Master's thesis

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

Vebjørn Bergsholm Bjørhovde

Performance Testing Real-Time Robot Communication for a Constraint-Based Robotic Welding System

Master's thesis in Robotics and Automation
Supervisor: Lars Tingelstad
June 2021

NTNU
Norwegian University of
Science and Technology

Vebjørn Bergsholm Bjørhovde

Performance Testing Real-Time Robot Communication for a Constraint-Based Robotic Welding System

Master's thesis in Robotics and Automation
Supervisor: Lars Tingelstad
June 2021

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

Performance Testing Real-Time Robot Communication for a Constraint-Based Robotic Welding System

Vebjørn Bergsholm Bjørhovde

2021-06-09

Preface

This is a master's thesis written by Vebjørn Bergsholm Bjørhovde under the supervision of Lars Tingelstad during the spring of 2021. The thesis is written for the Department of Mechanical and Industrial Engineering at the Norwegian University of Science and Technology (NTNU), Trondheim.

Acknowledgements

First and foremost I would like to thank my supervisor Lars Tingelstad for support and guidance while working on this thesis. Our weekly meetings have given me motivation and direction while working on a large subject with many unknown challenges.

I would also like to thank Mathias Hauan Arbo who unofficially has acted as a co-supervisor on this thesis. He has been helpful in answering any questions I had on control theory and latency in robotics.

Setting up the robot cell for welding would also be a lot more difficult without the help of Nikolai Marhaug, Morten Høgseth Danielsen and Børge Holen. I would like to thank them all for the technical support regarding welding in general as well as using robotics in welding.

My thanks also to Adam Leon Kleppe for support on all lab-related questions I could think of while working on this thesis.

Last but not least my thanks goes to Andrej Cibicik for useful insight to the use of laser scanners. I also appreciated our discussions at Manulab giving me useful input to my project as well as insight to the life of a researcher at NTNU.

Vebjørn B. Bjørhovde

Vebjørn B. Bjørhovde

Summary

In this thesis, a real-time performance test of the `moto` library is conducted. It is a Python library made for controlling Motoman robots from YASKAWA both using trajectory points and real-time control. Performance testing in this thesis is concerned with real-time velocity control, and focuses on response time and latency. Response time is defined as the time it takes for a robot to react after a command is sent. Latency is defined as the time it takes for a robot to reach a commanded velocity after the command is sent. These properties are found trough experiments conducted on a robot manipulator and a positioning table. Both the step response and the systems ability to follow a reference signal is tested.

Two modified version of the `moto` library is also tested. One system has an implemented PID controller while the other one is run with the High Accuracy Path Control Function enabled in the robot controller. These three versions of the system are referred to as system modes. All three systems modes are tested, followed by a comparison and discussion concerning real-time performance.

The thesis also presents a variety of theory starting with general robot kinematics and an introduction to control theory. Various software concepts are then presented, along with existing software frameworks for robot control. This theory is then used along with the `moto` library to suggest a complete system for constraint-based robotic welding. The challenges and considerations of such a system is then discussed along with its advantages and disadvantages.

Sammendrag

I denne masteroppgaven gjennomføres det en sanntid-ytelsestest for `moto` biblioteket. Dette er et Python bibliotek lagd for å styre Motoman roboter fra YASKAWA ved bruk av både banepunk og sanntidsstyring. Ytelsestestingen i denne oppgaven omfatter sanntids hastighetsstyring, og vil fokusere på responstid og forsinkelse i systemet. Responstid er definert som tiden det tar fra en kommando blir sendt til roboten før den reagerer. Forsinkelse er definert som tiden det tar for en robot å nå en ønsket hastighet etter at hastighetskommandoen er sendt. Både responstid og forsinkelse blir undersøkt gjennom eksperimenter på en robot manipulator og et posisjoneringsbord. Både steg responsen og systemets evne til å følge et referansesignal i endring blir testet.

To modifiserte versjoner av `moto` biblioteket blir også testet. Et av systemene har en implementert PID regulator, mens det andre systemet har aktivert "High Accuracy Path Control Function" i robotkontrolleren. De tre versjonene av systemet blir referert til som system versjoner. Alle tre system versjonene blir testet, etterfulgt av en sammenlikning og diskusjon av deres sanntidsytelse.

Masteroppgaven vil også presentere et utvalg av teori. Først vil generell robotkinematikk gjennomgåes sammen med en introduksjon til reguleringsteknikk. Ulike konsepter innen datateknologi vil dermed presenteres, før ulike eksisterende rammeverk for robotstyring vil forklares. Teorien sammen med `moto` biblioteket vil så brukes for å presentere et forslag for et komplett begrensningsbasert system for robotisert sveising. Utfordringer og hensyn som må taes i et slikt system vil så diskuteres sammen med fordeler og ulemper med et slikt system.

Contents

Preface i

Acknowledgements iii

Summary v

Sammendrag vii

1. Introduction 1
- 1.1. Problem Statement 1
- 1.2. Related Work 1
- 1.3. Methodology 4
- 1.4. Structure of the Thesis 4

2. Robot Kinematics 7
- 2.1. Degrees of Freedom 7
- 2.2. Redundancy 8
- 2.3. Rotation Matrices 9
  - 2.3.1. Exponential Representation of Rotation 11
  - 2.3.2. Matrix Logarithm of a Rotation Matrix 11
- 2.4. Transformation Matrices 12
- 2.5. Twists and Screws 14
  - 2.5.1. Adjoint Representation 16
- 2.6. Exponential Representation and Matrix Logarithm of Transformation Matrices 16
- 2.7. Forward Kinematics 17
- 2.8. The Jacobian 19
  - 2.8.1. The Pseudoinverse of the Jacobian 20
- 2.9. Singularities 20
- 2.10. Inverse Kinematics 20
- 2.11. Inverse Velocity Kinematics 22
- 2.12. URDF 23

3. Control Theory **25**
3.1. Terminology . . . 25
3.2. Feedback and Feedforward . . . 27
3.3. Stability . . . 28
3.4. PID controller . . . 31
3.5. Ziegler-Nichols Method . . . 32

4. Software Theory **35**
4.1. Internet Protocol Suite . . . 35
4.1.1. Telnet . . . 37
4.1.2. TCP . . . 38
4.1.3. UDP . . . 40
4.1.4. IP . . . 41
4.2. Real-Time Systems . . . 42
4.3. Kernels . . . 42
4.3.1. Linux Kernel . . . 44

5. Existing Software **47**
5.1. ROS 2 . . . 47
5.1.1. Graphs . . . 47
5.1.2. Nodes . . . 48
5.1.3. Topics and Messages . . . 48
5.1.4. Services . . . 49
5.1.5. Actions . . . 50
5.2. ROS-Industrial . . . 51
5.3. ROS2 control . . . 52
5.3.1. Hardware Components . . . 53
5.3.2. Resource Manager . . . 55
5.3.3. Controller Manager . . . 55
5.3.4. Controllers . . . 55
5.4. Moto . . . 56
5.4.1. Architecture . . . 58
5.4.2. Simple Message . . . 61
5.4.3. Real-Time Mode . . . 61
5.5. eTaSL/eTC . . . 62
5.5.1. eTC . . . 62
5.5.2. eTaSL . . . 64

6. MotoTester **67**
6.1. moto_tester.py . . . 67
6.2. rt_setup.py . . . 68
6.3. moto_tester_rt.py . . . 68

6.4. rt_server.py . . . 68
6.5. data_plotter.py . . . 71
6.6. utilities.py . . . 71

7. Hardware **73**
7.1. Robot Manipulator . . . 73
7.2. Positioning Table . . . 76
7.3. Robot Controller . . . 77
7.3.1. Teach Pendant . . . 77
7.4. Welding Apparatus . . . 79
7.5. Laser Scanner . . . 80
7.6. Computer Specifications . . . 80

8. Experiment Setup **81**
8.1. Robot Cell Setup for Welding . . . 81
8.2. Welding with Teach Pendant . . . 82
8.3. Moto Testing . . . 82
8.3.1. Frequency Verification . . . 83
8.3.2. Step Response . . . 83
8.3.3. Following a Reference Signal . . . 84

9. Results **85**
9.1. Welding with Teach Pendant . . . 85
9.2. Moto Testing, Manipulator . . . 86
9.2.1. Frequency . . . 86
9.2.2. Step Response . . . 86
9.2.3. Following a Reference Signal . . . 87
9.3. Moto Testing, Positioning Table . . . 87
9.3.1. Frequency . . . 88
9.3.2. Step Response . . . 88
9.3.3. Following a Reference Signal . . . 88
9.4. A System for Constraint-Based Robotic Welding . . . 89

10. Discussion **91**
10.1. Welding with Teach Pendant . . . 91
10.2. Moto Testing . . . 92
10.2.1. Frequency . . . 92
10.2.2. Step Response . . . 93
10.2.3. Following a Reference Signal . . . 95
10.3. Validity of the Real-Time Performance . . . 98
10.4. Latency Reduction . . . 98
10.4.1. Improving the PID Controller . . . 98

10.4.2. HTRAJ 99
10.4.3. Custom Robot Controller 99
10.5. Timeouts and Possible Solutions 100
10.6. Complete System for Constraint-Based Robotic Welding 101
10.6.1. Challenges and Considerations 101
10.6.2. Advantages and Disadvantages 102

11. Conclusion **105**
11.1. Further Work 106

A. A Selection of Robot Jobs **113**
A.1. INIT_ROS 113
A.2. INIT_ROS with HTRAJ 114
A.3. SIMPLE_ARC 114
A.4. CIRCULAR_ARC 115

B. YASKAWA Motoman GP25-12 **117**

C. YASKAWA MT1-500 S2HD **119**

D. Robot Manipulator Plots **121**
D.1. Step Response 121
D.2. Following a Reference Signal 122
D.3. Following a Modified Reference Signal 123

E. Positioning Table Plots **125**
E.1. Step Response 125
E.2. Following a Reference Signal 126
E.3. Following a Modified Reference Signal 127

List of Figures

1.1. Thesis structure . . . 5

2.1. Degrees of freedom in 3D . . . 8
2.2. Rotation matrix example . . . 10
2.3. Homogeneous transformation matrix example . . . 13
2.4. Linear velocity of a twist in a fixed frame . . . 15
2.5. Product of exponentials . . . 18
2.6. Singular configuration of manipulator . . . 21
2.7. Inverse kinematics algorithm . . . 22
2.8. URDF link example . . . 23
2.9. URDF joint example . . . 24

3.1. Common control theory terminology . . . 26
3.2. DC-motor block diagram . . . 26
3.3. Disturbance feedforward block diagram . . . 27
3.4. Reference feedforward block diagram . . . 28
3.5. Asymptotically stable step response . . . 29
3.6. Unstable step response . . . 30
3.7. Marginally stable step response . . . 30
3.8. PID controller block diagram . . . 31

4.1. The TCP/IP protocol suite . . . 36
4.2. TCP header . . . 38
4.3. TCP data transfer process . . . 40
4.4. UDP header . . . 41
4.5. IPv4 header . . . 42
4.6. Computer system definition . . . 43
4.7. Kernel communications . . . 44

5.1. ROS 2 graph . . . 48
5.2. Nodes communicating over a topic . . . 49
5.3. ROS 2 service . . . 50
5.4. ROS 2 action . . . 51

5.5. PR2 robot 53
5.6. ROS2 control architecture 54
5.7. Node life cycle 57
5.8. moto architecture 58
5.9. moto trajectory points 59
5.10. eTC architecture 63
5.11. eTaSL variable 65
5.12. eTaSL constraint 65
5.13. eTaSL monitor 66

6.1. PID controller in rt-server 69
6.2. Derivative gain effect on step response 70

7.1. Robot cell 74
7.2. Simplified drawing of YASKAWA Motoman GP25-12 75
7.3. Simplified drawing of YASKAWA MT1-500 S2HD 76
7.4. Teach pendant for YRC1000 78
7.5. Remote controller for Fronius TPS400-i 79
7.6. Micro-Epsilon scanCONTROL 2610-100 80

8.1. Step function 83

9.1. Welding seams from welding experiments 85
9.2. Constraint-based robotic welding architecture 90

10.1. Recommended welding direction and angle for MIG welding 92

A.1. INIT_ROS 113
A.2. INIT_ROS with HTRAJ 114
A.3. SIMPLE_ARC 114
A.4. CIRCULAR_ARC 115

B.1. Technical specifications for YASKAWA Motoman GP25-12 118

C.1. Technical specifications for YASKAWA MT1-500 S2HD 120

D.1. Step response joint S 121
D.2. Joint S following a reference signal 122
D.3. Joint S following a modified reference signal 123

E.1. Step response joint X 125
E.2. Joint X following a reference signal 126
E.3. Joint X following a modified reference signal 127

List of Tables

3.1. Ziegler-Nichols tunings 33

4.1. Three categories of real-time systems 43

9.1. Control frequency, robot manipulator 86
9.2. Response time and latency for step responses, robot manipulator . 86
9.3. Latencies following reference signal, robot manipulator 87
9.4. Latencies following modified reference signal, robot manipulator . . 87
9.5. Control frequency, positioning table 88
9.6. Response time and latency for step responses, positioning table . . 88
9.7. Latencies following reference signal, positioning table 89
9.8. Latencies following modified reference signal, positioning table . . 89

10.1. Response times for three different robots 95
10.2. Latency for three different robots 95

Chapter 1.

Introduction

Traditional robot programming using a teach pendant is common practice in the industry today. Despite its popularity this method comes short when more complex tasks are to be programmed. The programming task then becomes a cumbersome one, and researchers are now exploring the possibility of real-time constraint based systems for easier and faster robot programming. Running a robot in real-time introduces requirements to the system regarding execution times and delays. These are properties that should be tested and evaluated.

1.1. Problem Statement

Before developing a complete system for constraint-based robot programming for welding operations, core components have to be tested with regards to real-time performance to determine their suitability for the system. This thesis aims to test the real-time performance of the `moto` library which is a Python library developed for real-time control of Motoman robots from YASKAWA. With a focus on latency and response time, the library is to be tested by experimentation using a Motoman GP25-12 located at Manulab at the the Department of Mechanical and Industrial Engineering at NTNU Trondheim. Additionally, the thesis aims to suggest an architecture for a complete constraint-based system for robotic welding.

1.2. Related Work

This section will present related work done in the filed of performance testing real-time robotic systems. Different frameworks for constraint-based robot programming will also be presented. The section will end by presenting work done to introduce constraint-based robot programming to robotic welding applications.

Performance Testing a Real-Time System

Testing real-time system performance is a natural process when implementing a real-time system. This has been done several times in literature before, but the type of performance test and choice of performance parameters vary.

In [36] a ROS2 framework is implemented on an ABB IRB 14000 YuMi, which is an industrial robot with two 7 DOF arms. Real-time position control of the arms are implemented and real-time performance tested. Seven different performance parameters for a real-time system are suggested, and experiments to determine one of them are conducted. The performance parameter is the delay in the system when tracking a changing reference signal. Delay is defined as the time the robot uses to reach a position after the position command has been sent. This will later in this thesis be defined as latency.

[23] presents PyMoCo which is a Python framework for real-time trajectory generation made for industrial robots. Performance is measured using the response time of the system, which in [23] is defined as the time it takes for the system to reply to a command message. The response time is tested over 100 000 control cycles, and the worst and average response time along with its standard deviation is used to compare different controllers in the framework. Three experimental applications of PyMoCo can also be seen in [24]. In this paper the framework is applied for real-time control of three different robot manipulators, two of them are position controlled and the last one is velocity controlled. Real-time performance in this case is measured somewhat similar to [36] where a changing reference signal is tracked. Latency is then used as a performance indicator. In addition the response time in each of the three systems are measured. Response time in this case is defined as the time it takes from a command is sent until the robot starts moving.

A system for tracking a drawn line on a surface from a fixed distance is presented in [44]. The system consists of a six DOF industrial robot equipped with a camera and laser projection of the tool center. Delays occur in all parts of the system, and these are identified individually before a total system delay is presented. This delay is verified by simulating the system using higher and lower delay before comparing this to empirical data from the physical system. The same tracking experiment was run for all three systems and the physical systems performance proved to be in between the two other systems, thus verifying the estimated delay. In this paper the total system delay is used for measuring real-time performance.

A modular and portable control framework to be used in research applications is presented in [8]. It is implemented on two different robot systems, and real-time tests are conducted on one of them. The tested system consists of a Mitsubishi PA-10 which is connected to the developed control framework. It is controlled in real-time using velocity control. Real-time performance in this case is defined as the execution time of each control cycle in a motion controller.

Constraint-Based Robot Programming

Constraint-based robot programming has been researched for some time, and several frameworks have been developed. In [2] a task specification language eTaSL is presented. This is a Lua based language where robotic tasks are defined as a set of constraints. Joint velocities are then calculated in a corresponding controller, eTC, based on these constraints. This framework will be further explained in Section 5.5. eTaSL/eTc is based off another framework developed at the same university, namely iTaSC [9][10]. In this framework frames are defined in different parts of the robot workspace, and by imposing constraints on the relationship between them robotic tasks can be planned and executed. A different approach is the Stack of Tasks (SoT) presented in [27]. This framework is made for controlling many different tasks simultaneously, and is typically used in humanoid robots. It uses a network of tasks that communicate with each other, and a priority system where lower priority tasks are disregarded if they disrupt higher level tasks.

The three frameworks mentioned above are some of the most complete systems for constraint-based robot programming, but other alternatives also exist. In [45] and [46] a more intuitive approach to constraint-based task specification is introduced, where constraints are are defined as geometrical relations between parts in a CAD model. This makes for a more intuitive approach to setting constraints compared to other task-specification languages. The framework is further improved in [47] where a faster constraint solver is introduced.

In [20] a framework for controlling humanoid robots is introduced. It is based on the fact that the control of a human robot can be regarded as a set of hierarchically arranged subtasks that all have to work together. This framework also focuses on the dynamic behaviour of the system, and how to handle it in addition to robot kinematics. Another framework for multiple prioritised tasks can be found in [11]. In comparison to the previously presented frameworks, this work focuses on force control of robots. This is done efficiently by using a solver where the kinematics and dynamics of the robot are solved separately. Another focus in constraint-based robot programming is the concept of robot human collaboration. This is the main focus of [22] where a constraint-based system with prioritised tasks are used. In this system high priority constraints can be defined for collision avoidance with humans. This creates a robotic system where the safety of humans always have the highest priority and will cancel out any contradicting tasks. Lastly [39] presents a methodology for restricting the movement of objects by defining geometrical constraints between them. A solver is also introduced to find a solution to the defined constraints. This solver utilise the fact that movement can be decomposed into translation and rotation of the objects. The methodology and solver can be applied to the definition of robotic tasks.

Robotic Welding using Constraint-Based Robot Programming
Robotic welding using constraint-based robot programming has been mentioned in literature before without a complete framework being presented. Various welding methods are used as examples for applications of the work presented in [57], [45] and [47]. To the authors knowledge the only work resembling a constraint-based system for robotic welding is presented in [3]. In this paper a solution for controlling a seven-axis robot with a two axis positioning table during a welding operation is presented. The solution uses constraints for dealing with the redundancy in the system, avoiding singularities and keeping the weld seam horizontal. The solution is similar to modern frameworks for constraint-based robot programming, where a task is divided into smaller subtasks that are to be solved within the given constraints. The difference can be seen in constraint definition. In [3] there are three equally weighted constraints, that all subtasks strictly must follow. Modern frameworks however introduces weighted constraints so that some constraints are deemed more important than others and a solver fill calculate the optimal solution to for all subtasks. Also note that the solution presented in [3] is not a complete system for constraint-based robot programming for welding operations as it presents a solution for one specific task only.

1.3. Methodology

To give the reader a theoretical understanding of the results and discussion in this thesis, the first chapters has been dedicated to theory. For these chapters, as well as Section 1.2, a literature study was conducted.

For the results and discussion, empirical methods have been utilised. As the `moto` library was previously untested, hands on testing was regarded as the best approach to obtain relevant results.

1.4. Structure of the Thesis

This thesis is structured after the IMRaD model for scientific writing with one slight modification. The model divides a text into four different sections: Introduction, Methodology and Material, Results and Discussion. For this thesis methodology is included in the introduction, as it is not an elaborate part of the thesis. The four different sections, and the chapters that belong to each section, can be seen in Figure 1.1.

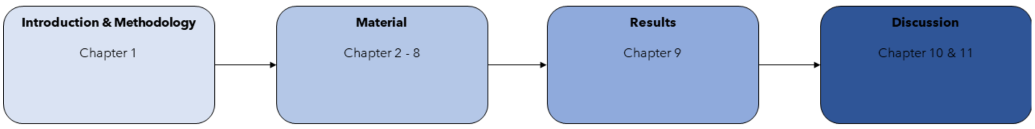


Figure 1.1.: The structure of this thesis based on the IMRaD model (Introduction, Method and Material, Results and Discussion).

Chapter 2.

Robot Kinematics

This chapter will summarise some important aspects of robot kinematics for open chain manipulators. This is a robot where one end is not fastened, like the robotic arms commonly seen in the industry. The theory is meant to give a better understanding of the work that will be presented later in the report. All sections in this chapter are based off the work done in [25] unless stated otherwise. All equations presented are also taken from [25]. This chapter, except for Section 2.12, is copied from the project report attached to this thesis [6].

2.1. Degrees of Freedom

Degrees of freedom (DOF) defines the range of motion for an object. For this chapter, 3D space will be considered. A rigid body in space has a configuration, or pose, which is a description of the body's position and orientation. The number of parameters needed to fully describe the body's configuration is equal to its number of DOF. To illustrate this concept, consider the cube shown in Figure 2.1. As the cube is not constrained in any way, it can be moved in the x, y and z-directions. It can also be rotated by an angle γ, β and α around the x, y and z-axis respectively. For rigid bodies it holds that:

$$\text{DOF} = \text{(sum of freedoms of the bodies)-(number of independent constraints)}$$

From this it can be concluded that a rigid body in space with no constraints has six DOF. The next step is to define the DOF for a robot based on the type and number of joints it has.

In robotics there are different types of joints with different DOF. The most common in industrial robots is the revolute joint, which yields one DOF as it rotates

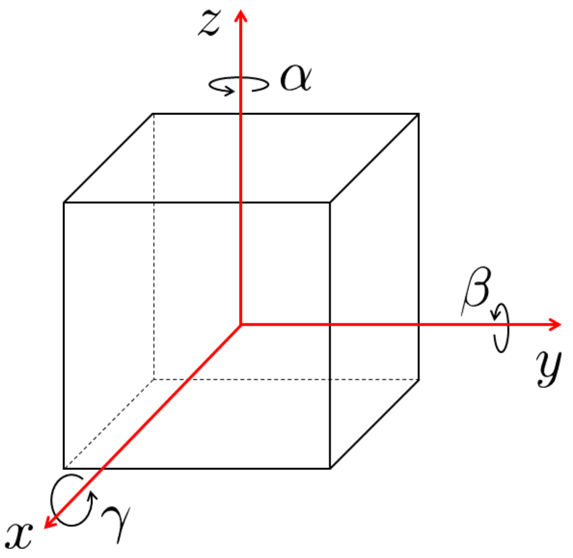


Figure 2.1.: Degrees of freedom (DOF) shown for a cube in 3D.

about one axis. The DOF for a robot is calculated by Grübler's formula:

$$\text{DOF} = m(N - 1 - J) + \sum_{i=1}^{J} f_i \tag{2.1}$$

Where m is the number of DOF for a rigid body, N is the number of robot links including the ground it is mounted on, J is the number of joints and f_i is the DOF for joint i. Note that for a robot with only revolute joints, the DOF is equal to the number of joints. From (2.1) it is seen that a robot can exceed the six DOF needed for free movement in 3D, this will cause redundancy which is the topic of the next section.

2.2. Redundancy

Redundancy has been defined differently by several authors in literature. Several definitions are presented in [7], and a common definition is suggested. Parts of this definition will be summarised in this section. Two definitions are needed before defining redundancy. The workspace of a robot is defined as the space a robot can reach with its end-effector, and the task space is defined as the space where a robot task can be expressed naturally.

The redundancy definition in [7] considers industrial robots with joints that yield one DOF. Let the number of joints in a robot be denoted n and the dimension of the robot's work- and taskspace denoted w and t respectively. With these parameters, three cases are worth noting:

- $n = w$: This is the normal case where most industrial robots operate today.
- $n > w$: This is a case of kinematic redundancy, where the number of robot joints exceeds the requirement for the robot to operate in the workspace. Kinematic redundancy will make the robot more flexible as it can move more joints without changing the end-effector pose.
- $n > t$: This case is similar to the previous one, but defined as task redundancy. In this case a robot has more DOF than is required for its given task.

In the two latter cases redundancy can be utilised to move the robot while still maintaining the same end-effector pose. This can be used to avoid external obstacles, avoid singularities (Section 2.9), minimise energy consumption, etc.

2.3. Rotation Matrices

Before presenting rotation matrices, the concept of frames must be defined. A frame is simply a coordinate system that can be placed freely in space. Usually a stationary space frame is defined and used as reference when working with robotics. A body frame is defined as a frame that is always coincident with a frame attached to a body. For practical reasons frames in this report are always right-handed. With these concepts in place, rotation matrices can be explained.

Rotation matrices are used to represent the orientation of a frame, change the reference frame for a vector or frame and rotate a vector or frame. In 3D these matrices are represented by the special orthogonal group $SO(3)$. These 3×3 real matrices $\boldsymbol{R}$ satisfy the following conditions:

$$\boldsymbol{R}^T\boldsymbol{R} = \boldsymbol{I}$$
$$\det(\boldsymbol{R}) = 1$$

where $\boldsymbol{I}$ is the 3×3 identity matrix. To illustrate the use of rotation matrices Figure 2.2 from [25] will be used, where three different frames are shown along with a point p. The first use of rotation matrices is representing orientation. Orientation is always represented with regards to a reference frame, so imagine a frame {s} coincident with frame {a}. The notation $\boldsymbol{R}_{ij}$ represents the orientation

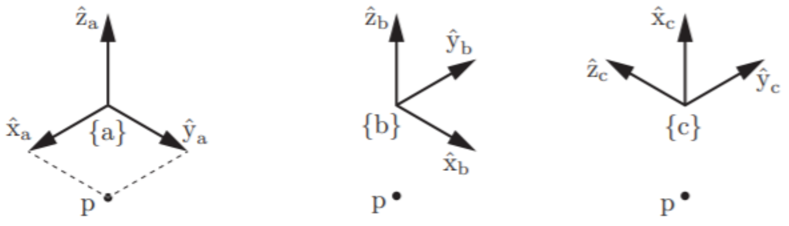


Figure 2.2.: Three coordinate frames with a point p used to demonstrate the use of rotation matrices. Figure from [25].

of frame $\{j\}$ with respect to frame $\{i\}$. This yields the following orientations for the three frames:

$$\boldsymbol{R}_{sa} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}, \boldsymbol{R}_{sb} = \begin{bmatrix} 0 & -1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}, \boldsymbol{R}_{sc} = \begin{bmatrix} 0 & -1 & 0 \\ 0 & 0 & -1 \\ 1 & 0 & 0 \end{bmatrix}$$

The point p is represented as a vector in each frame, note that $\boldsymbol{p}_i$ denotes the point p with reference frame $\{i\}$. This yields the following representations:

$$\boldsymbol{p}_a = \begin{bmatrix} 1 \\ 1 \\ 0 \end{bmatrix}, \boldsymbol{p}_b = \begin{bmatrix} 1 \\ -1 \\ 0 \end{bmatrix}, \boldsymbol{p}_c = \begin{bmatrix} 0 \\ -1 \\ -1 \end{bmatrix}$$

The second use of a rotation matrix is to change the reference frame of another frame or vector. Consider the matrix $\boldsymbol{R}_{sa}$. How can the orientation of frame {a} be represented with respect to frame {b}? This can be done with the following matrix multiplication:

$$\boldsymbol{R}_{ba} = \boldsymbol{R}_{bs}\boldsymbol{R}_{sa} \tag{2.2}$$

Note that $\boldsymbol{R}_{bs} = \boldsymbol{R}_{sb}^T = \boldsymbol{R}_{sb}^{-1}$. The reference frame for vectors can be changed in a similar manner:

$$\boldsymbol{p}_b = \boldsymbol{R}_{ba}\boldsymbol{p}_a \tag{2.3}$$

The last use of rotation matrices is the rotation of a frame or vector. A pure rotation around the unit $\hat{\boldsymbol{x}}$, $\hat{\boldsymbol{y}}$ and $\hat{\boldsymbol{z}}$-axis can be represented by (2.4), (2.5) and

(2.6) respectively, where θ is the rotation angle about the axis:

$$\mathbf{Rot}(\hat{\boldsymbol{x}}, \theta) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta \\ 0 & \sin\theta & \cos\theta \end{bmatrix} \tag{2.4}$$

$$\mathbf{Rot}(\hat{\boldsymbol{y}}, \theta) = \begin{bmatrix} \cos\theta & 0 & \sin\theta \\ 0 & 1 & 0 \\ -\sin\theta & 0 & \cos\theta \end{bmatrix} \tag{2.5}$$

$$\mathbf{Rot}(\hat{\boldsymbol{z}}, \theta) = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{2.6}$$

When rotating a frame with a rotation matrix, it is done by either post- or pre-multiplying the rotation matrix $\boldsymbol{R}$ with the orientation of the frame. Premultiplication, $\boldsymbol{R}\boldsymbol{R}_{ij}$, correlates to rotating $\boldsymbol{R}_{ij}$ with respect to frame {i}. Postmultiplication, $\boldsymbol{R}_{ij}\boldsymbol{R}$, correlates to rotating $\boldsymbol{R}_{ij}$ with respect to frame {j}. A vector must always be rotated with respect to its reference frame and is therefore always premultiplied with the rotation matrix.

