Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Engineering
Department of Mechanical and Industrial Engineering

Dag Lofthus

From CAD-Assemblies to Constraint Based Robot Control

Master's thesis in Production Technology
Supervisor: Lars Tingelstad
June 2019

NTNU
Norwegian University of
Science and Technology

Dag Lofthus

From CAD-Assemblies to Constraint Based Robot Control

Master's thesis in Production Technology
Supervisor: Lars Tingelstad
June 2019

Norwegian University of Science and Technology
Faculty of Engineering
Department of Mechanical and Industrial Engineering

Acknowledgements

First and foremost, I would like to thank my supervisor Lars Tingelstad for lots and lots of valuable help during my work. That was anything from debugging and fixing of the hardware system of the robot, to things as easy as how I should connect to internet. A lot of what I have learned about robotics and programming can I thank him for. I couldn't have done it without you Lars. Mathias Arbo have also been really helpful, by learning me things about constraint based programming in general and eTaSL in particular, that I wasn't able to wrap my head around myself.

I would also like to thank my flatmates, Rasmus, Anna Sara and Synne for motivating me and getting me out of bed on days where I was anything from motivated. From the bottom of my heart, thank you guys! Last but not least, I must thank the rest of the guys in my masters office (you know who you are) for being the best discussion partners ever. We have really made each other better, and wiser, by helping each other out whenever stuck. That would be both programming fundamentals and intricacies, writing, life in general, girls, etc. etc. Special thanks to Eirik Wik Haug for helping me with robot 3D-vision during busy times right before the thesis wrap up.

Abstract

This Master's thesis are aiming to find out if it is possible to program a robot solely by information contained in assembly data from 3D design software (CAD). More so, how this program can be implemented on a commercially available off-the-shelf robot manipulator. The solution should be developed with the potential relevance for the industry in mind. Ever more production firms need to give way for the robots to survive. How can the programming be easier and faster?

The programming is approached with the newly open-sourced framework for constraint based robot programming, called eTaSL. eTaSL is as of now the most complete system for constraint based robot control. It is already a fully functional framework, but to be available for more systems, parts of the project has been to adapt it to ROS and ROS Control.

CAD constraints are extracted from SolidWorks by a C# code which utilizes their API. The way those constraints are formulated mathematically led to the making of a new set of geometrical functions and constraint types for eTaSL. These have been used with the new eTaSL ROS controller to define a peg-in-hole test case.

It has been shown that the implemented solution is able to complete the peg-in-hole case with a KUKA KR6 R900 robot using their sensor interface RSI. This was tested with three different sets of pegs with different tolerances. The precision of the system is good enough to be able to even place the pegs with the tightest tolerance, which does not allow for any error larger than ± 0.016 mm on average.

It is concluded that even though this system is in early stages, does it present a promising future for robot programming. That is because it can be implemented on a commercial robot and that it has the possibility to reduce the time consumption from the design stage to a fully assembled product.

Sammendrag

Denne masteroppgaven har som mål å finne ut om det er mulig å programmere en robot utelukkende med informasjon tilgjengelig i 3D-design software (CAD). Mer spesifikt, hvordan dette programmet kan bli implementert på en kommersielt tilgjengelig robot. Løsningen skal implementeres med industrien og dens relevans for dem i bakhodet. Stadig flere produksjonsbedrifter skaffer seg roboter for å overleve. Hvordan kan programmeringen av disse gjøres enklere og raskere?

Programmeringsproblemet er tilnærmet med et nylig "*open-sourced*" rammeverk for "*constraint based*" robotprogrammering, kalt eTaSL. eTaSL er for øyeblikket det mest komplette systemet innen dette feltet. Det er et allerede fullt funksjonibelt rammeverk, men for å gjøre det mer tilgjengelig for flere robotsystemer, har en del av prosjektet blitt å tilpasse det for ROS og ROS Control.

Geometriske relasjoner fra CAD er hentet ut fra SolidWorks med en C# kode som bruker deres API. Måten SolidWorks formulerer denne geometriske informasjonen matematisk førte til at et nytt sett med geometriske funksjoner og relasjonstyper ble utviklet for eTaSL. De har blitt brukt med den nye eTaSL-ROS-kontrolleren til å definere et "*peg-in-hole*" testforsøk.

Det har blitt vist at den implementerte løsningen får til å gjennomføre "*peg-in-hole*"-forsøket med en KUKA KR6 R900 robot gjennom deres sensor grensesnitt RSI. Testen ble gjort med tre forskjellige sett med sylindere med forskjellige toleranser. Presisjonen i systemet er så bra at det er mulig å plassere selv de sylinderene med trangest toleranse, som ikke tillater et større avvik enn ± 0.016 mm.

Det konkluderes med at selv om systemet er i en tidlig fase, så leder det vei for en lovende fremtid for robotprogrammering. Det er fordi det kan bli implementert på en kommersiell robot og at det har muligheten til å redusere tidsforbruket mellom design fasen og et ferdig sammenstilt produkt.

Contents

Acronyms

API	Application Programming Interface
CAD	Computer-Aided Design
DH	Denavit-Hartenberg
DOF	Degrees of Freedom
eTaSL	Expressiongraph-based Task Specification Language
FSM	Finite-State Machines
ISO	International Organization for Standardization
iTaSC	Instantaneous Task Specification using Constraints
KRL	Kuka Robot Language
NTNU	Norwegian University of Science and Technology
QP	Quadratic Programming
qpOASES	Online Active Set Strategy for quadratic programming
ROS	Robot Operating System
RSI	Robot Sensor Interface
STEP	Standard for the Exchange of Product data
SW	SolidWorks
URDF	Unified Robot Description Format

List of Figures

List of Tables

Listings

Chapter 1.

Introduction

The main aim of the work in this Master's thesis is to present the possibilities of constraint based robot programming. Particularly the eTaSL framework when using it to define a constraint based task from geometrical constraints derived from CAD software. More so, this thesis will present how we can utilize interrelations between parts to develop software that can determine a trajectory and constrain the motion of an industrial robot in an assembly task. As these interrelations, more precisely termed geometrical constraints, are already defined by designers during product development in 3D design software (CAD), there should be no need for robot programmers to repeat that process.

The most common method for robot programming in an assembly task today is to manually program the sequence by the pose of the robot arm. This is done either by positioning the robot at intermediate steps saving the pose, called online programming, or by offline programming where changes are hard to make. Recent research has made it possible for the programmers to have a more intuitive job in offline programming, by taking advantage of CAD-models for graphical programming interfaces. Solutions to online control does also begin to be more and more robust. One of the best solutions so far is constraint based programming. More on that later.

The work in this thesis is also meant to be the initial work for a larger project on "*CAD to constraint based robot programming*" at the Department of Mechanical and Industrial Engineering at the Norwegian University of Science and Technology (NTNU).

1.1. Research Questions

This Master's thesis aims to answer:

1. **Is it possible to program a robot solely by information contained in CAD-constraints?**
2. **How can it be implemented on a commercially available off-the-shelf robot manipulator?**
3. **How can this project be of relevance to the industry?**

As we step into the realms of Industry 4.0, a need for more flexible and easier to use industrial manipulators are arising. Ever more businesses have to consider the adoption of automation in their production to survive, as payments increase and technology becomes cheaper. The field of robotics is one of the fields which looks into production automation. Industrial robots have already enabled the industry to speed up production and reduce cost, but to make the final step into Industry 4.0, the flexibility of production lines need to get down to *"size 1 batches."* That is, production where changing between different products on the same production line is not more time consuming than big batches. In robot assembly processes, this can be partly achieved by automating the programming process. Within the frame of robot control and assembly tasks, there have been developed many different ways of approaching this challenge. A presentation of the most related work follows.

1.2. Related Work

1.2.1. Constraint-Based Robot Control

Constraint-based control is built on the premises of spatial relationships between parts and the environment in a robot operation. These can be mathematically described by equalities and inequalities between features on rigid bodies. Instead of instructing the robot with end-effector positions and orientations, it is instructed on the relation between the bodies it transforms. That is, the instructions could be: *"keep at minimum some distance clearance from an obstacle"* or *"position the face of the part such that it is equal to the face of another part."*

One way of dividing a robot operation into manageable chunks is to use a task specification approach. In such an approach, the operation is divided into elementary sub-tasks. An elementary sub-task could be a single rotation of a part, a single linear translation, a maximal contact force or other similar tasks, where any further subdivision is impossible.

Aertbeliën and De Schutter [3], presents a constraint-based task specification language using expression graphs (eTaSL), where expression graphs are treelike representations of arithmetical operations. Any parameter can be constrained as long as it is possible to formulate it mathematically. The constraints are weighted position-level or velocity-level, where a position-level constraint expresses a desired position the robot evolves towards in a control loop. The weighting ensures that in a situation where different constraints conflicts, the highest prioritized constraint will be satisfied first. A lower priority can be a trajectory, while a higher priority can be a clearance to obstacles. Thus, the trajectory constraint will be temporarily omitted if the clearance constraint is violated.

A similar approach for constraint-based control, where a constraint is used in the broad term, is presented by Smits et al. [34] in an instantaneous task specification using constraints (iTaSC). The constraints could be a distance between robot and operator, a fixed relation between two parts in two robotic arms, a maximal end-effector velocity, etc. They have developed a control scheme, which is able to derive the control equations for a task involving ten primary constraints.

Mansard and Chaumette [24], divides the constraints into sub-tasks in a stack during sensor based control, to avoid locking the robot motion to the predefined constraints, thus being able to temporarily conflict the constraints for instance when avoidance of obstacles are necessary. The stack of tasks is prioritized bottom up, where the highest priority task, the position, is satisfied first. Only the sensor data concerning the position is at the stack until a secondary controller sees a potential obstacle. Then the necessary tasks for avoidance are added to the stack.

Somani et al. [30, 36] proposed a framework for robot programming using geometric constraints specified in CAD software. Part models are imported from STEP, which is a vendor-independent file format developed for the interaction of CAD data between different CAD software. Not all information is available in the STEP format, as for instance geometrical constraint information. Thus Somani et al. have the geometrical constraints specified by the robot programmer in a self-developed software. One advantage of using STEP is that they make a

system suited for businesses with different software in-house. Somani et al. have also worked on the mathematical models of the constraints defined in the software and geometric properties of constraint nullspaces, to make an exact solver for a robot trajectory optimization for geometrical constraints [37].

A CAD based approach to constraint based control in eTaSL is outlined by Arbo et al. [5]. The extracted CAD data are parameters as feature frames, insertion lengths etc. The task in eTaSL is defined by a skill set, where the extracted CAD data is used by the user to chose the necessary skills for the task. The CAD data is also used with fussy logic to define parameters for force control.

1.2.2. Robot Based Assembly

The robot-based assembly task is solved in many different ways in the literature. Classical approaches to robotic assembly are often cumbersome since the programming is done by either manual hardcoding or manual drag-and-teach. More recently a way of solving the problem is described by Perzylo et al. [29]. They have made a system where they want the programming to be more intuitive for a workshop floor programmer. They achieve this by making augmented cognitive helping systems for the programmer.

Ji et al. [18], are trying to reduce human intervention in the programming task by having made a programming-free robot assembly method based on virtual training, where assembly sequences are tested iteratively in the virtual world until a feasible assembly plan is generated. Gao et al. [15] are also solving the problem in a new way in considering a force feedback assembly approach for the peg-in-hole problem, using neural networks to process the online data from the force feedback.

1.2.3. Extraction of Mating Features

The method for extraction of mating features/geometrical constraints from CAD models used in this paper is highly influenced by papers on assembly sequencing. One of those papers is presented by Mathew et al. [25, 26]. They use SolidWorks API to extract mating information from an assembly model, for generation of all feasible assembly sequences. The SW data is then used for the determination of

the optimum sequence, where the results are represented in a liaison graph. A liaison graph is a simple graphical representation of parts as nodes and relations between parts as liaison arcs between the nodes. Mathew et al. aims at reducing the time needed for planning of the assembly sequencing, thus also the cost, for human-operated assembly. A similar approach is made by Agrawal et al. [4]. They use the SW API to extract information on interference directions to make an interference matrix. Together with a fitness function aimed at reducing tool changes and part re-orientations, a generic algorithm is used to find an optimal disassembly sequence for human operation. The interference matrix give a representation of possible disassembly directions in relation to the other parts in the assembly.

Pan et al. [28] does also generate interference matrices, but calculates the interference based on a 2D projection of the assembly from STEP files instead. Their goal is to generate the optimal assembly sequence.

1.3. Contributions

There are three main contributions of this thesis:

1. **Constraint based robot programming based solely on geometric relations between rigid bodies.** Each sub task in the assembly task are defined purely by geometric relations, derived from CAD constraints, with their end goal. There is for instance not sat any constraints on the intermediate trajectory. This is done to test whether the information contained in CAD assembly models can be enough to define an assembly task, hence minimizing the potential need for human interaction (programming), between a designers finalizing of an assembly model and the actual robot assembly of the parts.
2. **ROS based eTaSL controller.** Together with Lars Tingelstad, an eTaSL controller based on ROS has been developed. As many robots are or can be controlled by ROS and the original implemetation of eTaSL is based on Orocos, it seemed necessary to formulate the ROS controller. This would also be the easiest way of being able to run eTaSL on the KUKA robots in the robot lab at NTNU.
3. **eTaSL ROS controller implemented on a industrial off-the-shelf**

robot. To make this project relevant to the industry, the eTaSL ROS controller are implemented on a commercially available off-the-shelf robot manipulator. Specifically the KUKA KR6 R900 robot. For this to work, a joint limit interface had to be implemented in the hardware interface for KUKA RSI.

1.4. Outline of the Thesis

The robot assembly problem is approached by developing software which is based on SolidWorks [7] API. The developed software will be extracting the mathematical relations in all of the predefined geometrical constraints from assembly files in SW. The output of this software is then used to define a set of functions for geometrical relations, which can be used in any assembly task with the constraint based robot programming framework in eTaSL. eTaSL is a recently open sourced framework for constraint based robot programming, by Aertbeliën and De Schutter [3]. The geometrical functions are then used with eTaSL to define a constraint based assembly task for the classical peg-in-hole problem. By working on the peg-in-hole task, we want to expose possibilities and limitations in both the eTaSL framework, and a newly developed ROS based controller for eTaSL by Lars Tingelstad. Any missing links or bugs in the eTaSL ROS controller should be solved.

This thesis will first introduce some basics on robot kinematics and screw theory in chapter 2, before an outline of all necessary software and programming tools essential for understanding the work in this thesis, is presented in chapter 3. The main work and contribution of this thesis is presented in chapter 4 and 5, with the mathematics of the function set for geometrical relations, and how CAD constraints can be implemented in constraint based frameworks as eTaSL. The test setup and belonging results for validating the implemented task are given in chapter 6, before a discussion and the final conclusion rounds off this thesis in chapter 7 and 8 respectively.

Chapter 2.

Robot Kinematics and Screw Theory

In this chapter, we present the basics of robot kinematics that are necessary for the development of our assembly task. You will also be presented for the fundamentals behind the mathematics of screw theory. The mathematics presented are based on work by Siciliano et al. [33] in the book *"Robotics – Modelling, Planning and Control"* and a note on *"Advanced Robotics"* made available by O. Egeland [10].

2.1. Rigid Body Motions

The position and orientation of a rigid body in space can be fully described in respect to a reference frame by six parameters. An example of such a body is shown in Figure 2.1. The position and orientation are often denoted as the *pose* of the body. The position can be described by a vector,

$$\boldsymbol{p} = [p_x, p_y, p_z]^T \in \mathbb{R}^3 \tag{2.1}$$

pointing from the reference frame $\{\boldsymbol{a}\}$ to the frame $\{\boldsymbol{b}\}$ of the rigid body.

The direction of $\boldsymbol{x}'$, $\boldsymbol{y}'$, and $\boldsymbol{z}'$ can be described by parts of x, y, and z in the

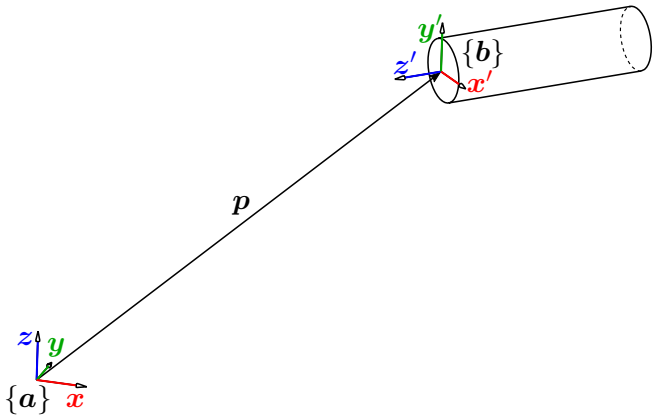


Figure 2.1.: Rigid body

reference frame;

$$\begin{aligned} \boldsymbol{x}' &= x'_x\boldsymbol{x} + x'_y\boldsymbol{y} + x'_z\boldsymbol{z} \\ \boldsymbol{y}' &= y'_x\boldsymbol{x} + y'_y\boldsymbol{y} + y'_z\boldsymbol{z}\,. \\ \boldsymbol{z}' &= z'_x\boldsymbol{x} + z'_y\boldsymbol{y} + z'_z\boldsymbol{z} \end{aligned} \tag{2.2}$$

2.1.1. Rotation Matrices

The vectors $\boldsymbol{x}'$, $\boldsymbol{y}'$, and $\boldsymbol{z}'$ from (2.2) can be derived from the vectors $\boldsymbol{x}$, $\boldsymbol{y}$, and $\boldsymbol{z}$ by multiplication with a matrix $\boldsymbol{R} \in \mathbb{R}^{3\times3}$ called a *rotation matrix*;

$$\boldsymbol{R} = \begin{bmatrix} x'_x & y'_x & z'_x \\ x'_y & y'_y & z'_y \\ x'_z & y'_z & z'_z \end{bmatrix}, \tag{2.3}$$

where

$$[\boldsymbol{x}', \boldsymbol{y}', \boldsymbol{z}']^T = \boldsymbol{R}[\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{z}]^T.$$

Rotation matrices can be used to present a vector $\boldsymbol{q}$ in $\{\boldsymbol{a}\}$ as a rotation relative to $\{\boldsymbol{b}\}$. This is possible, as $\boldsymbol{R} \in SO(3)$. $SO(3)$ is called the *special orthogonal group*, which is a set of all matrices $\boldsymbol{R} \in \mathbb{R}^{3\times3}$ that satisfies $\boldsymbol{R}^T\boldsymbol{R} = \boldsymbol{R}\boldsymbol{R}^T = \boldsymbol{I}$ and $\det \boldsymbol{R} = 1$. All members of $SO(3)$ are linear transformations that preserve both lengths and rotations.

A point in space $\boldsymbol{q}$ can be transformed from frame $\{\boldsymbol{b}\}$ to frame $\{\boldsymbol{a}\}$ by

$$\boldsymbol{q}^{\boldsymbol{a}} = \boldsymbol{R}_{\boldsymbol{b}}^{\boldsymbol{a}} \boldsymbol{q}^{\boldsymbol{b}}, \tag{2.4}$$

where $\boldsymbol{R}_{\boldsymbol{b}}^{\boldsymbol{a}}$ denotes a rotation from $\{\boldsymbol{b}\}$ to $\{\boldsymbol{a}\}$. If we rearrange (2.4), we can transform the point back

$$\boldsymbol{q}^{\boldsymbol{b}} = \left(\boldsymbol{R}_{\boldsymbol{b}}^{\boldsymbol{a}}\right)^{-1} \boldsymbol{q}^{\boldsymbol{a}}. \tag{2.5}$$

The properties of $SO(3)$ is giving this useful result:

$$\boldsymbol{R}^{-1} = \boldsymbol{R}^{T}. \tag{2.6}$$

Thus by (2.6), (2.5) can be rewritten to

$$\boldsymbol{q}^{\boldsymbol{b}} = \left(\boldsymbol{R}_{\boldsymbol{b}}^{\boldsymbol{a}}\right)^{T} \boldsymbol{q}^{\boldsymbol{a}}.$$

The rotation of a vector from one frame to another is equivalent to rotating the vector within a single frame by the same angels.

Quaternion

A useful mathematical description of rotations of rigid bodies in space is the quaternion. A unit quaternion is defined as $\mathcal{Q} = \{\boldsymbol{\epsilon}, \eta\}$ where

$$\boldsymbol{\epsilon} = \sin\left(\frac{\theta}{2}\right)\boldsymbol{n}$$

and

$$\eta = \cos\left(\frac{\theta}{2}\right)$$

are called the vector part and the scalar part of the quaternion respectively. The quaternion is useful to construct a rotation matrix for a rotation of θ radians about some unit vector $\boldsymbol{n}$. That is,

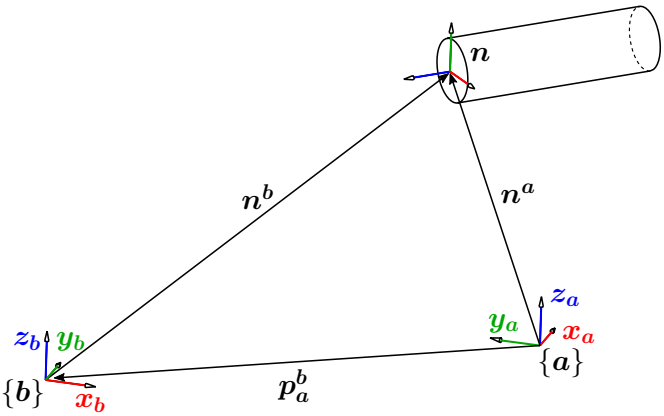


Figure 2.2.: Transformation from frame $\{\boldsymbol{a}\}$ to frame $\{\boldsymbol{b}\}$

$$\boldsymbol{R}(\boldsymbol{\epsilon}, \eta) = \begin{bmatrix} \eta^2 + \epsilon_x^2 - \epsilon_y^2 - \epsilon_z^2 & 2(\epsilon_x\epsilon_y - \eta\epsilon_z) & 2(\epsilon_x\epsilon_z + \eta\epsilon_y) \\ 2(\epsilon_x\epsilon_y + \eta\epsilon_z) & \eta^2 - \epsilon_x^2 + \epsilon_y^2 - \epsilon_z^2 & 2(\epsilon_y\epsilon_z - \eta\epsilon_x) \\ 2(\epsilon_x\epsilon_z - \eta\epsilon_y) & 2(\epsilon_y\epsilon_z + \eta\epsilon_x) & \eta^2 - \epsilon_x^2 - \epsilon_y^2 + \epsilon_z^2 \end{bmatrix}. \tag{2.7}$$

The quaternion is often given by (x, y, z, w) instead of $(\epsilon_x, \epsilon_y, \epsilon_z, \eta)$.

