NTNU
Norwegian University of
Science and Technology

Kinematic Control of Underwater Robotic System

Åsmund Pedersen Hugo

Master of Science in Engineering and ICT
Submission date: June 2016
Supervisor: Ingrid Schjølberg, IMT

Norwegian University of Science and Technology
Department of Marine Technology

NTNU Trondheim
Norwegian University of Science and Technology
Department of Marine Technology

MASTER THESIS IN MARINE CYBERNETICS

SPRING 2016

for

STUD. TECH. ÅSMUND PEDERSEN HUGO

Kinematic Control of Underwater Robotic System

Development, Simulations, Physical Implementation

Work Description

Underwater robotic systems can be used for a variety of operations, from reparations of sub-sea installations, to perform inspections and surveys. Such operations will demand robust and fast kinematic control, if they were to be executed automatically.

Development of an inverse kinematic solver for a vehicle-manipulator system is very intricate, and it is therefore implemented on the small, humanoid robot Nao. This enables frequent testing and refinement of the solution algorithm, to create a good foundation for further development on underwater platforms.

Scope of work

1. Perform a state of the art literature review
2. Develop and formulate a forward kinematic model.
3. Develop an inverse kinematic solver for the redundant system.
4. Develop functionality to handle singularities and enhance the solver's performance
5. Perform simulations
6. Perform laboratory experiments
7. Discuss results
8. Conclusions

The report shall be written in English and edited as a research report including literature survey, description of mathematical models, description of control algorithms, simulation results, model test results, discussion and a conclusion including a proposal for further work. Source code should be provided on a CD, memory stick or on a web-based repository. It is supposed that the Department of Marine Technology, NTNU, can use the results freely in its research work, unless otherwise agreed upon, by referring to the student's work. The thesis should be submitted in three copies within June 10, 2016.

Co-supervisor: John Reidar Mathiassen, SINTEF F&A

Professor Ingrid Schjølberg
Supervisor

Preface

This master thesis has been written to conclude a five year integrated masters program in Engineering and ICT, with specialization within marine cybernetics, at the Norwegian University of Science and Technology. The thesis is a result of autonomous work, with support from supervisor Ingrid Schjølberg and co-supervisor John Reidar Mathiassen. The period of work spans from January to June of 2016.

Sintef Fisheries and Aquaculture have started working on a long term humanoid robotics project, at the same time as the Department of Marine Technology is deeply committed to finding new autonomous solutions in the sub-sea environment. With one foot in each camp, this thesis became a symbiosis of developing new ways of doing inverse kinematic control of a humanoid robot, with the intention of implementing the same control strategy on an underwater robot in future work.

From a personal point of view, the reasons why I chose to initiate and undertake the topic of this thesis is down to the pure fun of it. Robotic control synthesizes theoretical mathematics with state of the art technology, which constitutes a venturous arena for exploring the toolbox of programming, natural science and an interest in "high-tech"-gadgets. From the start I wanted to implement the functionality on a platform as open as possible, to partake in the open scientific community.

Abstract

This master thesis presents a new way of solving the inverse kinematic problem for a *vehicle-manipulator system*. This was conducted due to the increasing demand for higher autonomy in the underwater environment, where robotic systems plays a leading role, but currently lacks the ability to solve inverse kinematic problems quickly enough. The inverse kinematic problem is particularly complicated for a *vehicle-manipulator system*, which constitutes an integral part of the desired technological capabilities.

To obtain the joint set solution quickly, and in a robust manner, the solver utilizes the *Damped Least Squares Method* with a new way of weighting each increment in joint value dynamically, a new way of switching between control states with fuzzy logic, and dynamically damping the singular values of the system's Jacobian if they approach zero. These features have ensured error convergence speed, stability and higher solution ratio. The solver also has functionality for saturating joint changes when they approach a physical limit.

We have used the humanoid robot Nao as the physical platform to implement the solver. Nao is an easy to use, low-level robot with well documented resources available. The solver is implemented in Python, which is also the language used to communicate with Nao, with the code available at `https://github.com/asmhug/Nao-inverse-kinematics-solver`.

Different experiments have been conducted, where the the solver was given a desired configuration, and the joint set solution was sent to Nao as sequential commands for every joint actuation. Nao moved around on a surface marked with Cartesian axes, to enable physical measurements for comparisons between the end configuration and the desired configuration. As well, Nao was set to perform a small movement on a gridded surface with a mounted camera, to determine inaccuracies in his movements precisely. The overall testing showed great inaccuracies in Nao's internal sensory measurements, but the physical measurements substantiated both a successful inverse kinematic solution and movement execution.

Experiments and simulations proved the inverse kinematic solver as a very good tool for solving the inverse kinematic problem of vehicle-manipulator systems. The method of weighting joint increments dynamically and the fuzzy logic to switch between control states, have shown to be particularly interesting new features to this scientific field.

Sammendrag

Denne masteroppgaven presenterer en ny måte å løse inverskinematikk-problemet for et *fartøy-manipulator-system*. Dette ble utført med tanke på den økende etterspørselen for mer automatikk i det teknologiske miljøet under vann, hvorav mangelen på en høyhastighets invers-kinematikk-løser hindrer oss i å møte den nevnte etterspørselen.

For å få tak i et løsningsett med leddverdier hurtig, og med en robust løsningsmetodikk, brukes den numeriske løseren metoden *Dempede Minste Kvadraters Løsning*. Metoden har blitt utvidet med en ny måte å vektlegge hver leddøkning dynamisk, en ny måte å bytte mellom hvilke tilstander man kontrollerer med fuzzy-logikk og dynamisk dempning av Jacobi-matrisens singulærverdier hvis disse nærmer seg null. Løseren har også funksjonalitet for å begrense leddøkninger i henhold til en kontinuerlig funksjon, hvis de nærmer seg sin fysiske begrensning.

Vi har anvendt en humanoid robot ved navn Nao, til å være den fysiske plattformen vi implementerer den numeriske løseren på. Nao er en lett-brukelig, lavnivå robot med god dokumentasjon og brukerveiledning tilgjengelig. Løseren er implementert i programmeringsspråket Python, som også er språket vi bruker til å kommunisere med Nao. Ved å anvende Python og tilgjengeliggjøre all kode på `https://github.com/asmhug/Nao-inverse-kinematics-solver`, håper vi å muliggøre forbedringer og utvikling til støtte av nye platformer, for alle interessenter.

Forskjellige forsøk har blitt utført, hvor en ønsket konfigurasjon har blitt innsatt i den numeriske løseren, og det resulterende løsningsettet med ledd-verdier har blitt sendt sekvensielt til Nao for hver enkelt ledd-bevegelse. Nao beveget seg rundt på en overflate med et oppmerket Cartesisk aksesystem for å muliggjøre fysiske målinger. Disse kunne så sammenliknes med den ønskede konfigurasjonen. I tillegg ble Nao satt til å gjennomføre en mindre manøver på en rutete overflate, med et fastmontert kamera, for å bestemme unøyaktigheter ved hans bevegelser mer presist. Alt i alt har testingen underbygget at den numeriske løseren suksessfullt finner inverskinematikken til et fartøy-manipulator-system.

Forsøk og simuleringer har bevist at den numeriske løseren er et veldig godt redskap. Spesielt kan man utheve metodikken med å løse vekting av ledd-økninger dynamisk og fuzzy-logikken for å bytte mellom tilstander å kontrollere, som spesielt interessante, og som nye tilskudd til dette vitenskapelige feltet.

Acknowledgements

Firstly, I would like to thank my supervisor Ingrid Schjølberg, from IMT, and co-supervisor John Reidar Mathiassen, from SINTEF F&A. Ingrid for giving me invaluable contributions on scope of work, draft feedback, relevant literature, and guidance on academic writing and structure. John Reidar for advising me on laboratory experiments, presentation of scientific data and guidance on academic writing and structure. Secondly, I would like to thank SINTEF F&A for providing me with the Nao robot and testing facilities, in addition to free gastronomic experiences at lunch hour. Thirdly, Jonatan Sjølund Dyrstad deserves my heartfelt gratitude for proofreading this bulky dissertation. Lastly, I want to thank my fellow students at both Tyholt and SINTEF for motivational and inspiring conversations, not to mention comradeship.

Contents

List of Figures

List of Tables

Nomenclature

$\mathbf{C}_{space}$	The collection of every possible position and orientation of a robot's manipulator
$\mathbf{Q}_{space}$	The collection of every possible permutation of joint values for a robot
$\mathscr{F}_b$	The basic coordinate frame to reference every of the robot's joint configurations
θ_i	The angle of rotation for a revolute joint i
API	Application Programming Interface
$C++$	C++ is a Multi-paradigm, procedural, functional, object-oriented programming lanuage
d_i	The extension or contraction for a prismatic joint i
DH	Denavit-Hartenberg parameters
$DLSM$	Damped Least Squares Method
DSS	Decision Support Systems
FK	Forward Kinematics
IK	Inverse Kinematics
$J^\dagger$	The Moore-Penrose inverse of a Jacobian matrix
JCM	Jacobian Control Method
JLA	Joint Limit Avoidance
Nao	A humanoid robot from Aldebaran
OS	Operating System
$Python$	Python is a high-level, general, interpreted programming language
q_i	The general term for a variable joint in the kinematic model (both prismatic and revolute)
ROV	Remotely Operated Vessel
SVD	Singular Value Decomposition
VSS	Variable-Step Solver

Chapter 1

Introduction

The introductory section is organized as following: background and motivation in Section 1.1, previous work in Section 1.2, the thesis objective in Section 1.3, how the thesis incorporates and builds on existing theory in 1.4 and the organization of the thesis in Section 1.5.

1.1 Background and Motivation

In the course of becoming an engineer, the desire to enhance the state of society by technology has been a prime motivational factor. That channels into the the marine environment and its challenges today. The creation of wealth in the Norwegian society is highly dependent on marine based industries, and for these industries to be competitive, environmental friendly and, especially, innovative in the twenty first century, leaps in technological development is axiomatic. The possibilities in this regard were clearly emphasized during the Ocean Week in Trondheim 2016, where the leading column was dedicated to future technology for enabling marine mining, robotized aquaculture and discovery of new ocean floor territories (Schjølberg, 2016). The growing demand for more autonomy in the marine industry was clearly emphasize in the annual TEKMAR conference in Trondheim. TEKMAR's main objective is to identify the technological solutions needed in the future for the fishing and aquaculture industry - only second to Oil and Gas as exporting industry in Norway (Fiskeridepartementet, 2015). A comprehensive contributing survey, conducted by SINTEF MARINTEK, gathered information about important future capabilities from the entire marine industry in Norway (Minsaas, 2015). It presented "safety and maintenance" and "automation and remote controllability" as the key capabilities by about 70% of the respondents, with both issues being addressed by the objective in this thesis. Underwater robotic systems can be used for a variety of operations, from reparations of sub-sea installations, to perform inspections and surveys. Such operations will demand robust and fast kinematic control, if they are to be executed automatically.

The motivation is further enhanced when looking deeper into the possible benefits from a generalized theoretical framework for highly dexterous manipulation. It can be easy applicable to solve a wide range of practical tasks on land as well. From a field trip to one of the worlds most technological advanced production lines of farmed salmon in the summer of 2015, at SalMar on Frøya Island, the possibilities

for further automation in the production line was vivid. Still, today, several easy executable - and more importantly; simply described - tasks in a production line are demanding human attention. One can foresee fatiguing, repetitive work being executed by more intelligent and robust robotic systems, and in a more economically efficient manner.

1.2 Previous Work

From a smaller thesis conducted as a lead up to this master thesis, the following work was performed and experiences taken:

Forward kinematics model

A simplified forward kinematics model was established for Nao. The simplifications due to the model only being constructed for learning about different inverse kinematics techniques, and these techniques were only to be tested through simulations. This forward kinematics model provided the foundation for how this thesis is approached.

Analytical inverse kinematics

An analytical inverse kinematics solution to the simplified forward kinematics model was obtained. This was possible because the system was separated into two different models - one for the body motion and one for the arm, where they individually were solvable on closed form. This proved as insufficient for further exploitation as it lacks any option to handle singularities (e.g. a vicinity approach to a desired singular configuration) as well as it did not give the desired "humanoid" behavior one would desire.

Numerical inverse kinematics

Several techniques for solving the inverse kinematics numerically were tested through simulation. This showed great performance by the *damped least squares method*, with a preset damping constant, which led to this method being prioritized for further development. This particular work also highlighted the need for actions to assure much quicker convergence.

Matlab implementation

The required equations and the different methods of inverse kinematics were implemented in Matlab. Matlab is a very low threshold tool for solving linear algebra, as well as it "hides" a lot of what is happening through matrix manipulation. The conclusions from using Matlab became to try and implement the same sort of algorithms and methods in a general-purpose language, which demands more intuition in how to perform every mathematical part of the solution.

1.3 Objective

The prime objective of this thesis is to provide an improved way of solving the inverse kinematic problem for an underwater robot, termed *vehicle manipulator-system*, so as to enable autonomous robots to perform tasks not achievable today. In this lies the need for robustness to singularities, performance time within a few seconds and solvability within it's defined configuration space. This is to be implemented on a the humanoid robot Nao, to ensure the applicability of the solver on a physical system, to test for possible enhancements before future implementation on an underwater system and to prove it's employability on a diverseness of platforms.

1.4 Incorporation to Modern Robotics Theory

For *redundant manipulators*, where identical serial links are not present, and for *vehicle-manipulator* systems, methods evolving around velocity relationships are the forefront strategy for solving the inverse kinematics problem. In (Siciliano and Khatib, 2008) they portray the method of *Damped Least Squares*, with a system of task priorities for the redundant degrees of freedom to handle, to solve the inverse kinematics of a vehicle-manipulator system. The same tactic of task priorities is used in (Antonelli and Antonelli, 2014), where fuzzy logic is added to create efficient movements. This thesis intends to build on these approaches, by using the relationship between joint and spatial velocities, and measures to avoid singularities. Furthermore, it will add to the preexisting methods

- a system of fuzzy logic to switch between which Cartesian states to control, to increase local redundancy at the given instance of numerical solution,
- dynamic weighting, which will ensure a bound step-size at each iteration and a stable convergence,
- a dynamic damping factor for each joint.

In addition, this is the first time the Damped Least Square method is applied to Nao, and probably at all to a humanoid robot for whole body control.

1.5 Organization

Chapter 2 provides a thorough walk-through of relevant mathematical and control theory, from the fundamental ideas to the utilized methods of inverse kinematic of redundant systems. This chapter of background theory also contains a detailed description of Nao's physical features and the software platform to be used for implementing the solution.

Chapter 3 contains the development of a kinematic model for the vehicle-manipulator system, including a new approach for modelling the degrees of freedom provided by the mobile body.

Chapter 4 contains the development of the inverse kinematics algorithm for Nao, from the basic Jacobian Control Method to the Damped Least Squares Method, with a dynamic weighting of joint increment and a switching control system.

Chapter 5 presents simulation results to uphold the functionality of the inverse kinematic algorithm, by showing singularity handling, solution efficiency, control switching and stable error convergence. The chapter provides joint trajectories, singular value development and the corresponding error of every degree of freedom.

Chapter 6 presents the physical experiments with Nao, from connectivity, software API needed, to the results from extensive testing. The results are presented as photo-series to highlight the actual robot movement, with physical measurements to backup the solution.

Chapter 7 provides a discussion regarding the initial objective, the theoretical development, simulations and the experimental results.
Chapter 8 concludes with some final thoughts on the thesis' results and the experience from partaking in the work. Lastly, it provides recommendations for future work.

Chapter 2

Background Theory

2.1 Forward Kinematics

2.1.1 Perception of Space

"Kinematics is the study of the geometry of motion" (Beggs, 1983). For humans the perception of space is connoted to terms as "length", "height", "width" and so forth, which we mathematically denote as "x", "y" and "z" in the Cartesian space. To describe rotations or an object's orientation, we use terms as "turn" and "roll" in everyday language, with the scientific parallel often using "yaw", "heave" and "roll". To understand how we can control a robot with our perception of space, we need to create transformations between our way and it's way of perceiving.
Forward Kinematics (FK), also called *Forward Kinematic Transformation*, is the transformation from the robot's mathematical perception of space into ours. A robot's ability to move is controlled by the actuators at every joint, which dictates the kinematics of a robot into the variation of joint parameters. In other words, from the robot's perspective this implicates a three dimensional space model described by joint parameters. A value of a prismatic joint will give an extension of reach for a manipulator or robot limb, while an angular value for a revolute joint will give a certain position and orientation. Only by controlling these values, one can control the position and orientation of the entire robotic structure. The entire space of reachable positions and orientations for a robot is defined as

$$Configurationspace := \mathbf{C}_{space} \in \mathbb{R}^3 \times \mathbf{SO}(3), \tag{2.1}$$

where $\mathbb{R}^3$ denotes the three values x, y and z in Euclidian space, and $\mathbf{SO}(3)$ represents the three rotation matrices that defines the orientation relative to a *base frame*. One can see the set of reachable positions and orientations denoted as the *configuration manifold*, a manifold of the complete set of generalized configurations. The equivalent of obtaining the positions and orientations in the *configurations space* is every possible permutation of joint values (Choset, 2014), defined as

$$Jointspace := \mathbf{Q}_{space} \in \mathbb{R}^n, \tag{2.2}$$

where n is the number of chained joints. FK defines the mapping from the jointspace to the configurationspace, described mathematically in Equation 2.3 and illustrated below in Figure 2.1.

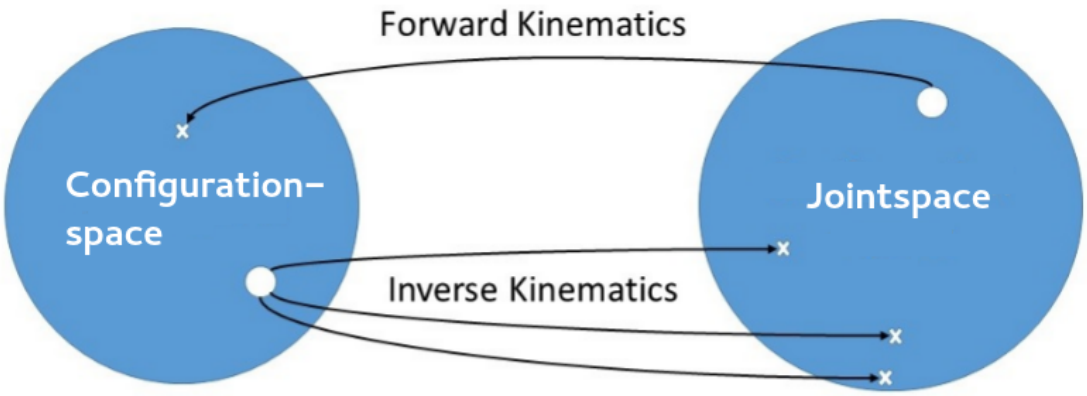

Figure 2.1: Transformation between configurationspace and jointspace

One very important notice from this illustration, explained in detail in 2.1.2, is the unique mapping from one joint configuration to one position and orientation, but the non-unique mapping from one position and orientation to several - often infinite - joint sets - the IK is mathematical surjective. In practical terms, this implies several (infinite) sets of joint values delivering the robot's manipulator to the desired position and orientation.

2.1.2 Mathematical Transformation

According to (Fahimi, 2008) the mathematical transformation between the two domains can be described by

$$\mathbf{x}_{6\times1} = \mathbf{T}_{4\times4}(\mathbf{q_{n\times1}}), \tag{2.3}$$

where $\mathbf{T}_{4\times4}$ is the FK transformation matrix ellaborated in Equation 2.6, $\mathbf{x}_{6\times1}$ is the vector containing position and orientation (as defined in Equation 2.4) and $\mathbf{q}_{n\times1}$ is the vector containing all robot joint values (as defined in Equation 2.4):

$$\mathbf{x}_{6\times1} = \begin{bmatrix} x \\ y \\ z \\ \phi \\ \theta \\ \psi \end{bmatrix}, \quad \mathbf{q}_{n\times1} = \begin{bmatrix} q_1 \\ q_2 \\ \vdots \\ q_n \end{bmatrix}. \tag{2.4}$$

To simplify the physical understanding of the notation, one can consider only a two-dimensional robot manipulator with revolute joints:

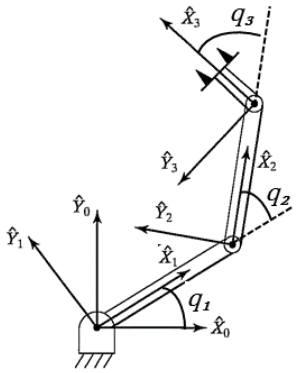

Figure 2.2: A two-dimensional manipulator illustration

where our $\mathbf{x}$ will contain the position and orientation of the end-effector in the base frame 0 and our $\mathbf{q}$ will contain the angular values of the three joints. Finding the FK transformation between the base frame and the manipulator, becomes a tidy process if one follows the standardized convention of *Denavit-Hartenberg parameters* (DH), first introduced in 1955 (Denavit, 1955). The problem of FK for a chained link with n joints is hashed into n sub-problems, each representing a link in relation to the previous. Each joint is given an affixed right hand coordinate system, starting in the base frame 0, where the z_i-axis is set normal to the angle of rotation for revolute joints, denoted θ_i, (Fig: 2.3) and where z_i is set parallel to the extension, denoted d_i, for a prismatic joint.

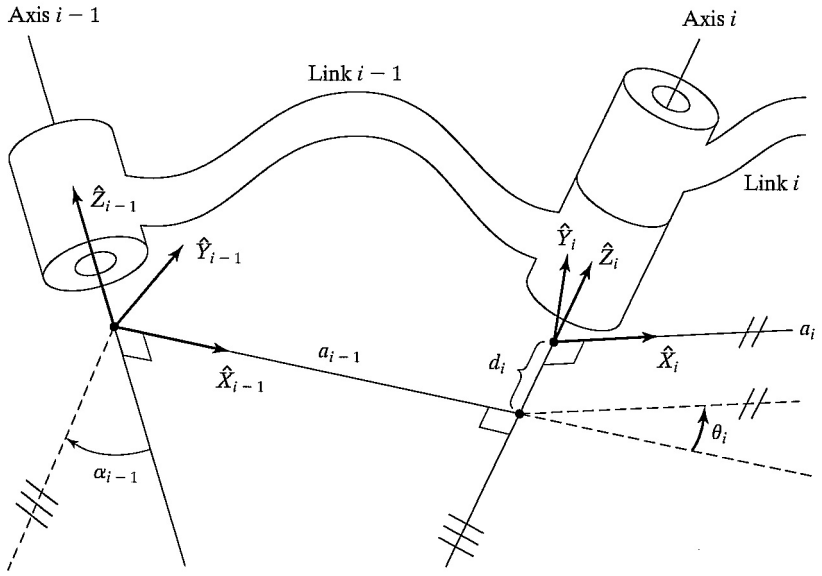

Figure 2.3: Link frames are attached so that frame i is attached rigidly to link i (Craig, 2005).

The DH parameters will then be set by the following definitions (Craig, 2005):

$a_i :=$ the distance from $\hat{Z}_i$ to $\hat{Z}_{i+1}$ measured along $\hat{X}_i$;

$\alpha_i :=$ the angle from $\hat{Z}_i$ to $\hat{Z}_{i+1}$ measured about $\hat{X}_i$;

$d_i :=$ the distance from $\hat{X}_{i-1}$ to $\hat{X}_i$ measured along $\hat{Z}_i$; and

$\theta_i :=$ the angle from $\hat{X}_{i-1}$ to $\hat{X}_i$ measured about $\hat{Z}_i$.

The transformation of each subproblem (link) is then established by the following explicit formula (Kofinas et al., 2015):

$$ {}^{i-1}_{i}T = \begin{bmatrix} c\theta_i & -s\theta_i & 0 & a_{i-1} \\ s\theta_i c\alpha_{i-1} & c\theta_i c\alpha_{i-1} & -s\alpha_{i-1} & -s\alpha_{i-1}d_i \\ s\theta_i s\alpha_{i-1} & c\theta_i s\alpha_{i-1} & c\alpha_{i-1} & c\alpha_{i-1}d_i \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{2.5} $$

where $cos(\theta)$ is denoted as $c\theta$ and $sin(\theta)$ is denoted as $s\theta$. The total FK of a chained manipulator with n joints, can now be established by the product of every sub-problem solution, or sub-transformation if you prefer:

$$ \mathbf{T}^{Base\ frame}_{End-effector}(\mathbf{q}) = \mathbf{T}^0_1\mathbf{T}^1_2....\mathbf{T}^{n-1}_n \tag{2.6} $$

2.2 Inverse Kinematics

Inverse Kinematics (IK) reverses the calculation from Forward Kinematics, to determine the joint parameters that achieves the desired position and orientation for the end effector (McCarthy, 1990). This is the very essence of kinematic robotic control. One targets a trajectory of movement or a position (and orientation) for a robot limb or end-effector, which needs to be translated into joint trajectories or values for the robot to execute. From Equation 2.3 we can observe that the relation between $\mathbf{x}$ and $\mathbf{q}$ is a matter of solving the equating system. From the involved trigonometric functions in Equation 2.5 we can also deduce our system to be non-linear, transcendental (Craig, 2005). This implies a concern regarding the existence of solutions, the size of the solution manifold - as illustrated inn Figure 2.4 - and how we go about solving it. Donald Peiper documented and classified the different instances in 1968, contributed by the following illustration:

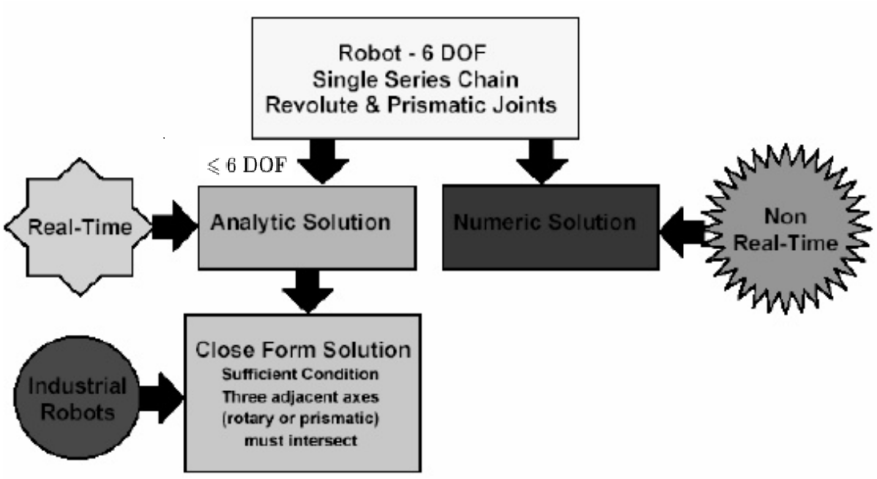

Figure 2.4: The different approaches and real time aspects to IK(Peiper, 1968)

2.2.1 Closed-form analytical solution

For the IK problem to be solvable in a closed-form we need physical properties regarding the robotic structure to be upheld. A robotic chain can only be solved in closed-form if the degrees of freedom (joints) are less than six, at the same time as three of the joints have intersecting axes of rotation (Peiper, 1968). If these conditions are present, the problem of IK can be solved either geometrically or algebraically (the numerical approach is due to Section 2.2.4) (Korein et al., 1982). The geometric approach uses the same geometrical structure as for developing the FK, while the algebraic approach is more methodical in it's attempt solve the derived FK explicitly for the joint variables (Kucuk and Bingul, 2006). The algebraic approach can be understood by exploring Equation 2.3:

- We assume that $\mathbf{x}$ is equivalent to a general transformation matrix - in three dimensions - with $\mathbf{x}$ as input:

$$\begin{aligned}\mathbf{x} &= \mathbf{R}(\psi)\mathbf{R}(\theta)\mathbf{R}(\phi)\mathbf{A}(x,y,z) \\ &= \begin{bmatrix} c(\theta)c(\psi) & -c(\phi)s(\psi)+s(\phi)s(\theta)c(\psi) & s(\phi)s(\psi)+c(\phi)s(\theta)c(\psi) & x \\ c(\theta)s(\psi) & c(\phi)c(\psi)+s(\phi)s(\theta)s(\psi) & -s(\phi)c(\psi)+c(\phi)s(\theta)s(\psi) & y \\ -s(\theta) & s(\phi)c(\theta) & c(\phi)c(\theta) & z \\ 0 & 0 & 0 & 1 \end{bmatrix}\end{aligned}$$

where $\mathbf{R}$ represents a pure rotation, $\mathbf{A}$ represents an affine translation, cos() is denoted c() and sin() is denoted s(). This leads to the equation

$$\mathbf{R}(\psi)\mathbf{R}(\theta)\mathbf{R}(\phi)\mathbf{A}(x,y,z) = \mathbf{T}_{4\times4}(\mathbf{q_{n\times1}}) = \mathbf{T}_1^0\mathbf{T}_2^1....\mathbf{T}_n^{n-1}, \tag{2.7}$$

which means that we have a system of six given equations and n unknown(s), that can be manipulated to obtain the n joint parameters on explicit form.

2.2.2 Jacobian of a Robotic Structure

A very useful relation, which can be derived directly from the relation between joint positions and end-effector position, is the mapping between velocities in each

coordinate system (From et al., 2014). The velocity space is easier to operate in when we want to determine the IK iteratively. In practical terms this implies what end-effector velocities will occur, relative to the base-frame, corresponding to certain joint velocities. This relationship is established through the *Jacobian* (Kaplan, 2002). Formally, the Jacobian is a set of partial differential equations - a multidimensional form of a derivative. Suppose that we have n functions f, each of which is a function of i independent variables (Craig, 2005):

$$\begin{aligned} y_1 &= f_1(x_1, x_2, ..., x_i) \\ y_2 &= f_2(x_1, x_2, ..., x_i) \\ &\vdots \\ y_n &= f_n(x_1, x_2, ..., x_i) \end{aligned}$$

If differentiated relative to every independent variable:

$$\partial Y = \frac{\partial F}{\partial X}\partial X \rightarrow \dot{Y} = J(X)\dot{X}.$$

Then, to find the Jacobian for a robotic system we simply find the first order partial derivative of our FK (Joubert, 2008), which leaves us with

$$\dot{\mathbf{x}} = \frac{\partial \mathbf{T_{4\times4}}}{\partial \mathbf{q_{n\times1}}}\dot{\mathbf{q}_{\mathbf{n\times1}}} = \mathbf{J}\dot{\mathbf{q}}. \tag{2.8}$$

We can split the Jacobian into a linear velocity contribution and an angular velocity contribution (Khatib, 2008)

$$\mathbf{J} = \begin{bmatrix} \mathbf{J_v} \\ \mathbf{J_w} \end{bmatrix}_{6\times n}. \tag{2.9}$$

We identify the Cartesian translations from the FK, $\mathbf{T_{4\times4}}$, as the three first entries in column 4:

$$\mathbf{T_{4\times4}}(\mathbf{q_{n\times1}}) = \begin{bmatrix} ... & ... & ... & p_x \\ ... & ... & ... & p_y \\ ... & ... & ... & p_z \\ 0 & 0 & 0 & 1 \end{bmatrix}. \tag{2.10}$$

The linear velocity contribution $\mathbf{J_v}$, can now be obtained through partial differentiation of the translational part of the forward kinematics (See Eq.: 2.10).

$$\mathbf{J_v} = \begin{bmatrix} \frac{\partial p_x}{\partial q_1} & \frac{\partial p_x}{\partial q_2} & ... & \frac{\partial p_x}{\partial q_n} \\ \frac{\partial p_y}{\partial q_1} & \frac{\partial p_y}{\partial q_2} & ... & \frac{\partial p_y}{\partial q_n} \\ \frac{\partial p_z}{\partial q_1} & \frac{\partial p_z}{\partial q_2} & ... & \frac{\partial p_z}{\partial q_n} \end{bmatrix}_{3\times n} \tag{2.11}$$

When we differentiate to find linear velocities with respect to prismatic joint changes, the assumption above in Equation 2.11 is correct. On the other hand, when we differentiate with respect to revolute joints, we actually linearize the system and obtain the instantaneous linear velocity contribution from an angular movement (Craig, 2005). This is demonstrated in Figure 2.5, where the tangent p is the instantaneous linear velocity from an angular change in joint q.

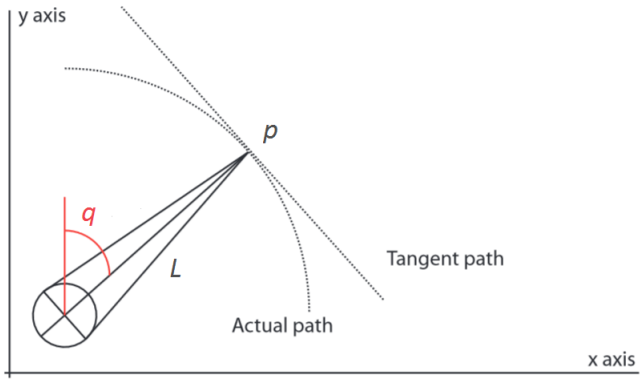

Figure 2.5: Linearized velocity from a revolute joint

To find the angular velocity contribution of our Jacobian, we have to evaluate for each joint:

- Is it revolute?
- Which axis of rotation?
- How is the joint frame relative to the base frame?

This materializes in the following formulae (Murray et al., 1994):

$$\omega_1^0 = \rho_1 \dot{q}_1 z \tag{2.12}$$
$$\omega_n^0 = \omega_{n-1}^0 + \rho_n \dot{q}_n R_{n-1}^0 z, \tag{2.13}$$

where $\rho = 1$ for a revolute joint and $\rho = 0$ for a prismatic joint, ω_n^0 is the angular velocity of joint n relative to base-frame (0), z is the vector giving the axis of rotation - $z = [0, 0, 1]^T$ - and R_{n-1}^0 is the rotational matrix from the base frame to the n'th joint frame. This corresponds to the third column of an affine rotation in our robotic system's FK (Khatib, 2008),

$$\mathbf{T_{4\times4}}(\mathbf{q_{i\times1}}) = \begin{bmatrix} \dots & \dots & {}_i^0 r_{13} & \dots \\ \dots & \dots & {}_i^0 r_{23} & \dots \\ \dots & \dots & {}_i^0 r_{33} & \dots \\ 0 & 0 & 0 & 1 \end{bmatrix}. \tag{2.14}$$

Extracting ${}_i^0 r_{13}$, ${}_i^0 r_{23}$ and ${}_i^0 r_{33}$ from every joint transformation matrix between joint 1 and joint n, yields the angular velocity contribution to the Jacobian:

$$\mathbf{J}_w = \begin{bmatrix} {}_1^0 r_{13} & {}_2^0 r_{13} & \dots & {}_n^0 r_{13} \\ {}_1^0 r_{23} & {}_2^0 r_{23} & \dots & {}_n^0 r_{23} \\ {}_1^0 r_{33} & {}_2^0 r_{33} & \dots & {}_n^0 r_{33} \end{bmatrix}. \tag{2.15}$$

2.2.3 Robotic Singularities

The American National Standard for Industrial Robots and Robot Systems defines a singularity as "a condition caused by the collinear alignment of two or more robot axes resulting in unpredictable robot motion and velocities." (ANSI, 1991) This

implicates their occurrences as local, instantaneous phenomena, but with global implications for kinematic control. The notion of singularities can straightforwardly be understood from the Jacobian of the robotic chain. For a series of joints, on the contrary to e.g. a fully parallel jointed manipulator or robotic system, the singularities of internal alignments of joint axes are of higher significance and interest than those of workspace constraints (Donelan, 2006).

Manipulator Singularity

To comprehend to geometrical situation of two internal alignments, the following figure of a planar manipulator is presented:

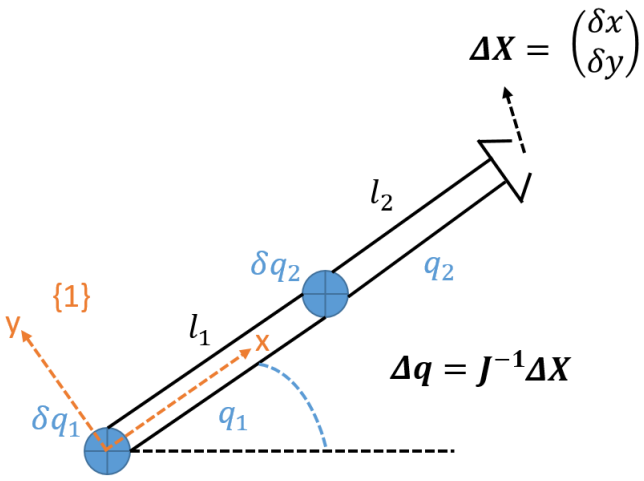

Figure 2.6: A singular joint configuration of a planar manipulator

from where we can derive that a very small displacement of the end-effector $\mathbf{\Delta X}$ will correspond to the following joint change and velocities

$$\mathbf{\Delta q} = \mathbf{J}^{-1}\mathbf{\Delta X} \rightarrow \tag{2.16}$$

$$\mathbf{J}^{-1}_{(1)} \equiv \begin{pmatrix} \frac{1}{l_1 q_2} & \frac{1}{l_1} \\ -\frac{l_1+l_2}{l_1 l_2 q_2} & -\frac{1}{l_1} \end{pmatrix}. \tag{2.17}$$

Geometrically, we can understand from Figure 2.6 that an alignment of two chains will cause the loss of one internal degree of freedom. When $q_2 \rightarrow 0$ we will approach a situation where a very small change in q_2 or q_1 will yield the approximately same end-effector velocity. Mathematically - observing a small displacement from frame $\{1\}$ in Figure 2.6 - we establish Equation 2.16, and note that any desired change in end-effector position will yield unfortunate large joint changes, when $q_2 \approx 0$. This is the equivalent to the Jacobian loosing full rank. In lieu of a singular configuration as elaborated above, the kinematic equations will yield the need for up to infinite joint change, corresponding only to the slightest end-effector configuration change. For industrial manipulators, one often characterize the different singularities of internal chain alignments to be *shoulder singularities*, *elbow singularities* or *wrist singularities* (Hayes et al., 2003).

The Singularities of Nao

In the case of Nao, we are at first glance only dealing with a 4 DOF manipulator, which would restrict the issue of singularities. But, when bearing in mind the body as a part of the manipulator, the intricacy of singularities advances. Though highly dependent on the numerical algorithm for solving the IK and the kinematic modelling of the robot body, we can certainly emphasize possible singularities we may encounter as:

- *Hip-pitch joint* with *Shoulder-pitch joint*
- *Shoulder-roll joint* with *Elbow-roll joint*
- *Body-pose* in $X-Y-$plane with *Elbow-roll* or *Shoulder-roll* joint

When considering the constraints of the elbow-roll joint, physically limited between -2.0^o and -88.5^o, we can irrefutably defy the physical existence of a singularity from alignment between upper and lower arm (a case of traditional *elbow singularity*). However, caution must be taken regarding this matter because of possible weaknesses in the numerical algorithm. The approach for upholding joint constraints and the sensitiveness around singularities can both lead to undesirable behavior, even though 0.0^o is not physically obtainable.

2.2.4 Numerical solution

When a manipulator consists of a chain of more than six DOF it is defined as redundant (Chiaverini et al., 2008). The key of enriching the robot's workspace with more natural and efficient poses is highly useful. It can either derive our manipulator with higher dexterity or greater mobility in larger spaces (Fahimi, 2008). For these applicable benefits, redundant manipulators have been subjected to extensive studies by researchers and engineers, especially since the last turn of the century. They have been employed in high-importance applications, such as the *Special Purpose Dexterous Manipulator* (Canada Arm 2) at the International Space Station (Dewar, 2011). From the perspective of this thesis, the greatly explored theory of solving IK for redundant manipulators is of high relevance, even though our physical platform uses a four DOF manipulator. When considering a completely dexterous mobile robot, as is the case for a remotely operated submersible vessel, the mobility itself represents six degrees of freedom, which in turn renders the vehicle-manipulator system redundant.

To solve the IK of redundant robotic systems one has to apply numerical theory. An important notice is the option of also solving IK for non-redundant manipulators by numerical mathematics, but this approach neglects the possibilities for multiple solutions, and is therefor rarely applied to such. There are several ways of attacking this from a mathematical point of view, the most utilized ones being:

- *Jacobian Transpose Method*, which involves an idea of virtual work. A force enacting on the end-effector will cause joint torques in the robotic structure, which will be connected by the transpose of the structures Jacobian (Pechev,

2008):

$$\tau \approx \dot{\theta} = \mathbf{J}^T\mathbf{F} \rightarrow$$
$$\Delta\theta = \alpha\mathbf{J}^T(\theta)\Delta\mathbf{x}.$$

This method have the advantages of simple computations (no costly matrix inversions) and numerical robustness, but at the same time the drawbacks of slow convergence and unpredictable joint configurations (Schaal, 2014).