2.3.1. Exponential Representation of Rotation

Imagine that instead of representing rotations as pure rotations about each coordinate axis, the rotation can be represented by a rotation θ about one arbitrary unit axis $\hat{\boldsymbol{\omega}}$. The resulting rotation matrix can be found by Rodrigues' formula:

$$\mathrm{Rot}(\hat{\boldsymbol{\omega}}, \theta) = e^{[\hat{\boldsymbol{\omega}}]\theta} = \boldsymbol{I} + \sin(\theta)[\hat{\boldsymbol{\omega}}] + (1 - \cos(\theta))[\hat{\boldsymbol{\omega}}]^2 \tag{2.7}$$

where $[\hat{\boldsymbol{\omega}}]$ is the skew-symmetric representation of $\hat{\boldsymbol{\omega}} = [\omega_1, \omega_2, \omega_3]^T$ defined as:

$$[\hat{\boldsymbol{\omega}}] = \begin{bmatrix} 0 & -\omega_3 & \omega_2 \\ \omega_3 & 0 & -\omega_1 \\ -\omega_2 & \omega_1 & 0 \end{bmatrix} \tag{2.8}$$

2.3.2. Matrix Logarithm of a Rotation Matrix

In (2.7) a rotation matrix $\boldsymbol{R}$ was found from an arbitrary unit rotation axis $\hat{\boldsymbol{\omega}}$ and a rotation angle θ. The matrix logarithm is the opposite operation where the axis and angle is found from the matrix. The two essential equations used to calculate the matrix logarithm form a rotation matrix $\boldsymbol{R}$ are:

$$\operatorname{tr}(\boldsymbol{R}) = 1 + 2\cos(\theta) \tag{2.9}$$

$$[\hat{\boldsymbol{\omega}}] = \begin{bmatrix} 0 & -\omega_3 & \omega_2 \\ \omega_3 & 0 & -\omega_1 \\ -\omega_2 & \omega_1 & 0 \end{bmatrix} = \frac{1}{2\sin(\theta)}(\boldsymbol{R} - \boldsymbol{R}^T) \tag{2.10}$$

From (2.9) it is possible to find the rotation angle θ, which in turn can be inserted in (2.10) to find the unit rotation axis $\hat{\boldsymbol{\omega}}$. Two special cases are worth noting. When $\boldsymbol{R} = \boldsymbol{I}$, $\theta = 0$ which in turn makes $\hat{\boldsymbol{\omega}}$ undefined. When $\operatorname{tr}(\boldsymbol{R}) = -1$, $\theta = \pi$ and $\hat{\boldsymbol{\omega}}$ can be set to any of the following solutions:

$$\hat{\boldsymbol{\omega}} = \frac{1}{\sqrt{2(1 + r_{33})}} \begin{bmatrix} r_{13} \\ r_{23} \\ 1 + r_{33} \end{bmatrix} \tag{2.11}$$

$$\hat{\boldsymbol{\omega}} = \frac{1}{\sqrt{2(1 + r_{22})}} \begin{bmatrix} r_{12} \\ 1 + r_{22} \\ r_{32} \end{bmatrix} \tag{2.12}$$

$$\hat{\boldsymbol{\omega}} = \frac{1}{\sqrt{2(1 + r_{11})}} \begin{bmatrix} 1 + r_{11} \\ r_{21} \\ r_{31} \end{bmatrix} \tag{2.13}$$

where r_{ij} are the elements of the rotation matrix $\boldsymbol{R}$.

2.4. Transformation Matrices

To describe translation as well as rotation in 3D, homogeneous transformation matrices are used. These matrices form the special Euclidean group $SE(3)$, and are 4×4 real matrices. They contain a rotational matrix $\boldsymbol{R} \in SO(3)$ as well as a column vector $\boldsymbol{p} \in \mathbb{R}^3$ representing translation along the three axes of a frame. Homogeneous transformation matrices $\boldsymbol{T}$ take the form:

$$\boldsymbol{T} = \begin{bmatrix} \boldsymbol{R} & \boldsymbol{p} \\ 0 & 1 \end{bmatrix} \tag{2.14}$$

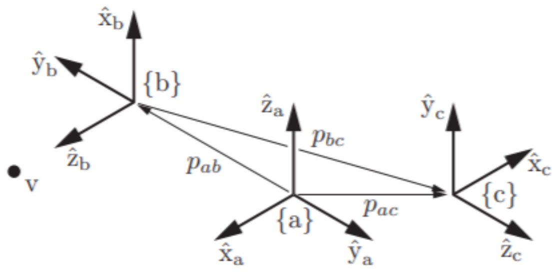


Figure 2.3.: Three frames with a point v used to describe homogeneous transformation matrices. Figure from [25].

The use of transformation matrices are analogous to that of rotation matrices, but also include position and translation of frames. Transformation matrices can be used to represent the position and orientation, i.e. configuration, of a frame, change the reference frame of a vector or frame or change configuration for a vector or frame. To illustrate the different uses of transformation matrices, consider Figure 2.3 taken from [25]. Imagine once again a reference frame {s} that is coincident with frame {a}.

By using transformation matrices as representations of frame configuration, the following matrices are obtained:

$$\boldsymbol{T}_{sa} = \begin{bmatrix} \boldsymbol{R}_{sa} & \boldsymbol{p}_a \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$\boldsymbol{T}_{sb} = \begin{bmatrix} \boldsymbol{R}_{sb} & \boldsymbol{p}_b \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & -1 & 0 & -2 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$\boldsymbol{T}_{sc} = \begin{bmatrix} \boldsymbol{R}_{sc} & \boldsymbol{p}_c \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} -1 & 0 & 0 & -1 \\ 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

The notation is equal to the one for rotation matrices so $\boldsymbol{T}_{ij}$ is the configuration of frame {i} with respect to {j}. The point v is represented by a homogeneous vector $\in \mathbb{R}^4$. In this case the three first elements of the vector are analogous to the case seen for p in Section 2.3, and the last element is equal to one. The rule for changing the reference frame of a transformation matrix is also equal to the one for rotation matrices. Changing the reference frame of $\boldsymbol{T}_{sa}$ from {s} to {b} can be done in the following manner:

$$\boldsymbol{T}_{ba} = \boldsymbol{T}_{bs}\boldsymbol{T}_{sa} \tag{2.15}$$

Note that $\boldsymbol{T}_{bs} = \boldsymbol{T}_{sb}^{-1}$, where the inverse of a transformation matrix is found as:

$$\boldsymbol{T}^{-1} = \begin{bmatrix} \boldsymbol{R}^T & -\boldsymbol{R}\boldsymbol{p} \\ 0 & 1 \end{bmatrix} \tag{2.16}$$

The rules for changing the configuration of a vector or frame are similar to the rules for rotation matrices. The difference between the two being the translation. When pre-multiplying with a transformation matrix the frame is first rotated with respect to the space frame according to the rotation matrix $\boldsymbol{R}$, before it is translated with respect to the space frame according to the vector $\boldsymbol{p}$. When post-multiplying by a transformation matrix the frame is first translated with respect to the body frame, before it is rotated with respect to the body frame. As vectors are only related to one frame they can only be premultiplied to change configuration.

2.5. Twists and Screws

Twists are representations of both the linear and angular velocity of a moving frame. It is a six-dimensional vector consisting of both velocities and can be represented in regards to the fixed frame {s} or the moving body frame {b}. These are the body twist and spacial twist respectively, and are written on the following form:

$$\boldsymbol{\mathcal{V}}_s = \begin{bmatrix} \boldsymbol{\omega}_s \\ \boldsymbol{v}_s \end{bmatrix}, \ \boldsymbol{\mathcal{V}}_b = \begin{bmatrix} \boldsymbol{\omega}_b \\ \boldsymbol{v}_b \end{bmatrix} \in \mathbb{R}^6$$

where $\boldsymbol{\omega}_s$ and $\boldsymbol{\omega}_b$ represents the angular velocity of the frame with respect to the fixed frame and the body frame respectively. $\boldsymbol{v}_b$ represents the linear velocity of the body frame, but $\boldsymbol{v}_s$ does not simply represent the velocity of a frame with respect to {s}.

To explain the physical interpretation of $\boldsymbol{v}_s$ it is useful to look at Figure 2.4 taken from [25]. The body containing both the body frame {b} and the fixed frame {s} is moving with a body twist $\boldsymbol{\mathcal{V}}_b$. Now imagine a point that is attached to the body

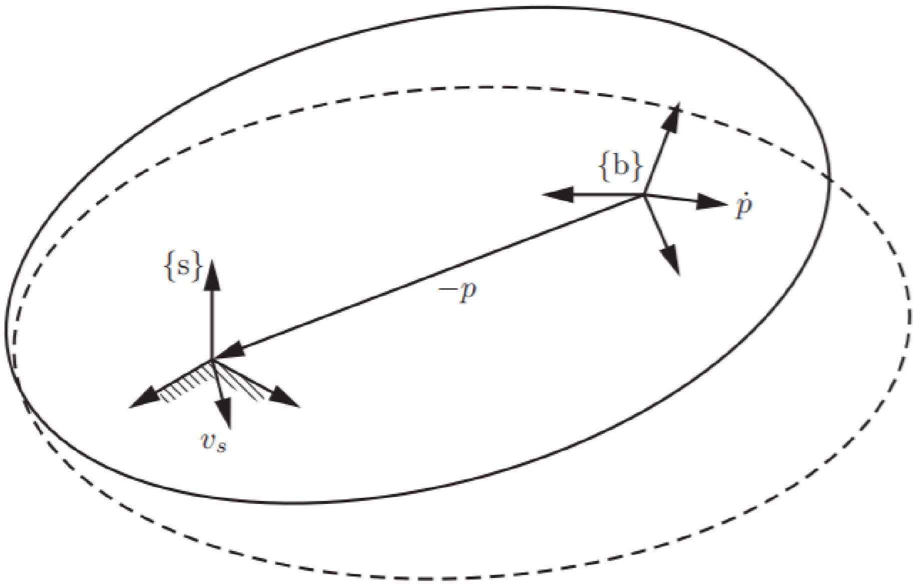


Figure 2.4.: A visual representation of the linear velocity of a twist represented in the fixed space frame {s}. Figure from [25].

and coincident with the origin of {s}. This point will have the linear velocity $\boldsymbol{v}_s$, which is a combination of the rotational and linear velocity of the body.

A twist can be represented by a normalised screw axis $\boldsymbol{\mathcal{S}}$ multiplied by an angular velocity $\dot{\theta}$. The motion of a frame along screw axis will then replicate the threads of a screw with a a translation along the axis happening simultaneously with a rotation about the axis. Angular velocity around this axis is represented by $\dot{\theta}$. The process of finding a screw axis $\boldsymbol{\mathcal{S}}$ and an angular velocity $\dot{\theta}$ from a twist $\boldsymbol{\mathcal{V}} = \begin{bmatrix}\boldsymbol{\omega}^T & \boldsymbol{v}^T\end{bmatrix}^T$ will now be presented.

If $\boldsymbol{\omega} \neq 0$ then the screw axis $\boldsymbol{\mathcal{S}}$ is equal to the twist normalised by $||\boldsymbol{\omega}||$. The rotational velocity $\dot{\theta}$ is then equal to $||\boldsymbol{\omega}||$. If $\boldsymbol{\omega} = 0$ there is no angular velocity and the twist is just a linear translation. The screw axis $\boldsymbol{\mathcal{S}}$ is then represented by the twist normalised by $||\boldsymbol{v}||$. The velocity $\dot{\theta}$ is then equal to $||\boldsymbol{v}||$. For both cases the twist $\boldsymbol{\mathcal{V}} = \boldsymbol{\mathcal{S}}\dot{\theta}$.

A more formal definition of the screw axis can be written as:

$$\boldsymbol{\mathcal{S}} = \begin{bmatrix}\boldsymbol{\omega} \\ \boldsymbol{v}\end{bmatrix} \in \mathbb{R}^6 \tag{2.17}$$

If $\omega = 0$ then $||\boldsymbol{v}|| = 1$ and the twist will be a pure translation along the axis defined by $\boldsymbol{v}$. When $||\boldsymbol{\omega}|| = 1$ the velocity $\boldsymbol{v} = -\boldsymbol{\omega} \times \boldsymbol{q} + h\boldsymbol{\omega}$ where $\boldsymbol{q}$ is a point on the screw axis and h is the pitch of the screw defined as $\frac{\boldsymbol{\omega}^T \boldsymbol{v}}{\dot{\theta}}$.

It can be useful to express the screw axis $\boldsymbol{\mathcal{S}}$ in matrix form. This expression will also hold for a twist $\boldsymbol{\mathcal{V}}$ as $\boldsymbol{\mathcal{S}}$ is a normalised twist.

$$[\boldsymbol{\mathcal{S}}] = \begin{bmatrix} [\boldsymbol{\omega}] & \boldsymbol{v} \\ 0 & 0 \end{bmatrix} \tag{2.18}$$

2.5.1. Adjoint Representation

It is useful to be able to change the reference frame of a twist. This can be done with the adjoint representation of a transformation matrix $\boldsymbol{T}$ which is defined as:

$$[\mathbf{Ad}_{\boldsymbol{T}}] = \begin{bmatrix} \boldsymbol{R} & 0 \\ [\boldsymbol{p}]\boldsymbol{R} & \boldsymbol{R} \end{bmatrix} \in \mathbb{R}^{6\times 6} \tag{2.19}$$

where $\boldsymbol{R}$ and $\boldsymbol{p}$ are the rotation matrix and translation vector associated with $\boldsymbol{T}$ respectively. The reference frame can be changed from the body frame to the fixed frame by the following equation:

$$\boldsymbol{\mathcal{V}}_s = [Ad_{\boldsymbol{T}_{sb}}]\boldsymbol{\mathcal{V}}_b = Ad_{\boldsymbol{T}_{sb}}(\boldsymbol{\mathcal{V}}_b) \tag{2.20}$$

Note the alternative notation of the adjoint representation multiplied by a twist, this will be used in later sections. (2.20) is also applicable to screw axes.

2.6. Exponential Representation and Matrix Logarithm of Transformation Matrices

Just as rotation matrices can be represented by a rotation θ around an arbitrary unit rotation axis $\hat{\boldsymbol{\omega}}$, homogeneous transformation matrices can be represented by a screw axis $\boldsymbol{\mathcal{S}}$ and a distance θ along the screw axis. To transform this representation into a homogeneous transformation matrix in SE(3), the matrix exponential is utilised. It is defined as:

$$e^{[\boldsymbol{\mathcal{S}}]\theta} = \begin{bmatrix} e^{[\hat{\boldsymbol{\omega}}]\theta} & (\boldsymbol{I}\theta + (1-\cos\theta)[\hat{\boldsymbol{\omega}}] + (\theta - \sin\theta)[\hat{\boldsymbol{\omega}}]^2)\boldsymbol{v} \\ 0 & 1 \end{bmatrix} \tag{2.21}$$

Analogous to rotation matrices, it is also possible to find a screw axis $\boldsymbol{S}$ and a distance θ from a transformation matrix $\boldsymbol{T}$ by using the matrix logarithm operation. Given a transformation matrix on the form shown in (2.14). The rotational axis $\hat{\boldsymbol{\omega}}$ and rotation angle θ can be found from the matrix logarithm of $\boldsymbol{R}$ as described in Section 2.3.2. The linear velocity $\boldsymbol{v}$ is found from the following equation:

$$\boldsymbol{v} = (\frac{1}{\theta}\boldsymbol{I} - \frac{1}{2}[\hat{\boldsymbol{\omega}}] + (\frac{1}{\theta} - \frac{1}{2}\cot(\frac{\theta}{2}))[\hat{\boldsymbol{\omega}}]^2)\boldsymbol{p}$$

For the matrix logarithm of $\boldsymbol{T}$ there is one special case to consider. If $\boldsymbol{R} = \boldsymbol{I}$ there is no rotation and the motion is purely transnational. If that is the case set $\hat{\boldsymbol{\omega}} = \boldsymbol{0}$, $\boldsymbol{v} = \frac{\boldsymbol{p}}{||\boldsymbol{p}||}$ and $\theta = ||\boldsymbol{p}||$.

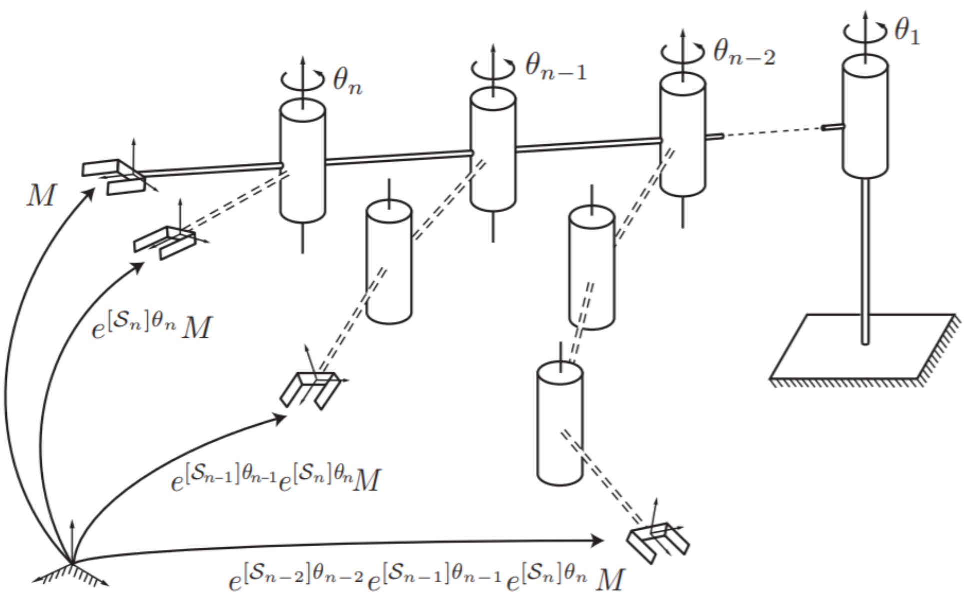
2.7. Forward Kinematics

Forward kinematics in robotics is the calculation of the end-effector configuration based on the joint angles of the robot. There are different ways of calculating forward kinematics, a widespread method uses Denavit-Hartenberg parameters. In this method each link of the robot is given a frame, and the forward kinematics of the robot can be calculated based on the relation between each frame. The method that will be explained in this section is the product of exponential (PoE) formula, as this method correlates well with the previously explained theory and is the method used in [25].

PoE is based off the exponential representation of homogeneous transformation matrices and their product. Consider the open chain manipulator with n rotational joints shown in Figure 2.5 taken from [25]. Define the fixed space frame {s} and an end-effector frame {b}. Also define a homogeneous transformation matrix $\boldsymbol{M} = \boldsymbol{T}_{sb}(\boldsymbol{0})$ representing the end-effector configuration with all n joints angles equal to zero. This is the zero-position or home-position.

PoE can now be divided into two different approaches depending on which frame is used as reference for defining the exponential representations. This section will focus on the space frame formulation where {s} is used as reference. After this explanation a brief summary of the calculations using the end-effector frame {b} as reference will be given.

Each of the n rotational joints on the robot can be defined by a screw axis $\boldsymbol{\mathcal{S}}_i$ where $i = 1, 2, ..., n$. To calculate the axes see (2.17). The upper part of $\boldsymbol{\mathcal{S}}_i$ is a vector representing the rotational axis of joint i. As the motion around the joints are purely rotational, the pitch h is zero. This simplifies the velocity calculation and we can write the screw axes for each joint of the robot as:



θ_n
θ_{n-1}
θ_{n-2}
θ_1
M
$e^{[\mathcal{S}_n]\theta_n}M$
$e^{[\mathcal{S}_{n-1}]\theta_{n-1}}e^{[\mathcal{S}_n]\theta_n}M$
$e^{[\mathcal{S}_{n-2}]\theta_{n-2}}e^{[\mathcal{S}_{n-1}]\theta_{n-1}}e^{[\mathcal{S}_n]\theta_n}M$

Figure 2.5.: An n-link open chain manipulator used to explain the product of exponentials (PoE) formula. Figure from [25].

$$\boldsymbol{\mathcal{S}}_i = \begin{bmatrix} \boldsymbol{\omega}_i \\ -\boldsymbol{\omega}_i \times \boldsymbol{q}_i \end{bmatrix} \quad i = 1, 2, ..., n \tag{2.22}$$

where $\boldsymbol{q}_i$ is a point on the rotational axis of joint i. After finding a screw axis for each joint, they can be represented in exponential form by using (2.21). The value θ_i is the joint angle of joint i. With the zero-position and an expression for each joint angle it is now possible to calculate the end-effector configuration $\boldsymbol{T}_{sb}(\boldsymbol{\theta})$ by pre-multiplying the chain of exponential representations with the zero-position:

$$\boldsymbol{T}_{sb}(\boldsymbol{\theta}) = e^{[\boldsymbol{\mathcal{S}}_1]\theta_1}e^{[\boldsymbol{\mathcal{S}}_2]\theta_2}...e^{[\boldsymbol{\mathcal{S}}_n]\theta_n}\boldsymbol{M} \tag{2.23}$$

Where $\boldsymbol{\theta}$ is a vector containing all joint angles $\theta_1, \theta_2 ... \theta_n$.

Calculating the forward kinematics using the end-effector frame {b} is similar to using the space frame {s}. There are two differences in the approach, the first one being how to calculate the screw axes. These are now calculated with respect to the end-effector frame instead of the base frame and denoted $\boldsymbol{\mathcal{B}}_i$. The second difference is that the zero-position is post-multiplied by the chain of exponential representations instead of pre-multiplied. This yields the following equation for finding the end-effector pose $\boldsymbol{T}$:

$$\boldsymbol{T}_{sb}(\boldsymbol{\theta}) = \boldsymbol{M}e^{[\boldsymbol{\mathcal{B}}_1]\theta_1}e^{[\boldsymbol{\mathcal{B}}_2]\theta_2}...e^{[\boldsymbol{\mathcal{B}}_n]\theta_n} \tag{2.24}$$

2.8. The Jacobian

For an open chain manipulator, the Jacobian represents the relationship between the velocity of the end-effector and the joint velocity such that:

$$\dot{\boldsymbol{x}} = \boldsymbol{J(\theta)}\dot{\boldsymbol{\theta}} \tag{2.25}$$

where $\dot{\boldsymbol{x}} \in \mathbb{R}^m$ is the end-effector velocity, $\boldsymbol{\theta} \in \mathbb{R}^n$ is a list of current joint angles, $\dot{\boldsymbol{\theta}}$ is a list of joint velocities and $\boldsymbol{J(\theta)} \in \mathbb{R}^{m \times n}$ is the Jacobian matrix.

The two types of Jacobians that are relevant for this section are the space Jacobian and the body Jacobian. The space Jacobian $\boldsymbol{J}_s$ correlates the spacial twist of the end-effector to the joint velocities and is defined for a robot manipulator with n joints as:

$$\boldsymbol{\mathcal{V}}_s = \boldsymbol{J}_s(\boldsymbol{\theta})\dot{\boldsymbol{\theta}} \tag{2.26}$$

where the ith column of $\boldsymbol{J}_s(\boldsymbol{\theta})$ is:

$$\boldsymbol{J}_{si}(\boldsymbol{\theta}) = \mathrm{Ad}_{e^{[\boldsymbol{\mathcal{S}}_1]\theta_1}e^{[\boldsymbol{\mathcal{S}}_2]\theta_2}...e^{[\boldsymbol{\mathcal{S}}_{i-1}]\theta_{i-1}}}(\boldsymbol{S}_i) \;\; i = 2, 3, ..., n \tag{2.27}$$

and the first column $\boldsymbol{J}_{s1} = \boldsymbol{S}_1$. Similarly the body Jacobian of a n-joint manipulator is defined as:

$$\boldsymbol{\mathcal{V}}_b = \boldsymbol{J}_b(\boldsymbol{\theta})\dot{\boldsymbol{\theta}} \tag{2.28}$$

where the ith column of $\boldsymbol{J}_b(\boldsymbol{\theta})$ is:

$$\boldsymbol{J}_{si}(\boldsymbol{\theta}) = \mathrm{Ad}_{e^{-[\boldsymbol{\mathcal{B}}_n]\theta_n}e^{-[\boldsymbol{\mathcal{B}}_{n-1}]\theta_{n-1}}...e^{-[\boldsymbol{\mathcal{B}}_{i+1}]\theta_{i+1}}}(\boldsymbol{\mathcal{B}}_i) \;\; i = n-1, n-2, ..., 1 \tag{2.29}$$

and the last column $\boldsymbol{J}_{bn} = \boldsymbol{\mathcal{B}}_n$.

2.8.1. The Pseudoinverse of the Jacobian

It can be useful to find the inverse of the Jacobian matrix. For a six-joint open chain manipulator the Jacobian will be a 6×6 matrix and the inverse can be found normally, but this is not always the case. In the case where there are more than six joints, the Jacobian is called "fat" as it has more columns than rows. Accordingly it is called "tall" when there are less than six joints as the Jacobian then will have more rows than columns.

In both cases the Jacobian will then no longer be square and the regular inverse can no longer be calculated. It is then possible to use the Moore-Penrose inverse commonly called the pseudoinverse. For a matrix $\boldsymbol{A}$ of full rank, the pseudoinverse $\boldsymbol{A}^{\dagger}$ can be calculated by the following formulas:

$$\boldsymbol{A}^{\dagger} = \boldsymbol{A}^T(\boldsymbol{A}\boldsymbol{A}^T)^{-1} \text{ if } \boldsymbol{A} \text{ is fat} \tag{2.30}$$

$$\boldsymbol{A}^{\dagger} = (\boldsymbol{A}^T\boldsymbol{A})^{-1}\boldsymbol{A}^T \text{ if } \boldsymbol{A} \text{ is tall} \tag{2.31}$$

2.9. Singularities

Singularities for open chain manipulators occur when it loses the ability to move its end-effector in one or more directions. Mathematically singularities occur when the Jacobin fails to be full rank. This happens in configurations where joint axes become linearly dependent. An example of a singularity can be seen in Figure 2.6 taken from [25]. In this case the wrist centre of the six-axis robot is placed directly above the shoulder joint making the four joint axes intersect in a common point, thus causing linear dependency.

2.10. Inverse Kinematics

The inverse kinematics problem can be regarded as the opposite of forward kinematics. Given an end-effector pose $\boldsymbol{T(\theta)}$ the task is to find a set of joint angles $\boldsymbol{\theta} = [\theta_1, \theta_2, ..., \theta_n]$ that will yield said pose.

Inverse kinematics can be calculated analytically and numerically. When solving the inverse kinematic problem analytically the geometry of the robot together with trigonometry is used to yield a solution. An exact analytical solution to the inverse kinematics problem can often be complicated and yield cumbersome calculations due to robot geometries not being ideal for analytical calculations. Simplifying the robot geometries for calculations will make the process simpler,

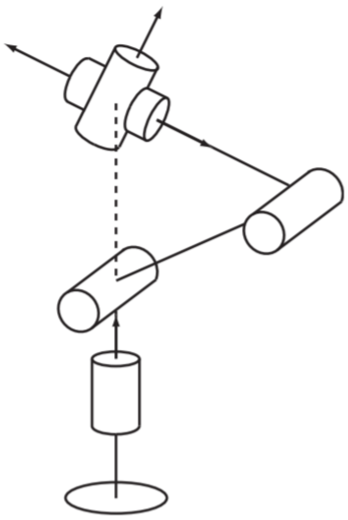


Figure 2.6.: A singular configuration for a six-axis open chain manipulator. Figure from [25].

but the solution will not be exact. In this case the joint angles can be found numerically using the analytical solution as an initial guess.

There are several numerical methods than can be used to solve the inverse kinematics problem. In [25], Newton's method is used for deriving the equations shown in this section.

To show how Newton's method is used to solve the inverse kinematics problem, consider a robot manipulator with an end-effector frame {b} and a fixed frame {s}. The desired end-effector configuration $\boldsymbol{T}_{sd}$ is given along with an initial guess for the values of $\boldsymbol{\theta}$ denoted $\boldsymbol{\theta}_0$. By applying the matrix logarithm the difference between the desired pose $\boldsymbol{T}_{sd}$ and the pose after i iterations $\boldsymbol{T}_{sb}(\theta_i)$ can be represented by a twist $\boldsymbol{\mathcal{V}}_b$ as:

$$[\boldsymbol{\mathcal{V}}_b] = \log(\boldsymbol{T}_{sb}^{-1}(\boldsymbol{\theta}_i)\boldsymbol{T}_{sd}) \tag{2.32}$$

A small deviation between the desired pose and the current pose of the end-effector will yield a small twist. In other words, the magnitude of the elements in $\boldsymbol{\mathcal{V}}_b$ can be used to measure the error in the estimation. Since numerical methods do not necessarily reach an exact solution, it is useful to set acceptable errors for the magnitude of the angular and linear velocity denoted ϵ_ω and ϵ_v respectively.

While $||\boldsymbol{\omega}_b|| > \epsilon_\omega$ or $||\boldsymbol{v}_b|| > \epsilon_v$, i is iterated and the angles $\boldsymbol{\theta}$ will get closer and

```
Result: $\boldsymbol{\theta}$
$i = 0, \boldsymbol{\theta} = \boldsymbol{\theta}_0$;
while $||\boldsymbol{\omega}_b|| > \epsilon_\omega$ or $||\boldsymbol{v}_b|| > \epsilon_v$ do
    $[\boldsymbol{\mathcal{V}}_b] = \log(\boldsymbol{T}_{sb}^{-1}(\boldsymbol{\theta}_i)\boldsymbol{T}_{sd})$;
    $\boldsymbol{\theta}_{i+1} = \boldsymbol{\theta}_i + \boldsymbol{J}_b^\dagger(\boldsymbol{\theta}_i)\boldsymbol{\mathcal{V}}_b$;
    $i = i + 1$
end
```

Figure 2.7.: An algorithm for solving the inverse kinematics problem using Newton's method.

closer to the desired position, as described by the following formula:

$$\boldsymbol{\theta}_{i+1} = \boldsymbol{\theta}_i + \boldsymbol{J}_b^\dagger(\boldsymbol{\theta}_i)\boldsymbol{\mathcal{V}}_b \tag{2.33}$$

An algorithm for solving the inverse kinematics using (2.32) and (2.33) can be set up as seen in Figure 2.7.

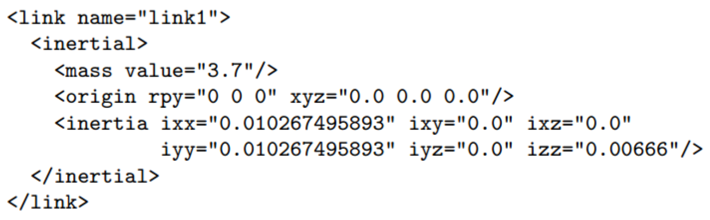
Recall (2.28), and note that $\boldsymbol{J}_b^\dagger(\theta_i)\boldsymbol{\mathcal{V}}_b$ is the joint velocities needed to reach the twist $\boldsymbol{\mathcal{V}}_b$. Each increment in (2.33) takes the previous joint angles and adds the joint angle travelled in the time between each increment. This can be utilised for joint position robot control, by slowly changing the desired joint angles along a planned trajectory without letting the actual joint angles catch up. The previously desired angles can then be used as the initial guess for the numerical algorithm.

2.11. Inverse Velocity Kinematics

Inverse velocity kinematics deals with the problem of finding the necessary joint velocities $\dot{\boldsymbol{\theta}}$ to create a desired twist $\boldsymbol{\mathcal{V}}_d$ represented in an arbitrary frame. As previously mentioned the relation between an end-effector twist and the joint velocities are given by the Jacobian as shown in (2.26) and (2.28). Introducing the pseudoinverse yields the following:

$$\dot{\boldsymbol{\theta}} = \boldsymbol{J}^\dagger(\boldsymbol{\theta})\boldsymbol{\mathcal{V}}_d \tag{2.34}$$

The pseudoinverse used in (2.34) will prioritise all joint movement equally. To change this prioritisation the Jacobian can be altered with respect to different weighting functions. In [25] the Jacobian is altered so that the kinematic energy in the robot is minimised. Inverse velocity kinematics can be used in velocity control of robots.



```
<link name="link1">
  <inertial>
    <mass value="3.7"/>
    <origin rpy="0 0 0" xyz="0.0 0.0 0.0"/>
    <inertia ixx="0.010267495893" ixy="0.0" ixz="0.0"
             iyy="0.010267495893" iyz="0.0" izz="0.00666"/>
  </inertial>
</link>
```

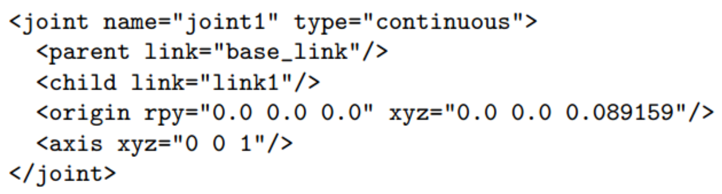
Figure 2.8.: An example of a link representation in a URDF file. Figure from [25].