2.1.2. Homogeneous Transformation Matrices

Unlike a rotation matrix, a transformation matrix includes both a rotational and a translational part. Thus, transformation matrices can be used not only to rotate a vector between coordinate frames, but also translate them when there is a physical distance between the two frames. A transformation matrix is on the form

$$\boldsymbol{T}_{\boldsymbol{a}}^{\boldsymbol{b}} = \begin{bmatrix} \boldsymbol{R}_{\boldsymbol{a}}^{\boldsymbol{b}} & \boldsymbol{p}_{\boldsymbol{a}}^{\boldsymbol{b}} \\ \boldsymbol{0} & 1 \end{bmatrix},$$

where $\boldsymbol{R}$ is the rotation matrix, as in (2.3), and $\boldsymbol{p}$ is the translation between the frames, as in (2.1). The sub- and superscript denotes the frame of reference and the referred frame respectively. If the vector $\boldsymbol{n}$ is rewritten to its homogeneous representation

$$\widetilde{\boldsymbol{n}} = \begin{bmatrix} \boldsymbol{n} \\ 1 \end{bmatrix},$$

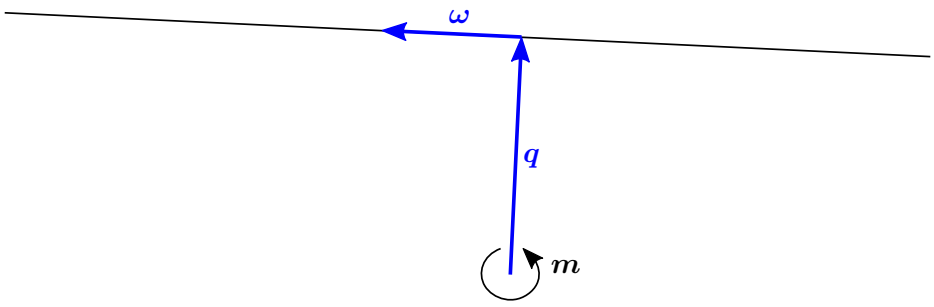


Figure 2.3.: Moment notation

it can be transformed from frame $\{\boldsymbol{a}\}$ to frame $\{\boldsymbol{b}\}$, given in Figure 2.2, by the homogeneous transformation matrix. That is,

$$\widetilde{\boldsymbol{n}}^{\boldsymbol{b}} = \boldsymbol{T}_{\boldsymbol{a}}^{\boldsymbol{b}} \widetilde{\boldsymbol{n}}^{\boldsymbol{a}}.$$

2.1.3. Screws

A screw is a useful tool in describing lines in 3D-space. A line in 3D has 4 degrees of freedom (DOF), which means that it can be fully described by four parameters. A usual description of a line has six parameters. These six parameters are given by two points $\boldsymbol{q}$ and $\boldsymbol{p}$, with their three Euclidean parameters each. Then, any point on the line can be defined as $\boldsymbol{r} = \boldsymbol{q} + \lambda(\boldsymbol{p} - \boldsymbol{q})$ where λ is a real scalar. If $\boldsymbol{p} - \boldsymbol{q}$ is rewritten to $\boldsymbol{\omega} = \boldsymbol{p} - \boldsymbol{q}$, then a line is described by a point $\boldsymbol{q}$ on the line and a direction vector $\boldsymbol{\omega}$.

To describe a line by a screw, as in Figure 2.3, we have to define the moment $\boldsymbol{m}$ of the line;

$$\boldsymbol{m} = \boldsymbol{p} \times \boldsymbol{\omega},$$

where $\boldsymbol{\omega}$ is a unit vector giving the direction of the line and $\boldsymbol{p}$ is a vector from the reference frame to an arbitrary point on the line. If the given parameters are the moment $\boldsymbol{m}$ and the direction vector $\boldsymbol{\omega}$, it follows that we can find the point $\boldsymbol{q}$ on the line which is closest to the reference frame. That is:

$$\boldsymbol{q} = \boldsymbol{\omega} \times \boldsymbol{m}.$$

Thus a line can be described by the direction vector $\boldsymbol{\omega}$ and the moment $\boldsymbol{m}$. This

is a six parameter description, with two conditions $|\boldsymbol{\omega}| = 1$ and $\boldsymbol{\omega} \times \boldsymbol{m} = 0$, which is in agreement with the fact that lines have 4 DOF.

A line described by $\boldsymbol{\omega}$ and $\boldsymbol{m}^a$ in reference to frame $\{\boldsymbol{a}\}$ can easily be transformed to frame $\{\boldsymbol{b}\}$ with the screw notation;

$$\boldsymbol{m}^{\boldsymbol{b}} = \boldsymbol{m}^{\boldsymbol{a}} + \boldsymbol{p}_{\boldsymbol{b}}^{\boldsymbol{a}} \times \boldsymbol{\omega},$$

where $\boldsymbol{p}_{\boldsymbol{b}}^{\boldsymbol{a}}$ is a vector from frame $\{\boldsymbol{b}\}$ to frame $\{\boldsymbol{a}\}$. A general screw is written in coordinate-free form as

$$\boldsymbol{s} = (\boldsymbol{u}, \boldsymbol{v})$$

and in coordinate form as

$$\mathcal{S} = \begin{Bmatrix} \boldsymbol{u} \\ \boldsymbol{v} \end{Bmatrix},$$

where $\boldsymbol{u}$ and $\boldsymbol{v}$ are 3-dimensional column vectors. Another common method for denoting a screw is

$$\mathcal{S} = \left[\begin{array}{c} \boldsymbol{u} \\ \hline \boldsymbol{v} \end{array}\right].$$

The important thing to remember with both of the denoting methods, is that the screw has special computation rules. A line in frame $\{\boldsymbol{a}\}$ can be written in screw form as

$$\mathcal{L}^a = \begin{Bmatrix} \boldsymbol{\omega} \\ \boldsymbol{m}^a \end{Bmatrix}.$$

The product of a scalar α and a screw $\mathcal{S}$ is component-wise, that is

$$\alpha\mathcal{S} = \begin{Bmatrix} \alpha\boldsymbol{u} \\ \alpha\boldsymbol{v} \end{Bmatrix},$$

the sum and difference of two screws are also component-wise;

$$\begin{Bmatrix} \boldsymbol{u}_0 \\ \boldsymbol{v}_0 \end{Bmatrix} + \begin{Bmatrix} \boldsymbol{u}_1 \\ \boldsymbol{v}_1 \end{Bmatrix} = \begin{Bmatrix} \boldsymbol{u}_0 + \boldsymbol{u}_1 \\ \boldsymbol{v}_0 + \boldsymbol{u}_1 \end{Bmatrix}$$

$$\begin{Bmatrix} \boldsymbol{u}_0 \\ \boldsymbol{v}_0 \end{Bmatrix} - \begin{Bmatrix} \boldsymbol{u}_1 \\ \boldsymbol{v}_1 \end{Bmatrix} = \begin{Bmatrix} \boldsymbol{u}_0 - \boldsymbol{u}_1 \\ \boldsymbol{v}_0 - \boldsymbol{u}_1 \end{Bmatrix}.$$

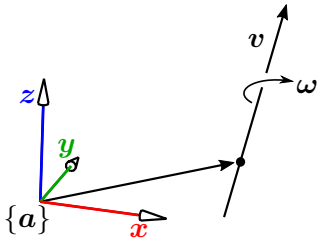


Figure 2.4.: Twist

The dot and cross product of two screws are given as

$$\begin{Bmatrix} \boldsymbol{u}_0 \\ \boldsymbol{v}_0 \end{Bmatrix} \cdot \begin{Bmatrix} \boldsymbol{u}_1 \\ \boldsymbol{v}_1 \end{Bmatrix} = \begin{Bmatrix} \boldsymbol{u}_0 \cdot \boldsymbol{u}_1 \\ \boldsymbol{u}_0 \cdot \boldsymbol{v}_1 + \boldsymbol{v}_0 \cdot \boldsymbol{u}_1 \end{Bmatrix}$$

$$\begin{Bmatrix} \boldsymbol{u}_0 \\ \boldsymbol{v}_0 \end{Bmatrix} \times \begin{Bmatrix} \boldsymbol{u}_1 \\ \boldsymbol{v}_1 \end{Bmatrix} = \begin{Bmatrix} \boldsymbol{u}_0 \times \boldsymbol{u}_1 \\ \boldsymbol{u}_0 \times \boldsymbol{v}_1 + \boldsymbol{v}_0 \times \boldsymbol{u}_1 \end{Bmatrix}.$$

2.1.4. Twist

A screw is a useful tool for representation of angular and linear velocities of rigid bodies, called a twist. A representation of a twist is given in Figure 2.4. As twists can be formulated as a screw, they will have the same computation rules as described above. A twist will trace out the helical field of a literal screw, where the ratio between the angular and the linear velocity would compare to the pitch of the screw.

A twist can be written as

$$\mathcal{T}_{\boldsymbol{a}} = \begin{Bmatrix} \boldsymbol{\omega} \\ \boldsymbol{v} \end{Bmatrix},$$

where $\boldsymbol{\omega}$ is the angular velocity and $\boldsymbol{v}$ is the linear velocity of the twist, relative to frame $\{\boldsymbol{a}\}$.

2.1.5. Wrench

A screw can also be used to represent force and torque applied to rigid bodies. This is called a wrench. A wrench can be written as

$$\mathcal{W}_{\boldsymbol{a}} = \begin{Bmatrix} \boldsymbol{F} \\ \boldsymbol{T} \end{Bmatrix},$$

where $\boldsymbol{F}$ is the x, y, z component of the force relative to frame $\{\boldsymbol{a}\}$ and $\boldsymbol{T}$ are equivalently the moments about x, y, z.

2.2. Robot Kinematics

An industrial robot is described by its degrees of freedom, which is equal to the number of joints on the robot. The types of joints are revolute or prismatic, that is rotational joints or translational joints. Most industrial robots have 6 DOF with all joints being revolute. In this chapter, we will present a brief, but essential introduction to robot kinematics. For more in-depth theory, see Siciliano et al. [33].

2.2.1. Direct Kinematics

The aim of direct kinematics is to compute the pose of the end-effector as a function of the joint variables. In robotics, a common convention for describing the direct kinematics is the Denavit-Hartenberg [8] (DH) convention. In the DH convention, a reference frame is placed in all joints. As described in [33], the position and orientation of a frame relative to the proceeding can be fully described by four parameters a_i, d_i, α_i and θ_i:

a_i : distance between O_i and $O_{i'}$,

d_i : coordinate of $O_{i'}$ along z_{i-1},

α_i : angle between axes z_{i-1} and z_i about axis x_i to be taken positive when rotation is made counter-clockwise,

θ_i : angle between axes x_{i-1} and x_i about axis z_{i-1} to be taken positive when

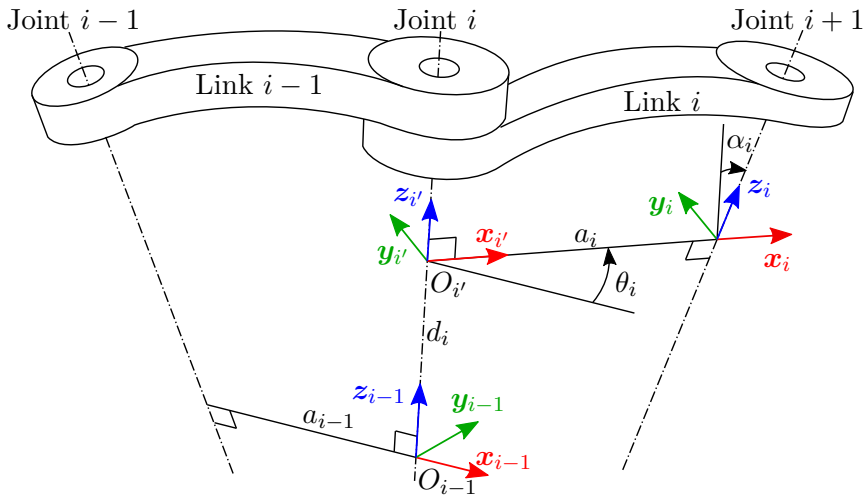


Figure 2.5.: The four parameters of the DH convention. [33]

rotation is made counter-clockwise.

An illustration of the parameters is given in Figure 2.5. By deriving all parameters for a robot, one is able to construct a link to link transformation matrix to get the final transformation matrix describing the relative position and rotation between the base frame and end-effector. The transformation matrices are all functions of the joint variables q_i. As mentioned earlier, most industrial robots the variable part of the transformation matrix is the rotation θ_i about each joint. The transformation matrix from Frame i to Frame $i-1$ is given as

$$\boldsymbol{T}_i^{i-1}(\theta_i) = \begin{bmatrix} c_{\theta_i} & -s_{\theta_i}c_{\alpha_i} & s_{\theta_i}s_{\alpha_i} & a_i c_{\theta_i} \\ s_{\theta_i} & c_{\theta_i}c_{\alpha_i} & -c_{\theta_i}s_{\alpha_i} & a_i s_{\theta_i} \\ 0 & s_{\alpha_i} & c_{\alpha_i} & d_i \\ 0 & 0 & 0 & 1 \end{bmatrix},$$

where by conventional typesetting in robotics, c_{θ_i} is equivalent to $\cos(\theta_i)$ and similarly s_{θ_i} is $\sin(\theta_i)$. To simplify writing even more, $c_1 = \cos(\theta_1)$ and $c_{12} = \cos(\theta_1 + \theta_2)$.

The resulting transformation matrix from base to end-effector is then

$$\boldsymbol{T}_6^0 = \boldsymbol{T}_1^0\boldsymbol{T}_2^1\boldsymbol{T}_3^2\boldsymbol{T}_4^3\boldsymbol{T}_5^4\boldsymbol{T}_6^5 = \begin{bmatrix} \boldsymbol{n}_6^0 & \boldsymbol{s}_6^0 & \boldsymbol{a}_6^0 & \boldsymbol{p}_6^0 \\ 0 & 0 & 0 & 1 \end{bmatrix},$$

where $\boldsymbol{p}_6^0$ is the position of the end-effector and $\boldsymbol{a}_6^0$, $\boldsymbol{s}_6^0$ and $\boldsymbol{n}_6^0$ are the unit approach, sliding and normal vectors of the end-effector respectively.

2.2.2. Inverse Kinematics

Inverse kinematics is the opposite of direct kinematics. In other words, the aim of inverse kinematics is to find a set of joint angles which gives the desired end-effector pose. This solution is usually not unique. Inverse kinematics is solved by either analytical methods, iterative solvers or a combination of both, where the result of the analytical method is used as an initial guess for the iterative solver.

For a 6 DOF manipulator, there are in general 16 possible solutions to the inverse problem if considering joints being able to rotate fully 2π radians. The different configurations are denoted by terms of the human arm. The robot shoulder could be facing towards or away, the elbow can be pointing up or down and the wrist can be either extended or contracted. To get a unique solution to the problem, one usually define the desired arm configuration in advance. Computation of solutions to the analytic problem require algebraic and geometric intuition, and will not be introduced here.

Geometrical Jacobian

While there, in general, is no linear relationship between the joint angles and the end effector position and orientation, a linear relationship is found between the angular joint velocities $\dot{\boldsymbol{q}}$ and the angular and linear velocities $\boldsymbol{\omega}_e$ and $\dot{\boldsymbol{p}}_e$ of the end-effector frame in Cartesian space. The relation between the two spaces is the *Jacobian* $\boldsymbol{J}$, where

$$\boldsymbol{\omega}_e(t) = \boldsymbol{J}_O\big(\boldsymbol{q}(t)\big)\dot{\boldsymbol{q}}(t)$$

and

$$\dot{\boldsymbol{p}}_e(t) = \boldsymbol{J}_P\big(\boldsymbol{q}(t)\big)\dot{\boldsymbol{q}}(t).$$

Within robotics, we usually omit (t) for readability, so remember that all joint angles and velocities, and Cartesian positions and velocities are dependent on

time. $\boldsymbol{J}_O$ and $\boldsymbol{J}_P$ are the orientation and position parts of the Jacobian $\boldsymbol{J}$;

$$\boldsymbol{J} = \begin{bmatrix} \boldsymbol{J}_O \\ \boldsymbol{J}_P \end{bmatrix}, \tag{2.8}$$

hence the relationship between the joint velocities and end-effector velocities can be written as

$$\boldsymbol{v} = \boldsymbol{J}(\boldsymbol{q})\dot{\boldsymbol{q}},$$

where

$$\boldsymbol{v} = \begin{bmatrix} \boldsymbol{\omega} \\ \dot{\boldsymbol{p}} \end{bmatrix}.$$

Geometrical Jacobian by Twists

The geometrical Jacobian can also be obtained by twists. A revolute joint will be a twist with zero pitch, that is, zero linear velocity. Thus, the twist about joint i is

$$\mathcal{T}_i = \begin{Bmatrix} \boldsymbol{z} \\ \boldsymbol{p} \times \boldsymbol{z} \end{Bmatrix},$$

where $\boldsymbol{z}$ is the unit vector of the z axis of frame i.

A prismatic joint on the other hand, will be a twist with infinite pitch, that is, zero angular velocity. The twist for a prismatic joint i is

$$\mathcal{T}_i = \begin{Bmatrix} \boldsymbol{0} \\ \boldsymbol{z} \end{Bmatrix}.$$

Then, the Jacobian becomes

$$\boldsymbol{J} = \sum_{i=1}^{n} \mathcal{T}_i,$$

in screw form, and

$$\boldsymbol{J} = \begin{bmatrix} \boldsymbol{\omega} \\ \boldsymbol{v} \end{bmatrix} = \begin{bmatrix} \boldsymbol{J}_O \\ \boldsymbol{J}_P \end{bmatrix},$$

in coordinate form, which is the same as the geometrical Jacobian in (2.8).

2.3. Resolved Motion Rate Control

As shown in the last section, we are able to calculate the angular and linear velocities of the end-effector through the Jacobian from the joint velocities by

$$\boldsymbol{v} = \boldsymbol{J}(\boldsymbol{q})\dot{\boldsymbol{q}}.$$

If instead, we want to find the necessary joint velocities for imposed angular and linear velocities, we get

$$\dot{\boldsymbol{q}} = \boldsymbol{J}(\boldsymbol{q})^{-1}\boldsymbol{v}. \tag{2.9}$$

There is one issue; the Jacobian is usually not invertible, as it is often not square. A solution to this is to take the *pseudoinverse* ($\boldsymbol{J}^{+}$) of the Jacobian instead. That is,

$$\boldsymbol{J}^{+} = \boldsymbol{J}^{T}(\boldsymbol{J}\boldsymbol{J}^{T})^{-1},$$

hence (2.9) becomes

$$\dot{\boldsymbol{q}} = \boldsymbol{J}(\boldsymbol{q})^{+}\boldsymbol{v}. \tag{2.10}$$

Resolved motion rate control is one way of solving the Cartesian end-effector problem, which uses (2.10) to compute incremental joint velocity commands by incremental changes in the desired end-effector velocities. The discrete counterpart to (2.10) is used in the following way in resolved motion rate control. The desired joint velocities are computed by

$$\dot{\boldsymbol{q}}_k = \boldsymbol{J}(\boldsymbol{q}_k)^{+}\boldsymbol{v}_k,$$

before the next joint configuration is calculated by

$$\boldsymbol{q}_{k+1} = \boldsymbol{q}_k + \dot{\boldsymbol{q}}_k\Delta t$$

Chapter 3.

Software Tools for Robotics Programming

This chapter will present basics of programming tools used through the work in this Master's thesis.

3.1. Industrial Robot Programming Status Quo

Common programming methods for industrial robots are tedious, and requires a great amount of skills. Co-bots are arriving fast, and they are meant to be easy to setup for anybody. Still, for bigger industrial applications and robot systems requiring communication with several sensor inputs, you would need a skilled robot programmer.

The most common methods are online programming either by the use of a teach pendant or manual guidance. Those methods are often associated with point-to-point motions, as the robot programmer will guide the robot to desired setpoints. The setpoints are recorded and played back with either a linear, circular, or spline motion between them at execution. Those methods are in-flexible and usually time consuming.

Online programming is most common, but offline programming is becoming more and more used by the industry. In offline programming, the robot program is created independently of the actual robot. Usually alongside a graphical 3D

visualization. The main advantage of offline programming compared to online programming is that the robot can be used to other production tasks parallel to programming. The offline programming tools does also often contain methods of helping the programmer, by for instance collision detection. The greatest weakness of both online and offline programming is that they usually can not cope with unforeseen events during execution. This is where constraint based programming really comes in handy.

3.2. ROS

"The Robot Operating System (ROS) is a flexible framework for writing robot software. It is a collection of tools, libraries, and conventions that aim to simplify the task of creating complex and robust robot behavior across a wide variety of robotic platforms" [32].

3.2.1. Master

The ROS master is in charge of detect which nodes, topics and services who needs peer-to-peer interconnection. The master is the first component to be started during a ROS execution, where it sets up an URI for TCP communication. This URI is unique, and thus can be accessed from other computers. This means that a ROS system can be set up and communicate over several hardware systems.

3.2.2. Nodes and Topics

The core advantage of ROS, is the possibility of dividing the code into small sub modules called nodes. Usually one node is made for every sub-task of the robot system, having only a single purpose. For instance, one node can have the purpose to drive the motors of the wheels on a mobile robot, while another is reading values of the velocity sensors of the wheels. For increased flexibility, one can choose the programming language most suited for the task for each node, mainly Python or C++.

The nodes are connected through messages published on and subscribed from top-

ics. Messages published to and subscribed from topics are continuous unidirectional streams. That is, all information is published continuously to the topics and both the publisher and the subscriber nodes are unaware about other nodes connected to the same topic. The sensor node, could for instance publish a message containing decimal points (`Float64`) on a `/velocity_sensor` topic. A controller node for the robot could then subscribe to the messages on the `/velocity_sensor` topic while at the same time being publishing data to another topic in which the motor driver node subscribes to. The ROS library is extensive, with many standard message types, such as `Bool`, `Float64`, `String`, etc., but also more complicated ones, such as `Pose` and `Wrench`. The `Pose` message is basically a message containing 7 `Float64`. 3 for the position (x, y, z) and 4 for the orientation as a quaternion (x, y, z, w). One is also able to define custom message types if necessary.

3.2.3. Services and Clients

As topics are anonymous continuous streams, and the nodes are unaware of each other, there is a need for direct single communication between nodes. This is achieved in ROS through services and clients. A service node can request an action from a client node, which processes this request and returns a response to the service node. Thus, only a single message is sent from the service node and likewise only a single message is returned. The messages could be of any type, and contain many different variables.