- *The Pseudoinverse Method*, which involves solving the relationship between Cartesian and joint space with a generalized inverse matrix, $J^\dagger$ - often refered to as the *Moore-Penrose* inverse (Moore, 1920, Penrose, 1955).

$$\Delta\theta = \alpha\mathbf{J}^\dagger(\theta)\Delta\mathbf{x}$$

It is a computationally fast method (beeing second order) operating on a basis of finding a shortest path in joint space. The disadvantages are singularities and unpredictable joint configurations.

- *Levenberg-Marqardt Damped Least Squares Method*, first implemented for IK by (Nakamura and Hanafusa, 1986) and (Wampler et al., 1986), is introducing a damping constant to the Pseudoinverse Method which makes it robust around singularities. The principal is to minimize the following quantity:

$$||J\Delta\mathbf{q} - \mathbf{e}||^2 + \lambda^2||\Delta\mathbf{q}||^2, \tag{2.18}$$

derived from the damping factor λ and the equality

$$\mathbf{e} = \mathbf{J}\Delta\mathbf{q}.$$

The advantage of handling singularities is at the same time introducing an error in reproduction of velocities, which slows down the convergence (Egeland et al., 1991).

- *The Cyclic Coordinate Descent Method*, a method first developed for IK by (Wang and Chen, 1991), which involves adjusting one degree of freedom at a time. The aim of every adjustment is to close the vectorial distance from end-effector to the target (Canutescu and Dunbrack, 2003). It iterates through the degrees of freedoms (joints) until the objective is achieved. The idea is most easily described illustratory:

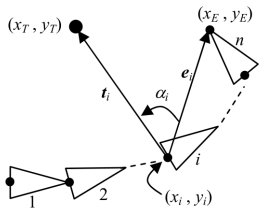

Figure 2.7: An n-link joint chain (Mukundan, 2008)

where the vector from the changing joint, i, to the target is denoted $\mathbf{t}_i$ and the angle to change is denoted α_i. The advantages with CCD are simplicity, high

computational speed and no occurrence of singularities, but at the same time you have drawbacks as the possibility of several iterations through the entire structure and very large, undesirable joint changes (Mukundan, 2008).

As one will get the impression of from the methods listed above, the relationship between joint velocities and Cartesian velocities, through the *Jacobian*, is very useful when deriving IK. While the CCD method, and similar recursive or locally iterative methods, are very, very fast, they are only heuristically correct, too unpredictable and only found to be implemented in robotic chains - mostly n-chains with symmetric properties. The latter often due to geometrical analysis of a complex, non-symmetric robotic structures being severely intricate. In state of the art literature on the prospect of a mobile robot in a kinematic system with its manipulator, a methodology derived on the basis of the DLS or an *Extended Jacobian Method* is the only attained one, and elaborated by the likes of (Antonelli and Antonelli, 2014, From et al., 2014). The properties of the vehicle-manipulator system will be further accounted for in Section 2.4. For reasons outlined above, the focus on this thesis will concentrate on the DLS method, and further enhancements of IK based on this approach.

The Jacobian Control Method

As defined in (Sciavicco and Siciliano, 1988), there exist alternatives to the normal relationship between joint space and Cartesian space,

$$\mathbf{x} = f(\mathbf{q}). \tag{2.19}$$

Where $\mathbf{x}_{6x1} := [p_x, p_y, p_z, r_x, r_y, r_z]^T$, $\mathbf{q}_{nx1} = [\theta_1, d_2, \theta_2..\theta_n,]^T$ and f is a known, continuous nonlinear function. Since solving this relationship in terms of $\mathbf{q}$, has the drawback of mapping several $\mathbf{q}$'s to every $f^{-1}(\mathbf{x})$, one must try to relate with other techniques. By basing the relationship on joint velocities $\dot{\mathbf{q}}$ and Cartesian velocities $\dot{\mathbf{x}}$, we derive the so-called *Jacobian control method* (JCM) for redundant manipulators (Whitney, 1972).

$$\dot{\mathbf{q}} = J^{-1}(\mathbf{q})\dot{\mathbf{x}} \tag{2.20}$$

In this case, with a robotic chain of 9 joints, the Jacobian will be a 6×9-matrix, and therefore not invertible. We can resolve this implication by several methods, all very well documented throughout literature of iterative solutions to IK. This thesis will undertake the method of approximating the inverse Jacobian by the *Moore-Penrose* inverse, also called *pseudoinverse* (Buss, 2004). This leads to

$$\dot{\mathbf{q}} = J^{\dagger}(\mathbf{q})\dot{\mathbf{x}} \tag{2.21}$$

where $J^{\dagger} = J^T(JJ^T)^{-1}$ is the pseudoinverse of J at full rank. The pseudoinverse will often experience instability close to singularities. In mathematical terms, the pseudoinverse becomes ill-conditioned, which we interpret physically as giving unrealistically high input velocities to our system (Perng and Hsiao, 1999).

From equation (2.21) we can develop the error dynamics to make the difference between desired position and orientation, and the end effector's actual position and orientation, converge towards zero.

$$\mathbf{e} := \mathbf{x_d} - \hat{\mathbf{x}}$$

where $\hat{\mathbf{x}}$ is given by $f(\hat{\mathbf{q}})$, which leads to

$$\Delta\mathbf{q} = J^{\dagger}\mathbf{e}. \tag{2.22}$$

This is effectively a form of linearization, where Euler integration of the change in joint values $\Delta\mathbf{q}$, with the initial joint condition, eventually will yield the joint values giving the desired configuration $\mathbf{x_d}$. For the error dynamics to converge appropriately, we need to apply a proportional gain to the error dynamics, defined α (Klein et al., 1983).

The Jacobian control method can be improved to further exploit the redundant degrees of freedom. The redundant degrees of freedom provides mobility in joint space while the end-effector keeps moving towards or stays at the desired speed of zero. In theoretical mathematics this is denoted as *null space*. One can operate within the null space of $\mathbf{J}$ to prevent collision (if some possible solutions cause collision with physical surroundings) or it can be used to avoid singular configurations (by perturbation e.g.). Figure 2.8 illustrates the concept of the null space of $\mathbf{J}$ in a delightful manner.

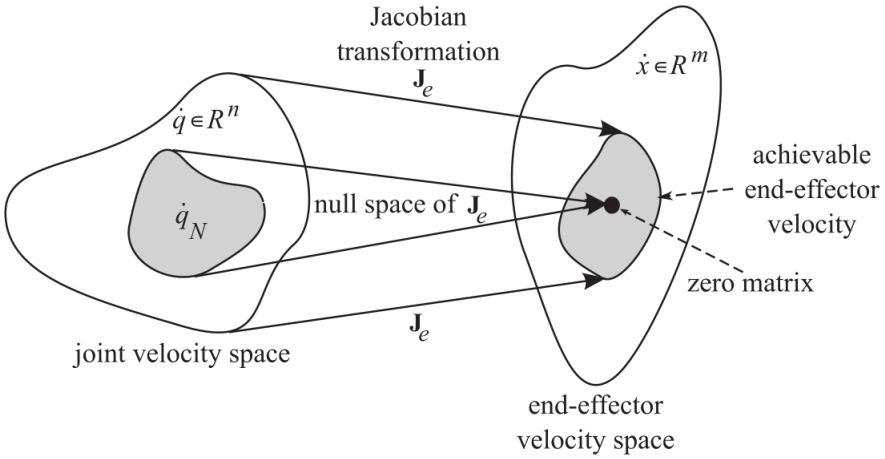

Figure 2.8: The Jacobian matrix J_e maps the joint velocity space onto the end-effector volcity space (Fahimi, 2008).

Joint Limitation

Since our system have defined limits for several joints, we have to handle situations where the JCM wants to exceed them. This is sometimes referred to as *Joint Limit Avoidance* (JLA). We look towards the usage of a weighted function with inequality constraints (Fahimi, 2008). This means that we want to weight each joint velocity

contribution according to their proximity to their respective joint limit. We define the weighted, diagonal matrix $\mathbf{C}$:

$$\mathbf{C} := \begin{bmatrix} c_{11} & & 0 \\ & \ddots & \\ 0 & & c_{99} \end{bmatrix}, \tag{2.23}$$

where each diagonal element is defined as a weight between 0 and 1, reflecting a reducing function within a user set domain around $q_{i,max}$, the upper joint limit, and $q_{i,min}$, the lower joint limit. To include this functionality in JCM, we define the limitation of joints as an *additional task* $\mathbf{z}$, with an additional control objective $\dot{\mathbf{z}}_d = 0$, containing all the joint variables $\mathbf{q}$.

$$\mathbf{z} := \begin{bmatrix} q_1 \\ d_2 \\ \vdots \\ q_9 \end{bmatrix}, \quad J_c = \frac{\partial \mathbf{z}}{\partial \mathbf{q}} = \mathbf{I}, \tag{2.24}$$

where $\mathbf{J_c}$ is the Jacobian of the joint limitation. This implies a matrix multiplication of the weighted matrix into (Eq.: 2.22), which gives us

$$\Delta\mathbf{q} = \mathbf{C}\mathbf{J}^{\dagger}\mathbf{e}, \tag{2.25}$$

Damped Least squares

As mentioned in Sub-Section (2.2.4), there is one possibly high-impact downside to the usage of a pseudoinverse for solving IK: singualrities. Close to the singularities, the *JCM* can cause very large changes in joint angles, as response to small target configuration changes. If the end-effector is exactly at a singularity, then the pseudoinverse will loose rank and yield zero change in joint angles (Eldén, 1982).

To avoid many of the described problems with singularities, we implement the *Damped Least Squares* method according to (Buss, 2004). We go about finding the change in joint values $\Delta\mathbf{q}$ that can minimize the quantity

$$||J\Delta\mathbf{q} - \mathbf{e}||^2 + \lambda^2||\Delta\mathbf{q}||^2 = || \begin{pmatrix} J \\ \lambda I \end{pmatrix} \Delta\mathbf{q} - \begin{pmatrix} \mathbf{e} \\ 0 \end{pmatrix} ||, \tag{2.26}$$

with $\lambda > 0$ as a damping constant. Equating out this expression will give

$$\begin{pmatrix} J \\ \lambda I \end{pmatrix}^T \begin{pmatrix} J \\ \lambda I \end{pmatrix} \Delta\mathbf{q} = \begin{pmatrix} J \\ \lambda I \end{pmatrix}^T \begin{pmatrix} \mathbf{e} \\ 0 \end{pmatrix} \rightarrow \tag{2.27}$$

$$(J^T J + \lambda^2 I)\Delta\mathbf{q} = J^T\mathbf{e}. \tag{2.28}$$

The term containing the Jacobian, which needs to be inverted, is shown to be non-singular by (Buss, 2004), which leads us to the equation for finding joint value changes

$$\Delta\mathbf{q} = (\mathbf{J}^T\mathbf{J} + \lambda^2 I)^{-1}\mathbf{J}^T\mathbf{e}. \tag{2.29}$$

Singular Value Decomposition

Analysis of how λ will impact the numerical behavior of the DLSM, have been performed in a brilliant manner by the likes of (Egeland et al., 1991) and (Buss and Kim, 2005). In virtually all walks of life we are subjected to *trade-offs*, and the DLSM is no different. We want IK solutions as accurate as possible, at the same time as we want to handle singularities to the highest standards of robustness. These two intentions will counter each other by the mathematical derivations in (Chiaverini et al., 1991). A singular value decomposition (SVD) is a very powerful tool for understanding the nature of singularities, since they bring forth the singular values explicitly (Golub and Reinsch, 1970).

The Jacobian can be decomposed into SVD form.

$$\mathbf{J} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^T, \tag{2.30}$$

which leads to

$$\begin{aligned}\mathbf{J}^\dagger = \mathbf{J}^T(\mathbf{J}\mathbf{J}^T)^{-1} &= \mathbf{V}\mathbf{\Sigma}^T\mathbf{U}^T(\mathbf{U}\mathbf{\Sigma}\mathbf{V}^T\mathbf{V}\mathbf{\Sigma}^T\mathbf{U}^T)^{-1} \\ &= \mathbf{V}\mathbf{\Sigma}^T\mathbf{U}^T(\mathbf{U}\mathbf{\Sigma}\mathbf{\Sigma}^T\mathbf{U}^T)^{-1} \\ &= \mathbf{D}^\dagger,\end{aligned}$$

where the diagonal entries of $\mathbf{D}^\dagger$ are $d_{i,i}$. This gives us the following property of the pseudoinverse

$$\Sigma^\dagger = \begin{cases} \frac{1}{\sigma_{i,i}}, & d_{i,i} \neq 0 \\ 0. \end{cases} \tag{2.31}$$

2.3 Physical Platform

2.3.1 Humanoid Platform

The two prime requirements for a physical platform was great mobility of the robot body, and at least four degrees of freedom in the manipulator. When testing different solution strategies with many possible sources of failure, the need for an easily accessible platform with a low threshold for testing, quickly became apparent as well. By these parameters, the chosen platform became Aldebaran's Nao, which have an extensive API at the same time as a land based robot have a much lower threshold for testing (than for instance an ROV). Such humanoid robots have become increasingly popular in recent times, both because of reasonable prices - which make them accessible for a larger audience - and their ability to perform complex tasks. The Nao manufacturer, Aldebaran, has created an Aldebaran community for exchanging experiences, software and algorithms, which make it a very suitable platform to explore the nature of more advanced and robust IK solutions.

2.3.2 Physical Properties

To be able to develop the kinematic control of our humanoid test robot, we first need to obtain the geometrical and physical properties of the the robot - all taken from (Aldebaran, 2015a).

Degrees of Freedom

Nao has a total of 25 degrees of freedom, distributed among the four limbs and the head. The physical parts of Nao in our field of interest is; each feet containing 4 degrees of freedom and each arm containing 5 degrees of freedom. Every degree of freedom is represented by a revolute joint with joint constraints, and they are situated - for a leg and an arm respectively - according to Table 2.3 and 2.1. For the DOF nomenclature to make sense, we need to affix each joint with a coordinate frame relative to a floored base frame, corresponding to the following approach: At a zero pose for each joint, all frames are oriented equally, with roll rotation around the x-axis, pitch around the y-axis and yaw around the z-axis.

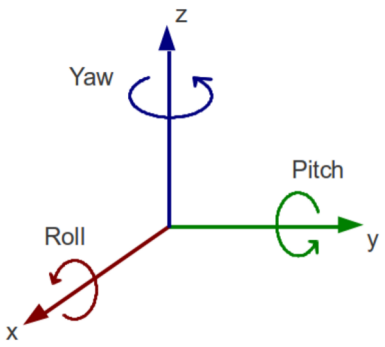

Figure 2.9: Global coordinate system affixed to zero positioned joints

Embarking on the left arm's physical features first, we have the following joints and constraints (All figures are gathered from (Aldebaran, 2015a)):

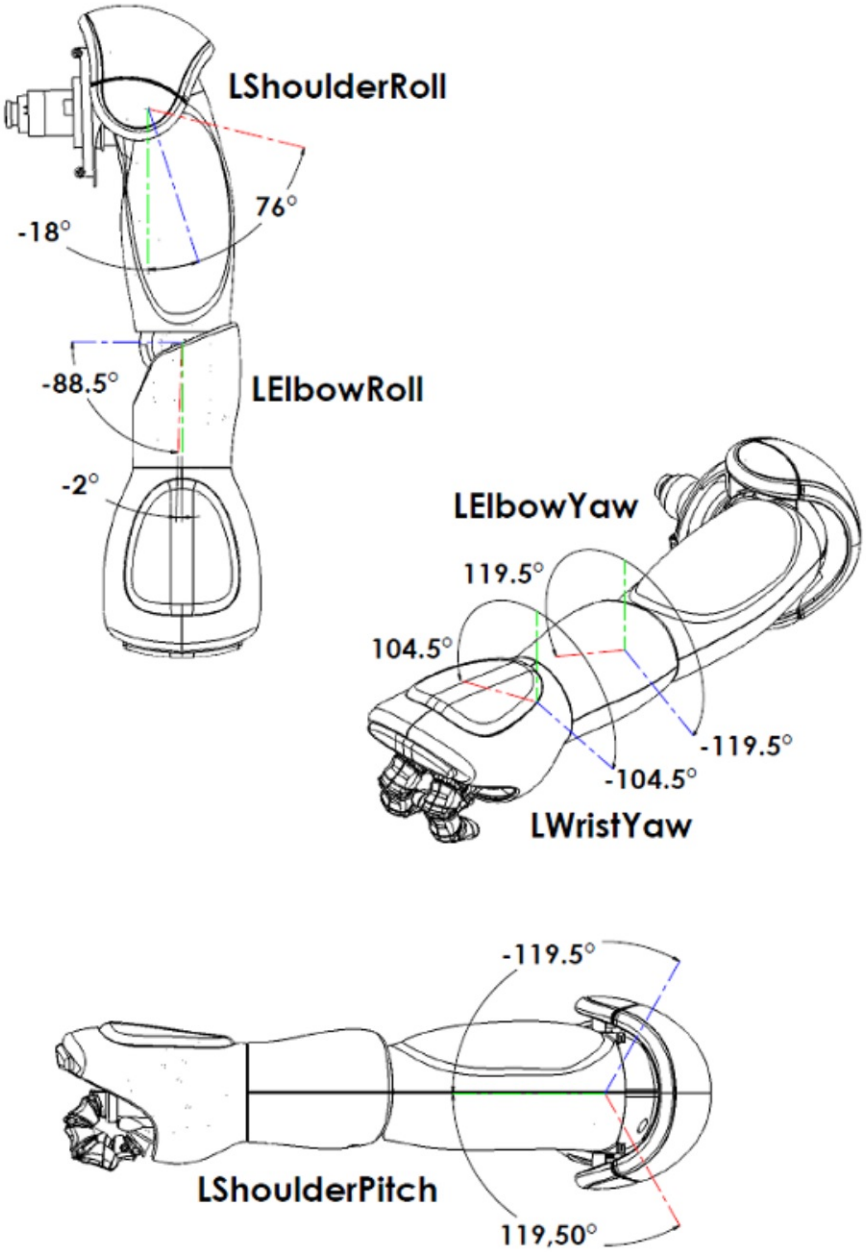

Figure 2.10: Nao's left arm

Joint name	**Motion**	**Min value [deg]**	**Max value [deg]**
LShoulderPitch	Shoulder joint, front and back	−119.5	119.5
LShoulderRoll	Shoulder joint, right and left	−18	76
LElbowYaw	Shoulder joint twist	−119.5	119.5
LElbowRoll	Elbow joint	−88.5	−2
LWristYaw	Wrist joint	−104.5	104.5
LHand	Hand, open and close	Open	Closed

Table 2.1: The joints of Nao's left arm

with the left arm measures in the figure below

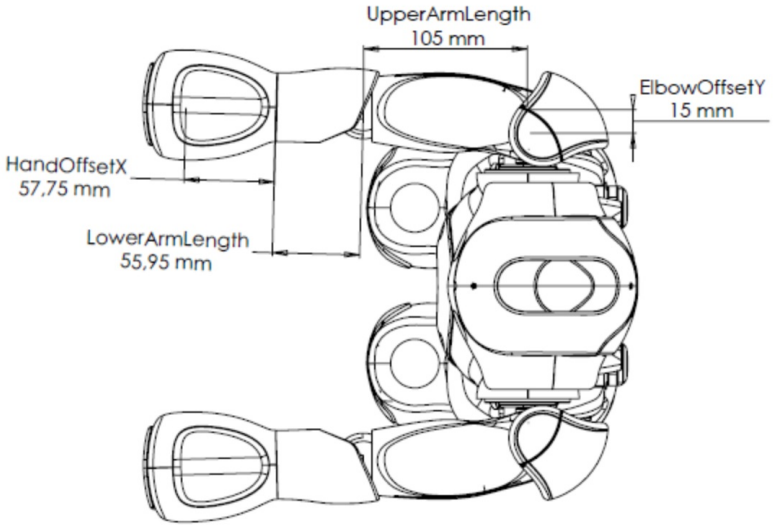

Figure 2.11: Nao seen from above, with streight arms

The corresponding measures, joints and constraints for the left leg follows.

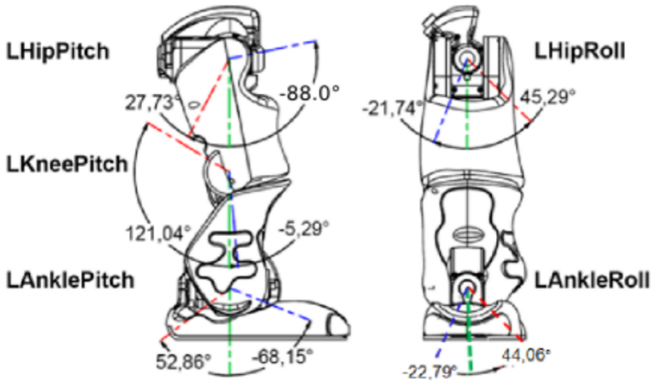

Figure 2.12: The joints of Nao's left leg

Joint name	**Motion**	**Min value [deg]**	**Max value [deg]**
LHipRoll	Hip joint right and left	−21.74	45.29
LHipPitch	Hip joint front and back	−88.00	27.73
LKneePitch	Knee joint	−5.29	121.04
LAnklePitch	Ankle joint front and back	−68.15	52.86
LAnkleRoll	Ankle joint right and left	−22.70	44.06

Table 2.2: The joints of Nao's left leg

As one can identify in Figure 2.13, Nao is completely symmetric about the inertial $x - z$-plane, which makes illustrations and measurements of only the left limbs sufficient for modelling the entire robotic structure.

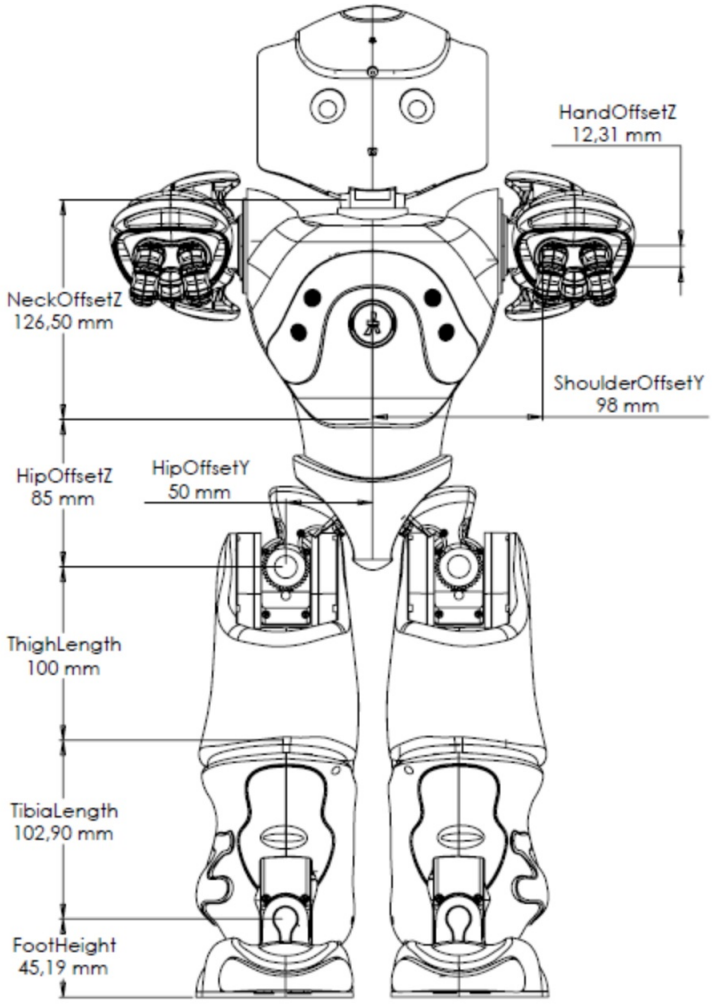

Figure 2.13: Nao seen from the front

From the front-viewed figure of Nao, we can determine the following important measurements:

Limb or body part	**Length [mm]**
Leg	45.29
LHipPitch	27.73
LKneePitch	121.04
LAnklePitch	52.86
LAnkleRoll	44.06

Table 2.3: Measurements of Nao's left leg

Sensing and Connectivity

It is worth mentioning that Nao has a lot of capabilities which can be explored in the world of sensing and connectivity. One can connect to the robot by using a local WiFi network, thereafter using the pre-made API with either Python or C++ to establish an idle connection to send commands. The commands and procedures

for establishing communication and sending the IK data to the robot is attached in Appendix C.

If sensory information is desirable for doing a specific task containing some sort of decision making, Nao has some capabilities of interest. There are sensors for physical "feel" in his head, feet and hands. Furthermore it has two high resolution cameras and a sonar for graphical perception and orientation of the surroundings. Lastly, it has an inertial unit and cooperating software in a "safety mode" that enables it to maintain its balance during locomotion or other greater motions.

2.3.3 Underwater Platform

The physical platform in mind when the objective of this thesis nurtured, was NTNU's most sophisticated underwater robot: ROV Minerva. It has a five functional manipulator and maneuverability in all six degrees of freedom. The expenditures following an expedition with this sort of ROV, and the low availability, makes this platform unfitting for this initial research into the matter, but it serves as a good inspiration or future implementation of the developed functionality.

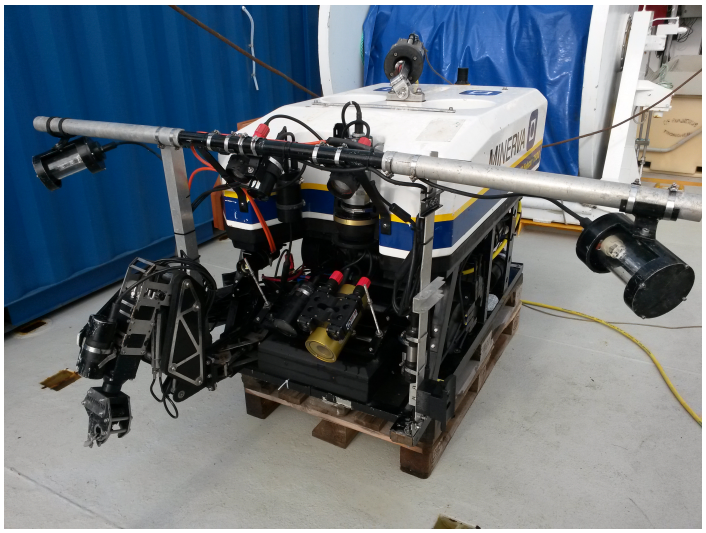

Figure 2.14: The NTNU research vessel ROV Minerva, (NTNU-AUR-lab, Accessed 11.02.16)

2.4 Mobile Robot-Manipulator system

2.4.1 Reference Frames

A very extensive part of newer literature on the kinematic control of humanoid robots is concentrated on what can be denoted as *partial limb control*. The robot is defined with a base coordinate system, defined $\mathscr{F}_b$ (also called *base-frame*) (See Figure 2.15), from where every limb movement is referenced. The normal practice is to define this global coordinate system in the torso, or another abdominal point of peculiarity, which functions as the base frame for every "limb manipulator". Hence one will obtain easy defined manipulator chains from torso to head, torso to hand and torso to feet. Individually, these may be analytically or numerically solved

depending on redundancy and requirements, where in the case of Nao one can solve every limb in closed form.

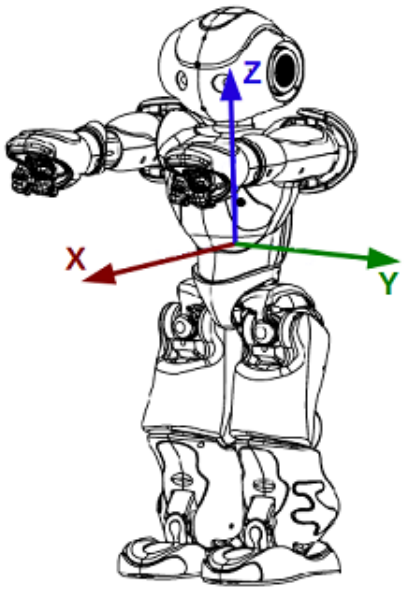

Figure 2.15: The base reference frame $\mathscr{F}_b$ of Nao

For mathematical simplicity, the theoretical work of developing the forward kinematics will use the reference frame of Nao's torso. Hence every desired configuration for the manipulator (one of Nao's hands) needs to be given in torso reference frame. In practical application of this system, the robot base-frame and the objective configuration, $\mathscr{F}_t$, will need to be tracked in a *global coordinate system*, $\mathscr{F}_0$, as the objective configuration then can be transformed from the global coordinate system to the base-frame by a homogeneous transformation matrix, $\mathbf{T}$ (See Figure 2.16 and Equation 2.32).

$$\mathbf{T}_t^b = (T_b^0)^{-1} T_t^0 \tag{2.32}$$

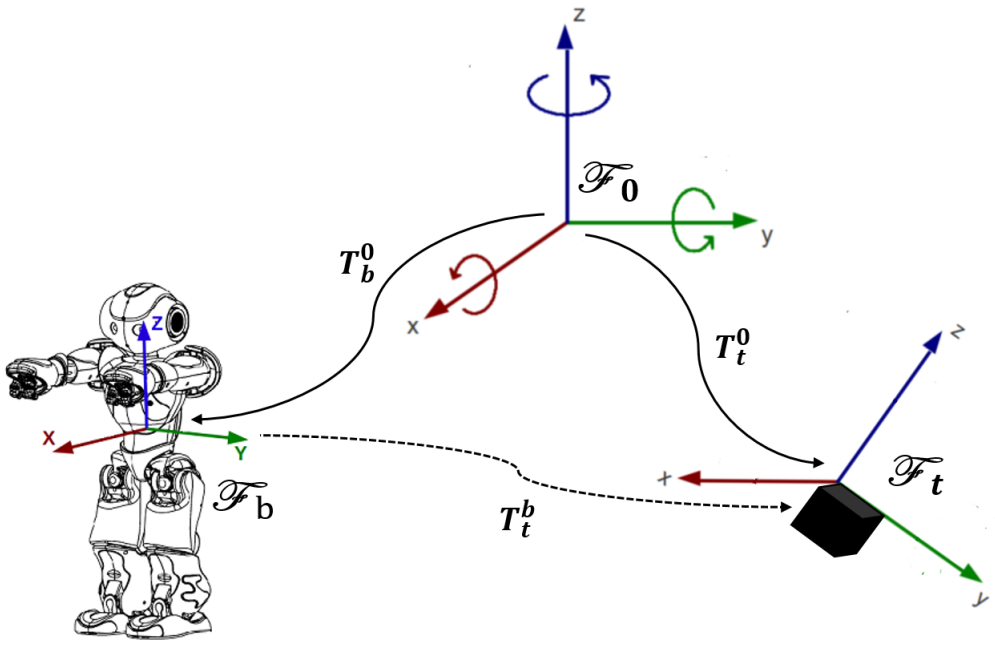

Figure 2.16: An overview of coordinate frames and their relationship

If Nao is operating without external sensory equipment, he can use a preconfigured *world reference frame*. His internal sensors will then try to reference every move and orientation from a configuration defined at the instance of receiving the first command.

2.4.2 Mobile Robot-Manipulator Kinematics

The ability to coordinate a hand operation (e.g. gripping an object) with the body-posture is intrinsic to advanced human execution of a miscellaneous of tasks. Take say a normal operation of picking up something from the floor. Naturally, one will probably bow or crouch to get the object within grasp. A replication of this movement with a humanoid robot will not function under normative kinematic control, because it defies the way of always doing single limb movements based on torso center coordinates. This points directly to the objective of this thesis; to explore the dexterous possibilities of a mobile robot-manipulator system when the whole body works as one kinematic system to achieve desired configurations. If one looks to (From et al., 2014) and (Antonelli and Antonelli, 2014) you have extensive, recent scholarly literature on the vehicle-manipulator system, with theories adoptable to a humanoid robot and further on explored for a variety of mobile robot-manipulator objectives.

2.5 Fuzzy Logic

Fuzzy logic appeared as a term after the famous *Fuzzy Sets* (Zadeh, 1965), in which the new computing approach of "degrees of truth" became recognizable, in contrast to the conservative binary reality. In the realm of mathematical sets and conjunctions, an element $x \in X$, where X is a crisp set, will have a "degree of membership" in a fuzzy set $A \subseteq X$ (Hajek, 2010). By this, one can dare to suggest that normal

binary logic is a special case of fuzzy logic, where only the absolute possibilities are present. The standard procedure for undertaking the fuzzy approach is:

- Fuzzify all input values into fuzzy membership functions.
- Execute all applicable rules in the rulebase to compute the fuzzy output functions.
- De-fuzzify the fuzzy output functions to get "crisp" output values.

The motivation for creating a fuzzy logical mapping to solve the IK, is somehow similar to other literature on fuzzy logic in robotics; it is a way to enable heuristic imitation of human execution (Xu et al., 1991). Fuzzy logic seems closely related to the way our own brains function, with a summation of weighted sub-problems according to experience, into a total probabilistic decision being made. Problems occurring with traditional kinematic control of mobile robots, like manipulator over extension, unnecessary locomotion and time-costly trajectory calculation, can be severely mitigated by developing some human "decision making" at the instance of a received target configuration. This introduces a new term being highly relevant in today's literature; *Decision Support Systems* (DSS). DSS is a kernel of rules that is organized to provide intelligent services consistent of criteria with subjective or objective attributes, working on the given input (Fodor and Fullér, 2014). The field of fuzzy logic and DSS leads indispensably towards data mining and deep-learning. One cannot plunge into the domain of fuzzy logic and smart decision making without mentioning the state of the art technology, which is the application of neural networks. Neural networks is an ubiquitous science field leapfrogging at the moment of writing. Briefly introduced, as it goes beyond the scope of this work - but being highly relevant nonetheless, neural networks are a collection of nodes, weighted objectively by extensive observations of successful executions (e.g. of grasping operations) through backtracking or convolution, to pickup significant features of input to induce an appropriate action. A brilliant example is the deep-learning robots on full scale, trying to grasp different objects, set up by Google to create a neural network with "knowledge" of how to perform any needed grasp operation for the respective manipulator type.

Figure 2.17: Googles deep-learning robot arms (Courtesy to Google Research Blog)

This thesis will try to facilitate a fuzzy decision making system in the closed control loop of the IK, but based on subjective weighting from the experience of our own way of reaching different configurations.

2.6 Software Platform

This thesis will utilize Python as the main programming platform for software implementation. This is due to several factors:

- Python is a free, interpreted, extensive programming platform that can run almost everywhere (Python-Foundation, 2016).
- Python can easily communicate with a lot of electronic devices and microcontrollers.
- Nao provides an API that make functionality very easy with Python (Aldebaran, 2015b).
- Python has powerful tools for visualizing results of simulations.
- Python has the possibility of solving high-level mathematical problems, with packages as NumPy to handle two-dimensional arrays and linear algebra (Numpy-Developers, Accessed 25.02.16).

The thesis will also take advantage of MATLAB as a platform for visualizing some results, and illustrating the system components and dataflow in block diagrams. This will just be with regards to presentation, so that MATLAB is not a preliminary requirement for using the resulting program.

Chapter 3

Kinematic Modelling

Before any forward kinematics can be obtained, one needs to create the kinematic model of the robot. Based on the theory in the previous chapter, the kinematic model will be constructed from three base elements; rotational joints, prismatic joints and rigid limbs. From documentation provided by Aldebaran, this is already established for every physical joint in Nao:

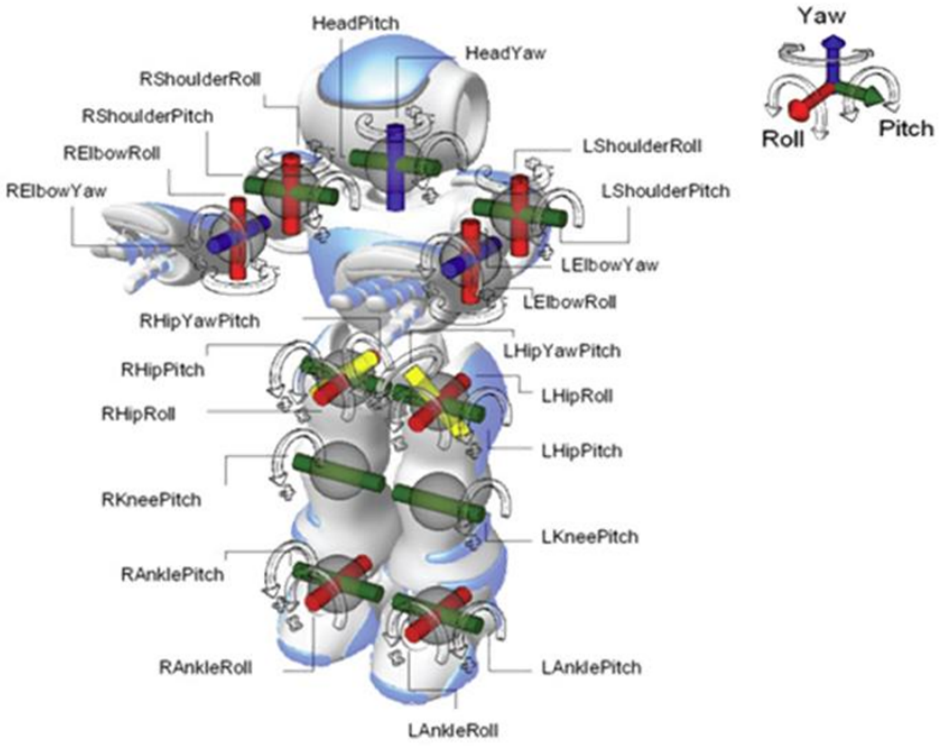

Figure 3.1: Nao's joints affixed with axes of rotation (Aldebaran, 2015a)

As will be presented in this chapter, the theoretical kinematic model of Nao's arm can be implemented directly from the information supplied in the data-sheets, but the theoretical kinematic model of the the body will be implemented in a different manner.

3.1 Rigid Body Kinematics

3.1.1 Establishing the Simplified Kinematic Model

From Figure 3.1, the leg joints are simplified down to two degrees of freedom, represented by one prismatic and one rotational joint. The prismatic joint is established because the height of the end-effector base (the left shoulder) is the only important impact from the lower legs. Moreover, the height supplied by the joint values in the lower legs can easily be transformed into joint values, in addition to built-in functionality from the Nao API which perform the same transformations. Further on, the hip pitch (what trivial language would denote as bending forward or backwards) is of importance, and is therefore represented by a revolute joint. To model the locomotion, inspiration has been granted by human behavior. A sane person would turn to the direction he or she intends to walk, then walk to the desired end-point and possibly turn again to reach an optimal body posture for performing a task (e.g. picking something up). This corresponds to one rotational joint, then a prismatic one and lastly another rotational. The only missing DOF is in roll direction, which Nao actually has to some degree, but it was considered too insignificant to be included in the model.

Using a combined equation- and graphical-based mathematical program called *GeoGebra*, the rigid body kinematics can be modeled in 3D, with proper labeling to better illustrate the line of thought. The resulting 3D model is pictured below in a simplified way, from the first joint q_1 to the shoulder joint q_6.

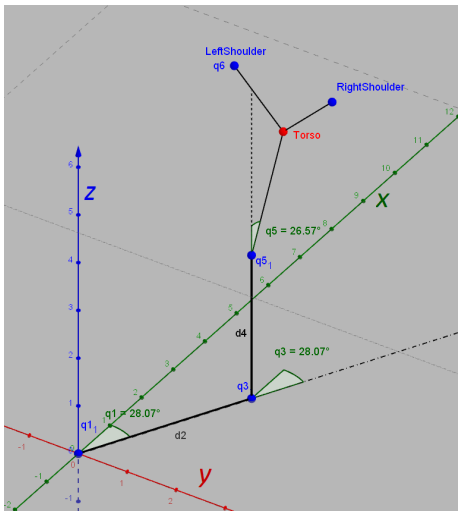

Figure 3.2: Simplified kinematic model of Nao's body, viewed from behind to the right

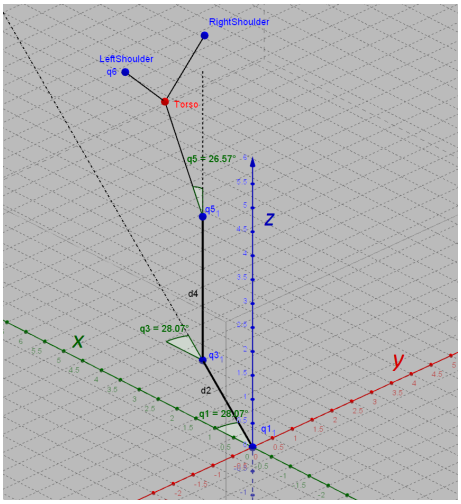

Figure 3.3: Simplified kinematic model of Nao's body, viewed from behind to the left with grid

The initial position is at the origin, and sequentially from there q_1 represents that the robot turned 28^o to the right, d_2 that it has locomoted approximately 3.3 in the fictitious model units, q_3 a turn of 28^o back to the left (parallel to the x-axis), d_4 that the legs are straightened up to lift q_5 to about 3 fictitious units and q_5, representing the hip-pitching joints, indicates that the body is bent forward at 26.6^o. Substantiated by the physical properties of Nao, a table with joint constraints for the rigid robot body can be generated.