2.12. URDF

The Universal Robot Description Format is an XML file used to describe the properties of a robot. A URDF file will contain information regarding the links and joints of a robot and the relation between them. This can be used in calculations regarding both the kinematics and dynamics of the robot. The representation of links and joints will now be explained.

Link
A link represented in a URDF file can be seen in Figure 2.8 taken from [25]. The link is first given a name, before its inertial properties are defined. It is given a mass in addition to an origin frame which is the pose of the link's centre of mass relative to the link's joint frame. The joint frame is defined by the joint that connects the link to its preceding link. In the example shown in Figure 2.8 the origin frame is coincident with the link frame. Inertial properties for the link are also defined. All the inertial properties are used when calculating the dynamics of the robot. In addition to the variables defined in the example, geometric properties of the link can be given. These are useful when checking for collisions in the planned robot trajectory. Other properties like colour and material can also be defined.

Joint
A joint represented in a URDF file can be seen in Figure 2.9 taken from [25]. Just as with the links it is given a name, but also a type. The type describes the kinematic property of the joint. Common types like rotational and prismatic joints are often used, but joints can also be fixed. This is a virtual joint that does not allow any movement, but still connects two links. The continuous joint shown in the example is a rotational joint without joint limits. A joint connects two links called the parent link and the child link. The joint is then described by the transformation matrix representing the child link with respect to the parent link when the joint angle is set to zero. In the example the transformation matrix is defined by the rotation angles `rpy` (roll, pitch, yaw) and a translation along the parent link joint frame. For the case shown in Figure 2.9 the child link joint frame



```
<joint name="joint1" type="continuous">
  <parent link="base_link"/>
  <child link="link1"/>
  <origin rpy="0.0 0.0 0.0" xyz="0.0 0.0 0.089159"/>
  <axis xyz="0 0 1"/>
</joint>
```

Figure 2.9.: An example of a joint representation in a URDF file. Figure from [25].

is translated 0.089159 meters along the z axis of the parent link joint frame, while maintaining the same orientation.

Chapter 3.

Control Theory

This chapter will give a brief introduction to control theory and its terminology before introducing some relevant concepts that will be used later in the thesis. Unless stated otherwise, the contents of this chapter is based off [4].

3.1. Terminology

In the field of control theory, the object is to control a process so that some variables are able to follow a set of reference values while adjusting for outside disturbances. The controlled variables are called states. The reference value could be anything depending on the process, for example the water level in a tank, the pitch angle for a ship or the temperature of a smelter. To control the process, a controller is implemented that will adjust the state so that it follows the reference value. The controller and the process is together regarded as a system. A system in control theory is defined by the user. Defining the system is an important task as it is crucial to include enough components for the system to be controllable, while excluding all irrelevant elements that would complicate the system. Figure 3.1, adapted from [4], shows some central terms in control theory some of which already has been briefly explained. These will now be explained further and the rest of the terminologies will be presented.

As mentioned earlier a process is the part that is to be controlled, and it is affected by disturbances. For this explanation a process containing a DC-motor will be used as an example. The angular velocity of the rotary shaft is the parameter that is to be controlled i.e. the state. A block diagram showing this system can be seen in Figure 3.2. The process in this case is then the DC-motor itself, while the system refers to all of the elements seen in Figure 3.2. Disturbances on the system $\mathbf{v}$ are loads on the rotary shaft that would affect its angular velocity.

The controlled parameter is called the state $\mathbf{x}$ and has to be measured by some

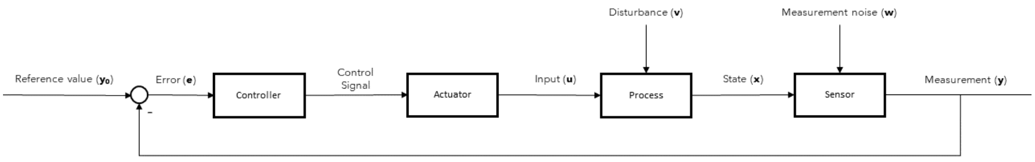


Figure 3.1.: Some common terminology in control theory shown in a block diagram. Adapted from [4].

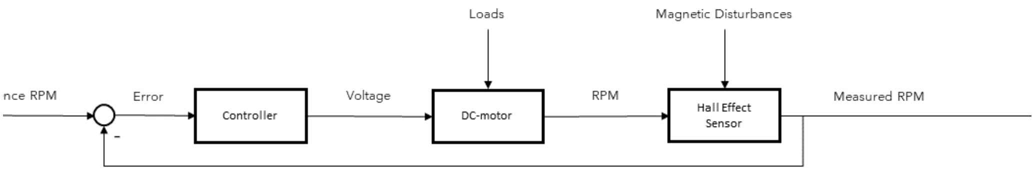


Figure 3.2.: A block diagram showing a DC-motor where the angular velocity of the rotary shaft is controlled.

kind of sensor. In this example a hall effect sensor can be used to measure the angular velocity, resulting in a measurement $\mathbf{y}$. Sensors are prone to measurement noise $\mathbf{w}$ which could cause deviation between the actual state and the measured state. Measurement noise is often dampened or removed by different filters e.g. a low pass filter.

As previously mentioned a controller is implemented for the state to be able to follow the reference signal. Different controllers are suitable for different processes and states, and choosing the right one is crucial for the performance of the system. The PID controller is a typical controller that will be explained in Section 3.4. As seen in Figure 3.1 the controller uses a reference value $\boldsymbol{y}_0$ set by the user and compares it to the measured value of the state $\mathbf{y}$. The difference between the measured value and the reference value is the error $\mathbf{e}$ which is used by the controller to determine a control signal.

The control signal is then sent to an actuator that will affect the process with an input $\mathbf{u}$. Using the example with a DC-motor the actuator would be a power supply with adjustable voltage. The input to the DC-motor is then a voltage that in turn will affect the rotational velocity of the rotary shaft. It is common to combine the actuator and controller so that in Figure 3.1 the controller will send

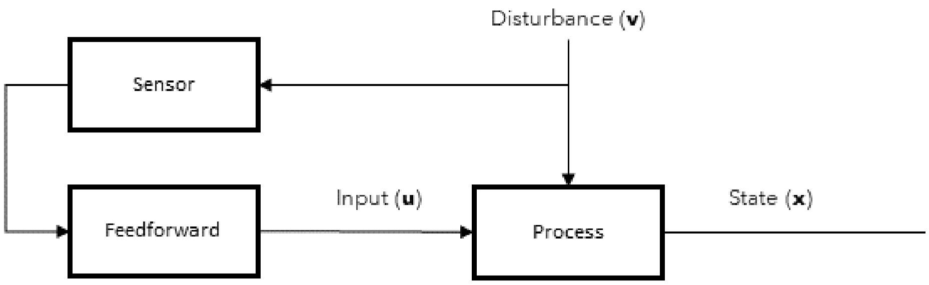


Figure 3.3.: A block diagram of a system using feedforward to compensate for disturbances in the system. Figure adapted from [4].

the input directly to the process. This was done in Figure 3.2 and will be done in all further figures in this chapter.

3.2. Feedback and Feedforward

When controlling a process there are two main principles being used, feedback and feedforward. Figure 3.2 shows the DC motor system using feedback. Notice that this is the same principle seen in Figure 3.1. A sensor provides feedback to the controller, and the measured state is compared to a reference value. The error is then used as an input to the controller, which calculates an input that is sent back to the process to adjust the state accordingly.

Feedforward is based around the knowledge of incoming changes to the system. If the disturbances in a system is measurable, it is possible to use feedforward to make the system "anticipate" the change. This will in turn make the system react more quickly. As an example imagine the DC-motor from previously being mounted in an electric car, and the speed of the car is to be held constant. An uphill could in this case be considered a disturbance. By measuring inclination of the car, this disturbance can be sensed and compensated for by applying more voltage to the motor. A block diagram of feedforward being used to compensate for disturbances can be seen in Figure 3.3, adapted from [4]. This type of feedforward depends on the disturbances being measurable.

In a similar manner, feedforward can be used to adjust to changes in the reference signal. The reference signal is then sent directly to the process without using a controller. By using the same example with the DC-motor mounted in a car,

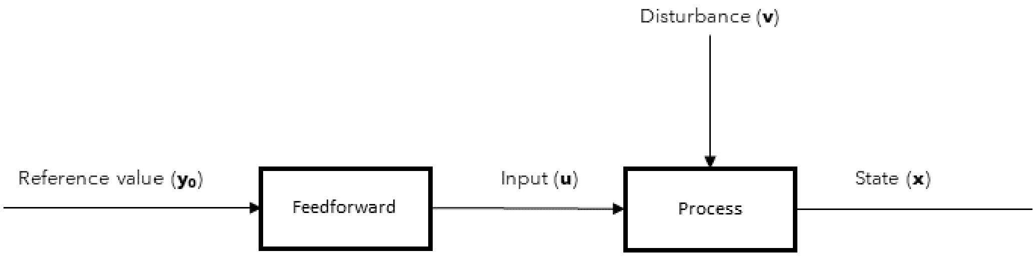


Figure 3.4.: A block diagram of a system using feedforward from the reference signal. Figure adapted from [4].

feedforward of the reference signal can be used if the route of the car is already known. All uphills and downhills can then be predicted and all changes in the reference signal can be known before the car starts to drive. An example of using feedforward of the reference signal is shown by a block diagram in Figure 3.4, adapted from [4].

A system using only feedforward will never receive any feedback on the actual state of the system. Unforeseen events could then cause the states to deviate from the reference value without the system noticing. A common practice is to combine feedforward with feedback. This way the system can react to disturbances and changes in the reference signal quickly using feedforward, while a controller using feedback assures that the states are actually following the reference value.

3.3. Stability

When controlling a process, the goal is to make the states in the system follow a reference signal with the presence of disturbances. Stability is a way of defining the systems behaviour based on its ability to deal with disturbances and changes in the reference signal. The stability of a system can be divided into three different categories, which will now be presented.

Asymptotically stable

An asymptotically stable system is a system that is able to follow the reference signal even with disturbances present. Figure 3.5 shows an asymptotically stable step response for a robot joint following a reference joint velocity. In this case some oscillation occurs, but the system eventually settles around the reference joint velocity. This behaviour of dampening oscillations is typical for an asymptotically

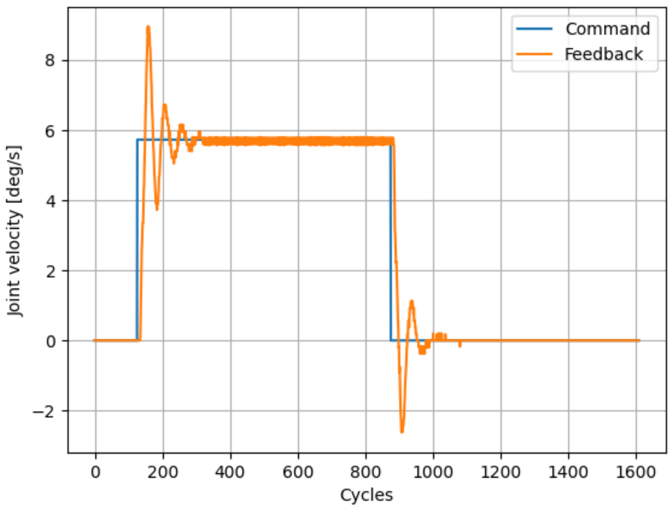


Figure 3.5.: An asymptotically stable step response while controlling the joint velocity of a robot joint.

stable system. The system could oscillate more or less than shown in the figure and still be called asymptotically stable as long as it settles on the reference velocity.

Unstable

The unstable system is not able to follow the reference signal and will oscillate with increasing amplitude over time. This kind of system is unusable as it can not be controlled and will often cause unwanted and potentially dangerous behaviour. Unstable behaviour from the robot joint mentioned previously can be seen in Figure 3.6.

Marginally stable

A marginally stable step response can be seen in Figure 3.7. Again it shows the response of a robot joint following a reference velocity. In contrast to the two previous cases the joint velocity is not able to follow the reference velocity, but it is not unstable. A marginally stable system can be regarded as the transition between a stable and an unstable system. In this case the joint velocity will simply continue to oscillate with the same amplitude.

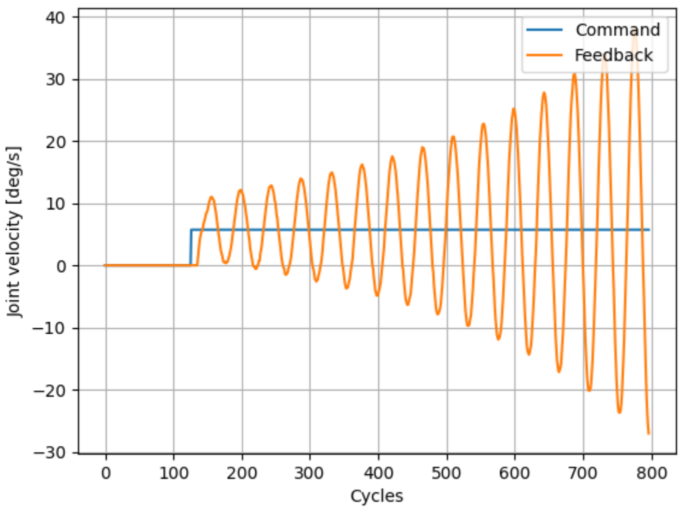


Figure 3.6.: An unstable step response while controlling the joint velocity of a robot joint.

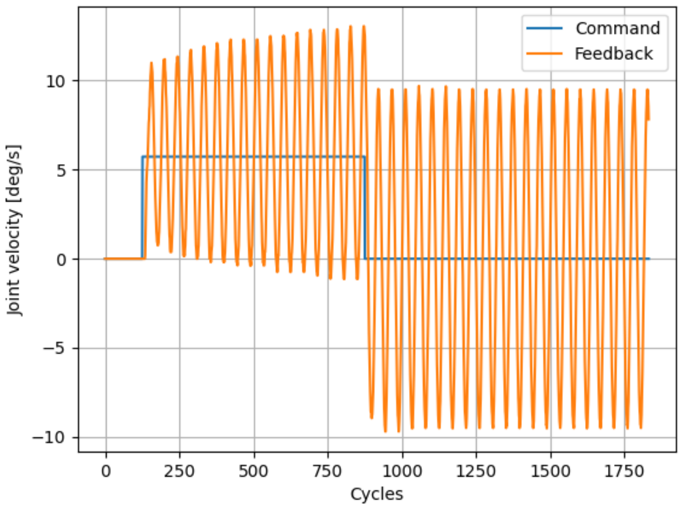


Figure 3.7.: A marginally stable step response while controlling the joint velocity of a robot joint.

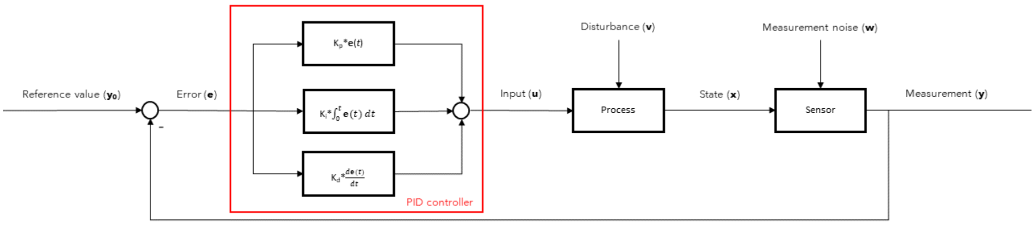


Figure 3.8.: A block diagram of a typical system with an implemented PID controller.

3.4. PID controller

The PID controller is a widely used controller, and it consists of three different controllers: the proportional controller, the derivative controller and the integral controller. A block diagram showing a PID controller can be seen in Figure 3.8. It is seen that all three components uses the error to calculate the input to the the process, and the final input is the summation of the three contributions. Each of the controllers yields a different characteristic. The characteristics of the PID controller can be changed by adjusting how much each of the components contribute to the final input. This process is called tuning and will be further explained in Section 3.5. The remainder of this section will explain the characteristic behaviour of each of the three controllers in a PID controller.

Proportional controller

The proportional (P) controller simply amplifies the measured error in the system by a constant K_p called the proportional gain. This amplified error is then used as the input to the system. Mathematically it can be written as:

$$\boldsymbol{u}(t) = K_p(\boldsymbol{y}_o - \boldsymbol{y}(t)) \tag{3.1}$$

Where $\boldsymbol{u}$ is the input, $\boldsymbol{y}_0$ is the reference value and $\boldsymbol{y}$ is the measured state. By adjusting K_p the systems behaviour is changed. Setting K_p too high will result in an unstable system, while setting it too low will cause the system to react slowly or not reach the reference value at all.

Integral controller

While the P controller responds to the instantaneous error, the integral (I) controller responds to error over time and can be modelled by the following equation:

$$\boldsymbol{u} = K_i \int_0^t (\boldsymbol{y}_0 - \boldsymbol{y}(t)) \tag{3.2}$$

Where K_i is the integral gain analogous to the proportional gain. The I controller is useful for removing constant offsets from the reference value. A constant offset will build up error over time in the I controller, which will cause the offset to be removed. Using only an I controller will cause the system to be slow as it takes time before the I controller starts to react. For a more responsive system it is common to combine the I controller with a P controller making a PI controller. Just as for the P controller, setting K_i too high will cause the system to be unstable. Setting K_i too low will simply remove the integral effect.

Derivative controller
Unlike the two other controllers the derivative (D) controller is not able to bring the error to zero by itself. Instead it tries to counteract changes in error. The mathematical representation can be written as:

$$\boldsymbol{u} = K_d \frac{d(\boldsymbol{y}_0 - \boldsymbol{y}(t))}{dt} \tag{3.3}$$

Where K_d is the derivative gain similar do K_p and K_i. As the derivative effect will counteract changes in the error, its desired effect is dampening oscillations in the state. Adjusting K_d affects how much dampening is applied. Setting K_d low will remove the derivative effect in the system, while a very high K_p will dampen changes too much eventually preventing the state to follow the reference value properly. This will cause the system to react slowly and improper to changes in the reference value.

In theory the derivative effect should be able to remove oscillations completely which would for example smooth out the feedback velocity shown in Figure 3.5 so that no overshoot occurs. In practice this can be challenging if the feedback signal has noise, which is almost always the case. The D controller will then amplify the noise which will eventually make the system unstable.

3.5. Ziegler-Nichols Method

When implementing a PID regulator, all three gains K_p, K_i and K_d have to be set bu the user. This is called tuning the controller and can be done using different established methods. This section will explain the Ziegler-Nichols method (ZN method) presented in [58], which is a method made for tuning a physical system based on empirical data. This is useful for systems that has not been represented

Controller	$\boldsymbol{K_p}$	$\boldsymbol{K_i}$	$\boldsymbol{K_d}$
P	$0.5K_u$	-	-
PI	$0.45K_u$	$1.2/T_u$	-
PID	$0.6K_u$	$2.0/T_u$	$T_u/8$

Table 3.1.: Gains for different controllers using Ziegler-Nichols method.

mathematically.

The ZN method is based on setting gains by trial and error. All tuning is based around the asymptotically stable behaviour of the system, and this is the first property that has to be found. It is done by gradually increasing K_p while keeping both K_d and K_i at zero. The proportional gain used when obtaining a marginally stable response is called the ultimate gain K_u, and the period of the oscillations at this stage is denoted T_u. All three gains can then be set based upon K_u, T_u as well as the type of controller that is to be implemented, by following Table 3.1. The ZN method describes how to set parameters for P controllers, PI controllers and PID controllers.

Chapter 4.

Software Theory

This chapter will present a variety of theory regarding different software concepts. These are presented here first, and will be relevant for the work done later in the thesis or the discussion in Chapter 10.

4.1. Internet Protocol Suite

This section will give an introduction to the internet protocol suite and some of its concepts. The contents of this section is based off [49] unless stated otherwise.

Communication between two computers is a task that requires a lot of coordination between the two involved parts. To enable this process, it is usually divided into several sub-tasks. These sub-tasks are solved in modular frameworks consisting of layers called layered protocol architectures. Each layer in the framework can be regarded as a node that solves one of the sub-task in the communication process. Each layers only communicate with the previous or next layer in the architecture with standard communications. This makes the framework modular as each layer can be changed to the needs of the user without having to adjust the whole architecture. The method each layer uses to solve its sub-task is called a protocol. Of all the architectures the most important framework is called the internet protocol suite, also known as the TCP/IP protocol suite.

The layers in the TCP/IP protocol suite as well as some example protocols in each layer can be seen in Figure 4.1. Note that the number of layers in the TCP/IP protocol suite varies between different authors, but [49] uses five. Starting with the top layer, the sub-task of each layer will now be explained. The next subsections will further explain some protocols found in the TCP/IP protocol suite that will be used later in the thesis.

Application Layer

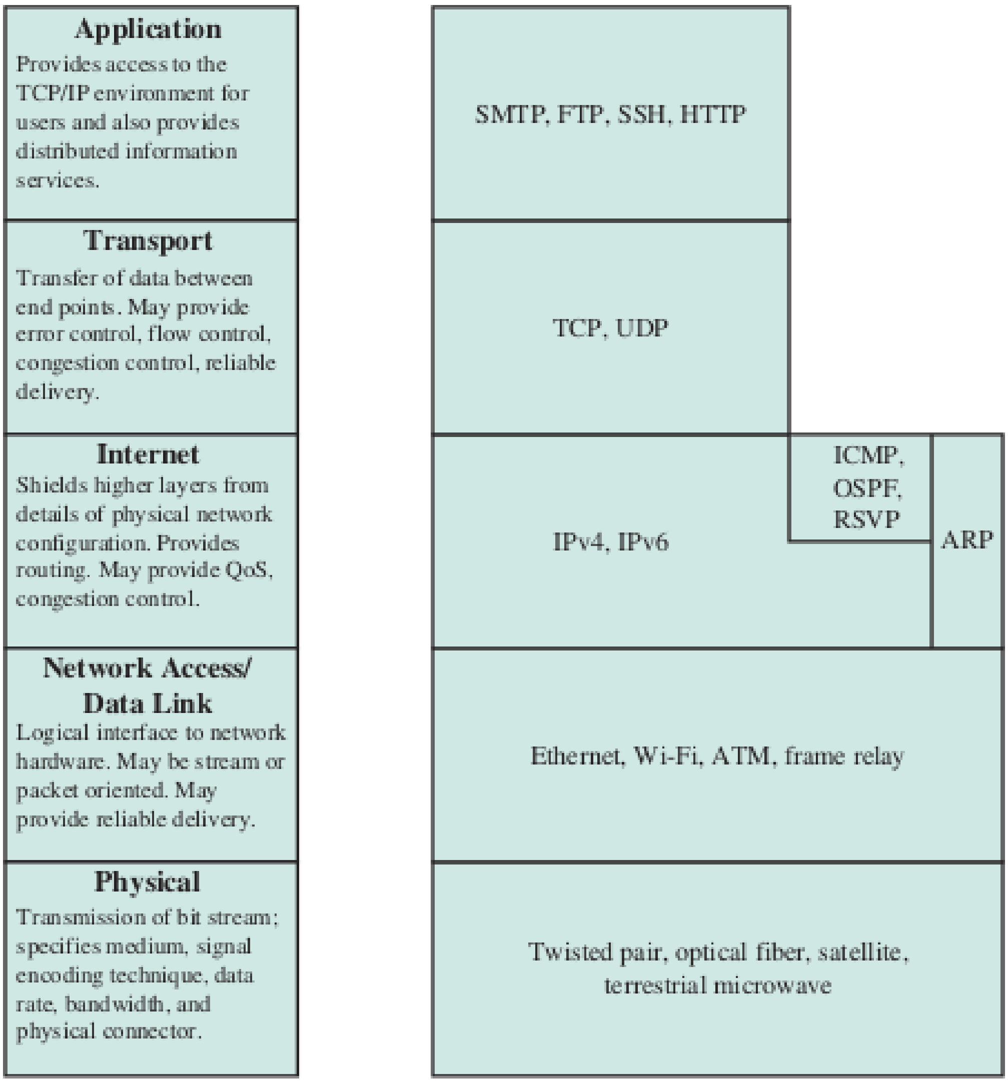

Application
Provides access to the TCP/IP environment for users and also provides distributed information services.
Transport
Transfer of data between end points. May provide error control, flow control, congestion control, reliable delivery.
Internet
Shields higher layers from details of physical network configuration. Provides routing. May provide QoS, congestion control.
Network Access/ Data Link
Logical interface to network hardware. May be stream or packet oriented. May provide reliable delivery.
Physical
Transmission of bit stream; specifies medium, signal encoding technique, data rate, bandwidth, and physical connector.
SMTP, FTP, SSH, HTTP
TCP, UDP
ICMP, OSPF, RSVP
IPv4, IPv6
ARP
Ethernet, Wi-Fi, ATM, frame relay
Twisted pair, optical fiber, satellite, terrestrial microwave

Figure 4.1.: The layers and examples of protocols in each layer in the TCP/IP protocol suite. Figure from [49].

The application layer is the program running on a computer that wants to communicate with some other application. To have separate communications for each application on a computer, they are assigned a service access point or port. The data that is to be sent has its origin in the application layer, which is the sent to the transport layer.

Transport Layer
The transport layer receives data from the application layer trough the designated port of the application that wants to send data. In the transportation layer the data is divided into smaller elements more suited for being sent over a network. Each element also is also given some metadata in the transport layer contained in a header. The data plus the header is called a segment. A header from the transportation layer contains metadata like which port sent the data, which port is the recipient of the data and other information depending on the type of protocol that is used. After adding a header, each segment is sent to the internet layer.

Internet Layer
When communicating between computers that are connected to different networks, there needs to be some system for finding a correct transportation route from the sender to the receiver. This is the task of the internet layer. In this layer a header is added containing information about the next destination of the segment. The segment from the transport layer plus the header added in this layer then forms what is called a packet. When sending packets over larger distances it is sent trough multiple units called routers. The routers takes the packets and reads the information in the header, then compares this information to a table and finds the next destination that will bring the packet closest to the final destination. When it is ready to be sent the packet is sent to the data link layer.

Data Link Layer
The data link layer reads the header from the internet layer and then sends the packet to the appropriate destination.

Physical layer
Packets can be sent in different ways with different hardware. The physical layer is responsible for the interaction between the computer and the medium that the packet is sent over. Examples include satellite, optical fibre and twisted pair cable.

4.1.1. Telnet

Telnet is an application protocol for establishing remote access between two units. It establishes a two way text communication where terminal commands can be sent. Due to telnet being developed before the internet it sends raw commands without encryption. This makes it a fast tool, but insecure if used in online appli-

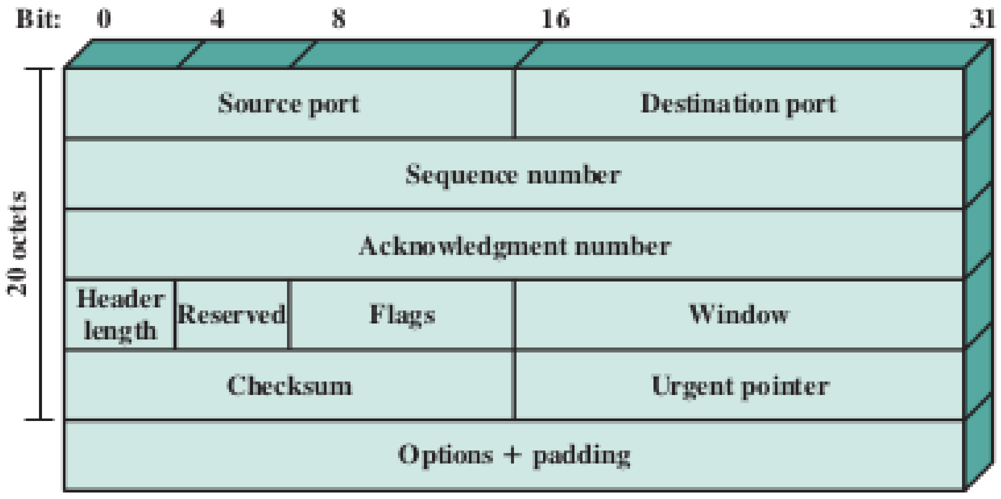


Figure 4.2.: The header added to data to form a TCP segment. Figure from [49].

cations. In most applications Telnet has been replaced by Secure Shell (SSH) [51].

4.1.2. TCP

Transmission Control Protocol (TCP) is the most commonly used protocol in the transport layer. It is a protocol designed for reliable communication between two parts called TCP users. This section will present the TCP header before introducing important mechanisms in TCP.

TCP Header
TCP will add a header to the data that is to be sent making it a TCP segment. The contents of a TCP header can be seen in Figure 4.2, taken from [49]. From the top it is seen that the header contains a source and destination port, which defines which port the data is sent form and which port it is sent to. As the data is divided into several TCP segments, it is important that the TCP user receiving the segments is able to put them back together in the correct order. The sequence number is used for this purpose and a number is assigned to each segment before transmitting. Each sequence also includes an acknowledgement number which defines the sequence number of the next TCP segment in the transmission, as well as acknowledge that the previous sequence was received.

Data offset contains information on the size of the header so that it is possible to separate the header from the data after transmission. The reserved field is currently not in use and should be set to zero. Flags can be used to give information about the segment being sent, or the transmission in general. Details on different flags will not be explained in this thesis. When transmitting the TCP segments, there is a limit to how much data the receiving TCP user can receive in each segment. This limit is indicated in the window field, and is sent from the receiver to the sender.

The checksum is a field that is responsible for detecting any errors that might have occurred during transmission. When the receiver obtains a segment, the checksum is read to assure that no errors occurred. The urgent pointer gives the option of marking certain parts of the data as urgent. How urgent data is handled has to be defined by the receiver.

The last field in the header seen in Figure 4.2 is options and padding. This field allows for different preferences to be set for transmission of the TCP segments. Individual options will not be explained further in this thesis. The padding is extra bits that can be added to an option so that the options and padding field always is a multiple of eight bits.

Connection Establishment
TCP connections are exclusive between two ports, and only one connection may exist between them at a time. The connection is started by a three way handshake. First the sender sends a message to the recipient requesting communication and stating the sequence number of the first TCP segment that is to be sent. The recipient then responds by acknowledging the connection as well as sending the first sequence number that it will send back. The sender then sends a confirmation back to the receiver and communication can start.

Data Transfer
When transferring data there are different policies that can be set depending on the requirements of the system. This section will not go into detail on the different policies, but rather explain broadly how data transfer occurs.

A simple visualisation of the data transfer between two computers can be seen in Figure 4.3. During transmission the sender keeps a list of all segment numbers for sent segments. When the receiver receives a segment, it is either accepted of rejected due to some error. When a segment is accepted, an acknowledgement of the segment number is sent back to the sender to confirm that the segment was received. Upon receiving the acknowledgement the sender knows that the segment has arrived without error, and can mark the segment number as accepted. If an acknowledgement is not received for a segment number within a given time, the segment is retransmitted by the sender. This way TCP transfers data without the loss of segments. As the receiver receives segments, the data can be reconstructed using segment numbers.