3.2.4. Parameters

The ROS master has a server containing parameters. That is, variables that can be changed and retrieved by any node in the system. The parameters are useful for setting variable values that many nodes need but does not need to change continuously. Preferably only a couple of times during run-time.

3.2.5. URDF

In ROS, the Unified Robot Description Format (URDF) is the standard for defining and describing robots. It is an XML document modelling the kinematics and

dynamics of any robot type. The URDF is ROS' alternative to the DH convention described in chapter 2. In addition, it also contains 3D models for visualization purposes. URDFs are easily defined in so-called xacro-files, where we define the robot by its links, which can contain meshed geometries for visualization, and its joints. Joints are defined by their type (prismatic, revolute or continous), a parent and a child link, and the relative position and rotation between the links. A limit for joint positions, velocities, and torque can also be defined. The description for the first joint of the KUKA KR6 is shown in Listing 3.1.

Listing 3.1: First Joint of KUKA KR6

```
<joint name="${prefix}joint_a1" type="revolute">
    <origin xyz="0 0 0.400" rpy="0 0 0"/>
    <parent link="${prefix}base_link"/>
    <child link="${prefix}link_1"/>
    <axis xyz="0 0 -1"/>
    <limit effort="0" lower="${-DEG2RAD*170}" upper="${DEG2RAD*170}"
        velocity="${DEG2RAD*360}"/>
</joint>
```

3.2.6. Rviz

Rviz is ROS' visualization tool. It is the GUI of choice when debugging 3D-data from ROS. Rviz can visualize anything from coordinate frames to URDF models to 3D-markers of basic or meshed shapes. The Rviz environment is configured by selecting the parts of the robot you want to show, which frames of the robot to be visualized or by subscribing to topics that contains 3D-data.

3.2.7. ROS Control

While ROS is used to calculate desired setpoints for a robot, ROS Control is the code that takes those setpoints and translate them into the actual control of the robot. Hence, ROS Control is the bridge between software and hardware. A ROS controller can be written to communicate and control any robot that has an open communication platform to computers. There are many convenient functions build in to ROS Control. One example is the `joint_limits_interface` which lets you ensure that the robot does not receive commands that are physically impossible to achieve.

3.3. eTaSL

The creator of eTaSL describes it the following way:

"eTaSL is a task **specification** language for **reactive** control of robot systems. It is a language that allows you to describe how your robotic system has to move and interact with sensors. This description is based on a **constraint-based** methodology. Everything is specified as an optimization problem subject to constraints"– E. Aertbeliën [2].

eTaSL became open source in 2018, and is as of now the most complete system for constraint based robot control. Hence, it has been chosen as the framework in this project. The following subsections explains all necessary parts of eTaSL used in this project.

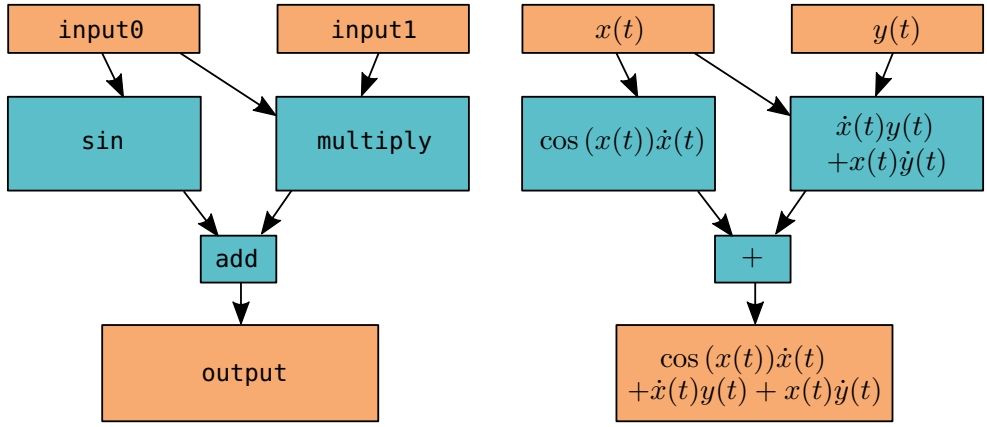
3.3.1. Expression Graphs

Expression graphs are treelike structures which defines sets of single mathematical operations, deriving an output expression from an arbitrarily number of input values. These inputs can be either constant or variable relative to time. The formulation of expression graphs are used extensively in eTaSL, firstly because it makes numerical calculations easy, but mainly because forward accumulated automatic differentiation falls almost directly out of the expression trees.

Automatic differentiation is a technique for numerical differentiation of a function, where the basic arithmetic operations in computer calculation are used together with the chain rule. This results in calculations of derivatives of arbitrary order to be computed efficiently and without loss of precision.

A simple example of an expression graph and its derivative is shown in Figure 3.1. From the figure we can see an example of a expression graph where we want to sum the value of sine of `input0` and the product of `input0` and `input1`. If `input0` is some function $x(t)$ dependent on time and `input1` is some other function $y(t)$ dependent on time, the `output` in Figure 3.1a is

$$\sin\left(x(t)\right) + x(t)y(t).$$



(a) Expression **(b)** Derivative of expression

Figure 3.1.: Expression graphs and their derivatives

Each block will have its own derivation rule, and are combined with the other derivatives following the chain rule. Hence we get the output in Figure 3.1b as

$$\cos(x(t))\dot{x}(t) + \dot{x}(t)y(t) + x(t)\dot{y}(t).$$

eTaSL supports six different expression types. They are the `C++` types `double`, `KDL::Vector`, `KDL::Rotation`, `KDL::Frame`, `KDL::Twist`, and `KDL::Wrench`. Their corresponding derivatives are `double`, `KDL::Vector`, `KDL::Vector`, `KDL::Twist`, `KDL::Twist`, and `KDL::Wrench` respectively. KDL stands for Kinematics and Dynamics Library [35], and is a `C++` library used by eTaSL.

3.3.2. Joint and Feature Variables

There are three different types of variables, two that the eTaSL solver will optimize during a solver step and *time*. The two solver variables are *robot joint variables* and *feature variables*. The controller has to provide relative velocity steps for all the joints of the robot in each cycle. Each solved joint variable will restrain one degree of freedom for the robot motion. Feature variables is auxiliary additional variables, that can be used to free up degrees of freedom. One example of the use of feature variables are a case where we want a laser to track a path, but the actual distance between the laser origin and the laser point on the surface is free

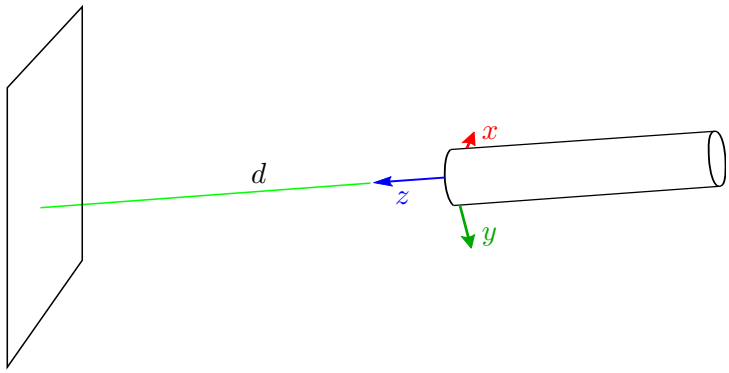


Figure 3.2.: Laser with feature variable

to vary.

Listing 3.2: Feature variable

```
d = Variable{
        context = ctx,
        name    = "d",
        vartype = "feature"
    }
```

We can define the distance between the laser origin and the laser point on the surface as shown in Figure 3.2 as a feature variable `d` as given in Listing 3.2. The laser point can then be modeled as

```
laserpoint = laserorigin*translate_z(d),
```

where `laserpoint` and `laserorigin` are frames and the multiplication with the `translate_z(d)` is equivalent with a transformation of the `laserorigin` frame by a transformation matrix with the rotation part equal to the identity matrix, and the translation equal to $[0.0, 0.0, d]^T$. The new frame `laserpoint` can then be used together with constraints to define the path tracking.

3.3.3. Constraints

A set of constraints can be applied on the robot motion. We can call this a task. The three variables *time*, *joint*, and *feature*, described above is solved as a first

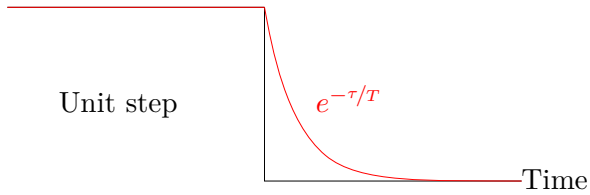


Figure 3.3.: Exponential behavior of a first order system

order system. In a first order system, a proportional control constant K is used. This means that the system will evolve towards its goal in an exponential manner, as shown in Figure 3.3.

Following Aertbeliën [1] we will see that in the first order task function, the feature variables $\boldsymbol{\chi}_f(t)$ and the robot joint variables $\boldsymbol{q}(t)$ can be solved for simultaneously as a combined state $\widetilde{\boldsymbol{q}}(t)$.

With the combined state $\widetilde{\boldsymbol{q}}$ and a control constant K, the first order system can be written as:

$$\frac{\mathrm{d}}{\mathrm{d}t}e(\widetilde{\boldsymbol{q}}) = -Ke(\widetilde{\boldsymbol{q}})$$

By expanding this to its partial derivatives

$$\frac{\partial}{\partial t}e(\widetilde{\boldsymbol{q}}) + \frac{\partial}{\partial \widetilde{\boldsymbol{q}}}e(\widetilde{\boldsymbol{q}})\dot{\widetilde{\boldsymbol{q}}} = -Ke(\widetilde{\boldsymbol{q}}),$$

we can see that the Jacobian falls directly out. That is,

$$\boldsymbol{J}(\widetilde{\boldsymbol{q}})\dot{\widetilde{\boldsymbol{q}}} = -Ke(\widetilde{\boldsymbol{q}}) - \frac{\partial}{\partial t}e(\widetilde{\boldsymbol{q}}), \tag{3.1}$$

which is then used to formulate a hard constraint in an optimization problem for the task. A hard constraint is a constraint that can not be violated. We might want to have a set of constraints that constrain more than all 6 DOFs. In that case, some of the constraints has to have the possibility of being violated temporary. Such a constraint is called a soft constraint, where a slack variable ϵ is introduced:

$$\frac{\partial}{\partial t}e(\widetilde{\boldsymbol{q}}) + \frac{\partial}{\partial \widetilde{\boldsymbol{q}}}e(\widetilde{\boldsymbol{q}})\dot{\widetilde{\boldsymbol{q}}} = -Ke(\widetilde{\boldsymbol{q}}) + \epsilon$$

$$\boldsymbol{J}(\widetilde{\boldsymbol{q}})\dot{\widetilde{\boldsymbol{q}}} = -Ke(\widetilde{\boldsymbol{q}}) - \frac{\partial}{\partial t}e(\widetilde{\boldsymbol{q}}) + \epsilon.$$

A weight w is introduced to the slack variable

$$w\epsilon^2,$$

which decides how important the soft constraint is.

A constraint is formulated in eTaSL as shown in Listing 3.3, where `K` is the control constant K, `weight` is the weight w of the constraint and `priority` determines whether the slack variable is introduced or not. `priority = 1` means a hard constraint, that is, no slack variable and `priority = 2` means a soft constraint, that is, the slack variable is introduced. A constraint can be an inequality constraint, which means that the constraint converges to a bound region instead of a single target. To ensure convergence to an inequality region, the constraint has to be a hard constraint.

Listing 3.3: Constraint formulation

```
Constraint{
    context  = ctx,
    name     = "name"[optional],
    expr     = (expression),
    target   = (expression/scalar)[optional] (if other than 0),
    K        = (scalar)[optional],
    weight   = (scalar)[optional],
    priority = (1 or 2)
}
```

If we want to make a constraint which makes a laser track a path at some distance `d` as described earlier in section 3.3, the constraint could be defined as shown in Listing 3.4, where `tgt` can be a vector input from a ROS publisher node defining the path. This constraint makes the origin of the laser point coincide with the target point, but as the laser point is a function of a feature variable, the distance between the laser pointer and the laser point is free to vary. The key feature of the feature variable in this example, is that as the distance is free to vary, the robot is able to choose its own distance to the laser point, thus having the possibility to avoid singularities.

Listing 3.4: Laser constraint

```
Constraint{
    context  = ctx,
    name     = "follow_path",
    expr     = tgt - origin(laserpoint),
    target   = vector(0.0, 0.0, 0.0),
    K        = 4,
    priority = 2
}
```

3.3.4. Monitors

eTaSL includes the possibility of monitoring expressions to be able to make actions whenever an event has happened. For instance, one can monitor the expression of a constraint, and find out when the the value of the expression is within some threshold from the target value. This can then be used to switch to the next task or activation and deactivation of groups of constraints.

Listing 3.5: Monitor formulation

```
Monitor{
    context     = ctx,
    name        = "name",
    expr        = (expression),
    lower       = (scalar)[optional],
    upper       = (scalar)[optional],
    actionname  = "exit",
    argument    = (string)[optional]
}
```

Monitors are defined by giving an expression to monitor and a lower or an upper limit or both, which defines when the monitor is triggered. They also take an action name which defines how the controller behaves upon trigger. The schematics of how a monitor is formulated is shown in Listing 3.5. A monitor is edge triggered, which means that it is activated whenever the value of the expression passes the upper or the lower bound, as illustrated in Figure 3.4. Hence, a monitor will never trigger if the bound is set higher than the initial value of the expression.

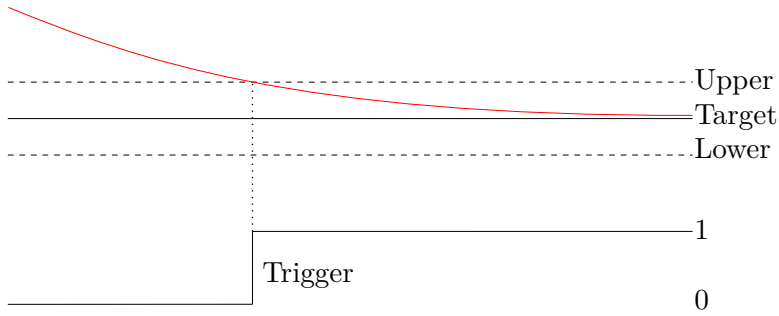


Figure 3.4.: Triggering of monitors

3.3.5. QP-solver

qpOASES is a C++ based software for solving a quadratic programming (QP) problem. eTaSL uses qpOASES to solve the constraint problem. The algorithm of qpOASES will try to

$$\begin{aligned} \text{minimize} \quad & \frac{1}{2}\dot{\tilde{\boldsymbol{q}}}^T H \dot{\tilde{\boldsymbol{q}}} \\ \text{subjected to} \quad & \text{lb} \leq \dot{\tilde{\boldsymbol{q}}} \leq \text{ub} \\ & \text{lbJ} \leq \boldsymbol{J}\dot{\tilde{\boldsymbol{q}}} \leq \text{ubJ}. \end{aligned}$$

H is the Hessian which is an $n \times n$ square matrix where n is the number of joints of the robot plus the number of soft constraints. The Hessian is a $\mathbf{0}$ matrix filled with the defined regularization factor on the diagonal elements for the elements of the joints and the regularization factor plus the `weight` on the rest of the diagonal elements. The regularization factor is a small value, typically 0.001 set to discourage the complexity of the model. For a robot with 6 joints and 3 soft

constraints with `weight = 1.5`, the Hessian would look like this:

$$\begin{bmatrix} 0.001 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0.001 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0.001 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0.001 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0.001 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0.001 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1.501 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1.501 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1.501 \end{bmatrix}$$

lb and ub are the lower and upper bound for the box constraints sat on the joints. That is, the min and max angles of each joint, and $-\infty$ and ∞ for the soft constraints. In eTaSL $\infty = 1E20$. $\boldsymbol{J}$ is the Jacobian of the left hand side of (3.1) with the derivatives of all active constraints. lbJ and ubJ are the intermediate lower and upper bounds, which is calculated by the right hand side of (3.1).

3.3.6. URDF Parser

One core advantage of eTaSL, is that the solver does not treat any trajectory or robot joint differently, thus a task will be solved independently of the kinematic chain of the robot. eTaSL has a built-in URDF-reader that generates an expression graph for the robot. The geometrical Jacobian of the robot is calculated from this expression graph by the joint twists. The Jacobian becomes a $6 \times n$ matrix, with each column describing the resulting end-effector linear and angular velocity of a unit change of the corresponding joint velocity.

We showed in subsection 3.3.3 that the Jacobian falls directly out of the equation of the first order system of the applied constraints. This means that the closed loop inverse kinematics of the robot is never directly computed. A result of that, is that if any of the joints does not have any constraint constraining it, it will not be considered by the solver. As an example, the orientation of a symmetrical gripper around the symmetry axis is not important, and can have no constraints defining its direction. If so, the orientation of that joint will never be calculated. An interesting result of this, is that the corresponding singularity problem will be

avoided.

3.4. SMACH

SMACH is a Python library for easy implementation of finite-state machines (FSM) with ROS. FSMs are abstract machines constructed of a finite number of *states*. The machine can only be in a single state at any instant, and are executed until a logical condition triggers a *transition* to another state. SMACH can be used to model complex robot behaviour at task-level. Each state in SMACH are classes with a `__init__`-function where possible outcomes of the state and potential member variables are declared. Then, an `execute`-function executes the state task. This can be anything from switching of controllers to a call of a service requests to trigger a publisher node. The transition to the next state can be triggered by conditional `if`-sentences, service responses, message arrivals, etc.

3.5. KUKA Robot Sensor Interface

KUKA has two controller schemes, known as KUKA Robot Language (KRL) and KUKA Robot Sensor Interface (RSI). KRL is the most common, and is used by setting end-frames for desired goals, either by position and orientation of the end-effector frame by (X, Y, Z, A, B, C) or by joint angles ($A1$, $A2$, $A3$, $A4$, $A5$, $A6$), and whether the trajectory should be linear, circular or splines. The intermediate trajectory is set by the robot controller itself. RSI is build on top of KRL to allow for small position corrections of the programmed paths due to sensor data. Connected to an external computer monitoring sensor data, these corrections can be sent to the controller as commands in the XML format. An implementation for RSI commands through ROS is available on github in the `kuka_experimental` repository [12], which can be used to fully control a KUKA robot by incremental joint position steps, where the underlying KRL commands are not used.

Sensor based control needs to react fast to events to execute properly. Today, sensor based robot systems are usually calculating paths and corrections at 250 to 1000 Hz. The fastest operating mode of RSI is at 250 Hz, that is, a new command is executed every 4 ms.

Chapter 4.

Constraint-Specification from CAD-Assemblies

This chapter will give an overview of the possibilities CAD software gives, particularly SolidWorks [7], in defining geometrical constraints. The constraints can then be used to define the needed expressions in eTaSL to solve the task, as we give the general mathematics behind the constraints.

4.1. Mates in SolidWorks

SW is one of many available 3D modeling software. As a product is designed in such software, designers are able to design single parts which are later given an interrelation in an assembly. Such relations are generally called geometrical constraints, while SW is calling them *mates*. The reason for choosing SW for this application is that SW gives access to an API in which the functionality of the software can be exploited in standalone code.

There are five basic mate types in SW that are defined by up to eight parameters. The output of the Mate API call is, as described at the SW reference page [11], the following array of doubles:

[`pointX`, `pointY`, `pointZ`, `vectorI`, `vectorJ`, `vectorK`, `radius1`, `radius2`]

where

`pointX`: the x component of the location vector of this mate entity in the assembly model space

`pointY`: the y component of the location vector of this mate entity in the assembly model space

`pointZ`: the z component of the location vector of this mate entity in the assembly model space

`vectorI`: the i component of the assembly mate direction vector

`vectorJ`: the j component of the assembly mate direction vector

`vectorK`: the k component of the assembly mate direction vector

`radius1`: the value for the first radius

`radius2`: the value for the second radius

The mate entities defined above can be used to describe the mate types given in Table 4.1. All coordinate information is given in terms of the assembly coordinate system where the mate resides.

The standard mate types of SW are given in Table 4.2, together with their return value. A graphical description of the possible mate relations are shown in Figure 4.1.

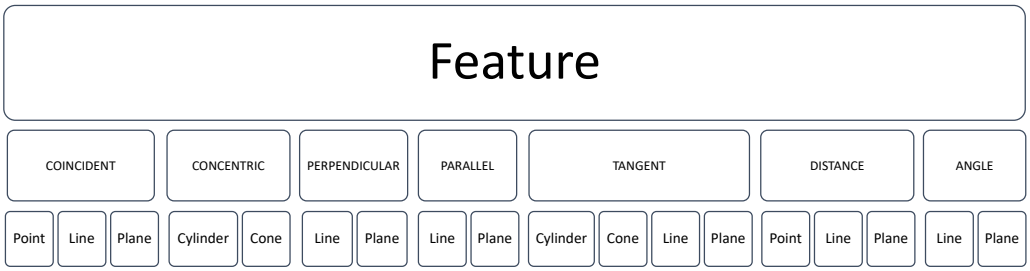


Figure 4.1.: SW mate inheritance

Table 4.1.: List of mate entity types

Mate Type	Returned
`swMatePoint(0)`	[`pointX`, `pointY`, `pointZ`]
`swMateLine(1)`	[`pointX`, `pointY`, `pointZ`, `vectorI`, `vectorJ`, `vectorK`] where the point is a point on the line and the vector represents the line direction.
`swMatePlane(3)`	[`pointX`, `pointY`, `pointZ`, `vectorI`, `vectorJ`, `vectorK`] where the point is a point on the plane and the vector represents the plane normal.
`swMateCylinder(4)`	[`pointX`, `pointY`, `pointZ`, `vectorI`, `vectorJ`, `vectorK`, `radius1`] where the point is a point on the cylinder axis, the vector represents the cylinder axis direction and radius the radius of the cylinder.
`swMateCone(5)`	[`pointX`, `pointY`, `pointZ`, `vectorI`, `vectorJ`, `vectorK`, `radius1`, `radius2`] where the point is a point on the cone axis, the vector represents the cone axis direction and `radius1` and `radius2` the first and second radius of the cone.
`swMateSphere(6)`	[`pointX`, `pointY`, `pointZ`, `radius1`] where the point is the center point of the sphere, and `radius1` the radius of the sphere.

4.2. Geometrical Constraints

Geometrical constraints define the relative positioning and orientation between rigid bodies. Such can be that the edge of one body is coincident with the face of the other body or that a hole and cylinder is concentric. This section describes these kind of constraints.