Joint name	**Motion**	**Min value**	**Max value**	**Unit**
q_1	Walking direction	-180	180	deg
d_2	Walking distance	$-\infty$	∞	mm
q_3	Body posture in yaw	-180	180	deg
d_4	Leg elevation of hip	-88.5	-2.0	mm
q_5	Hip pitch	-88.0	27.73	deg

Table 3.1: The modeled joint constraints of the mobile robot body

3.1.2 Forward Kinematics

The methodology provided by (Craig, 2005) can be used to establish the Denavit-Hartenberg parameters for the rigid body.

n_i	α_{i-1}	a_{i-1}	d_i	θ_i
1	0	0	0	θ_1
R_1				
2	90°	0	d_2	0
3	$-90°$	0	0	θ_3
4	0	0	d'_4	0
R_4				
5	90°	0	0	θ_5
R_5				
A_5				

Table 3.2: Denavit-Hatenberg paramters for the mobile robot body

where $R_1 = Rz(90°)$, $R_4 = Rz(90°)$, $R_5 = Rz(180°)Rx(90°)$ and $A_5 = A(0, 0, z_{torso})$. Notice also that $d'_4 = d_4 + h$, where h is the minimum height of the robot hip-pitching joint, i.e. when it is crouching, and d_4 is a prismatic joint. Table 3.2 gives us the needed information to compute the transformation matrices, iteratively from $q_1 \rightarrow q_5$, in compliance with the general transformation matrix between two joints in Equation 2.6.

$$T_1^0 = \begin{bmatrix} c\theta_1 & -s\theta_1 & 0 & 0 \\ s\theta_1 & c\theta_1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad T_2^1 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & -1 & -d_2 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad T_3^2 = \begin{bmatrix} c\theta_3 & -s\theta_3 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -s\theta_3 & -c\theta_3 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{3.1}$$

$$T_4^3 = \begin{bmatrix} 0 & -1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & d'_4 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad T_5^4 = \begin{bmatrix} c\theta_5 & -s\theta_5 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ s\theta_5 & c\theta_5 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{3.2}$$

where $sin(\theta_i)$ is denoted as $s\theta_i$, $cos(\theta_i)$ is denoted as $c\theta_i$ and $d_2(t)$ and $d'_4(t)$ are time varying lengths because of the prismatic joints simplified to d_2 and so forth.

By matrix multiplication, $T_1^0 R_1 ... T_5^4 R_5 A_5$, we obtain the forward kinematic transformation of the entire system

$$T_{torso}^{base} = \begin{bmatrix} r_{11} & r_{12} & r_{13} & p_x \\ r_{21} & r_{22} & r_{23} & p_y \\ r_{31} & r_{32} & r_{33} & p_z \\ 0 & 0 & 0 & 1 \end{bmatrix}, \tag{3.3}$$

where

$$\begin{aligned}
r_{11} &= c\theta_5(c\theta_1 s\theta_3 + c\theta_3 s\theta_1) \\
r_{12} &= c\theta_1 c\theta_3 - s\theta_1 s\theta_3) \\
r_{13} &= s\theta_5(c\theta_1 s\theta_3 + s\theta_1 c\theta_3) \\
r_{21} &= -c\theta_5(c\theta_1 c\theta_3 - s\theta_1 s\theta_3) \\
r_{22} &= c\theta_1 s\theta_3 + c\theta_3 s\theta_1 \\
r_{23} &= -s\theta_5(c\theta_1 c\theta_3 - s\theta_1 s\theta_3) \\
r_{31} &= -s\theta_5 \\
r_{32} &= 0 \\
r_{33} &= c\theta_5 \\
p_x &= d_2 s\theta_1 + z_{off} s\theta_5(c\theta_1 s\theta_3 + s\theta_1 s\theta_3) \\
p_y &= -d_2 c\theta_1 - z_{off} s\theta_5(c\theta_1 c\theta_3 - s\theta_1 s\theta_3) \\
p_z &= d_4' z_{off} c\theta_5.
\end{aligned}$$

Table 3.3: Forward kinematic solution to a model of Nao's lower body movement

The validity of our forward kinematics can easily be checked, by insertion of joint control values with known mapping to Cartesian space, using the FK program in Appendix B.

$$\begin{aligned}
ForwardKinematics(\frac{-\pi}{2}, 200, \frac{\pi}{2}, 15, \frac{\pi}{6}) \Rightarrow P_x = 42.5\,[mm], \quad P_y = -200.0\,[mm], \\
P_z = 229.6\,[mm], \quad \phi = 0\,[rad], \quad \theta = 0.524\,[rad], \quad \psi = 0\,[rad].
\end{aligned}$$

With $z_{torso} = 85\,[mm]$, we can confirm this by performing a simple geometrical calculation on paper or in *GeoGebra*.

3.2 Manipulator Kinematics

On the premise of the Denavit-Hartenberg parameters in (Craig, 2005) and in Figure 3.1, we can establish the DH for Nao's left arm. An important notice is duly taken to our simplification of the two yawing joints "LElbowYaw" and "LWristYaw" from Figure 2.10 into *one* yawing joint. This is undertaken without complications because both joints have intersecting and parallel axes of rotation. In practice, this is complied by with a predominance for "LElbowYaw" rotation, and a possible further rotation from "LWristYaw" if necessary. It is also worth mentioning that q_6, the *shoulder pitch joint*, is modelled to be zero when having 90^o to the upper body. This is to follow the convention of zero rotation about the base-fixed reference frame when all joints are zero. We have

q_i	α_{i-1}	a_{i-1}	d_i	θ_i
$A_{shoulder}$				
6	-90	0	0	θ_1
7	$90°$	0	0	θ_2
$R_z(90°)$				
8	$90°$	b_3	l_3	θ_3
9	$-90°$	0	0	θ_4
$R_z(-90°)$				
A_{hand}				

Table 3.4: Denavit-Hatenberg Paramters for the left arm

where $R_7 = R_z(90°)$, $R_9 = R_z(90°)$, $A_{shoulder} = A(0, y_{shoulder}, z_{shoulder})$ and $A_{hand} = A(x_{hand}, 0, z_{hand})$ are affine pure rotational matrices about the indicated axes and affine pure translational matrices in the indicated directions. Applying the tabular parameters on Equation 2.6 yet another time, the transformations beneath transpires.

$$A_{shoulder} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & y_{shoulder} \\ 0 & 0 & 1 & z_{shoulder} \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad T_6^5 = \begin{bmatrix} c\theta_1 & -s\theta_1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -s\theta_1 & -c\theta_1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad T_7^6 = \begin{bmatrix} c\theta_2 & -s\theta_2 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ s\theta_2 & c\theta_2 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{3.4}$$

$$R_z = \begin{bmatrix} 1 & -1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad T_8^7 = \begin{bmatrix} c\theta_3 & -s\theta_3 & 0 & b_3 \\ 0 & 0 & -1 & 0 \\ s\theta_3 & c\theta_3 & 0 & l_3 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad T_9^8 = \begin{bmatrix} c\theta_4 & -s\theta_4 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -s\theta_4 & -c\theta_4 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}, \tag{3.5}$$

where $y_{shoulder}$ and $z_{shoulder}$ denotes the distance from the robot's *torso base-frame* to the shoulder joints. By matrix multiplication , $A_{shoulder}T_6^5...T_9^8R_9A_9$, we obtain the forward kinematic transformation of the manipulator:

$$T_{hand}^{torso} = \begin{bmatrix} r_{11} & r_{12} & r_{13} & p_x \\ r_{21} & r_{22} & r_{23} & p_y \\ r_{31} & r_{32} & r_{33} & p_z \\ 0 & 0 & 0 & 1 \end{bmatrix}, \tag{3.6}$$

where

$$
\begin{aligned}
r_{11} &= c\theta_9(s\theta_6 s\theta_8 + c\theta_6 c\theta_7 c\theta_8) - c\theta_6 s\theta_7 s\theta_4 \\
r_{12} &= -s\theta_9(s\theta_6 s\theta_8 + c\theta_6 c\theta_7 c\theta_7) - c\theta_6 c\theta_9 s\theta_7 \\
r_{13} &= c\theta_8 s\theta_6 - c\theta_6 c\theta_7 s\theta_8) \\
r_{21} &= c\theta_7 s\theta_9 + c\theta_8 c\theta_9 s\theta_7 \\
r_{22} &= c\theta_7 c\theta_9 - c\theta_8 s\theta_7 s\theta_9 \\
r_{23} &= -s\theta_7 s\theta_8 \\
r_{31} &= c\theta_9(c\theta_6 s\theta_8 - c\theta_7 c\theta_8 s\theta_6) + s\theta_6 s\theta_7 s\theta_9 \\
r_{32} &= c\theta_9 s\theta_6 s\theta_7 - s\theta_9(c\theta_6 s\theta_8 - c\theta_7 c\theta_8 s\theta_6) \\
r_{33} &= c\theta_6 c\theta_8 + c\theta_7 s\theta_6 s\theta_8 \\
p_x &= z_{hand}(c\theta_8 s\theta_6 - s\theta_6 c\theta_7 s\theta_8) + l_8 s\theta_6 + b_3 c\theta_6 + c\theta_7 \\
p_y &= y_{shoulder} + b_3 s\theta_7 - z_{hand} s\theta_7 s\theta_8 \\
p_z &= z_{shoulder} + z_{hand}(c\theta_6 c\theta_8 + c\theta_7 s\theta_6 s\theta_8) + l_3 c\theta_6 - b_3 c\theta_7 s\theta_6.
\end{aligned}
$$

Table 3.5: Forward kinematic solution to Nao's left arm

The variables $b_3 = 15$, $l_3 = 105$, $y_{shoulder} = 98$, $z_{shoulder} = 100$ and $z_{hand} = -113.7$ represents "ElbowOffset", "UpperArmLength", distance from torso to shoulder in y, distance from torso to shoulder in z and "LowerArmLength" plus "HandOffset" from Figure 2.11.

3.3 System Forward Kinematics

The forward kinematics of the "robot body"-"left arm"-system is conjoined accordingly:

$$
T_{hand}^{base} = T_{torso}^{base} T_{hand}^{torso} = \begin{bmatrix} r_{11} & r_{12} & r_{13} & p_x \\ r_{21} & r_{22} & r_{23} & p_y \\ r_{31} & r_{32} & r_{33} & p_z \\ 0 & 0 & 0 & 1 \end{bmatrix}, \tag{3.7}
$$

The resulting total forward kinematics is obtained through symbolic programming, but it is left in Appendix A because of its extensiveness. The solution can nonetheless be verified in the exact same manner as the robot body kinematics.

$$
\begin{aligned}
ForwardKinematics(0, 10, 0, 15, 0, \frac{-\pi}{2}, 0, 0, \frac{-\pi}{6}) \Rightarrow \\
P_x = 213.5\,[mm], \quad P_y = 56.2\,[mm], \quad P_z = 331.2\,[mm], \\
\phi = 0.0\,[rad], \quad \theta = 0.0\,[rad], \quad \psi = -0.524\,[rad]
\end{aligned}
$$

$$
\begin{aligned}
ForwardKinematics(\frac{-\pi}{2}, 100, \frac{\pi}{2}, 50, 0, \frac{-\pi}{2}, \frac{\pi}{4}, \frac{\pi}{6}, \frac{-\pi}{6}) \Rightarrow \\
P_x = 163.7\,[mm], \quad P_y = 122.0\,[mm], \quad P_z = 339.4\,[mm], \\
\phi = 0.463\,[rad], \quad \theta = 0.253\,[rad], \quad \psi = 0.322\,[rad]
\end{aligned}
$$

Both of the examples above are confirmed by geometric evaluation on paper.

Chapter 4

Inverse Kinematics

4.1 Establishing the Jacobian

We start with the relation between Cartesian coordinates and joint coordinates; the FK transformation from Sub-Chapter 3.3:

$$T_{hand}^{base} = \begin{bmatrix} {}^{0}_{9}r_{11} & {}^{0}_{9}r_{12} & {}^{0}_{9}r_{13} & {}^{0}_{9}p_x \\ {}^{0}_{9}r_{21} & {}^{0}_{9}r_{22} & {}^{0}_{9}r_{23} & {}^{0}_{9}p_y \\ {}^{0}_{9}r_{31} & {}^{0}_{9}r_{32} & {}^{0}_{9}r_{33} & {}^{0}_{9}p_z \\ 0 & 0 & 0 & 1 \end{bmatrix}, \tag{4.1}$$

Using the theory provided in the background theory of 2.2.2, we start by obtaining the linear velocities for a joint chain consisting of 9 joints.

$$\mathbf{J_v} = \begin{bmatrix} \frac{\partial {}^{0}_{9}p_x}{\partial q_1} & \frac{\partial {}^{0}_{9}p_x}{\partial q_2} & \cdots & \frac{\partial {}^{0}_{9}p_x}{\partial q_9} \\ \frac{\partial {}^{0}_{9}p_y}{\partial q_1} & \frac{\partial {}^{0}_{9}p_y}{\partial q_2} & \cdots & \frac{\partial {}^{0}_{9}p_y}{\partial q_9} \\ \frac{\partial {}^{0}_{9}p_z}{\partial q_1} & \frac{\partial {}^{0}_{9}p_z}{\partial q_2} & \cdots & \frac{\partial {}^{0}_{9}p_z}{\partial q_9} \end{bmatrix}_{3\times 9} \tag{4.2}$$

Further on, the more convoluted process of obtaining the angular velocities commence by separating the FK transformation into each link transformation:

$$\mathbf{T}_1^{base} = \begin{bmatrix} c\theta_1 & -s\theta_1 & 0 & 0 \\ s\theta_1 & c\theta_1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \Rightarrow \begin{bmatrix} \frac{\partial \phi}{\partial q_1} \\ \frac{\partial \theta}{\partial q_1} \\ \frac{\partial \psi}{\partial q_1} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} \tag{4.3}$$

The third column of the first rotation matrix is the first column of the angular value contributions. Furthermore, the contributions from q_2 becomes:

$$\begin{bmatrix} \frac{\partial \phi}{\partial q_2} \\ \frac{\partial \theta}{\partial q_2} \\ \frac{\partial \psi}{\partial q_2} \end{bmatrix} = \mathbf{T}_2^{base}(1:3,3) = \mathbf{T}_1^{base}\mathbf{T}_2^{1}(1:3,3) = \begin{bmatrix} {}^{0}_{2}r_{13} \\ {}^{0}_{2}r_{23} \\ {}^{0}_{2}r_{33} \end{bmatrix} \tag{4.4}$$

Doing this iteratively until the transformation of joint q_9, we obtain all nine columns of the angular Jacobian:

$$\mathbf{J}_w = \begin{bmatrix} {}^0_1r_{13} & {}^0_2r_{13} & \cdots & {}^0_9r_{13} \\ {}^0_1r_{23} & {}^0_2r_{23} & \cdots & {}^0_9r_{23} \\ {}^0_1r_{33} & {}^0_2r_{33} & \cdots & {}^0_9r_{33} \end{bmatrix}. \tag{4.5}$$

As the Jacobian for a system with nine joints is rather complicated and demands very diligent mathematical work to be obtained by hand, it is acquired through symbolic mathematical programming. A script in *MATLAB* to obtain the Jacobian and the resulting, mathematical Jacobian can be viewed in Appendix A.

4.2 The Jacobian Control Method

Now the *Jacobian Control Method* (JCM) can be developed. The relation between Cartesian changes and joint changes is

$$\mathbf{\Delta q}_{9\times1} = \mathbf{J}^{\dagger}_{9\times6}\mathbf{\Delta x}_{6\times1} = \alpha\mathbf{J}^{\dagger}_{9\times6}\mathbf{e}_{6\times1}, \tag{4.6}$$

where α is the proportional constant (or the step-size in numerical terms), and the Jacobian have to be pseudoinverse because of being non-square.

4.2.1 Closing the Control Loop

After one iteration of the JCM is performed, we obtain an incremental change in joint values, $\mathbf{\Delta q}$, which is integrated to amount a new total of joint values, $\mathbf{q}$. This new set of joint values needs to be fed back into the numerical solver for a new iteration. The numerical solver and the control system are actually two sides of the same coin, and this process is the equivalent to feedback and control of error dynamics, in the world of control theory. To follow this parallel, the output data (joint values) and the input data (Cartesian values) are in different coordinate systems, which requires a transformation of the feedback states. This transformation, from joint values to Cartesian values, is the extensively discussed forward kinematics transformation, but here an obstacle arrives. The input data is represented by $\mathbf{x} \in \mathbb{R}^{\mathbf{6}}$ while the output data from the FK is represented by $\mathbf{T} \in \mathbb{R}^{4\times4}$. Below is a block diagram of this particular instance, with the feedback transformation emphasized in red and green.

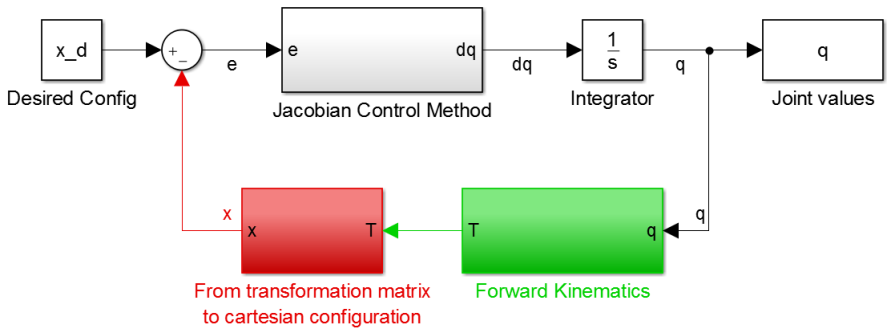

Figure 4.1: Feedback of the Jacobian Control Method

As the figure above demonstrates, there is a need for a function to transform a transformation matrix into a Cartesian configuration with Euler angles (red marking). This functionality is built-in in the case of e.g. MATLAB, but does not exist in Python, and therefor needs to be established. As outlined in Sub-Chapter 2.2.2, the following can be extracted from a transformation matrix $\mathbf{T}_{4\times4}$:

$$\mathbf{x}_{6\times1} = \begin{bmatrix} x \\ y \\ z \\ \phi \\ \theta \\ \psi \end{bmatrix}, \quad \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} \mathbf{T}_{1,4} \\ \mathbf{T}_{2,4} \\ \mathbf{T}_{3,4} \end{bmatrix}, \quad \begin{bmatrix} \phi \\ \theta \\ \psi \end{bmatrix} = \begin{bmatrix} \mathbf{T}_{1,1} & \mathbf{T}_{1,2} & \mathbf{T}_{1,3} \\ \mathbf{T}_{2,1} & \mathbf{T}_{2,2} & \mathbf{T}_{2,3} \\ \mathbf{T}_{3,1} & \mathbf{T}_{3,2} & \mathbf{T}_{3,3} \end{bmatrix} = \mathbf{R}_{3\times3}, \tag{4.7}$$

where $\mathbf{R}$ is a general rotation matrix. Assuming that $\mathbf{R}$ is constructed by the following sequence of rotation matrix multiplication

$$\mathbf{R} = \mathbf{R}(\psi)\mathbf{R}(\theta)\mathbf{R}(\phi) \tag{4.8}$$

$$= \begin{bmatrix} c(\theta)c(\psi) & -c(\phi)s(\psi) + s(\phi)s(\theta)c(\psi) & s(\phi)s(\psi) + c(\phi)s(\theta)c(\psi) \\ c(\theta)s(\psi) & c(\phi)c(\psi) + s(\phi)s(\theta)s(\psi) & -s(\phi)c(\psi) + c(\phi)s(\theta)s(\psi) \\ -s(\theta) & s(\phi)c(\theta) & c(\phi)c(\theta) \end{bmatrix}, \tag{4.9}$$

the easiest way forward is by obtaining θ first. But, since θ is aquired by

$$\theta = -asin(R_{3,1}), \tag{4.10}$$

it will accordingly have two possible values,

$$\theta = -asin(R_{3,1}) \vee \theta = \pi + asin(R_{3,1}). \tag{4.11}$$

To elect the correct value of θ one can take advantage of further information provided by the JCM. E.g. one will desirably have small, incremental changes in $\mathbf{q}$ from one iteration to the next, and on the basis of this, a possible selection of the θ closest to the value from the previous iteration can be made. Next, we can obtain ϕ by

$$tan(\phi) = \frac{R_{3,2}}{R_{3,3}} \Rightarrow \phi = atan2(R_{3,2}, R_{3,3}), \tag{4.12}$$

but an obstacle appears here as well. As elaborated in (Slabaugh, 1999), if $cos(\theta) < 0$ then we actually have $\phi = atan2(-R_{3,2}, -R_{3,3})$. This, one can overcome by including $cos(\theta)$ in the equation

$$\phi = atan2(\frac{R_{3,2}}{cos(\theta)}, \frac{R_{3,3}}{cos(\theta)}). \tag{4.13}$$

ψ can be obtained in a similar way:

$$\psi = atan2(\frac{R_{2,1}}{cos(\theta)}, \frac{R_{1,1}}{cos(\theta)}). \tag{4.14}$$

When $cos(\theta)$ is 0, at either $\theta = -\pi/2$ or $\theta = \pi/2$, the rotation matrix is in a state termed *Gimbal Lock* (Popa, 1998). At this specific configuration we have an infinite number of solutions, but when again applying the information about the previous configuration, the gimbal lock can be worked around by using the previous value of ψ. The theoretical development can be concluded in the following pseudo code, and implemented in Python as shown in Appendix B:

rotationMatrix2Euler($\mathbf{R}$, θ_{prev}, ψ_{prev})

- **if** ($\theta_{prev} == None$ **or** $\psi_{prev} == None$)
  - $\theta_{prev} = \theta_{init}$ and $\psi_{prev} = \psi_{init}$
- **if** ($R_{3,1} != 1$)
  - $\theta = -asin(R_{3,1})$
  - **if** ($(\pi - \theta)$ closer to θ_{prev} than θ)
    - $\theta = \pi - \theta$
  - $\phi = atan2(\frac{R_{3,2}}{cos(\theta)}, \frac{R_{3,3}}{cos(\theta)})$
  - $\psi = atan2(\frac{R_{2,1}}{cos(\theta)}, \frac{R_{1,1}}{cos(\theta)})$
- **else if** ($R_{3,1} == -1$)
  - $\psi = \psi_{prev}$
  - $\theta = -\pi/2$
  - $\phi = \psi + atan2(R_{1,2}, R_{1,3})$
- **else**
  - $\psi = \psi_{prev}$
  - $\theta = -\pi/2$
  - $\phi = -\psi + atan2(-R_{1,2}, -R_{1,3})$

Table 4.1: Pseudo code for computing Euler angles from a rotation matrix

4.3 Joint Constraints

Referring to Sub-Chapter 2.2.4, we want to construct a diagonal matrix, $\mathbf{C}$, where the diagonal entries $c_{i,i}$ limits the respective joints within its physical boundaries. A continuous, reducing function to decide $c_{i,i}$, enforced when the joint enters a specified domain close to the limit, will yield smooth trajectories (on the contrary to a switching function, e.g. *sign(x)*). One would also want quick reduction when

entering the domain, since reaching a joint limit in high speed would be thoroughly undesirable, but we prioritize continuity and simplicity since high speed action is not present. We can then achieve our objective by a cosine function with the following behavior within the exemplified domain of $\delta = \frac{\pi}{2}$.

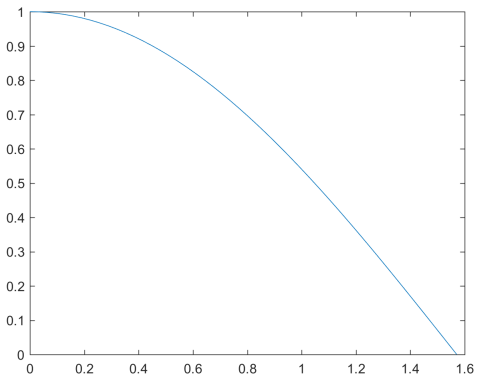

Figure 4.2: Behavior of the cosine function within reducing domain.

The weighted function can now be implemented into our algorithm from Equation 2.22, by using the following inequality constraints:

$$c_{i,i} = \begin{cases} 1 & if \quad q_i + \delta_i < q_{i,max} \vee q_i - \delta_i > q_{i,min} \\ cos\left[\frac{\pi(q_i - q_{i,max} + \delta_i)}{2\delta_i}\right] & if \quad q_i \geqslant q_{i,max} + \delta_i \wedge \dot{q}_i > 0 \\ cos\left[\frac{\pi(q_{i,min} - q_i + \delta_i)}{2\delta_i}\right] & if \quad q_i \leqslant q_{i,min} - \delta_i \wedge \dot{q}_i < 0 \\ 0 & if \quad (q_i = q_{i,max} \wedge \dot{q}_i > 0) \vee (q_i = q_{i,min} \wedge \dot{q}_i < 0) \end{cases} \tag{4.15}$$

where $q_{i,max}$ is the upper joint limit, $q_{i,min}$ is the lower joint limit and δ_i is the domain in vicinity of a joint limit where the continuous, reducing function should operate. It is important to notice that $c_{2,2} = 0$ in our case, since walking distance is unlimited within a reasonable workspace. To prohibit unnecessary turning, the following limits are set:

$$\frac{-\pi}{2} \leq q_1 \leq \frac{\pi}{2} \tag{4.16}$$

$$\frac{-\pi}{2} \leq q_1 \leq \frac{\pi}{2}, \tag{4.17}$$

even though there are no physical limitations to q_1 and q_3. To be able to operate with the highest dexterity in $\mathbf{Q}_{space}$ possible, these small domains have been established after extensive simulations:

$$\delta_i = 5\ [deg]\ \forall\ revolute\ joints, \quad \delta_i = 0.05\ [m]\ \forall\ prismatic\ joints. \tag{4.18}$$

4.4 The Damped Least Squares Method

From Sub-Chapter 2.2.4 we can evoke the DLSM and how it provides the IK solver with robustness to singularities. We add a *damping constant* λ^2, to the invertible factor of the pseudoinverse.

$$\Delta\mathbf{q} = \alpha(J^T J + \lambda^2\mathbf{I})^{-1}J^T\mathbf{e} \tag{4.19}$$

This damping constant is a tuning parameter, which will cause a small disturbance to the system proportional to its size. In the course of minimizing this undesired impact, the next section will turn to how λ should be determined.

4.5 The Damping Factor Function

To apply the theoretical work of (Egeland et al., 1991) and (Buss and Kim, 2005) on Nao, we start by generating the SVD of Nao's Jacobian, to better understand the relations between accuracy and singularity avoidance.

The Jacobian can be decomposed into SVD form.

$$\mathbf{J}_{6\times 9} = \mathbf{U}_{6\times 6}\mathbf{\Sigma}_{6\times 9}\mathbf{V}^T_{9\times 9}, \tag{4.20}$$

If we now use the notation of SVD in the DLSM, we can investigate the relations between singular values and the damping factors. At the same time we introduce the diagonal damping factor matrix $\mathbf{\Lambda}$, which gives the mobility of having individual damping factors $\lambda_{i,i}$ for each singular value.

$$\Delta\mathbf{q} = \alpha\mathbf{J}^T(\mathbf{J}\mathbf{J}^T + \mathbf{\Lambda}^2\mathbf{I})^{-1}\mathbf{e} \tag{4.21}$$

henceforth, we will only work with the interesting factor regarding singularities; $\mathbf{J}^T(\mathbf{J}\mathbf{J}^T + \mathbf{\Lambda}^2\mathbf{I})^{-1}$, which leads to

$$\begin{aligned}\mathbf{J}^T(\mathbf{J}\mathbf{J}^T + \mathbf{\Lambda}^2\mathbf{I})^{-1} &= \mathbf{U}\mathbf{\Sigma}\mathbf{V}^T(\mathbf{U}\mathbf{\Sigma}\mathbf{V}^T\mathbf{V}\mathbf{\Sigma}^T\mathbf{U}^T + \mathbf{\Lambda}^2\mathbf{I})^{-1} &(4.22)\\ &= \mathbf{U}\mathbf{\Sigma}\mathbf{V}^T(\mathbf{U}\mathbf{\Sigma}\mathbf{\Sigma}^T\mathbf{U}^T + \mathbf{\Lambda}^2\mathbf{I})^{-1}, &(4.23)\end{aligned}$$

since $\mathbf{V}^T\mathbf{V} = \mathbf{I}$ because of it's orthogonal properties. Further manipulation of the expression gives

$$\begin{aligned}\mathbf{U}\mathbf{\Sigma}\mathbf{V}^T(\mathbf{U}\mathbf{\Sigma}\mathbf{\Sigma}^T\mathbf{U}^T + \mathbf{\Lambda}^2\mathbf{I})^{-1} &= \mathbf{V}\mathbf{\Sigma}^T\mathbf{U}^T(\mathbf{U}\mathbf{\Sigma}\mathbf{\Sigma}^T\mathbf{U}^T + \mathbf{\Lambda}^2\mathbf{I})^{-1} &(4.24)\\ &= \mathbf{V}\mathbf{\Sigma}^T\mathbf{U}^T\mathbf{U}(\mathbf{\Sigma}\mathbf{\Sigma}^T + \mathbf{\Lambda}^2\mathbf{I})^{-1}\mathbf{U}^T &(4.25)\\ &= \mathbf{V}\mathbf{\Sigma}^T(\mathbf{\Sigma}\mathbf{\Sigma}^T + \mathbf{\Lambda}^2\mathbf{I})^{-1}\mathbf{U}^T &(4.26)\end{aligned}$$

By again extracting the most interesting factor, $\mathbf{\Sigma}^T(\mathbf{\Sigma}\mathbf{\Sigma}^T + \mathbf{\Lambda}^2\mathbf{I})^{-1}$, we can apply the theory of singular values for diagonal matrices from Chapter 2.2.3.

$$\mathbf{\Sigma}^T(\mathbf{\Sigma}\mathbf{\Sigma}^T + \mathbf{\Lambda}^2\mathbf{I})^{-1} = \sum_{i=1}^{6}\frac{\sigma_{i,i}}{\sigma_{i,i}^2 + \lambda_{i,i}^2}, \tag{4.27}$$

where $\lambda_{i,i}$ and $\sigma_{i,i}$ are the diagonal components of $\mathbf{\Lambda}$ and $\mathbf{\Sigma}$ respectively, and i sums up to six because our row-space is of $\mathbb{R}^6$. Contemplating Equation 4.27 we observe that

$$\frac{\sigma_{i,i}}{\sigma_{i,i}^2 + \lambda_{i,i}^2} \approx \frac{1}{\sigma_{i,i}}$$

when $\sigma_{i,i} \gg \lambda_{i,i}$. On the other hand, when approaching a singularity, the smallest singular value will go towards zero while the corresponding component of the solution will go towards zero by the factor $\sigma_{i,i}/\lambda i, i$ (Chiaverini et al., 1991).

Considering the observations above, we have three "first look" ways of creating a function for $\mathbf{\Lambda}$, according to relevant literature: predefining a maximum joint change and introduce damping to uphold this limit when needed (Buss and Kim, 2005); estimating the smallest singular value $min(\sigma_{i,i})$ to predict an interval for increased damping (Egeland et al., 1991); establishing a relationship between input Cartesian changes and output joint changes to detect abnormal behavior (Buss and Kim, 2005). With the processing capabilities of today, an approach for estimating singular values may well find itself superfluous. In the case of $\mathbf{J}_{6\times9}$, considered as a rather minuscule matrix in modern data processing, we can easily tolerate the thoroughness of developing the SVD at every iteration, and control for tolerable singular values. A brief investigation of computational time for developing SVD of a 6×9 matrix, which can be found in Appendix A, exhibits an average computational time of

$$averageTime(SVD(\mathbf{J_{6\times9}})) = 1.7943 \cdot 10^{-5}\ [s]. \tag{4.28}$$

Now we have to inspect the particular singularities in our system, in order to properly treat them. As shown in Equation 2.16, for every joint outward in the robotic chain there is one more possible joint axis to align with, which means an increased contingency for singular values. Physically, this implies a greater concern for *elbow singularities* (in the absence of wrist joints), which mathematically equates to $\sigma_{5,5}$ and $\sigma_{6,6}$. Ascending the diagonal of the singular value matrix, the likelihood should decline.

4.5.1 Regulating the damping factor

Since the development of a SVD executes within about 0.005 times a full iteration of the IK solver (obtained through an average of 1800 iterations), we can simply inspect the singular values of Σ and develop a damping factor for the individual singular value. To achieve continuity, the evolving damping factor should follow a differentiable dynamic. This could be solved by the same functionality used in Sub-Chapter 4.3, a simple cosine. The final solution to the damping factor then becomes

$$\lambda_{i,i} = \begin{cases} [1 - cos(\frac{(\epsilon - \sigma_{i,i})\pi}{2\epsilon})]\lambda_{max} & if \quad \sigma_{i,i} < \epsilon \\ 0 & else \end{cases} \tag{4.29}$$

where $\lambda_{i,i}$ is the damping constant of the $i'th$ diagonal element in $\mathbf{\Lambda}$, λ_{max} is the maximum damping constant value, $\sigma_{i,i}$ is the singular value of the $i'th$ diagonal

element in $\mathbf{\Sigma}$ and ϵ is a user set parameter to regulate the interval of damping. After extensive testing, an interval of damping by 0.15 was established as a good trade-off between robustness and accuracy. Making the solver do a zero-crossing for the two most vulnerable joints, the DLS with dynamic damping coefficients handles it with a continuous damped behavior, keeping the smallest singular value above 0.15 (See Figure 4.3).

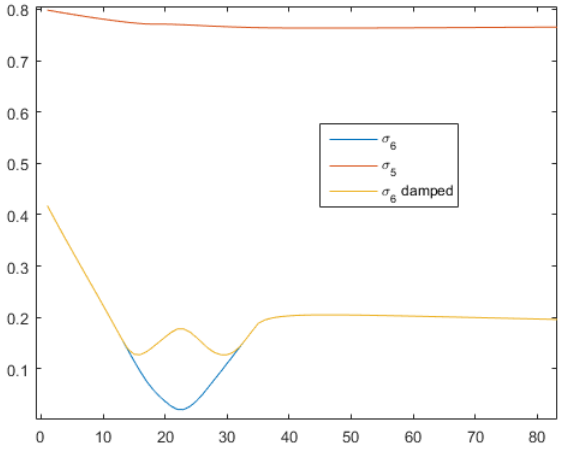

Figure 4.3: The variable damping handling a zero-crossing

4.6 Linearization and inaccuracy

Due to the linearized velocities, described in Sub-Chapter 2.2.2, a loss of accuracy occurs at every discrete instance. The degree of accuracy from the linear solution will be decided upon how often the system is linearized, i.e. how large the step-size in the numerical solver is - which is proportional to α in this case. But, to increase the mobility of our numerical solver, we introduce the diagonal, factorial matrix $\mathbf{W}$. $\mathbf{W}$ decides proportional step-size according to the error - just as the case for proportional gains in a control system - for every joint change. To initialize a benchmark for desired accuracy, the deviation for physical sensory equipment is a very good place to start. As stated in Sub-Chapter 2.3.2, Nao operates with about $0.1° = 0.0017\,[rad]$ precision in joint measurements. Since we have a closed loop, the inaccuracy due to linearization will be corrected at every iteration, so to determine the maximum entry of $\mathbf{W}$, w_{max}, we only need to consider deviation from one iteration. Because Nao's body is kinematically modelled with seven rotational joints in a chain, the deviation can be approximated with the product of e.g. seven cosines. With a simple script, attached in Appendix A, a good estimate of the step-size is obtained, by increasing the step-size until the error of linearization becomes larger than the deviation of the joint sensors.

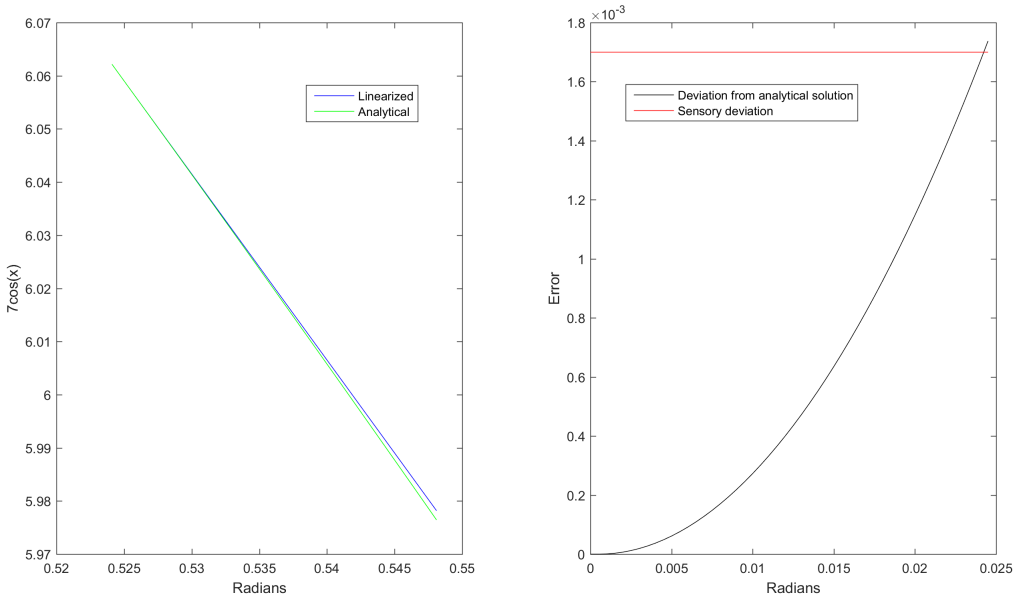

Figure 4.4: Linear cosine deviation from analytical, for different step-sizes

The simulation above starts at the angle of $\pi/6\,[rad]$, and increases the step-size by $0.0005\,[rad]$ at every iteration, until the deviation reaches $0.0017\,[rad]$. That results in a maximum step-size of

$$w_{max} \approx 0.0245. \tag{4.30}$$

4.7 Dynamic Weighting of the Step Size

The diagonal matrix $\mathbf{W}$ defines the step size of every joint increment, proportional to the error vector $\mathbf{e}$ - i.e. a proportional gain. We have from Sub-Chapter 4.6 a maximum desired change of $w_{i,i}^{max} = 0.0245$ at each iteration, to prevent solver inaccuracies from overreaching physical inaccuracies.

With inspiration from automatic gain controlling, an idea came up to create a variable step-size, changing according to the system parameters. This is of course, to further track a previous parallel, essential to state of the art numerical solvers. It is termed *Variable-Step*-solver (VSS), with the intention of reducing step-size to increase accuracy during rapid variations of model states or to increase step-size when the model changes slowly (MathWorks, 2016). The prime concern with implementing a VSS to the DLS method, is the matter of redundancy. The undertaken solution to this matter is influenced by the principle, being also severely intertwined with VSS, of predictor-corrector methods in numerical mathematics. Predictor-corrector methods proceed by extrapolating a derivative function fitting on the basis of previous states (predictor), and then using this predictor to interpolate the derivative again (corrector) (Weisstein, 2016). In the case of the DLS, a derivative $\mathbf{\Delta q}$ (predictor) is calculated with the step-size of $w_{i,i} = 0.0245$.

$$\mathbf{\Delta q} = 0.0245\,[\mathbf{J}^T(\mathbf{J}\mathbf{J}^T + \mathbf{\Lambda}^2\mathbf{I})^{-1}\mathbf{e}]. \tag{4.31}$$

The resulting $\mathbf{q}$ needs to be utilized for finding a VSS to satisfy the accuracy demands - both in the matter of slowly varying and highly volatile states. This is

done accordingly:
- The change in Cartesian space, delivered by $\mathbf{\Delta q}$ is obtained:

$$\mathbf{\Delta x}_{6\times 1} = forwardKinematics(\mathbf{q} + \mathbf{\Delta q}) - \mathbf{x}. \tag{4.32}$$

- The desired Cartesian change relative to the total Cartesian error, δ, is obtained

$$\delta_{6\times 1} = 0.0245|\frac{\mathbf{e}}{\mathbf{\Delta x}}|. \tag{4.33}$$

- The weighting matrix $\mathbf{W}$ is established, according to the dimensions of $\mathbf{q}_{i\times 1}$

$$W_{i,i} = \begin{bmatrix} 0 & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & 0 \end{bmatrix}. \tag{4.34}$$

- The row norm $||\tilde{\mathbf{J}}^{\dagger}||_{row}$ of the damped pseudoinverse $\tilde{\mathbf{J}}^{\dagger}_{i\times 6}$, represents the total increment to each joint. This principle can be exploited to obtain each joint step-size, $w_{i,i}$, accordingly

$$w_{i,i} = |\sum_{k=1}^{6} \tilde{\mathbf{J}}^{\dagger}_{i,k}\delta_k|. \tag{4.35}$$

Every entry of a column in $\tilde{\mathbf{J}}^{\dagger}$ is "participatory" in creating a joint movement to mitigate the error corresponding to the same column. Since this system is linearized, we can assume that the relation between one Cartesian coordinate change and one Cartesian error is a linear combination.