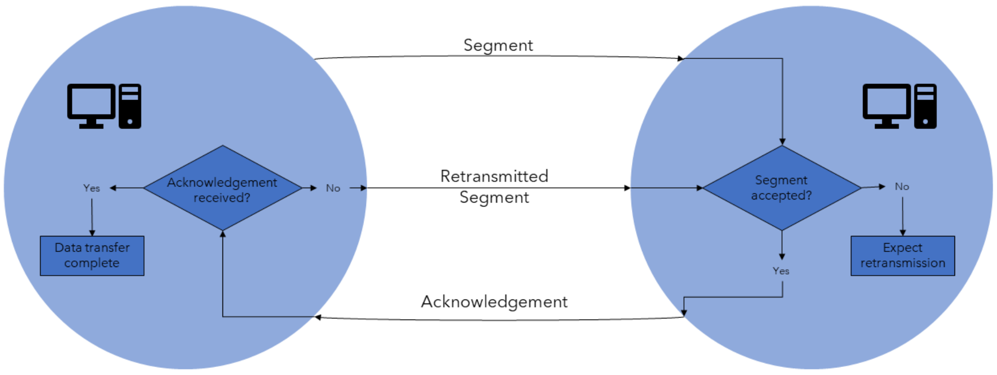


Figure 4.3.: A visualisation of the data transfer process in the TCP protocol.

Connection Termination

Terminating the connection can either happen gracefully or abruptly. A graceful close happens in a similar manner to the three way handshake used when opening the connection. In the flag field of the header one of the TCP users indicates that it wants to terminate the connection. The request is then acknowledged by the other user, and a similar closing request is sent back. The original user then acknowledges the closing request and the connection is gracefully closed.

An abrupt termination happens when one of the users sends an abort command. Communications is then stopped immediately and any segments that are in the process of being transmitted are deleted.

4.1.3. UDP

The User Datagram Protocol (UDP) is another important transport protocol in the TCP/IP protocol suite. In contrast to TCP where a connection is established between to users, UDP is a connectionless protocol. Due to UDP being connectionless its header is less comprehensive than the TCP header. Figure 4.4 from [49] shows the elements found in a UDP header.

The header is similar to the first part of the TCP header. It includes a source and destination port for the data sender and receiver respectively. Unlike the TCP header, the UDP header is always the same length. Due to this fact the length

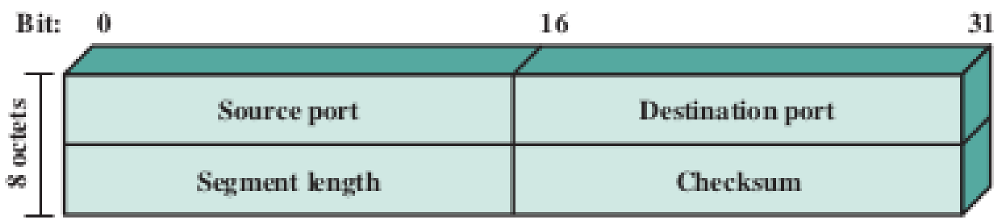


Figure 4.4.: The header added to data to form a UDP segment. Figure from [49].

field in a UDP header contains the length of the whole segment instead of just the header, as this will always be the same. The checksum works similarly for UDP as it does for TCP and is used for error detection.

A UDP server can be connected to multiple clients listening to the transmission. Where TCP can be regarded as a two-way communication, UDP is more of a one-way communication. UDP segments are transmitted form the server to the client, and any segments with the wrong checksum are discarded without any retransmission. This is what makes UDP connections ideal for real-time applications. It can send segments on a more reliable time interval than a TCP connection, as it does not concern with retransmitting lost segments. This is an important property of real-time systems which will be explained in Section 4.2.

4.1.4. IP

The internet protocol (IP) is a protocol in the internet layer. This is the most used internet protocol and the TCP/IP protocol suite is based upon IP. The subject of IP is too large to be handled in detail by this thesis, but a brief introduction will be given. When the IP receives a segment form the transport layer, an IP header is added to form an IP packet.

IP generally refers to IPv4 which is the fourth version of IP. The structure of an IPv4 header can be seen in Figure 4.5. Some of the fields seen here are analogous the the TCP header. The source and destination address can be compared to ports in TCP, and defines the addresses of the sending and receiving computer. A header checksum is also present to detect any errors in transmission. The other fields in the header will not be explained further in this section.

One concept worth noting is that of static and dynamic IP addresses. A dynamic IP address is one that is assigned to a device when it connects to a network. This address will change over time as reflected by the name. A static IP address is than naturally an address that does not change, which can be useful when

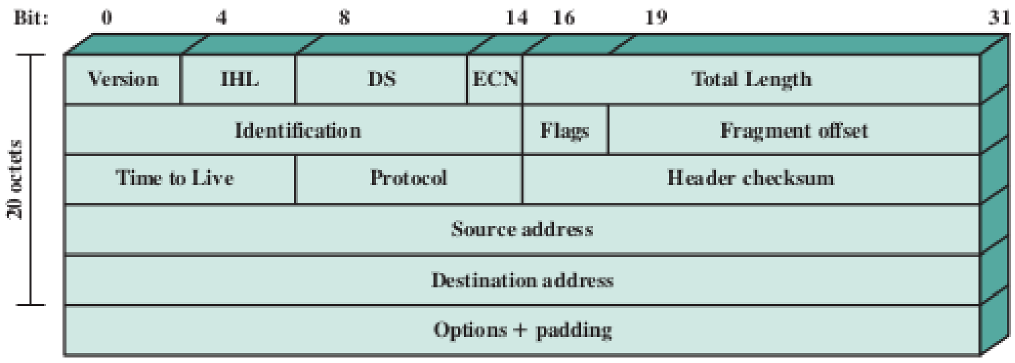


Figure 4.5.: The IPv4 header, figure from [49].

communicating with external devices that has to remember a specific address [18].

4.2. Real-Time Systems

This section will give a brief introduction to real-time systems, as it is an important part for several of the following chapters. The content in this section is based off [21].

To describe a real-time system, a regular computer system first has to be defined. A computer system is defined as something that takes a series of inputs and converts them to a series of outputs. This definition is visualised in Figure 4.6 taken from [21]. A computer system by this definition has no time limit on when the output has to be presented from the system. Using this as a basis a real-time system can be seen as a computer system where the output has to be presented at a specific time. One definition of real-time systems can be seen in [21] as:

"A real-time system is one whose logical correctness is based on both the correctness of the outputs and their timeliness."

Depending on the consequence of output being delayed, real-time systems are classified in three different categories: soft, firm and hard. The definition of each system as seen in [21] can be seen in Table 4.1.

4.3. Kernels

This section will explain the basic concept of kernels in a computer as well as the two main categories of kernels. Lastly the Linux kernel will be explained shortly. This section is based off [50].

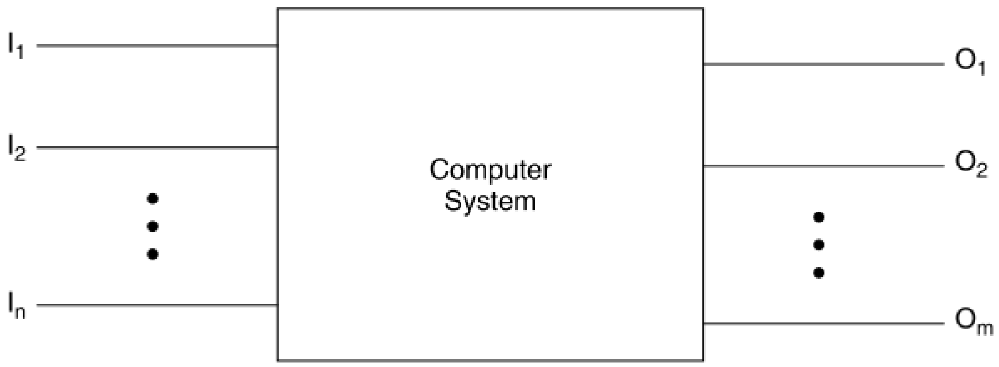


Figure 4.6.: A computer system taking a series of input and converting them to a series of outputs. Figure from [21].

Category	**Definition**
Soft	A real-time system where performance is degraded but not destroyed by failure to meet response-time constraints.
Firm	A real-time system where missing a few deadlines will not lead to total failure, but missing more than a few may lead to complete and catastrophic system failure.
Hard	A real-time system where missing a single deadline may lead to complete and catastrophic failure.

Table 4.1.: The three categories of real-time systems. Definitions from [21].

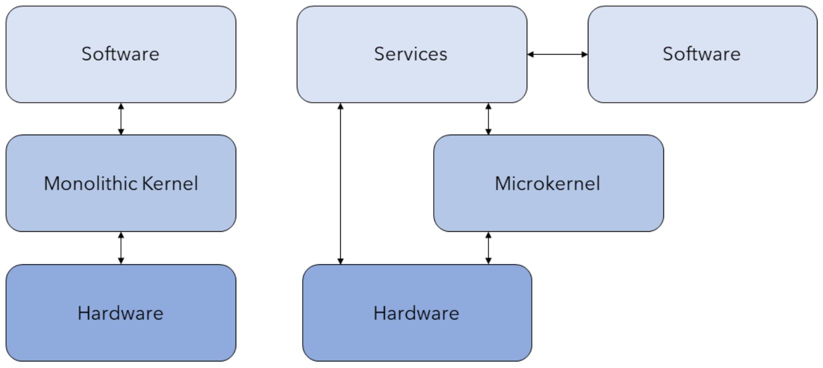


Figure 4.7.: Communications for both monolithic kernels and microkernels.

When operating a computer there are various internal processes running where for example input/output are handled, memory is managed and operations are scheduled to the CPU. These are all processes that cannot be handled by the user directly, so this task is assigned to the kernel. The kernel can be regarded as the link between software and hardware components of the computer and is a part of the computers operating system. To prevent user applications from interfering with kernel operations it runs in a separate part of the memory not accessible by the user.

Kernels are divided into two main categories, monolithic kernels and microkernels. The difference between the two being how they are built. A monolithic kernel contains all kernel functionality which is practically the whole operating system. A microkernel only contains some core kernel functions like memory management, while the rest of the kernel functions are handled by applications running alongside user applications called servers. This makes for a more modular and customisable kernel. How each of the kernel types operate has been visualised in Figure 4.7

4.3.1. Linux Kernel

Most Linux kernels are monolithic. Compared to microkernels the monolithic kernels are usually not modular, and making changes to them can be an elaborate process where much of the kernel has to be changed. As Linux is open-source this would potentially be a problem, but a work-around has been implemented. The monolithic Linux kernel is build using modules that can be loaded and unloaded in the kernel. This enables Linux to have a monolithic kernel while also having the modularity of a microkernel.

Particularly interesting for this thesis are kernels and modifications that enables the Linux system to reliably run real-time systems. This can for example be done with the PREEMPT-RT kernel patch [12]. This will update a regular Linux kernel making it able to satisfy hard real-time constraints. In short it enables the system to prioritise tasks over one another so that particular processes can be given highest priority and run undisturbed.

Chapter 5.

Existing Software

This chapter will give an introduction to existing software that will be relevant for experiments and discussion in this thesis. First ROS 2 will be presented along with ROS-Industrial and ROS2 control. The Moto framework [53] made for controlling Yaskawa MOTOMAN robots will then be presented by highlighting important aspects and showing the architecture of the framework. Lastly an introduction to the constraint-based framework eTaSL/eTC will be given.

5.1. ROS 2

The Robotic Operating System (ROS) is a collection of libraries made for developing control applications for robotic systems. It originated from research done at Stanford University, and was further developed by Willow Garage [38]. ROS is currently maintained by Open Robotics and further developed by the ROS community. To adapt to changes in the world of robotics as well as the needs of the ROS community, ROS 2 was developed [43]. This section will cover the basic concepts of ROS 2. Unless stated otherwise the content of this section is based off [43].

5.1.1. Graphs

In the following subsections it is seen that a ROS 2 system consists of different elements like nodes, topics and services. All of the different components in a system are gathered in the ROS 2 graph. The graph can be seen as a type of flowchart visualising how the different components of the system communicates with each other. A typical graph can be seen in Figure 5.1 which shows a graph of the system obtained by following the tutorials at [43].

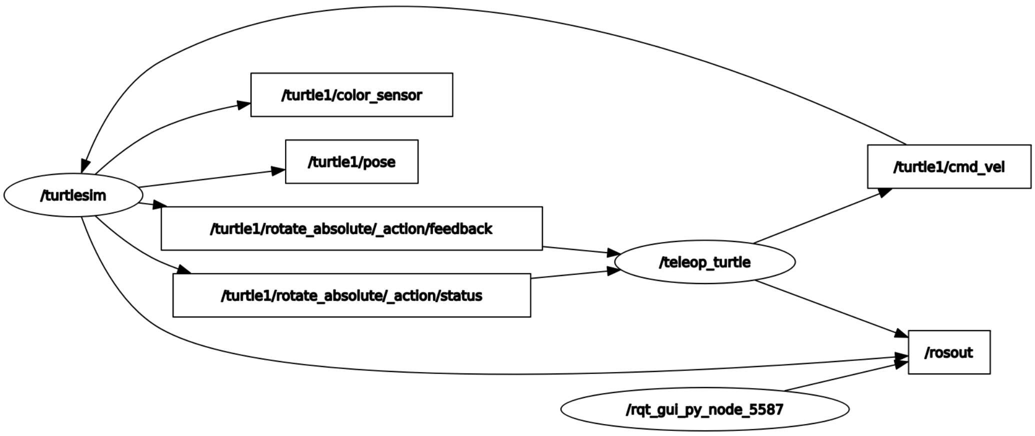


Figure 5.1.: A ROS 2 graph obtained trough the tutorials at [43].

5.1.2. Nodes

The nodes in a ROS 2 system are the components doing most of the actual work. They are used to read sensor data, do calculations, communicate with other systems etc. Each node run a separate piece of code suited for the task it is set to complete. Usually this involves taking some input and processing it, before sending it to another part in the ROS 2 system. In Figure 5.1, nodes are represented by the ellipsoid shapes. Different nodes in the same graph can be written in different programming languages due to the ROS 2 client libraries. Officially there are two client libraries: rclpy for Python and rclcpp for C++. In addition to these there are several community developed client libraries for other programming languages.

Each node is defined by its parameters. These are variables that can be changed by the system or manually by the user. This allows the user to simulate different scenarios and observe how the system reacts by setting parameter values. It also allows the system to adjust the nodes behaviour, which makes it adaptable to different situations.

5.1.3. Topics and Messages

Nodes can communicate with each other trough messages that are sent and received over different topics. A message can be a standard message type like an `int` or a `string` or it can be any custom type defined by the user. Common for all messages is that they contain some kind of information that works as input for a node, output to the user or output to some other system. In Figure 5.1 messages

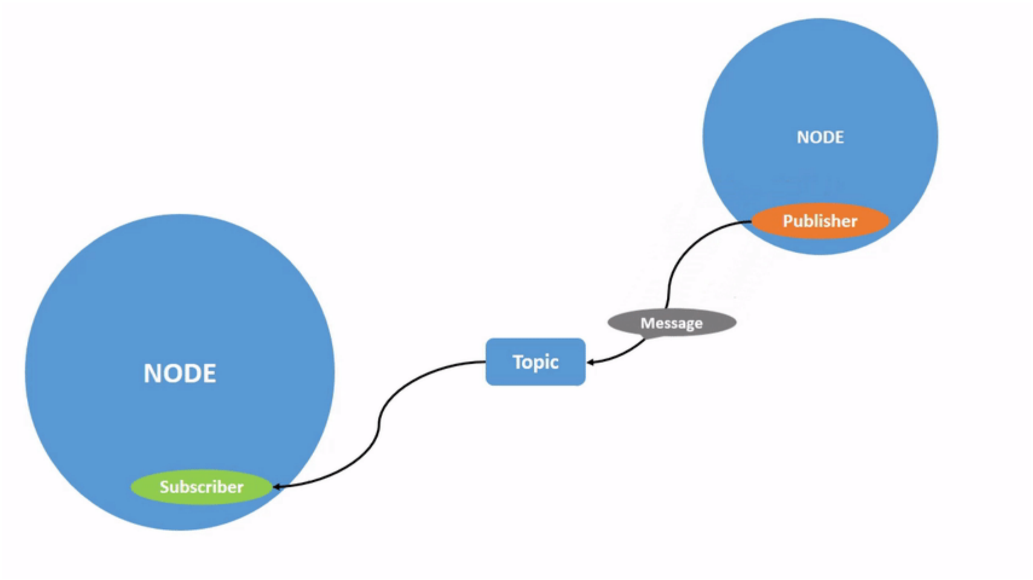


Figure 5.2.: Two nodes communicating with messages over a topic, taken from [43].

can be seen as the lines connecting all the nodes.

Topics can be regarded as different channels that messages are sent over. A node that sends a message to a topic is called a publisher, it publishes a message to the topic. A receiving node is called a subscriber as it subscribes to a certain topic, receiving all messages that are published there. A node can publish and subscribe to several topics at the same time. In Figure 5.1, topics are represented by the squares that the nodes interact with. Figure 5.2 from [43] shows a publish-subscriber model, where the publisher node publishes a message to a topic before the subscriber node receives it.

5.1.4. Services

In contrast to the publish-subscriber model where a node publishes messages to a topic continuously and other nodes subscribe to it, a service is a way for nodes to communicate with each other on-demand. A service is visualised in Figure 5.3 taken from [43]. It consists of a service client node, a service server node and the service itself.

The service is activated by a request message being sent from the service client to the service. Upon receiving the request, the service sends the request to the service client and waits for a response. When the service receives the response from the

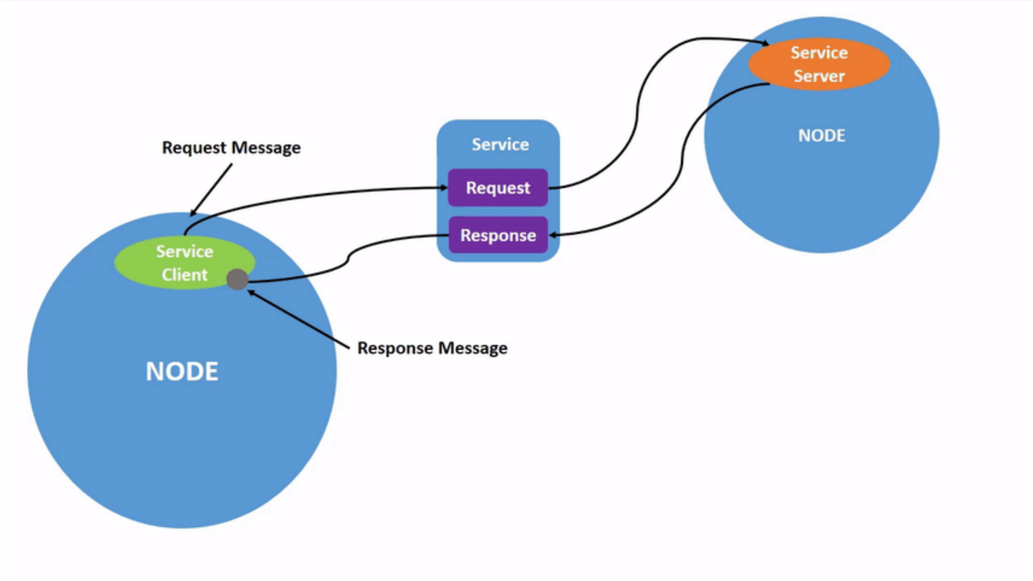


Figure 5.3.: A visualisation of a ROS 2 service. Figure from [43].

service server, it is sent forward to the service client and the cycle is complete. Multiple service clients can be connected to a service, but only one service server may be connected. When multiple service clients are connected, only the client sending a request message will receive a response. This way services are also a way for nodes to communicate more individually than with the publish-subscriber model, where all nodes that subscribe to a topic will receive the published message.

5.1.5. Actions

Actions are similar to services but are more suited for long running tasks. It consists of two services, one feedback topic, an action client and an action server. An action is visualised in Figure 5.4 taken from [43].

To activate an action, the goal service client sends a request message to the goal service before it is sent forward to the goal service server. The goal service server sends a response message trough the goal service and back to the goal service client confirming that the goal of the action is set. After the goal has been set the result service client sends a request message to the result service where the message is relayed to the result service server. The goal of the action has now been set, and the action client is waiting for a result from the action server. This will trigger the feedback publisher to start publishing feedback to the feedback topic, where the feedback subscriber is subscribed. The action client will now receive feedback from the action server on the progress of reaching the set goal. Feedback

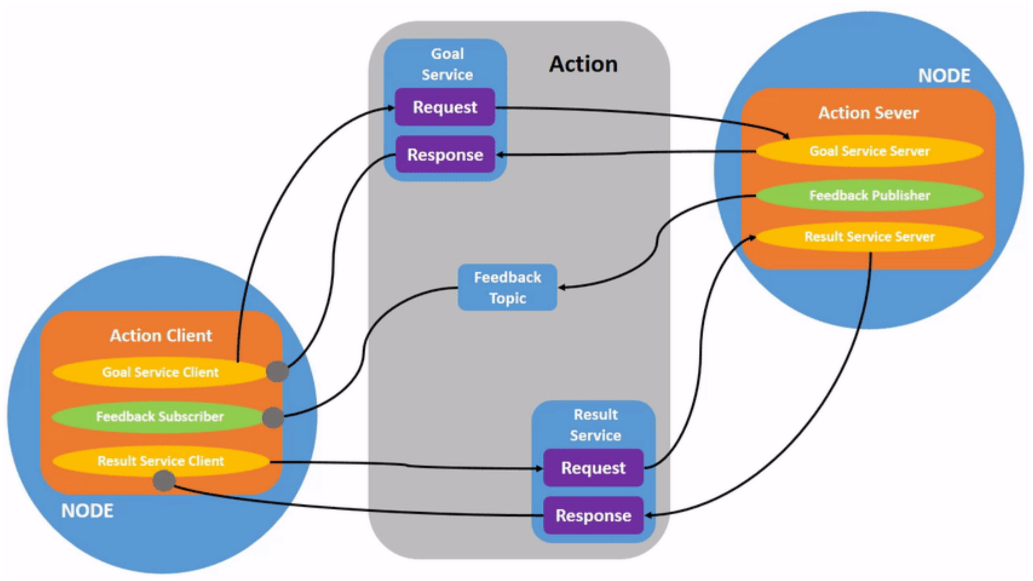


Figure 5.4.: A ROS 2 action, figure from [43].

is sent until the goal is completed, which will trigger the result service server to send a response message to the result service. The result service will then send the result message to the result service client and the action is completed.

5.2. ROS-Industrial

According to [42] the majority of robots sold in North America are set to do welding, material handling or dispensing/coating. The percentage of sold robots in North America set to do other operations has decreased from 1993 to 2013 before increasing slightly in 2017. Even after the increase under 10% of sold robots in 2017 was set to do other tasks than the three most popular mentioned above. Broadening the field of use for industrial robots requires research which is not profitable for most companies due to its expensive and time consuming nature.

ROS-Industrial (ROS-I) is a collection of software packages that aims to apply ROS to manufacturing automation. It as an open source project initiated by Yaskawa Motoman Robotics, Southwest Research Institute and Willow Garage [41]. The Git repository for ROS-I was founded by Shaun Edwards and can be found at [33].

The repository contains packages for different components commonly found in industrial robotics like grippers, sensors and robot manipulators [40]. This gives researchers a framework for quickly building new robot applications by using the pre-made packages. With this framework more time can be spent on developing new solutions, as the need for implementing existing solutions are negated. By reducing the time researchers need to develop new robot applications, ROS-I hopes to make it economically feasible for companies to develop new robotic solutions. Over time this will hopefully lead to robots being deployed in a wider variety of tasks than what is seen today.

Packages in ROS-I can either be general or vendor specific. The general packages can be deployed on all robots, while the vendor specific ones are specially made for robots from Yaskawa, ABB, Fanuc etc.

5.3. ROS2 control

This section is based off the presentation found at [55] regarding ROS control and the documentation at [26] regrading ROS2 control. The slides from the presentation can be found at [56].

In 2004 Willow Garage developed the `pr2_control_manager`, found at [31], for real time control of the PR2 robot seen in Figure 5.5 from [48]. Due to popular demand for controlling other robots with the same method, ROS Control was developed in 2012 by PAL robotics together with the community. ROS Control can be regarded as the robot agnostic version of `pr2_control_manager`, and has the following goals:

- Lower entry barrier for exposing hardware to ROS
- Promote reuse of control code
- Provide ready-to-use tools
- Real-time ready implementation

To summarise, ROS Control aims to fill the gap between the users custom software and the physical robot system. Filling this gap allows developers to focus on research and writing unique code instead of spending time rewriting code for interacting with hardware. After learning the limitations and weaknesses of ROS control, ROS2 control was developed for ROS2. ROS2 control kept the basic idea of ROS control while improving existing features and implementing new components. The ROS2 control framework is divided into five different repositories:

- `ros2_control` [34]
- `ros2_controllers` [35]
- `control_toolbox` [29]

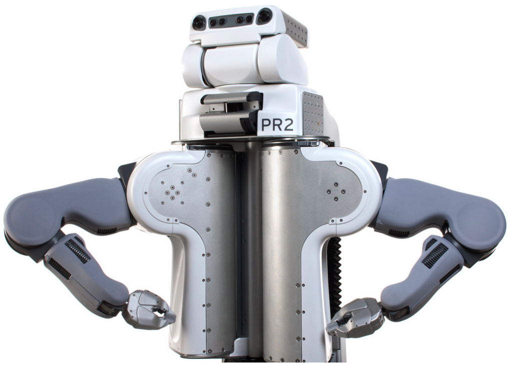


Figure 5.5.: The PR2 robot, image taken from [48].

- `realtime_tools` [32]
- `control_msgs` [28]

For ROS2 Control to be suitable for a variety of applications, it has a modular architecture where components can be changed individually without having to change the rest of the system. The general ROS2 control architecture can be seen in Figure 5.6 from [34]. A ROS2 control system consists of several key components: controllers, the controller manager, the resource manager and hardware components. Each of these components and how they work together will now be presented.

5.3.1. Hardware Components

The physical components that are to be controlled are called the hardware resources, which can be sensors, robot joints, grippers etc. All hardware resources are abstracted by using hardware components in ROS2 control. The hardware components main task is to communicate with the hardware resources by reading and writing information. To describe a complete robot cell, hardware components are divided into three categories: sensors, systems and actuators.

Sensors are typically encoders for measuring joint angles or force-torque sensors but also includes all other types of sensors. Common for all sensors is their read-only property as their only task is to gather information. A system represents complex hardware with multiple DOF, for example an industrial robot. The system can both be read from and written two, and is used when the robot as a whole needs to be represented as one entity. This can be practical for sending commands to a system as a whole instead of its individual components. An actuator on the other hand represents a single component of a system, usually with one DOF.

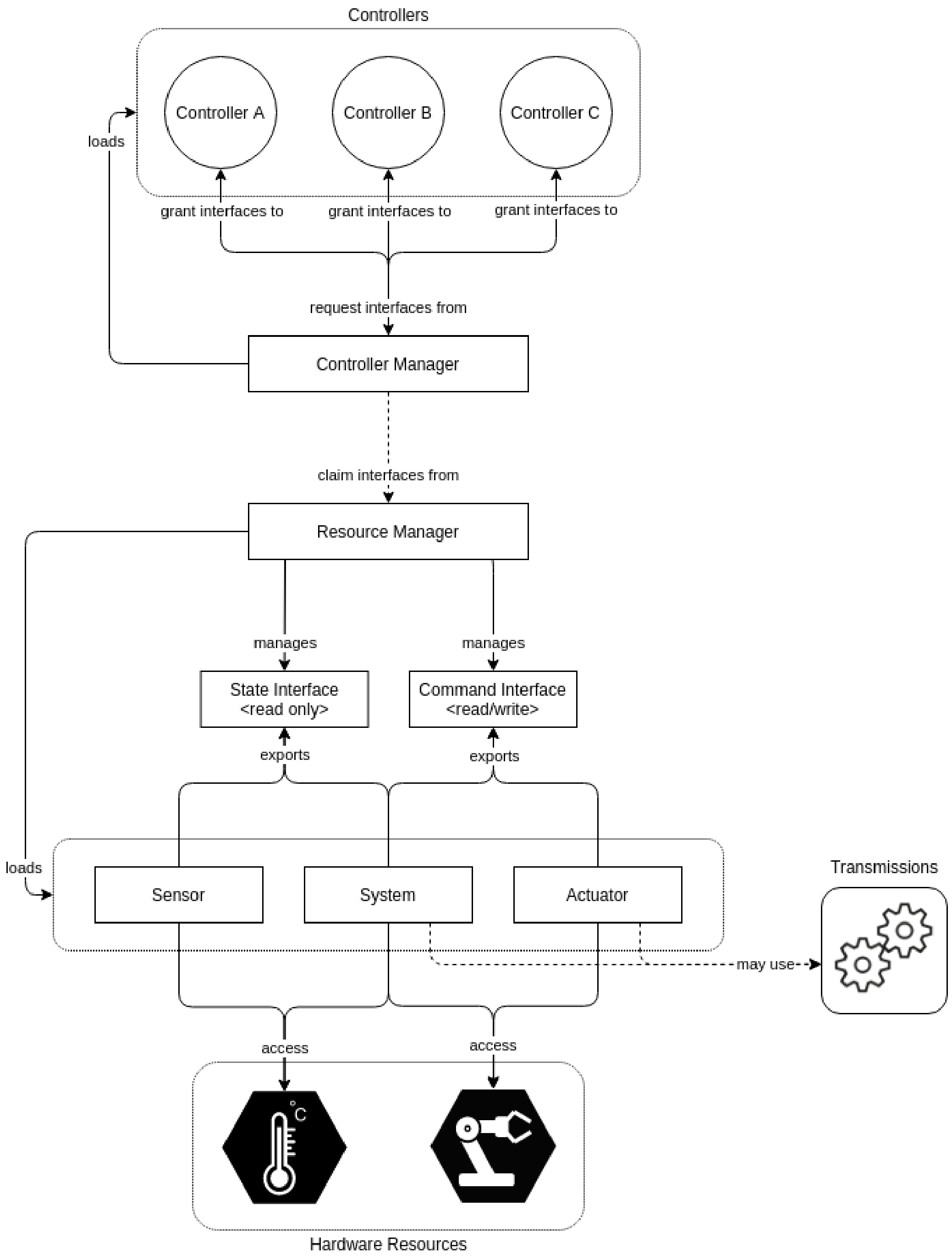


Figure 5.6.: The ROS2 control architecture, figure taken from [34].

Examples of typical actuators are motors, valves and linear actuators. Actuators have both read and write capabilities, although reading is not a requirement. They can be used when there is need for controlling individual components in a system or when the hardware to be controlled is not too complex.

5.3.2. Resource Manager

The resource manager abstracts hardware even further by managing and monitoring hardware components in the system. Sensor data as well as data from systems or actuators are obtained trough the state interface, and commands are sent trough the command interface as seen in Figure 5.6. Both of these interfaces are controlled by the resource manager. At startup the resource manager is responsible for loading the hardware components of a system from for example a URDF file.

Different hardware components may require different forms of communication. A system containing a six DOF industrial robot will require different types of commands than a single DC-motor. The resource manager provides flexibility in the system by being able to communicate with all hardware components regardless of the type of communication required. This allows the user to utilise already premade hardware components in new systems, reducing development time for new robot applications.

5.3.3. Controller Manager

The controller manager is what brings together the hardware side of the system and the controllers. Similar to how the resource manager loads hardware components, the control manager is responsible for loading and maintaining all the controllers in the system.Communication between the controller manager and the controllers will be covered further later in this section. The controller manager is also responsible for checking that the resources required by each controller is available. Control loops for the architecture is executed from the controller managers `update()` method where data is obtained from the resource manager and sent to the controllers before updated commands are obtained from the controllers and sent to hardware trough the resource manager.