4.2.1. Degrees of Freedom

The number of DOF defines how an object can be translated and rotated in the 3D space. A 3D-object in space will have 6 DOF. That is, translation in x-, y- and z-direction and rotation about i, j and k. A point in space is defined by only

Table 4.2.: List of standard mate types

Mate Type	CODE
swMateCOINCIDENT	0
swMateCONCENTRIC	1
swMatePERPENDICULAR	2
swMatePARALLEL	3
swMateTANGENT	4
swMateDISTANCE	5
swMateANGLE	6

3 DOF since any rotation of a point gives the same point. A line is defined by 4 DOF. Here only two translations and two rotations would change the line. A plane is defined by 3 DOF, as only one translation and two rotations change the plane.

In CAD software, mates or constraints can be defined between parts to define how they behave or are positioned relative to each other. Mates can be defined on points, lines, and planes, and each mate would lock some DOF of a part. In SW, the first part selected in an assembly is defining the global reference coordinate system for the entire assembly. All other parts in the assembly are given coordinates relative to this. That means that the first part will have all 6 DOF defined. When a mate is defined between that part and the next, the mate will constrain some DOF of the new part dependent on the type of mate. The mates and their constraining DOF is described in subsection 4.2.3.

4.2.2. Kinematic Pairs

A kinematic pair is a relation between two rigid bodies imposing constraints on their relative movements [19]. Kinematic pairs are significant in understanding how different geometrical constraints from CAD, limit the motion between parts. Kinematic pairs are divided into two sub-classes called lower and higher pairs. A lower pair is all kinematic pair in which both rigid bodies has a surface contact with the other body. In higher pairs, one of the bodies edges gets constrained to the surface of the other body. Cases of lower pairs are described in [19] as:

Revolute pair: requires a line in the moving body to remain co-linear with a line in the fixed body, and a plane perpendicular to this line in the moving body maintain contact with a similar perpendicular plane in the fixed body. This imposes five constraints on the relative movement of the links, which therefore has 1 independent DOF.

Prismatic joint: requires that a line in the moving body remain co-linear with a line in the fixed body, and a plane parallel to this line in the moving body maintain contact with a similar parallel plane in the fixed body. This imposes five constraints on the relative movement of the links, which therefore has 1 independent DOF.

Screw pair: requires cut threads in two links, so that there is a turning as well as sliding motion between them. This joint has 1 independent DOF.

Cylindrical joint: requires that a line in the moving body remain co-linear with a line in the fixed body. It is a combination of a revolute joint and a sliding joint. This joint has 2 independent DOF.

Spherical joint: or ball and socket joint requires that a point in the moving body remain stationary in the fixed body. This joint has 3 independent DOF, corresponding to rotations around orthogonal axes.

Planar joint: requires that a plane in the moving body maintain contact with a plane in the fixed body. This joint has 3 independent DOF. The moving plane can slide in two dimensions along the fixed plane, and it can rotate on an axis normal to the fixed plane.

4.2.3. The Mathematics of SW Constraints

Following below is a description of all mate types in SW, and the mate entities that can be defined within each mate. The tables describe two features, with the mate entity that is defined on that feature, how the dependency is between the features and how many DOF that are constrained on feature 2 after the mate is defined. Feature 1 is considered fixed to the reference world coordinate system. All direction vectors are described by unit vectors, thus simplifying some of the equations.

Distance

A distance mate can be assigned between points, lines and planes. The dependencies of these are shown in Table 4.3.

Table 4.3.: Distance mates

Feat. 1	**Feat. 2**	**Dependency**	**Constrained DOF**
Point	Point	The point is at a sphere with origin in the other point and a radius of the defined distance (Spherical joint)	3 DOF
Point	Line	The line is tangential to a sphere with origin in the point and a radius of the defined distance (Spherical joint)	3 DOF
Point	Plane	Makes the plane tangential to a sphere with origin in the point and a radius of the defined distance (Spherical joint)	3 DOF
Line	Line	The defined distance is the shortest distance between two lines that cannot intersect (Cylindrical joint)	4 DOF
Line	Plane	The line and plane are parallel at the defined distance	2 DOF
Plane	Plane	The planes are parallel at the defined distance	3 DOF

The distance d between two points $\boldsymbol{p}_a$ and $\boldsymbol{p}_b$ are

$$d = ||\boldsymbol{p}_a - \boldsymbol{p}_b||. \tag{4.1}$$

Given a point $\boldsymbol{p}$ and a line $(\boldsymbol{q}, \boldsymbol{\omega})$, the distance d between the point and the line is

$$d = ||(\boldsymbol{p} - \boldsymbol{q}) - \boldsymbol{\omega}(\boldsymbol{\omega} \cdot (\boldsymbol{p} - \boldsymbol{q}))||. \tag{4.2}$$

The shortest distance d between a point $\boldsymbol{p}$ and a plane $(\boldsymbol{q}, \boldsymbol{n})$ is

$$d = |(\boldsymbol{p} - \boldsymbol{q}) \cdot \boldsymbol{n}| \tag{4.3}$$

The shortest distance d between two lines $(\boldsymbol{q}_a, \boldsymbol{\omega}_a)$ and $(\boldsymbol{q}_b, \boldsymbol{\omega}_b)$ can be described as

$$d = \begin{cases} ||(\boldsymbol{q}_a - \boldsymbol{q}_b) - \boldsymbol{\omega}_b(\boldsymbol{\omega}_b \cdot (\boldsymbol{q}_a - \boldsymbol{q}_b))||, & \text{if } ||\boldsymbol{\omega}_a \times \boldsymbol{\omega}_b|| = 0 \\ \dfrac{|(\boldsymbol{q}_a - \boldsymbol{q}_b) \cdot \boldsymbol{\omega}_a \times \boldsymbol{\omega}_b|}{||\boldsymbol{\omega}_a \times \boldsymbol{\omega}_b||}, & \text{otherwise} \end{cases} \tag{4.4}$$

The distance d between a line $(\boldsymbol{q}_l, \boldsymbol{\omega})$ and a plane $(\boldsymbol{q}_p, \boldsymbol{n})$ is

$$d = \begin{cases} |(\boldsymbol{q}_l - \boldsymbol{q}_p) \cdot \boldsymbol{n}|, & \text{if } \boldsymbol{\omega} \cdot \boldsymbol{n} = 0 \\ 0, & \text{otherwise} \end{cases} \tag{4.5}$$

The distance d between two planes $(\boldsymbol{q}_a, \boldsymbol{n}_a)$ and $(\boldsymbol{q}_b, \boldsymbol{n}_b)$ is

$$d = \begin{cases} |(\boldsymbol{q}_a - \boldsymbol{q}_b) \cdot \boldsymbol{n}_b|, & \text{if } ||\boldsymbol{n}_a \times \boldsymbol{n}_b|| = 0 \\ 0, & \text{otherwise} \end{cases} \tag{4.6}$$

Angle

An angle mate can be assigned between lines and planes. The dependencies of these as shown in Table 4.4. The function `atan2` is used for calculating angles between two vectors. This is a function which returns the angle in the interval $[-\pi, \pi]$. `atan2` is defined as

$$\texttt{atan2}(y, x) = \begin{cases} \arctan\left(\dfrac{y}{x}\right), & \text{if } x > 0 \\ \arctan\left(\dfrac{y}{x}\right) + \pi, & \text{if } x < 0 \text{ and } y \geq 0 \\ \arctan\left(\dfrac{y}{x}\right) - \pi, & \text{if } x < 0 \text{ and } y < 0 \\ +\dfrac{\pi}{2}, & \text{if } x = 0 \text{ and } y > 0 \\ -\dfrac{\pi}{2}, & \text{if } x = 0 \text{ and } y > 0 \\ \text{undefined}, & \text{if } x = 0 \text{ and } y = 0 \end{cases}$$

Table 4.4.: Angle mates

Feat. 1	**Feat. 2**	**Dependency**	**Constrained DOF**
Line	Line	The defined angle is the angle between the two infinite lines	1 DOF
Line	Plane	The defined angle is the angle in the intersection between the infinite line and the infinite plane	1 DOF
Plane	Plane	The defined angle is the angle between the two intersecting infinite planes	1 DOF

The angle θ between two lines $(\boldsymbol{q}_a, \boldsymbol{\omega}_a)$ and $(\boldsymbol{q}_b, \boldsymbol{\omega}_b)$ is

$$\theta = \texttt{atan2}\left(||\boldsymbol{\omega}_a \times \boldsymbol{\omega}_b||, \boldsymbol{\omega}_a \cdot \boldsymbol{\omega}_b\right), \tag{4.7}$$

while the angle θ between a line $(\boldsymbol{q}_l, \boldsymbol{\omega})$ and a plane $(\boldsymbol{q}_p, \boldsymbol{n})$ is

$$\theta = \frac{\pi}{2} - |\texttt{atan2}\left(||\boldsymbol{\omega} \times \boldsymbol{n}||, \boldsymbol{\omega} \cdot \boldsymbol{n}\right)|. \tag{4.8}$$

The angle θ between two planes $(\boldsymbol{q}_a, \boldsymbol{n}_a)$ and $(\boldsymbol{q}_b, \boldsymbol{n}_b)$ is

$$\theta = \texttt{atan2}\left(||\boldsymbol{n}_a \times \boldsymbol{n}_b||, \boldsymbol{n}_a \cdot \boldsymbol{n}_b\right). \tag{4.9}$$

Coincident

A coincident mate can be assigned between points, lines, and planes on the assembled parts. The dependencies of these are shown in Table 4.5.

Two points $\boldsymbol{p}_a$ and $\boldsymbol{p}_b$ are coincident if the distance between them is 0. That is, if the distance d in (4.1) is equal to 0. Namely, if

$$||\boldsymbol{p}_a - \boldsymbol{p}_b|| = 0.$$

A point $\boldsymbol{p}$ is coincident with a line $(\boldsymbol{q}, \boldsymbol{\omega})$ if the distance between them are 0. That is, if the distance d in (4.2) is equal to 0. Namely, if

$$||(\boldsymbol{p} - \boldsymbol{q}) - \boldsymbol{\omega}(\boldsymbol{\omega} \cdot (\boldsymbol{p} - \boldsymbol{q}))|| = 0.$$

Table 4.5.: Coincident mates

Feat. 1	**Feat. 2**	**Dependency**	**Constrained DOF**
Point	Point	Locks the points together	3 DOF
Point	Line	Locks the point on the infinite line	2 DOF
Point	Plane	Locks the point on the infinite plane	1 DOF
Line	Line	Locks the lines to the same infinite line	4 DOF
Line	Plane	Locks the line to the infinite plane	2 DOF
Plane	Plane	Locks the planes to the same infinite plane	3 DOF

A point $\boldsymbol{p}$ is coincident with a plane $(\boldsymbol{q}, \boldsymbol{n})$ if the distance d in (4.3) is equal to 0. Namely,

$$|(\boldsymbol{p} - \boldsymbol{q}) \cdot \boldsymbol{n}| = 0.$$

Two lines $(\boldsymbol{q}_a, \boldsymbol{\omega}_a)$ and $(\boldsymbol{q}_b, \boldsymbol{\omega}_b)$ are coincident if the angle and the distance between them is 0. That is, when θ in (4.7) and d in (4.4) is equal to 0. Namely, if

$$\texttt{atan2}\left(||\boldsymbol{\omega}_a \times \boldsymbol{\omega}_b||, \boldsymbol{\omega}_a \cdot \boldsymbol{\omega}_b\right) = 0$$

and

$$||(\boldsymbol{q}_a - \boldsymbol{q}_b) - \boldsymbol{\omega}_b(\boldsymbol{\omega}_b \cdot (\boldsymbol{q}_a - \boldsymbol{q}_b))|| = 0.$$

A line $(\boldsymbol{q}_l, \boldsymbol{\omega})$ is coincident with a plane $(\boldsymbol{q}_p, \boldsymbol{n})$ if the angle and the distance between the line and the plane are 0. That is, when θ in (4.8) and d in (4.5) is equal to 0. Namely if,

$$\begin{aligned} &\frac{\pi}{2} - \left|\texttt{atan2}\left(||\boldsymbol{\omega} \times \boldsymbol{n}||, \boldsymbol{\omega} \cdot \boldsymbol{n}\right)\right| = 0 \\ \Rightarrow \quad &\left|\texttt{atan2}\left(||\boldsymbol{\omega} \times \boldsymbol{n}||, \boldsymbol{\omega} \cdot \boldsymbol{n}\right)\right| = \frac{\pi}{2} \end{aligned}$$

and

$$|(\boldsymbol{q}_l - \boldsymbol{q}_p) \cdot \boldsymbol{n}| = 0.$$

Two planes $(\boldsymbol{q}_a, \boldsymbol{n}_a)$ and $(\boldsymbol{q}_b, \boldsymbol{n}_b)$ are coincident if the angle and the distance between them are 0. That is, when θ in (4.9) and d in (4.6) is equal to 0. Namely

if,

$$\texttt{atan2}\left(||\boldsymbol{n}_a \times \boldsymbol{n}_b||, \boldsymbol{n}_a \cdot \boldsymbol{n}_b\right) = 0$$

and

$$|(\boldsymbol{q}_a - \boldsymbol{q}_b) \cdot \boldsymbol{n}_b| = 0.$$

Concentric

A concentric mate can be assigned between center lines of cylindrical, conical or spherical faces. The dependencies of these are shown in Table 4.6.

Table 4.6.: Concentric mates

Feat. 1	**Feat. 2**	**Dependency**	**Constrained DOF**
Line	Line	Aligns the center lines to the same infinite line	4 DOF

With center lines described by the lines $(\boldsymbol{q}_a, \boldsymbol{\omega}_a)$ and $(\boldsymbol{q}_b, \boldsymbol{\omega}_b)$, concentricity is the same as coincident lines. That is, if

$$\texttt{atan2}\left(||\boldsymbol{\omega}_a \times \boldsymbol{\omega}_b||, \boldsymbol{\omega}_a \cdot \boldsymbol{\omega}_b\right) = 0$$

and

$$||(\boldsymbol{q}_a - \boldsymbol{q}_b) - \boldsymbol{\omega}_b(\boldsymbol{\omega}_b \cdot (\boldsymbol{q}_a - \boldsymbol{q}_b))|| = 0$$

Perpendicular

A perpendicular mate can be assigned between lines and planes. The dependencies of these are shown in Table 4.7.

Two lines $(\boldsymbol{q}_a, \boldsymbol{\omega}_a)$ and $(\boldsymbol{q}_b, \boldsymbol{\omega}_b)$ are perpendicular if the angle between them are $\pm\pi/2$. That is, when the absolute value of the angle θ in (4.7) is equal to $\pi/2$. Namely if,

$$|\texttt{atan2}\left(||\boldsymbol{\omega}_a \times \boldsymbol{\omega}_b||, \boldsymbol{\omega}_a \cdot \boldsymbol{\omega}_b\right)| = \frac{\pi}{2}.$$

A line $(\boldsymbol{q}_l, \boldsymbol{\omega})$ and a plane $(\boldsymbol{q}_p, \boldsymbol{n})$ is perpendicular if the angle between them is

Table 4.7.: Perpendicular mates

Feat. 1	**Feat. 2**	**Dependency**	**Constrained DOF**
Line	Line	Makes the lines perpendicular	1 DOF
Line	Plane	Makes the line and the plane perpendicular	2 DOF
Plane	Plane	Makes the planes perpendicular	1 DOF

$\pm\pi/2$. That is, when the angle θ in (4.8) is equal to $\pi/2$. Namely if,

$$\begin{aligned} &\frac{\pi}{2} - |\texttt{atan2}\left(||\boldsymbol{\omega} \times \boldsymbol{n}||, \boldsymbol{\omega} \cdot \boldsymbol{n}\right)| = \frac{\pi}{2} \\ \Rightarrow \quad &|\texttt{atan2}\left(||\boldsymbol{\omega} \times \boldsymbol{n}||, \boldsymbol{\omega} \cdot \boldsymbol{n}\right)| = 0. \end{aligned}$$

Two planes $(\boldsymbol{q}_a, \boldsymbol{n}_a)$ and $(\boldsymbol{q}_b, \boldsymbol{n}_b)$ are perpendicular if the angle between them is $\pm\pi/2$. That is, when the absolute value of the angle θ in (4.9) is equal to $\pi/2$. Namely if,

$$|\texttt{atan2}\left(||\boldsymbol{n}_a \times \boldsymbol{n}_b||, \boldsymbol{n}_a \cdot \boldsymbol{n}_b\right)| = \frac{\pi}{2}.$$

Parallel

A parallel mate can be assigned between lines and planes. The dependencies of these are shown in Table 4.8.

Table 4.8.: Parallel mates

Feat. 1	**Feat. 2**	**Dependency**	**Constrained DOF**
Line	Line	Makes the lines parallel	2 DOF
Line	Plane	Makes the line and the plane parallel	1 DOF
Plane	Plane	Makes the planes parallel	2 DOF

Two lines $(\boldsymbol{q}_a, \boldsymbol{\omega}_a)$ and $(\boldsymbol{q}_b, \boldsymbol{\omega}_b)$ are parallel if the angle between them are 0. That is, when the angle θ in (4.7) is equal to 0. Namely if,

$$\texttt{atan2}\left(||\boldsymbol{\omega}_a \times \boldsymbol{\omega}_b||, \boldsymbol{\omega}_a \cdot \boldsymbol{\omega}_b\right) = 0.$$

A line $(\boldsymbol{q}_l, \boldsymbol{\omega})$ and a plane $(\boldsymbol{q}_p, \boldsymbol{n})$ is parallel if the angle between them is 0. That is, when the angle θ in (4.8) is equal to 0. Namely if,

$$\begin{aligned} &\frac{\pi}{2} - |\texttt{atan2}\left(||\boldsymbol{\omega} \times \boldsymbol{n}||, \boldsymbol{\omega} \cdot \boldsymbol{n}\right)| = 0 \\ \Rightarrow \quad &|\texttt{atan2}\left(||\boldsymbol{\omega} \times \boldsymbol{n}||, \boldsymbol{\omega} \cdot \boldsymbol{n}\right)| = \frac{\pi}{2}. \end{aligned}$$

Two planes $(\boldsymbol{q}_a, \boldsymbol{n}_a)$ and $(\boldsymbol{q}_b, \boldsymbol{n}_b)$ are parallel if the angle between them is 0. That is, when the angle θ in (4.9) is equal to 0. Namely if,

$$|\texttt{atan2}\left(||\boldsymbol{n}_a \times \boldsymbol{n}_b||, \boldsymbol{n}_a \cdot \boldsymbol{n}_b\right)| = 0.$$

Tangent

A tangent mate can be assigned between the face of a cylinder, cone or sphere and a line or a plane. The dependencies are shown in Table 4.9.

Table 4.9.: Tangent mates

Feat. 1	**Feat. 2**	**Dependency**	**Constrained DOF**
Cylinder	Line	The infinite line intersects *a* line on the face (Cylindrical joint)	2 DOF
Cylinder	Plane	The infinite plane intersects *a* line on the face (Cylindrical joint)	3 DOF
Sphere face	Line	*A* point on the sphere is on the infinite line (Spherical joint)	3 DOF
Sphere face	Plane	*A* point on the sphere is on the infinite plane (Spherical joint)	3 DOF

A line $(\boldsymbol{q}_a, \boldsymbol{\omega}_a)$ is a tangent of a cylinder $(\boldsymbol{q}_c, \boldsymbol{\omega}_c, r)$ if the distance between the centerline of the cylinder and the line is equal to the radius r. That is, if the distance d in (4.4) is equal to r, where $\boldsymbol{q}_b$ and $\boldsymbol{\omega}_b$ is substituted by $\boldsymbol{q}_c$ and $\boldsymbol{\omega}_c$ respectively. Namely if,

$$||(\boldsymbol{q}_a - \boldsymbol{q}_c) - \boldsymbol{\omega}_c(\boldsymbol{\omega}_c \cdot (\boldsymbol{q}_a - \boldsymbol{q}_c))|| = r.$$

A plane $(\boldsymbol{q}_p, \boldsymbol{n})$ is a tangent of a cylinder $(\boldsymbol{q}_c, \boldsymbol{\omega}_c, r)$ if the distance between the centerline of the cylinder and the plane is equal to the radius r. That is, if the distance d in (4.5) is equal to r, where $\boldsymbol{q}_l$ is substituted by $\boldsymbol{q}_c$. Namely if,

$$|(\boldsymbol{q}_c - \boldsymbol{q}_p) \cdot \boldsymbol{n}| = r.$$

A line $(\boldsymbol{q}, \boldsymbol{\omega})$ is a tangent of a sphere $(\boldsymbol{p}, r)$ if the distance between the center point of the sphere and the line is equal to the radius r. That is, if the distance d in (4.2) is equal to r. Namely if,

$$||(\boldsymbol{p} - \boldsymbol{q}) - \boldsymbol{\omega}(\boldsymbol{\omega} \cdot (\boldsymbol{p} - \boldsymbol{q}))|| = r.$$

A plane $(\boldsymbol{q}, \boldsymbol{n})$ is a tangent of a sphere $(\boldsymbol{p}, r)$ if the distance between the center point of the sphere and the plane is equal to the radius r. That is, if the distance d in (4.3) is equal to r. Namely if,

$$|(\boldsymbol{p} - \boldsymbol{q}) \cdot \boldsymbol{n}| = r.$$

Chapter 5.

Software Implementation

The contribution made in software by this thesis is presented in this chapter. That is, a CAD constraint extractor for SolidWorks, geometric functions for eTaSL, the eTaSL ROS controller, and a joint limit interface for KUKA RSI.

5.1. Constraints from SolidWorks

SolidWorks (SW) is one of many software packages made for 3D modeling. CAD in general terms can be used to describe either 2D vector (line), 3D surface or 3D solid body based software. In the rest of this report, CAD is used to describe a 3D solid body modeling program in general terms, independent of software supplier. The source code which controls the workings of CAD software is usually hidden from the users, as much money goes into the development. SW is different from many other CAD software suppliers, as they have an accessible API. API is short for application programming interface and is a tool for software building. The API of SW gives access to predefined functions which makes it possible to interact with their software system.

SW has a large library of API calls implemented for C#. By an implementation in a stand-alone .NET [27] application interface, we are able to implement a set of calls to extract the necessary information from the CAD assembly model. The workings of this application is inspired by [4] and [26] among others with their approach described in section 1.2. The code is available in its entirety at GitHub [21]. In short, the three most important API interfaces used are `IModel-`

`Doc2`, `IModelDocExtension`, and `IDragOperator`. These three interfaces have many member functions, that open possibilities for interaction with the CAD model. `IModelDoc2` is used to open and give access to the assembly and parts within the assembly, while the `IModelDocExtension` lets you select and interact with a range of parameters in SW. In this instance, it was used to select the parts for mate suppression and also for the export of mesh files. The `IDragOperator` was used to be able to transform the parts for generation of possible assembly directions.