4.8 Fuzzy Logic to shift between different solvers

As was briefly elaborated in the underlying academical theory for the fuzzy logical implementation, the prime incentive for this approach is to mimic human behavior in reaching a configuration with one hand. First things first; a wide range of human performances need to be inspected, to understand *which degrees of freedom are being exploited* and which are of less importance. The original idea was for this to be handled to the intricacy of discretizing the entire path, so as to understand if there are specific parts of a trajectory being specifically dependent on a subset of joints. A lot of effort was directed into this theory. There have been undertaken methods to discretize the space around the end-effector into spheres, where each joint have a degree of contribution, defined by studies of human execution. This approach proved though insufficient after a considerable amount of testing, and the reasons came to be: extreme difficulty in tuning the weighting for each joint in each sphere (declare there degree of contribution), unwanted - sometimes unstable - behavior when entering a new sphere because of the sudden steps in weighting and no solution provided to the problems of very slow convergence at instances with low dexterity in one or more degrees of freedom.

The apparent dead-end above became the lead to the present solution: switching between the set of controller states, instead of shifting between their respective joint change gains. In this way, the counteraction between states contribution to increment is decreased. In other words: an increase in redundancy. The first priority is to control the states regarding orientation - the Euler angles:

$$\mathbf{e}_{4:6} := \begin{bmatrix} e_\phi \\ e_\theta \\ e_\psi \end{bmatrix}_{3\times 1}, \quad \mathbf{J_r} = \begin{bmatrix} \frac{\partial\phi}{\partial q_1} & \frac{\partial\phi}{\partial d_2} & \frac{\partial\phi}{\partial q_3} & \cdots & \frac{\partial\phi}{\partial q_9} \\ \frac{\partial\theta}{\partial q_1} & \frac{\partial\theta}{\partial d_2} & \frac{\partial\theta}{\partial q_3} & \cdots & \frac{\partial\theta}{\partial q_9} \\ \frac{\partial\psi}{\partial q_1} & \frac{\partial\psi}{\partial d_2} & \frac{\partial\psi}{\partial q_3} & \cdots & \frac{\partial\psi}{\partial q_9} \end{bmatrix}_{3\times 9} \tag{4.36}$$

$$\mathbf{\Delta q} = \alpha(\mathbf{J_r}^T\mathbf{J_r} + \mathbf{\Lambda_r})^{-1}\mathbf{J_r}^T\mathbf{e}_{4:6}, \tag{4.37}$$

with Λ being 3×3, which leads to only the Euler angles being minimized, with every 9 DOF active (i.e. a very high degree of redundancy). This approach is though in need of a further slight modification, because of Nao's restrictions as a land based robot - this would not be the case for an underwater robot. Heave, the desired configuration parameter in z directon, can dependent on the rotational joints to reach its desired value because of the small workspace (Nao's change in height), which is just over 100 $[mm]$. This implies an extension of Equation 4.36 to include the state of z, with corresponding e_z, into the variables:

$$\mathbf{e}_{3:6} := \begin{bmatrix} e_z \\ e_\phi \\ e_\theta \\ e_\psi \end{bmatrix}_{3\times 1}, \quad \mathbf{J_r} = \begin{bmatrix} \frac{\partial z}{\partial q_1} & \frac{\partial z}{\partial d_2} & \frac{\partial z}{\partial q_3} & \cdots & \frac{\partial z}{\partial q_9} \\ \frac{\partial\phi}{\partial q_1} & \frac{\partial\phi}{\partial d_2} & \frac{\partial\phi}{\partial q_3} & \cdots & \frac{\partial\phi}{\partial q_9} \\ \frac{\partial\theta}{\partial q_1} & \frac{\partial\theta}{\partial d_2} & \frac{\partial\theta}{\partial q_3} & \cdots & \frac{\partial\theta}{\partial q_9} \\ \frac{\partial\psi}{\partial q_1} & \frac{\partial\psi}{\partial d_2} & \frac{\partial\psi}{\partial q_3} & \cdots & \frac{\partial\psi}{\partial q_9} \end{bmatrix}_{4\times 9} \tag{4.38}$$

$$\mathbf{\Delta q} = \alpha(\mathbf{J_r}^T\mathbf{J_r} + \mathbf{\Lambda_r})^{-1}\mathbf{J_r}^T\mathbf{e}_{3:6}, \tag{4.39}$$

with Λ_r now being 4×4. Now, we would like to control the end-effector by x and y, and possibly z, to reach the desired configuration for all six states. This is also subjugated to a bit more modification than just controlling by these desired states. Due to the kinematic model of the robot body, the ability to control x and y is coupled to the state of ψ. From this principal the next control system, based on mitigating the error in *surge*, *sway* and *heave* becomes:

$$\mathbf{e}_{1:3,6} := \begin{bmatrix} e_x \\ e_y \\ e_\psi \end{bmatrix}_{3\times 1}, \quad \mathbf{J_p} = \begin{bmatrix} \frac{\partial x}{\partial q_1} & \frac{\partial x}{\partial d_2} & 0 & \cdots & 0 \\ \frac{\partial y}{\partial q_1} & \frac{\partial y}{\partial d_2} & 0 & \cdots & 0 \\ \frac{\partial\psi}{\partial q_1} & \frac{\partial\psi}{\partial d_2} & \frac{\partial\psi}{\partial q_3} & \cdots & \frac{\partial\psi}{\partial q_9} \end{bmatrix}_{3\times 9} \tag{4.40}$$

$$\mathbf{\Delta q} = \alpha(\mathbf{J_p}^T\mathbf{J_p} + \mathbf{\Lambda_p})^{-1}\mathbf{J_p}^T\mathbf{e}_{1:2,6}, \tag{4.41}$$

where Λ_p is 4×4. The last important feature is how to switch between them. Due to ψ being regulated to zero, if the orientation focused control system operates first, the control system operating on x, y and ψ will easily keep ψ on zero, because of symmetric contributions by q_1 and q_3 (both of them only contributing to ψ by the factor of 1) canceling each other out. Both x, y and z are in this system decoupled from the first control system, with ϕ and θ being decoupled from the latter.

The strategy, based on degrees of truth on the contrary to Boolean truth, becomes the following:

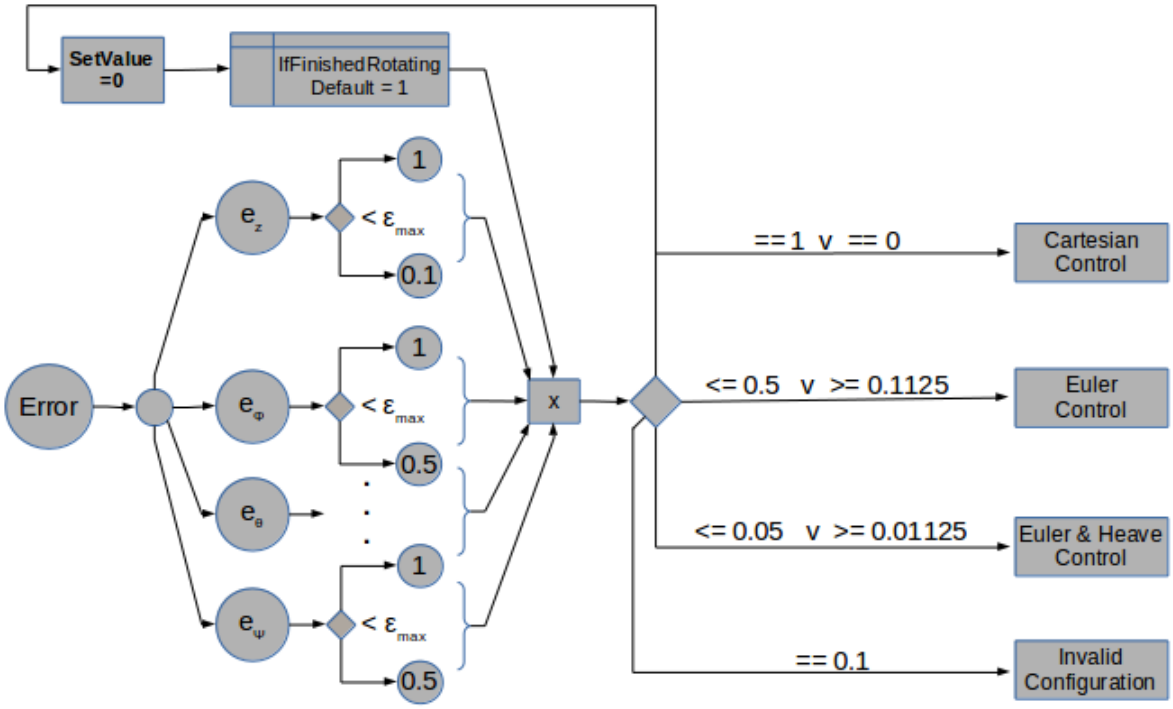

Figure 4.5: Block scheme of the fuzzy logic to switch between control systems

Where $\epsilon_{max} = 0.0017$ for the Euler angles and ϵ_{max} for e_z is set to the joint limits in d_4. This fuzzy logic produces an output scalar which indicates which states that have a partial membership in the current control problem, which by the mathematical models of the Jacobians will transform into a degree of membership for the joints. In some cases, where the output scalar is 0.1, no joints will have a membership in the control problem to the desired configuration and therefore the configuration is invalid (outside the workspace). One has to clarify that invalid configurations could appear within this logic as well, due to the physical joint constraints, but they will need to be detected by other measures. To relate Figure 4.5 to the control equations in this Sub-Chapter, we construct a small table:

Output Scalar	States to Control	Control Equation
1	x, y, z, ψ	4.41
$0.5 - 0.125$	ϕ, θ, ψ	4.37
$0.05 - 0.0125$	z, ϕ, θ, ψ	4.39
0	x, y, z, ψ	4.41
0.1	No valid control state	-

Table 4.2: Translation from fuzzy logic output into control system

4.9 An Overview - The Final Inverse Kinematic Algorithm

When every demand and desired ability for the inverse kinematic solver is applied, we obtain the final equation:

$$\Delta \mathbf{q} = \mathbf{CW}(\mathbf{J}^T\mathbf{J} + \mathbf{\Lambda})^{-1}\mathbf{J}^T\tilde{\mathbf{e}}, \quad \tilde{\mathbf{e}} = 0.0245\mathbf{e} \tag{4.42}$$

$$\Delta \mathbf{q} = \mathbf{CW}\tilde{\mathbf{J}}^{\dagger}\tilde{\mathbf{e}}, \quad \tilde{\mathbf{J}}^{\dagger} = (\mathbf{J}^T\mathbf{J} + \mathbf{\Lambda})^{-1}\mathbf{J}^T, \tag{4.43}$$

where $\tilde{\mathbf{J}}^{\dagger}$ can be termed *damped pseudoinverse Jacobian*. To better illustrate the flow of information in the inverse kinematic algorithm, the block diagram beneath exhibits the different components and how they communicate (with the most essential; the DLS approach, emphasized in the center).

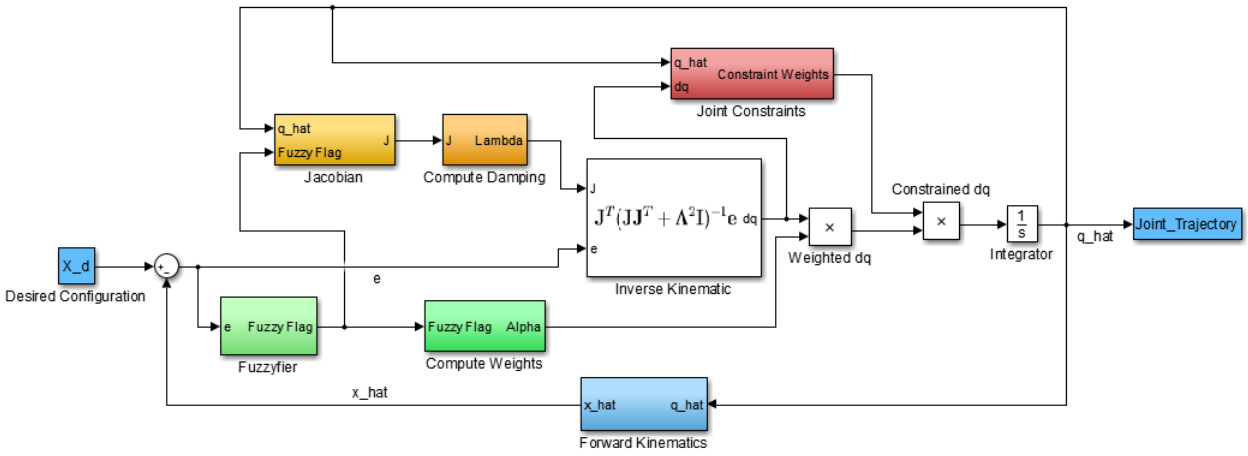

Figure 4.6: Block diagram showing the data flow between components

Chapter 5

Simulations and Results

Clarifications about some governing principals for the simulations need to be made. Firstly, the x-axis represents iterations and not time, because computation time is too hardware-dependent and may not even reflect the corresponding y-axis exactly. Iterations serve as an exact parameter across hardware platforms, with much higher relevance to the theoretical development of the numerical solver (e.g. one can identify the exact situation of a numerical anomaly). Secondly, the presentation consists of real-time Euler angles alongside rotational joint trajectories, and Cartesian coordinates alongside prismatic joint trajectories. This is to translate the joint development to a "human perceived" coordinate system, and to split up y-axes values in $[rad]$ and $[mm]$ (due to quite self-evident differences in value). Lastly, the different simulations aim at unveiling the important functionality implemented; singularity robustness, interacting control systems, variable step-size and error convergence (which from a theoretical point of view is the most important aspect, due to it being the actual control variable).

5.1 Successful Solution to a Desired Configuration with Locomotion

The final IK solver was given a reasonable, desired configuration as input, with the intention of revealing solution stability, singular value development, error convergence, efficiency and the handling of locomotion. By the usage of function $solveIK(q_{init}, x_{desired})$, attached in Appendix B, with the input

$$q_{init} = [0, 0, 0, 50, 0.2, -0.4, 0.51, 0.3, -0.7]$$
$$x_{desired} = [400, 500, 340, 0.3, -0.6, 0.1]$$
$$solveIK(q_{init},\ x_{desired})$$

and using the *IKplot* function provided in Appendix B, the following simulations results were obtained:

Joint Trajectory with Cartesian coordinates

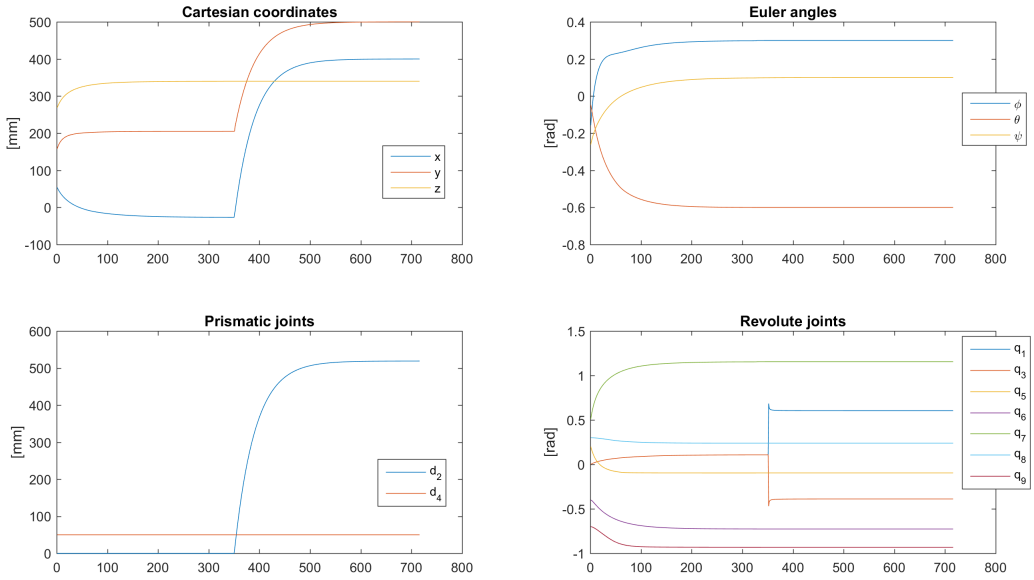

Figure 5.1: Joint trajectory and Cartesian coordinates for a reasonable desired configuration.

We can observe several interesting behaviors in this simulation. First and foremost, the solution is obtained within ≈ 720 iterations, which is very quick. Secondly, the joint trajectories indicates efficient joint development and stable numerical performance. The abnormal behavior of q_1 and q_3 at around iteration 350, is due to the change in control states, from only controlling Euler angles to controlling yaw and the Cartesian states. This is also the cause of non-continuity for d_2 at the same instance.

Error convergence for Cartesian coordinates and Euler angles

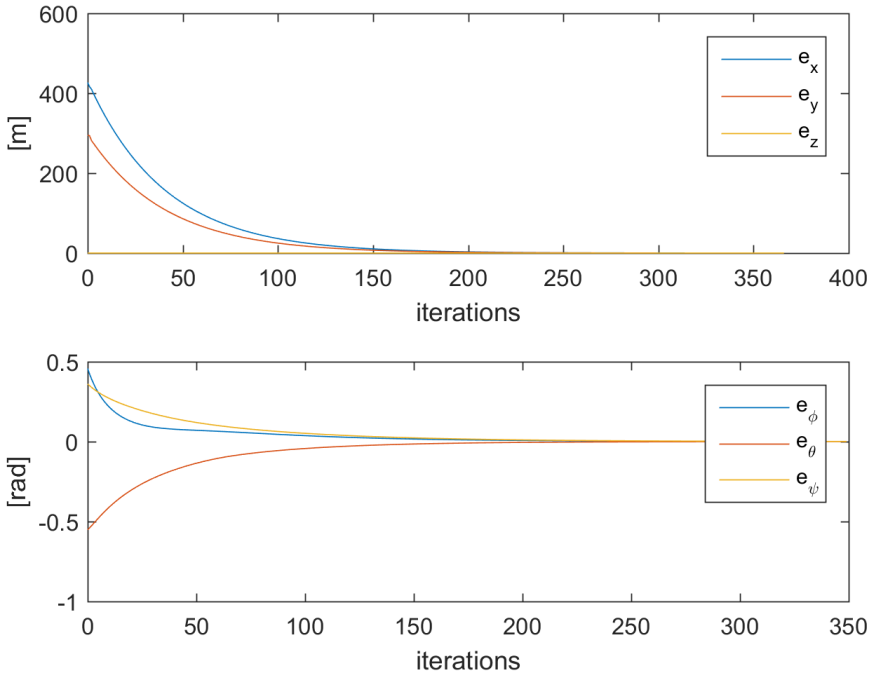

Figure 5.2: Joint trajectory and Cartesian coordinates for a reasonable desired configuration.

Even though the numerical solution experience non-continuity when changing control states, they perform stable control independently from when they are initiated. This is reinforced by the error convergence of all six states above. The convergence is quick, stable and follows the wanted exponential curvature.

5.2 Successful Solution after running through a Singularity

We deliver an initial joint set and a desired configuration, which should induce a zero-crossing, to the solver.

$q_{init} = [0.0,\ 0.0,\ 0.0,\ 50.0,\ 0.2,\ -0.4,\ 0.5,\ 0.3,\ -0.7]$
$x_{desired} = [-200.0000, 143.7317, 177.5141, -0.1574, -0.5000, -0.2612]$
$solveIK(q_{init},\ x_{desired})$

Joint Trajectory with Cartesian coordinates

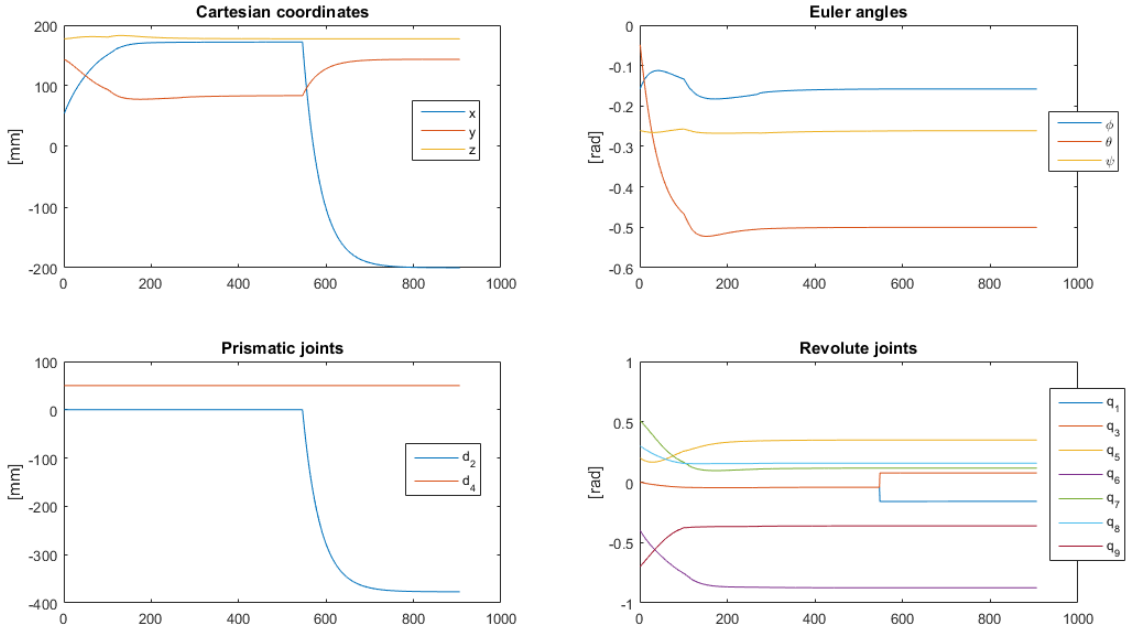

Figure 5.3: Joint trajectory and Cartesian coordinates running through a singularity

The Euler angles develops a bit volatile through first 100 iterations, but the revolute joint trajectories are smooth at the same time. We have non-continuity when the control states changes, similar to what we observed in the previous simulation. The joint trajectories seems efficient.

Error convergence for Cartesian coordinates and Euler angles

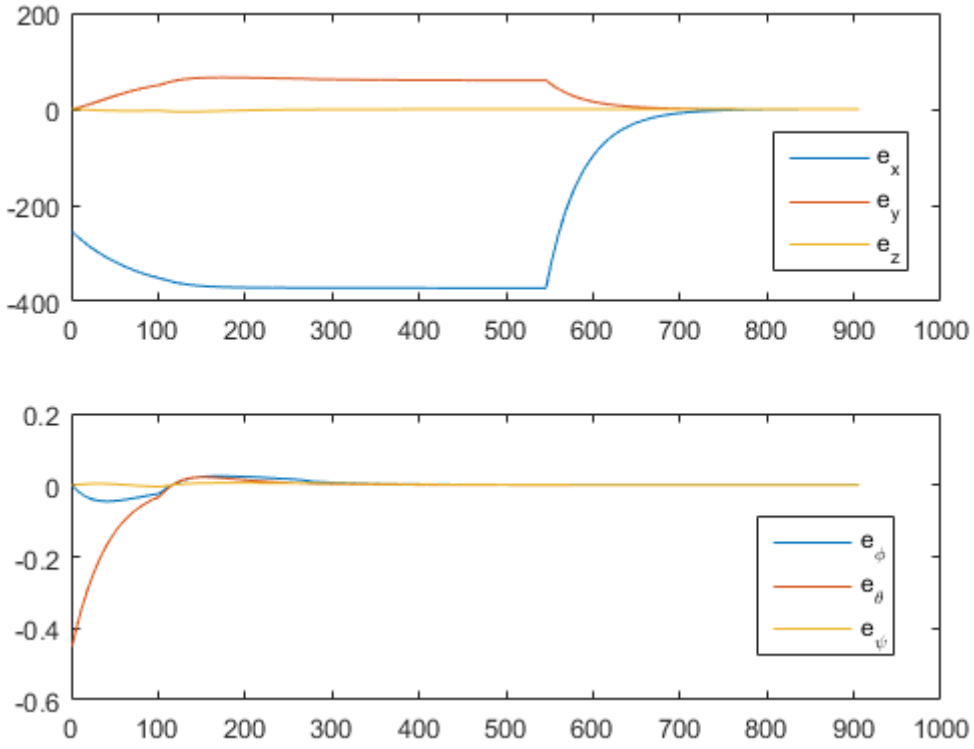

Figure 5.4: Error convergence when running through a singularity

The configuration errors converges desirably, except for e_ϕ which experience a small overshoot. Every iteration is presented in the x-axis in these plots, which elevates when the Cartesian values are not being controlled for the first ≈ 540 iterations. When the Cartesian states are controlled, these states converges quickly and stably. The key positive is the lack of any singular value behavior, which would induce large, sudden steps.

Singular and damped singular values

The solution provided above runs through a singularity right at the instance where it changes from a control system operating with the states $[z, \phi, \theta, \psi]$ (left figure), to another one working with $[x, y, \psi]$ (right figure). This is a particularly vulnerable singularity for the solver, because the value of ψ causes a singular value to be absolute zero in the new control system. This is anyhow handled by the solver, as was witnessed in Figure 5.3 and Figure 5.4, even though it fails to keep the smallest singular value above 0.15. The two figures below illustrates the two smallest singular values for the two different control systems. Notice the singularity occurring in Figure 5.6 at the initialization of the Cartesian control system (iteration 0).

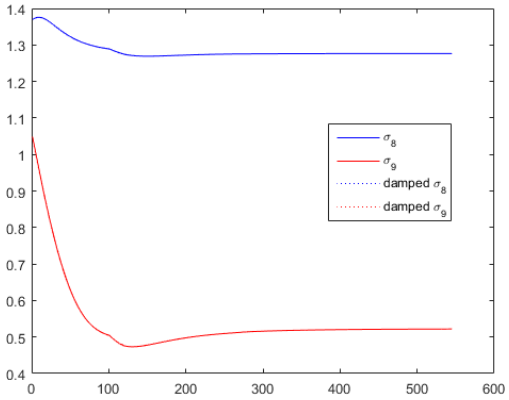

Figure 5.5: Development of the two smallest singular values with corresponding damped values

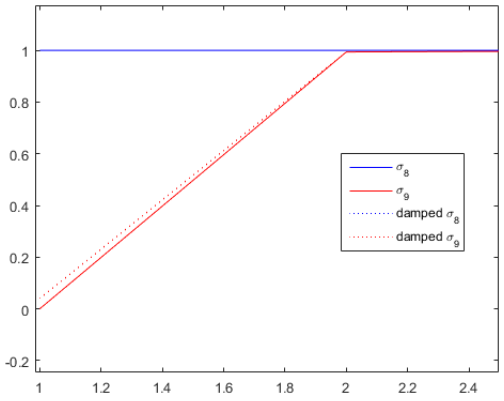

Figure 5.6: Development of the two smallest singular values with corresponding damped values

5.3 Successful Solution to a Desired Configuration with Close Inverse Kinematics

Here, the goal is to provide an example of how the algorithm handles desired configurations in vicinity of the current configuration. If a solution to the desired configuration exist in the null space of the combined system with *hip-pitch* and *left arm*, one would like the robot to refrain from walking or turning around (for clear efficiency matters).

$q_{init} = [0.0,\ 0.0,\ 0.0,\ 50.0,\ 0.2,\ -0.4,\ 0.5,\ 0.3,\ -0.7]$
$q_{desired} = [0.0,\ 0.0,\ 0.0,\ 50.0,\ 0.2,\ -0.9,\ 0.3,\ 0.3,\ -0.7]$
$solveIK(q_{init},\ FK(q_{desired}))$

Joint Trajectory with Cartesian coordinates

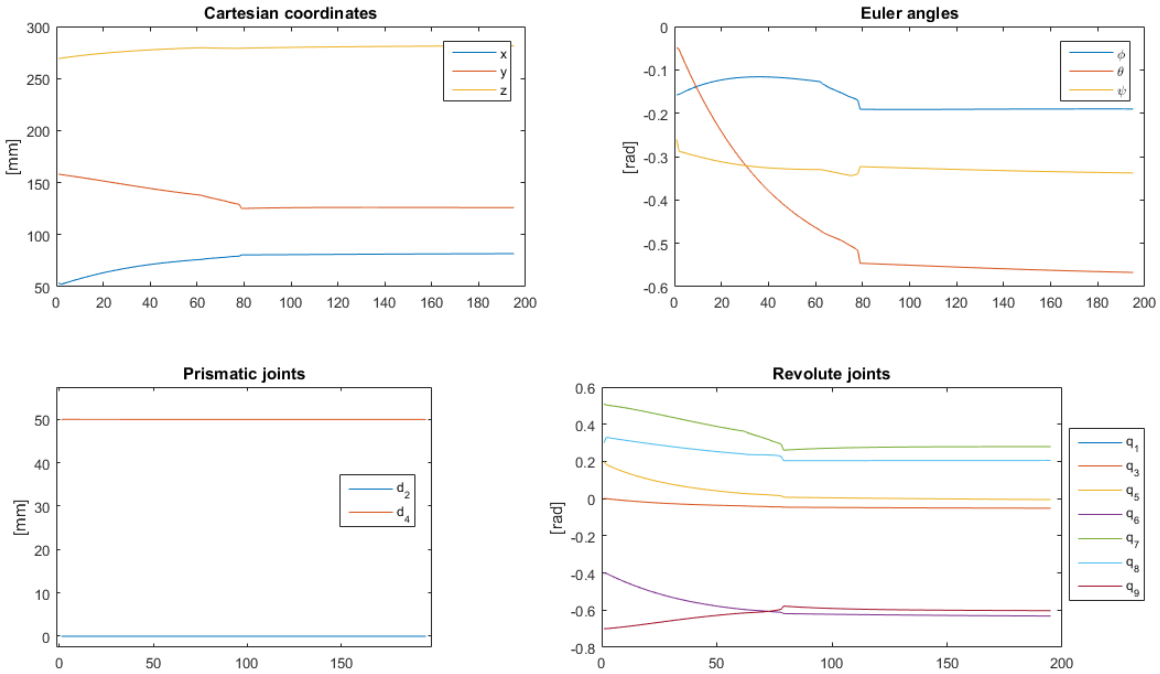

Figure 5.7: Joint trajectory and Cartesian coordinates for a desired configuration in vicinity

This underlines the desired objective of zero movement in the lower body (no turning, locomotion or standing up). It also shows quick convergence, with about 200 iterations (which corresponds to about 0.2 seconds on the lower level simulation hardware). On the downside, it elevates what was observed in the first documented simulation (Sub-Chapter 5.1), that the solver is not completely robust to non-continuity. The occurring steps are not outside the constrained maximum joint changes, so the performance is technically "legal" in terms of the governing principals for the solver. These abnormalities happens at the instance of control system switching, and sometimes due to the dynamical weighting, but this is explained by the relatively small changes and the minuscule scope (e.g. is a scope of 200 iterations almost destined to show non-continuity. The error convergence in the following illustration indicates the same behavior as witnessed above; fast convergence following the desired slope, but with some abnormalities at control system switching.

Error convergence for Cartesian coordinates and Euler angles

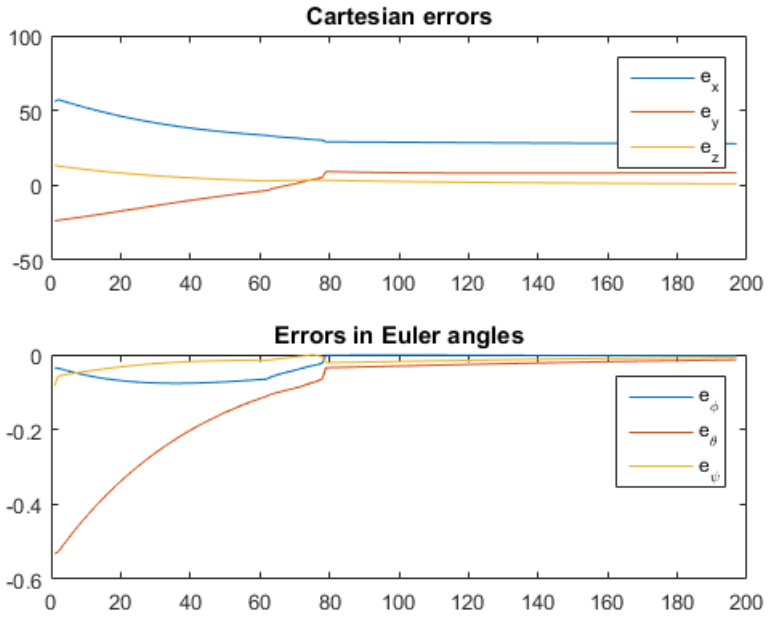

Figure 5.8: Error convergence for a desired configuration in vicinity

5.4 Dynamic weighting to counteract volatile singular values

The IK solver is called to handle another change in position and orientation, which demands zero-crossings for the revolute joints. This is upheld by inserting the negative of the input joint configuration to the FK function (named $FK(q)$ in Appendix B), and using the received configuration as input to the IK solver. By this, one sought to trigger very rapid changes in joint values, which in turn could prove the functionality of the variable step-size implemented.

$q_{init} = [0.0,\ 0.0,\ 0.1,\ 30.0,\ 0.1,\ 0.1,\ -0.1,\ 0.1,\ -0.2]$
$q_{desired} = [0.0,\ 0.0,\ -0.1,\ 30.0,\ -0.1,\ 0.1,\ 0.1,\ -0.1,\ 0.2]$
$solveIK(q_{init},\ FK(q_{desired}))$

Joint configuration and singular values without dynamic weighting

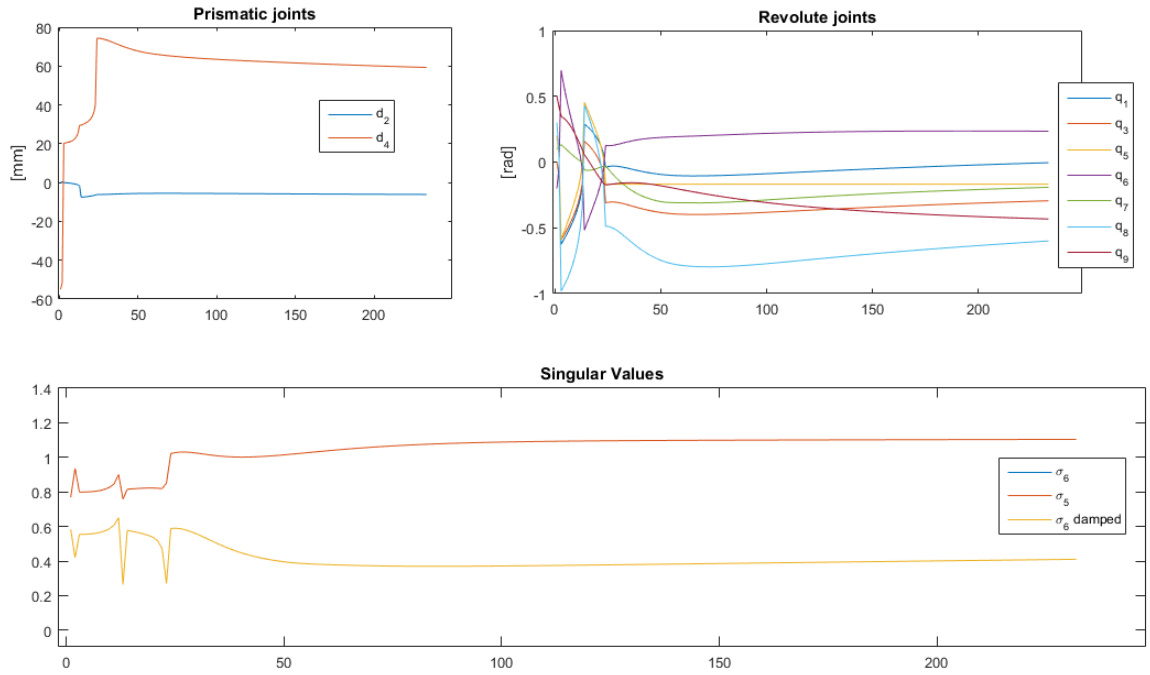

Figure 5.9: Solver handling of zero crossings with constant step-size

Figure 5.9 clearly substantiates the hypothesis of large instantaneous joint changes, up to 1.3 $[rad]$, due to the forced zero-crossings. The singular values, which in a practical sense is the scaling between inputted error and demanded joint change, experience very rapid changes between two iterations, amidst the 25 first.

Joint configuration and singular values with dynamic weighting

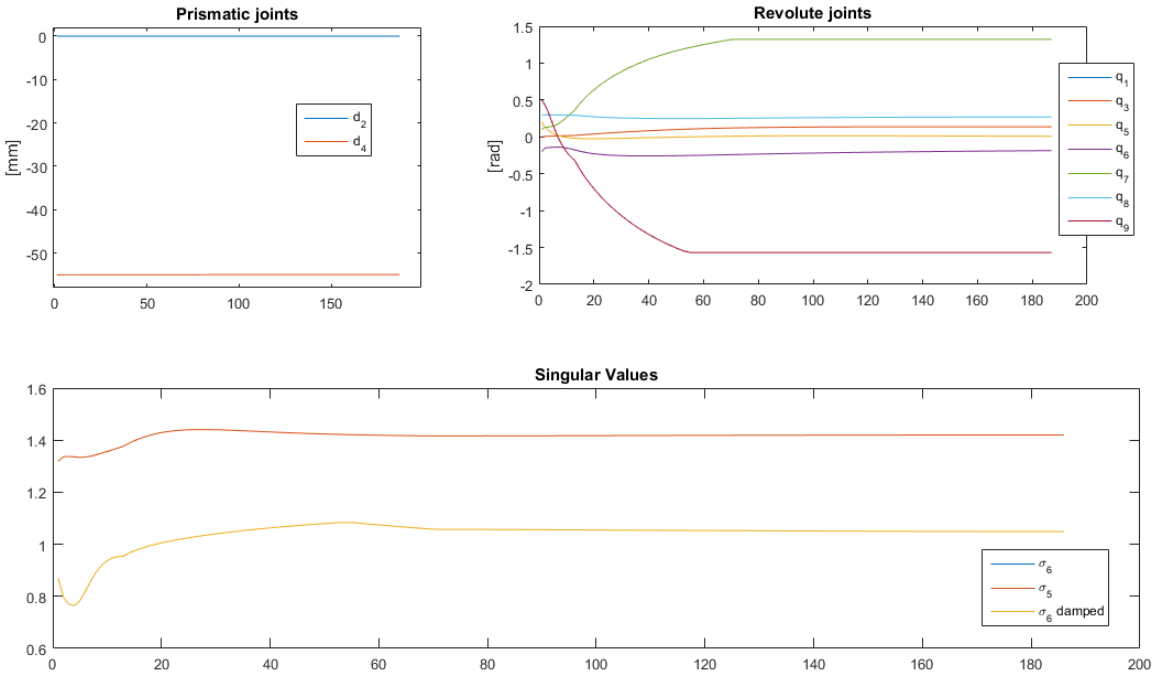

Figure 5.10: Solver handling of zero crossings with variable step-size

Figure 5.10 further strengthens the ability of a dynamic weighted proportional gain, to create variable step-sizes, in order to prevent non-continuity, stabilize the numerical solver and furthermore enhance the solution efficiency. To elaborate on the latter attribute, it converges to the vicinity of a solution at approximately 60 iterations, which the constantly weighted is far from achieving.

Chapter 6

Physical Experiments

6.1 Connecting to and Communication with Nao

Establishing Connection

Nao can be communicated with using an Ethernet cable or a local WIFI network. There are ways of using free, open source platforms for controlling Nao (e.g. through ROS), but accompanying Nao is a simple package of Python scripts that provide functions for sending a wide range of motion commands to Nao through his native, specific OS. Going by the example code attached in Appendix C, one can easily send configurations for Nao's hands or feet (using built-in IK), make Nao walk to a specific point or at such a speed, and set values in radians for every joint. Because Nao's own OS supplies one with real-time calculation of balance, *clipping* of steps to be within physical capabilities and so forth, this platform is chosen to not waste time on development for digressions (even though it imparts some restrictions, outlined later). When connection is successfully established, and the result of an inverse kinematic solution has been transmitted to Nao, the following result appears in the terminal:

```
aasmundhugo@aasmundhugo-U46SV:~$ cd Documents/
aasmundhugo@aasmundhugo-U46SV:~/Documents$ cd Python/
aasmundhugo@aasmundhugo-U46SV:~/Documents/Python$ python naoMove.py
Usage python almotion_setangles.py robotIP (optional default: 127.0.0.1)
[I] 8778 qimessaging.session: Session listener created on tcp://0.0.0.0:0
[I] 8778 qimessaging.transportserver: TransportServer will listen on: tcp://127.0.0.1:40553
[I] 8778 qimessaging.transportserver: TransportServer will listen on: tcp://192.168.1.101:40553

('Time to solve: ', 0.5045580863952637)
('Final joint values: ', [0.5186636200619319, 287.81067565518504, -0.18256711009821508, 67.70987871727526, 1.738
3589093710088, -0.43243741742388786, 0.25809240118447346, 0.03871167197821449, -0.412713199114015])
('Final Cartesian position: ', [289.2063484097021, 285.4306191405825, 191.95289530329882, -0.14468201696658362,
-0.2548070237065449, 0.2974181905355415])
```

Figure 6.1: Representation of a successful movement of Nao

How commands are communicated

Since communication follows Nao's own OS, and since Nao is a walking robot, which implies harsh restrictions on concurrent actuation (moving left arm e.g.) when moving around in the x-y-plane, movement of the different joints have to be performed subsequently. By this, we cannot create a specific path for the end-effector to follow from start to finish, but we can move some joints at a time to achieve the desired end-effector position. The movement happens in this order, and is implemented in Python in Appendix C:

set walking direction (q_1)
set walking distance (q_2)
set body posture (q_3)
set height and pitch of torso ($q_4 - q_5$)
set angles for all left arm joints ($q_6 - q_9$)

Table 6.1: The order of movement commands to Nao

6.2 Experiment Formalities

6.2.1 Objectives

- Prove the ability to control Nao kinematically.
- Prove the correctness of the inverse kinematic algorithm
- Investigate inaccuracies
- Detect other problems or possibilities regarding further work

6.2.2 Method of approach

The following approach is found as sufficient, with technical capabilities considered.