5.3.4. Controllers

The controllers are the calculation nodes of a ROS2 control system. From the controller manager they receive data from hardware components and execute calculations before returning the results back to the controller manager. A system can have multiple controllers with different tasks, and each controller is loaded and unloaded by the controller manager as needed. The life cycle of a controller is based on that of a node in ROS2 which is a state machine. A flowchart of the life cycle can be seen in Figure 5.7 taken from [13].

As seen in Figure 5.7 the controllers can be set to four different states, and the transitions between them are controlled by functions in the controller manager.

When the controllers are first loaded by the controller manager they are set to the unconfigured state. In this state the controller does not preform any work, and awaits configuration by the controller manager. Configuring the controller includes setting parameters and defining which topics the controller is supposed to subscribe and publish to.

After begin configured the controller goes to the inactive state. In this state the controller does not do any work, but it is configured and ready to be activated. From here the controller can be cleaned and reconfigured or be set into the active state.

When set to the active state the controller will execute its task, which is usually receiving and outputting data from and to the controller manager. A controller can be switched on and off by switching in between the active and inactive state.

From all three states explained so above the controller manager has the ability to call the function `shutdown()`, which will send the controller into the finalised state. From this state the controller is destroyed and reaches its end of life.

There are several pre made controllers that are available in the `ros2_controllers` repository [35]. These are controllers for common robotic operations like trajectory planning and velocity control. In addition to the pre-made controllers, users are able to write their own controllers to fit their needs.

5.4. Moto

Motoman robots from YASKAWA can be controlled with ROS commands by installing the ROS-I code found at [30] on the robot controller. This will enable the robot controller to communicate with a computer trough an Ethernet cable. The `moto` library found at [53] is a Python API for controlling Motoman robots without any knowledge of ROS-I. It also adds functionality for real-time velocity control of Motoman robots. For real-time control an altered version of the ROS-I code has to be installed on the robot controller as well. This code can be found at [54]. This section will give an overview of the architecture of the`moto` library and explain essential classes and concepts used.

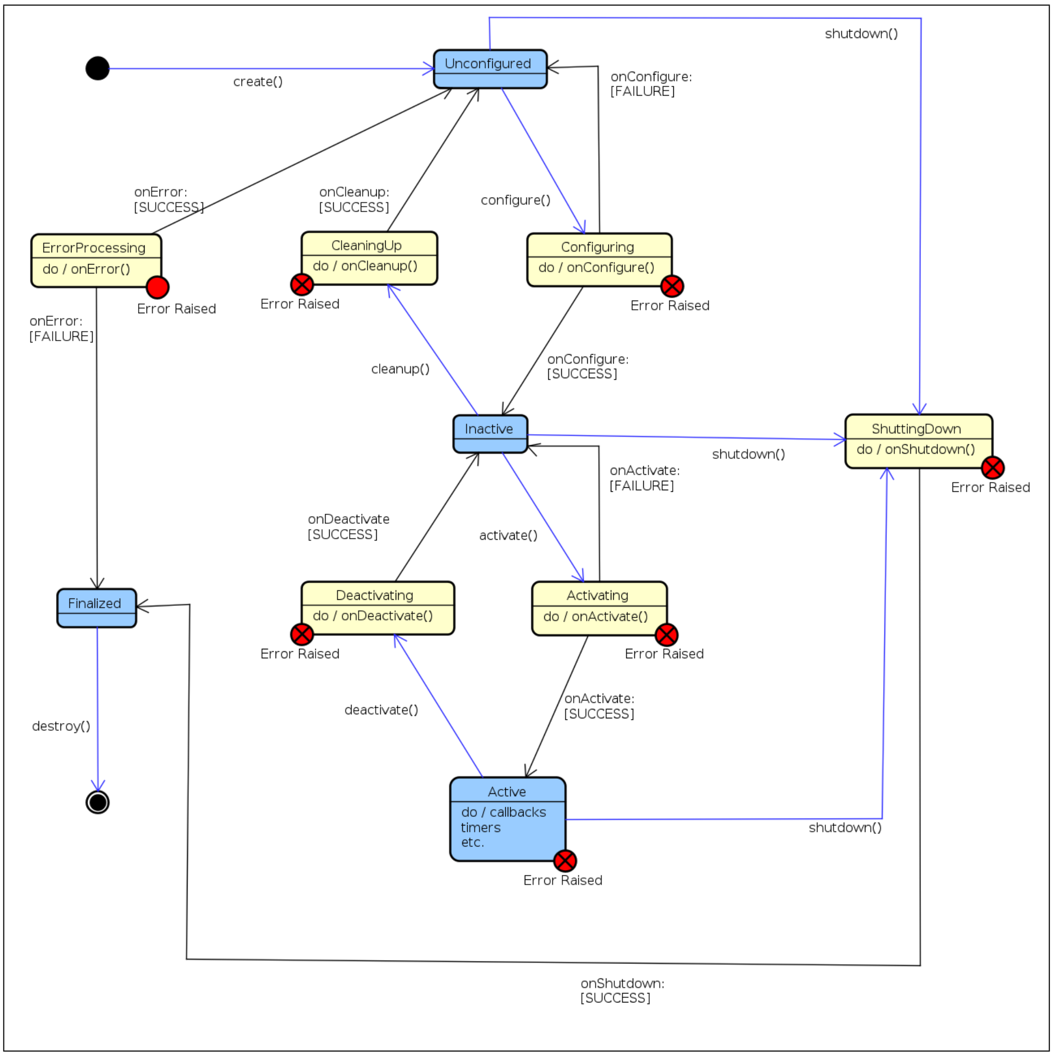


Figure 5.7.: The life cycle of a node, figure taken from [13].

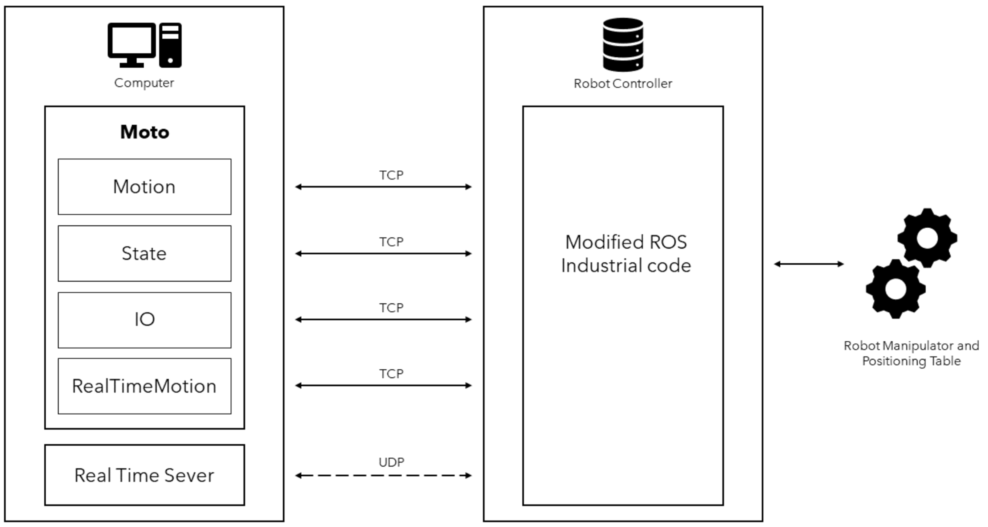


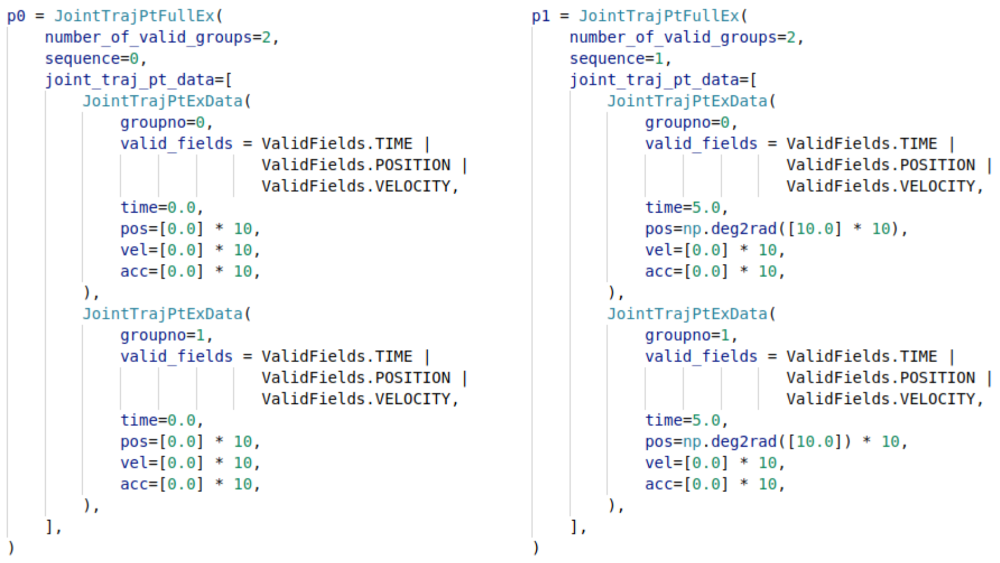
Figure 5.8.: The architecture of the `moto` library from [53].

5.4.1. Architecture

Figure 5.8 shows the general architecture of the `moto` library and how it communicates with the robot controller running the modified ROS-I code. The architecture of `moto` is built around the class `Moto` which represent the hardware present in a robot cell like a robot manipulator or positioning table. Each hardware component is defined as a control group, and up to four control groups can be included in one instance of `Moto`. As seen in Figure 5.8 `Moto` communicates with the robot controller using four TCP connections and one UDP connection. `Moto` contains four different classes each corresponding to one TCP connection. All four classes inherits from the class `SimpleMessageConnection`. This class establishes a TCP connection to the robot controller to allow communication. The UDP connection is used for real-time control. Each of the four classes connected to a TCP connection has their own member functions which together make up all the functionality of `Moto`. The four different classes will now be presented along with their functionality.

Motion

`Motion` contains functions regarding the movement of a control group. This includes support functions like turning on or off the servos and checking if the control group is ready for movement. To move a control group the system has to be set in trajectory mode, which is a mode where the system is ready to receive trajectory points before generating and executing a trajectory from these. The structure of a trajectory point can be seen in Figure 5.9, and will now be explained in detail.



```
p0 = JointTrajPtFullEx(
    number_of_valid_groups=2,
    sequence=0,
    joint_traj_pt_data=[
        JointTrajPtExData(
            groupno=0,
            valid_fields = ValidFields.TIME |
                           ValidFields.POSITION |
                           ValidFields.VELOCITY,
            time=0.0,
            pos=[0.0] * 10,
            vel=[0.0] * 10,
            acc=[0.0] * 10,
        ),
        JointTrajPtExData(
            groupno=1,
            valid_fields = ValidFields.TIME |
                           ValidFields.POSITION |
                           ValidFields.VELOCITY,
            time=0.0,
            pos=[0.0] * 10,
            vel=[0.0] * 10,
            acc=[0.0] * 10,
        ),
    ],
)
```

```
p1 = JointTrajPtFullEx(
    number_of_valid_groups=2,
    sequence=1,
    joint_traj_pt_data=[
        JointTrajPtExData(
            groupno=0,
            valid_fields = ValidFields.TIME |
                           ValidFields.POSITION |
                           ValidFields.VELOCITY,
            time=5.0,
            pos=np.deg2rad([10.0] * 10),
            vel=[0.0] * 10,
            acc=[0.0] * 10,
        ),
        JointTrajPtExData(
            groupno=1,
            valid_fields = ValidFields.TIME |
                           ValidFields.POSITION |
                           ValidFields.VELOCITY,
            time=5.0,
            pos=np.deg2rad([10.0]) * 10,
            vel=[0.0] * 10,
            acc=[0.0] * 10,
        ),
    ],
)
```

Figure 5.9.: Two trajectory points from the `moto` library.

Referring to the figure it is seen that a trajectory point is an instance of the class `JointTrajPtFullEx`. The number of control groups in the instance of `Moto` that is to be controlled is defined in `number_of_valid_groups`, while `sequence` helps the controller interpret which point comes before another. Sequence zero will be executed before sequence one etc. The last variable in the class is a list containing instances of the class `JointTrajPtExData`, which is where information regarding the movement of a control group is stored.

In `JointTrajPtExData` relevant control groups are defined by `groupno`, which has been set in `Moto`. `velid__fields` contains information on whether the position, velocity and acceleration data can be trusted. This is mostly relevant for feedback from the robot to determine if a measurement is correct or not. `Time` defines when the robot is supposed to be at the given position, while `pos`, `vel` and `acc` contains the position velocity and acceleration of the control group at that time. These are all defined as a vector of 10 elements to support control groups with up to 10 DOF. All movement is defined in *radians*, *radians/second* or *radians/second*^{2}. When generating a trajectory by sending trajectory points, the first point always have to be the current position of the control group. By sending the two points in Figure 5.9 both control groups will move from the home position `p0` to a position where all joint angles are 10 degrees. This movement will occur over five seconds, since `time` in `p1` is set to five. The system keeps track of time only while moving, so the timer stops when the system is standing still. The duration of a new movement from `p1` would then be set as `time = 5 + duration`.

Individual control groups can also be controlled by sending trajectory points which are instances of the class `JointTrajPtFull`. These are similar to the trajectory points seen in Figure 5.9, but does not include a list of trajectory points for all control groups. Instead they only contain time, position, velocity and acceleration data for one single control group.

State
`State` is the class containing functions that allow `Moto` to send feedback on the state of the system to the user. Different information form the system can be gained from the function `robot_status()`, which returns an instance of the class `RobotStatus`. This class contains information like the state of the servos, the presence of any alarms in the system, the operation mode of the teach pendant (Section 7.3) etc. Information regarding the position, velocity and acceleration of each joint in a control group are gained trough the function `joint_feedback_ex()`. This will return an instance of `JointFeedbackEx` which contains information on all joints in all control groups. Similar to trajectory points, feedback from one individual control group can be gained from the function `joint_feedback()`.

A bug in the feedback function results in a velocity 27 *rad/sec* being returned for joint joint T. This will raise an alarm in the robot controller and stop the system. It is also worth noting that the system does not take the direction of the velocity into account and will always return the absolute value of the velocity.

IO
The robot controller contains memory addresses that hosts software and stores parameters. The `IO` class in `Moto` is responsible for interacting with these memory addresses. As with all memory management this task consists of writing and reading to memory. The `IO` class can either read/write bits or bytes in the robot controller. Most of the memory addresses in the controller are read-only as they are used by the robot controller software and therefore inaccessible by the user. There are exceptions where memory addresses are writable. According to [30] these are 10010 and up and 27010 and up. These are the "Universal/General Outputs" and the "Network Inputs" respectively.

RealTimeMotion
The final class is `RealTimeMotion` which sets up the system for real-time control. The class contains three member functions:

- `connect()` establishes the TCP connection with the robot controller, enabling the use of the other two member functions in `RealTimeMotion`.
- `start_rt_mode()` will set the system in real-time mode by opening a UDP connection as shown in Figure 5.8. When real-time mode is not enabled the

computer acts as a client that sends requests to the robot controller that acts as a server. In real-time mode this dynamic is changed so that the computer acts as a server and the robot controller as a client. A server code then has to run at 250 Hz on the computer, giving joint commands to the robot controller. The real-time server will be further explained in Section 5.4.3.

- `stop_rt_mode()` closes the UDP connection and takes the system out of real-time mode.

For real-time mode to work properly, the system has to be set to trajectory mode without having an active TCP motion connection. This is further discussed in Section 6.2.

5.4.2. Simple Message

All communication that happens with TCP utilise the class `SimpleMessage` found in ROS-I. This message format consist of two classes, a `Body` and a `Header`.

`Header` contains metadata on the content of the message, and will have the same structure independent on what the contents of the message is. Three variables are central in a header, and all inherits from the class `Enum`:

- `msg_type` is an instance of the `MsgType` This variable describes the content of the message, which can for example be a `JointTrajPtFullEx` or a `RobotStatus` like mentioned previously.
- `comm_type` is an instance of the class `CommType`. This class describes which type the message is. The most used ones are `SERVICE_REPLY` and `SERVICE_REQUEST` representing a reply from or request to the robot controller.
- `reply_type` is an instance of the class `ReplyType`. This tells the system if the sent message was a success, failure or invalid.

The body of the message contains the actual data of the message, and will vary in structure depending on the message type. Examples of contents in the body of a message are `JointTrajPtFull`, `RobotStatus` and `JointFeedbackEx`.

5.4.3. Real-Time Mode

As mentioned previously, when the system is set to real-time mode, the computer acts as the server and the robot controller as the client. The robot controller will then listen for joint velocity commands that are being sent from the server.

An example of such a server can be found in [53] in the file `test_rt_motion_server.py`. When running this file a UDP server is opened that awaits for the robot controller to start listening for commands. After the robot controller starts to listen, communication between the computer and the robot controller starts at a frequency of 250 Hz. This communication also use the `SimpleMessage` class. The server runs in a loop where first feedback is gained from all control groups in `Moto`. A reply message is then constructed where the `Body` is an instance of the class `MotoRealTimeMotionJointCommandEx`. This class contains commands for all joints in all control groups. Currently only velocity control is possible, but position and acceleration control is to be implemented. The command is then sent to the robot controller before it is executed.

5.5. eTaSL/eTC

The expressiongraph-based Task Specification Language (eTaSL) along with the expressiongraph-based Task Controller (eTC) compose a framework for constraint-based robot programming. This framework is presented in [2] and will be summarised in this section. Documentation for eTaSL/eTC can be found at [1]. Unless stated otherwise this section is based off the material found in [2].

While eTaSL provides a language for representing robot tasks using constraints, the robot controller eTC solves a numerical optimisation problem for the given constraints and yields joint velocities to the robot being controlled. This section will first present the architecture of eTC, before presenting important aspects in eTaSL. This section is copied from the project report that is attached to this thesis [6].

5.5.1. eTC

The architecture of eTC focuses on the 5C's found in [37], and separates computation, coordination, configuration, composition and communication. Figure 5.10 from [2] shows that the controller has been separated into three layers, each with their own separate task.

The first layer is the specification layer where the context is built. This is a complete description of the robot task. Robot geometry can be loaded into the specification layer through a URDF file. The task can be specified with a C++ task specification API, other task specification methods like iTaSC [9] or more commonly with eTaSL. After specifying a task, the context is sent as input to the second layer.

The second layer is the solver layer, where a context is converted to a numerical

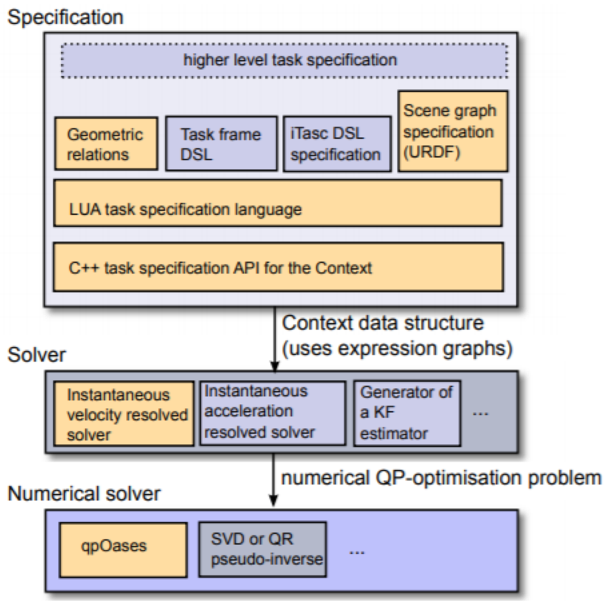

Specification
higher level task specification
Geometric relations
Task frame DSL
iTasc DSL specification
Scene graph specification (URDF)
LUA task specification language
C++ task specification API for the Context
Context data structure (uses expression graphs)
Solver
Instantaneous velocity resolved solver
Instantaneous acceleration resolved solver
Generator of a KF estimator
...
numerical QP-optimisation problem
Numerical solver
qpOases
SVD or QR pseudo-inverse
...

Figure 5.10.: The layers used in the eTC architecture, highlighted boxes are the implemented solutions. Figure from [2].

optimisation problem. Referring to Figure 5.10 it can be seen that the current implementation uses a solver to control the joint velocities of the robot. eTC can also be used for acceleration control or torque control, but for this section velocity control is assumed. The optimisation problem has a goal of minimising the variable $\boldsymbol{x}$ in the following formula:

$$\boldsymbol{x}^T \boldsymbol{H} \boldsymbol{x} \tag{5.1}$$

with respect to the upper and lower limits $\boldsymbol{U}$ and $\boldsymbol{L}$ such that:

$$\boldsymbol{L}_A \leq \boldsymbol{A}\boldsymbol{x} \leq \boldsymbol{U}_A \tag{5.2}$$

$$\boldsymbol{L} \leq \boldsymbol{x} \leq \boldsymbol{U} \tag{5.3}$$

Where $\boldsymbol{A}$ is the task Jacobian and $\boldsymbol{L}_A$ and $\boldsymbol{B}_A$ are task limits. The variable $\boldsymbol{x}$ is written as:

$$\boldsymbol{x} = \begin{bmatrix} \dot{\boldsymbol{q}} \\ \dot{\boldsymbol{\chi}}_f \\ \boldsymbol{\epsilon} \end{bmatrix} \tag{5.4}$$

Where $\dot{\boldsymbol{q}}$ is the joint velocities, $\dot{\boldsymbol{\chi}}_f$ is the feature variable velocities and $\boldsymbol{\epsilon}$ is a slack variable to enable lower priority constraints to act on the system. The matrix $\boldsymbol{H}$ seen in (5.1) is written as:

$$\boldsymbol{H} = \begin{bmatrix} \mu \boldsymbol{W}_r & \boldsymbol{0} & \boldsymbol{0} \\ \boldsymbol{0} & \mu \boldsymbol{W}_f & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{0} & \mu \boldsymbol{I} + \boldsymbol{W}_s \end{bmatrix} \tag{5.5}$$

Where $\boldsymbol{W}_r$ and $\boldsymbol{W}_f$ are weights for the robot joint space and feature space respectively. $\boldsymbol{W}_s$ is a collection of the weight variable in each constraint, which will be presented later, while μ is a numerical value that can be used to tune the system. $\boldsymbol{W}_r$ and $\boldsymbol{W}_f$ will mostly affect the system when task redundancy occurs. In this case they will affect how the robot moves in the null-space of the task, i.e. how the rest of the robot is moved while holding the end-effector in the desired configuration.

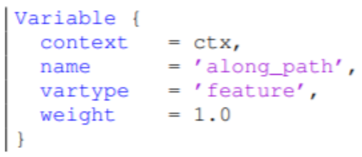
After the optimisation problem is defined it is sent into the third and last layer where it is solved by a numerical solver, in this case the qpOASES solver. For each control cycle joint data are taken in as an input to the solver layer where the optimisation problem is generated before being sent to the numerical solver. The computed joint velocities are then sent to the robot. This process is repeated each time cycle.

5.5.2. eTaSL

eTaSL is a Lua based language using abstractions to communicate with the underlying C++ API generating the context in the specification layer. The context consists of different elements that will now be explained, starting with variables.

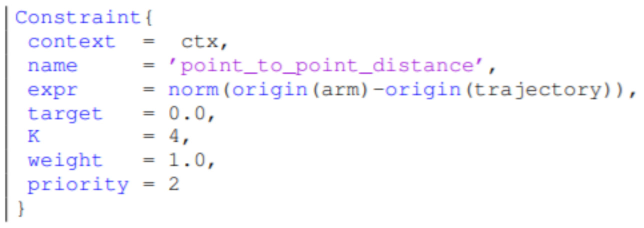
An example of a variable can be seen in Figure 5.11 taken from [2]. Variables are set by the user and is required to be connected to a context and have a name. They also need a weight which will affect the way they are handled by eTC. The `vartype` argument decides what variable type the variable is. Robot joint variables determines the joint angles in the robot, the time variable keeps track of time and a feature variable can be used more freely to express movements.

Another type of variable is the expression variable. This is a variable containing an expression graph, which is a function often representing geometric relations



```
Variable {
  context    = ctx,
  name       = 'along_path',
  vartype    = 'feature',
  weight     = 1.0
}
```

Figure 5.11.: Example of a variable in eTaSL. Figure from [2].



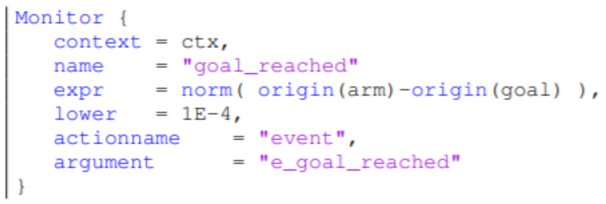
```
Constraint{
 context   =  ctx,
 name      = 'point_to_point_distance',
 expr      = norm(origin(arm)-origin(trajectory)),
 target    = 0.0,
 K         = 4,
 weight    = 1.0,
 priority = 2
}
```

Figure 5.12.: Example of a constraint in eTaSL. Figure from [2].

between rigid bodies. Expression variables are used as an argument when defining constraints and monitors, this will be presented next.

An example of a constraint can be seen in Figure 5.12 taken from [2]. Like variables it has to be connected to a context, have a name and a weight. In addition to this it uses an expression variable to define a function. The goal of the expression variable is to reach the value of `target`, in this case 0.0. How the expression variable will reach the target is determined by the value of `K`, which is used in the controller. Lastly the constraint has a priority. This allows the user to define a hierarchy were constraints with high priority are more important than the ones with a lower priority. For multiple constraints with equal priority, their importance is measured by the `weight` argument.

Monitors can detect when a certain condition is met and notify the system of this event. An example of a monitor can be seen in Figure 5.13 taken from [2]. It is seen that the monitor observes an expression variable defined by `expr`. As this expression exceeds some limit `lower` or `upper`, an action is triggered and the argument is sent as an output. The name of the action and the argument are set in `actionname` and `argument` respectively.



```
Monitor {
    context = ctx,
    name    = "goal_reached"
    expr    = norm( origin(arm)-origin(goal) ),
    lower   = 1E-4,
    actionname     = "event",
    argument       = "e_goal_reached"
}
```

Figure 5.13.: Example of a monitor in eTaSL. Figure from [2].

Chapter 6.

MotoTester

The moto_tester library [5] is a Python library made for testing properties of the code presented in Section 5.4 mostly focusing on real-time performance. For this thesis, real-time performance is measured by response time and latency. Response time is defined as the time it takes from a command is sent from the computer until the control group starts to react to that command. Latency is defined as the time it takes for a control group to reach a specified joint velocity after the command has been sent. This section will present the most important files form the library and some of the functions they contain.

6.1. moto_tester.py

This file implements the class MotoTester which inherits from the Moto class presented in Section 5.4. The class was made for testing the functions found in the three following parts of Moto: Motion, State and IO. Member functions in MotoTester worth noting are logger() and test_io_latency(). logger() moves a control group to a specified position while recording the joint positions during the movement. All joint positions are stored in a CSV file. test_io_latency() writes and reads a memory address to test the io-latency of the system. This is defined as the time it takes from the write command is sent, until the bit/byte actually changes value.

The development of MotoTester was discontinued, as the behaviour of the real-time part of the system is more relevant for this system.

6.2. rt_setup.py

As mentioned in Section 5.4.1 the system needs to be in trajectory mode without an active motion connection for real-time control to work properly. The purpose of `rt_setup.py` is to put the system in a state where real-time control can be executed.

When setting up the system for real-time control an instance of `Moto` is first created. If nothing else is specified, `Moto` will create TCP connections for `Motion`, `State` and `IO`. After initiating the instance of `Moto`, trajectory mode is started. An open door to the robot cell or the robot not being in remote mode (Section 7.3) will stop the system from going into trajectory mode. This is checked and if any problem occurs the error code is sent to the user along with the option to retry the setup.

When the system is set to trajectory mode properly, the script will close using the `exit()` function from Python. Closing the script in this way without properly disconnection the TCP sockets will leave the system in trajectory mode without any proper connection to the computer. This leaves the system in a state where it is ready to run in real-time. The next section will explain how this is done.

6.3. moto_tester_rt.py

Similarly to `moto_tester.py` this file also introduces a new class that inherits on the `Moto` class. The class defined in this file is the `MotoTesterRt`, and is a class made for testing the real-time performance of `moto`. Unlike `Moto`, `MotoTesterRt` does not establish TCP connections for `Motion`, `State` and `IO`. Instead a TCP connection is established for `RealTimeMotion`, so that member functions for real-time control can be used like explained in Section 5.4.1.

There is only one member function in `MotoTesterRt` which is `check_frequency()`. The purpose of this function is calculating the control frequency of the system when running in real-time. This is done by initiating real-time mode and letting it run for two seconds. The amount of commands sent during those two seconds are then used to calculate the control frequency.

6.4. rt_server.py

The real-time server presented here is based off the one found at [53] which was explained in Section 5.4.3. Some alterations are made to the server which will now be explained. As mentioned in Section 5.4.1 the robot controller gives joint velocities as absolute values. This became problematic when trying to implement

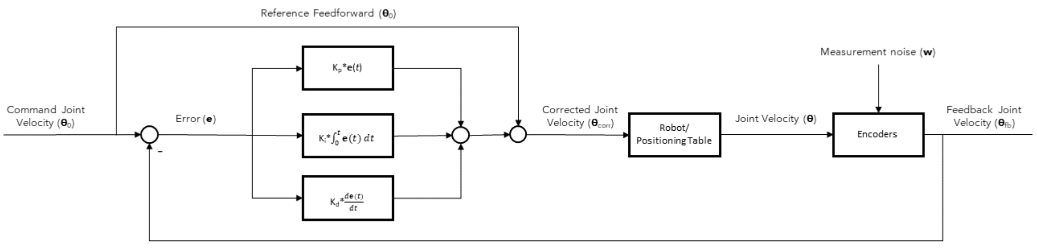

Reference Feedforward ($\boldsymbol{\theta}_0$)
Measurement noise ($\mathbf{w}$)
Command Joint Velocity ($\boldsymbol{\theta}_0$)
Error ($\mathbf{e}$)
$K_p * \mathbf{e}(t)$
$K_i * \int_0^t \mathbf{e}(t)\,dt$
$K_d * \frac{de(t)}{dt}$
-
Corrected Joint Velocity ($\boldsymbol{\theta}_{corr}$)
Robot/ Positioning Table
Joint Velocity ($\boldsymbol{\theta}$)
Encoders
Feedback Joint Velocity ($\boldsymbol{\theta}_{fb}$)

Figure 6.1.: A block diagram showing the PID controller with reference feedforward implemented in the real-time server.

a PID controller, and therefore had to be fixed. It was done by comparing the position of each joint to the position of the same joint in the previous control cycle. From this a simple `if` statement decides in which direction the joint is moving. The positive direction was defined to be the same as the directions shown on the control buttons of the teach pendant, as seen in Figure 7.4.

The next and biggest change to the real-time server is the implementation of a PID controller. This allows the user to tune the system to an appropriate behaviour depending on the task that is to be executed. In this system the PID controller consists of ten different PID controllers, one for each possible joint in a control group. Each of the controllers are tuned individually. The vectors `K_p`, `K_i` and `K_d` all have ten elements, each corresponding to a PID controller. In addition to the PID controller, feedforward of the reference joint velocity is implemented. A block diagram showing the control system can be seen in Figure 6.1. Currently the server only supports one control group using the PID controller at a time.