The `OpenDoc6()` member of `IModelDoc2` was used to open the assembly file and get the parts. The `SaveAs()` member of `IModelDocExtension` was used to save the parts into the .stl file format, which is a file format giving a meshed representation of the parts. The `MathUtility` interface was used to extract the x, y and z components of the transform attached to the part origin. The mate parameters, as described thoroughly in section 4.1, can be exploited using the `IMate2` interface. There is no direct API interface giving possible assembly directions, but by using the `IDragOperator` interface, we can make an algorithm that moves the parts in increments along the three principal axes, while checking for a collision between the assembly parts. By acknowledging that most engineering assemblies are assembled along the principal axes [6], a collision along an axis would suggest an unable assembly direction, while no collision would give a possible approach direction during the assembly operation. This is illustrated in Figure 5.1, where it is shown that a translation of the cylinder in the positive x-direction would result in a collision between the parts.

It is easy to see that the only possible direction of assembly would be along the y-axis. The result of the assembly direction check is an array on the form $(x+, x-, y+, y-, z+, z-)$, where the parameters would be either 0 or 1 depending on whether it collides or not along that direction. Thus, the interference array for this assembly will be as given in Listing 5.1. Any automatic transition from the output of the SW API to eTaSL constraints is out of scope for this project, but the structure and information is used when defining the geometrical functions described previously in chapter 4 and the constraints used for the peg-in-hole task.

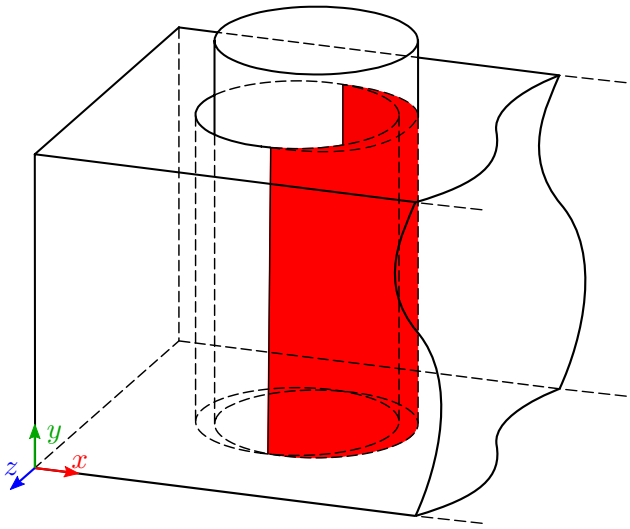


Figure 5.1.: A collision is detected in the positive x-direction

Listing 5.1: Part of code output

```
Coincident1
  Type              = 0
  Alignment         = 1
  Can be flipped    = False

  Component         = peg1
  Origin            = (25, 0, -25)
  Mate entity type  = 3
  (x,y,z)           = (25, 0, -25)
  (i,j,k)           = (0, -1, 0)
  Radius 1          = 0
  Radius 2          = 0

(x+, x-, y+, y-, z+, z-) = (0, 0, 1, 1, 0, 0)
1 is possible, 0 is not possible

All dimensions are in mm
```

5.2. Geometric Functions in eTaSL

The mathematics of the constraints presented in section 4.2 have been implemented in `geometric3.lua` as functions that can be used with eTaSL to set up

geometrical constraints. The eTaSL framework does already have a function library for geometric entities in a script called `geometric2.lua`. We saw a need for writing our own geometric functions as `geometric2.lua` where missing some of the functions presented in section 4.2. We also wanted the inputs for the functions to be more in line with the output of the SolidWorks output generator. In addition, instead of formulating a constraint as presented in Listing 3.3, the constraints are now formulated as given by the examples in Listing 5.2 and 5.3

For example in Listing 5.2, the arguments from `context` to `priority` are the same as the original constraint formulation. Where `expr` was used before, `point_a`, `dir_a`, `point_b`, `dir_b` should be used instead. For a concentric constraint, points are points on the lines and directions are unit direction vectors of the lines. Not all CAD constraints need all four vector expressions, and some would need more. `Coincident_point_point{}` would only need `point_a` and `point_b` for instance, and `Tangent_plane_cylinder{}` would need `r`, the radius, as well. The formulation for all the different constraints are given in Appendix A. The source code is available in its entirety at GitHub [22]. To sum up, `point` is a necessary argument for all geometric entities, `dir` is necessary for lines, planes and cylinders, and `r` for cylinders and spheres.

The functions are implemented in a way that helps the user to call the functions correctly. If unsure of what parameters are necessary, the function call can be left empty. That is, as an example `Coincident_point_line{}`. A message describing the possible inputs will then be printed. If the function is written in the opposite order, for instance `Coincident_line_point{}`, the same message will be printed. If defining too many arguments, the functions would discard what ever is in excess and it would print the information message if any non optional arguments are missing. The order of the arguments are indifferent.

Listing 5.2: Constraint formulation for concentricity

```
Concentric{
    context  = ctx,
    name     = "name"[optional],
    K        = (scalar)[optional],
    weight   = (scalar)[optional],
    priority = (1 or 2)[2 default],
    point_a  = (vector expression)[point on one line],
    dir_a    = (vector expression)[unit vector along that line],
    point_b  = (vector expression)[point on other line],
    dir_b    = (vector expression)[unit vector along that line]
}
```

Listing 5.3: Constraint formulation for a plane tangent to a sphere

```
Tangent_plane_sphere{
    context  = ctx,
    name     = "name"[optional],
    K        = (scalar)[optional],
    weight   = (scalar)[optional],
    priority = (1 or 2)[2 default],
    point_a  = (vector expression)[point on the plane],
    dir_a    = (vector expression)[unit normal vector of plane],
    center   = (vector expression)[center point],
    r        = (positive scalar)[radius of sphere]
}
```

5.3. eTaSL ROS Controller

As many robots can be controlled by ROS control, but the developers of eTaSL uses the Orocos framework as the controller layer, there was a need for a ROS based controller for eTaSL. In the beginning of this project, Lars Tingelstad wrote the initial code for an eTaSL ROS controller. One of the aims of this project has been to develop this further. The code is available in its entirety at GitHub [40].

5.3.1. Schematics of eTaSL ROS Controller

The eTaSL ROS controller is set up in a way that ROS message types can be used as inputs and outputs of the eTaSL expression types presented in section 3.3. The ROS message types and their corresponding eTaSL expression types are given in Table 5.1. As ROS uses quaternions to represent rotations and eTaSL uses rotation matrices, the quaternion is translated into a rotation matrix with the formula (2.7) presented in chapter 2. Equivalently, a `geometry_msgs::Pose` is a composition of a `geometry_msgs::Point` and a `geometry_msgs::Quaternion`, while the `KDL::Frame` is a transformation matrix. Hence, the quaternion is translated into the rotation part of the transformation matrix as in (2.7) and the point becomes the translation part of the transformation matrix.

The main purpose of the use of ROS message types, is that other nodes in the ROS system can publish information to the eTaSL task or for instance graph outputs of eTaSL. As an example, we can have a 3D-image node that publishes the position and orientation of a part on a `Pose` message, which then can be used

Table 5.1.: ROS message types vs eTaSL expression types

ROS message type	eTaSL expression type
`std_msgs::Float64`	`double`
`geometry_msgs::Point`	`KDL::Vector`
`geometry_msgs::Quaternion`	`KDL::Rotation`
`geometry_msgs::Pose`	`KDL::Frame`
`geometry_msgs::Twist`	`KDL::Twist`
`geometry_msgs::Wrench`	`KDL::Wrench`

to make an expression for a constraint.

The eTaSL ROS controller is driven by the controller manager of ROS Control, which at startup initiates the controller, starts it, and calls update until it is stopped. The initiation of eTaSL ROS controller sets up a ROS Control `hardware_interface` of the type `VelocityJointInterface`. The type of robot which communicates with the hardware interface is indifferent as long as that robot is set up with a ROS Control hardware interface of the same type and that it has an available URDF model. When started, the joint positions of the robot is read from the robot, and eTaSL is initiated with the values. At every update cycle, any inputs from other ROS nodes are read, before the current joint positions are read from the hardware interface. The inputs and joint positions are written to eTaSL, and the optimization problem is solved. Then, the new desired joint velocities are read from eTaSL and written to the hardware interface. An illustration of the data flow during the update cycle is shown in Figure 5.2, where a KUKA robot with a RSI hardware interface is used as an example.

5.3.2. Task Switching

When setting up a task in eTaSL, one would often like to divide the task into subtasks which are being switched between, either because a subtask is finished or some event has happened. ROS has an implementation for task switching where a ROS service call can be made to switch between different controllers. This service call is time consuming, and needs the subtasks to be divided into several task scripts. To try to both decrease the time needed for a switch and and

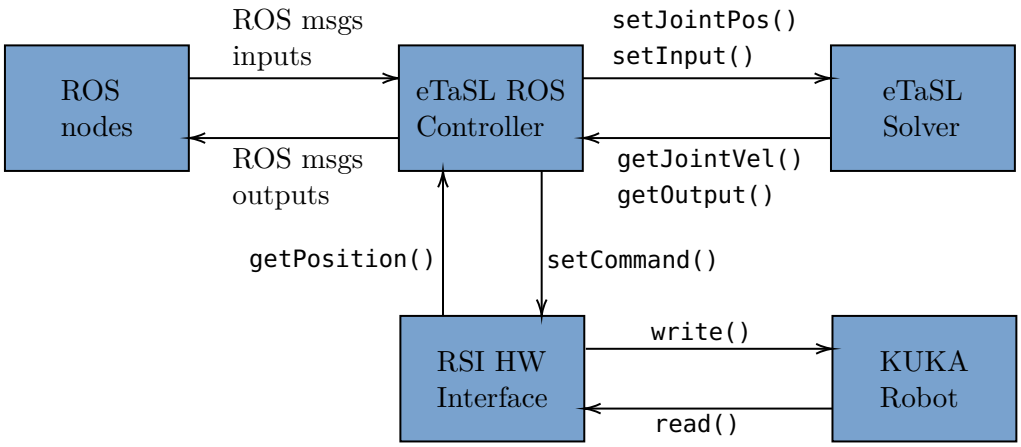


Figure 5.2.: Data flow in eTaSL ROS controller

make it all happen within eTaSL itself using monitors and groups, the following has been done.

Monitors

eTaSL is made to be able to activate and deactivate constraints from within the task specification script through monitors with grouping observers. This can be achieved by having `actionname = "activate"` and `argument = "+global.group_to_activate -global.group_to_deactivate"`, ref. Listing 3.5.

Unfortunately, this implementation is not possible with the ROS controller directly, as the controller has to be stopped before activating any group. Hence, another method is used. The actionname for the default observer is `"exit"` where no argument is given, but the ROS controller will allow for an argument similar to the grouping observer monitor. As the monitor is triggered, the ROS controller will read the argument, stop the controller, edit the active groups, and start the controller again. This is achieved by checking if any monitor has sat a `finishStatus`, before iterating through all monitors checking for an active monitor that has been triggered, giving a `stopRequest` to the controller, reading the argument into the `activate_cmd()`-function, and finally giving a `startRequest` to the controller. A task switching monitor needs to be structured like given in Listing 5.4.

Listing 5.4: Task switching monitor

```
Monitor{
    context     = ctx,
    name        = "name",
    expr        = (expression),
    lower       = (scalar)[optional],
    upper       = (scalar)[optional],
    actionname  = "exit",
    argument    = "+global.group_to_activate
                   -global.group_to_deactivate"
}
```

Service Request

Another solution to activate and deactivate groups, that is available to any ROS node, is made. This solution is a ROS service named `activate_cmd`. When the eTaSL ROS controller is initiated, a service is advertised which gives access to the built in eTaSL function `activate_cmd()`. Any ROS node, for instance the SMACH state machine, could then set up a service client on `/controller_name/activate_cmd`, and call it with a command similar to the argument of the task switching monitors. This will stop the controller, edit the active groups, and start the controller again. In a Python node, it would look as in Listing 5.5

Listing 5.5: Task switching service client

```
rospy.wait_for_service('/controller_name/activate_cmd')
activate_cmd = rospy.ServiceProxy('/controller_name/activate_cmd',
    Command)
resp = activate_cmd("+global.group_to_activate")
return resp.ok
```

During debugging and testing, this service comes to handy, as any service can be called from command line during execution. Thus, one is able to test the effect of some constraints or temporary stop the execution, by the `activate_cmd`-service. It can be called in terminal as shown in Listing 5.6, which shows an example of how all constraints are deactivated.

Listing 5.6: Task switching from terminal

```
rosservice call /controller_name/activate_cmd '!!str -global.*'
```

5.3.3. Exit Event Publisher

When a monitor triggers it sets a finish flag to a `finishStatus`-variable. A function is implemented in the ROS controller which checks this variable, and publishes the string `"exit"` to a topic named `e_event`. This can for instance be used by SMACH to know when the robot is in its pick up position, thus when to activate the gripper.

5.4. Hardware Interface for KUKA RSI

Robots have different kinematic and dynamic properties. Thus, they are able to behave differently to the same joint commands from eTaSL. Within the ROS Control `hardware_interface` there exists a `joint_limits_interface` which lets you impose robot specific velocity and acceleration limits. This is essential when using the robot for an eTaSL task with CAD-constraints because of the exponential behaviour of eTaSL constraints, which were presented in section 3.3. eTaSL does not know the dynamical properties of the robot, so it will try to reach the goal at an exponential rate. This can mean that both the desired joint velocities from eTaSL and the corresponding accelerations are too high for the robot. This is where the `joint_limits_interface` comes in handy.

The hardware interface for KUKA RSI communicates with the robot controller via Ethernet UDP/IP with XML strings. KUKA RSI runs at 250 Hz, which dictates the update cycle for all calculations in the entire eTaSL system. When the function `read()` is called in the hardware interface, an XML string with the current joint angles of the robot is read of the robot controller and sent to the hardware interface. Similarly, when the `write()` function is called, desired incremental joint positions are sent as an XML string back to the robot controller. As the clock of the robot controller dictates the execution, it will always execute every 4 ms. If no command is available from the hardware interface at that moment, the execution will fail.

Joint Limits

As KUKA RSI operates with incremental joint positions in degrees, and eTaSL solves for incremental joint velocities in rad/s, they have to be translated before `write()` is called. This is also where the joints limits are enforced. When the `setCommand()` is called, it will give the desired joint velocities of eTaSL to the hardware interface. The hardware interface then takes these joint velocities and compares them to the defined velocity and acceleration limits. The maximum allowed velocity would be lowest velocity of either the previous velocity plus the acceleration limit times the time step or the velocity limit. That is,

```
vel_high = min(prev_vel + max_acceleration*dt, max_velocity);
```

If the commanded velocity is lower than this value it will be kept. Otherwise `vel_high` is kept. The commanded incremental joint angle sent to the robot is then calculated as the kept velocity times the time step and converted to degrees. The command `write()` is then called. The KUKA hardware interface with the implemented joint limits written during this project is available at GitHub [39].

Chapter 6.

Testing eTaSL with a KUKA Robot

Three tests have been set up to verify the behaviour of the eTaSL ROS controller and the defined assembly task, when paired with a physical robot. The first is looking at the switching times that can be achieved with the different methods of task switching presented in section 5.3. The second is looking at the behaviour of the robot when an eTaSL task is ran with the eTaSL ROS controller. The last is looking at the ability to perform a peg-in-hole task operation with peg and hole combinations with different tolerances.

6.1. Test of Switching Times

Three different methods for task switching was presented in section 5.3. That is, one method using smach to call a switching service on the controller manager, one method calling a service on the eTaSL ROS controller directly activating and deactivating groups of constraints, and one method where monitors in the eTaSL task specification activates and deactivates constraint groups directly in the eTaSL ROS controller.

To find out which method of task switching that is the fastest, and if any of them are fast enough for realtime control, a timer was implemented in code to time the time needed to stop the controller, change task, and start back up again. The task was run three times and the time consumption for all switches with all the

different switching methods are plotted. The results are given in the next section, and discussed in section 7.1.

6.2. Results of Switching Timing

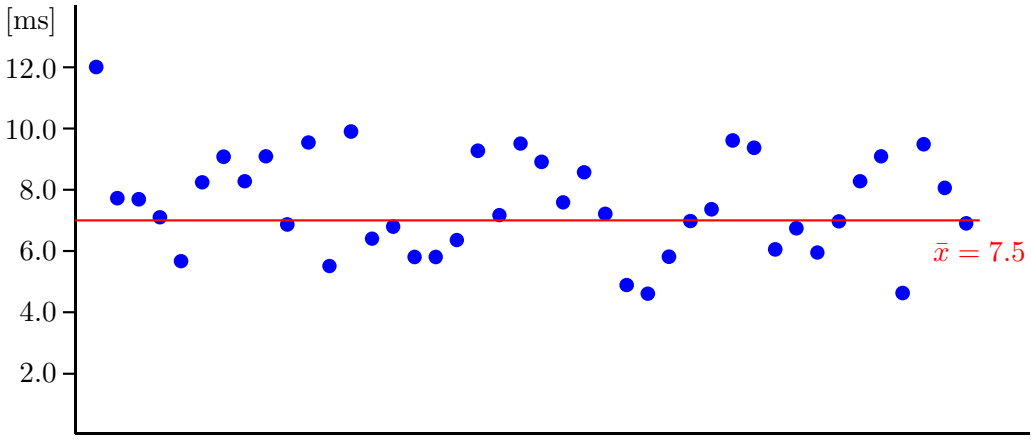


Figure 6.1.: Switching times with switching service

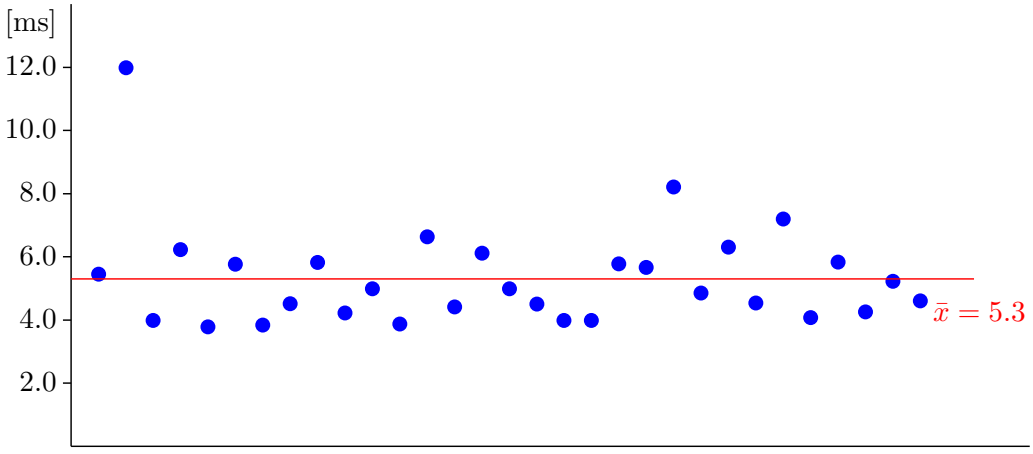


Figure 6.2.: Switching times with service call

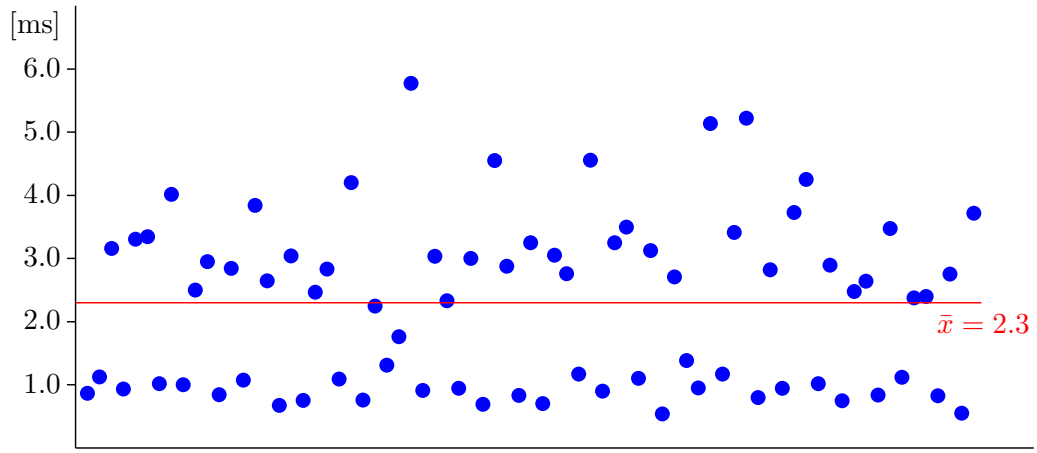
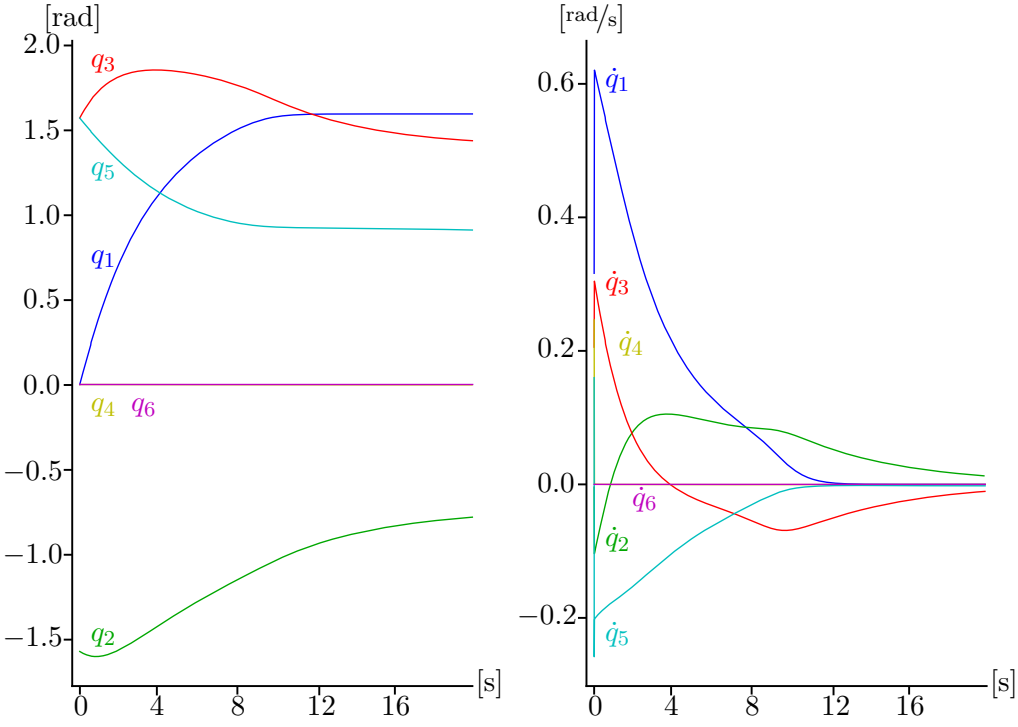


Figure 6.3.: Switching times with monitors

6.3. Test of Robot Behaviour with eTaSL

The software has been run several times both in simulation and on the physical robot during the entire project, to continuously strive for a better performance. Some plots of intermediate and final behaviours of the robot will be presented in the next section and will be discussed thoroughly in section 7.2.