- A 3D coordinate system is established with physical axes, in which Nao is supposed to move to enable physical measurements.
- Nao is placed at the initial point and the desired target configuration is marked up with x, y and z values.
- A camera stand is mounted to document the movement.
- The program establishes contact with Nao, solves a desired configuration and sends commands to move Nao according to Table 6.1.
- After Nao is finished, the new configuration is gathered from Nao's sensors and stored, and physical measures are taken.

This is performed several times to obtain sufficient data for uncovering how successful the solution is.

6.3 Successful Inverse Kinematic Performance

It is not easy to visualize something in 4D in a written report. How Nao performs joint changes, rendered by the IK solver, can most easily be understood through a movie. One can see Nao moving around to different desired configurations, by following this link: `https://www.youtube.com/watch?v=RINPJoZJPzk&feature=youtu.be`.

6.3.1 Configuration with locomotion and change in all orientations

A configuration was given to the program connecting to Nao and delivering a solution joint set.

$$q_{init} = [0.0,\ 0.0,\ 0.0,\ 93.2,\ 0.10,\ 1.40,\ 0.30,\ -1.39,\ -1.10]$$
$$q_{desired} = [0.50,\ 200.0,\ -0.05,\ 70.0,\ 0.30,\ 1.10,\ 0.10,\ -1.60,\ -1.20]$$
$$solveIK(q_{init},\ FK(q_{desired}))$$

Which resulted in the following actual move:

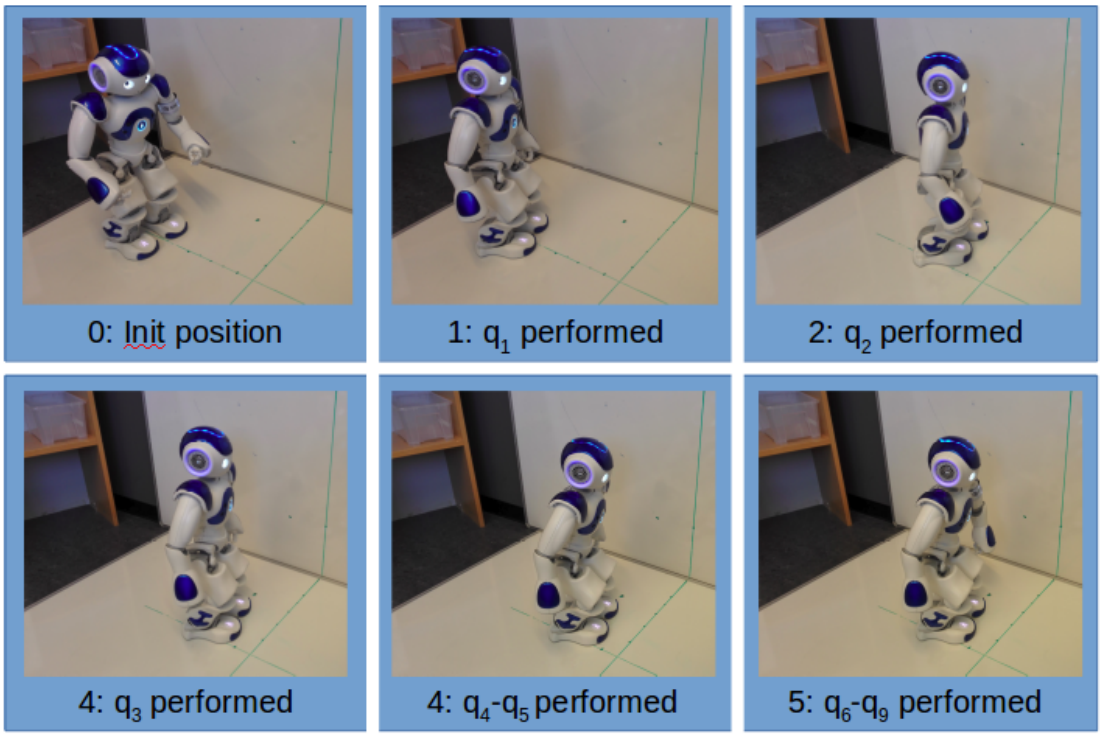

Figure 6.2: Nao moving to a given configuration

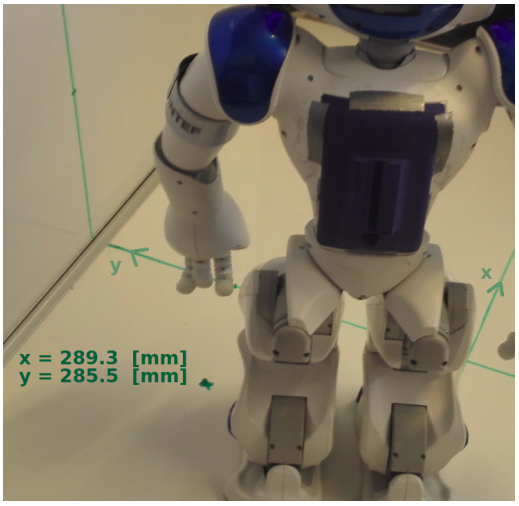

Figure 6.3: End configuration x-y-plane

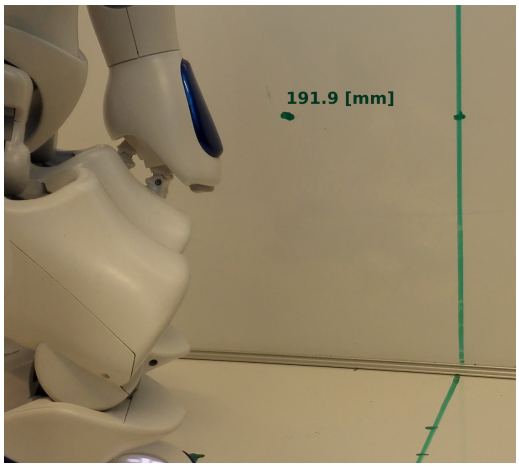

Figure 6.4: End configuration height

Physical measurements

The physical measurements are done to the highest precision possible for the human eye, which limits them to $[mm]$ and infer degrees rather than radians.

$$\begin{bmatrix} x \\ y \\ z \\ \phi \\ \theta \\ \psi \end{bmatrix}_{measured} = \begin{bmatrix} 280 \\ 277 \\ 195 \\ -80^o \\ 10^o \\ 20^o \end{bmatrix}, \quad \begin{bmatrix} x \\ y \\ z \\ \phi \\ \theta \\ \psi \end{bmatrix}_{desired} = \begin{bmatrix} 289.3 \\ 285.5 \\ 191.9 \\ -84.5^o \\ 11.2^o \\ 29.4^o \end{bmatrix} \Rightarrow \begin{bmatrix} 9.3mm \\ 11.5mm \\ 3.1mm \\ 11.2^o \\ 8.8^o \\ 9.4^o \end{bmatrix}_{error} \tag{6.1}$$

There are obvious issues regarding physical measurements when it comes to evaluating the accuracy of robotic solutions, since the execution of these require great preciseness. But, they are rather useful in this particular case, where we just want to validate if the solution is theoretically correct. This attempt does indicate correspondence between the configuration we desired and the configuration performed, which can be considered a positive result.

Nao's measurements

Nao provides the following end-configuration, but some of these measurements are very wrong. It is not possible to get an initialization of Nao's *world coordinate frame* at a desired position, and therefore the measurements, especially in x, y and ψ it will never be calibrated to operate on the same origin as the IK system.

$$[1002.2,\ 514.1,\ 188.4,\ -77.37,\ 22.20,\ -43.38] \tag{6.2}$$

There are very positive results from the corresponding values in z first and foremost (which does have the same origin in both Nao's world frame and the IK system).

6.3.2 Configuration with negative locomotion and change in all orientations

This configuration is particularly interesting, because it displays how a pure mathematical based kinematic control will solve a desired configuration behind the robot. This differs in efficiency from how a human would solve the same task.

$q_{init} = [0.0,\ 0.0,\ 0.0,\ 93.2,\ 0.35,\ -0.171,\ 0.3,\ -1.39,\ -1.1]$
$x_{desired} = [-160,\ 180,\ 160,\ 0.262, 1.22, 0.175]$
$solveIK(q_{init},\ x_{desired})$

Which resulted in the following actual move:

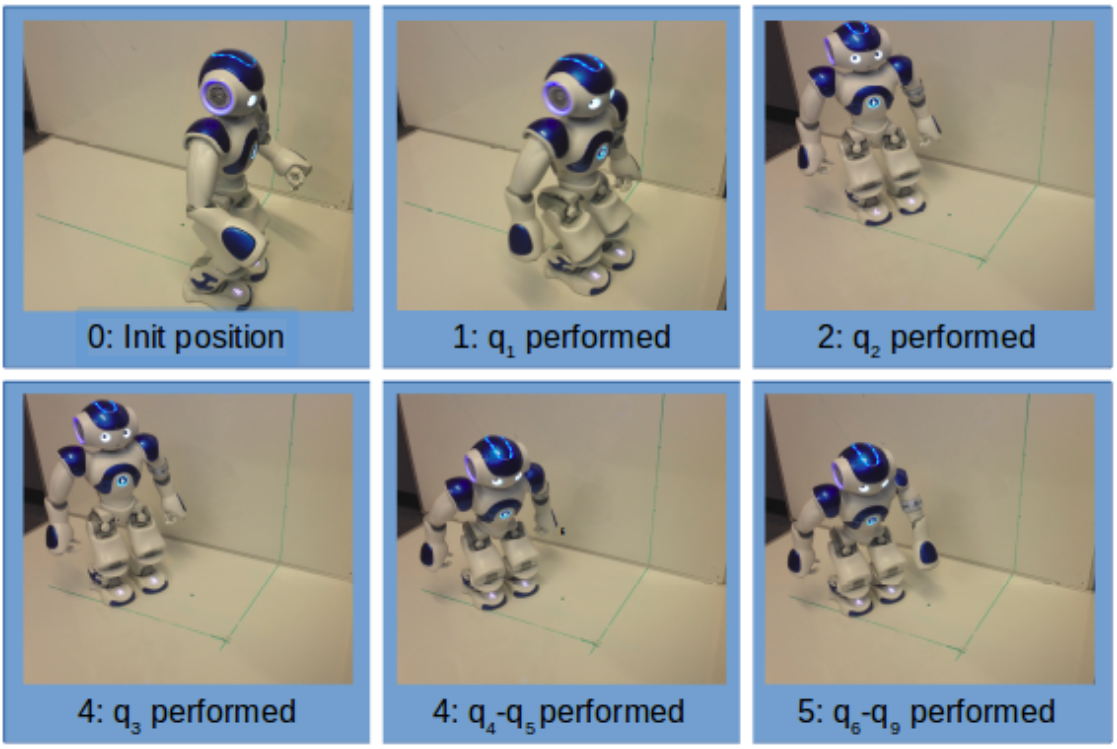

Figure 6.5: Nao moving to a configuration behind

Physical measurements

$$\begin{bmatrix} x \\ y \\ z \\ \phi \\ \theta \\ \psi \end{bmatrix}_{measured} = \begin{bmatrix} -188 \\ 165 \\ 160 \\ 23^o \\ 80^o \\ 18^o \end{bmatrix}, \quad \begin{bmatrix} x \\ y \\ z \\ \phi \\ \theta \\ \psi \end{bmatrix}_{desired} = \begin{bmatrix} -180 \\ 180 \\ 165 \\ 15^o \\ 70^o \\ 10^o \end{bmatrix} \Rightarrow \begin{bmatrix} 8mm \\ 15mm \\ 5mm \\ 8^o \\ 10^o \\ 8^o \end{bmatrix}_{error} \tag{6.3}$$

We see again errors in the range of $[cm]$ when Nao moves around. From this attempt we can, as in the previous one, get the impression that Nao moves quite well to the desired place. The detailed photos from this attempt was lost due to computer breakdown, unfortunately. This execution does therefore not count for more than

an indication of efficient handling of negative x-values. Nao's measurements were neither appropriate to assist any validation.

Nao's measurements

$$[-508.9,\ 481.2,\ 158.4,\ -58.33,\ 24.2,\ 44] \tag{6.4}$$

Yet again, Nao measures some very invalid data for x, y and ψ, but the measurements for z does underline that there are parts of the IK system that is correct.

6.4 Test of Accuracy

To test the accuracy of this control solution when Nao needs to walk or turn, Nao was made to perform an inverse kinematic operation with only a change in x, y and ψ, on a gridded surface. The surface consist of 20 $[mm]$ grids, and this is to simplify the process of taking physical measurements. The idea behind performing this test was born through the constant deviations and unwanted behavior in the three given states, which cannot be supported by the equally unviable sensory measurements. A sharp object was placed in Nao's end-effector (left hand) to enhance precision, and a camera was mounted on Nao's shoulder to capture the sharp object from a constant configuration.

$q_{init} = [0.0,\ 0.0,\ 0.0,\ 93.2,\ 0.35,\ -0.171,\ 0.3,\ -1.39,\ -1.1]$
$q_{desired} = [-\pi/8,\ 80.0,\ \pi/8,\ 93.2,\ 0.35,\ -0.171,\ 0.3,\ -1.39,\ -1.1]$
$solveIK(q_{init},\ \mathrm{FK}(q_{desired}))$

This corresponds to a configuration change of

$$\Delta \mathbf{x}_{desired} = \begin{bmatrix} -55.4\ mm \\ 22.9\ mm \\ 0^o \end{bmatrix} in \begin{bmatrix} x \\ y \\ \psi \end{bmatrix}. \tag{6.5}$$

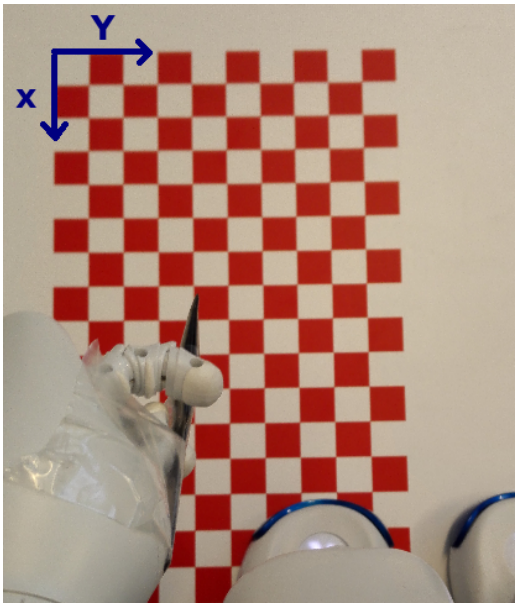

Figure 6.6: Init configuration for accuracy measurement

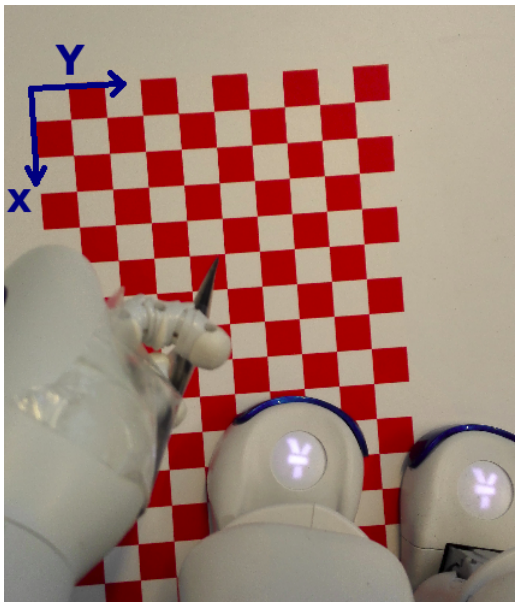

Figure 6.7: End configuration for accuracy measurement

The measured changes are:

$$init\ measure = \begin{bmatrix} 145\ mm \\ 81\ mm \\ 0^o \end{bmatrix}, \quad end\ measure = \begin{bmatrix} 102\ mm \\ 96\ mm \\ 19^o \end{bmatrix} \tag{6.6}$$

$$\Rightarrow \Delta\mathbf{x}_{measured} = \begin{bmatrix} -43\ mm \\ 15\ mm \\ 19^o \end{bmatrix} in \begin{bmatrix} x \\ y \\ \psi \end{bmatrix} \Rightarrow \Delta\mathbf{x}_{desired} - \Delta\mathbf{x}_{measured} = \begin{bmatrix} -12.4\ mm \\ 7.19\ mm \\ 19^o \end{bmatrix}. \tag{6.7}$$

This indicates very substantial deviations when Nao turns or walks. From many executions, it seem to depend strongly on how much friction he experiences (when turning especially). These errors were from the very first execution, and represents the larger deviations obtained, but intolerable discrepancies appear too often nonetheless.

6.5 Configuration Without Body Movement in the X-Y-Plane

In the view of the fact that body movement in the x-y-plane causes great inaccuracies, with the equipment at hand, a desired configuration that would yield no walking or turning was considered useful to show the algorithms correctness for both orientations and translations. This would also involve the possibility for direct tracking of Nao's measured versus the algorithms calculated left arm configuration. To visualize this in the foremost way, another plotting function was created to show the Cartesian coordinates in 3D, with vectors representing the corresponding orientations along the graph (See Appendix B). Nao was given a desired configuration determined by small changes in "hip-height" and left arm joints.

$q_{init} = [0.00,\ 0.00,\ 0.00,\ 93.20,\ 0.10,\ 1.40,\ 0.30,\ -1.39,\ -1.01]$
$q_{desired} = [0.00,\ 0.00,\ 0.00,\ 73.20,\ 0.10,\ 0.00,\ 0.10,\ -1.39,\ -0.51]$
$solveIK(q_{init}, \mathrm{FK}(q_{desired}))$

The performance is documented by initial configuration and end configuration below.

Figure 6.8: Init configuration with affixed lower body

Figure 6.9: End configuration with affixed lower body

The corresponding 3D plots of Cartesian coordinates and orientations is presented in Figure 6.10. The plot in Figure 6.11 is of the same experiment, but with a small offset to the x and y values of the generated configurations, to observe the trajectories and vectors more easily.

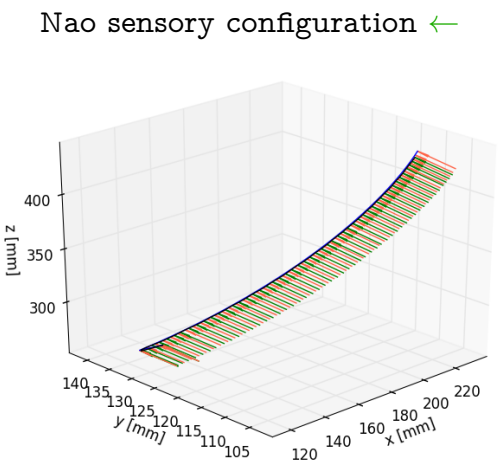

Figure 6.10: Cartesian trajectory with vectorial orientation

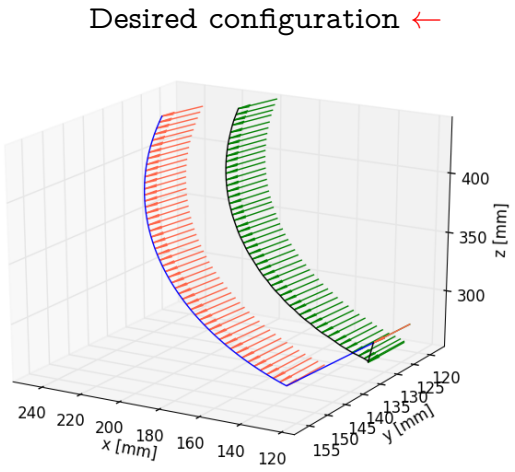

Figure 6.11: Cartesian trajectory with offset, and vectorial orientation

There is an abnormality at the very first registered point, which may be caused by an error in the program which refines the tracked data. This error is anyway

neglectable compared the crucial features this experiment sheds light on. There is a near perfect match between the generated configurations and Nao's measured configurations, for the left hand. This contributes good evidence pointing in the direction of correct mathematical modelling and inverse kinematic control of Nao. At the end, when the desire is to move the left arm high up, where the shoulder experiences approximately maximum torque, it fails to follow by 8 $[mm]$. To further illustrate physical shortcomings of Nao as a platform, a similar experiment was undertaken, but with movements yielding high torques on the robot.

$q_{init} = [0.00,\ 0.00,\ 0.00,\ 93.20,\ 0.10,\ 1.40,\ 0.30,\ -1.39,\ -1.01]$
$q_{desired} = [0.00,\ 0.00,\ 0.00,\ 53.20,\ 0.10,\ 2.00,\ 0.10,\ -1.39,\ -0.20]$
$solveIK(q_{init}, \mathrm{FK}(q_{desired}))$

This proved difficult for Nao to perform, inducing irregular joint movement in the left arm and small oscillations in both legs.

Nao sensory configuration ←

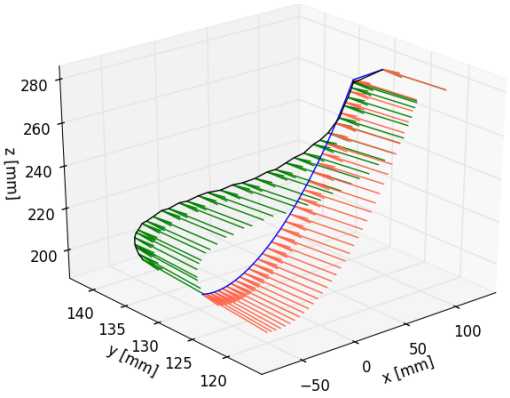

Figure 6.12: Cartesian trajectory with vectorial orientation showing physical shortcomings

Desired configuration ←

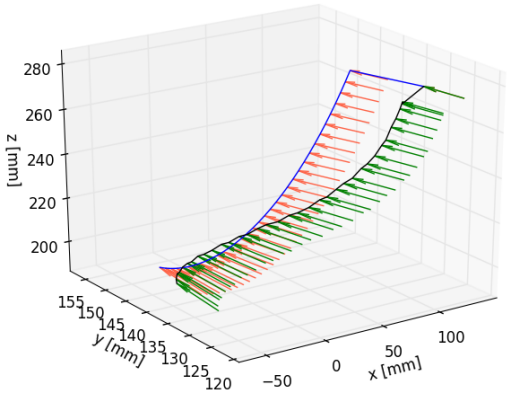

Figure 6.13: Cartesian trajectory with offset, and vectorial orientation showing physical shortcomings

The deviations are in the range of 10 $[mm]$ in y direction, caused by some joints moving different than commanded, and the oscillations are visible if one looks closely at the non-smooth curvature. Photographic documentation of this execution is provided in the given YouTube link, since the occurring problems are best visualized by a movie.

Chapter 7

Discussions

7.1 Simulations versus Experiments

An import remark has to be made from the authors point of view: the simulations provides the best scientific foundation for analysis of the theoretical results in this thesis. They exhibit error convergence, estimations of computational time, handling of singularities etc. in an explicit manner. When evaluating if the developed dynamical weightings, the singularity avoidance and the fuzzy logic provides new functionality to the scientific community within robotics and inverse kinematics, the theoretical work and simulations shows the honest and possibly uniform contributions, without hardware inadequacy, implementation errors and so on. The experiments participated to confirm the correctness of the inverse kinematic solution, implemented with Nao's physical features, but it also brought along obstacles regarding Nao's own OS, especially regarding command scheduling, and most importantly regarding tracking accuracy. The experimental results also gave an insight into the physical challenge of torque limitation - leading further into power consumption - which is not confronted by the simulations. For automated robots, particularly submerged ones with exposure to currents, wave and buoyancy, this is a very important matter, which is being addressed in the literature at present. When developing theoretical solvers for inverse kinematic problems, requirements for optimal trajectories relating to power consumption, alignment with currents and real time calculation of joint limits as a function of loading, are clearly important aspects to consider even in the early theoretical development. In addition, physical experiments in general, and this one in particular, are important tools to introduce interested people to the theoretical work, and to prove a physical applicability to motivate further exploration.

7.2 Inaccuracies

From the simple configurations in Sub-Chapter 6.5 excluding locomotion or turns, the IK program can control Nao to the same precision as his own IK functions.

Physical measurements infer large inaccuracies, which one need to account for when evaluating the experimental results. In this case, where validation of the IK solver being correct or not is the most important, physical measurements can be considered as sufficient. It was hard to measure down to less than $5\,[mm]$, maybe even $10\,[mm]$,

but this would nonetheless reveal malfunction or wrong theoretical development for configurations over $200 - 600\ [mm]$. The angular values are even harder to measure physically, but the correctness of the end-effectors orientation is more or less confirmed by the simulation in 6.10.

From the experiments in 6.3.1 and 6.4, there are very evident inaccuracies regarding the control of Nao without external sensory equipment, when walking or turning is required. Though, the measurements from Nao in z indicates a precision of the physical control in the range of $e_z < 10\ [mm]$, supported by both internal sensory and physical measurements. This can be considered as good control for a low-end robot with deviations from a single "set-angle" of up to 1^o. Another positive is that Nao provides way larger inaccuracies in controlling his own joint values, than the tolerable deviations from the IK solver.

7.3 Solution Validity and Accomplishment

To be able to discuss the validity and the degree of accomplishment of the solution, a retrospect to the objective is duly taken. Briefly repeated, it stated a desire to solve the inverse kinematic problem for a *vehicle-manipulator-system*, with singularity robustness and efficient configurations, within a few seconds. Having this in mind, the evaluation is approached with a list of negatives and positives from testing and simulation.

Positives

- *Every desired configuration within the configuration space, inputted to the program, have resulted in a solution of corresponding joint coordinates in less than 1500 iterations. 1500 iterations accounted for less than 2 seconds of computational time on a low-end computer.*
- The obtained solutions forms efficient joint trajectories.
- The solver handles mathematical singularities, and even prevents undesirable behavior in vicinity of them.
- The dynamic gains, or variable step-size, ensures quick convergence and stable joint trajectories.
- Nao can be controlled to reach any desired configuration within his configuration space.
- When Nao had sufficient grip to the ground and managed to behave as wanted, the results were very affirmative to the control systems correctness.
- Nao is a complex robot to control kinematically, and even better performance would likely be achieved on a wheel-based or underwater robot.

Table 7.1: Positive results from the IK solution

Negatives

- The system has not yet been verified with sensory equipment capable of detecting errors in the range of the solver's accuracy.
- It is impossible to control Nao's hand to follow a joint trajectory.
- Control of all six states at the same time is not achieved, which implies that eventual collision avoidance or other objectives of that sort may need to be concurrently controlled by another routine.

Table 7.2: Negative results from the IK solution

Chapter 8

Conclusions

8.1 Summary

This master thesis has presented a new way of solving the inverse kinematic problem for a vehicle-manipulator system, implemented on the humanoid robot Nao. The solution was based on the method of Damped Least Squares, but was extended with dynamical damping factors for each singular value, fuzzy logic to switch between different control states and dynamical weighting of every joint increment. It applies the dynamical damping by developing a singular value decomposition, and enforcing the singular values to follow a cosine function of the maximum damping, equal to 0.2, if the singular values drop below 0.15. The fuzzy logic is organized by "fuzzifying" the real-time error, before a series of vertices weight each control state's membership in the current configuration, and the crisp output of a fuzzy coefficient yielding which control system to be used for the next iteration. The dynamical weighting is based on a prediction for the derivatives of Cartesian coordinates and Euler angles, which is then used to create a corrected joint increment and a new derivative in Cartesian coordinates and Euler angles. The kinematic vehicle-manipulator system was established by a simplified model of the robot body, with two prismatic and three revolute joints. The forward kinematics and the Jacobian was obtained through symbolic mathematics in MATLAB. The solver is implemented in Python, and through Python it can communicate to Nao's native OS using the Naoqi package.

The implementation has been tested both by simulations and by physical experiments. The simulations have exhibited that the solver can obtain a joint set for a variety of desired configurations quickly, with every problem in the simulations or the experiments solved within 1500 iterations. This, in addition to the handling of singularities, the properties of the fuzzy logic-engined control switching and the preferable error convergence provided by the dynamic weighting. The physical experiments proved the correctness of the solver, at same time as it elevated problems of global tracking and integral inaccuracies in the Nao platform. The overall experience from physical testing was a validation of the solver's ability to create correct joint sets, and the inappropriateness of Nao as a platform for accurate execution.

8.2 Conclusions

The inverse kinematic solver has proved to be robust and very efficient, and a useful asset to solving inverse kinematic problems of vehicle-manipulator systems. The dynamic weighting of every joint increment, equivalent to a variable step-size, demonstrated a very nice contribution to enforce stable and quick convergence, and is to the authors knowledge a new feature in the field of Jacobian Control Methods. The method of switching between control states with fuzzy logic have neither been observed in the literature before, and it resulted in very quick solutions. As well, the Damped Least Squares method has proven itself as an useful tool to control vehicle-manipulator systems. This thesis has provided new way of solving the inverse kinematic problem for vehicle-manipulator systems, which is easy applicable to a wide range of systems, and which can now be explored on smaller underwater platforms. It has also made the solver available to everyone by using Python as a platform, enabling further enhancements of the control strategy or optimization of the code.

8.3 Recommendations for Future Work

External Sensors

From previous experience, tracking with a present high-end, kinematic control device attached to the robot will ensure very high precision when moving around in Cartesian space. This should be implemented with the IK control developed in this thesis, to obtain the ability to control a robot around in space with very high precision. Tracking could also be implemented with computer vision, lidar, sonar etc., or most preferred: in a sensory fusion of all the mentioned (e.g. with a Kalman filter (Kalman, 1960)). This is already the case for how underwater vehicles navigate, to mention ROV Minerva and its usage of *HiPap* as an example (NTNU-AUR-lab, Accessed 11.02.16). If simple, closed control loops use these measurements for the individual control states, one could easily do kinematic control of any vehicle with the work from this thesis.

Path Generation and Collision Avoidance

To use the inverse kinematic solver this thesis provides in an autonomous robotic environment, another control layer is required above. This control layer needs to generate collision free, desired configurations from the initial configurations. For a static environment, one could apply a A^* based path planner, which could even work for a multi-robot system (Jose and Pratihar, 2016), or some sort of probabilistic roadmap learning (Kavraki et al., 1996). Further down the road, one could imagine a real time path planner with collision avoidance to handle dynamic environments (Schwesinger et al., 2015). These path planners could generate $\mathbf{x}_{desired}$ continuously, while the inverse kinematic solver provided here will deliver corresponding joint coordinates to the robot.

Selective State Control

In many different scenarios one or several of the six states in $\mathbf{x}$ will be superfluous to the control objective. Imagine when picking up a small ball laying on the ground. The preferred way would be to pick up the ball with x and y equal the ball center, z equal the top of the ball and $\phi = \theta \approx 0$. The value of ψ however would be indifferent to a successful grip. Hence, ignoring ψ as a control state would leave additional redundancy to control the other states in an efficient manner, while still executing the desired task impeccably. An algorithm that could dynamically change the global set of control states, the size of n in $\mathbf{x}_{n\times 1} \in \mathbb{R}^n$ by mathematical terms, could render leaps in the direction of more humanoid kinematic control.

Bibliography

Aldebaran. *NAO Software 1.14.5 documentation*, 2015a. http://doc.aldebaran.com/1-14/nao/index.html [Accessed: 19.10.15].

Aldebaran. *Python SDK, Nao Software Documentation*, 2015b. http://doc.aldebaran.com/1-14/dev/python/index.html [Accessed: 25.02.16].

A. ANSI. *Safety Requirements*, 1991. http://www.paragonproducts-ia.com/documents/RIA%20R15_06-1999.pdf [Accessed: 08.03.16].

G. Antonelli and G. Antonelli. *Underwater robots*. Springer, 2014.

J. S. Beggs. *Kinematics*. CRC Press, 1983.

S. R. Buss. Introduction to inverse kinematics with jacobian transpose, pseudoinverse and damped least squares methods. *University of California, San Diego*, 2004.

S. R. Buss and J.-S. Kim. Selectively damped least squares for inverse kinematics. *journal of graphics, gpu, and game tools*, 10(3):37–49, 2005.

A. A. Canutescu and R. L. Dunbrack. Cyclic coordinate descent: A robotics algorithm for protein loop closure. *Protein science*, 12(5):963–972, 2003.

S. Chiaverini, O. Egeland, and R. Kanestrom. Achieving user-defined accuracy with damped least-squares inverse kinematics. In *Advanced Robotics, 1991.'Robots in Unstructured Environments', 91 ICAR., Fifth International Conference on*, pages 672–677. IEEE, 1991.

S. Chiaverini, G. Oriolo, and I. D. Walker. Kinematically redundant manipulators. In *Springer handbook of robotics*, pages 245–268. Springer, 2008.

H. Choset. Robotic motion planning: Configuration space. Institute of Robotics, Carnegie Mellon University, 2014. http://www.cs.cmu.edu/~motionplanning/lecture/Chap3-Config-Space_howie.pdf [Accessed: 09.02.16].

J. J. Craig. *Introduction to Robotics, Mechanics and Control*. Pearson Education International, Upper Saddle River, NJ 07458, 3 edition, 2005. ISBN 0131236296.

J. Denavit. A kinematic notation for lower-pair mechanisms based on matrices. *Trans. of the ASME. Journal of Applied Mechanics*, 22:215–221, 1955.

R. Dewar. Case study: Mda - canadian space arm. AdaCore, 2011. http://www.adacore.com/uploads/customers/CaseStudy_SpaceArm.pdf [Accessed: 11.02.16].

P. Donelan. *Singularities of Robot Manipulators*. School of Mathematics, Statistics and Computer Science, Victoria University of Wellington, 2006. http://homepages.ecs.vuw.ac.nz/~donelan/papers/srm.pdf [Accessed: 09.03.16].

O. Egeland, J. R. Sagli, I. Spangelo, and S. Chiaverini. A damped least-squares solution to redundancy resolution. In *Robotics and Automation, 1991. Proceedings., 1991 IEEE International Conference on*, pages 945–950. IEEE, 1991.

L. Eldén. A weighted pseudoinverse, generalized singular values, and constrained least squares problems. *BIT Numerical Mathematics*, 22(4):487–502, 1982.

F. Fahimi. *Autonomous robots: modeling, path planning, and control*, volume 107. Springer Science & Business Media, 2008.

Fiskeridepartementet. *Fiskeri og Havbruk - fremtidens næring i Norge*, 2015. https://www.regjeringen.no/no/aktuelt/fiskeri_og_havbruk_fremtidens/id263754/ [Accessed: 08.02.16].

J. Fodor and R. Fullér. *Advances in Soft Computing, Intelligent Robotics and Control*, volume 8. Springer Science & Business Media, 2014.

P. J. From, J. T. Gravdahl, and K. Y. Pettersen. *Vehicle-Manipulator Systems*. Springer, 2014.

G. H. Golub and C. Reinsch. Singular value decomposition and least squares solutions. *Numerische mathematik*, 14(5):403–420, 1970.

P. Hajek. Fuzzy logic. In E. N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Fall 2010 edition, 2010.

M. Hayes, M. Husty, and P. Zsombor-Murray. Singular configurations of wrist-partitioned 6r serial robots: a geometric perspective for users. *Transactions of the Canadian Society for Mechanical Engineering 26.1 (2002): 41-55*, 2003. http://faculty.mae.carleton.ca/John_Hayes/Papers/KR15SingCSME.pdf [Accessed: 17.03.15].

K. Jose and D. K. Pratihar. Task allocation and collision-free path planning of centralized multi-robots system for industrial plant inspection using heuristic methods. *Robotics and Autonomous Systems*, 80:34 – 42, 2016. ISSN 0921-8890. doi: http://dx.doi.org/10.1016/j.robot.2016.02.003. URL http://www.sciencedirect.com/science/article/pii/S0921889016000282.

N. Joubert. Numerical methods for inverse kinematics. *UC Berkeley*, 2008.

R. E. Kalman. A new approach to linear filtering and prediction problems. *Transactions of the ASME–Journal of Basic Engineering*, 82(Series D):35–45, 1960.

W. Kaplan. Advanced calculus. 2002.

L. E. Kavraki, P. Svestka, J. C. Latombe, and M. H. Overmars. Probabilistic roadmaps for path planning in high-dimensional configuration spaces. *IEEE Transactions on Robotics and Automation*, 12(4):566–580, Aug 1996. ISSN 1042-296X. doi: 10.1109/70.508439.

O. Khatib. Jacobians - introduction to robotics. *Dept. of Computer Science, Stanford University*, 2008.

C. Klein, C.-H. Huang, et al. Review of pseudoinverse control for use with kinematically redundant manipulators. *Systems, Man and Cybernetics, IEEE Transactions on*, (2):245–250, 1983.

N. Kofinas, E. Orfanoudakis, and M. G. Lagoudakis. Complete analytical forward and inverse kinematics for the nao humanoid robot. *Journal of Intelligent & Robotic Systems*, 77(2):251–264, 2015.

J. U. Korein, N. I. Badler, et al. Techniques for generating the goal-directed motion of articulated structures. *IEEE Computer Graphics and Applications*, 2(9):71–81, 1982.

S. Kucuk and Z. Bingul. *Robot kinematics: forward and inverse kinematics*. INTECH Open Access Publisher, 2006.

N. MathWorks. *Solvers*, 2016. `http://se.mathworks.com/help/simulink/ug/choosing-a-solver.html` [Accessed: 12.05.15].

J. M. McCarthy. *Introduction to theoretical kinematics*. MIT press, 1990.

A. Minsaas. Havteknologi - kan havbruk høste fra offshore og maritime næringer. Presented at TEKMAR, Trondheim, 2015. URL `http://www.tekmar.no/`.

E. H. Moore. On the reciprocal of the general algebraic matrix, abstract. *Bull. Amer. Math. Soc*, 26:394–395, 1920.

R. Mukundan. A fast inverse kinematics solution for an n-link joint chain. 2008.

R. M. Murray, Z. Li, and S. S. Sastry. *A Mathematical Introduction to Robotic Manipulation*, volume 107. CRC Press, 1994.

Y. Nakamura and H. Hanafusa. Inverse kinematic solutions with singularity robustness for robot manipulator control. *Journal of dynamic systems, measurement, and control*, 108(3):163–171, 1986.

NTNU-AUR-lab. Vehicles, Accessed 11.02.16. URL `http://www.ntnu.edu/aur-lab/vehicles`.

Numpy-Developers. Numpy, Accessed 25.02.16. URL `http://www.numpy.org/`.

A. N. Pechev. Inverse kinematics without matrix inversion. In *Robotics and Automation, 2008. ICRA 2008. IEEE International Conference on*, pages 2005–2012. IEEE, 2008.

D. L. Peiper. The kinematics of manipulators under computer control. Technical report, DTIC Document, 1968.

R. Penrose. A generalized inverse for matrices. *Mathematical Proceedings of the Cambridge Philosophical Society*, 51:406–413, 7 1955. ISSN 1469-8064. doi: 10.1017/S0305004100030401. URL `http://journals.cambridge.org/article_S0305004100030401`.

M.-H. Perng and L. Hsiao. Inverse kinematic solutions for a fully parallel robot with singularity robustness. *The international journal of Robotics Research*, 18 (6):575–583, 1999.

A. Popa. *What is ment by the term gimball lock?* Hughes Research Laboratories, 1998. `http://www.madsci.org/posts/archives/aug98/896993617.Eg.r.html` [Accessed: 04.03.16].

Python-Foundation. *Python, About*, 2016. `https://www.python.org/about/` [Accessed: 25.02.16].

S. Schaal. Jacobian methods for inverse kinematics and planning. Max Planck Institute for Intelligent Systems, 04 2014. URL `https://homes.cs.washington.edu/~todorov/courses/cseP590/06_JacobianMethods.pdf`.