When tuning the controller a modified version of the ZN-method from Section 3.5 was used. While tuning no constant offset from the reference value was observed, so an integral controller was deemed obsolete. As a test some integral gain was applied and this only destabilised the system and made it less reactive. In reality the PID controller then became a PD controller, but for the remainder of this thesis it will be called a PID controller as it has the option of applying integral effect. The ZN-method has no specific tuning for PD controllers and the solution became tuning the controller like a pure P controller before experimenting with increasing derivative gains to find the one yielding the best results. As the velocity signal from the controller is not without noise, the D controller will at some point start amplifying noise instead of dampening oscillations. A graph showing the development of the step response from joint S using different derivative gains can be seen in Figure 6.2.

For testing the system and tuning the PID controller, as step response was also implemented in the server. The step response is defined by using the number of

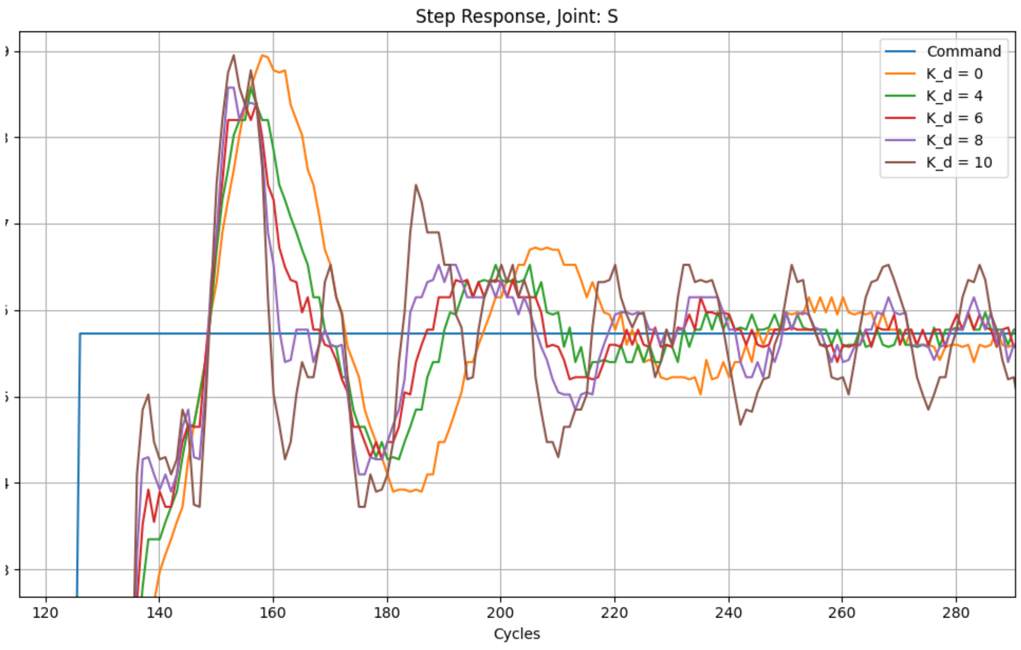


Figure 6.2.: The effect of changing the derivative gain shown for a step response from joint S.

control cycles as a time variable, and sending velocity commands based on this information.

Lastly a logger similar to the one seen in 6.3 was implemented, the difference being that this logger collects data continuously while the system is running in real-time mode. All data is written to a CSV file. For the experiments done in this thesis the command joint velocity sent from the computer and feedback joint velocity received from the robot controller was logged.

6.5. data_plotter.py

Functions for plotting the data contained in the CSV files are located in `data_plotter.py`. Different plotters are made for real-time mode and running the system regularly as the size of the CSV file is different for the two. `data_plotter()` and `data_plotter_rt()` plots the joint velocity of a given control group joint in respectively normal and real-time mode. The latter function also plots the command velocity sent from the computer to the robot controller.

Similarly to `data_plotter_rt()` where a file is plotted, `data_multiplotter_rt()` can plot all files in a directory. These are all plotted at the same time but in different plots. This function is useful for searching for specific files in a directory. Same as for `data_plotter_rt()` this function only works with files that are collected while the system is running in real-time mode.

For comparing different plots, `compare_plots()` can be used. This function takes in a list of CSV files from running in real-time and plots them together. Only one joint from each file is plotted, and it has to be the same joint for each file. The feedback velocity of all files are plotted, and additionally the command velocity of the first file is plotted. This function was made for comparing the step response of the system to tune the PID controller.

6.6. utilities.py

This last file contains miscellaneous functions for different uses in the system. The most important are the functions calculating latencies and response times in the real-time system from CSV files. There are two different functions for doing this.

`calculate_latency()` compares the command velocity from the computer to the feedback velocity gained from the robot controller. The latency is the defined as the distance in control cycles when both velocities are at zero. This is used for calculating the latency on a continuously changing signal where the velocity alters between negative and positive.

`step_analysis()` is used for calculating both latency and response time of a step function. Response time is calculated by observing the amount of cycles that passes before a joint starts moving after the joint velocity command is sent from the computer. Latency is calculated by observing how many control cycles passes after the command step joint velocity is sent until the joint reaches that joint velocity.

As the system runs on 250Hz both latency and response time can easily be converted from control cycles to ms. It is done by multiplying the amount of control cycles measured with the duration of a control cycle which is $4ms$.

Chapter 7.

Hardware

In this chapter the hardware used in experiments for this thesis will be presented. All hardware presented in this chapter forms a robotic welding cell located at Manulab at the Department of Mechanical and Industrial Engineering, NTNU Trondheim. Figure 7.1 shows the content of the welding cell.

7.1. Robot Manipulator

The robot manipulator used in the cell is a YASKAWA Motoman GP25-12. The technical specifications of the robot can be seen in Appendix B, obtained form [15]. To summarise quickly the GP25-12 is a six axis multi purpose industrial robot. It has a lifting capacity of 12 kg and is typically used in assembly, dispensing or material handling. The six joints are named as follows from the shoulder joint and outward: S, L, U, R, B and T.

From the specifications a simplified drawing of the robot has been made showing the screw axis of each joint in the end-effector frame and space frame like explained in Section 2.7. The simplified drawing has been copied from the attached project report [6], and can be seen in Figure 7.2.

The home position of the robot can then be represented by the transformation matrix:

$$\boldsymbol{M} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1.332 \\ 0 & 0 & 1| & 1.465 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{7.1}$$

From the simplified drawing it is possible to extract matrices for the screw axis

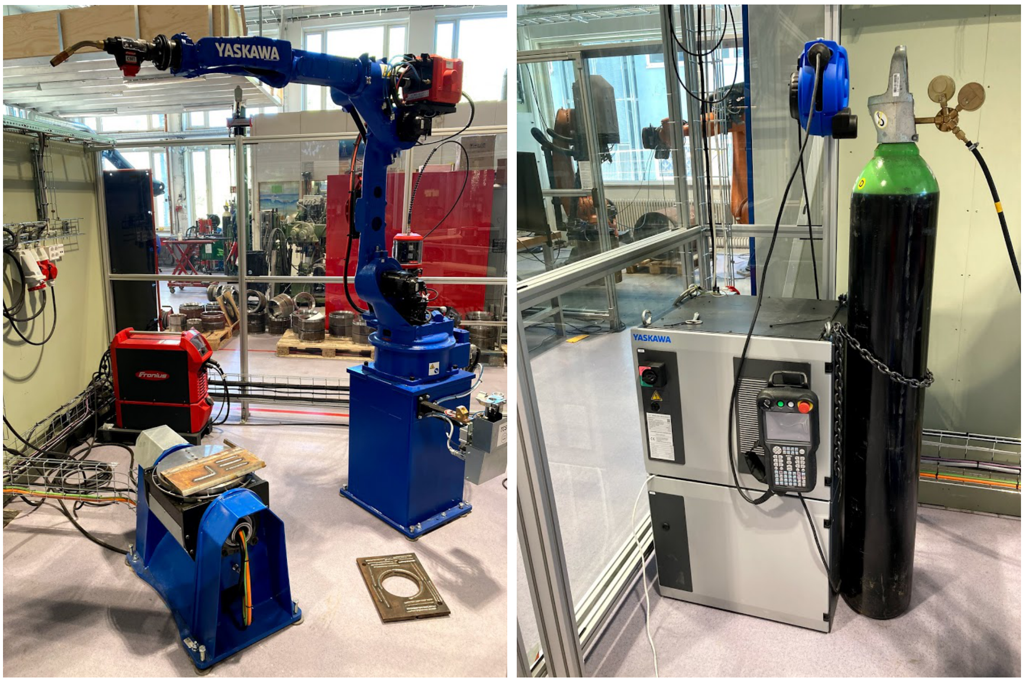


Figure 7.1.: Left: YASKAWA Motoman GP25-12 robot manipulator equipped with a Fronius TPS 400-i welding apparatus along with the YASKAWA MT1-500 S2HD positioning table. Right: YASKAWA YRC1000 robot controller with standard teach pendant and gas for welding.

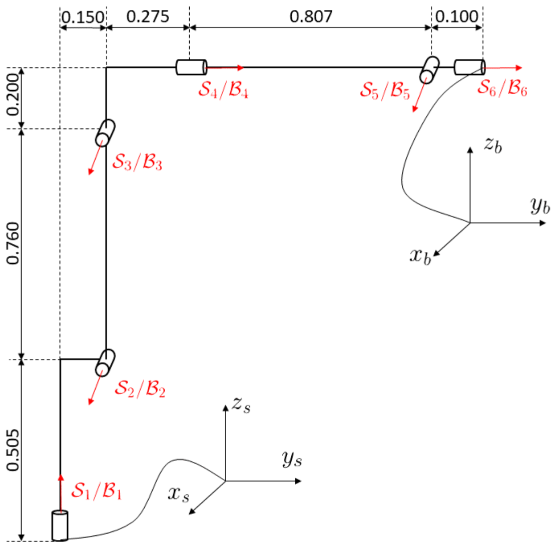

0.150
0.275
0.807
0.100
0.200
0.760
0.505
$\mathcal{S}_1/\mathcal{B}_1$
$\mathcal{S}_2/\mathcal{B}_2$
$\mathcal{S}_3/\mathcal{B}_3$
$\mathcal{S}_4/\mathcal{B}_4$
$\mathcal{S}_5/\mathcal{B}_5$
$\mathcal{S}_6/\mathcal{B}_6$
z_s
y_s
x_s
z_b
y_b
x_b

Figure 7.2.: A simplified drawing of the Motoman GP25-12 showing the screw axis in both the end-effector frame and space frame. Figure taken from the project report[6].

in both the space-frame frame and the end-effector frame, which are shown in 7.2 and 7.3 respectively.

$$\boldsymbol{\mathcal{S}} = \begin{bmatrix} 0 & 1 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & -1.465 & 0 & -1.465 \\ 0 & 0.505 & 1.265 & 0 & 1.465 & 0 \\ 0 & -0.150 & -0.150 & 0 & -1.232 & 0 \end{bmatrix} \tag{7.2}$$

$$\boldsymbol{\mathcal{B}} = \begin{bmatrix} 0 & 1 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ -1.332 & 0 & 0 & 0 & 0 & 0 \\ 0 & -0.960 & -0.200 & 0 & 0 & 0 \\ 0 & 1.182 & 1.182 & 0 & 0.100 & 0 \end{bmatrix} \tag{7.3}$$

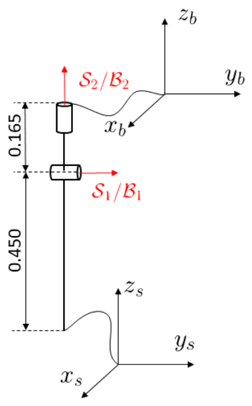


Figure 7.3.: A simplified drawing of the YASKAWA MT1-500 S2HD showing the screw axis in both the end-effector frame and space frame.

7.2. Positioning Table

In addition to the robot manipulator the cell is equipped with a YASKAWA MT1-500 S2HD positioning table. Technical specifications for the positioning table can be seen in Appendix C, taken from [16]. It is a compact positioning table with two axis and a load capacity of 500 kg. The joints on the positioning table are not named by the manufacturer, but in this thesis the lowest joint is referred to a s joint X and the upper one joint Y.

Same as for the GP25-12 a simplified drawing of the positioning table has been made, this can be seen in Figure 7.3. The home position $\boldsymbol{M}_{table}$ can be seen in (7.4) and the screw axis in both the stationary space frame and the end-effector frame can be seen in (7.5) and (7.6) respectively.

$$\boldsymbol{M}_{table} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0.615 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{7.4}$$

$$\boldsymbol{\mathcal{S}}_{table} = \begin{bmatrix} 0 & 0 \\ 1 & 0 \\ 0 & 1 \\ -0.450 & 0 \\ 0 & 0 \\ 0 & 0 \end{bmatrix} \tag{7.5}$$

$$\boldsymbol{\mathcal{B}}_{table} = \begin{bmatrix} 0 & 0 \\ 1 & 0 \\ 0 & 1 \\ 0.165 & 0 \\ 0 & 0 \\ 0 & 0 \end{bmatrix} \tag{7.6}$$

7.3. Robot Controller

The robot manipulator and positioning table are both controlled by a robot controller, which in this cell is the YASKAWA YRC10000. This section will first give a brief overview of the robot controller itself, before explaining the teach pendant and how it can be used to control both the robot manipulator and positioning table in different modes.

The YRC10000 is a versatile robot controller for several of YASKAWA's robot series including the GP-Series. In total it is capable of controlling up to 4 robots plus additional external axis up to a total of 72 axis [17]. The controller is either delivered with a Smart pendant for beginners, or a standard teach pendant for professional users. At MTP all YASKAWA robot controllers use the standard teach pendant, which will be presented next.

7.3.1. Teach Pendant

The standard teach pendant can be seen in Figure 7.4. Looking at the top left of the pendant there is a key which can be switched between three positions. Each position represents a different way of controlling the robot, teach mode, play mode and remote mode.

In teach mode all joint velocities are restricted and the door to the robot cell can be open while the robot is moving. This mode is typically used when making jobs, which are programs for moving the robot, using the teach pendant. In teach mode the robot is either controlled manually using the control buttons shown in

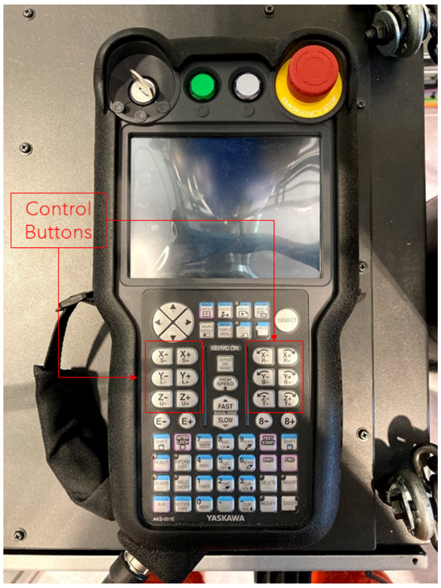


Figure 7.4.: The teach pendant connected to the YRC1000 robot controller. Control buttons for each joint are marked by the red squares.

Figure 7.4, or playing jobs with limited speeds. For all movement the dead man's switch needs to be pressed.

Play mode is the regular operating mode of the robot when it is controlled with the teach pendant. When in this mode the door to the cell is required to be locked to prevent harm to humans. Joints can now run at full speed and the robot can only be controlled by using pre-programmed jobs. Jobs can be run without the dead man's switch being pressed.

The last mode is the remote mode. When switched to remote mode the pendant will look for a saved job named `INTI_ROS`. A computer can then be connected to the controller by an Ethernet cable, and ROS commands can be used to control both the robot and positioning table. This assumes that some code supporting ROS commands has been installed in the robot controller. The standard `INIT_ROS` job can be seen in Appendix A.1.

High Accuracy Path Control Function In [19] a suggestion for reducing latency is to modify the `INIT_ROS` job seen in Appendix A.1 by including `HTRAJON` and `HTRAJOFF` as seen in Appendix A.2. This enables HTRAJ while running the system in remote mode.

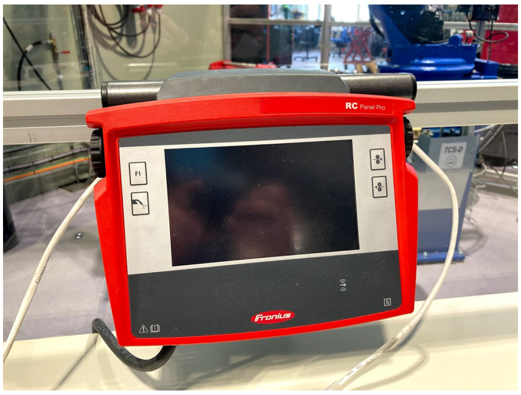


Figure 7.5.: Remote controller for the Fronius TPS 400-i.

7.4. Welding Apparatus

The GP25-12 is equipped with a Fronius TPS 400-i welding apparatus, as seen in Figure 7.1. Information presented in this section is gathered from [14]. The Fronius TPS 400-i is a digital welding system, enabling different welding packages to be installed on the apparatus. A welding package contains different welding methods. This makes the system adaptable to a variety of applications. Welding parameters and other settings can be controlled by the user on the power source itself which is equipped with a touch screen, or the remote controller seen in Figure 7.5. This enables the user to adjust welding parameters even outside the robot cell during welding operations.

The welding system is equipped with two wire feeders, one at the elbow of the robot and one in the welding gun itself. This allows for precise wire feeding, making the robot capable of precise welding methods like cold metal transfer (CMT). To avoid damage to the welding gun in case of the robot crashing, it is equipped with a magnetised socket. This will make the welding gun detach if the robot makes it collide with an obstacle in the cell. As the welding gun is not taken into account when calculating kinematics using (7.2), (7.3) and (7.1) a transformation matrix from the end-effector to the tip of the welding gun is required. This matrix is written as:

$$\boldsymbol{T}_{bt} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(54^\circ) & -\sin(54^\circ) & 0.450 \\ 0 & \sin(54^\circ) & \cos(54^\circ) & -0.084 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{7.7}$$

Figure 7.6.: The Micro-Epsilon scanCONTROL 2610-100 laser scanner.

7.5. Laser Scanner

The robot cell presented in this thesis does not yet have any mounted sensors, but is planned to have one installed. An equivalent robot cell is equipped with a Micro-Epsilon scanCONTROL 2610-100. This is a 3D laser scanner for scanning geometrical features. The scan is returned as a point cloud to the computer. A model of the scanner can be seen in Figure 7.6.

7.6. Computer Specifications

All tests presented in this thesis was run on a computer with the following specifications:

- OS: Ubuntu 20.04.2 LTS
- RAM: 3.8 GB
- CPU: Intel® Core$^{\text{TM}}$ i5-3210M CPU @ 2.50 GHz $\times$ 4

The computer is connected to the robot controller by an Ethernet cable.

Chapter 8.

Experiment Setup

This chapter will present the experiments that has been conducted for this thesis. The results from each experiment can be seen in Chapter 9. First the setup of the robot cell will be explained followed by an explanation of some simple welding tests using the teach pendant. Finally experiments for testing the real-time performance of `moto` will be presented.

8.1. Robot Cell Setup for Welding

The robot cell at Manulab had to be set up for welding, and this section will explain some points that are worth noting for future work on the welding cell. The section will also explain some of the required setup for connecting the computer to the robot controller.

The correct welding wire has to be chosen and installed in the welding apparatus. The required diameter of the welding wire can be seen on the colour of the feeding wheels inside the wire feeder. A table shows which diameter corresponds to which colour. The wire is fastened in the first feeder and the n fed trough to the welding gun using control buttons on the power source or remote control.

A second thing worth noting is the gas supply used for welding. This welding cell is set up for MIG (metal inert gas) welding, which is a welding method that requires a shielding gas. A gas tank is coupled to the power source with a hose as seen in Figure 7.1. The amount of gas flow required varies on the welding operation that is to be performed and should be set by experienced welders. After connecting the gas tank and opening the valve, gas flow can be tested by using one of the buttons on the power source or the welding gun itself.

Setting up the computer for running `moto` in real-time requires two different IP addresses to be set. These are static IP addresses as explained in Section 4.1.4, and are used by the ROS-I code running on the robot controller. The two addresses both uses netmask 2255.255.255.0 and are as follows:

- **192.168.255.*X***, where ***X*** is an arbitrary number picked by the user. All numbers except 200 can be used, as this is occupied by the robot controller. The address above is used for TCP communication.
- **192.168.255.3** This address is used for the UDP connection between the computer and the robot controller when running the system in real-time.

The computer can also be st up to receive feedback from the ROS-I code running on the robot controller.This is done by Telnet as explained in section 4.1.1. After turning on the robot controller and connecting it to the computer by an Ethernet cable, type the following command in a terminal:

```
Telnet 192.168.255.200
```

The requested username and password for logging on to the robot controller is `MOTOMANrobot`. Messages printed by the ROS-I code will now be displayed in the terminal.

8.2. Welding with Teach Pendant

To verify that the robot cell is set up correctly and explore the possibilities of pendant programming, some basic welding operations were performed. By using the teach pendant the robot together with the positioning table was programmed to weld a straight line, an angle and a circle. Straight lines were welded with three different welding methods: MIG, MIG pulse and CMT. The welding job for a straight line can be seen in Appendix A.3, while the job for the circular weld can be seen in Appendix A.4. Welding parameters were set directly on the welding apparatus with the help of more experienced welders. How these parameters affect the welding process is out of the scope of this thesis.

8.3. Moto Testing

The `moto` library presented in Section 5.4 was tested to determine real-time performance. Testing was done using the code presented in Chapter 6. This section will present the setup for each performed test. All tests were performed on the robot manipulator as well as the positioning table. Three different real-time "system modes" are defined and tested:

- Regular mode, where the real-time server is run without any modification to the behaviour of the system.

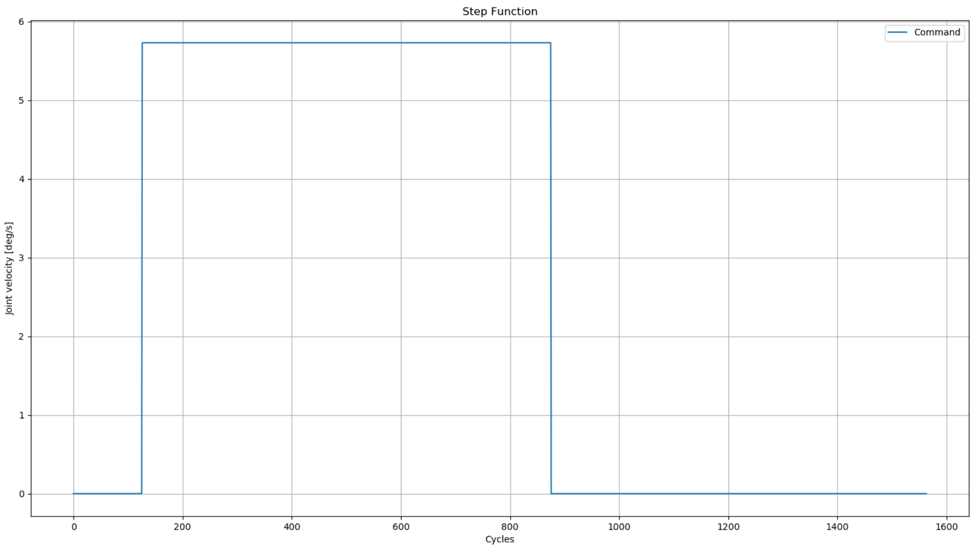


Figure 8.1.: The step function used to test the Moto code.

- PID mode, where the PID controller explained in Section 6.4 is implemented and tuned.
- HTRAJ mode, where HTRAJ is enabled as explained in Section 7.3.

8.3.1. Frequency Verification

As mentioned in 5.4, the real time system is set to run at 250 Hz. Parts of the code in [5], as well as latency calculations, is based on this fact and it is therefore important to verify the frequency. The test was performed using the `check_frequency()` function found in the `MotoTesterRt` class. Frequency was tested five times and the measured frequency was noted for each test. In this experiment only the regular system mode was considered.

8.3.2. Step Response

The step response of each joint was tested using the step function shown in Figure 8.1. Using the step function as the reference joint velocity yields a step response from each joint which was recorded in a CSV file. From the code presented in Section 6.6 it is then possible to calculate the step response and latency of each joint. Again the test was performed five times for each joint and the average value of all tests were used as the result.

8.3.3. Following a Reference Signal

While the step response shows the systems ability to adapt to a sudden change in the reference signal, it is also useful to look at how the system responds to a constantly changing smooth signal. The reference signal in this test was generated using (8.1), where $\dot{\theta}$ is the joint velocity and t is the time in seconds.

$$\dot{\theta} = 0.3 \sin 3t \tag{8.1}$$

The systems ability to follow the reference signal is then evaluated using latency, which is calculated by the function `calculate_latency()` explained in Section 6.6. In contrast to the step response experiment where each joint was tested individually, for this test the reference signal was sent to all joints at the same time. This was done based on the observations that controlling all joints simultaneously does not make a difference in latency compared to controlling one joint at the time.

Tests were also conducted to observe the systems response when altering the reference signal in 8.1. The frequency was increased to 15 and reduced to 1 and the amplitude was set to 0.1 and 0.6. These four tests were only conducted for the S joint on the robot manipulator and the X joint on the positioning table.

Just as with the previously explained experiments, each of the tests explained above was performed five times each. The presented result is then the average of each of the five tests.

Chapter 9.

Results

This chapter will present the results obtained from the experiments described in Chapter 8. The results will be presented in the same order as the experiments were described. Real-time performance testing will be presented in two sections, one for the robot manipulator and one for the positioning table. Lastly a suggested architecture for a constraint-based welding system will be presented.

9.1. Welding with Teach Pendant

Welding seams from the different welding processes can be seen in Figure 9.1. Starting from the left the figure shows straight welds, welding an angle and circular welding using the positioning table. For the straight welds three different welding methods were used. Starting from the left these are: MIG, MIG pulse and CMT.

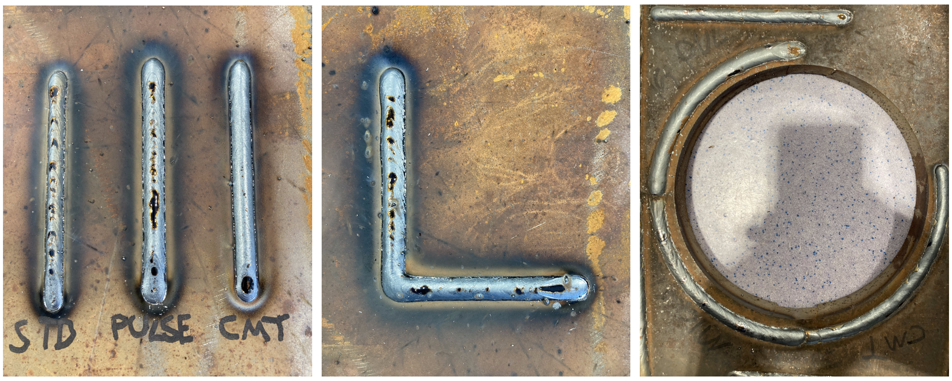


Figure 9.1.: Welding seams from the three different welding operations. Left: straight welds with three different welding methods: MIG, MIG pulse and CMT. Middle: welding an angle. Right: welding a circle using the positioning table.

9.2. Moto Testing, Manipulator

This section will present the results obtained from testing the `moto` library on the YASKAWA Motoman GP25-12 manipulator. Results will be presented in categories divided by the type of test that was performed. Where it is reasonable, results from the three different system modes will be presented collectively.

9.2.1. Frequency

Results form checking the control frequency while controlling the robot manipulator can be seen in Table 9.1.

Test nr.	**Measured Frequency**
1	$249.8966 \approx 250$
2	$248.5196 \approx 249$
3	$249.4807 \approx 249$
4	$249.4148 \approx 249$
5	$249.5827 \approx 250$

Table 9.1.: Measured control frequencies of the real-time system controlling the robot manipulator.

9.2.2. Step Response

The results from testing the step response of the three different system modes can be seen in Table 9.2 were response times and latencies are presented for each joint. Appendix D.1 shows the step response of joint S for each system mode plotted against each other.

Joint	**Response Time** [# of cycles]			**Latency** [# of cycles]		
	Regular	PID	HTRAJ	Regular	PID	HTRAJ
S	$9.6 \approx 10$	$9.0 \approx 9$	$9.0 \approx 9$	$41.6 \approx 42$	$23.0 \approx 23$	$25.8 \approx 26$
L	$10.0 \approx 10$	$9.0 \approx 9$	$9.0 \approx 9$	$50.6 \approx 51$	$21.0 \approx 21$	$26.4 \approx 27$
U	$11.0 \approx 11$	$10.0 \approx 10$	$9.0 \approx 9$	$48.6 \approx 49$	$21.0 \approx 21$	$27.0 \approx 27$
R	$14.6 \approx 15$	$12.8 \approx 13$	$11.0 \approx 11$	$43.4 \approx 44$	$21.8 \approx 22$	$23.8 \approx 24$
B	$13.0 \approx 13$	$12.0 \approx 12$	$11.0 \approx 11$	$38.2 \approx 39$	$22.0 \approx 22$	$25.8 \approx 26$
T	$14.6 \approx 15$	$14.2 \approx 15$	$12.2 \approx 13$	$30.2 \approx 31$	$27.4 \approx 28$	$23.4 \approx 24$

Table 9.2.: Average response time and latency for the step response of each joint in the three different system modes while controlling the robot manipulator.

9.2.3. Following a Reference Signal

The resulting average latencies from following the reference signal can be seen in Table 9.3. Examples of joint S following the reference signal in all three system modes can be seen in Appendix D.2. Only joint S was tested in the HTRAJ system mode, the reason for this will be explained in Section 10.2.3.

Joint	**Latency** [# of cycles]		
	Regular	PID	HTRAJ
S	$23.4545 \approx 24$	$10.1636 \approx 11$	$15.0422 \approx 16$
L	$23.4181 \approx 24$	$11.6909 \approx 12$	N/A
U	$23.3818 \approx 24$	$12.4000 \approx 13$	N/A
R	$22.0364 \approx 23$	$11.7273 \approx 12$	N/A
B	$23.6364 \approx 24$	$12.5273 \approx 13$	N/A
T	$22.6000 \approx 23$	$11.2909 \approx 12$	N/A

Table 9.3.: Latencies for the three system modes while setting each joint in the robot manipulator to follow the reference joint velocity (8.1).

Tests with modified frequency and amplitude in the reference signal were only conducted on joint S with the regular and PID system modes. The resulting latencies can be seen in Table 9.4. Appendix D.3 shows command and feedback joint velocity for both system modes when following the reference joint velocity with a frequency of 15.

Modification	**Latency** [# of cycles]	
	Regular	PID
Amp. = 0.1	$22.2015 \approx 23$	$9.1636 \approx 10$
Amp. = 0.6	$23.6622 \approx 24$	$10.3273 \approx 11$
Freq. = 1	$22.1467 \approx 23$	$8.8667 \approx 9$
Freq. = 15	$23.4153 \approx 24$	$10.5269 \approx 11$

Table 9.4.: Average latencies on joint S when following the reference signal (8.1) with modifications to amplitude and frequency.

9.3. Moto Testing, Positioning Table

This section will present the results obtained from testing the `moto` library on the YASKAWA MT1-500 S2HD positioning table. Results will be presented in the same manner as for the robot manipulator.