6.4. Results of Robot Behaviour Test



(a) Joint position **(b)** Joint velocity

Figure 6.4.: Exponential behavior of joints

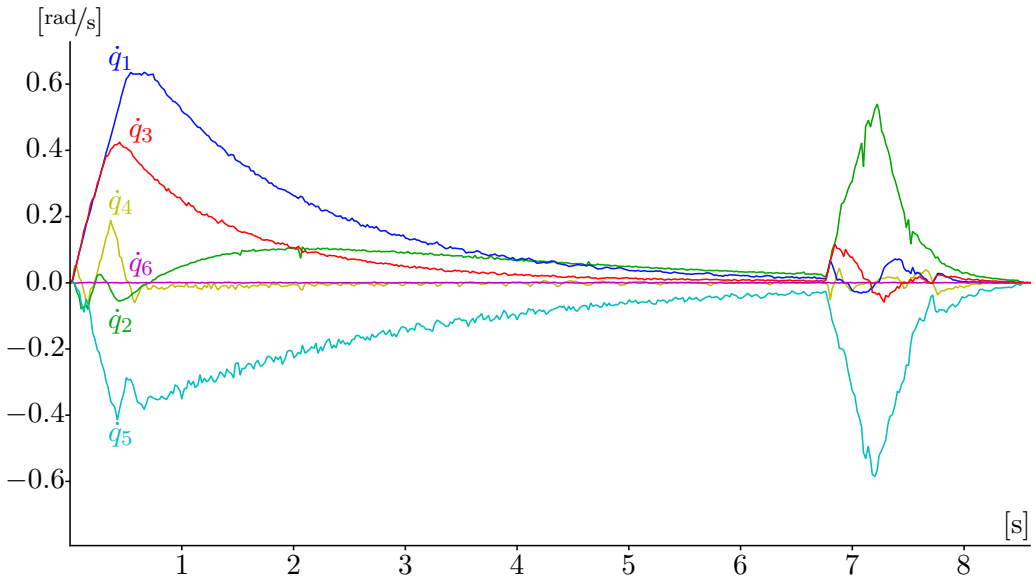


Figure 6.5.: Joint velocities with limits derived from joint positions

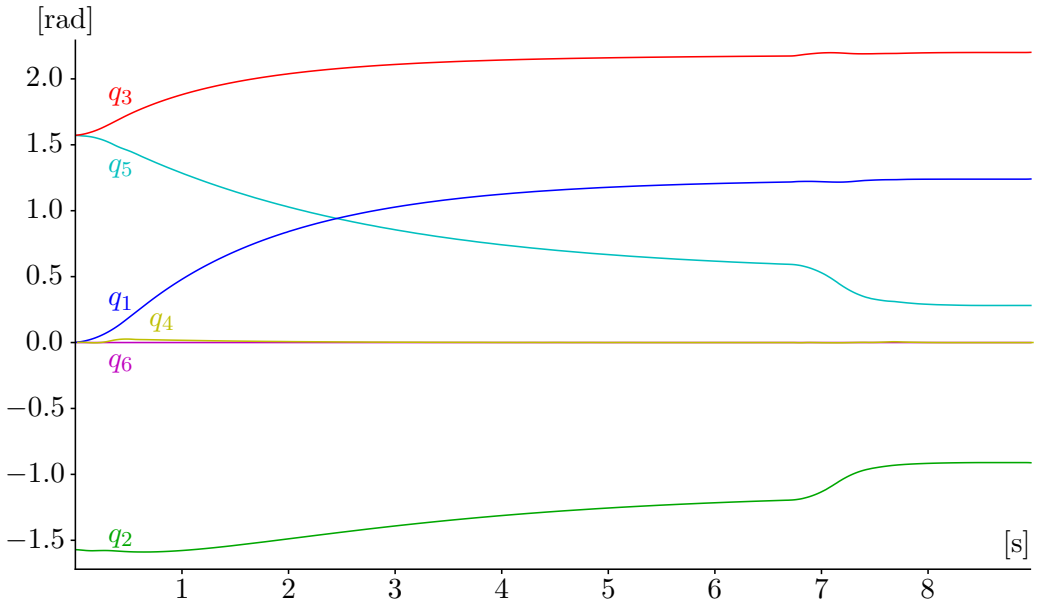


Figure 6.6.: Joint positions with limits

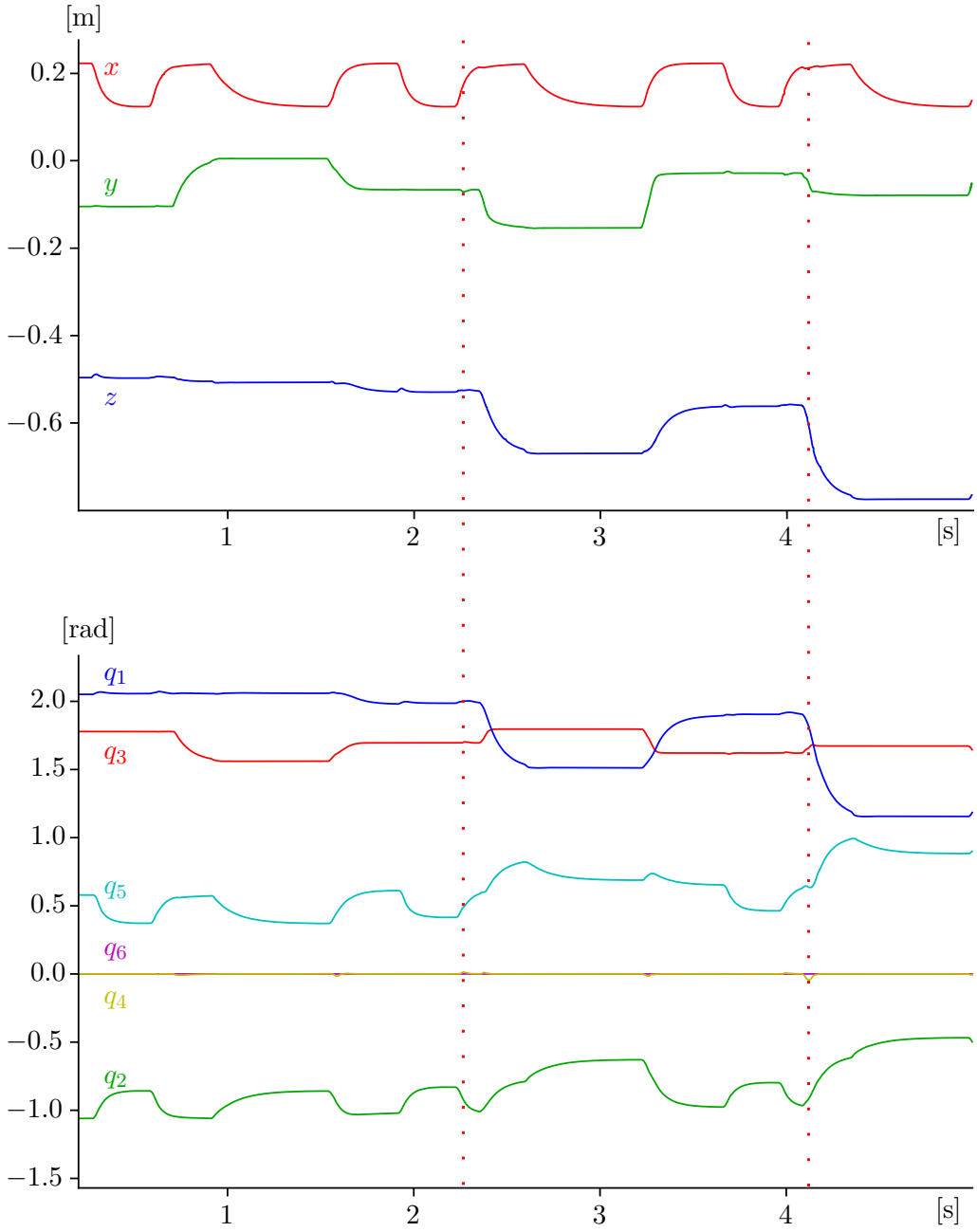


Figure 6.7.: Small wobbles

6.5. Hardware Setup Peg-In-Hole Test

The following gear where used in the experiments, and can be seen in Figure 6.8 as well as in Figure 6.9.

- KUKA Agilus KR6 R900 sixx (Robot)
- Zivid One (3D camera)
- ATI Gamma IP60 (Force sensor)
- Robotiq 2F–85 (Grippers)
- Steel bar with 5 holes
- 3 sets of 5 pegs

KUKA Agilus KR6 R900 sixx

The robot used in the tests are KUKA's Agilus KR6 R900 sixx. Technical data for the robot can be found in [38]. The most important factor of the robot for the tests, are what is referred to as "*Pose repeatability (ISO 9283)*", which for this robot is ±0.03 mm. This is a measure of how much variance there is in the actual pose of the end-effector frame approaching a goal pose from the same direction. It does not really tell how accurate the robot is at positioning, but gives an estimate of how precise one could expect it to be. It is also worth noticing that the maximum payload is 6 kg.

Zivid One

Zivid One is a 3D-camera delivering point clouds in the RGB-D format. That is, each point in the cloud has a distance relative to the camera frame together with its x and y position and a color value. According to their user guide [31], a point cloud frame is 2.3 Mpixels at 1920x1200 pixels and has a depth resolution of 0.1 mm. At a distance of 1100 mm, which the camera is mounted at, the resolution in the plane becomes aprx. 0.41 mm.

(a) KUKA KR6 R900 [20]

(b) ATI Gamma IP60 [13]

(c) Robotiq 2F–85 gripper [16]

(d) Zivid One [41]

(e) Bar with holes and pegs

Figure 6.8.: Gear used in lab setup

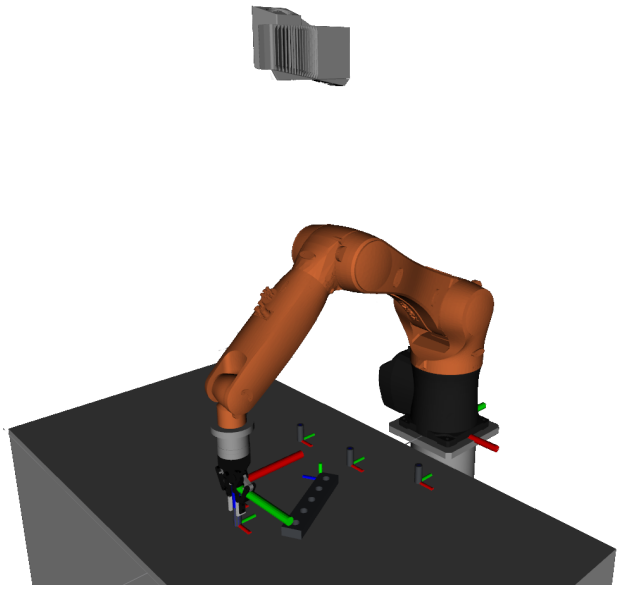


Figure 6.9.: RVIZ model of lab setup

ATI Gamma IP60

The force/torque sensor used is the Gamma IP60 by ATI Industrial Automation. The measuring ranges and resolutions are as given in Table 6.1 according to their operation manual [14].

Table 6.1.: Sensing ranges ATI Gamma IP60

	F_x, F_y	F_z	T_x, T_y	T_z
Sensing ranges	65 N	200 N	5 Nm	5 Nm
Resolution	1/80 N	1/40 N	10/13333 Nm	10/13333 Nm
Resonant Frequency	1200 Hz	1200 Hz	1200 Hz	1200 Hz

Robotiq 2F–85 gripper

The two finger gripper 2F–85 by Robotiq has been used together with custom v-grooved fingers for picking and holding of the pegs. It can according to Robo-

Table 6.2.: Selection of ISO tolerances for holes and shafts [17]

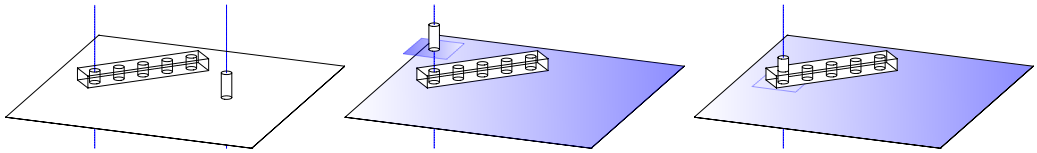
Nominal size mm										
Above	3	6	10	18	30	50	80	120	180	250
Up to and inc.	6	10	18	30	50	80	120	180	250	315
Deviation limits μm										
H9	+30 0	+36 0	+43 0	+52 0	+62 0	+74 0	+87 0	+100 0	+115 0	+130 0
d9	-30 -60	-40 -76	-50 -93	-65 -117	-80 -142	-100 -174	-120 -207	-145 -245	-170 -285	-190 -320
h6	0 -8	0 -9	0 -11	0 -13	0 -16	0 -19	0 -22	0 -25	0 -29	0 -32

tiqs documentation [16], apply clamping forces between 20 and 235 N and has a maximum payload of 5 kg. The position repeatability of the fingertips are 0.05 mm.

Test objects

The test objects are a set of one steel bar with 5 holes, and 3 sets of five steel pegs, with different ISO tolerances. The applicable tolerances is given in Table 6.2. The holes in the bar are all of $\varnothing$20H9 mm. That is, $\varnothing 20^{+0.052}_{0.000}$ mm, which means that all holes have a diameter between 20.000 and 20.052 mm.

The pegs are divided into three sets of five pegs where each set is at different tolerances. The first set with a clearance to the holes are $\varnothing$19.6 mm. The second set has a "*free running clearance fit*" with the holes and are $\varnothing$20d9 = $\varnothing 20^{-0.065}_{-0.117}$ mm. That is, all peg have a diameter between 19.883 and 19.955 mm. The last set has a "*sliding clearance fit*" with the holes and are $\varnothing$20h6 = $\varnothing 20^{0.000}_{-0.013}$ mm.



(a) Pick up **(b)** Line up **(c)** Insertion

Figure 6.10.: Assembly steps

6.6. Software Setup Peg-In-Hole Test

All the necessary constraints for the peg-in-hole task will be presented in the following section. The lua code with the task specification is available at GitHub [23].

6.6.1. Implementation in eTaSL

For the peg-in-hole assembly task, there are four main subtasks; "*Pick up line up*", "*pick up*", "*insertion line up*" and "*insertion*". The states of the peg during those subtasks are shown in Figure 6.10.

For the pick up line up of the peg, three constraints are needed, defined through two functions of the `geometric3.lua` script. The first is the function `Coincident_line_line{}` which takes the origin and direction of the peg which can be found by `origin()` and by `unit_z(rotation())` of the peg frame input respectively, and the origin and the direction of the gripper frame, which is a variable of the joint positions. The `Coincident_line_line{}`-function will set up two constraints with the same `K`, `weight`, and `priority` values. One constraint using the `distance_line_line()`-function as the expression and one using the `angle_line_line()`-function. The third is a constraint using the `distance_plane_plane()`-function as the expression, which takes the gripper origin and direction, and the peg origin and direction as inputs. It also takes a target value that makes the gripper line up a little bit above the peg before the final approach.

The pick up subtask takes the same arguments as the pick up line up subtask, except the `distance_plane_plane()` now has its target set to zero.

The line up subtask, does also take three constraints, which are defined through two functions. The first is the function `Concentric{}` which does the same as the

`Coincident_line_line{}`-function described above, although the input is now the origin and direction of the hole center input frame and the origin and the direction of the peg relative to the gripper.The third constraint takes the `distance_plane_plane()`-function as the expression, which takes the same variables as the `Concentric{}`-function, together with a target value a little bit above the hole.

The insertion subtask has the same constraints as the insertion line up subtask, except that the target of the `distance_plane_plane()` constraint is set to zero.

These 4 subtasks are then looped 5 times for the five different pegs before a set of constraints are set on the joint angles for the robot to go back to "*home*" position.

6.6.2. ROS Nodes for the Peg-in-Hole Task

The eTaSL framework allows for a set of message types from ROS to be used as input variables in the task specification script. Those are `Float64`, `Point`, `Quaternion`, `Pose`, `Twist`, and `Wrench`. The eTaSL ROS controller requires the inputs to be subscribed from `/node_handle/input_name` topics, where `/node_handle` is the name of the assigned controller. The node map given in Figure 6.11 shows all the nodes and all the topics of the peg-in-hole task, and how they are connected. This map is automatically generated from ROS, and seems complex. Robot systems may often be complex, but the beauty of ROS, is that the real implementation is far less complex. A presentation of all the nodes are given in the following sections.

Pose Publisher Node

As the output message of the 3D-scanner and matching node publishes a `PoseStamped` message and the eTaSL framework takes the simpler `Pose` message as input, a republisher node was made, that subscribes to the messages from the 3D-scanner node, and publishes them back out as `Pose` messages. It does also publish `Marker` messages that publishes meshes of the parts for visualization in Rviz.

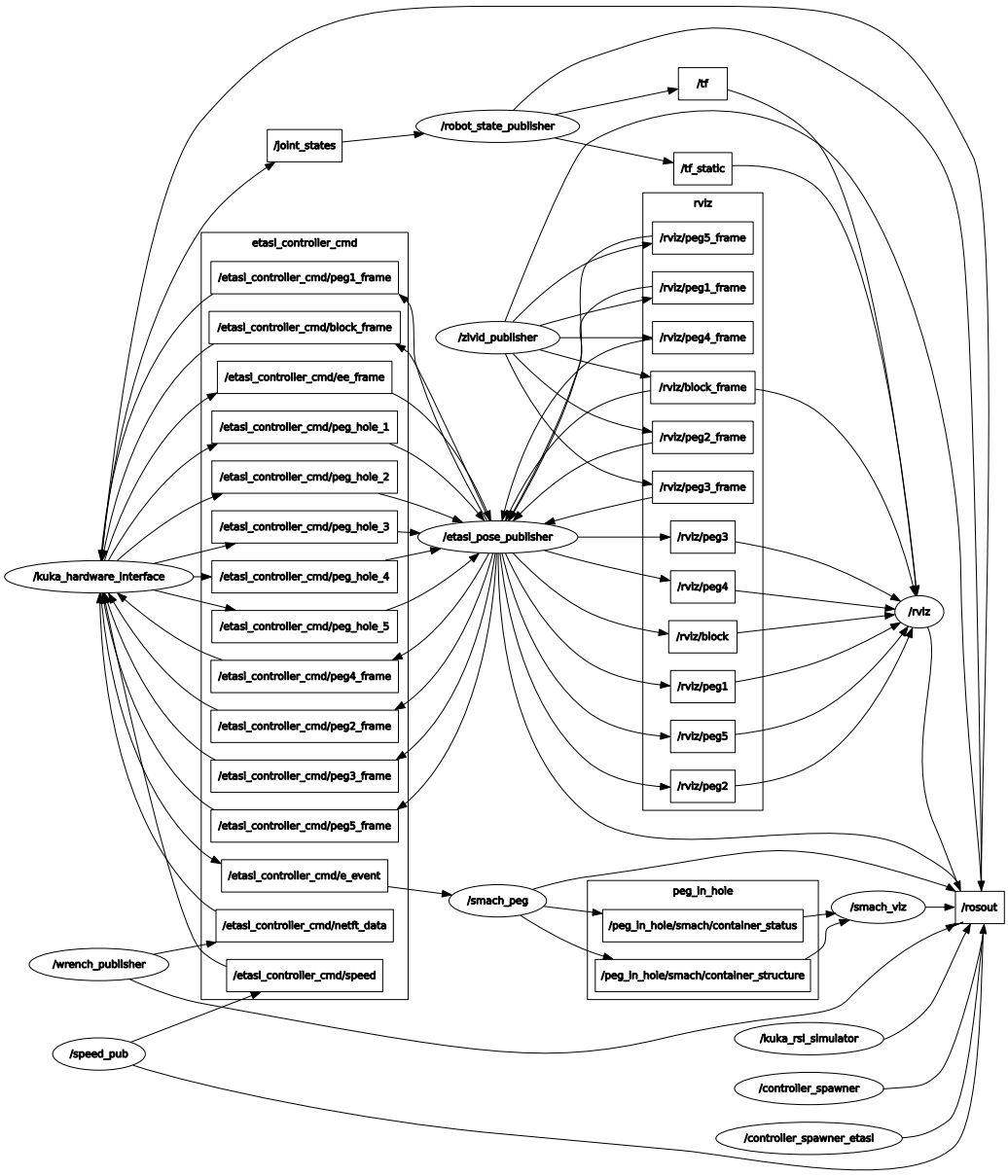


Figure 6.11.: All nodes and topics

Gripper Controller Node

The 2F-85 gripper from Robotiq, has its own ROS library with a TCP controller node, amongst others, that is used in our task. When the controller node is initiated, it will subscribe to an `outputMsg` message on a `/Robotiq2FGripperRobotOutput` topic and act accordingly while publishing back an `inputMsg` message on a `/Robotiq2FGripperRobotInput` topic with the status of the gripper. A node has been written that publishes opening and closing commands on the `/Robotiq2FGripperRobotOutput` topic and reads the status on the `/Robotiq2FGripperRobotInput` topic.

Force Sensor Publisher Node

A ROS library for the force sensor used is already made by the user fsuarez6 and is available at GitHub [14]. The publisher node for the force sensor in this library publishes by default a `Wrench` message on the `/netft_data` topic. The only thing needed to use this publisher was to use the `remap` statement in the launch file. This is done as the eTaSL task needs the controller name as node handle on the topics. Thus, the force sensor node publishes on a `/"node_handle"/netft_data` topic instead of the `/netft_data` topic.

6.6.3. State Switching in SMACH

SMACH is used to control the switching between the eTaSL task and the actuation of the gripper and request for a scan with the 3D-camera. SMACH subscribes to the published exit events from the eTaSL controller to keep track of the states of the robot. When SMACH has received an exit event from the pick up monitor or the insertion monitor it will close and open the grippers accordingly. A service request is send to the eTaSL controller to activate the robot again as soon as the gripper actuation is complete. A node map generated by SMACH with all the states are shown in Figure 6.12.