I. Schjølberg. Hav til folket. Ocean Week, 2016. URL `https://www.ntnu.edu/web/ocean-week/news`.

U. Schwesinger, R. Siegwart, and P. Furgale. Fast collision detection through bounding volume hierarchies in workspace-time space for sampling-based motion planners. In *Robotics and Automation (ICRA), 2015 IEEE International Conference on*, pages 63–68. IEEE, 2015.

L. Sciavicco and B. Siciliano. A solution algorithm to the inverse kinematic problem for redundant manipulators. *Robotics and Automation, IEEE Journal of*, 4(4): 403–410, 1988.

B. Siciliano and O. Khatib. *Springer handbook of robotics*. Springer Science & Business Media, 2008.

G. G. Slabaugh. Computing euler angles from a rotation matrix. *Retrieved on August*, 6(2000):39–63, 1999.

C. W. Wampler et al. Manipulator inverse kinematic solutions based on vector formulations and damped least-squares methods. *Systems, Man and Cybernetics, IEEE Transactions on*, 16(1):93–101, 1986.

L.-C. T. Wang and C. C. Chen. A combined optimization method for solving the inverse kinematics problems of mechanical manipulators. *Robotics and Automation, IEEE Transactions on*, 7(4):489–499, 1991.

E. W. Weisstein. *Predictor-Corrector Methods. MathWorld* - A Wolfram Web Resource, 2016. `http://mathworld.wolfram.com/Predictor-CorrectorMethods.html` [Accessed: 12.05.15].

D. E. Whitney. The mathematics of coordinated control of prosthetic arms and manipulators. *Journal of Dynamic Systems, Measurement, and Control*, 94(4): 303–309, 1972.

Y. Xu, R. P. Paul, and H.-Y. Shum. Fuzzy control of robot and compliant wrist system. In *1991 Conference Record of the Industry Applications Society Annual Meeting*, pages 1431–1437, 1991.

L. A. Zadeh. Fuzzy sets. *Information and control*, 8(3):338–353, 1965.

Appendix A

Underlying in this Appendix are the needed MATLAB scripts to perform some of the simple analyses used in the thesis, and to obtain crucial elements such as the Jacobian matrix (by symbolic mathematics).

Obtaining the Jacobian

```
% Finding the Jacobian and the Jacobian Pseudoinverse, for NAO system
% February 9, 2016, by AAsmund P. Hugo

% Body
syms q1; syms d2; syms q3; syms d4; syms q5
% Left Arm
syms q6; syms q7; syms q8; syms q9
syms l3; syms b3; syms z_s; syms y_s; syms z_off; syms z_h; x_h

% Rotation Matrix of pi/2
rotZ = [0 -1 0 0; 1 0 0 0; 0 0 1 0; 0 0 0 1];

% Establishing the affine transformation from base to left hand
T01 = [[cos(q1) -sin(q1) 0 0];[sin(q1) cos(q1) 0 0];[0 0 1 0];[0 0 0 1]];
T01_ = T01*rotZ;
T12 = [[1 0 0 0];[0 0 -1 -d2];[0 1 0 0];[0 0 0 1]];
T23 = [[cos(q3) -sin(q3) 0 0];[0 0 1 0];[-sin(q3) -cos(q3) 0 0];[0 0 0 1]];
T34 = [[0 -1 0 0];[1 0 0 0];[0 0 1 d4];[0 0 0 1]];
T45 = [[cos(q5) -sin(q5) 0 0];[0 0 -1 0];[sin(q5) cos(q5) 0 0];[0 0 0 1]];
R5 = [[-1 0 0 0];[0 0 1 0];[0 1 0 0];[0 0 0 1]];
A5 = [[1 0 0 0];[0 1 0 0];[0 0 1 z_off];[0 0 0 1]];

Ab = [[1 0 0 0];[0 1 0 y_s];[0 0 1 z_s];[0 0 0 1]];
T56 = [[cos(q6) -sin(q6) 0 0];[0 0 1 0];[-sin(q6) -cos(q6) 0 0];[0 0 0 1]];
```

```
T56_ = T56*rotZ;
T67 = [[cos(q7) -sin(q7) 0 0];[0 0 -1 0];[sin(q7) cos(q7) 0 0];[0 0 0 1]];
T67_ = T67*rotZ;
T78 = [[cos(q8) -sin(q8) 0 b3];[0 0 -1 -l3];[sin(q8) cos(q8) 0 0];[0 0 0 1]];
T89 = [[cos(q9) -sin(q9) 0 0];[0 0 1 0];[-sin(q9) -cos(q9) 0 0];[0 0 0 1]];
R9 = [[0 1 0 0];[0 0 1 0];[1 0 0 0];[0 0 0 1]];
A9 = [[1 0 0 x_h];[0 1 0 0];[0 0 1 z_h];[0 0 0 1]];

T02 = T01_*T12;
T03 = T02*T23;
T04 = T03*T34;
T05 = T04*T45;
T06 = T05*R5*A5*Ab*T56;
T07 = T06*rotZ*T67;
T08 = T07*rotZ*T78;
T09 = T08*T89;
TSys = T09*R9*A9;
TSys(1,3)

% Extracting and differentiating to obtain J
m = 6; % Because of operation in 3D
n = 9; % Number of joints
J = sym(zeros(m,n));
joints = [q1,d2,q3,d4,q5,q6,q7,q8,q9];

% Obtaining Jv (end effector translation velocity)
Jv = sym(zeros(m/2,1));
for i=1:n
    Jv(:,i) = jacobian(TSys(1:3,4),joints(i));
end
% Obtaining Jw (end effector rotation velocity)
Jw = sym(zeros(m/2,n));
Jw(1:3,1) = T01(1:3,3);
Jw(1:3,2) = T02(1:3,3)*0;
Jw(1:3,3) = T03(1:3,3);
Jw(1:3,4) = T04(1:3,3)*0;
Jw(1:3,5) = T05(1:3,3);
Jw(1:3,6) = T06(1:3,3);
Jw(1:3,7) = T07(1:3,3);
Jw(1:3,8) = T08(1:3,3);
Jw(1:3,9) = T09(1:3,3);

J = [Jv;Jw]
```

The Jacobian

```
1
2 Jacobian =
3
4 [[z_h*(sin(q8)*(cos(q7)*(cos(q1)*cos(q3) - sin(q1)*sin(q3))
   + sin(q7)*(cos(q5)*sin(q6)*(cos(q1)*sin(q3) + cos(q3)*sin
   (q1)) + cos(q6)*sin(q5)*(cos(q1)*sin(q3) + cos(q3)*sin(q1
   )))) - cos(q8)*(cos(q5)*cos(q6)*(cos(q1)*sin(q3) + cos(q3
   )*sin(q1)) - sin(q5)*sin(q6)*(cos(q1)*sin(q3) + cos(q3)*
   sin(q1)))) - y_s*(cos(q1)*cos(q3) - sin(q1)*sin(q3)) -
   x_h*(cos(q9)*(sin(q7)*(cos(q1)*cos(q3) - sin(q1)*sin(q3))
    - cos(q7)*(cos(q5)*sin(q6)*(cos(q1)*sin(q3) + cos(q3)*
   sin(q1)) + cos(q6)*sin(q5)*(cos(q1)*sin(q3) + cos(q3)*sin
   (q1)))) + sin(q9)*(cos(q8)*(cos(q7)*(cos(q1)*cos(q3) -
   sin(q1)*sin(q3)) + sin(q7)*(cos(q5)*sin(q6)*(cos(q1)*sin(
   q3) + cos(q3)*sin(q1)) + cos(q6)*sin(q5)*(cos(q1)*sin(q3)
    + cos(q3)*sin(q1)))) + sin(q8)*(cos(q5)*cos(q6)*(cos(q1)
   *sin(q3) + cos(q3)*sin(q1)) - sin(q5)*sin(q6)*(cos(q1)*
   sin(q3) + cos(q3)*sin(q1))))) - d2*sin(q1) - b3*(cos(q7)
   *(cos(q1)*cos(q3) - sin(q1)*sin(q3)) + sin(q7)*(cos(q5)*
   sin(q6)*(cos(q1)*sin(q3) + cos(q3)*sin(q1)) + cos(q6)*sin
   (q5)*(cos(q1)*sin(q3) + cos(q3)*sin(q1)))) - l3*(sin(q7)
   *(cos(q1)*cos(q3) - sin(q1)*sin(q3)) - cos(q7)*(cos(q5)*
   sin(q6)*(cos(q1)*sin(q3) + cos(q3)*sin(q1)) + cos(q6)*sin
   (q5)*(cos(q1)*sin(q3) + cos(q3)*sin(q1)))) - z_off*sin(q5
   )*(cos(q1)*sin(q3) + cos(q3)*sin(q1)) - z_s*sin(q5)*(cos(
   q1)*sin(q3) + cos(q3)*sin(q1)), cos(q1), z_h*(sin(q8)*(
   cos(q7)*(cos(q1)*cos(q3) - sin(q1)*sin(q3)) + sin(q7)*(
   cos(q5)*sin(q6)*(cos(q1)*sin(q3) + cos(q3)*sin(q1)) + cos
   (q6)*sin(q5)*(cos(q1)*sin(q3) + cos(q3)*sin(q1)))) - cos(
   q8)*(cos(q5)*cos(q6)*(cos(q1)*sin(q3) + cos(q3)*sin(q1))
   - sin(q5)*sin(q6)*(cos(q1)*sin(q3) + cos(q3)*sin(q1)))) -
    y_s*(cos(q1)*cos(q3) - sin(q1)*sin(q3)) - x_h*(cos(q9)*(
   sin(q7)*(cos(q1)*cos(q3) - sin(q1)*sin(q3)) - cos(q7)*(
   cos(q5)*sin(q6)*(cos(q1)*sin(q3) + cos(q3)*sin(q1)) + cos
   (q6)*sin(q5)*(cos(q1)*sin(q3) + cos(q3)*sin(q1)))) + sin(
   q9)*(cos(q8)*(cos(q7)*(cos(q1)*cos(q3) - sin(q1)*sin(q3))
    + sin(q7)*(cos(q5)*sin(q6)*(cos(q1)*sin(q3) + cos(q3)*
   sin(q1)) + cos(q6)*sin(q5)*(cos(q1)*sin(q3) + cos(q3)*sin
   (q1)))) + sin(q8)*(cos(q5)*cos(q6)*(cos(q1)*sin(q3) + cos
   (q3)*sin(q1)) - sin(q5)*sin(q6)*(cos(q1)*sin(q3) + cos(q3
   )*sin(q1))))) - b3*(cos(q7)*(cos(q1)*cos(q3) - sin(q1)*
   sin(q3)) + sin(q7)*(cos(q5)*sin(q6)*(cos(q1)*sin(q3) +
   cos(q3)*sin(q1)) + cos(q6)*sin(q5)*(cos(q1)*sin(q3) + cos
   (q3)*sin(q1)))) - l3*(sin(q7)*(cos(q1)*cos(q3) - sin(q1)*
   sin(q3)) - cos(q7)*(cos(q5)*sin(q6)*(cos(q1)*sin(q3) +
   cos(q3)*sin(q1)) + cos(q6)*sin(q5)*(cos(q1)*sin(q3) + cos
   (q3)*sin(q1)))) - z_off*sin(q5)*(cos(q1)*sin(q3) + cos(q3
   )*sin(q1)) - z_s*sin(q5)*(cos(q1)*sin(q3) + cos(q3)*sin(
```

```
q1)), 0, z_off*cos(q5)*(cos(q1)*cos(q3) - sin(q1)*sin(q3)
) - z_h*(cos(q8)*(cos(q5)*sin(q6)*(cos(q1)*cos(q3) - sin(
q1)*sin(q3)) + cos(q6)*sin(q5)*(cos(q1)*cos(q3) - sin(q1)
*sin(q3))) + sin(q7)*sin(q8)*(cos(q5)*cos(q6)*(cos(q1)*
cos(q3) - sin(q1)*sin(q3)) - sin(q5)*sin(q6)*(cos(q1)*cos
(q3) - sin(q1)*sin(q3)))) - x_h*(sin(q9)*(sin(q8)*(cos(q5
)*sin(q6)*(cos(q1)*cos(q3) - sin(q1)*sin(q3)) + cos(q6)*
sin(q5)*(cos(q1)*cos(q3) - sin(q1)*sin(q3))) - cos(q8)*
sin(q7)*(cos(q5)*cos(q6)*(cos(q1)*cos(q3) - sin(q1)*sin(
q3)) - sin(q5)*sin(q6)*(cos(q1)*cos(q3) - sin(q1)*sin(q3)
))) + cos(q7)*cos(q9)*(cos(q5)*cos(q6)*(cos(q1)*cos(q3) -
 sin(q1)*sin(q3)) - sin(q5)*sin(q6)*(cos(q1)*cos(q3) -
sin(q1)*sin(q3)))) + z_s*cos(q5)*(cos(q1)*cos(q3) - sin(
q1)*sin(q3)) - l3*cos(q7)*(cos(q5)*cos(q6)*(cos(q1)*cos(
q3) - sin(q1)*sin(q3)) - sin(q5)*sin(q6)*(cos(q1)*cos(q3)
 - sin(q1)*sin(q3))) + b3*sin(q7)*(cos(q5)*cos(q6)*(cos(
q1)*cos(q3) - sin(q1)*sin(q3)) - sin(q5)*sin(q6)*(cos(q1)
*cos(q3) - sin(q1)*sin(q3))), b3*sin(q7)*(cos(q5)*cos(q6)
*(cos(q1)*cos(q3) - sin(q1)*sin(q3)) - sin(q5)*sin(q6)*(
cos(q1)*cos(q3) - sin(q1)*sin(q3))) - z_h*(cos(q8)*(cos(
q5)*sin(q6)*(cos(q1)*cos(q3) - sin(q1)*sin(q3)) + cos(q6)
*sin(q5)*(cos(q1)*cos(q3) - sin(q1)*sin(q3))) + sin(q7)*
sin(q8)*(cos(q5)*cos(q6)*(cos(q1)*cos(q3) - sin(q1)*sin(
q3)) - sin(q5)*sin(q6)*(cos(q1)*cos(q3) - sin(q1)*sin(q3)
))) - l3*cos(q7)*(cos(q5)*cos(q6)*(cos(q1)*cos(q3) - sin(
q1)*sin(q3)) - sin(q5)*sin(q6)*(cos(q1)*cos(q3) - sin(q1)
*sin(q3))) - x_h*(sin(q9)*(sin(q8)*(cos(q5)*sin(q6)*(cos(
q1)*cos(q3) - sin(q1)*sin(q3)) + cos(q6)*sin(q5)*(cos(q1)
*cos(q3) - sin(q1)*sin(q3))) - cos(q8)*sin(q7)*(cos(q5)*
cos(q6)*(cos(q1)*cos(q3) - sin(q1)*sin(q3)) - sin(q5)*sin
(q6)*(cos(q1)*cos(q3) - sin(q1)*sin(q3)))) + cos(q7)*cos(
q9)*(cos(q5)*cos(q6)*(cos(q1)*cos(q3) - sin(q1)*sin(q3))
- sin(q5)*sin(q6)*(cos(q1)*cos(q3) - sin(q1)*sin(q3)))),
b3*(sin(q7)*(cos(q1)*sin(q3) + cos(q3)*sin(q1)) + cos(q7)
*(cos(q5)*sin(q6)*(cos(q1)*cos(q3) - sin(q1)*sin(q3)) +
cos(q6)*sin(q5)*(cos(q1)*cos(q3) - sin(q1)*sin(q3)))) -
x_h*(cos(q9)*(cos(q7)*(cos(q1)*sin(q3) + cos(q3)*sin(q1))
 - sin(q7)*(cos(q5)*sin(q6)*(cos(q1)*cos(q3) - sin(q1)*
sin(q3)) + cos(q6)*sin(q5)*(cos(q1)*cos(q3) - sin(q1)*sin
(q3)))) - cos(q8)*sin(q9)*(sin(q7)*(cos(q1)*sin(q3) + cos
(q3)*sin(q1)) + cos(q7)*(cos(q5)*sin(q6)*(cos(q1)*cos(q3)
 - sin(q1)*sin(q3)) + cos(q6)*sin(q5)*(cos(q1)*cos(q3) -
sin(q1)*sin(q3))))) - l3*(cos(q7)*(cos(q1)*sin(q3) + cos(
q3)*sin(q1)) - sin(q7)*(cos(q5)*sin(q6)*(cos(q1)*cos(q3)
- sin(q1)*sin(q3)) + cos(q6)*sin(q5)*(cos(q1)*cos(q3) -
sin(q1)*sin(q3)))) - z_h*sin(q8)*(sin(q7)*(cos(q1)*sin(q3
) + cos(q3)*sin(q1)) + cos(q7)*(cos(q5)*sin(q6)*(cos(q1)*
cos(q3) - sin(q1)*sin(q3)) + cos(q6)*sin(q5)*(cos(q1)*cos
```

```
(q3) - sin(q1)*sin(q3)))),    z_h*(cos(q8)*(cos(q7)*(cos(
q1)*sin(q3) + cos(q3)*sin(q1)) - sin(q7)*(cos(q5)*sin(q6)
*(cos(q1)*cos(q3) - sin(q1)*sin(q3)) + cos(q6)*sin(q5)*(
cos(q1)*cos(q3) - sin(q1)*sin(q3)))) - sin(q8)*(cos(q5)*
cos(q6)*(cos(q1)*cos(q3) - sin(q1)*sin(q3)) - sin(q5)*sin
(q6)*(cos(q1)*cos(q3) - sin(q1)*sin(q3)))) + x_h*sin(q9)
*(sin(q8)*(cos(q7)*(cos(q1)*sin(q3) + cos(q3)*sin(q1)) -
sin(q7)*(cos(q5)*sin(q6)*(cos(q1)*cos(q3) - sin(q1)*sin(
q3)) + cos(q6)*sin(q5)*(cos(q1)*cos(q3) - sin(q1)*sin(q3)
))) + cos(q8)*(cos(q5)*cos(q6)*(cos(q1)*cos(q3) - sin(q1)
*sin(q3)) - sin(q5)*sin(q6)*(cos(q1)*cos(q3) - sin(q1)*
sin(q3)))),  x_h*(sin(q9)*(sin(q7)*(cos(q1)*sin(q3) + cos
(q3)*sin(q1)) + cos(q7)*(cos(q5)*sin(q6)*(cos(q1)*cos(q3)
 - sin(q1)*sin(q3)) + cos(q6)*sin(q5)*(cos(q1)*cos(q3) -
sin(q1)*sin(q3)))) - cos(q9)*(cos(q8)*(cos(q7)*(cos(q1)*
sin(q3) + cos(q3)*sin(q1)) - sin(q7)*(cos(q5)*sin(q6)*(
cos(q1)*cos(q3) - sin(q1)*sin(q3)) + cos(q6)*sin(q5)*(cos
(q1)*cos(q3) - sin(q1)*sin(q3)))) - sin(q8)*(cos(q5)*cos(
q6)*(cos(q1)*cos(q3) - sin(q1)*sin(q3)) - sin(q5)*sin(q6)
*(cos(q1)*cos(q3) - sin(q1)*sin(q3))))),
[z_h*(sin(q8)*(cos(q7)*(cos(q1)*sin(q3) + cos(q3)*sin(q1)) -
 sin(q7)*(cos(q5)*sin(q6)*(cos(q1)*cos(q3) - sin(q1)*sin(
q3)) + cos(q6)*sin(q5)*(cos(q1)*cos(q3) - sin(q1)*sin(q3)
))) + cos(q8)*(cos(q5)*cos(q6)*(cos(q1)*cos(q3) - sin(q1)
*sin(q3)) - sin(q5)*sin(q6)*(cos(q1)*cos(q3) - sin(q1)*
sin(q3)))) - y_s*(cos(q1)*sin(q3) + cos(q3)*sin(q1)) -
x_h*(cos(q9)*(sin(q7)*(cos(q1)*sin(q3) + cos(q3)*sin(q1))
 + cos(q7)*(cos(q5)*sin(q6)*(cos(q1)*cos(q3) - sin(q1)*
sin(q3)) + cos(q6)*sin(q5)*(cos(q1)*cos(q3) - sin(q1)*sin
(q3)))) + sin(q9)*(cos(q8)*(cos(q7)*(cos(q1)*sin(q3) +
cos(q3)*sin(q1)) - sin(q7)*(cos(q5)*sin(q6)*(cos(q1)*cos(
q3) - sin(q1)*sin(q3)) + cos(q6)*sin(q5)*(cos(q1)*cos(q3)
 - sin(q1)*sin(q3)))) - sin(q8)*(cos(q5)*cos(q6)*(cos(q1)
*cos(q3) - sin(q1)*sin(q3)) - sin(q5)*sin(q6)*(cos(q1)*
cos(q3) - sin(q1)*sin(q3))))) + d2*cos(q1) - b3*(cos(q7)
*(cos(q1)*sin(q3) + cos(q3)*sin(q1)) - sin(q7)*(cos(q5)*
sin(q6)*(cos(q1)*cos(q3) - sin(q1)*sin(q3)) + cos(q6)*sin
(q5)*(cos(q1)*cos(q3) - sin(q1)*sin(q3)))) - l3*(sin(q7)
*(cos(q1)*sin(q3) + cos(q3)*sin(q1)) + cos(q7)*(cos(q5)*
sin(q6)*(cos(q1)*cos(q3) - sin(q1)*sin(q3)) + cos(q6)*sin
(q5)*(cos(q1)*cos(q3) - sin(q1)*sin(q3)))) + z_off*sin(q5
)*(cos(q1)*cos(q3) - sin(q1)*sin(q3)) + z_s*sin(q5)*(cos(
q1)*cos(q3) - sin(q1)*sin(q3)), sin(q1), z_h*(sin(q8)*(
cos(q7)*(cos(q1)*sin(q3) + cos(q3)*sin(q1)) - sin(q7)*(
cos(q5)*sin(q6)*(cos(q1)*cos(q3) - sin(q1)*sin(q3)) + cos
(q6)*sin(q5)*(cos(q1)*cos(q3) - sin(q1)*sin(q3)))) + cos(
q8)*(cos(q5)*cos(q6)*(cos(q1)*cos(q3) - sin(q1)*sin(q3))
- sin(q5)*sin(q6)*(cos(q1)*cos(q3) - sin(q1)*sin(q3)))) -
```

```
 y_s*(cos(q1)*sin(q3) + cos(q3)*sin(q1)) - x_h*(cos(q9)*(
sin(q7)*(cos(q1)*sin(q3) + cos(q3)*sin(q1)) + cos(q7)*(
cos(q5)*sin(q6)*(cos(q1)*cos(q3) - sin(q1)*sin(q3)) + cos
(q6)*sin(q5)*(cos(q1)*cos(q3) - sin(q1)*sin(q3)))) + sin(
q9)*(cos(q8)*(cos(q7)*(cos(q1)*sin(q3) + cos(q3)*sin(q1))
 - sin(q7)*(cos(q5)*sin(q6)*(cos(q1)*cos(q3) - sin(q1)*
sin(q3)) + cos(q6)*sin(q5)*(cos(q1)*cos(q3) - sin(q1)*sin
(q3)))) - sin(q8)*(cos(q5)*cos(q6)*(cos(q1)*cos(q3) - sin
(q1)*sin(q3)) - sin(q5)*sin(q6)*(cos(q1)*cos(q3) - sin(q1
)*sin(q3))))) - b3*(cos(q7)*(cos(q1)*sin(q3) + cos(q3)*
sin(q1)) - sin(q7)*(cos(q5)*sin(q6)*(cos(q1)*cos(q3) -
sin(q1)*sin(q3)) + cos(q6)*sin(q5)*(cos(q1)*cos(q3) - sin
(q1)*sin(q3)))) - l3*(sin(q7)*(cos(q1)*sin(q3) + cos(q3)*
sin(q1)) + cos(q7)*(cos(q5)*sin(q6)*(cos(q1)*cos(q3) -
sin(q1)*sin(q3)) + cos(q6)*sin(q5)*(cos(q1)*cos(q3) - sin
(q1)*sin(q3)))) + z_off*sin(q5)*(cos(q1)*cos(q3) - sin(q1
)*sin(q3)) + z_s*sin(q5)*(cos(q1)*cos(q3) - sin(q1)*sin(
q3)), 0, z_off*cos(q5)*(cos(q1)*sin(q3) + cos(q3)*sin(q1)
) - z_h*(cos(q8)*(cos(q5)*sin(q6)*(cos(q1)*sin(q3) + cos(
q3)*sin(q1)) + cos(q6)*sin(q5)*(cos(q1)*sin(q3) + cos(q3)
*sin(q1))) + sin(q7)*sin(q8)*(cos(q5)*cos(q6)*(cos(q1)*
sin(q3) + cos(q3)*sin(q1)) - sin(q5)*sin(q6)*(cos(q1)*sin
(q3) + cos(q3)*sin(q1)))) - x_h*(sin(q9)*(sin(q8)*(cos(q5
)*sin(q6)*(cos(q1)*sin(q3) + cos(q3)*sin(q1)) + cos(q6)*
sin(q5)*(cos(q1)*sin(q3) + cos(q3)*sin(q1))) - cos(q8)*
sin(q7)*(cos(q5)*cos(q6)*(cos(q1)*sin(q3) + cos(q3)*sin(
q1)) - sin(q5)*sin(q6)*(cos(q1)*sin(q3) + cos(q3)*sin(q1)
))) + cos(q7)*cos(q9)*(cos(q5)*cos(q6)*(cos(q1)*sin(q3) +
 cos(q3)*sin(q1)) - sin(q5)*sin(q6)*(cos(q1)*sin(q3) +
cos(q3)*sin(q1)))) + z_s*cos(q5)*(cos(q1)*sin(q3) + cos(
q3)*sin(q1)) - l3*cos(q7)*(cos(q5)*cos(q6)*(cos(q1)*sin(
q3) + cos(q3)*sin(q1)) - sin(q5)*sin(q6)*(cos(q1)*sin(q3)
 + cos(q3)*sin(q1))) + b3*sin(q7)*(cos(q5)*cos(q6)*(cos(
q1)*sin(q3) + cos(q3)*sin(q1)) - sin(q5)*sin(q6)*(cos(q1)
*sin(q3) + cos(q3)*sin(q1))), b3*sin(q7)*(cos(q5)*cos(q6)
*(cos(q1)*sin(q3) + cos(q3)*sin(q1)) - sin(q5)*sin(q6)*(
cos(q1)*sin(q3) + cos(q3)*sin(q1))) - z_h*(cos(q8)*(cos(
q5)*sin(q6)*(cos(q1)*sin(q3) + cos(q3)*sin(q1)) + cos(q6)
*sin(q5)*(cos(q1)*sin(q3) + cos(q3)*sin(q1))) + sin(q7)*
sin(q8)*(cos(q5)*cos(q6)*(cos(q1)*sin(q3) + cos(q3)*sin(
q1)) - sin(q5)*sin(q6)*(cos(q1)*sin(q3) + cos(q3)*sin(q1)
))) - l3*cos(q7)*(cos(q5)*cos(q6)*(cos(q1)*sin(q3) + cos(
q3)*sin(q1)) - sin(q5)*sin(q6)*(cos(q1)*sin(q3) + cos(q3)
*sin(q1))) - x_h*(sin(q9)*(sin(q8)*(cos(q5)*sin(q6)*(cos(
q1)*sin(q3) + cos(q3)*sin(q1)) + cos(q6)*sin(q5)*(cos(q1)
*sin(q3) + cos(q3)*sin(q1))) - cos(q8)*sin(q7)*(cos(q5)*
cos(q6)*(cos(q1)*sin(q3) + cos(q3)*sin(q1)) - sin(q5)*sin
(q6)*(cos(q1)*sin(q3) + cos(q3)*sin(q1)))) + cos(q7)*cos(
```

```
    q9)*(cos(q5)*cos(q6)*(cos(q1)*sin(q3) + cos(q3)*sin(q1))
    - sin(q5)*sin(q6)*(cos(q1)*sin(q3) + cos(q3)*sin(q1)))),
    x_h*(cos(q9)*(cos(q7)*(cos(q1)*cos(q3) - sin(q1)*sin(q3))
     + sin(q7)*(cos(q5)*sin(q6)*(cos(q1)*sin(q3) + cos(q3)*
    sin(q1)) + cos(q6)*sin(q5)*(cos(q1)*sin(q3) + cos(q3)*sin
    (q1)))) - cos(q8)*sin(q9)*(sin(q7)*(cos(q1)*cos(q3) - sin
    (q1)*sin(q3)) - cos(q7)*(cos(q5)*sin(q6)*(cos(q1)*sin(q3)
     + cos(q3)*sin(q1)) + cos(q6)*sin(q5)*(cos(q1)*sin(q3) +
    cos(q3)*sin(q1))))) - b3*(sin(q7)*(cos(q1)*cos(q3) - sin(
    q1)*sin(q3)) - cos(q7)*(cos(q5)*sin(q6)*(cos(q1)*sin(q3)
    + cos(q3)*sin(q1)) + cos(q6)*sin(q5)*(cos(q1)*sin(q3) +
    cos(q3)*sin(q1)))) + l3*(cos(q7)*(cos(q1)*cos(q3) - sin(
    q1)*sin(q3)) + sin(q7)*(cos(q5)*sin(q6)*(cos(q1)*sin(q3)
    + cos(q3)*sin(q1)) + cos(q6)*sin(q5)*(cos(q1)*sin(q3) +
    cos(q3)*sin(q1)))) + z_h*sin(q8)*(sin(q7)*(cos(q1)*cos(q3
    ) - sin(q1)*sin(q3)) - cos(q7)*(cos(q5)*sin(q6)*(cos(q1)*
    sin(q3) + cos(q3)*sin(q1)) + cos(q6)*sin(q5)*(cos(q1)*sin
    (q3) + cos(q3)*sin(q1)))), - z_h*(cos(q8)*(cos(q7)*(cos(
    q1)*cos(q3) - sin(q1)*sin(q3)) + sin(q7)*(cos(q5)*sin(q6)
    *(cos(q1)*sin(q3) + cos(q3)*sin(q1)) + cos(q6)*sin(q5)*(
    cos(q1)*sin(q3) + cos(q3)*sin(q1)))) + sin(q8)*(cos(q5)*
    cos(q6)*(cos(q1)*sin(q3) + cos(q3)*sin(q1)) - sin(q5)*sin
    (q6)*(cos(q1)*sin(q3) + cos(q3)*sin(q1)))) - x_h*sin(q9)
    *(sin(q8)*(cos(q7)*(cos(q1)*cos(q3) - sin(q1)*sin(q3)) +
    sin(q7)*(cos(q5)*sin(q6)*(cos(q1)*sin(q3) + cos(q3)*sin(
    q1)) + cos(q6)*sin(q5)*(cos(q1)*sin(q3) + cos(q3)*sin(q1)
    ))) - cos(q8)*(cos(q5)*cos(q6)*(cos(q1)*sin(q3) + cos(q3)
    *sin(q1)) - sin(q5)*sin(q6)*(cos(q1)*sin(q3) + cos(q3)*
    sin(q1)))), -x_h*(sin(q9)*(sin(q7)*(cos(q1)*cos(q3) - sin
    (q1)*sin(q3)) - cos(q7)*(cos(q5)*sin(q6)*(cos(q1)*sin(q3)
     + cos(q3)*sin(q1)) + cos(q6)*sin(q5)*(cos(q1)*sin(q3) +
    cos(q3)*sin(q1)))) - cos(q9)*(cos(q8)*(cos(q7)*(cos(q1)*
    cos(q3) - sin(q1)*sin(q3)) + sin(q7)*(cos(q5)*sin(q6)*(
    cos(q1)*sin(q3) + cos(q3)*sin(q1)) + cos(q6)*sin(q5)*(cos
    (q1)*sin(q3) + cos(q3)*sin(q1)))) + sin(q8)*(cos(q5)*cos(
    q6)*(cos(q1)*sin(q3) + cos(q3)*sin(q1)) - sin(q5)*sin(q6)
    *(cos(q1)*sin(q3) + cos(q3)*sin(q1))))),
6   [0, 0, 0, 1, l3*cos(q7)*(cos(q5)*sin(q6) + cos(q6)*sin(q5))
    - z_off*sin(q5) - z_s*sin(q5) - x_h*(sin(q9)*(sin(q8)*(
    cos(q5)*cos(q6) - sin(q5)*sin(q6)) + cos(q8)*sin(q7)*(cos
    (q5)*sin(q6) + cos(q6)*sin(q5))) - cos(q7)*cos(q9)*(cos(
    q5)*sin(q6) + cos(q6)*sin(q5))) - z_h*(cos(q8)*(cos(q5)*
    cos(q6) - sin(q5)*sin(q6)) - sin(q7)*sin(q8)*(cos(q5)*sin
    (q6) + cos(q6)*sin(q5))) - b3*sin(q7)*(cos(q5)*sin(q6) +
    cos(q6)*sin(q5)), l3*cos(q7)*(cos(q5)*sin(q6) + cos(q6)*
    sin(q5)) - x_h*(sin(q9)*(sin(q8)*(cos(q5)*cos(q6) - sin(
    q5)*sin(q6)) + cos(q8)*sin(q7)*(cos(q5)*sin(q6) + cos(q6)
    *sin(q5))) - cos(q7)*cos(q9)*(cos(q5)*sin(q6) + cos(q6)*
```

```
    sin(q5))) - z_h*(cos(q8)*(cos(q5)*cos(q6) - sin(q5)*sin(
    q6)) - sin(q7)*sin(q8)*(cos(q5)*sin(q6) + cos(q6)*sin(q5)
    )) - b3*sin(q7)*(cos(q5)*sin(q6) + cos(q6)*sin(q5)), x_h
    *(cos(q9)*sin(q7)*(cos(q5)*cos(q6) - sin(q5)*sin(q6)) +
    cos(q7)*cos(q8)*sin(q9)*(cos(q5)*cos(q6) - sin(q5)*sin(q6
    ))) + b3*cos(q7)*(cos(q5)*cos(q6) - sin(q5)*sin(q6)) + l3
    *sin(q7)*(cos(q5)*cos(q6) - sin(q5)*sin(q6)) - z_h*cos(q7
    )*sin(q8)*(cos(q5)*cos(q6) - sin(q5)*sin(q6)), z_h*(sin(
    q8)*(cos(q5)*sin(q6) + cos(q6)*sin(q5)) - cos(q8)*sin(q7)
    *(cos(q5)*cos(q6) - sin(q5)*sin(q6))) - x_h*sin(q9)*(cos(
    q8)*(cos(q5)*sin(q6) + cos(q6)*sin(q5)) + sin(q7)*sin(q8)
    *(cos(q5)*cos(q6) - sin(q5)*sin(q6))), -x_h*(cos(q9)*(sin
    (q8)*(cos(q5)*sin(q6) + cos(q6)*sin(q5)) - cos(q8)*sin(q7
    )*(cos(q5)*cos(q6) - sin(q5)*sin(q6))) - cos(q7)*sin(q9)
    *(cos(q5)*cos(q6) - sin(q5)*sin(q6)))],
[0, 0, 0, 0, - cos(q1)*sin(q3) - cos(q3)*sin(q1), - cos(q1)*
    sin(q3) - cos(q3)*sin(q1), cos(q5)*cos(q6)*(cos(q1)*cos(
    q3) - sin(q1)*sin(q3)) - sin(q5)*sin(q6)*(cos(q1)*cos(q3)
     - sin(q1)*sin(q3)), - sin(q7)*(cos(q1)*sin(q3) + cos(q3)
    *sin(q1)) - cos(q7)*(cos(q5)*sin(q6)*(cos(q1)*cos(q3) -
    sin(q1)*sin(q3)) + cos(q6)*sin(q5)*(cos(q1)*cos(q3) - sin
    (q1)*sin(q3))), sin(q8)*(cos(q7)*(cos(q1)*sin(q3) + cos(
    q3)*sin(q1)) - sin(q7)*(cos(q5)*sin(q6)*(cos(q1)*cos(q3)
    - sin(q1)*sin(q3)) + cos(q6)*sin(q5)*(cos(q1)*cos(q3) -
    sin(q1)*sin(q3)))) + cos(q8)*(cos(q5)*cos(q6)*(cos(q1)*
    cos(q3) - sin(q1)*sin(q3)) - sin(q5)*sin(q6)*(cos(q1)*cos
    (q3) - sin(q1)*sin(q3)))],
[0, 0, 0, 0, cos(q1)*cos(q3) - sin(q1)*sin(q3), cos(q1)*cos(
    q3) - sin(q1)*sin(q3), cos(q5)*cos(q6)*(cos(q1)*sin(q3) +
     cos(q3)*sin(q1)) - sin(q5)*sin(q6)*(cos(q1)*sin(q3) +
    cos(q3)*sin(q1)),sin(q7)*(cos(q1)*cos(q3) - sin(q1)*sin(
    q3)) - cos(q7)*(cos(q5)*sin(q6)*(cos(q1)*sin(q3) + cos(q3
    )*sin(q1)) + cos(q6)*sin(q5)*(cos(q1)*sin(q3) + cos(q3)*
    sin(q1))), cos(q8)*(cos(q5)*cos(q6)*(cos(q1)*sin(q3) +
    cos(q3)*sin(q1)) - sin(q5)*sin(q6)*(cos(q1)*sin(q3) + cos
    (q3)*sin(q1))) - sin(q8)*(cos(q7)*(cos(q1)*cos(q3) - sin(
    q1)*sin(q3)) + sin(q7)*(cos(q5)*sin(q6)*(cos(q1)*sin(q3)
    + cos(q3)*sin(q1)) + cos(q6)*sin(q5)*(cos(q1)*sin(q3) +
    cos(q3)*sin(q1))))],
[1, 0, 1, 0, 0, 0, - cos(q5)*sin(q6) - cos(q6)*sin(q5), -cos
    (q7)*(cos(q5)*cos(q6) - sin(q5)*sin(q6)), - cos(q8)*(cos(
    q5)*sin(q6) + cos(q6)*sin(q5)) - sin(q7)*sin(q8)*(cos(q5)
    *cos(q6) - sin(q5)*sin(q6))]]
```

