 2016-06-06.

Sixense Entertainment, Inc., 2014. Stem system. Accessed: 2016-06-17.
URL http://sixense.com/wireless

Stack Overflow, 2008. How can i use a dll-file from python. Accessed: 2016-05-21.
URL http://stackoverflow.com/questions/252417/how-can-i-use-a-dll-file-from-python

Strang, G., 1988. Linear Algebra and its Applications, 3rd Edition. Harcourt Brace Jovanovich, Publishers.

Suay, H. B., 2010a. Humanoid robot control and interaction. Website. Accessed: 2012-06-02.
URL http://wiki.ros.org/openni/Contests/ROS%203D/Humanoid%20Robot%20Control%20and%20Interaction

Suay, H. B., 2010b. Nao_rail. Accessed: 2016-06-02.
URL https://github.com/benersuay/nao_rail/tree/fuerte-devel/nao_openni

Team Kouretes, 2014. Naokinematics. Accessed 2016-01-30.

URL https://github.com/kouretes/NAOKinematics

The Sixense Team, 2014a. 08/01/2014 - stem pack details. Accessed: 2016-06-16.

URL http://sixense.com/stemupdate-stem-pack-details

The Sixense Team, 2014b. Kickstarter: Sixense stem. Accessed: 2016-06-16.

URL https://www.kickstarter.com/projects/89577853/stem-system-the-best-way-to-interact
posts/827198

Wikipedia, 2016. Euler angles. Used for pictures. Accessed: 2016-06-02.

URL https://en.wikipedia.org/wiki/Euler_angles