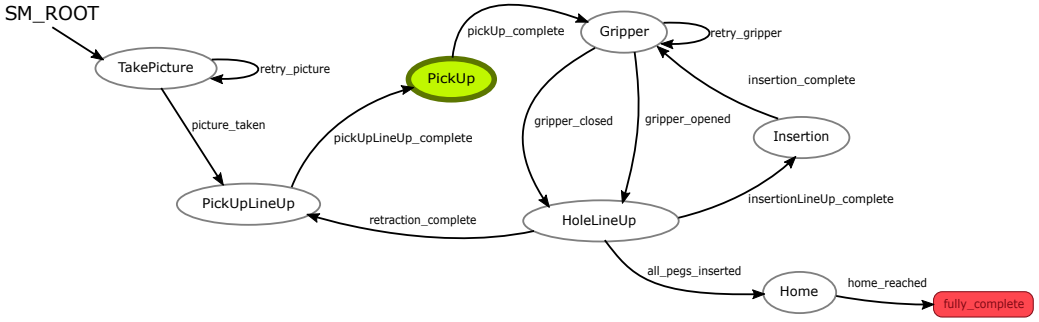


Figure 6.12.: Smach State Map

6.7. The Peg-In-Hole Test

To test the precision of the eTaSL peg-in-hole task, the robot will be run through the entire task with the three different sets of pegs. The peg-in-hole task was presented in section 6.6. In short, the robot starts from "*home*"-position, picks up the first peg and inserts that peg. Then the next peg is picked up and positioned until all five are inserted. The robot should then return to "*home*." In the robot lab, the "*home*"-position is an upright position besides the table. The pegs will be positioned randomly around the center of the workspace range of the robot. The robot will have to show that it is able to position the pegs from different distances and angles from the block with the holes.

For the first set of pegs, which is ⌀19.6 mm, the clearance to the holes will be between 0.400 mm and 0.462 mm. This means that even though the robot misses its target position at insertion, it can on average be off by ±0.215 mm in any direction. The second set at ⌀20d9 mm, the clearance with the holes will be between 0.065 mm and 0.169 mm. Thus the position of the robot can in this case be off by ±0.059 mm on average. The last set at ⌀20h6 mm has a clearance between 0.000 mm and 0.065 mm. Worst case, the robot has to be spot on. On average it can be off by ±0.016 mm.

Looking at the precision of the position of the peg in the grippers after it has been picked up by the robot, we can assume that the precision of the 3D-camera is irrelevant. That is because as long as the gripper is approximately in the right position when it closes, the v-shape of the gripper fingers will center the peg in the gripper. Thus, only the precision of the robot and the gripper determines the position precision of the peg. Assuming that the precision of both the robot and

the grippers are normal distributed, the total precision becomes

$$T = \sqrt{T_r^2 + T_g^2}$$
$$T = \sqrt{(0.03 \text{ mm})^2 + (0.05 \text{ mm})^2}$$
$$T = 0.058 \text{ mm}.$$

If we assume that the block with the holes is perfectly in place, this means that we could expect the robot to be able to place all of the ⌀19.6 mm and ⌀20d9 mm pegs, but struggle with the ⌀20h6 mm pegs. Another assumption has to be made for this to be correct. That is, that there is no or only a small error in the orientation of the peg. The peg can easily become jammed during insertion if the robot is inserting it at an inclination.

The assumption that the block is perfectly placed, can be almost true if it is placed manually with a preprogrammed position. When determining it with a 3D-camera on the other hand, that assumption is not valid. The pose determination is done with a matching scheme borrowed from Eirik Wik Haug, another Master's student at NTNU. It is based on the algorithm of Drost et al. [9]. Theoretically, will this algorithm not perform worse than the resolution of the 3D-camera used. For the configuration of the Zivid camera used, that is 0.41 mm. Thus, the total precision of the system becomes

$$T = \sqrt{(0.03 \text{ mm})^2 + (0.05 \text{ mm})^2 + (0.41 \text{ mm})^2}$$
$$T = 0.414 \text{ mm}.$$

In that case we could not expect any of the pegs to be placed correctly. The results are given in the next section, and discussed in section 7.3.

6.8. Results of Peg-in-Hole Task

Unfortunately, the 3D-matching did not work properly within the time frame of the project. But as that only were meant to be an extra addition to the test, it does not make any difference to the validation of the eTaSL peg-in-hole task. Thus, a preprogrammed position of the pegs and block where used instead. The pegs and the block where placed manually on the table with the corresponding

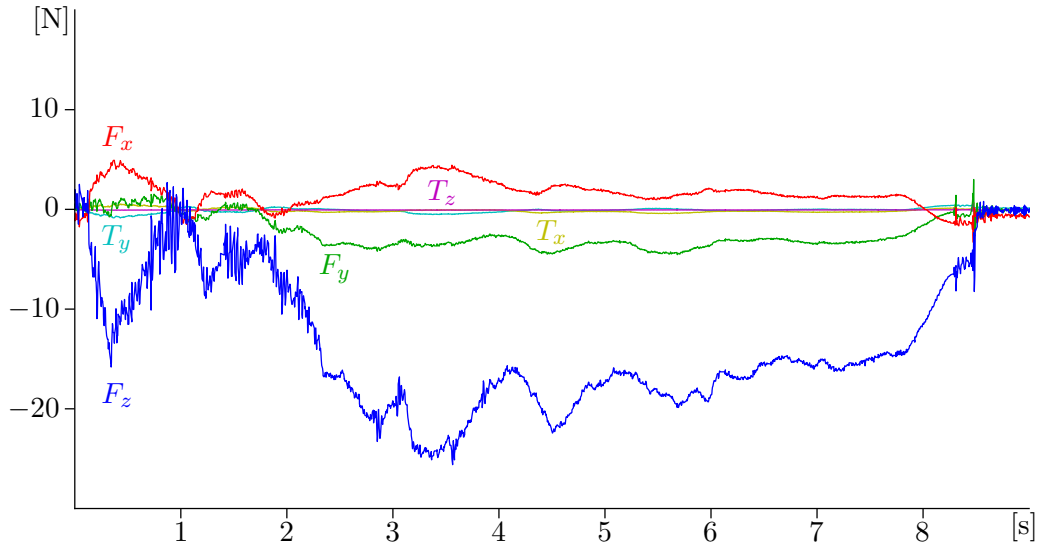


Figure 6.13.: Forces and torques acting on one of the h6 pegs during insertion

position of the robot. With that in mind, the precision of 0.058 mm can be assumed.

The robot where able to complete the entire eTaSL peg-in-hole task for all the three sets of pegs. All pegs where placed without getting stuck or hitting the block. The entire task took about 1 min 30 sec to complete.

Even though the task where finished, there is one point during the task where the robot vibrates at a high frequency. This is the same spot for all runs. There is also some noticeable wobbles once in a while.

In Figure 6.13 a plot of the forces and torques acting on the tip of the peg during insertion is given. This is with one of the ⌀20h6 mm pegs. The forces acting on the other h6 pegs where similar but with varying magnitudes. The forces acting on the other sets of pegs where insignificant.

Chapter 7.

Discussion

In this chapter, the results of the three different tests outlined in chapter 6.

7.1. Switching Times in eTaSL ROS Controller

To be able to react to sensor inputs, as a reading of the force/torque sensor on the robot, there are real-time requirements to be met. For instance, if the controller is not able to switch fast enough in case of a high force measurement, the actual force acting before the task is switched can be at a damaging level. In the case of KUKA RSI, its real-time capability pushes 250 Hz. That is, one executed command every 4 ms. Real-time control is a challenge to ROS, more so, being able to switch between subtasks as fast as any other controller tick is something it is not capable of yet.

The different switching methods were timed and plotted with their average in Figure 6.1, 6.2 and 6.3 from the results in section 6.2. In each run of the peg-in-hole task, the switching of subtasks happens 25 times with monitors and 10 times with services. Hence, three times that is shown in the two last figures. In the first figure, an other eTaSL task was used when timing the switch controller service, but the task is irrelevant for the time consumption here.

Looking at the "*switch controller*"-method from Figure 6.1 first, we can see that the average is 7.5 ms and not a single switch is capable of getting beneath 4.0 ms which is clearly not good enough.

The "*ROS controller service*"-method from Figure 6.2 is faster, but with an average of 5.3 ms and only 6 out of 30 times at or beneath 4.0 ms it is still not fast enough. A reason for the better performance of this method compared to the previous, is that this method is only activating and not deactivating tasks. It will therefore not need to use time on finding out which controllers to stop. This does also mean that it is not really necessary for this method to be below 4.0 ms, as this is only used when the robot is starting after standstill.

The "*monitor*"-method from Figure 6.3, is clearly the fastest method, with an average of 2.3 ms. In most cases this would be good enough, but as 7 instances are above 4.0 ms, this method are not mission critical proof. It seems like there are two clear separate groupings of switching times. This is most likely due to the fact that every two out of five switches are actually turning all active tasks off, and not activating any. Which would be less demanding to initiate.

7.2. Robot Behaviour with eTaSL

The task function for a control task in eTaSL is, as mentioned in section 3.3, a first order system, which means that the system evolves in an exponential manner towards its goal. This was illustrated in Figure 3.3 and has particularly one significant challenge in use with geometrical constraints. A first order system will follow an input with small deviations really well, but in case of constraints where the deviation between the goal and the initial state are large, it will try to evolve towards the goal really fast in the first part of the movement. What that does, in physical terms, is that if for instance the robot end-effector is far of its goal position, the velocities applied to the joints will create an acceleration a real robot cannot cope with. This is exactly what happens when the peg-in-hole task starts from "*home position*" and are lining up for the first peg. The velocities applied in the first split seconds, creates an almost infinite acceleration. The plots in Figure 6.4 from the results in section 6.4 shows the joint positions and velocities when the simulated robot is lining up for the fist peg where no limitations on velocities or accelerations are set. Here we can clearly see the exponential behaviour of the first order system.

This exponential behaviour is the main reason why constraints based solely on geometrical relations are a hard task. During the work of this thesis, different methods of solving that problem has been tested. The KUKA RSI interface takes

by default incremental joint angle commands, hence the hardware interface was originally set up as a joint position interface (`PositionJointInterface`). The `joint_limits_interface` of the hardware interface lets you apply joint velocity limits to a position controlled interface. This helps in restraining the robot maximum velocity, but are not able to slow down the acceleration.

eTaSL is originally set up to solve for joint velocities in each solver step, hence the position interface of RSI, where possibly limiting optimal behaviour of eTaSL. A solution to that, became to rewrite both the hardware interface, and the RSI interface on the robot controller. By implementing a `VelocityJointInterface`, the behaviour shown in Figure 6.5 and 6.6 where obtained on the physical robot. The `VelocityJointInterface` makes it possible to apply acceleration limits as well as position limits to the joints. The results given in Figure 6.5 and 6.6 where obtained by applying the limits given in Listing 7.1. Compared to the initial simulated behaviour in Figure 6.4, where eTaSL was free to output any joint angle and velocity, we can clearly see now that the acceleration are of a magnitude that makes more sense for a physical structure.

Listing 7.1: joint_limits.yaml

```
joint_limits:
  joint_a1:
    max_velocity: 0.6283185307179586
    max_acceleration: 10.0
  joint_a2:
    max_velocity: 0.5235987755982989
    max_acceleration: 10.0
  joint_a3:
    max_velocity: 0.6283185307179586
    max_acceleration: 10.0
  joint_a4:
    max_velocity: 0.6649704450098396
    max_acceleration: 10.0
  joint_a5:
    max_velocity: 0.6771877497737998
    max_acceleration: 10.0
  joint_a6:
    max_velocity: 1.0733774899765127
    max_acceleration: 10.0
```

Comparing the joint velocity plot and the joint limits, we can see that the robot adheres to the limits. The slope of the velocities are flattened according to the acceleration limits, and the joint velocities are capped off at the velocity limits. In this case, this can be seen in the beginning of the trajectory for joint q_1. The rest

of the joints never reach their limit. This method works, but has its drawbacks. That is, this method makes eTaSL and the hardware interface fighting each other over the joint velocities for the robot. eTaSL will try to impose joint velocities that makes the robot behave as a first order system, while the hardware interface shuts eTaSL down in its attempt every time they do not adhere to the applied limits.

The fighting between eTaSL and the hardware controller, seems to make the robot behave strangely at times. Firstly, the limits needs some tuning for the robot to behave. When there is a large difference between the velocity limits and the acceleration limits, the robot starts to wobble. When the acceleration is set too low, the robot starts to overshoot the constraints, since it is not able to decelerate fast enough. When all velocity limits are set to the same value, some constraints will get ahead of others, as there are different efforts needed to obtain the same velocity for different joints. Thus, any limiting of joint velocities has to be proportional to their robot specific joint velocity limits.

The plots of Figure 6.7 shows the Cartesian coordinates of the gripper in the upper plot, and the corresponding joint angles, for some 5 seconds during the peg-in-hole task with the final implementation of the controller. Both in plots and while looking at the robot, it now seems to run quite smooth. But, there are still some issues that are hard to explain. Two of those instances are marked with the vertical dotted lines, but more can be seen. What happens, is that the robot suddenly shifts a little bit off course before returning immediately. This happens immediately after switching of subtasks, where the applied velocities increases again. Most surprising are the wobbles in joint q_4, as it should theoretically never move with the set of constraints applied to the robot in this task.

7.3. Peg-In-Hole Test

The robot was able to complete the peg-in-hole task for all the three sets of pegs. The two first where expected, the last where a little bit more surprising. The reason why it where able also with the tightest tolerance is most likely due to the fact that all pegs and all holes are chamfered. That is, a small trimming of the edge of about 0.5 mm. An illustration can be seen in Figure 7.1. What the chamfer allows is that if the peg is a off by less than 1.0 mm when contacting the hole, and there is some flex in the system, the peg can be lined up properly

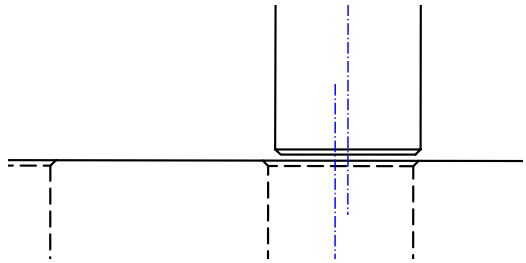


Figure 7.1.: Possible deviation with chamfer

with the hole. In this test, the most likely source of flex is the table at which the assembly happens and the fact that the bar with the holes where not fixed to the table. Even so, there where no noticeable shift in the bar during the tests.

Another thing we can draw from this test is that the deviation in the alignment angle of the peg must have been insignificant. When trying to force it in by hand, any angle more than 1 degree would make the peg jam. We can see in Figure 6.13, from the results in section 6.8, that the pegs with the tightest tolerance where not running freely during insertion. Thus, some small deviations in position and orientation of the pegs where most likely present after all.

Even though the positioning of the pegs are really good in the xy-plane, there are some clearly visible deviations in the positioning of the grippers along the approach axis z. It seemed to vary with up to aprx. 3 mm. The most likely reason why the xy positioning is almost perfect while z is quite bad, is that eTaSL has far longer time to converge to the `Concentric` constraint than the `Coincident_plane_plane` constraint. But, the monitor that monitors the "*pick up*" and "*insertion*" is set to not trigger before the distance between the planes are less than 1 mm. In this test, it was not really important that the positioning along z is perfect. But, a better positioning could be achieved by having a lower bound on the monitor. This comes at cost of time. And, a robot may never be able to reach the bound if the bound is too low, because of encoder errors in the robot joints.

The vibration of the robot during the testing is a known problem. Further investigation showed that the vibration would happen when the robot where stretched out at a radius in the region between 600 to 700 mm from the base frame. Any further out than that, and the robot would more often than not vibrate before shutting out with the error "*Command gear torque A1.*" The error message means

that the commanded torque to the joint is to high, or not smooth enough. The issue is twofold, and related to RSI. eTaSL has a tendency, as described in the previous section, to command joint velocities that has an acceleration that the robot can not cope with. Before applying joint limits to the commanded joint positions, the robot would stop with the gear torque error at the moment the robot started. Even though this behaviour has been restrained, there is still some irregularities in the commands. This is the first part of the issue. The second is the way RSI is used with ROS and how it does not comply with the dynamic models of KUKA.

When using RSI with ROS, only incremental joint positions are commanded, and the robot controller will try to execute that command in 4 ms as described in section 3.5. But the motors at the joints are not controlled by joint positions, its controlled by an applied current that produces a desired torque. For the controller to know how much torque to apply, it needs a good dynamical model for the robot. Without precise data on the weight, center of mass, and inertia matrix of the force/torque sensor and gripper combination, the controller will output the wrong torque to the joints. Thus, resulting in vibrations and gear torque errors. The gear torque errors stems from a contradiction between where the robot believes it should be after applying some torque and its actual position after applying that torque.

7.4. Industrial Relevance

eTaSL ROS Control with CAD-constraints has the possibility to be really useful for the production industry. But as seen through testing, there is some instability of the controller that has to be resolved first. But when that is solved, there is some real benefits to this system.

First of all, is the eTaSL ROS controller interchangeable between all robot brands that has a ROS control hardware interface. As of now, that is KUKA and Universal Robotics (UR), but that can change over time. One should be able to use the controller directly with any KUKA robot sat up with RSI capability. Secondly, does eTaSL, with CAD-constraints, allow for a short transition from the design phase to the actual robot assembly.

There are some requirements. Any business wanting to implement eTaSL in their

factory will need a robot programmer with knowledge in ROS, as well as sensor systems that can be implemented in ROS. ROS is also, as of now, only available for the Ubuntu platform, and eTaSL is restricted to Ubuntu 16.04 and ROS Kinetic. Thus, the controller PC communicating with the robot controller has to be running on Ubuntu.

The solution for extracting CAD-constraints are dependent on SolidWorks. If the business uses anything other than SolidWorks, they would not be able to extract the constraint information as done here. A vision for the future is that this CAD-information could be extracted from STEP information instead, which is a vendor independent file format for CAD-data. That STEP file could then be opened in a program which loads the CAD-constraints and the robot cell information and generates a finished assembly task ready for execution. This program would not need to know the position of the parts to assembly, as the position can be an input from 3D-cameras.

Chapter 8.

Conclusion

This thesis began by giving an introduction to robot kinematic and screw theory, before presenting the necessary software tools for the programming in this thesis, such as eTaSL, ROS, SMACH and KUKA RSI. Then the constraint formulation of SolidWorks where presented, with the belonging mathematics.

The aim of this thesis have been to discover whether it was possible or not to program a robot solely by information contained in CAD-constraints and if it could be implemented on a commercially available off-the-shelf industrial robot manipulator. In chapter 5, it was shown how this where solved during the work on this thesis by a ROS implementation of eTaSL and constraints from SolidWorks. The CAD-data is collected from SolidWorks by a C# code using their API, before they are used with the geometrical functions presented in section 5.2 to define constraints in the eTaSL ROS controller. It has been showed that it actually is possible to program a robot solely by CAD-constraints, and that it can be implemented on a commercially available off-the-shelf industrial robot. That is, at least the KUKA KR6 R900 robot and other KUKA robots. The most important discovery made to make it work on the physical robot where the `joint_limits_interface` of ROS control. By applying joint limits, it is possible to restrain the exponential behaviour of eTaSL to give feasible joint commands to the robot.

Through a set of tests presented in chapter 6, it has been showed how the formulation of a CAD constraint based task for a peg-in-hole test case where done. In the test case five pegs where successfully inserted in five different holes by the KUKA KR6 R900 robot in less than 1.5 minutes. The pegs placed had a tight tolerance that did not allow for a positioning error larger than ± 0.016 mm on average. Still,

the pegs where successfully placed with only a small insertion force of 25 N. But, even though the robot completes the test case, there are some irregularities in the trajectories that have not been fully understood. There are some vibrations that stems from combined faults in eTaSL and the ROS implementation for KUKA RSI. There are also some wobbles and unwanted behaviour by the robot, in which the source is still unidentified.

The fact that the assembly program is implemented on a KUKA robot, without any modifications to the robot controller, and that it is not done on some hightech research robot, makes this project relevant to the industry. With some knowledge in python, C++ and ROS, the eTaSL ROS controller should be relatively easily implemented on any KUKA or UR robot. After the system is up an running, the implementation of a new assembly task will probably be faster than it can be done today.

This work is still in the early stages of what CAD-constraint based programming can be. Never the less, does it lead the way for a promising future within industrial robot programming. Constraint based robot programming with CAD-data has the possibility to remove almost all human intervention between product design and fully assembled parts. This means that concurrent product development is even easier, in the way that product refinements can be easier tested in its finished state. This is really important in reducing the cost of product changes. It also means that the time cost of giving costumers the possibility of product customization is reduced, giving more satisfied consumers, increasing their willingness to pay, without noticeable increase in cost.

8.1. Future Work

Future work should continue to try to find the cause of the small wobbles of the robot. It should also debug the reason for the early exits that happens once in a while during initiation of a new task when switching.

eTaSL allows for the use of sensor data of all kind. This possibility has not been utilized in this work. Future work could try to find out the possibilities of sensor data in an assembly task. For instance for force controlled assembly, safety measures, etc. The intention of using a 3D-camera for capturing the position of the parts where also abandoned. That should be solved for a more robust system.

A late discovery of the possibility of different controllers within eTaSL, made it only possible to briefly test an implementation before the end of the project. Future work should seek to find out how the `ControllerProportionalSaturated`-class, within the `controller.cpp` script, can help restraining the robot when constrained by CAD-constraints. It should also find out if a self developed controller class would be a better solution.

To make the eTaSL ROS Controller with CAD-constraints even more relevant to the industry, two main developments should be made. That would be to make the CAD-information independent on SolidWorks. In other words, independent on CAD-software vendors. A suggestion is to find the possibilities of extracting geometrical constraint data from the ISO STEP-format. The other thing, would be to find a way to generate all necessary eTaSL constraints automatically from the STEP information, or at least a semi-automatic solution. This would make the eTaSL ROS Controller even more relevant because of less need for knowledgeable programmers and shorter time used at implementation.

References

[1] E. Aertbeliën. *control, constraints, model, measurement*. The constraints function within eTaSL and its semantics and relation to control. Sept. 15, 2016. URL: https://etasl.pages.mech.kuleuven.be/appnotes/appnote0.html (visited on 05/01/2019).

[2] E. Aertbeliën. *What eTaSL/eTC is and is not*. Version 1.3. 2016. URL: https://etasl.pages.mech.kuleuven.be/intro.html (visited on 06/03/2019).

[3] E. Aertbeliën and J. De Schutter. "eTaSL/eTC: A constraint-based task specification language and robot controller using expression graphs". In: *2014 IEEE/RSJ International Conference on Intelligent Robots and Systems*. 2014, pp. 1540–1546. DOI: 10.1109/IROS.2014.6942760.