The forward kinematic transformation

```
FK =

```

```
[[ -(cos(q9)*sin(q7) + cos(q7)*cos(q8)*sin(q9))*(cos(q1)*cos(q3) - sin(q1)*sin(q3)) - cos(q5)*(cos(q1)*sin(q3) + cos(q3)*sin(q1))*(sin(q9)*(sin(q6)*sin(q8) + cos(q6)*cos(q8)*sin(q7)) - cos(q6)*cos(q7)*cos(q9)) - sin(q5)*(sin(q9)*(cos(q6)*sin(q8) - cos(q8)*sin(q6)*sin(q7)) + cos(q7)*cos(q9)*sin(q6))*(cos(q1)*sin(q3) + cos(q3)*sin(q1)), (sin(q7)*sin(q9) - cos(q7)*cos(q8)*cos(q9))*(cos(q1)*cos(q3) - sin(q1)*sin(q3)) - cos(q5)*(cos(q1)*sin(q3) + cos(q3)*sin(q1))*(cos(q9)*(sin(q6)*sin(q8) + cos(q6)*cos(q8)*sin(q7)) + cos(q6)*cos(q7)*sin(q9)) - sin(q5)*(cos(q9)*(cos(q6)*sin(q8) - cos(q8)*sin(q6)*sin(q7)) - cos(q7)*sin(q6)*sin(q9))*(cos(q1)*sin(q3) + cos(q3)*sin(q1)), cos(q7)*sin(q8)*(cos(q1)*cos(q3) - sin(q1)*sin(q3)) - sin(q5)*(cos(q6)*cos(q8) + sin(q6)*sin(q7)*sin(q8))*(cos(q1)*sin(q3) + cos(q3)*sin(q1)) - cos(q5)*(cos(q8)*sin(q6) - cos(q6)*sin(q7)*sin(q8))*(cos(q1)*sin(q3) + cos(q3)*sin(q1)), d2*sin(q1) - z_h*(cos(q5)*(cos(q1)*sin(q3) + cos(q3)*sin(q1))*(cos(q9)*(sin(q6)*sin(q8) + cos(q6)*cos(q8)*sin(q7)) + cos(q6)*cos(q7)*sin(q9)) - (sin(q7)*sin(q9) - cos(q7)*cos(q8)*cos(q9))*(cos(q1)*cos(q3) - sin(q1)*sin(q3)) + sin(q5)*(cos(q9)*(cos(q6)*sin(q8) - cos(q8)*sin(q6)*sin(q7)) - cos(q7)*sin(q6)*sin(q9))*(cos(q1)*sin(q3) + cos(q3)*sin(q1))) - (cos(q1)*cos(q3) - sin(q1)*sin(q3))*(b3*cos(q7) - y_s + l3*sin(q7)) - x_h*((cos(q9)*sin(q7) + cos(q7)*cos(q8)*sin(q9))*(cos(q1)*cos(q3) - sin(q1)*sin(q3)) + cos(q5)*(cos(q1)*sin(q3) + cos(q3)*sin(q1))*(sin(q9)*(sin(q6)*sin(q8) + cos(q6)*cos(q8)*sin(q7)) - cos(q6)*cos(q7)*cos(q9)) + sin(q5)*(sin(q9)*(cos(q6)*sin(q8) - cos(q8)*sin(q6)*sin(q7)) + cos(q7)*cos(q9)*sin(q6))*(cos(q1)*sin(q3) + cos(q3)*sin(q1))) - sin(q5)*(l3*cos(q7)*sin(q6) - b3*sin(q6)*sin(q7))*(cos(q1)*sin(q3) + cos(q3)*sin(q1)) - cos(q5)*(cos(q1)*sin(q3) + cos(q3)*sin(q1))*(z_s - l3*cos(q6)*cos(q7) + b3*cos(q6)*sin(q7))],
[cos(q5)*(cos(q1)*cos(q3) - sin(q1)*sin(q3))*(sin(q9)*(sin(q6)*sin(q8) + cos(q6)*cos(q8)*sin(q7)) - cos(q6)*cos(q7)*cos(q9)) - (cos(q9)*sin(q7) + cos(q7)*cos(q8)*sin(q9))*(cos(q1)*sin(q3) + cos(q3)*sin(q1)) + sin(q5)*(sin(q9)*(cos(q6)*sin(q8) - cos(q8)*sin(q6)*sin(q7)) + cos(q7)*cos(q9)*sin(q6))*(cos(q1)*cos(q3) - sin(q1)*sin(q3)), (sin(q7)*sin(q9) - cos(q7)*cos(q8)*cos(q9))*(cos(q1)*sin(q3) + cos(q3)*sin(q1)) + cos(q5)*(cos(q1)*cos(q3) - sin(q1)*sin(q3))*(cos(q9)*(sin(q6)*sin(q8) + cos(q6)*cos(q8)*sin(q7)) + cos(q6)*cos(q7)*sin(q9)) + sin(q5)*(cos(q9)*(cos(q6)*sin(q8) - cos(q8)*sin(q6)*sin(q7)) - cos(q7)*sin(q6)*sin(q9))*(cos(q1)*cos(q3) - sin(q1)*sin(q3)), cos(q5)*(cos(q8)*sin(q6) - cos(q6)*sin(q7)*sin(q8))*(cos(q1)*cos(q3) - sin(q1)*sin(q3)) + sin(q5)*(cos(q6)*cos(q8) + sin(q6)*sin(q7)*sin(q8))*(cos(q1)*cos(q3) - sin(q1)*sin(q3)) + cos(
```

```
    q7)*sin(q8)*(cos(q1)*sin(q3) + cos(q3)*sin(q1)), x_h*(cos
    (q5)*(cos(q1)*cos(q3) - sin(q1)*sin(q3))*(sin(q9)*(sin(q6
    )*sin(q8) + cos(q6)*cos(q8)*sin(q7)) - cos(q6)*cos(q7)*
    cos(q9)) - (cos(q9)*sin(q7) + cos(q7)*cos(q8)*sin(q9))*(
    cos(q1)*sin(q3) + cos(q3)*sin(q1)) + sin(q5)*(sin(q9)*(
    cos(q6)*sin(q8) - cos(q8)*sin(q6)*sin(q7)) + cos(q7)*cos(
    q9)*sin(q6))*(cos(q1)*cos(q3) - sin(q1)*sin(q3))) + z_h
    *((sin(q7)*sin(q9) - cos(q7)*cos(q8)*cos(q9))*(cos(q1)*
    sin(q3) + cos(q3)*sin(q1)) + cos(q5)*(cos(q1)*cos(q3) -
    sin(q1)*sin(q3))*(cos(q9)*(sin(q6)*sin(q8) + cos(q6)*cos(
    q8)*sin(q7)) + cos(q6)*cos(q7)*sin(q9)) + sin(q5)*(cos(q9
    )*(cos(q6)*sin(q8) - cos(q8)*sin(q6)*sin(q7)) - cos(q7)*
    sin(q6)*sin(q9))*(cos(q1)*cos(q3) - sin(q1)*sin(q3))) - (
    cos(q1)*sin(q3) + cos(q3)*sin(q1))*(b3*cos(q7) - y_s + l3
    *sin(q7)) - d2*cos(q1) + sin(q5)*(l3*cos(q7)*sin(q6) - b3
    *sin(q6)*sin(q7))*(cos(q1)*cos(q3) - sin(q1)*sin(q3)) +
    cos(q5)*(cos(q1)*cos(q3) - sin(q1)*sin(q3))*(z_s - l3*cos
    (q6)*cos(q7) + b3*cos(q6)*sin(q7))],
[sin(q5)*(sin(q9)*(sin(q6)*sin(q8) + cos(q6)*cos(q8)*sin(q7)
    ) - cos(q6)*cos(q7)*cos(q9)) - cos(q5)*(sin(q9)*(cos(q6)*
    sin(q8) - cos(q8)*sin(q6)*sin(q7)) + cos(q7)*cos(q9)*sin(
    q6)), sin(q5)*(cos(q9)*(sin(q6)*sin(q8) + cos(q6)*cos(q8)
    *sin(q7)) + cos(q6)*cos(q7)*sin(q9)) - cos(q5)*(cos(q9)*(
    cos(q6)*sin(q8) - cos(q8)*sin(q6)*sin(q7)) - cos(q7)*sin(
    q6)*sin(q9)), sin(q5)*(cos(q8)*sin(q6) - cos(q6)*sin(q7)*
    sin(q8)) - cos(q5)*(cos(q6)*cos(q8) + sin(q6)*sin(q7)*sin
    (q8)), d4 + sin(q5)*(z_s - l3*cos(q6)*cos(q7) + b3*cos(q6
    )*sin(q7)) - cos(q5)*(l3*cos(q7)*sin(q6) - b3*sin(q6)*sin
    (q7)) - x_h*(cos(q5)*(sin(q9)*(cos(q6)*sin(q8) - cos(q8)*
    sin(q6)*sin(q7)) + cos(q7)*cos(q9)*sin(q6)) - sin(q5)*(
    sin(q9)*(sin(q6)*sin(q8) + cos(q6)*cos(q8)*sin(q7)) - cos
    (q6)*cos(q7)*cos(q9))) + z_h*(sin(q5)*(cos(q9)*(sin(q6)*
    sin(q8) + cos(q6)*cos(q8)*sin(q7)) + cos(q6)*cos(q7)*sin(
    q9)) - cos(q5)*(cos(q9)*(cos(q6)*sin(q8) - cos(q8)*sin(q6
    )*sin(q7)) - cos(q7)*sin(q6)*sin(q9)))],
[0, 0, 0, 1]]
```

Investigate average performance time for SVD

```
% Get the average time for performing a SVD
% March, 2016, by AAsmund Pedersen Hugo

function avgTime = avgSVDanalysis()

n = 10000; % Number of tests
J = rand(6,9);

tic
```

```
for i = 1:n
    svd(J);
end
endTime = toc;

avgTime = endTime/n;
end
```

Obtaining the Desired Step-Size

```
% Obtaining solver step-size for Nao
% By Aasmund P. Hugo, April 2016

function stepsize = getStepsize()

    close all;
    clear all;

    in = pi/6;                %Init angle
    increment = 0.0005;
    linear = 7*cos(in);       %Init linearized
    analytical = 7*cos(in); %Init analytical
    errors = 0;
    error = 0;
    stepsize = 0.0005;        %Init step-size
    max_dev = 0.0017;         %Sensor inaccuracy
    while error<max_dev
        %Temp linearized value
        temp = 7*cos(in)-7*sin(in)*stepsize;
        linear = [linear temp];
        %Temp analytical value
        temp_a = 7*cos(in+stepsize);
        analytical = [analytical temp_a];
        error = abs(temp_a-temp);
        errors = [errors error];
        stepsize = stepsize + increment;
    end

    xvalues = linspace(in,in+stepsize,(stepsize/increment));
    iterations = linspace(0,stepsize,stepsize/increment);
    figure(1)
    subplot(1,2,1)
    plot(xvalues,linear,'b');
    hold on
    plot(xvalues,analytical,'g');
    legend('Linearized','Analytical')
    ylabel('7cos(x)');
    xlabel('Radians');
```

```
39      subplot(1,2,2)
40      plot(iterations,errors,'k')
41      hold on
42      plot(iterations,linspace(max_dev,max_dev,length(xvalues)
            ),'r');
43      xlabel('Radians');
44      ylabel('Error');
45      legend('Deviation from analytical solution','Sensory
            deviation')
46
47  end
```

Appendix B

This chapter contain the key programs to use the IK solver. The included commentary provides adequate information about each program's objectivity and key features.

The main solver function

```
# -*- coding: utf-8 -*-
"""
Created on Wed Apr 13 18:15:19 2016
@author: Aasmund P. Hugo

Fuzzy Logical, Dynamic Weighted, Damped Least Squares
"""

import numpy as np
from jointConstraints import checkConstraints
from dof import getDofList
from fuzzify import fuzzify
import forwardKinematics as fk
import time as t
import IKplots as ikp
from IK import cartesianIK, eulerIK, orientationIK

"Error boundaries"
eps_prism = 0.1      # Max deviation, position [mm]
eps_ang = 0.0017     # Max deviation,orientation [rad]
q1 = np.array([0.5, 300.0, 0.0, 70.0, 0.2, -0.5, 0.2, 0.2,
    -0.4])

def solveIK(q0, xd, dof, isTracking):

    "Init values"
    q_hat = np.array(q0)            #Updated joint vector
    x_hat = fk.FK(q0)               #Updated Cartesian vector
    print(x_hat)
    e = (xd - x_hat)                #Initial error

```

```
31      q_hats = np.array(q_hat)          #List of joint values
32      errors = np.array(e)              #List of deviation from
           desired
33      x_hats = np.array(x_hat)           #List of Cartesian
           values
34
35      "Fuzzy constant to ensure orientation error stays at
           zero"
36      ifRotationFinished = 1
37
38      "Establish the degrees of freedom desired to control"
39      dof_1, dof_2, dof_3 = getDofList(dof)
40
41      start_time = t.time()
42
43      while np.max(np.abs(e[3:6])) > eps_ang or np.max(np.abs(
           e[0:3])) > eps_prism:
44
45          fuzzyflag = fuzzify(e, q_hat, eps_ang,
               ifRotationFinished)
46          if fuzzyflag == 0 or fuzzyflag == 1:
47              dq = cartesianIK(q_hat, x_hat, e, dof_1)
48              ifRotationFinished = 0
49
50          elif fuzzyflag >= 0.125 and fuzzyflag <= 0.5:
51              dq = eulerIK(q_hat, x_hat, e, dof_2)
52
53          elif fuzzyflag >= 0.0125 and fuzzyflag <= 0.05:
54              dq = orientationIK(q_hat, x_hat, e, dof_3)
55
56          else:
57              return 'xd is not a viable configuration'
58
59          # Checking for joint constraints
60          if ifRotationFinished:
61              dq = checkConstraints(q_hat, dq)
62
63          # Updating calculated real time joint og Cartesian
               values
64          q_hat = q_hat + dq
65          x_hat = fk.FK(q_hat, x_hat[3:6])
66          e = xd - x_hat
67          print(e)
68          #t.sleep(0.02)
69
70          if isTracking:
71              q_hats = np.c_[q_hats, q_hat]
72              x_hats = np.c_[x_hats, x_hat]
```

```
73              errors = np.c_[errors, e]
74
75      end_time = t.time()-start_time
76      q_hat_print = [float(i) for i in q_hat]
77      x_hat = [float(i) for i in x_hat]
78
79      print('Time to solve: ', end_time)
80      print('Final joint values: ', q_hat_print)
81      print('Final Cartesian position: ', x_hat)
82      if isTracking:
83          ikp.vectorPlot(x_hats)
84          ikp.errorTraj(errors)
85          ikp.jointTraj(q_hats)
86          ikp.cartesianTraj(x_hats)
87
88      return q_hat
```

Forward kinematics function

```
1  # -*- coding: utf-8 -*-
2  """
3  Created on Thu Feb 18 21:06:34 2016
4  @author: Aasmund Hugo
5
6  Forward Kinematics of the NAO system
7  """
8
9  import numpy as np
10 import matMath as mm
11
12
13 def FK(q, prev_euler=None):
14
15     q1 = q[0]
16     d2 = q[1]
17     q3 = q[2]
18     d4 = q[3]
19     q5 = q[4]
20     q6 = q[5]
21     q7 = q[6]
22     q8 = q[7]
23     q9 = q[8]
24
25     h = 143.5
26     z_off = 85 # Offset from hip joint to torso
27     y_off = 0
28     x_off = 0
29
```

```
30      d4 = d4+h
31      q1 = q1+(np.pi/2)
32
33      T01 = np.array([[np.cos(q1), -np.sin(q1), 0, 0],[np.sin(
            q1), np.cos(q1), 0, 0],[0, 0, 1, 0],[0, 0, 0, 1]])
34      T12 = np.array([[1, 0, 0, 0],[0, 0, -1, -d2],[0, 1, 0,
            0],[0, 0, 0, 1]])
35      T23 = np.array([[np.cos(q3), -np.sin(q3), 0, 0],[0, 0,
            1, 0],[-np.sin(q3), -np.cos(q3), 0, 0],[0, 0, 0, 1]])
36      T34 = np.array([[0, -1, 0, 0],[1, 0, 0, 0],[0, 0, 1, d4
            ],[0, 0, 0, 1]])
37      T45 = np.array([[np.cos(q5), -np.sin(q5), 0, 0],[0, 0,
            -1, 0],[np.sin(q5), np.cos(q5), 0, 0],[0, 0, 0, 1]])
38      R5 = np.array([[-1, 0, 0, 0],[0, 0, 1, 0],[0, 1, 0,
            0],[0, 0, 0, 1]])
39      A5 = np.array([[1, 0, 0, x_off],[0, 1, 0, y_off],[0, 0,
            1, z_off],[0, 0, 0, 1]])
40      TBody = np.matmul(np.matmul(np.matmul(np.matmul(np.
            matmul(np.matmul(T01,T12),T23),T34),T45),R5),A5)
41
42      l3 = 105     # Length of upper arm
43      b3 = 15      # Offset in upper arm
44      z_s = 100    # Shoulder offset in z
45      y_s = 98     # Shoulder offset in y
46      x_h = 55.95+57.75 # Length of lower arm
47      z_h = -12.31 # Offset of hand
48
49      q7=q7+(np.pi/2) # Shoulder roll is allways rotated +90
            deg
50
51      Ab = np.array([[1, 0, 0, 0],[0, 1, 0, y_s],[0, 0, 1, z_s
            ],[0, 0, 0, 1]])
52      T56 = np.array([[np.cos(q6), -np.sin(q6), 0, 0],[0, 0,
            1, 0],[-np.sin(q6), -np.cos(q6), 0, 0],[0, 0, 0, 1]])
53      T67 = np.array([[np.cos(q7), -np.sin(q7), 0, 0],[0, 0,
            -1, 0],[np.sin(q7), np.cos(q7), 0, 0],[0, 0, 0, 1]])
54      T78 = np.array([[np.cos(q8), -np.sin(q8), 0, b3],[0, 0,
            -1, -l3],[np.sin(q8), np.cos(q8), 0, 0],[0, 0, 0,
            1]])
55      T89 = np.array([[np.cos(q9), -np.sin(q9), 0, 0],[0, 0,
            1, 0],[-np.sin(q9), -np.cos(q9), 0, 0],[0, 0, 0, 1]])
56      R9 = np.array([[0, 1, 0, 0],[-1, 0, 0, 0],[0, 0, 1,
            0],[0, 0, 0, 1]])
57      A9 = np.array([[1, 0, 0, x_h],[0, 1, 0, 0],[0, 0, 1, z_h
            ],[0, 0, 0, 1]])
58      #R10 = np.array
            ([[0,0,-1,0],[0,1,0,0],[1,0,0,0],[0,0,0,1]])
59
```

```
60      TLarm = np.matmul(np.matmul(np.matmul(np.matmul(np.
            matmul(np.matmul(Ab,T56),T67),T78),T89),R9),A9)
61      Tsystem = np.matmul(TBody,TLarm)
62      [phi, theta, psi] = mm.mat2euler(Tsystem[0:3, 0:3])
63      return np.array([Tsystem[0][3], Tsystem[1][3], Tsystem
            [2][3], phi, theta, psi])
```

Rotation matrix to Euler angles

```
1  # -*- coding: utf-8 -*-
2  """
3  Created on Thu Feb 18 21:17:18 2016
4  @author: Aasmund Hugo
5
6  Different matrix operations
7  """
8
9  import math as m
10 import numpy as np
11
12 def mat2euler(M, prev_euler=None):
13
14     """
15     Using the most frequent sequence of rotations, in
           engineering:
16
17         RotatinMatrix M = RzRyRx = R(psi)R(theta)R(phi)
18
19     """
20     if prev_euler==None:
21         prev_euler = [0,0,0]
22
23     prev_theta = prev_euler[1]
24     prev_psi = prev_euler[2]
25
26     if (np.abs(M[2,0]) != 1):
27         theta = -m.asin(M[2,0])
28         if (np.abs((theta-prev_theta))>np.abs(m.pi-theta-
               prev_theta)):
29             theta = m.pi-theta
30         phi = m.atan2((M[2,1]/m.cos(theta)),(M[2,2]/m.cos(
               theta)))
31         psi = m.atan2((M[1,0]/m.cos(theta)),(M[0,0]/m.cos(
               theta)))
32     elif (M[2,0] == -1):
33         theta = m.pi/2
34         phi = prev_psi + m.atan2(M[0,1],M[0,2])
35     else:
```

```
36          theta = -m.pi/2
37          phi = -prev_psi + m.atan2(-M[0,1], -M[0,2])
38      prev_euler = np.array([phi, theta, psi])
39      return np.array([phi, theta, psi])
```

Inverse kinematic control systems

```
1  # -*- coding: utf-8 -*-
2  """
3  Created on Thu Apr 14 14:55:54 2016
4  @author: Aasmund P. Hugo
5
6  Inverse Kinematic of the different (sub)systems
7  """
8
9  from determineDamping import getLambda
10 from dynamicWeighting import getDynamicWeighting
11 import numpy as np
12 import math as m
13 from jacobian import euler_jacobian, orientation_jacobian,
       cartesian_jacobian, locomotion_jacobian
14
15 step_size = 0.0245
16
17 def eulerIK(q_hat, x_hat, e, dof):
18     e_local = np.array([e[int(i)] for i in dof])
19     #print('format problem e: ', e_local)
20     J = euler_jacobian(q_hat)
21     Lambda = getLambda(J)
22     J_T = np.transpose(J)
23     damped_square = np.matmul(J,J_T)+Lambda
24     damped_square_inv = np.linalg.inv(damped_square)
25     dq = np.matmul((np.matmul(J_T,damped_square_inv)),
           step_size*e_local)
26     #print('arm dq is',dq)
27     #dq_max = np.max(np.abs(dq))
28     #if (dq_max>step_size or dq_max<0.005):
29     #    alpha = getDynamicWeighting(e, dq, q_hat,
           damped_square_inv, x_hat, dof)
30     #    dq = np.matmul(alpha, dq)
31     return dq
32
33 def cartesianIK(q_hat, x_hat, e, dof):
34     e_local = np.array([e[int(i)] for i in dof])
35     J = cartesian_jacobian(q_hat)
36     #print J
37     Lambda = getLambda(J)
38     J_T = np.transpose(J)
```

```
39      damped_square = np.matmul(J,J_T)+Lambda
40      #print('the dampd J: ', damped_square)
41      try:
42          damped_square_inv = np.linalg.inv(damped_square)
43          dq = np.matmul((np.matmul(J_T,damped_square_inv)),
                step_size*e_local)
44      except:
45          dq = np.zeros(9)
46          dq1 = m.atan2(e_local[1],e_local[0])
47          dq2 = m.sqrt(e_local[0]**2+e_local[1]**2)
48          if dq1 > np.pi/2:
49              dq1 = -np.pi + dq1
50              dq2 = -dq2
51          dq[0] = dq1*step_size
52          dq[1] = dq2*step_size
53          dq[2] = -dq1*step_size
54          dq[3] = e_local[3]*step_size
55      return dq
56
57  def locomotionIK(q_hat, x_hat, e, dof):
58      e_local = np.array([e[int(i)] for i in dof])
59      J = locomotion_jacobian(q_hat)
60      Lambda = getLambda(J)
61      J_T = np.transpose(J)
62      damped_square = np.matmul(J,J_T)+Lambda
63      #print('the dampd J: ', damped_square)
64      damped_square_inv = np.linalg.inv(damped_square)
65      dq = np.matmul((np.matmul(J_T,damped_square_inv)),
            step_size*e_local)
66      #print('loco dq is',dq)
67      return dq
68
69  def orientationIK(q_hat, x_hat, e, dof):
70      e_local = np.array([e[int(i)] for i in dof])
71      #print('format problem e: ', e_local)
72      J = orientation_jacobian(q_hat)
73      Lambda = getLambda(J)
74      J_T = np.transpose(J)
75      damped_square = np.matmul(J,J_T)+Lambda
76      damped_square_inv = np.linalg.inv(damped_square)
77      dq = np.matmul((np.matmul(J_T,damped_square_inv)),
            step_size*e_local)
78      #print('orient dq is',dq)
79      dq_max = np.max(np.abs(dq))
80      if (dq_max>step_size or dq_max<0.005):
81          alpha = getDynamicWeighting(e, dq, q_hat,
                damped_square_inv, x_hat, dof)
82          dq = np.matmul(alpha, dq)
```

```
83      return dq
```