9.3.1. Frequency

The control frequency was tested for the server running the positioning table, and the results can be seen in Table 9.5.

Test nr.	**Measured frequency**
1	$249.9661 \approx 250$
2	$249.6340 \approx 250$
3	$249.9954 \approx 250$
4	$249.7687 \approx 250$
5	$249.8112 \approx 250$

Table 9.5.: Measured control frequencies of the real-time system controlling the positioning table.

9.3.2. Step Response

As with the manipulator, all three system modes were tested. The resulting response times and latencies can be seen in Table 9.6. A plot comparing the three system responses for joint X can be seen in Appendix E.1

Joint	**Response Time** [# of cycles]			**Latency** [# of cycles]		
	Regular	PID	HTRAJ	Regular	PID	HTRAJ
X	$12.0 \approx 12$	$9.0 \approx 9$	$11.2 \approx 12$	$46.6 \approx 47$	$21.0 \approx 21$	$47.0 \approx 47$
Y	$11.0 \approx 11$	$10.4 \approx 11$	$12.0 \approx 12$	$44.6 \approx 45$	$22.0 \approx 22$	$45.0 \approx 45$

Table 9.6.: Average response time and latency for the step response of each joint in the three different system modes.

9.3.3. Following a Reference Signal

The resulting average latencies from this experiment can be seen in Table 9.7. Joint X was also tested using the modified reference joint velocity and the results from these tests can be seen in Table 9.8. Example plots for joint X in each system mode can be seen in Appendix E.2, while plots for joint X in each system mode following the reference signal with a frequency of 15 can be seen in Appendix E.3.

Joint	**Latency** [# of cycles]		
	Regular	PID	HTRAJ
X	$23.3636 \approx 24$	$12.4545 \approx 13$	$23.3455 \approx 24$
Y	$23.1818 \approx 24$	$12.5818 \approx 13$	$23.2000 \approx 24$

Table 9.7.: Average latency while following the reference signal (8.1) for both joints on the positioning table running in the three different system modes.

Modification	**Latency** [# of cycles]		
	Regular	PID	HTRAJ
Amp. $= 0.1$	$22.6530 \approx 23$	$11.7818 \approx 12$	$22.7273 \approx 23$
Amp. $= 0.6$	$23.5091 \approx 24$	$12.6545 \approx 13$	$22.5273 \approx 23$
Freq. $= 1.0$	$22.6833 \approx 23$	$11.5333 \approx 12$	$22.6000 \approx 23$
Freq. $= 15.0$	$23.3841 \approx 24$	$12.9130 \approx 13$	$23.3714 \approx 24$

Table 9.8.: Average latencies on joint X when following modified versions of the reference signal (8.1).

9.4. A System for Constraint-Based Robotic Welding

This section will suggest a complete system for using constraint-based robot programming in welding applications. The architecture is based on the theory presented in this thesis and can be seen in Figure 9.2.

This explanation will be based on a simple welding operation similar to the ones seen in Section 9.1. In this case a simple line is welded to join together two metal plates. This simple process is used to simplify the explanation of the system. The process would start by scanning the groove between the plates with a laser scanner like the one seen in Section 7.5. From the scan a point cloud will be obtained, and it is possible to extract a best fit line. Some representation of this line is then sent to a script where it is translated into constraints in Lua that is usable by eTC as explained in Section 5.5.

After generating the constraints they are sent to an eTaSL/eTC ROS2 control controller. This is an implementation of eTC in ROS2 control developed by Lars Tingelstad which can be found at [52]. From this controller joint velocities are generated in the eTC solver which then can be sent to the robot controller using the `moto` library. At the end of each control cycle the robot controller gives feedback on the joint velocities and angles to the computer that are to be used in the next cycle. The position of the end-effector in each cycle can then be obtained from forward kinematics using the matrices presented in 7.1 and the PoE method as explained in Section 2.7. To find the pose of the positioning table the PoE

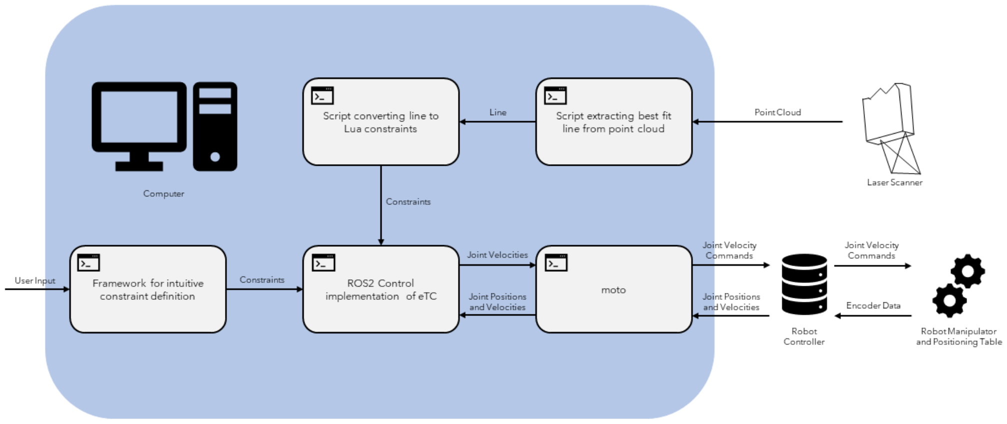

Computer
Script converting line to Lua constraints
Line
Script extracting best fit line from point cloud
Point Cloud
Laser Scanner
Constraints
User Input
Framework for intuitive constraint definition
Constraints
ROS2 Control implementation of eTC
Joint Velocities
Joint Positions and Velocities
moto
Joint Velocity Commands
Joint Positions and Velocities
Robot Controller
Joint Velocity Commands
Encoder Data
Robot Manipulator and Positioning Table

Figure 9.2.: Suggested architecture for a complete constraint-based system for robotic welding.

method can be used with the matrices presented in 7.2 instead.

Chapter 10.

Discussion

In this chapter the results presented in Chapter 9 will be discussed. First some lessons learned from the welding tests will be discussed before a discussion on the real-time performance of `moto`. Following this discussion a complete architecture for implementing constraint-based robot programming in robotic welding will be suggested. The suggested framework will be based on the previously presented theory. Finally some of the advantages of such an architecture will be discussed.

10.1. Welding with Teach Pendant

This section will discuss some of the lessons learned from doing welding experiments with the teach pendant, and the requirements this imposes on the architecture suggested later in the discussion.

One of the most obvious lessons learned is the simplicity of programming robotic welding operations with the pendant. The standard commands fount in the robot controller allows for a wide variety of welding operations, with a relatively low programming time. The simplicity of the teach pendant makes other software for programming robots almost obsolete for simple welding operations. The goal of a constraint-based architecture should therefore be more complex welding operations where teach pendant programming becomes more time consuming.

While moving the robot manually it was quickly observed that the kinematics in the robot quickly leads to certain joints reaching their angle limit. This should be taken into account when using constraints to move the robot, as to avoid the robot driving itself into a position which it cant get out of. Constraining the joint angles to be less than the limits of the robot might help the robot avoid this type of situation.

As there is practically no difference in programming a welding job and a regular

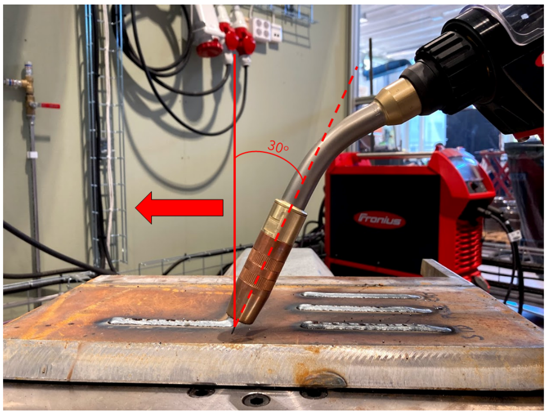


Figure 10.1.: The recommended direction of welding and angle of the welding gun for MIG welding processes.

job moving the end-effector, there is no speed limit on the robot while welding. Moving the welding gun too quickly while welding will result in a bad weld, and should be avoided. A speed limit on the end-effector while the arc is active should therefore be implemented.

Lastly the angle and direction of the welding gun should be considered both of which are illustrated in Figure 10.1. In a MIG welding process the welding gun should be "pushed" along the welding seam to cover the weld in shielding gas. The angle of the welding gun should be around 30 degrees. This direction and angle is also something to consider when defining constraints.

10.2. Moto Testing

This section will discuss the results obtained from testing the real-time performance of the `moto` library. Each test will be discussed in separate subsections. The main focus of discussion will be the results form the robot manipulator, but where it is necessary a note on the positioning table will be included as well.

10.2.1. Frequency

The expected frequency of the system is 250 Hz. As seen in Table 9.1 and 9.5 there is a small deviation in the expected and measured frequency both for the robot manipulator and the positioning table. Although there is a deviation it too small to be regarded as a problem for the system. For all practical proposes the system can be regarded as running at 250 Hz.

The cause of the slightly lower measured frequency is assumed to be the order in which commands are executed in `check_frequency()`. First the system is set in real-time mode before a timer is started. After two seconds the system is taken out of real-time mode before the timer is stopped. Since the timer is started before and stopped after the system is set to and taken out of real-time mode, the elapsed time will exceed the expected two seconds by a small amount. As the control frequency is calculated by the number of given commands divided by the elapsed time, this would result in a slightly lower control frequency than expected.

10.2.2. Step Response

From Table 9.2 it is seen that the response time is almost constant for each joint in all three system modes, and the general trend is that joints further from the base of the robot has a longer response time. The response time could be affected by both software and hardware delays, and it is difficult to pinpoint exact causes for high response times. In this case some delay probably occurs in communication between the computer and the robot controller, but some delay may also be caused by hardware. As the system goes from standstill to a set velocity when reacting to the step function, the robot manipulator has to turn off servo brakes to start the motion. This is believed to affect the system delay to some extent.

It is seen that latency reducing measures like the PID controller or having HTRAJ enabled does not affect the response time significantly. The most significant difference is found between the ordinary system and HTRAJ system on joint R where the response time is reduced by four cycles. Still the response times are somewhat constant for the three system modes and can therefore be regarded as a sort of minimum latency. This should be taken into account when looking at latency in the next subsections.

While the three system modes response times remain relatively unchanged, the difference in latency is more significant. Running the regular system mode, all joints show a latency between 31 and 51 control cycles. Running at 250 Hz this corresponds to 124 and 204 ms respectively, which is somewhat high for a system running in real-time. It is worth noting that the system usually will have a more gradually changing command joint velocity, which should result in lower latencies than what is measured in the step response. Still, both the PID and HTRAJ system mode show significant reduction in latency for most joints. It is also worth noting that in both of these two system modes the variation in latency reduced, so that all joints have a more equal latency than before. Having all joints respond with the same latency will be important for timing considerations when sending joint velocity commands in real-time.

Appendix D.1 shows that the reduction in latency is not achieved without drawbacks. The PID system mode achieved the lowest latency, but also showed the most oscillatory behaviour of the three system modes. The oscillations could be dampened more by increasing the derivative gain in the PID controller, but that would also require some sort of noise filtering of the velocity feedback. When the system was tested the increased derivative gain lead to amplified noise and a more oscillatory behaviour, rather than dampening oscillations.

In an attempt to allow for higher derivative gains a simple low pass filter was applied to the joint velocity contribution from the D controller. The filter can be described with the following equation:

$$\boldsymbol{\theta}_d = 0.5\boldsymbol{\theta}_{d,prev} + 0.5\boldsymbol{\theta}_{d,curr} \tag{10.1}$$

where $\boldsymbol{\theta}_d$ is the filtered joint velocity contribution from the D controller, $\boldsymbol{\theta}_{d,prev}$ is the previous contribution and $\boldsymbol{\theta}_{d,prev}$ is the current contribution before filtering. This filter had no noticeable effect which could indicate that a stronger filter is required. The problem using heavy filtering is that it can be time consuming, which is undesirable in a real-time system.

The HTRAJ system mode shows similar latency to the PID system mode, with only a couple of more cycles on most joints. As seen in Appendix D.1 it also has less oscillatory behaviour than the PID system mode. Still it is worth noting the violent oscillations that occur while the system is accelerating. The source of these oscillations are currently not known and it is unknown if they could cause problems when implementing `moto` in a complete architecture.

To set the response times into perspective, Table 10.1 shows response times from [24] where a similar experiment regarding response time was conducted. A real-time Python framework for robot control was used in this paper as well. Comping these data to the ones shown in Table 9.2 it is seen that the response times in `moto` are slightly higher than the two slowest systems in [24]. Comparing response time with the UR shows that the UR has almost a quarter of the response time of the other systems. The explanation for this lies in the robot controller that was used. When testing `moto` as well as the other two systems in [24] the robot controller from the manufacturer was used. These perform interpolation to generate trajectories and to various checks on the system which will stop the robot in case of an error. The UR was connected to a custom made velocity control not having the same amount of functions. This yields a fast system where errors have larger consequences.

To summarise the discussion, both the PID and HTRAJ system modes performed better than the ordinary system regarding latency, while response times remained mostly the same for all three system modes. While the cost of lower latency is

Robot	**Response Time** [ms]
NACHI SC15F	45
KUKA KR60L30 HA	42
UR-6-85-5-A	12

Table 10.1.: Response times for three different robots, data from [24].

a more oscillatory behaviour, this is most present in the PID system mode. The HTRAJ system mode show some large oscillations while accelerating but despite this seems to perform the best of the three system modes. Judging only by the step response it shows a good mix of low latencies and not too bad oscillatory behaviour.

Positioning Table
The positioning table shows much of the same behaviour as the robot manipulator in this experiment with regards to response time and latency for the step response, but one difference stands out. Looking at Appendix E.1 it is clear that the regular and HTRAJ system modes have the same step response. This implies that setting the system in HTRAJ mode only affects the robot manipulator, and does not affect the behaviour of the positioning table. Table 9.6 also confirms this as the regular and HTRAJ system mode have the same latencies.

10.2.3. Following a Reference Signal

Looking at Table 9.3 it is clear that the system starting from standstill does affect the latency. Running the regular system mode yields a latency of 23 or 24 control cycles on all joints when following a reference signal. This is an improvement to the latency shown by the step response, but it still correlates to almost 100 ms. For reference similar experiments to the ones done in this thesis was conducted in [24] as well. These latency results can be seen in Table 10.2.

Robot	**Latency** [ms]
NACHI SC15F	120
KUKA KR60L30 HA	115
UR-6-85-5-A	9

Table 10.2.: Latency for three different robots, data from [24].

Comparing the results in Table 9.3 to the values seen in Table 10.2 it is clear that the regular system mode compares roughly to the two slowest robots from [24]. It is still considerably slower than the UR, which is expected due to the UR's custom robot controller. The latency for the regular system mode is still higher than what is desirable for a system running in real-time. Having high latencies could especially be a problem for collision detection, where the system would react to a crash 100 ms after the crash has actually happened. For a constraint based system this could be especially undesirable as one of the goals of many constraint based system is to be able to collaborate with humans.

For this thesis where a welding operation is to be executed, the high latency might not be as big of an issue as for other robot applications. Welding processes are naturally slow moving due to the nature of welding where the metal needs to be heated properly. The system will in this case move slow and evenly so that no fast changes in joint velocities are required, and the latency could possibly be tolerated. In the case of welding corners like seen in Figure 9.1 latency would result in a lack of precision, as the welding gun would overshoot when changing the welding direction. If this is a problem or not depends on the required precision of the individual welding job that is to be executed. Still it could be desirable to be able to reduce the latency of the system, which is attempted with the PID and HTRAJ system modes.

As with the step responses it is seen that latency in the system is significantly reduced in the PID system mode. When tuning the controller it was also observed that the latency could be reduced further for the cost of more oscillations by further increasing the proportional gain. This would eventually make the system unstable, and the current tuning is regarded as a good mix between low latency and low oscillations.

When considering the step responses it is again seen that the PID system mode is the most oscillating of the three. For a PID controller the oscillations occurs with abrupt changes in the reference signal, like what is seen in a step function. Looking at Appendix D.2 it is clear that oscillations are no longer a problem when the reference signal is continuous and starts at zero joint velocity. With regards to this observation a system using the PID controller could have a filter for smoothing out step functions if oscillations are to be avoided. This would in turn increase latency the system, so a choice between oscillations and latency would have to be taken. Similarly to previous choices the right choice depends mostly on the task that is to be performed. Appendix D.2 also reflects the reduced latency by the shorter distance between the feedback and command graph compared to what is seen from the regular system mode.

Again looking at Table 9.3 it is seen that the HTRAJ system mode has a latency that is slightly higher than the PID system mode, but still significantly lower than the regular regular system mode. The reason that the HTRAJ system mode was not tested any further can be seen at the bottom of Appendix D.2. Although the feedback joint velocity follows the command joint velocity nicely, it shows tendencies to spike at random intervals. These are observed as sudden twitches in the robot while operating, and in fear of damaging the robot no further tests where performed with HTRAJ enabled.

Testing with the modified reference signal was done to check if the latency of the system would be affected by how fast the reference velocity changed. From Table 9.4 it is clear that the latency remains the same regardless of the frequency or amplitude of the reference velocity. Even though the latency remains unchanged another important factor which is not reflected by the latency can be seen in Appendix D.3. As the frequency is increased the feedback joint velocity seems to be unable to keep up with the command joint velocity in the regular system mode. This results in the feedback joint velocity taking the shape of a sinus wave with a lower amplitude than desired. Implementing the PID controller seems to fix this problem, but the system then overshoots and achieves a larger amplitude than the command joint velocity. This behaviour is not reflected in the latency calculations as it is calculated based off zero points in the graph. Whether undershooting or overshooting is the best solution is entirely up to the task that is to be executed by the robot, and has to be evaluated for each task. As mentioned previously welding is a naturally slow process, so the reference signal would probably not change as fast as seen in Appendix D.3. It may therefore not be a problem at all that the system is not able to track the reference joint velocity precisely at this speed.

As a summary, the regular system mode shows a latency that is typically higher than what is desired in a real-time system. This might not be a problem due to the slow nature of welding, but latency reduction would still be desirable. Of the two latency reducing measures introduced the PID controller seems to be the best due to the random spikes in joint velocity observed in the HTRAJ system mode. The PID system mode shows reduced latencies and works without oscillations when the reference joint velocity acts as a continuous single starting from zero joint velocity. Looking at the results from this experiment the PID system modes seems to perform the best overall.

Positioning Table
As with the step response the positioning table responds similarly to the robot manipulator in all experiments, with the exception of the HTRAJ system mode. Again it is seen from Table 9.7 and 9.8 that the regular and HTRAJ system mode behaves in the same way confirming that the HTRAJ function does not apply to the positioning table.

10.3. Validity of the Real-Time Performance

In this section the validity of the results obtained from the real-time performance testing will be discussed. This discussion will primarily be concerned with the code presented in Chapter 6 used to conduct the real-time performance testing.

As mentioned in Section 10.2.1 the results from this experiment has probably been altered a bit due to the order the functions are called in the test code. This is not considered to be a big enough alteration that it has any significant effect on the results.

The biggest factor concerning the validity of the latencies when the system is set to follow the changing reference signal, is the method that has been used to calculate latency. It was seen during experiments that in all three system modes the system was able to follow the reference signal with a constant latency. Using zero-points to calculate latency was then regarded as a simple and effective method. One of the problems using this method is that it only detects the latency at the zero-point, which would be problematic in a system with varying latency. This was observed during an attempt to implement the I controller. In such cases latency should have been calculated in a selection of points spread evenly along the time axis, not only the zero-points.

Another problem with the used method is that it does not detect cases like the one seen in Appendix D.3 and E.3 where the system does not reach the peak joint velocity at all. In this situation a total error parameter could have been included that calculates the average error during the whole runtime. This way these kind of deviations would have been detectable without looking at the plots. Additionally, this would introduce a more qualitative way of comparing the performance of different system modes when this type of deviation is present.

10.4. Latency Reduction

As much of the testing done for the `moto` library revolves around the latency of the system, this section will present possible solutions for reducing the latency in the system even further. Discussion regarding the PID and HTRAJ system mode will first be presented before finishing the section with some discussion regarding a custom robot controller.

10.4.1. Improving the PID Controller

By further tuning the PID controller, it could be possible to reduce latency even further. The current problem for further tuning is the noisy feedback signal from the encoders on the robot. This was mentioned previously in the discussion but will be mentioned once again here. As seen in Figure 6.2, setting the derivative gain too high will amplify noise and destabilise the system instead of dampening oscillations. By filtering the noisy signal it could be possible to implement a higher derivative gain, making the system oscillate less. This would in turn lead to the possibility for higher proportional gains resulting in reduced latency. The challenge of implementing a filter is the run-time of each control cycle. As the system needs to output joint velocity commands at 250 Hz, heavy filtering might not be possible.

10.4.2. HTRAJ

HTRAJ could also be further investigated. As the source of the joint velocity spikes is currently unknown, this is a problem that might be fixable. Somehow removing the velocity spikes would give system with slightly higher latency than what is obtained with the PID controller, but oscillations would not be present. This would make the HTRAJ system mode perform the best overall of the three system modes on both the step function and reference signal experiments.

The disadvantage of HTRAJ is that it only applies to the robot manipulator. A solution including HTRAJ would then still have high latency while moving the positioning table. For complex welding operations where the positioning table and robot manipulator cooperates, the difference in latency could prove problematic. The robot would then always execute commands before the positioning table, which could disrupt the cooperative movements between the two.

10.4.3. Custom Robot Controller

The last option for latency reduction that will be discussed is the possibility for a custom robot controller. As seen in [24] running the UR with a custom robot controller yields a much lower latency than the other tested robots. This could also be investigated as a possible solution for reducing latency in the Motoman system.

As mentioned briefly before the problem with custom controllers is the lack of safety functions that are usually found in the manufacturers controllers. These apply speed limits to the joints avoiding them from running to fast, detects collisions, joint limits etc. If any alarm is triggered in the controller the system will stop to prevent damage to the equipment. With a custom controller the system would not stop if an unforeseen error occurs, which could lead to major damage to the equipment.

The standard robot controllers also performs interpolation between two given points to generate a smooth trajectory for the end-effector. A custom low latency robot controller would probably not have this feature, and commands would have to be sent in smaller increments that do not require interpolation between them. This might not be a problem for a system running at 250 Hz as new commands are sent every 4 ms.

One challenge with a custom robot controller could be the willingness of YASKAWA to implement it. It can be seen in [19] that any major measures for real-time latency reduction would disrupt the warranty of the robot. Implementing a custom robot controller for reducing latency is probably regarded as such a major measure.

10.5. Timeouts and Possible Solutions

It was not tested explicitly during experimentation, but observation shows that the real-time mode in `moto` is prone to timeouts. This occurs when the robot does not receive commands from the computer for a set amount of control cycles. Timeout issues has to be addressed before further work on the system can be done. This section will present possible solutions to this problem.

The first and possibly easiest solution can be found in the ROS-I code running on the robot controller. In this code the number of missed control cycles required to trigger a timeout can be changed. While running experiments for this thesis, a single missed control cycle would trigger a timeout. This makes the system a hard real-time system as explained in Section 4.2. For a system like the one seen in this thesis running at 250 Hz, a single missed cycle would not be of too much importance as new ones are sent every 4 ms. This behaviour resembles more a firm real-time system. In the light of this the number of missed control cycles required to trigger a timeout could probably be raised. If this solution fails to fix the timeout problem, some other solutions will now be suggested.

The experiments done in this thesis was run on a regular Ubuntu system. Though few other programs were run on the computer at the same time as `moto`, prioritisation on the computer side could be a potential source for timeouts. If this is the case a solution would be to install a real-time kernel like explained in Section 4.3. This would give the real-time system highest priority on the computer which would make for a more stable execution time of each control cycle. This could possibly prevent it from exceeding the required 4 ms.

Another approach to the problem would be to time individual parts of the real-time cycle. If the execution time of each control cycle is up to 4 ms the system could sometimes exceed this time limit and not be able to send a command in time. By timing each individual piece of the control cycle it would be possible to identify any processes with high execution times and try to optimise them. A possibility in this process would be that none of the processes can be optimised for better execution time, this leaves two options. Either porting the code to C/C++ or reducing the control frequency.

By reducing the control frequency, the system would be able to use more time per control cycle for calculations. As an example the UR robot used in [24] was controlled using a control frequency of 125 Hz. This is half of the one used in this thesis and would allow an execution time of 8 ms for each control cycle.

Another solution would be porting the code to C/C++. Generally, code written in C/C++ runs faster than the same code written n for Python. Porting could then solve the issue with timeouts but would introduce the problem of not being as intuitive as Python. The advantage of having `moto` written in Python is the readability of the code and ease of further development. This is one of the advantages of Python and should not be discarded too quickly.

10.6. Complete System for Constraint-Based Robotic Welding

This section will discuss challenges and consideration regarding the system suggested in Section 9.4. Following this the advantages and disadvantages of such a system compared to traditional robotic welding systems will be presented.

10.6.1. Challenges and Considerations

In welding operations a problem that can occur is thermal expansion in the workpiece caused by the heat from the welding process. In robotic welding this can prove to be a challenge if the welding path is pre-programmed and no compensation for the expansion is implemented. Compensation can be done trough continuous sensing of the welding groove during the welding process either by trough-arc sensing or camera vision like explained in the project report attached to this thesis [6]. The problem with these methods are that trough-arc sensing does not work with all welding methods like e.g. CMT. Cameras used for continuous monitoring also requires heavy filtering due to the bright light from the welding arc, making the sensors expensive. In the system suggested in Section 9.4 the welding grove is only scanned before the welding process. A solution to avoid continuous monitoring is heavy clamping of the workpiece to prevent warping of the metal. This is the solution that should be utilised in the system suggested above to keep costs down. With particularly heavy or thick components warping should not be a problem.

In addition to the constraints generated by the laser scan of the welding groove, the user should also be able to define their own constraints on the system. This could for example include constraints for avoiding singularities (Section 2.9) or constraining the robots movement to a confined space like the robot cell while avoiding other equipment like the positioning table. Like mentioned in Section 10.1 constraints should also be applied regarding the direction and angle of the welding gun. Since defining constraints directly in Lua can be nonintuitive, a system like what is seen in [45] should be implemented where CAD models can be used to define constraints. Making constraint definition intuitive, the system can be operated by a wider range of personnel making it more applicable to the industry.

10.6.2. Advantages and Disadvantages

Advantages and disadvantages of constraint-based systems for robotic welding was also a subject of discussion in the attached project report [6], and similar arguments will be made here.

Given intuitive constraint definitions as well as constraints obtained from scans of the welding groove, the system will help define complex welding tasks more efficiently. This will make robotic welding a more viable option for a larger set of welding tasks. As task definition will be more efficient it also enables robotic welding to be used in productions with smaller batch sizes and more customised products.

As the solver for a constraint-based system optimises movement for all joints of the robot, this system will allow for easier implementation of redundant robots as explained in Section 2.2. Higher degree of redundancy in welding robots will in turn make the robots able to perform welding tasks that has previously been deemed to complex for robots to handle. Taking this argument even further the system could be used to control highly-redundant robots. These can be regarded as snake-like robots with DOF » 6. Highly redundant welding robots will be able to reach tight spaces while avoiding obstacles and singularities.

The most clear disadvantage of a constraint-based system for robotic welding is the potential complexity of the system. Even though an intuitive GUI for constraint definitions could be developed, it will have a hard time being more intuitive than many of the manufacturers teach pendants. This will make the system inapplicable to simple welding operations, limiting the systems range of tasks.

A system like this could also require more extensive training of personnel, compared to traditional robot programming methods. This would make it more expensive to hire new operators of such a system, and some manufacturers might not see this as a profitable investment.

Chapter 11.

Conclusion

Real-time performance testing of the three system modes were conducted with a focus on response time and latency. Based on the test results, a discussion regarding real-time performance and how it would affect a welding process followed. The discussion focused on the following points:

- The response time of all three system modes are practically equal, and can be regarded as a minimum latency.
- Latency is different in all three system modes for the robot manipulator. The regular system mode has the highest latency and the PID system mode has the lowest latency.
- There is a correspondence between low latency and oscillations in the system when reacting to a step function.
- In a continuously changing reference signal where joint velocities start at zero, oscillations does not appear.
- The HTRAJ system mode shows random joint velocity spikes in the robot manipulator and also does not apply to the positioning table.
- Though not tested explicitly, timeouts in the system have proved to be a problem. This is regarded as a solvable problem, and solutions are suggested.
- In the case of robotic welding, high latencies might not be a problem due to the slow nature of the welding process. Some latency reducing measures are still suggested.

In conclusion the `moto` library does show similar real-time performance as similar Python frameworks, but the latency in the system is still high. Implementing a PID-controller lowers latency with the cost of introducing oscillations to the system. The high latency might not be a problem for the naturally slow welding process.

Based on the presented theory, the architecture for a complete constraint-based system for robotic welding is suggested. This system uses the `moto` library for robot communications, while a ROS2 control implementation of eTC/eTaSL computes joint velocities based on constraints, joint velocity feedback and joint position feedback. A laser scanner is also included along with a system for user defined constraints. The whole system is yet to be implemented physically, and needs to be tested. Challenges and considerations of the system is discussed along with its advantages and disadvantages.

11.1. Further Work

The first thing that should be addressed of the work done in this thesis is the timeouts while running in real-time. Having constant timeouts is not sustainable when tying to implement `moto` into the suggested system. Changing the ROS-I code in the robot controller to a firm real-time system should first be attempted, before attempting the other suggested solutions to the problem.

After timeouts are fixed, the next step would be to begin implementing the suggested system. The system should first be implemented without the sensor by manually defining constraints for basic welding operations. This should be done to individually test the ROS2 controller with the `moto` library, and confirm if high latencies will be problematic or not.

Given that basic welding operations can be completed with the partially implemented system, the rest of the system should be implemented. Again simple welding tasks should be executed to confirm the systems reliability before moving on to more complex tasks.

References

[1] Erwin Aertbeliën. *eTaSL Documentation*. 2020. URL: https://etasl.pages.gitlab.kuleuven.be/contents.html (visited on 05/10/2021).

[2] Erwin Aertbeliën and Joris De Schutter. “eTaSL / eTC : A constraint-based Task Specification Language and Robot Controller using Expression Graphs”. In: ().

[3] Shaheen Ahmad and Shengwu Luo. “Coordinated Motion Control of Multiple Robotic Devices for Welding and Redundancy Coordination through Constrained Optimization in Cartesian Space”. In: *IEEE Transactions on Robotics and Automation* 5.4 (1989), pp. 409–417. ISSN: 1042296X. DOI: 10.1109/70.88055.

[4] Jens G. Balchen, Trond Andresen, and Bjarne A. Foss. *Reguleringsteknikk*. 6th ed. Institutt for Teknisk Kybernetikk, NTNU, Trondheim, 2016.

[5] Vebjørn B. Bjørhovde. *moto_tester*. 2021. URL: https://github.com/Vebjorbb/moto_tester (visited on 03/09/2021).

[6] Vebjørn B. Bjørhovde. *The Potential of Constraint-Based Robot Programming for Welding Robots in the Norwegian Industry*. Unpublished work, attached to the thesis. 2020.

[7] E. Sahin Conkur and Rob Buckingham. “Clarifying the definition of redundancy as used in robotics”. In: *Robotica* 15.5 (1997), pp. 583–586. ISSN: 02635747. DOI: 10.1017/S0263574797000672.