[4] D. Agrawal and S. S. Pande. "Automatic disassembly sequence planning and optimization". In: *Dual Degree Thesis, IIT Bombay* (2011).

[5] M. H. Arbo, Y. Pane, E. Aertbeliën, and W. Deeré. "A System Architecture for Constraint-Based Robotic Assembly with CAD Information". In: *2018 IEEE 14th International Conference on Automation Science and Engineering (CASE)*. Aug. 2018, pp. 690–696. DOI: 10.1109/COASE.2018.8560450.

[6] C. F. Chen. "Assembly planning - a genetic approach". In: *ASME Design Engineering Technical Conferences*. DETC98/DAC-5798. 1998.

[7] Dassault Systèmes SolidWorks Corporation. *SolidWorks*. Version Student Edition - Academic Year 2018-2019. Jan. 29, 2019. URL: https://www.solidworks.com/.

[8] J. Denavit and R. S. Hartenberg. "A kinematic notation for lower-pair mechanisms based on matrices". In: *ASME Journal of Applied Mechanics* 22 (1955), pp. 215–221.

[9] B. Drost, M. Ulrich, N. Navab, and S. Ilic. "Model globally, match locally: Efficient and robust 3D object recognition". eng. In: IEEE Publishing, 2010, pp. 998–1005. ISBN: 9781424469840.

[10] O. Egeland. *TPK4171 Advanced Industrial Robotics*. Tech. rep. Department of Mechanical and Industrial Engineering, May 2018.

[11] *EntityParams Property (IMateEntity2)*. 2019. URL: http://help.solidworks.com/2019/english/api/sldworksapi/solidworks.interop.sldworks~solidworks.interop.sldworks.imateentity2~entityparams.html?verRedirect=1 (visited on 06/14/2019).

[12] *Experimental packages for KUKA manipulators within ROS-Industrial*. Version "velocity-interface". May 13, 2019. URL: https://github.com/ra-mtp-ntnu/kuka_experimental.

[13] *F/T Sensor: Gamma IP60*. 2019. URL: https://www.ati-ia.com/products/ft/ft_models.aspx?id=Gamma+IP60 (visited on 06/14/2019).

[14] *F/T Transducer – Six-Axis Force/Torque Sensor System – Installation and Operation Manual*. 9620-05-Transducer Section. ATI Industrial Automation. 2019. URL: https://www.ati-ia.com/app_content/documents/9620-05-Transducer%20Section.pdf (visited on 06/02/2019).

[15] Q. Gao, J. Zhao, and M. Wang. "Research of peg - in - hole assembly for a redundant dual-arm robot based on neural networks". In: *2017 IEEE International Conference on Mechatronics and Automation (ICMA)*. Aug. 2017, pp. 252–257. DOI: 10.1109/ICMA.2017.8015823.

[16] *Instruction manual*. Robotiq. 2019. URL: https://assets.robotiq.com/website-assets/support_documents/document/2F-85_2F-140_Instruction_Manual_Gen_PDF_20190524.pdf?_ga=2.170612320.1735060602.1560935758-86300000.1554121293 (visited on 06/19/2019).

[17] ISO Central Secretary. *Geometrical product specifications (GPS) – ISO code system for tolerances on linear sizes – Part 2: Tables of standard tolerance classes and limit deviations for holes and shafts*. EN. Standard ISO 286-2:2010. Geneva, CH: International Organization for Standardization, 2010. URL: https://www.iso.org/standard/54915.html.

[18] W. Ji, S. Yin, and L. Wang. "A Virtual Training Based Programming-Free Automatic Assembly Approach for Future Industry". In: *IEEE Access* 6 (2018), pp. 43865–43873. ISSN: 2169-3536. DOI: 10.1109/ACCESS.2018.2863697.

[19] *Kinematic Pair*. 2019. URL: https://en.wikipedia.org/wiki/Kinematic_pair (visited on 06/14/2019).

[20] *KR 6 R900 sixx*. V22.1. KUKA Deutschland GmbH. 2018. URL: https://www.kuka.com/-/media/kuka-downloads/imported/6b77eecacfe542d3b736af377562ecaa/0000205456_en.pdf (visited on 06/14/2019).

[21] D. Lofthus. *CAD to eTaSL*. Version "master". Jan. 2019. URL: https://github.com/daglofthus/CADtoETASL.

[22] D. Lofthus. *geometric3.lua*. Version "master". May 2019. URL: https://github.com/daglofthus/geometric3.

[23] D. Lofthus. *peg_in_hole.lua*. June 2019. URL: https://github.com/daglofthus/etasl_ros_control/blob/develope/etasl_ros_control_cmd/scripts/peg_in_hole.lua.

[24] N. Mansard and F. Chaumette. "Task Sequencing for High-Level Sensor-Based Control". In: *IEEE Transactions on Robotics* 23.1 (2007), pp. 60–72. ISSN: 1552-3098. DOI: 10.1109/TRO.2006.889487.

[25] A. T. Mathew and C. S. P. Rao. "A novel method of using API to generate liaison relationships from an assembly.(Applications Programming Interface)(Report)". English. In: *Journal of Software Engineering and Applications (JSEA)* 3.2 (2010). ISSN: 1945-3116.

[26] A. Mathew and C. S. P Rao. "A CAD system for extraction of mating features in an assembly". eng. In: *Assembly Automation* 30.2 (2010), pp. 142–146. ISSN: 0144-5154.

[27] Microsoft. *.NET Framework*. Version 4.7.02558. Dec. 11, 2018. URL: https://dotnet.microsoft.com/.

[28] C. Pan, S. S. Smith, and G. C. Smith. "Automatic assembly sequence planning from STEP CAD files". In: *International Journal of Computer Integrated Manufacturing* 19.8 (2006), pp. 775–783. DOI: 10.1080/09511920500399425. eprint: https://doi.org/10.1080/09511920500399425. URL: https://jsdai.net/.

[29] A. Perzylo, N. Somani, S. Profanter, I. Kessler, M. Rickert, and A. Knoll. "Intuitive instruction of industrial robots: Semantic process descriptions for small lot production". In: *2016 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. Oct. 2016, pp. 2293–2300. DOI: 10.1109/IROS.2016.7759358.

[30] A. Perzylo, N. Somani, M. Rickert, and A. Knoll. "An ontology for CAD data and geometric constraints as a link between product models and semantic robot task descriptions". In: *2015 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. Sept. 2015, pp. 4197–4203. DOI: 10.1109/IROS.2015.7353971.

[31] *Quick User Guide*. 01. Zivid Labs AS. 2019. URL: https://www.zivid.com/hubfs/softwarefiles/Zivid-Quick-User-Guide.pdf (visited on 06/02/2019).

[32] *ROS*. URL: https://www.ros.org/about-ros/ (visited on 05/01/2019).

[33] B. Siciliano, L. Sciavicco, L. Villani, and G. Oriolo. *Robotics: Modelling, Planning and Control*. eng. Advanced Textbooks in Control and Signal Processing. London: Springer London, 2009. ISBN: 978-1-84628-641-4.

[34] R. Smits, T. De Laet, K. Claes, H. Bruyninckx, and J. De Schutter. "iTASC: A Tool for Multi-Sensor Integration in Robot Manipulation". In: *Multisensor Fusion and Integration for Intelligent Systems: An Edition of the Selected Papers from the IEEE International Conference on Multisensor Fusion and Integration for Intelligent Systems 2008*. Ed. by H. Hahn, H. Ko, and S. Lee. Springer Berlin Heidelberg, 2009, pp. 235–254. ISBN: 978-3-540-89859-7. DOI: 10.1007/978-3-540-89859-7_17. URL: https://doi.org/10.1007/978-3-540-89859-7_17.

[35] Ruben Smits. *The Kinematics and Dynamics Library*. Version 1.0.2. Feb. 25, 2010. URL: http://www.orocos.org/wiki/orocos/kdl-wiki (visited on 06/15/2019).

[36] N. Somani, A. Gaschler, M. Rickert, A. Perzylo, and A. Knoll. "Constraint-based task programming with CAD semantics: From intuitive specification to real-time control". In: *2015 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. Sept. 2015, pp. 2854–2859. DOI: 10.1109/IROS.2015.7353770.

[37] N. Somani, M. Rickert, and A. Knoll. "An Exact Solver for Geometric Constraints With Inequalities". In: *IEEE Robotics and Automation Letters* 2.2 (Apr. 2017), pp. 1148–1155. ISSN: 2377-3766. DOI: 10.1109/LRA.2017.2655113.

[38] *Spez KR AGILUS sixx*. V14. KUKA Deutschland GmbH. 2018. URL: https://www.kuka.com/-/media/kuka-downloads/imported/48ec812b1b2947898ac2598aff70abc0/spez_kr_agilus_sixx_en.pdf?modified=1052831294 (visited on 06/14/2019).

[39] L. Tingelstad and D. Lofthus. *kuka_rsi_hw_interface*. Version "velocity-interface". May 2019. URL: https://github.com/ra-mtp-ntnu/kuka_experimental/tree/velocity-interface/kuka_rsi_hw_interface.

[40] L. Tingelstad, D. Lofthus, and M. Arbo. *eTaSL ROS Control*. Version "develope". June 2019. URL: https://github.com/daglofthus/etasl_ros_control/tree/develope.

[41] *Zivid One*. The original. 2019. URL: https://www.zivid.com/zivid-one (visited on 06/14/2019).

Appendix A.

Constraints in geometric3.lua

Listing A.1: Constraint formulation for two coincident points

```
Coincident_point_point{
    context  = ctx,
    name     = "name"[optional],
    K        = (scalar)[optional],
    weight   = (scalar)[optional],
    priority = (1 or 2)[2 default],
    point_a  = (vector expression)[first point],
    point_b  = (vector expression)[second point]
}
```

Listing A.2: Constraint formulation for a point coincident with a line

```
Coincident_point_line{
    context  = ctx,
    name     = "name"[optional],
    K        = (scalar)[optional],
    weight   = (scalar)[optional],
    priority = (1 or 2)[2 default],
    point_a  = (vector expression)[the point],
    point_b  = (vector expression)[point on the line],
    dir_b    = (vector expression)[unit vector along the line]
}
```

Listing A.3: Constraint formulation for a point coincident with a plane

```
Coincident_point_plane{
    context  = ctx,
    name     = "name"[optional],
    K        = (scalar)[optional],
    weight   = (scalar)[optional],
    priority = (1 or 2)[2 default],
    point_a  = (vector expression)[the point],
    point_b  = (vector expression)[point on the plane],
    dir_b    = (vector expression)[unit normal vector of plane]
}
```

Listing A.4: Constraint formulation for two coincident lines

```
Coincident_line_line{
    context  = ctx,
    name     = "name"[optional],
    K        = (scalar)[optional],
    weight   = (scalar)[optional],
    priority = (1 or 2)[2 default],
    point_a  = (vector expression)[point on one line],
    dir_a    = (vector expression)[unit vector along that line],
    point_b  = (vector expression)[point on other line],
    dir_b    = (vector expression)[unit vector along that line]
}
```

Listing A.5: Constraint formulation for a line coincident with a plane

```
Coincident_line_plane{
    context  = ctx,
    name     = "name"[optional],
    K        = (scalar)[optional],
    weight   = (scalar)[optional],
    priority = (1 or 2)[2 default],
    point_a  = (vector expression)[point on the line],
    dir_a    = (vector expression)[unit vector along the line],
    point_b  = (vector expression)[point on the plane],
    dir_b    = (vector expression)[unit normal vector of plane]
}
```

Listing A.6: Constraint formulation for two coincident planes

```
Coincident_plane_plane{
    context  = ctx,
    name     = "name"[optional],
    K        = (scalar)[optional],
    weight   = (scalar)[optional],
    priority = (1 or 2)[2 default],
    point_a  = (vector expression)[point on one plane],
    dir_a    = (vector expression)[unit normal vector of that plane],
    point_b  = (vector expression)[point on other plane],
    dir_b    = (vector expression)[unit normal vector of that plane]
}
```

Listing A.7: Constraint formulation for concentricity

```
Concentric{
    context  = ctx,
    name     = "name"[optional],
    K        = (scalar)[optional],
    weight   = (scalar)[optional],
    priority = (1 or 2)[2 default],
    point_a  = (vector expression)[point on one line],
    dir_a    = (vector expression)[unit vector along that line],
    point_b  = (vector expression)[point on other line],
    dir_b    = (vector expression)[unit vector along that line]
}
```

Listing A.8: Constraint formulation for two perpendicular lines

```
Perpendicular_line_line{
    context  = ctx,
    name     = "name"[optional],
    K        = (scalar)[optional],
    weight   = (scalar)[optional],
    priority = (1 or 2)[2 default],
    point_a  = (vector expression)[point on one line],
    dir_a    = (vector expression)[unit vector along that line],
    point_b  = (vector expression)[point on other line],
    dir_b    = (vector expression)[unit vector along that line]
}
```

Listing A.9: Constraint formulation for a line perpendicular to a plane

```
Perpendicular_line_plane{
    context  = ctx,
    name     = "name"[optional],
    K        = (scalar)[optional],
    weight   = (scalar)[optional],
    priority = (1 or 2)[2 default],
    point_a  = (vector expression)[point on the line],
    dir_a    = (vector expression)[unit vector along the line],
    point_b  = (vector expression)[point on the plane],
    dir_b    = (vector expression)[unit normal vector of plane]
}
```

Listing A.10: Constraint formulation for two perpendicular planes

```
Perpendicular_plane_plane{
    context  = ctx,
    name     = "name"[optional],
    K        = (scalar)[optional],
    weight   = (scalar)[optional],
    priority = (1 or 2)[2 default],
    point_a  = (vector expression)[point on one plane],
    dir_a    = (vector expression)[unit normal vector of that plane],
    point_b  = (vector expression)[point on other plane],
    dir_b    = (vector expression)[unit normal vector of that plane]
}
```

Listing A.11: Constraint formulation for two parallel lines

```
Parallel_line_line{
    context  = ctx,
    name     = "name"[optional],
    K        = (scalar)[optional],
    weight   = (scalar)[optional],
    priority = (1 or 2)[2 default],
    point_a  = (vector expression)[point on one line],
    dir_a    = (vector expression)[unit vector along that line],
    point_b  = (vector expression)[point on other line],
    dir_b    = (vector expression)[unit vector along that line]
}
```

Listing A.12: Constraint formulation for a line parallel to a plane

```
Parallel_line_plane{
    context  = ctx,
    name     = "name"[optional],
    K        = (scalar)[optional],
    weight   = (scalar)[optional],
    priority = (1 or 2)[2 default],
    point_a  = (vector expression)[point on the line],
    dir_a    = (vector expression)[unit vector along the line],
    point_b  = (vector expression)[point on the plane],
    dir_b    = (vector expression)[unit normal vector of plane]
}
```

Listing A.13: Constraint formulation for two parallel planes

```
Parallel_plane_plane{
    context  = ctx,
    name     = "name"[optional],
    K        = (scalar)[optional],
    weight   = (scalar)[optional],
    priority = (1 or 2)[2 default],
    point_a  = (vector expression)[point on one plane],
    dir_a    = (vector expression)[unit normal vector of that plane],
    point_b  = (vector expression)[point on other plane],
    dir_b    = (vector expression)[unit normal vector of that plane]
}
```

Listing A.14: Constraint formulation for a line tangent to a cylinder

```
Tangent_line_cylinder{
    context  = ctx,
    name     = "name"[optional],
    K        = (scalar)[optional],
    weight   = (scalar)[optional],
    priority = (1 or 2)[2 default],
    point_a  = (vector expression)[point on the line],
    dir_a    = (vector expression)[unit vector along the line],
    point_b  = (vector expression)[point on center line],
    dir_b    = (vector expression)[unit vector along the center line],
    r        = (positive scalar)[radius of cylinder]
}
```

Listing A.15: Constraint formulation for a plane tangent to a cylinder

```
Tangent_plane_cylinder{
    context  = ctx,
    name     = "name"[optional],
    K        = (scalar)[optional],
    weight   = (scalar)[optional],
    priority = (1 or 2)[2 default],
    point_a  = (vector expression)[point on the plane],
    dir_a    = (vector expression)[unit normal vector of plane],
    point_b  = (vector expression)[point on center line],
    dir_b    = (vector expression)[unit vector along the center line],
    r        = (positive scalar)[radius of cylinder]
}
```

Listing A.16: Constraint formulation for a line tangent to a sphere

```
Tangent_line_sphere{
    context  = ctx,
    name     = "name"[optional],
    K        = (scalar)[optional],
    weight   = (scalar)[optional],
    priority = (1 or 2)[2 default],
    point_a  = (vector expression)[point on the line],
    dir_a    = (vector expression)[unit vector along the line],
    center   = (vector expression)[center point],
    r        = (positive scalar)[radius of sphere]
}
```

Listing A.17: Constraint formulation for a plane tangent to a sphere

```
Tangent_plane_sphere{
    context  = ctx,
    name     = "name"[optional],
    K        = (scalar)[optional],
    weight   = (scalar)[optional],
    priority = (1 or 2)[2 default],
    point_a  = (vector expression)[point on the plane],
    dir_a    = (vector expression)[unit normal vector of plane],
    center   = (vector expression)[center point],
    r        = (positive scalar)[radius of sphere]
}
```

Appendix B.

Running Peg-In-Hole with eTaSL

Build etasl_ros_control from source

For now, etasl_ros_control is only supported on Linux and Ubuntu 16.04 / ROS Kinetic

Follow all the instructions to install ROS Kinetic. Please make sure you have followed all steps and have the latest versions of packages installed:

```
$ rosdep update
$ sudo apt-get update
$ sudo apt-get dist-upgrade
```

Source installation requires wstool:

```
$ sudo apt-get install python-wstool
```

Optionally create a new workspace, you can name it whatever:

```
$ mkdir ~/etasl_ros_control_ws
$ cd ~/etasl_ros_control_ws
```

Next, source your ROS workspace to load the necessary environment variables.

```
$ source /opt/ros/kinetic/setup.bash
```

This will load the `${ROS_DISTRO}` variable, needed for the next step.

Download and build etasl_ros_control

For now, the examples described in this thesis is only available at the develop branch. But will hopefully be available at the master branch later on. The url

```
https://raw.githubusercontent.com/daglofthus/etasl_ros_control/develope/
    etasl_ros_control.rosinstall
```

can then be exchanged by

```
https://raw.githubusercontent.com/tingelst/etasl_ros_control/master/
    etasl_ros_control.rosinstall
```

Pull down the required repositories and build from within the root directory of your catkin workspace:

```
$ wstool init src
$ wstool merge -t src https://raw.githubusercontent.com/daglofthus/
    etasl_ros_control/develope/etasl_ros_control.rosinstall
$ wstool update -t src
$ rosdep install -y --from-paths src --ignore-src --rosdistro
    ${ROS_DISTRO}
$ catkin_init_workspace src
$ catkin_make --cmake-args -DCMAKE_BUILD_TYPE=Release
```

Source the catkin workspace

Setup your environment - you can do this every time you work with this particular source install of the code, or you can add this to your .bashrc (recommended):

```
$ source ~/etasl_ros_control_ws/devel/setup.bash
```

Running Peg-In-Hole Example

For simulation, run in terminal:

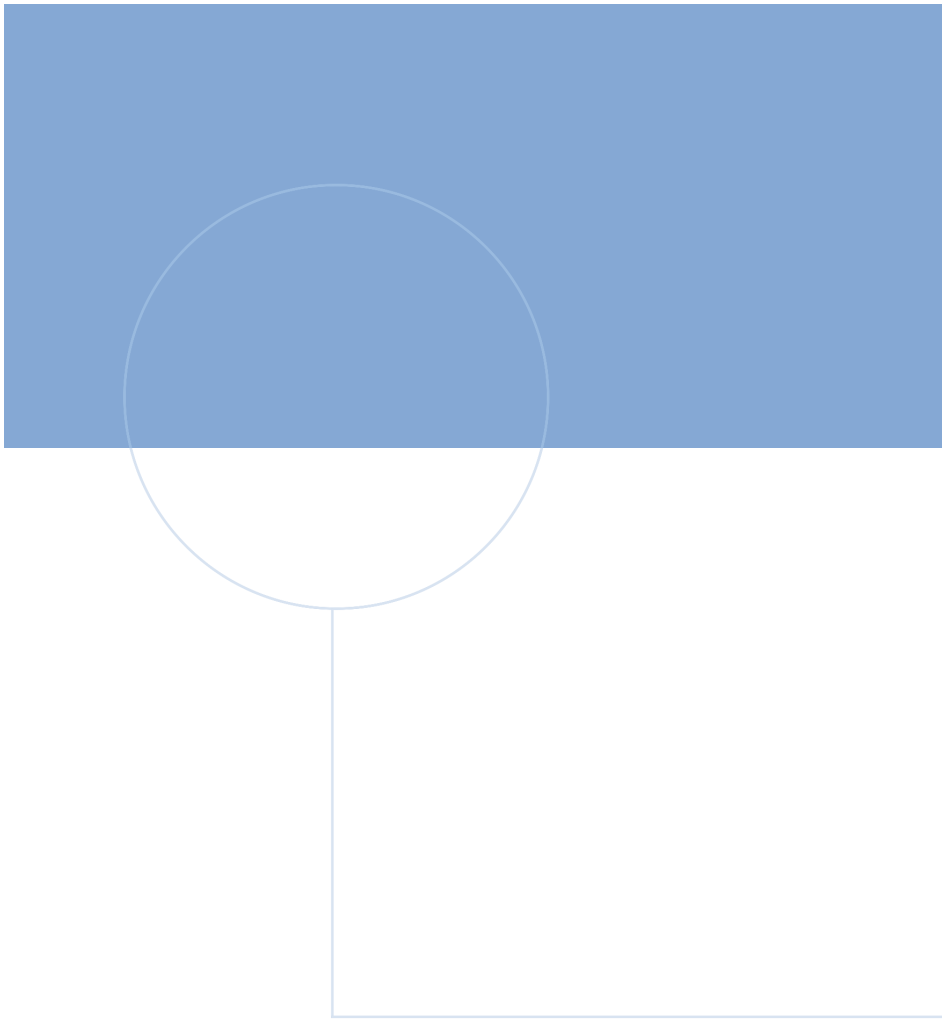
```
$ roslaunch etasl_ros_control_cmd smach_peg.launch
```

To run with a higher gain on all constraints (i.e. 5x faster):

```
$ roslaunch etasl_ros_control_cmd smach_peg.launch speed:=5.0
```

When not simulating, and connected to force sensor and gripper:

```
$ roslaunch etasl_ros_control_cmd smach_peg.launch sim:=false
    connected:=true
```



NTNU

Norwegian University of
Science and Technology