Jacobians

```
1  # -*- coding: utf-8 -*-
2  """
3  Created on Thu Feb 18 19:45:46 2016
4  @author: Aasmund Hugo
5
6  Finding the Jacobian of NAO
7  """
8
9  from numpy import *
10
11 def jacobian(q):
12      q1 = q[0]
13      d2 = q[1]
14      q3 = q[2]
15      d4 = q[3]
16      q5 = q[4]
17      q6 = q[5]
18      q7 = q[6]
19      q8 = q[7]
20      q9 = q[8]
21
22      "Constants"
23      h = 145
24      d4 = d4+h
25      z_off = 85     # Offset from hip joint to torso
26      l3 = 105
27      b3 = 15
28      z_s = 100
29      y_s = 98
30      x_h = 57.75+55.95
31      z_h = -12.31
32
33      "Jacobian of the NAO system"
34
35      J = array([[ z_h*(sin(q8)*(cos(q7)*(cos(q1)*cos(q3) -
            sin(q1)*sin(q3)) + sin(q7)*(cos(q5)*sin(q6)*(cos(q1)*
            sin(q3) + cos(q3)*sin(q1)) + cos(q6)*sin(q5)*(cos(q1)
            *sin(q3) + cos(q3)*sin(q1)))) - cos(q8)*(cos(q5)*cos(
            q6)*(cos(q1)*sin(q3) + cos(q3)*sin(q1)) - sin(q5)*sin
            (q6)*(cos(q1)*sin(q3) + cos(q3)*sin(q1)))) - y_s*(cos
            (q1)*cos(q3) - sin(q1)*sin(q3)) - x_h*(cos(q9)*(sin(
            q7)*(cos(q1)*cos(q3) - sin(q1)*sin(q3)) - cos(q7)*(
            cos(q5)*sin(q6)*(cos(q1)*sin(q3) + cos(q3)*sin(q1)) +
             cos(q6)*sin(q5)*(cos(q1)*sin(q3) + cos(q3)*sin(q1)))
```

```
) + sin(q9)*(cos(q8)*(cos(q7)*(cos(q1)*cos(q3) - sin(
q1)*sin(q3)) + sin(q7)*(cos(q5)*sin(q6)*(cos(q1)*sin(
q3) + cos(q3)*sin(q1)) + cos(q6)*sin(q5)*(cos(q1)*sin
(q3) + cos(q3)*sin(q1)))) + sin(q8)*(cos(q5)*cos(q6)
*(cos(q1)*sin(q3) + cos(q3)*sin(q1)) - sin(q5)*sin(q6
)*(cos(q1)*sin(q3) + cos(q3)*sin(q1))))) - d2*sin(q1)
 - b3*(cos(q7)*(cos(q1)*cos(q3) - sin(q1)*sin(q3)) +
sin(q7)*(cos(q5)*sin(q6)*(cos(q1)*sin(q3) + cos(q3)*
sin(q1)) + cos(q6)*sin(q5)*(cos(q1)*sin(q3) + cos(q3)
*sin(q1)))) - l3*(sin(q7)*(cos(q1)*cos(q3) - sin(q1)*
sin(q3)) - cos(q7)*(cos(q5)*sin(q6)*(cos(q1)*sin(q3)
+ cos(q3)*sin(q1)) + cos(q6)*sin(q5)*(cos(q1)*sin(q3)
 + cos(q3)*sin(q1)))) - z_off*sin(q5)*(cos(q1)*sin(q3
) + cos(q3)*sin(q1)) - z_s*sin(q5)*(cos(q1)*sin(q3) +
 cos(q3)*sin(q1)), cos(q1), z_h*(sin(q8)*(cos(q7)*(
cos(q1)*cos(q3) - sin(q1)*sin(q3)) + sin(q7)*(cos(q5)
*sin(q6)*(cos(q1)*sin(q3) + cos(q3)*sin(q1)) + cos(q6
)*sin(q5)*(cos(q1)*sin(q3) + cos(q3)*sin(q1)))) - cos
(q8)*(cos(q5)*cos(q6)*(cos(q1)*sin(q3) + cos(q3)*sin(
q1)) - sin(q5)*sin(q6)*(cos(q1)*sin(q3) + cos(q3)*sin
(q1)))) - y_s*(cos(q1)*cos(q3) - sin(q1)*sin(q3)) -
x_h*(cos(q9)*(sin(q7)*(cos(q1)*cos(q3) - sin(q1)*sin(
q3)) - cos(q7)*(cos(q5)*sin(q6)*(cos(q1)*sin(q3) +
cos(q3)*sin(q1)) + cos(q6)*sin(q5)*(cos(q1)*sin(q3) +
 cos(q3)*sin(q1)))) + sin(q9)*(cos(q8)*(cos(q7)*(cos(
q1)*cos(q3) - sin(q1)*sin(q3)) + sin(q7)*(cos(q5)*sin
(q6)*(cos(q1)*sin(q3) + cos(q3)*sin(q1)) + cos(q6)*
sin(q5)*(cos(q1)*sin(q3) + cos(q3)*sin(q1)))) + sin(
q8)*(cos(q5)*cos(q6)*(cos(q1)*sin(q3) + cos(q3)*sin(
q1)) - sin(q5)*sin(q6)*(cos(q1)*sin(q3) + cos(q3)*sin
(q1))))) - b3*(cos(q7)*(cos(q1)*cos(q3) - sin(q1)*sin
(q3)) + sin(q7)*(cos(q5)*sin(q6)*(cos(q1)*sin(q3) +
cos(q3)*sin(q1)) + cos(q6)*sin(q5)*(cos(q1)*sin(q3) +
 cos(q3)*sin(q1)))) - l3*(sin(q7)*(cos(q1)*cos(q3) -
sin(q1)*sin(q3)) - cos(q7)*(cos(q5)*sin(q6)*(cos(q1)*
sin(q3) + cos(q3)*sin(q1)) + cos(q6)*sin(q5)*(cos(q1)
*sin(q3) + cos(q3)*sin(q1)))) - z_off*sin(q5)*(cos(q1
)*sin(q3) + cos(q3)*sin(q1)) - z_s*sin(q5)*(cos(q1)*
sin(q3) + cos(q3)*sin(q1)), 0, z_off*cos(q5)*(cos(q1)
*cos(q3) - sin(q1)*sin(q3)) - z_h*(cos(q8)*(cos(q5)*
sin(q6)*(cos(q1)*cos(q3) - sin(q1)*sin(q3)) + cos(q6)
*sin(q5)*(cos(q1)*cos(q3) - sin(q1)*sin(q3))) + sin(
q7)*sin(q8)*(cos(q5)*cos(q6)*(cos(q1)*cos(q3) - sin(
q1)*sin(q3)) - sin(q5)*sin(q6)*(cos(q1)*cos(q3) - sin
(q1)*sin(q3)))) - x_h*(sin(q9)*(sin(q8)*(cos(q5)*sin(
q6)*(cos(q1)*cos(q3) - sin(q1)*sin(q3)) + cos(q6)*sin
(q5)*(cos(q1)*cos(q3) - sin(q1)*sin(q3))) - cos(q8)*
sin(q7)*(cos(q5)*cos(q6)*(cos(q1)*cos(q3) - sin(q1)*
```

```
sin(q3)) - sin(q5)*sin(q6)*(cos(q1)*cos(q3) - sin(q1)
*sin(q3)))) + cos(q7)*cos(q9)*(cos(q5)*cos(q6)*(cos(
q1)*cos(q3) - sin(q1)*sin(q3)) - sin(q5)*sin(q6)*(cos
(q1)*cos(q3) - sin(q1)*sin(q3)))) + z_s*cos(q5)*(cos(
q1)*cos(q3) - sin(q1)*sin(q3)) - l3*cos(q7)*(cos(q5)*
cos(q6)*(cos(q1)*cos(q3) - sin(q1)*sin(q3)) - sin(q5)
*sin(q6)*(cos(q1)*cos(q3) - sin(q1)*sin(q3))) + b3*
sin(q7)*(cos(q5)*cos(q6)*(cos(q1)*cos(q3) - sin(q1)*
sin(q3)) - sin(q5)*sin(q6)*(cos(q1)*cos(q3) - sin(q1)
*sin(q3))), b3*sin(q7)*(cos(q5)*cos(q6)*(cos(q1)*cos(
q3) - sin(q1)*sin(q3)) - sin(q5)*sin(q6)*(cos(q1)*cos
(q3) - sin(q1)*sin(q3))) - z_h*(cos(q8)*(cos(q5)*sin(
q6)*(cos(q1)*cos(q3) - sin(q1)*sin(q3)) + cos(q6)*sin
(q5)*(cos(q1)*cos(q3) - sin(q1)*sin(q3))) + sin(q7)*
sin(q8)*(cos(q5)*cos(q6)*(cos(q1)*cos(q3) - sin(q1)*
sin(q3)) - sin(q5)*sin(q6)*(cos(q1)*cos(q3) - sin(q1)
*sin(q3)))) - l3*cos(q7)*(cos(q5)*cos(q6)*(cos(q1)*
cos(q3) - sin(q1)*sin(q3)) - sin(q5)*sin(q6)*(cos(q1)
*cos(q3) - sin(q1)*sin(q3))) - x_h*(sin(q9)*(sin(q8)
*(cos(q5)*sin(q6)*(cos(q1)*cos(q3) - sin(q1)*sin(q3))
 + cos(q6)*sin(q5)*(cos(q1)*cos(q3) - sin(q1)*sin(q3)
)) - cos(q8)*sin(q7)*(cos(q5)*cos(q6)*(cos(q1)*cos(q3
) - sin(q1)*sin(q3)) - sin(q5)*sin(q6)*(cos(q1)*cos(
q3) - sin(q1)*sin(q3)))) + cos(q7)*cos(q9)*(cos(q5)*
cos(q6)*(cos(q1)*cos(q3) - sin(q1)*sin(q3)) - sin(q5)
*sin(q6)*(cos(q1)*cos(q3) - sin(q1)*sin(q3)))), b3*(
sin(q7)*(cos(q1)*sin(q3) + cos(q3)*sin(q1)) + cos(q7)
*(cos(q5)*sin(q6)*(cos(q1)*cos(q3) - sin(q1)*sin(q3))
 + cos(q6)*sin(q5)*(cos(q1)*cos(q3) - sin(q1)*sin(q3)
))) - x_h*(cos(q9)*(cos(q7)*(cos(q1)*sin(q3) + cos(q3
)*sin(q1)) - sin(q7)*(cos(q5)*sin(q6)*(cos(q1)*cos(q3
) - sin(q1)*sin(q3)) + cos(q6)*sin(q5)*(cos(q1)*cos(
q3) - sin(q1)*sin(q3)))) - cos(q8)*sin(q9)*(sin(q7)*(
cos(q1)*sin(q3) + cos(q3)*sin(q1)) + cos(q7)*(cos(q5)
*sin(q6)*(cos(q1)*cos(q3) - sin(q1)*sin(q3)) + cos(q6
)*sin(q5)*(cos(q1)*cos(q3) - sin(q1)*sin(q3))))) - l3
*(cos(q7)*(cos(q1)*sin(q3) + cos(q3)*sin(q1)) - sin(
q7)*(cos(q5)*sin(q6)*(cos(q1)*cos(q3) - sin(q1)*sin(
q3)) + cos(q6)*sin(q5)*(cos(q1)*cos(q3) - sin(q1)*sin
(q3)))) - z_h*sin(q8)*(sin(q7)*(cos(q1)*sin(q3) + cos
(q3)*sin(q1)) + cos(q7)*(cos(q5)*sin(q6)*(cos(q1)*cos
(q3) - sin(q1)*sin(q3)) + cos(q6)*sin(q5)*(cos(q1)*
cos(q3) - sin(q1)*sin(q3)))),   z_h*(cos(q8)*(cos(q7)
*(cos(q1)*sin(q3) + cos(q3)*sin(q1)) - sin(q7)*(cos(
q5)*sin(q6)*(cos(q1)*cos(q3) - sin(q1)*sin(q3)) + cos
(q6)*sin(q5)*(cos(q1)*cos(q3) - sin(q1)*sin(q3)))) -
sin(q8)*(cos(q5)*cos(q6)*(cos(q1)*cos(q3) - sin(q1)*
sin(q3)) - sin(q5)*sin(q6)*(cos(q1)*cos(q3) - sin(q1)
```

```
*sin(q3)))) + x_h*sin(q9)*(sin(q8)*(cos(q7)*(cos(q1)*
sin(q3) + cos(q3)*sin(q1)) - sin(q7)*(cos(q5)*sin(q6)
*(cos(q1)*cos(q3) - sin(q1)*sin(q3)) + cos(q6)*sin(q5
)*(cos(q1)*cos(q3) - sin(q1)*sin(q3)))) + cos(q8)*(
cos(q5)*cos(q6)*(cos(q1)*cos(q3) - sin(q1)*sin(q3)) -
 sin(q5)*sin(q6)*(cos(q1)*cos(q3) - sin(q1)*sin(q3)))
), x_h*(sin(q9)*(sin(q7)*(cos(q1)*sin(q3) + cos(q3)*
sin(q1)) + cos(q7)*(cos(q5)*sin(q6)*(cos(q1)*cos(q3)
- sin(q1)*sin(q3)) + cos(q6)*sin(q5)*(cos(q1)*cos(q3)
 - sin(q1)*sin(q3)))) - cos(q9)*(cos(q8)*(cos(q7)*(
cos(q1)*sin(q3) + cos(q3)*sin(q1)) - sin(q7)*(cos(q5)
*sin(q6)*(cos(q1)*cos(q3) - sin(q1)*sin(q3)) + cos(q6
)*sin(q5)*(cos(q1)*cos(q3) - sin(q1)*sin(q3)))) - sin
(q8)*(cos(q5)*cos(q6)*(cos(q1)*cos(q3) - sin(q1)*sin(
q3)) - sin(q5)*sin(q6)*(cos(q1)*cos(q3) - sin(q1)*sin
(q3)))))],
[ z_h*(sin(q8)*(cos(q7)*(cos(q1)*sin(q3) + cos(q3)*
   sin(q1)) - sin(q7)*(cos(q5)*sin(q6)*(cos(q1)*cos(
   q3) - sin(q1)*sin(q3)) + cos(q6)*sin(q5)*(cos(q1)
   *cos(q3) - sin(q1)*sin(q3)))) + cos(q8)*(cos(q5)*
   cos(q6)*(cos(q1)*cos(q3) - sin(q1)*sin(q3)) - sin
   (q5)*sin(q6)*(cos(q1)*cos(q3) - sin(q1)*sin(q3)))
   ) - y_s*(cos(q1)*sin(q3) + cos(q3)*sin(q1)) - x_h
   *(cos(q9)*(sin(q7)*(cos(q1)*sin(q3) + cos(q3)*sin
   (q1)) + cos(q7)*(cos(q5)*sin(q6)*(cos(q1)*cos(q3)
    - sin(q1)*sin(q3)) + cos(q6)*sin(q5)*(cos(q1)*
   cos(q3) - sin(q1)*sin(q3)))) + sin(q9)*(cos(q8)*(
   cos(q7)*(cos(q1)*sin(q3) + cos(q3)*sin(q1)) - sin
   (q7)*(cos(q5)*sin(q6)*(cos(q1)*cos(q3) - sin(q1)*
   sin(q3)) + cos(q6)*sin(q5)*(cos(q1)*cos(q3) - sin
   (q1)*sin(q3)))) - sin(q8)*(cos(q5)*cos(q6)*(cos(
   q1)*cos(q3) - sin(q1)*sin(q3)) - sin(q5)*sin(q6)
   *(cos(q1)*cos(q3) - sin(q1)*sin(q3))))) + d2*cos(
   q1) - b3*(cos(q7)*(cos(q1)*sin(q3) + cos(q3)*sin(
   q1)) - sin(q7)*(cos(q5)*sin(q6)*(cos(q1)*cos(q3)
   - sin(q1)*sin(q3)) + cos(q6)*sin(q5)*(cos(q1)*cos
   (q3) - sin(q1)*sin(q3)))) - l3*(sin(q7)*(cos(q1)*
   sin(q3) + cos(q3)*sin(q1)) + cos(q7)*(cos(q5)*sin
   (q6)*(cos(q1)*cos(q3) - sin(q1)*sin(q3)) + cos(q6
   )*sin(q5)*(cos(q1)*cos(q3) - sin(q1)*sin(q3)))) +
    z_off*sin(q5)*(cos(q1)*cos(q3) - sin(q1)*sin(q3)
   ) + z_s*sin(q5)*(cos(q1)*cos(q3) - sin(q1)*sin(q3
   )), sin(q1), z_h*(sin(q8)*(cos(q7)*(cos(q1)*sin(
   q3) + cos(q3)*sin(q1)) - sin(q7)*(cos(q5)*sin(q6)
   *(cos(q1)*cos(q3) - sin(q1)*sin(q3)) + cos(q6)*
   sin(q5)*(cos(q1)*cos(q3) - sin(q1)*sin(q3)))) +
   cos(q8)*(cos(q5)*cos(q6)*(cos(q1)*cos(q3) - sin(
   q1)*sin(q3)) - sin(q5)*sin(q6)*(cos(q1)*cos(q3) -
```

```
 sin(q1)*sin(q3)))) - y_s*(cos(q1)*sin(q3) + cos(
q3)*sin(q1)) - x_h*(cos(q9)*(sin(q7)*(cos(q1)*sin
(q3) + cos(q3)*sin(q1)) + cos(q7)*(cos(q5)*sin(q6
)*(cos(q1)*cos(q3) - sin(q1)*sin(q3)) + cos(q6)*
sin(q5)*(cos(q1)*cos(q3) - sin(q1)*sin(q3)))) +
sin(q9)*(cos(q8)*(cos(q7)*(cos(q1)*sin(q3) + cos(
q3)*sin(q1)) - sin(q7)*(cos(q5)*sin(q6)*(cos(q1)*
cos(q3) - sin(q1)*sin(q3)) + cos(q6)*sin(q5)*(cos
(q1)*cos(q3) - sin(q1)*sin(q3)))) - sin(q8)*(cos(
q5)*cos(q6)*(cos(q1)*cos(q3) - sin(q1)*sin(q3)) -
 sin(q5)*sin(q6)*(cos(q1)*cos(q3) - sin(q1)*sin(
q3))))) - b3*(cos(q7)*(cos(q1)*sin(q3) + cos(q3)*
sin(q1)) - sin(q7)*(cos(q5)*sin(q6)*(cos(q1)*cos(
q3) - sin(q1)*sin(q3)) + cos(q6)*sin(q5)*(cos(q1)
*cos(q3) - sin(q1)*sin(q3)))) - l3*(sin(q7)*(cos(
q1)*sin(q3) + cos(q3)*sin(q1)) + cos(q7)*(cos(q5)
*sin(q6)*(cos(q1)*cos(q3) - sin(q1)*sin(q3)) +
cos(q6)*sin(q5)*(cos(q1)*cos(q3) - sin(q1)*sin(q3
)))) + z_off*sin(q5)*(cos(q1)*cos(q3) - sin(q1)*
sin(q3)) + z_s*sin(q5)*(cos(q1)*cos(q3) - sin(q1)
*sin(q3)), 0, z_off*cos(q5)*(cos(q1)*sin(q3) +
cos(q3)*sin(q1)) - z_h*(cos(q8)*(cos(q5)*sin(q6)
*(cos(q1)*sin(q3) + cos(q3)*sin(q1)) + cos(q6)*
sin(q5)*(cos(q1)*sin(q3) + cos(q3)*sin(q1))) +
sin(q7)*sin(q8)*(cos(q5)*cos(q6)*(cos(q1)*sin(q3)
 + cos(q3)*sin(q1)) - sin(q5)*sin(q6)*(cos(q1)*
sin(q3) + cos(q3)*sin(q1)))) - x_h*(sin(q9)*(sin(
q8)*(cos(q5)*sin(q6)*(cos(q1)*sin(q3) + cos(q3)*
sin(q1)) + cos(q6)*sin(q5)*(cos(q1)*sin(q3) + cos
(q3)*sin(q1))) - cos(q8)*sin(q7)*(cos(q5)*cos(q6)
*(cos(q1)*sin(q3) + cos(q3)*sin(q1)) - sin(q5)*
sin(q6)*(cos(q1)*sin(q3) + cos(q3)*sin(q1)))) +
cos(q7)*cos(q9)*(cos(q5)*cos(q6)*(cos(q1)*sin(q3)
 + cos(q3)*sin(q1)) - sin(q5)*sin(q6)*(cos(q1)*
sin(q3) + cos(q3)*sin(q1)))) + z_s*cos(q5)*(cos(
q1)*sin(q3) + cos(q3)*sin(q1)) - l3*cos(q7)*(cos(
q5)*cos(q6)*(cos(q1)*sin(q3) + cos(q3)*sin(q1)) -
 sin(q5)*sin(q6)*(cos(q1)*sin(q3) + cos(q3)*sin(
q1))) + b3*sin(q7)*(cos(q5)*cos(q6)*(cos(q1)*sin(
q3) + cos(q3)*sin(q1)) - sin(q5)*sin(q6)*(cos(q1)
*sin(q3) + cos(q3)*sin(q1))), b3*sin(q7)*(cos(q5)
*cos(q6)*(cos(q1)*sin(q3) + cos(q3)*sin(q1)) -
sin(q5)*sin(q6)*(cos(q1)*sin(q3) + cos(q3)*sin(q1
))) - z_h*(cos(q8)*(cos(q5)*sin(q6)*(cos(q1)*sin(
q3) + cos(q3)*sin(q1)) + cos(q6)*sin(q5)*(cos(q1)
*sin(q3) + cos(q3)*sin(q1))) + sin(q7)*sin(q8)*(
cos(q5)*cos(q6)*(cos(q1)*sin(q3) + cos(q3)*sin(q1
)) - sin(q5)*sin(q6)*(cos(q1)*sin(q3) + cos(q3)*
```

```
sin(q1)))) - l3*cos(q7)*(cos(q5)*cos(q6)*(cos(q1)
*sin(q3) + cos(q3)*sin(q1)) - sin(q5)*sin(q6)*(
cos(q1)*sin(q3) + cos(q3)*sin(q1))) - x_h*(sin(q9
)*(sin(q8)*(cos(q5)*sin(q6)*(cos(q1)*sin(q3) +
cos(q3)*sin(q1)) + cos(q6)*sin(q5)*(cos(q1)*sin(
q3) + cos(q3)*sin(q1))) - cos(q8)*sin(q7)*(cos(q5
)*cos(q6)*(cos(q1)*sin(q3) + cos(q3)*sin(q1)) -
sin(q5)*sin(q6)*(cos(q1)*sin(q3) + cos(q3)*sin(q1
)))) + cos(q7)*cos(q9)*(cos(q5)*cos(q6)*(cos(q1)*
sin(q3) + cos(q3)*sin(q1)) - sin(q5)*sin(q6)*(cos
(q1)*sin(q3) + cos(q3)*sin(q1)))), x_h*(cos(q9)*(
cos(q7)*(cos(q1)*cos(q3) - sin(q1)*sin(q3)) + sin
(q7)*(cos(q5)*sin(q6)*(cos(q1)*sin(q3) + cos(q3)*
sin(q1)) + cos(q6)*sin(q5)*(cos(q1)*sin(q3) + cos
(q3)*sin(q1)))) - cos(q8)*sin(q9)*(sin(q7)*(cos(
q1)*cos(q3) - sin(q1)*sin(q3)) - cos(q7)*(cos(q5)
*sin(q6)*(cos(q1)*sin(q3) + cos(q3)*sin(q1)) +
cos(q6)*sin(q5)*(cos(q1)*sin(q3) + cos(q3)*sin(q1
))))) - b3*(sin(q7)*(cos(q1)*cos(q3) - sin(q1)*
sin(q3)) - cos(q7)*(cos(q5)*sin(q6)*(cos(q1)*sin(
q3) + cos(q3)*sin(q1)) + cos(q6)*sin(q5)*(cos(q1)
*sin(q3) + cos(q3)*sin(q1)))) + l3*(cos(q7)*(cos(
q1)*cos(q3) - sin(q1)*sin(q3)) + sin(q7)*(cos(q5)
*sin(q6)*(cos(q1)*sin(q3) + cos(q3)*sin(q1)) +
cos(q6)*sin(q5)*(cos(q1)*sin(q3) + cos(q3)*sin(q1
)))) + z_h*sin(q8)*(sin(q7)*(cos(q1)*cos(q3) -
sin(q1)*sin(q3)) - cos(q7)*(cos(q5)*sin(q6)*(cos(
q1)*sin(q3) + cos(q3)*sin(q1)) + cos(q6)*sin(q5)
*(cos(q1)*sin(q3) + cos(q3)*sin(q1)))), - z_h*(
cos(q8)*(cos(q7)*(cos(q1)*cos(q3) - sin(q1)*sin(
q3)) + sin(q7)*(cos(q5)*sin(q6)*(cos(q1)*sin(q3)
+ cos(q3)*sin(q1)) + cos(q6)*sin(q5)*(cos(q1)*sin
(q3) + cos(q3)*sin(q1)))) + sin(q8)*(cos(q5)*cos(
q6)*(cos(q1)*sin(q3) + cos(q3)*sin(q1)) - sin(q5)
*sin(q6)*(cos(q1)*sin(q3) + cos(q3)*sin(q1)))) -
x_h*sin(q9)*(sin(q8)*(cos(q7)*(cos(q1)*cos(q3) -
sin(q1)*sin(q3)) + sin(q7)*(cos(q5)*sin(q6)*(cos(
q1)*sin(q3) + cos(q3)*sin(q1)) + cos(q6)*sin(q5)
*(cos(q1)*sin(q3) + cos(q3)*sin(q1)))) - cos(q8)
*(cos(q5)*cos(q6)*(cos(q1)*sin(q3) + cos(q3)*sin(
q1)) - sin(q5)*sin(q6)*(cos(q1)*sin(q3) + cos(q3)
*sin(q1)))), -x_h*(sin(q9)*(sin(q7)*(cos(q1)*cos(
q3) - sin(q1)*sin(q3)) - cos(q7)*(cos(q5)*sin(q6)
*(cos(q1)*sin(q3) + cos(q3)*sin(q1)) + cos(q6)*
sin(q5)*(cos(q1)*sin(q3) + cos(q3)*sin(q1)))) -
cos(q9)*(cos(q8)*(cos(q7)*(cos(q1)*cos(q3) - sin(
q1)*sin(q3)) + sin(q7)*(cos(q5)*sin(q6)*(cos(q1)*
sin(q3) + cos(q3)*sin(q1)) + cos(q6)*sin(q5)*(cos
```

```
        (q1)*sin(q3) + cos(q3)*sin(q1)))) + sin(q8)*(cos(
        q5)*cos(q6)*(cos(q1)*sin(q3) + cos(q3)*sin(q1)) -
         sin(q5)*sin(q6)*(cos(q1)*sin(q3) + cos(q3)*sin(
        q1))))))],
37      [

        0,          0,

        0, 1,

        l3*cos(q7)*(cos(q5)*sin(q6) + cos(q6)*sin(q5)) -
        z_off*sin(q5) - z_s*sin(q5) - x_h*(sin(q9)*(sin(
        q8)*(cos(q5)*cos(q6) - sin(q5)*sin(q6)) + cos(q8)
        *sin(q7)*(cos(q5)*sin(q6) + cos(q6)*sin(q5))) -
        cos(q7)*cos(q9)*(cos(q5)*sin(q6) + cos(q6)*sin(q5
        ))) - z_h*(cos(q8)*(cos(q5)*cos(q6) - sin(q5)*sin
        (q6)) - sin(q7)*sin(q8)*(cos(q5)*sin(q6) + cos(q6
        )*sin(q5))) - b3*sin(q7)*(cos(q5)*sin(q6) + cos(
        q6)*sin(q5)),

        l3*cos(q7)*(cos(q5)*sin(q6) + cos(q6)*sin(q5)) -
        x_h*(sin(q9)*(sin(q8)*(cos(q5)*cos(q6) - sin(q5)*
        sin(q6)) + cos(q8)*sin(q7)*(cos(q5)*sin(q6) + cos
        (q6)*sin(q5))) - cos(q7)*cos(q9)*(cos(q5)*sin(q6)
         + cos(q6)*sin(q5))) - z_h*(cos(q8)*(cos(q5)*cos(
        q6) - sin(q5)*sin(q6)) - sin(q7)*sin(q8)*(cos(q5)
        *sin(q6) + cos(q6)*sin(q5))) - b3*sin(q7)*(cos(q5
        )*sin(q6) + cos(q6)*sin(q5)),

        x_h*(cos(q9)*sin(q7)*(cos(q5)*cos(q6) - sin(q5)*
        sin(q6)) + cos(q7)*cos(q8)*sin(q9)*(cos(q5)*cos(
        q6) - sin(q5)*sin(q6))) + b3*cos(q7)*(cos(q5)*cos
        (q6) - sin(q5)*sin(q6)) + l3*sin(q7)*(cos(q5)*cos
        (q6) - sin(q5)*sin(q6)) - z_h*cos(q7)*sin(q8)*(
        cos(q5)*cos(q6) - sin(q5)*sin(q6)),

        z_h*(sin(q8)*(cos(q5)*sin(q6) + cos(q6)*sin(q5))
        - cos(q8)*sin(q7)*(cos(q5)*cos(q6) - sin(q5)*sin(
        q6))) - x_h*sin(q9)*(cos(q8)*(cos(q5)*sin(q6) +
        cos(q6)*sin(q5)) + sin(q7)*sin(q8)*(cos(q5)*cos(
        q6) - sin(q5)*sin(q6))),

        -x_h*(cos(q9)*(sin(q8)*(cos(q5)*sin(q6) + cos(q6)
        *sin(q5)) - cos(q8)*sin(q7)*(cos(q5)*cos(q6) -
        sin(q5)*sin(q6))) - cos(q7)*sin(q9)*(cos(q5)*cos(
        q6) - sin(q5)*sin(q6)))],
38      [
```

```
        0,          0,

        0,  0,

        - cos(q1)*sin(q3) - cos(q3)*sin(q1),

        - cos(q1)*sin(q3) - cos(q3)*sin(q1),

        cos(q5)*cos(q6)*(cos(q1)*cos(q3) - sin(q1)*sin(q3
        )) - sin(q5)*sin(q6)*(cos(q1)*cos(q3) - sin(q1)*
        sin(q3)),

        - sin(q7)*(cos(q1)*sin(q3) + cos(q3)*sin(q1)) -
        cos(q7)*(cos(q5)*sin(q6)*(cos(q1)*cos(q3) - sin(
        q1)*sin(q3)) + cos(q6)*sin(q5)*(cos(q1)*cos(q3) -
         sin(q1)*sin(q3))),

        sin(q8)*(cos(q7)*(cos(q1)*sin(q3) + cos(q3)*sin(
        q1)) - sin(q7)*(cos(q5)*sin(q6)*(cos(q1)*cos(q3)
        - sin(q1)*sin(q3)) + cos(q6)*sin(q5)*(cos(q1)*cos
        (q3) - sin(q1)*sin(q3)))) + cos(q8)*(cos(q5)*cos(
        q6)*(cos(q1)*cos(q3) - sin(q1)*sin(q3)) - sin(q5)
        *sin(q6)*(cos(q1)*cos(q3) - sin(q1)*sin(q3)))],
39      [

        0,          0,

        0,  0,

        cos(q1)*cos(q3) - sin(q1)*sin(q3),

        cos(q1)*cos(q3) - sin(q1)*sin(q3),

        cos(q5)*cos(q6)*(cos(q1)*sin(q3) + cos(q3)*sin(q1
        )) - sin(q5)*sin(q6)*(cos(q1)*sin(q3) + cos(q3)*
        sin(q1)),

        sin(q7)*(cos(q1)*cos(q3) - sin(q1)*sin(q3)) - cos
        (q7)*(cos(q5)*sin(q6)*(cos(q1)*sin(q3) + cos(q3)*
        sin(q1)) + cos(q6)*sin(q5)*(cos(q1)*sin(q3) + cos
        (q3)*sin(q1))),

        cos(q8)*(cos(q5)*cos(q6)*(cos(q1)*sin(q3) + cos(
        q3)*sin(q1)) - sin(q5)*sin(q6)*(cos(q1)*sin(q3) +
         cos(q3)*sin(q1))) - sin(q8)*(cos(q7)*(cos(q1)*
        cos(q3) - sin(q1)*sin(q3)) + sin(q7)*(cos(q5)*sin
        (q6)*(cos(q1)*sin(q3) + cos(q3)*sin(q1)) + cos(q6
        )*sin(q5)*(cos(q1)*sin(q3) + cos(q3)*sin(q1))))],
```

```
40          [
                1,          0,
                1,  0,
                0,
                0,
                - cos(q5)*sin(q6) - cos(q6)*sin(q5),
                -cos(q7)*(cos(q5)*cos(q6) - sin(q5)*sin(q6)),
                - cos(q8)*(cos(q5)*sin(q6) + cos(q6)*sin(q5)) -
                sin(q7)*sin(q8)*(cos(q5)*cos(q6) - sin(q5)*sin(q6
                ))]])
41
42      return J
43
44  def orientation_jacobian(q):
45
46      q1 = q[0]
47      q3 = q[2]
48      d4 = q[3]
49      q5 = q[4]
50      q6 = q[5]
51      q7 = q[6]
52      q8 = q[7]
53      q9 = q[8]
54
55      "Constants"
56      h = 145
57      d4 = d4+h
58      z_off = 85      # Offset from hip joint to torso
59      l3 = 105
60      b3 = 15
61      z_s = 100
62      x_h = 57.75+55.95
63      z_h = -12.31
64
65      J = array([[
            0,          0,
            0,  1,
            l3*cos(q7)*(cos(q5)*sin(q6) + cos(q6)*sin(q5)) -
```

```
z_off*sin(q5) - z_s*sin(q5) - x_h*(sin(q9)*(sin(q8)*(
cos(q5)*cos(q6) - sin(q5)*sin(q6)) + cos(q8)*sin(q7)
*(cos(q5)*sin(q6) + cos(q6)*sin(q5))) - cos(q7)*cos(
q9)*(cos(q5)*sin(q6) + cos(q6)*sin(q5))) - z_h*(cos(
q8)*(cos(q5)*cos(q6) - sin(q5)*sin(q6)) - sin(q7)*sin
(q8)*(cos(q5)*sin(q6) + cos(q6)*sin(q5))) - b3*sin(q7
)*(cos(q5)*sin(q6) + cos(q6)*sin(q5)),

l3*cos(q7)*(cos(q5)*sin(q6) + cos(q6)*sin(q5)) - x_h
*(sin(q9)*(sin(q8)*(cos(q5)*cos(q6) - sin(q5)*sin(q6)
) + cos(q8)*sin(q7)*(cos(q5)*sin(q6) + cos(q6)*sin(q5
))) - cos(q7)*cos(q9)*(cos(q5)*sin(q6) + cos(q6)*sin(
q5))) - z_h*(cos(q8)*(cos(q5)*cos(q6) - sin(q5)*sin(
q6)) - sin(q7)*sin(q8)*(cos(q5)*sin(q6) + cos(q6)*sin
(q5))) - b3*sin(q7)*(cos(q5)*sin(q6) + cos(q6)*sin(q5
)),

x_h*(cos(q9)*sin(q7)*(cos(q5)*cos(q6) - sin(q5)*sin(
q6)) + cos(q7)*cos(q8)*sin(q9)*(cos(q5)*cos(q6) - sin
(q5)*sin(q6))) + b3*cos(q7)*(cos(q5)*cos(q6) - sin(q5
)*sin(q6)) + l3*sin(q7)*(cos(q5)*cos(q6) - sin(q5)*
sin(q6)) - z_h*cos(q7)*sin(q8)*(cos(q5)*cos(q6) - sin
(q5)*sin(q6)),

z_h*(sin(q8)*(cos(q5)*sin(q6) + cos(q6)*sin(q5)) -
cos(q8)*sin(q7)*(cos(q5)*cos(q6) - sin(q5)*sin(q6)))
- x_h*sin(q9)*(cos(q8)*(cos(q5)*sin(q6) + cos(q6)*sin
(q5)) + sin(q7)*sin(q8)*(cos(q5)*cos(q6) - sin(q5)*
sin(q6))),

-x_h*(cos(q9)*(sin(q8)*(cos(q5)*sin(q6) + cos(q6)*sin
(q5)) - cos(q8)*sin(q7)*(cos(q5)*cos(q6) - sin(q5)*
sin(q6))) - cos(q7)*sin(q9)*(cos(q5)*cos(q6) - sin(q5
)*sin(q6)))],
66   [

     0,          0,

     0,  0,

     - cos(q1)*sin(q3) - cos(q3)*sin(q1),

     - cos(q1)*sin(q3) - cos(q3)*sin(q1),

     cos(q5)*cos(q6)*(cos(q1)*cos(q3) - sin(q1)*sin(q3
     )) - sin(q5)*sin(q6)*(cos(q1)*cos(q3) - sin(q1)*
     sin(q3)),
```

```
        - sin(q7)*(cos(q1)*sin(q3) + cos(q3)*sin(q1)) -
        cos(q7)*(cos(q5)*sin(q6)*(cos(q1)*cos(q3) - sin(
        q1)*sin(q3)) + cos(q6)*sin(q5)*(cos(q1)*cos(q3) -
         sin(q1)*sin(q3))),

        sin(q8)*(cos(q7)*(cos(q1)*sin(q3) + cos(q3)*sin(
        q1)) - sin(q7)*(cos(q5)*sin(q6)*(cos(q1)*cos(q3)
        - sin(q1)*sin(q3)) + cos(q6)*sin(q5)*(cos(q1)*cos
        (q3) - sin(q1)*sin(q3)))) + cos(q8)*(cos(q5)*cos(
        q6)*(cos(q1)*cos(q3) - sin(q1)*sin(q3)) - sin(q5)
        *sin(q6)*(cos(q1)*cos(q3) - sin(q1)*sin(q3)))],
[

        0,          0,

        0,  0,

        cos(q1)*cos(q3) - sin(q1)*sin(q3),

        cos(q1)*cos(q3) - sin(q1)*sin(q3),

        cos(q5)*cos(q6)*(cos(q1)*sin(q3) + cos(q3)*sin(q1
        )) - sin(q5)*sin(q6)*(cos(q1)*sin(q3) + cos(q3)*
        sin(q1)),

        sin(q7)*(cos(q1)*cos(q3) - sin(q1)*sin(q3)) - cos
        (q7)*(cos(q5)*sin(q6)*(cos(q1)*sin(q3) + cos(q3)*
        sin(q1)) + cos(q6)*sin(q5)*(cos(q1)*sin(q3) + cos
        (q3)*sin(q1))),

        cos(q8)*(cos(q5)*cos(q6)*(cos(q1)*sin(q3) + cos(
        q3)*sin(q1)) - sin(q5)*sin(q6)*(cos(q1)*sin(q3) +
         cos(q3)*sin(q1))) - sin(q8)*(cos(q7)*(cos(q1)*
        cos(q3) - sin(q1)*sin(q3)) + sin(q7)*(cos(q5)*sin
        (q6)*(cos(q1)*sin(q3) + cos(q3)*sin(q1)) + cos(q6
        )*sin(q5)*(cos(q1)*sin(q3) + cos(q3)*sin(q1))))],
[

        1,          0,

        1,  0,

        0,

        0,

        - cos(q5)*sin(q6) - cos(q6)*sin(q5),
```

```
            -cos(q7)*(cos(q5)*cos(q6) - sin(q5)*sin(q6)),

            - cos(q8)*(cos(q5)*sin(q6) + cos(q6)*sin(q5)) -
            sin(q7)*sin(q8)*(cos(q5)*cos(q6) - sin(q5)*sin(q6
            ))]])
69
70      return J
71
72  def euler_jacobian(q):
73
74      q1 = q[0]
75      q3 = q[2]
76      q5 = q[4]
77      q6 = q[5]
78      q7 = q[6]
79      q8 = q[7]
80
81      J = array([         [

          0,        0,

          0, 0,

          - cos(q1)*sin(q3) - cos(q3)*sin(q1),

          - cos(q1)*sin(q3) - cos(q3)*sin(q1),

          cos(q5)*cos(q6)*(cos(q1)*cos(q3) - sin(q1)*sin(q3)) -
           sin(q5)*sin(q6)*(cos(q1)*cos(q3) - sin(q1)*sin(q3)),


          - sin(q7)*(cos(q1)*sin(q3) + cos(q3)*sin(q1)) - cos(
          q7)*(cos(q5)*sin(q6)*(cos(q1)*cos(q3) - sin(q1)*sin(
          q3)) + cos(q6)*sin(q5)*(cos(q1)*cos(q3) - sin(q1)*sin
          (q3))),

          sin(q8)*(cos(q7)*(cos(q1)*sin(q3) + cos(q3)*sin(q1))
          - sin(q7)*(cos(q5)*sin(q6)*(cos(q1)*cos(q3) - sin(q1)
          *sin(q3)) + cos(q6)*sin(q5)*(cos(q1)*cos(q3) - sin(q1
          )*sin(q3)))) + cos(q8)*(cos(q5)*cos(q6)*(cos(q1)*cos(
          q3) - sin(q1)*sin(q3)) - sin(q5)*sin(q6)*(cos(q1)*cos
          (q3) - sin(q1)*sin(q3)))],
82          [

              0,        0,

              0, 0,
```

```
            cos(q1)*cos(q3) - sin(q1)*sin(q3),

            cos(q1)*cos(q3) - sin(q1)*sin(q3),

            cos(q5)*cos(q6)*(cos(q1)*sin(q3) + cos(q3)*sin(q1
            )) - sin(q5)*sin(q6)*(cos(q1)*sin(q3) + cos(q3)*
            sin(q1)),

            sin(q7)*(cos(q1)*cos(q3) - sin(q1)*sin(q3)) - cos
            (q7)*(cos(q5)*sin(q6)*(cos(q1)*sin(q3) + cos(q3)*
            sin(q1)) + cos(q6)*sin(q5)*(cos(q1)*sin(q3) + cos
            (q3)*sin(q1))),

            cos(q8)*(cos(q5)*cos(q6)*(cos(q1)*sin(q3) + cos(
            q3)*sin(q1)) - sin(q5)*sin(q6)*(cos(q1)*sin(q3) +
             cos(q3)*sin(q1))) - sin(q8)*(cos(q7)*(cos(q1)*
            cos(q3) - sin(q1)*sin(q3)) + sin(q7)*(cos(q5)*sin
            (q6)*(cos(q1)*sin(q3) + cos(q3)*sin(q1)) + cos(q6
            )*sin(q5)*(cos(q1)*sin(q3) + cos(q3)*sin(q1))))],
        [

            1,          0,

            1, 0,

            0,

            0,

            - cos(q5)*sin(q6) - cos(q6)*sin(q5),

            -cos(q7)*(cos(q5)*cos(q6) - sin(q5)*sin(q6)),

            - cos(q8)*(cos(q5)*sin(q6) + cos(q6)*sin(q5)) -
            sin(q7)*sin(q8)*(cos(q5)*cos(q6) - sin(q5)*sin(q6
            ))]])

    return J

def cartesian_jacobian(q):
    q1 = q[0]
    d2 = q[1]

    J = array([[-d2*sin(q1), cos(q1), 0, 0, 0, 0, 0, 0, 0],
    [d2*cos(q1), sin(q1), 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 1, 0, 0, 0, 0, 0],
    [1, 0, 1, 0, 0, 0, 0, 0, 0]])

```

```
96      return J
97
98
99  def locomotion_jacobian(q):
100     q1 = q[0]
101     d2 = q[1]
102
103     J = array([[-d2*sin(q1), cos(q1), 0, 0, 0, 0, 0, 0, 0],
104     [d2*cos(q1), sin(q1), 0, 0, 0, 0, 0, 0, 0],
105     [1, 0, 1, 0, 0, 0, 0, 0, 0]])
106
107     return J
```

Calculate dynamic weighting

```
1   # -*- coding: utf-8 -*-
2   """
3   Created on Thu Apr 14 14:55:54 2016
4   @author: Aasmund P. Hugo
5
6   Setting the weighting - or step size if you will -
        dynamically
7   """
8
9   import numpy as np
10  from forwardKinematics import FK
11
12  error_step_size = 0.0245
13
14  def getDynamicWeighting(e, dq, q_hat, damped_pseudo_inverse,
        x_hat, dof):
15
16      # Get an estimate of predicted config change
17      dx = FK(q_hat+dq, x_hat[3:6]) - x_hat
18      # Get a ratio to make a corrector
19      e_dx_ratio = error_step_size*np.abs((e/dx))
20
21      rows, columns = damped_pseudo_inverse.shape
22      alpha = np.identity(9)
23
24      # Iterate through every joint
25      for i in range(rows):
26          alpha_temp = 0
27
28          # Iterate through every error contributioon
29          for k in range(columns):
30              alpha_temp = alpha_temp + (damped_pseudo_inverse
                    [i][k]*e_dx_ratio[int(dof[k])])
```

```
31
32      return alpha
```

Fuzzify the error

```
1  # -*- coding: utf-8 -*-
2  """
3  Created on Thu Apr 14 15:00:20 2016
4  @author: Aasmund P. Hugo
5
6  Fuzzify the error, return a flag
7  """
8
9  import numpy as np
10
11 def fuzzify(e, q_hat, eps_ang, ifRotationFinished):
12
13     e_z = 1
14     e_theta = 0.5
15     e_phi = 0.5
16     e_psi = 0.5
17
18     if (e[2] + q_hat[3]) > 100 or (e[2] + q_hat[3]) < 0:
19         e_z = 0.1
20
21     if np.abs(e[3]) < eps_ang:
22         e_phi = 1
23
24     if np.abs(e[4]) < eps_ang:
25         e_theta = 1
26
27     if np.abs(e[5]) < eps_ang:
28         e_psi = 1
29
30     return e_z*e_phi*e_theta*e_psi*ifRotationFinished
```

Determine dynamic damping

```
1  # -*- coding: utf-8 -*-
2  """
3  Created on Wed Apr 13 18:15:19 2016
4  @author: Aasmund Hugo
5
6  Select Damping Factor
7  """
8  import numpy as np
9
10 "Singularity constants"
```

```
11 eps_sing = 0.15        # Minimum singular value for the
       Jacobian
12 lambda_max = 0.2      # Maximum damping of the Jacobian
       inversion
13
14 def getLambda(J):
15
16     U,S,V = np.linalg.svd(J)
17     columns = len(S)
18     Lambda = np.zeros([columns,columns])
19     for i in range(columns):
20         s = S[i]
21
22         if s<eps_sing:
23             Lambda[i][i] = ((1-np.cos(((eps_sing-s)/eps_sing
                   )*np.pi/2))*lambda_max)**2
24
25         else:
26             Lambda[i][i] = 0
27
28     return Lambda
```

Plot resulting trajectories, error developments and 3D-vectors

```
1  # -*- coding: utf-8 -*-
2  """
3  Plot inverse kinematic performances for Nao
4
5  Created on Thu Apr 14 14:06:31 2016
6
7  @author:  smund
8  """
9
10 import matplotlib.pyplot as plt
11 import numpy as np
12 from mpl_toolkits.mplot3d import Axes3D
13
14 # Plot in 3D the Cartesian trajectories with orientation
       vectors
15 def vectorPlot(x_hats, x_hats2):
16     #print('x hats:', x_hats)
17     n = len(x_hats[1,:])
18     #n = int(round(len(x_hats[1,:])/10))
19     u = []
20     v = []
21     w = []
22     xu = []
```

```
23      yv = []
24      zw = []
25      for i in range(0,n):
26          [rx, ry, rz] = x_hats[3:6,i]
27          rotmat = np.array([[np.cos(ry)*np.cos(rz), np.cos(rz
                )*np.sin(rx)*np.sin(ry)-np.sin(rz)*np.cos(rx), np
                .cos(rx)*np.cos(rz)*np.sin(ry)+np.sin(rx)*np.sin(
                rz)],
28                             [np.cos(ry)*np.sin(rz), np.sin(rz
                                   )*np.sin(rx)*np.sin(ry)+np.cos
                                   (rz)*np.cos(rx), np.cos(rx)*np
                                   .sin(rz)*np.sin(ry)-np.sin(rx)
                                   *np.cos(rz)],
29                             [-np.sin(ry), np.cos(ry)*np.sin(
                                   rx), np.cos(rx)*np.cos(ry)]])
30          local_x = np.array(x_hats[0:3,i])
31          vector_speed = np.matmul(rotmat,local_x)
32          u.append(vector_speed[0])
33          xu.append(local_x[0])
34          v.append(vector_speed[1])
35          yv.append(local_x[1])
36          w.append(vector_speed[2])
37          zw.append(local_x[2])
38      xlim = [np.min(xu)-10, np.max(xu)+10]
39      ylim = [np.min(yv)-10, np.max(yv)+10]
40      zlim = [np.min(zw)-10, np.max(zw)+10]
41      fig = plt.figure()
42      ax = fig.add_subplot(111, projection='3d')
43      #ax = fig.gca(projection='3d')
44      ax.set_xlim(xlim)
45      ax.set_ylim(ylim)
46      ax.set_zlim(zlim)
47      plt.plot(x_hats[0,:],x_hats[1,:],x_hats[2,:], 'b', label
            ='Sent to Nao')
48      ax.quiver(xu,yv,zw,u,v,w,
49                length=10,
50                color='Tomato'
51                )
52
53      u = []
54      v = []
55      w = []
56      xu = []
57      yv = []
58      zw = []
59      for i in range(0,n):
60          [rx, ry, rz] = x_hats2[3:6,i]
61          rotmat = np.array([[np.cos(ry)*np.cos(rz), np.cos(rz
```

```
            )*np.sin(rx)*np.sin(ry)-np.sin(rz)*np.cos(rx), np
            .cos(rx)*np.cos(rz)*np.sin(ry)+np.sin(rx)*np.sin(
            rz)],
62                                  [np.cos(ry)*np.sin(rz), np.sin(rz
                                        )*np.sin(rx)*np.sin(ry)+np.cos
                                        (rz)*np.cos(rx), np.cos(rx)*np
                                        .sin(rz)*np.sin(ry)-np.sin(rx)
                                        *np.cos(rz)],
63                                  [-np.sin(ry), np.cos(ry)*np.sin(
                                        rx), np.cos(rx)*np.cos(ry)]])
64          local_x = np.array(x_hats2[0:3,i])
65          vector_speed = np.matmul(rotmat,local_x)
66          u.append(vector_speed[0])
67          xu.append(local_x[0])
68          v.append(vector_speed[1])
69          yv.append(local_x[1])
70          w.append(vector_speed[2])
71          zw.append(local_x[2])
72      plt.plot(x_hats2[0,:],x_hats2[1,:],x_hats2[2,:], 'k',
            label='Measured by Nao')
73      ax.quiver(xu,yv,zw,u,v,w,
74                length=10,
75                color='Green'
76                )
77      ax.set_xlabel('x [mm]')
78      ax.set_ylabel('y [mm]')
79      ax.set_zlabel('z [mm]')
80      plt.show()
81
82
83  # Plot joint trajectories
84  def jointTraj(q_hats):
85      n = range(0,len(q_hats[0,:]))
86      fig = plt.figure(1)
87      plt.subplot(2,1,1)
88      plt.plot(n,q_hats[0,:],label='q1',color='y')
89      plt.plot(n,q_hats[2,:],label='q3',color='c')
90      plt.plot(n,q_hats[4,:],label='q5',color='r')
91      plt.plot(n,q_hats[5,:],label='q6',color='k')
92      plt.plot(n,q_hats[6,:],label='q7',color='b')
93      plt.plot(n,q_hats[7,:],label='q8',color='g')
94      plt.plot(n,q_hats[8,:],label='q9',color='m')
95      plt.legend(bbox_to_anchor=(0.98, 1), loc=2,
            borderaxespad=0.)
96      plt.ylabel('[rad]')
97      plt.xlabel('Iterations')
98      plt.title('Revolute Joints')
99      plt.subplot(2,1,2)
```

```
100         plt.plot(n,q_hats[1,:],label='d2',color='b')
101         plt.plot(n,q_hats[3,:],label='d4',color='r')
102         plt.ylabel('[mm]')
103         plt.xlabel('Iterations')
104         plt.title('Prismatic Joints')
105         #plt.legend(loc=3)
106         plt.legend(bbox_to_anchor=(0.98, 1), loc=2,
                borderaxespad=0.)
107         fig.subplots_adjust(hspace=0.4)
108         plt.show()
109
110
111 # Plot trajectories of position in Cartesian space, with
        Euler angles
112 def cartesianTraj(x_hats):
113         n = range(0,len(x_hats[0,:]))
114         fig = plt.figure(2)
115         plt.subplot(2,1,1)
116         plt.plot(n,x_hats[0,:],label='x',color='r')
117         plt.plot(n,x_hats[1,:],label='y',color='k')
118         plt.plot(n,x_hats[2,:],label='z',color='b')
119         plt.legend(bbox_to_anchor=(0.98, 1), loc=2,
                borderaxespad=0.)
120         plt.ylabel('[mm]')
121         plt.xlabel('Iterations')
122         plt.title('Cartesian coordinates')
123         #plt.legend(loc=1)
124         plt.subplot(2,1,2)
125         plt.plot(n,x_hats[3,:],label='$\phi$',color='g')
126         plt.plot(n,x_hats[4,:],label='$\\theta$',color='b')
127         plt.plot(n,x_hats[5,:],label='$\psi$',color='r')
128         #plt.legend(loc=4)
129         plt.legend(bbox_to_anchor=(0.98, 1), loc=2,
                borderaxespad=0.)
130         plt.ylabel('[rad]')
131         plt.xlabel('Iterations')
132         plt.title('Euler angles')
133         fig.subplots_adjust(hspace=0.4)
134         plt.show()
135
136
137 # Plot the converging errors
138 def errorTraj(errors):
139         n = range(0,len(errors[0,:]))
140         fig = plt.figure(2)
141         plt.subplot(2,1,1)
142         plt.plot(n,errors[0,:],label='x',color='r')
143         plt.plot(n,errors[1,:],label='y',color='k')
```

```
144     plt.plot(n,errors[2,:],label='z',color='b')
145     plt.legend(bbox_to_anchor=(0.98, 1), loc=2,
            borderaxespad=0.)
146     plt.ylabel('[mm]')
147     plt.xlabel('Iterations')
148     plt.title('Cartesian errors')
149     #plt.legend(loc=1)
150     plt.subplot(2,1,2)
151     plt.plot(n,errors[3,:],label='$\phi$',color='g')
152     plt.plot(n,errors[4,:],label='$\\theta$',color='b')
153     plt.plot(n,errors[5,:],label='$\psi$',color='r')
154     #plt.legend(loc=4)
155     plt.legend(bbox_to_anchor=(0.98, 1), loc=2,
            borderaxespad=0.)
156     plt.ylabel('[rad]')
157     plt.xlabel('Iterations')
158     plt.title('Euler angle errors')
159     fig.subplots_adjust(hspace=0.4)
160     plt.show()
```

Appendix C

This Appendix provides the basic examples for connecting with Nao and sending easy commands for Cartesian and joint control. This is performed in *Python*, but can be done through other languages as well (first and foremost *C++*).

How to connect to Nao

This is a connection example which have been used as a basis for connecting and communicating. The basic examples are gathered from Aldebaran's documentation for Nao, but this is also cited as head comments in the respective listing.

```
1  # -*- coding: utf-8 -*-
2  """
3  Created on Wed April 25 21:29:15 2016
4
5  From:
6  http://doc.aldebaran.com/2-1/dev/python/examples/motion/
7  cartesian.html#python-example-motion-cartesian
8  """
9
10 import sys
11 from naoqi import ALProxy
12 import numpy as np
13 import time as t
14 from FLDLS import FLDLS
15 from goToConfiguration import goToConfig
16
17 port = 9559
18 naoIP = "192.168.1.100"
19
20 def main(naoIP):
21
22     try:
23         motionProxy = ALProxy("ALMotion", naoIP, port)
24     except Exception as e:
25         print("Could not create proxy to ALMotion")
26         print("Error was: ",e)
27         sys.exit(1)
28     try:
```

```
29          postureProxy = ALProxy("ALRobotPosture", naoIP, port
                )
30      except:
31          print("Could not create proxy to ALRobotPosture")
32          sys.exit(1)
33
34
35  if __name__ == "__main__":
36      robotIp = "192.168.1.100"
37
38      if len(sys.argv) <= 1:
39          print("Usage python almotion_setangles.py robotIP (
                optional default: 127.0.0.1)")
40      else:
41          robotIp = sys.argv[1]
42
43  main(naoIP)
```

How to send commands to Nao

This example is gathered from Aldebaran's documentation for Nao, and exemplifies concurrent control of torso, right arm and left arm.

```
1  # -*- coding: utf-8 -*-
2  """
3  Created on Wed April 25 21:29:15 2016
4
5  From:
6  http://doc.aldebaran.com/2-1/dev/python/examples/motion/
7  cartesian.html#python-example-motion-cartesian
8  """
9
10 import argparse
11 import motion
12 import almath
13 from naoqi import ALProxy
14
15 port = 9559
16 naoIP = "192.168.1.100"
17
18 def main(robotIP, PORT=9559):
19     ''' Simultaneously control three effectors:
20     the Torso, the Left Arm and the Right Arm
21     Warning: Needs a PoseInit before executing
22     '''
23
24     motionProxy  = ALProxy("ALMotion", robotIP, PORT)
25     postureProxy = ALProxy("ALRobotPosture", robotIP, PORT)
```

```
26
27      # Wake up robot
28      motionProxy.wakeUp()
29
30      # Send robot to Stand Init
31      postureProxy.goToPosture("StandInit", 0.5)
32
33      frame      = motion.FRAME_WORLD
34      coef       = 0.5                    # motion speed
35      times      = [coef, 2.0*coef, 3.0*coef, 4.0*coef]
36      useSensorValues = False
37
38      # Relative movement between current and desired
            positions
39      dy         = +0.03                  # translation axis Y
            (meters)
40      dz         = -0.03                  # translation axis Z
            (meters)
41      dwx        = +8.0*almath.TO_RAD     # rotation axis X (
            radians)
42
43      # Motion of Torso with post process
44      effector   = "Torso"
45
46      path = []
47      initTf = almath.Transform(motionProxy.getTransform(
            effector, frame, useSensorValues))
48      # point 1
49      deltaTf  = almath.Transform(0.0, -dy, dz)*almath.
            Transform().fromRotX(-dwx)
50      targetTf = initTf*deltaTf
51      path.append(list(targetTf.toVector()))
52
53      # point 2
54      path.append(list(initTf.toVector()))
55
56      # point 3
57      deltaTf  = almath.Transform(0.0, dy, dz)*almath.
            Transform().fromRotX(dwx)
58      targetTf = initTf*deltaTf
59      path.append(list(targetTf.toVector()))
60
61      # point 4
62      path.append(list(initTf.toVector()))
63
64      axisMask   = almath.AXIS_MASK_ALL  # control all the
            effector axes
```

```
65      motionProxy.post.transformInterpolations(effector, frame
            , path,
66                                               axisMask, times)
67
68      # Motion of Arms with block process
69      frame      = motion.FRAME_TORSO
70      axisMask   = almath.AXIS_MASK_VEL  # control just the
            position
71      times      = [1.0*coef, 2.0*coef]  # seconds
72
73      # Motion of Right Arm during the first half of the Torso
            motion
74      effector   = "RArm"
75
76      path = []
77      currentTf = motionProxy.getTransform(effector, frame,
            useSensorValues)
78      targetTf  = almath.Transform(currentTf)
79      targetTf.r2_c4 -= 0.04 # y
80      path.append(list(targetTf.toVector()))
81      path.append(currentTf)
82
83      motionProxy.transformInterpolations(effector, frame,
            path, axisMask, times)
84
85      # Motion of Left Arm during the last half of the Torso
            motion
86      effector    = "LArm"
87
88      path = []
89      currentTf = motionProxy.getTransform(effector, frame,
            useSensorValues)
90      targetTf  = almath.Transform(currentTf)
91      targetTf.r2_c4 += 0.04 # y
92      path.append(list(targetTf.toVector()))
93      path.append(currentTf)
94
95      motionProxy.transformInterpolations(effector, frame,
            path, axisMask, times)
96
97
98  if __name__ == "__main__":
99      parser = argparse.ArgumentParser()
100     parser.add_argument("--ip", type=str, default="127.0.0.1
            ",
101                         help="Robot ip address")
102     parser.add_argument("--port", type=int, default=9559,
103                         help="Robot port number")
```

```
104
105         args = parser.parse_args()
106         main(args.ip, args.port)
```