[8] D Dallefrate, D Colombo, and L Molinari Tosatti. “Development of robot controllers based on PC hardware and open source software”. In: January 2005 (2015), pp. 2–7.

[9] Joris De Schutter, Tinne De Laet, Johan Rutgeerts, Wilm Decré, Ruben Smits, Erwin Aertbeliën, Kasper Claes, and Herman Bruyninckx. “Constraint-based Task Specification and Estimation for Sensor-Based Robot Systems in the Presence of Geometric Uncertainty”. In: *The International Journal of Robotics Research* 26.5 (2007), pp. 433–455. DOI: 10.1177/0278364907078091. URL: http://ijr.sagepub.com.

[10] Wilm Decŕe, Ruben Smits, Herman Bruyninckx, and Joris De Schutter. “Extending iTaSC to support inequality constraints and non-instantaneous task specification”. In: *Proceedings - IEEE International Conference on Robotics and Automation* (2009), pp. 964–971. ISSN: 10504729. DOI: 10.1109/ROBOT.2009.5152477.

[11] Andrea Del Prete, Francesco Nori, Giorgio Metta, and Lorenzo Natale. “Prioritized motion-force control of constrained fully-actuated robots: "task Space Inverse Dynamics"”. In: *Robotics and Autonomous Systems* 63.P1 (2015), pp. 150–157. ISSN: 09218890. DOI: 10.1016/j.robot.2014.08.016. arXiv: 1410.3863. URL: http://dx.doi.org/10.1016/j.robot.2014.08.016.

[12] The Linux Foundation. *Intro to Real-Time Linux for Embedded Developers*. 2013. URL: https://linuxfoundation.org/blog/intro-to-real-time-linux-for-embedded-developers/ (visited on 06/04/2021).

[13] Tully Foote Geoffrey Biggs. *Managed Nodes*. 2021. URL: https://design.ros2.org/articles/node_lifecycle.html (visited on 04/21/2021).

[14] Fronius International GmbH. *TPS/i*. 2021. URL: https://www.fronius.com/en/welding-technology/products/manual-welding/migmag/tpsi/tpsi/tps-400i (visited on 05/22/2021).

[15] Yaskawa Europe GmbH. *GP25-12*. 2021. URL: https://www.yaskawa.eu.com/products/robots/handling-mounting/productdetail/product/gp25-12_698 (visited on 05/21/2021).

[16] Yaskawa Europe GmbH. *MT1*. 2021. URL: https://www.yaskawa.eu.com/products/robots/peripherals/productdetail/product/mt1_800 (visited on 05/21/2021).

[17] Yaskawa Europe GmbH. *Yaskawa MOTOMAN Robot Controllers: YRC1000*. 2021. URL: https://www.yaskawa.eu.com/products/robots/controller/productdetail/product/yrc1000_583 (visited on 06/04/2021).

[18] Google. *Static vs. dynamic IP addresses*. 2021. URL: https://support.google.com/fiber/answer/3547208?hl=en (visited on 06/07/2021).

[19] *Implications of high Command - Feedback latency*. 2020. URL: https://github.com/ros-industrial/motoman/issues/219 (visited on 04/15/2021).

[20] O. KHATIB, L. SENTIS, J. PARK, and J. WARREN. “Whole-Body Dynamic Behavior and Control of Human-Like Robots”. In: *International Journal of Humanoid Robotics* 01.01 (2004), pp. 29–43. ISSN: 0219-8436. DOI: 10.1142/s0219843604000058.

[21] Phillip A. Laplante and Seppo J. Ovaska. *Real-Time Systems Design and Analysis*. 2011. ISBN: 3175723993. DOI: 10.1002/9781118136607.

[22] Claus Lenz, Markus Rickert, Giorgio Panin, and Alois Knoll. “Constraint task-based control in industrial settings”. In: *2009 IEEE/RSJ International Conference on Intelligent Robots and Systems, IROS 2009* (2009), pp. 3058–3063. DOI: 10.1109/IROS.2009.5354631.

[23] Morten Lind, Johannes Schrimpf, and Morten Lind. “6.7 Real-Time Robot Trajectory Generation with Python”. In: *Sensor-based Real-time Control of Industrial Robots* (2013), p. 129.

[24] Morten Lind, Johannes Schrimpf, and Thomas Ulleberg. “Open Real-Time Robot Controller Framework”. In: *2010 3rd CIRP Conference on Assembly Technology and Systems - Responsive, customer demand driven, adaptive assembly* May (2010), pp. 13–18.

[25] Kevin M Lynch and Frank C Park. *Modern Robotics*. Cambridge University Press, 2017.

[26] ro2_control maintainers. *ros2_control documentation*. 2021. URL: https://ros-controls.github.io/control.ros.org/ (visited on 04/21/2021).

[27] Nicolas Mansard, Olivier Stasse, Paul Evrard, and Abderrahmane Kheddar. “A versatile generalized inverted kinematics implementation for collaborative working humanoid robots: The stack of tasks”. In: *2009 International Conference on Advanced Robotics, ICAR 2009* (2009).

[28] Misc. *control_msgs*. 2021. URL: https://github.com/ros-controls/control_msgs (visited on 04/21/2021).

[29] Misc. *control_toolbox*. 2020. URL: https://github.com/ros-controls/control_toolbox (visited on 04/21/2021).

[30] Misc. *motoman*. 2021. URL: https://github.com/ros-industrial/motoman (visited on 04/21/2021).

[31] Misc. *pr2_mechanism*. 2020. URL: https://github.com/pr2/pr2_mechanism (visited on 04/19/2021).

[32] Misc. *realtime_tools*. 2021. URL: https://github.com/ros-controls/realtime_tools (visited on 04/21/2021).

[33] Misc. *ROS-indusrial*. 2021. URL: https://github.com/ros-industrial (visited on 04/23/2021).

[34] Misc. *ros2_control*. 2021. URL: https://github.com/ros-controls/ros2_control (visited on 04/21/2021).

[35] Misc. *ros2_controllers*. 2021. URL: https://github.com/ros-controls/ros2_controllers (visited on 04/21/2021).

[36] Marius Nilsen. “ROS2 Intergration of ABB IRB 14000 YuMi”. MA thesis. Norwegian University of Science and Technology, 2019.

[37] Matthias Radestock and Susan Eisenbach. “Coordination in evolving systems”. In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 1161 (1996), pp. 162–176. ISSN: 16113349. DOI: 10.1007/3-540-61842-2_34.

[38] Open Robotics. *ROS Hisory*. 2021. URL: https://www.ros.org/history/ (visited on 02/16/2021).

[39] Adolfo Rodríguez, Luis Basañez, and Enric Celaya. “A relational positioning methodology for robot task specification and execution”. In: *IEEE Transactions on Robotics* 24.3 (2008), pp. 600–611. ISSN: 15523098. DOI: 10.1109/TRO.2008.924263.

[40] ROS-Industrial. *Description*. 2021. URL: https://rosindustrial.org/about/description (visited on 04/23/2021).

[41] ROS-Industrial. *Our Brief History*. 2021. URL: https://rosindustrial.org/briefhistory (visited on 04/23/2021).

[42] ROS-Industrial. *The Challenge: Transitioning Robotics R&D to the Factory Floor*. 2021. URL: https://rosindustrial.org/the-challenge (visited on 04/23/2021).

[43] ros-infrastructure. *ROS 2 Documentation*. 2021. URL: https://index.ros.org/doc/ros2/ (visited on 02/16/2021).

[44] Johannes Schrimpf, Morten Lind, and Geir Mathisen. “Time-analysis of a real-time sensor-servoing system using line-of-sight path tracking”. In: *IEEE International Conference on Intelligent Robots and Systems* (2011), pp. 2861–2866. DOI: 10.1109/IROS.2011.6048078.

[45] Nikhil Somani, Andre Gaschler, Markus Rickert, Alexander Perzylo, and Alois Knoll. “Constraint-based task programming with CAD semantics: From intuitive specification to real-time control”. In: *IEEE International Conference on Intelligent Robots and Systems* 2015-Decem (2015), pp. 2854–2859. ISSN: 21530866. DOI: 10.1109/IROS.2015.7353770.

[46] Nikhil Somani, Markus Rickert, Andre Gaschler, Caixia Cai, Alexander Perzylo, and Alois Knoll. “Task level robot programming using prioritized non-linear inequality constraints”. In: *IEEE International Conference on Intelligent Robots and Systems* 2016-Novem (2016), pp. 430–437. ISSN: 21530866. DOI: 10.1109/IROS.2016.7759090.

[47] Nikhil Somani, Markus Rickert, and Alois Knoll. “An Exact Solver for Geometric Constraints with Inequalities”. In: *IEEE Robotics and Automation Letters* 2.2 (2017), pp. 1148–1155. ISSN: 23773766. DOI: 10.1109/LRA.2017.2655113.

[48] IEEE Spectrum. *PR2*. 2021. URL: https://robots.ieee.org/robots/pr2/?gallery=photo1 (visited on 04/19/2021).

[49] William Stallings. *Data and Computer Communications*. 10th ed. Pearson Education Limited, Edinburgh Gate, Harlow, 2014.

[50] William Stallings. *Operating Systems: Internals and Design Principles*. 9th ed. Pearson Education Limited, Edinburgh Gate, Harlow, 2017.

[51] Telnet.org. *Telnet*. 2020. URL: telnet.org (visited on 05/31/2021).

[52] Lars Tingelstad. *etasl_ros2_control*. 2019. URL: https://github.com/tingelst/etasl_ros2_control (visited on 06/02/2021).

[53] Lars Tingelstad. *moto*. 2021. URL: https://github.com/tingelst/moto (visited on 04/21/2021).

[54] Lars Tingelstad. *motoman*. 2020. URL: https://github.com/tingelst/motoman (visited on 04/21/2021).

[55] Adolfo Rodríguez Tsouroukdissian. *ROS control, an overview*. 2014. URL: https://vimeo.com/107507546 (visited on 04/19/2021).

[56] Adolfo Rodríguez Tsouroukdissian. *ROS control, an overview*. 2014. URL: https://roscon.ros.org/2014/wp-content/uploads/2014/07/ros_control_an_overview.pdf (visited on 04/19/2021).

[57] G. C. Vosniakos and A. Chronopoulos. "Industrial robot path planning in a constraint-based computer-aided design and kinematic analysis environment". In: *Proceedings of the Institution of Mechanical Engineers, Part B: Journal of Engineering Manufacture* 223.5 (2009), pp. 523–533. ISSN: 09544054. DOI: 10.1243/09544054JEM1234.

[58] John G Ziegler, Nathaniel B Nichols, et al. "Optimum settings for automatic controllers". In: *trans. ASME* 64.11 (1942).

Appendix A.

A Selection of Robot Jobs

A.1. INIT_ROS

```
NOP
DOUT OT#(890) OFF
DOUT OT#(889) OFF
TIMER T=0.05
DOUT OT#(889) ON
WAIT OT#(890)=ON
DOUT OT#(890) OFF
END
```

Figure A.1.: The INIT_ROS job running while the teach pendant is set in remote mode.

A.2. INIT_ROS with HTRAJ

```
NOP
GETS LPX000 $PX000
MOVJ LP000 VJ=5.00
HTRAJON
MOVJ LP000 VJ=5.00
DOUT OT#(890) OFF
DOUT OT#(889) OFF
TIMER T=0.10
DOUT OT#(889) ON
WAIT OT#(890)=ON
DOUT OT#(890) OFF
HTRAJOF
END
```

Figure A.2.: The altered INIT_ROS job used to enable the HTRAJ function.

A.3. SIMPLE_ARC

```
NOP
MOVJ VJ=5.00
MOVJ = 0.78
ARCON
MOVL V=40
ARCOF
MOVJ VJ=5.00
```

Figure A.3.: The welding job used to weld the straight lines shown in Figure 9.1.

A.4. CIRCULAR_ARC

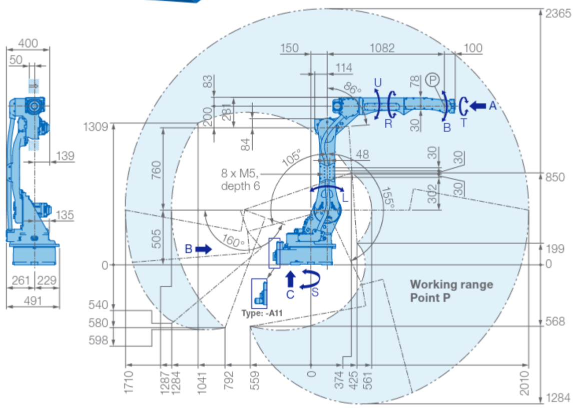
```
MOVJ VJ=10.00
    +MOVJ VJ=10
MOVJ VJ=0.78
    +MOVJ VJ=0.78
ARCON
MOVJ VJ=0-78
    +MOVJ VJ=2.00
ARCOF
MOVJ VJ=0.78
    +MOVJ VJ=0.78
```

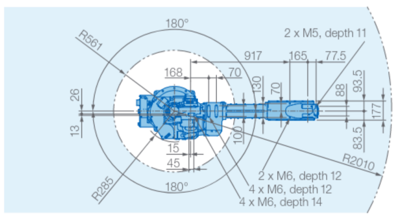
Figure A.4.: The welding job used to weld the circular seam shown in Figure 9.1.

Appendix B.

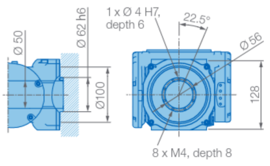
YASKAWA Motoman GP25-12

MOTOMAN GP25-12

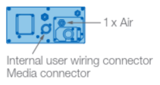




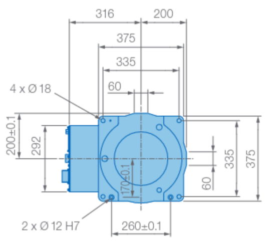
View A



View B



View C



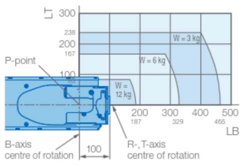
Prevent interaction of the robot with:
- Corrosive gases, liquids or explosive gases
- Exposure to water, oil or dust
- Excessive electrical noise (plasma)

Mounting options: Floor, ceiling, wall, tilt*

Protection class: Main axes (S, L, U) IP54 (option 65), wrist IP67

* tilt with condition of angle – see table below

Allowable wrist load



Robot installation angle θ [deg.]	S-axis operating range [deg.]
$0 \le \theta \le 30$	±180 degrees or less (no limit)
$30 < \theta \le 35$	±60 degrees or less
$35 < \theta$	±30 degrees or less

Specifications GP25-12

Axes	Maximum motion range [°]	Maximum speed [°/sec.]	Allowable moment [Nm]	Allowable moment of inertia [kg · m²]
S	±180	210	–	–
L	+155/–105	210	–	–
U	+160/–86	220	–	–
R	±200	435	22	0.65
B	±150	435	22	0.65
T	±455	700	9.8	0.17

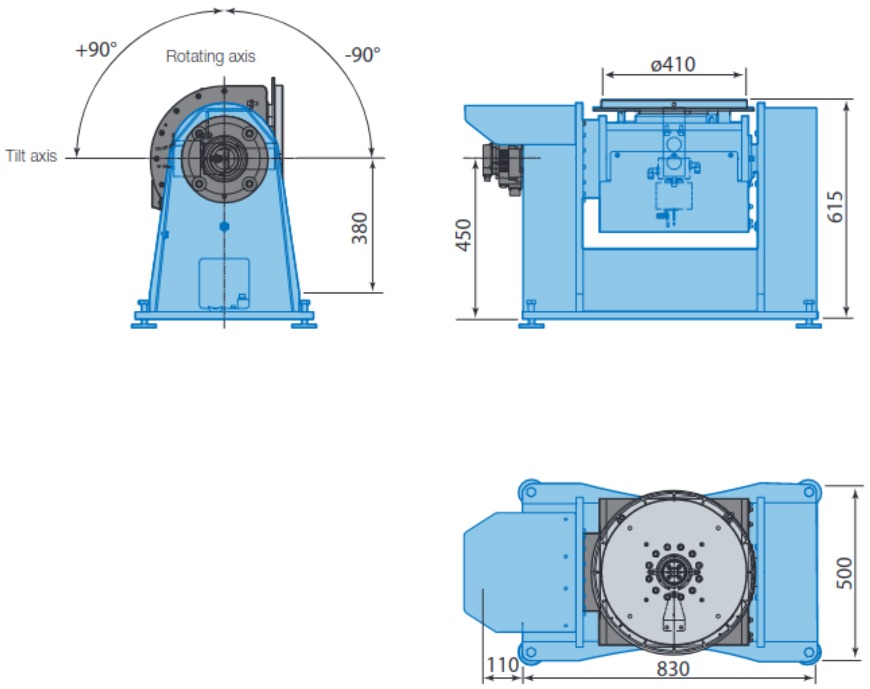
Controlled axes	6
Max. payload (on U-axis) [kg]	12 (9)
Repeatability [mm]	±0.03*
Max. working range R [mm]	2010
Temperature [°C]	0 to +45
Humidity [%]	20 – 80
Weight [kg]	260
Power supply, average [KVA]	2.0**

* Conforms to ISO 9283 ** Varies in accordance with applications and motion patterns **Note:** SI units are used for specifi cations. All dimensions in mm

Figure B.1.: Technical specifications for YASKAWA Motoman GP25-12. Figure from [15].

Appendix C.

YASKAWA MT1-500 S2HD



Technical data		MT1-500S2HD
Max. payload per station		**500 kg**
Welding capacity	100% duty cycle 60% duty cycle	350 A 460 A
Tilt axis torque	dynamic static	1612 Nm 1330 M.
Tilt axis rated speed		0-10.6 rpm
Rotating axis torque	dynamic static	1108 Nm 896 Nm
Rotating axis rated speed		0-11.3 rpm
Max. inertia		100 kgm^2
Rated offset from centre of gravity (COG)		145 mm
Weight		321 kg

Figure C.1.: Technical specifications for the YASKAWA MT1-500 S2HD positioning table. Figure from [16].

Appendix D.

Robot Manipulator Plots

D.1. Step Response

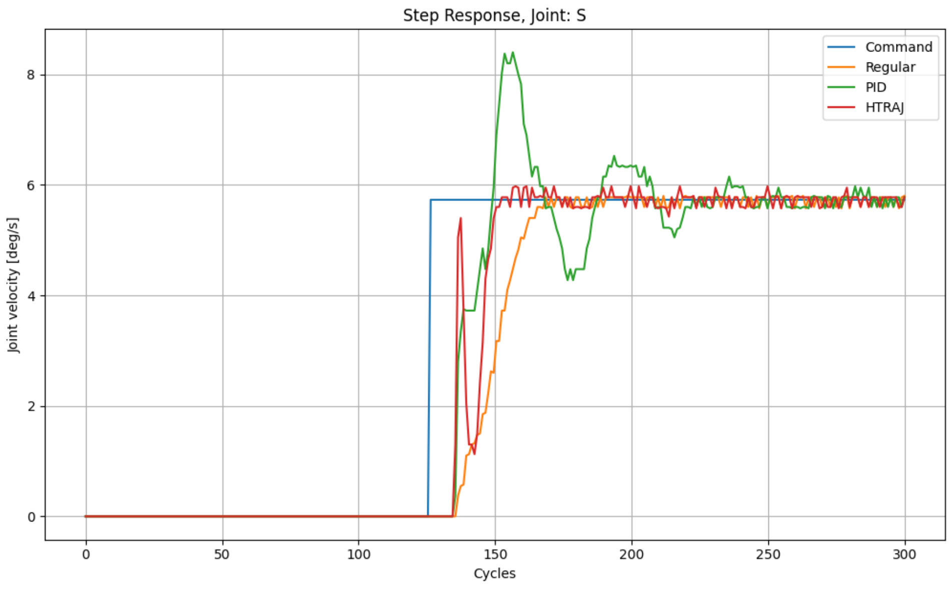


Figure D.1.: Step response for joint S in the three different systems modes plotted against each other.

D.2. Following a Reference Signal

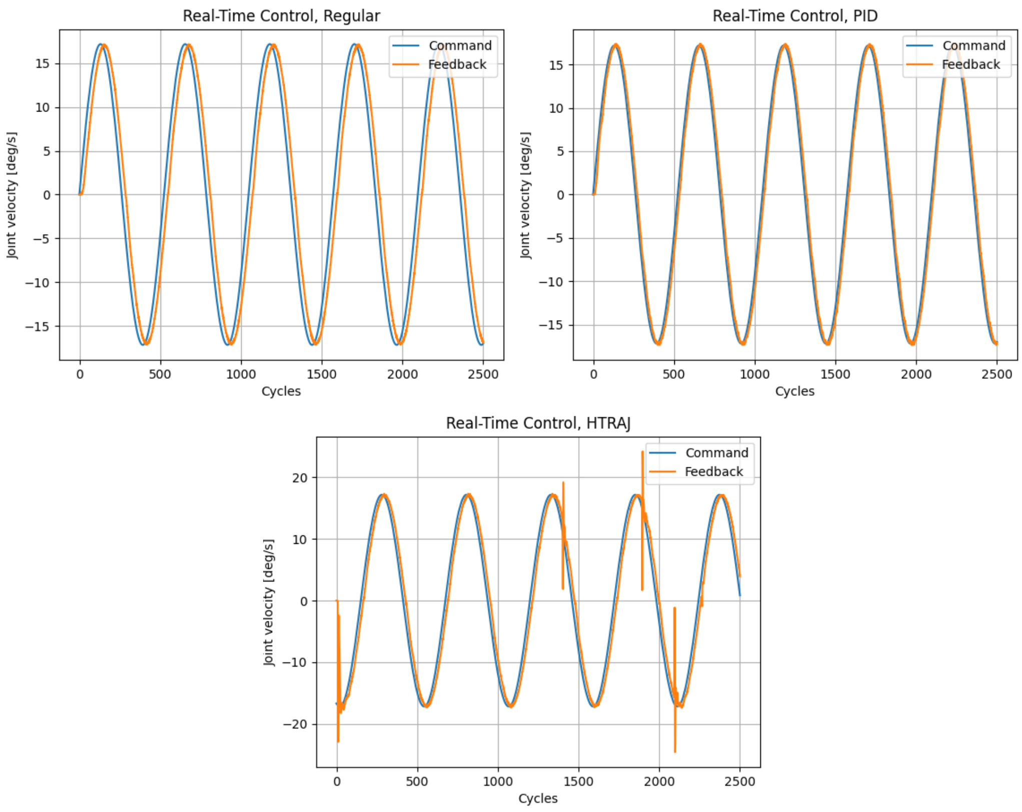


Figure D.2.: Joint S following the reference joint velocity (8.1) in all three system modes.

D.3. Following a Modified Reference Signal

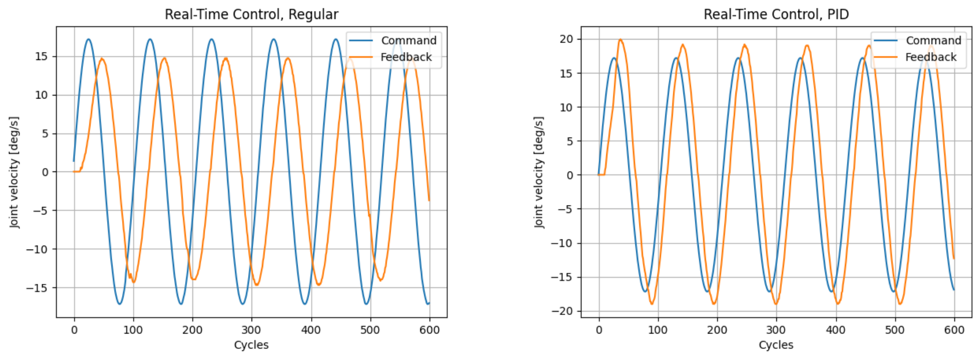


Figure D.3.: Command and feedback velocity for joint S with the regular and PID system modes using the reference velocity in (8.1) with a frequency of 15.

Appendix E.

Positioning Table Plots

E.1. Step Response

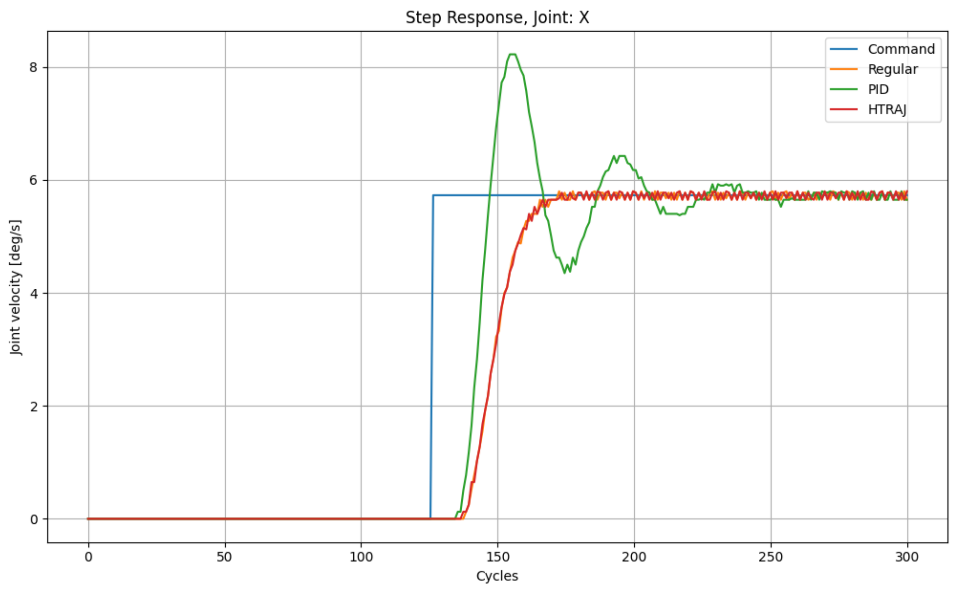


Figure E.1.: Comparison of the step response for joint X in all three system modes.

E.2. Following a Reference Signal

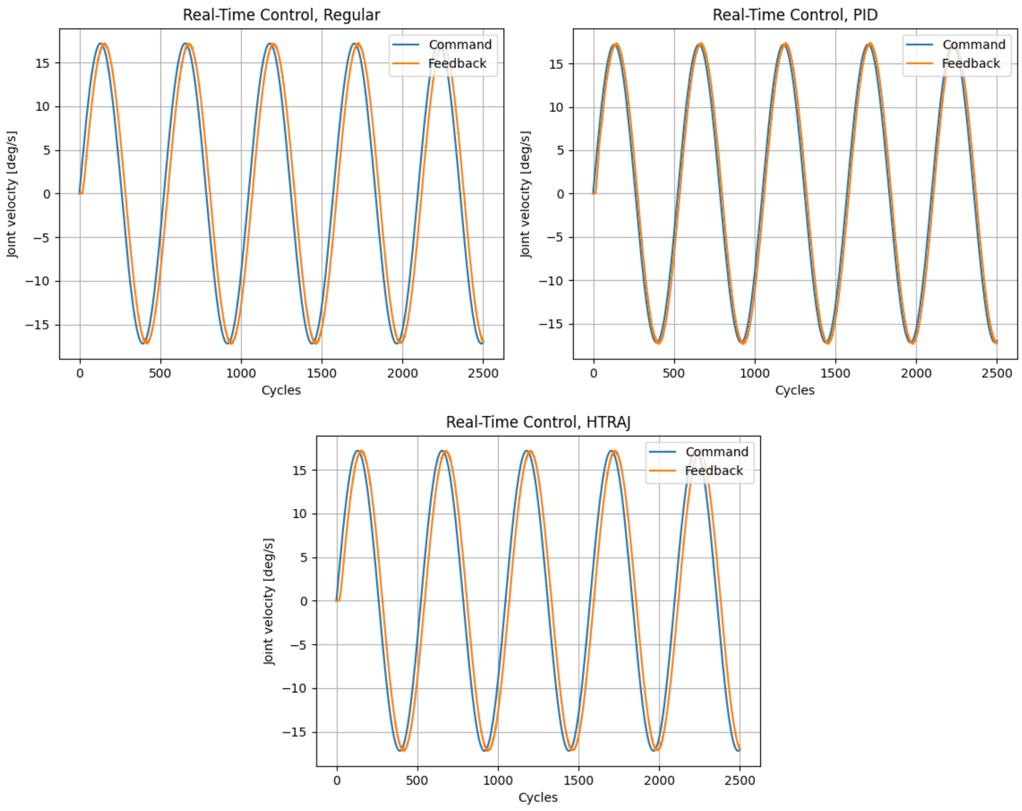


Figure E.2.: Joint X following the reference joint velocity (8.1) in all three system modes.

E.3. Following a Modified Reference Signal

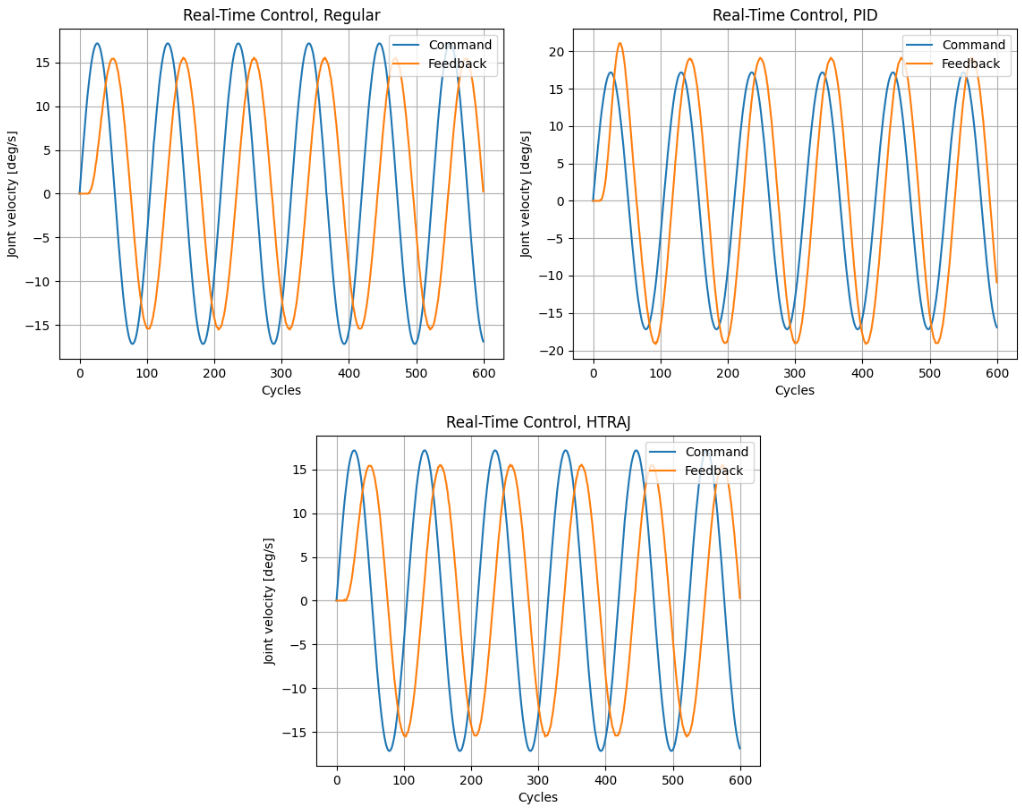


Figure E.3.: All three system modes running on the positioning table following a reference signal in (8.1) with a modified frequency of 15.