UiA University of Agder
Norway

Modelling, Identification and Control of a 5-DOF Shotcrete Robot

Development of a Framework for Automatic Application of Shotcrete for AMV 4200H

A thesis submitted in partial fulfilment of the requirements for the degree of
Master of Science in Mechatronics

Andreas K. Auen
Tomas S. Lyngroth

Supervisor
Geir E. Hovland

This Master's Thesis is carried out as a part of the education at the University of Agder and is therefore approved as a part of this education. However, this does not imply that the University answers for the methods that are used or the conclusions that are drawn.

University of Agder, 2019
Faculty of Engineering and Science
Department of Engineering Sciences

Acknowledgements

This thesis could not have been accomplished without the support and knowledge of those around us. Therefore, we would like to show gratitude to our contributors.

First, we would like to thank AMV for giving us the opportunity of researching their shotcrete machine for our master's thesis. They have been very helpful and responsive to all our enquiries. In addition, they allowed us to use their machine for physical measurements even if it meant halting production in the meanwhile.

Thanks to the University of Agder for providing office facilities at which the project could be carried out. Moreover, thanks to the staff, who are always kind and helpful.

We want to show special appreciation for supervisor, Professor Geir E. Hovland, who has guided us for the duration of the project. His expertise in robotics, modelling, identification and control has proven invaluable.

Abstract

Today, process automation is the primary area of development in the shotcrete industry. Automatic shotcrete operations can yield an increase in operational efficiency and personnel safety as well as reductions in cost and environmental impact. This thesis develops a framework for automatic application of shotcrete using the AMV 4200H and provides an automatic spraying mode using interoceptive sensing. The shotcrete vehicle is equipped with a five degrees-of-freedom manipulator and is currently operated manually. Our contributions include solving the forward kinematics through the Denavit-Hartenberg convention, and the inverse kinematics using kinematic decoupling and iteration. Furthermore, a spraying trajectory is created based on a minimal number of reference points provided by the operator. Curve fitting and parameterisation techniques are employed for generating a spraying trajectory based on the desired settings. All interactions with the machine are conveniently managed from an intuitive human-machine interface. Individual control of each joint with feedback has been implemented on the electronic control unit from the actual machine. Through hardware-in-the-loop simulation, the concepts have been proven, and the functionality of the framework has been verified.

Keywords: Shotcrete Machine, 5-Degree-of-Freedom Manipulator, Kinematics, Trajectory Generation, Human-Machine Interface, Control, Hardware-In-the-Loop Simulation.

Contents

Nomenclature

Abbreviations

AADT Average Annual Daily Traffic

AMV Andersen Mekaniske Verksted

ARMAX Autoregressive-moving-average model with exogenous inputs

CAN Controller Area Network

CGA Conformal Geometric Algebra

CiA CAN in Automation

COB-ID Communication Object Identifier

CPU Central Processing Unit

CR Centre of Rotation

DH Denavit–Hartenberg

DOF Degrees Of Freedom

ECU Electronic Controller Unit

GIL Global Interpreter Lock

HIL Hardware-In-the-Loop

HMI Human-Machine Interface

I/O Input/Output

LIDAR Light Detection And Ranging

LTI Linear Time-Invariant

NCP Nozzle Centre Point

NRMSE Normalised Root Mean Square Error

PDO Process Data Object

PID Proportional-Integral-Derivative

RT Real-Time

SIMO Single-Input and Multiple-Outputs

UDP User Datagram Protocol

Symbols

α_i	Angle about common normal, from z_{i-1} to z_i	rad
β	Roll angle	rad
δ	Logarithmic decrement	$-$
γ	Pitch angle	rad
$\dot{\mathbf{x}}$	Derivative of state vector	$-$
$\mathbf{A}$	System matrix	$-$
$\mathbf{B}$	Input matrix	$-$
$\mathbf{C}$	Output matrix	$-$
$\mathbf{D}$	Feedforward matrix	$-$
$\mathbf{u}$	Input vector	$-$
$\mathbf{x}$	State vector	$-$
$\mathbf{y}$	Output vector	$-$
ω	Nutation velocity	rad/s
ω_d	Damped frequency	rad/s
ω_n	Natural frequency	rad/s
$\psi_{4,stat}$	Static deflection	rad
τ_{stat}	Static torque	Nm
θ_i	Joint angle of joint i	rad
φ	Nozzle spread angle	rad
ζ	Damping ratio	$-$
A	Amplitude of peak	deg
a_i	Length of the common normal	m
b_{eq}	Equivalent viscous damping	Ns/m
d_i	Offset along previous z to the common normal	m
D_s	Distance from nozzle to surface	m
k_{eq}	Equivalent spring stiffness	Nm/rad
L_{arc}	Arc length of tunnel	m
P_d	Damped period	s
Q_m	Volumetric flow rate of hydraulic motor	m^3/s
r_s	Spread radius of nozzle jet on surface	m
T_w	Transformation matrix for wrist frame	$-$
v_m	Hydraulic motor displacement	m^3/rad
x_w	x-position of wrist frame	m
y_w	y-position of wrist frame	m
z_w	z-position of wrist frame	m

Chapter 1

Introduction

1.1 A Brief History of Shotcrete

Application of concrete by means of spraying was first introduced by American inventor and taxidermist Carl Ethan Akeley (1864–1926), (Teichert, 2002). In June 1907, Akeley presented the first modern concrete applicator called the "Plastergun". This device pumped dry plaster through a hose using compressed air to convey the material to the nozzle where it was mixed with water before being projected at high velocity onto a receiving surface. Later in 1911, a patent was issued for an "Apparatus for mixing and applying plastic or adhesive materials" named the "Cement gun", and in 1912 the sprayed material was patented under the name "Gunite". Today, gunite is no longer proprietary and is commonly called sprayed concrete or shotcrete.

Since Akeley‘s time, there have been tremendous developments in shotcrete technology. Noteworthy is the invention of wet-mix concrete in 1955, (Dhir, 1976). Wet-mixed concrete refers to concrete that has been premixed with water and admixtures before being pumped. Compared to dry-mix ("Gunite"), wet-mix shotcrete has several advantages in large scale operations; it can be prefabricated in large external batch plants before being transported on-site and allows larger diameter hoses with higher volumetric flow rates. It also ensures a more homogeneous mix and consistency, while producing less dust and rebound, contributing to increased operational safety and efficiency. Wet-mixed concrete was later adopted by the shotcrete industry in the mid ’90s and has since become the industry standard for large scale operations. Other significant developments in shotcrete technology include the rotary applicator gun, more effective pumps, higher quality materials and admixtures and the advancements of the modern computer. The modern computer made it possible to introduce robotic placement of shotcrete and employ advanced inspection methods for monitoring deposition and automated operations. The future of shotcrete technology is only limited by the rate of new innovations.

1.2 Motivation

In Norway, shotcrete is heavily used in tunnel construction both as temporary lining before placement of concrete elements and as permanent lining for rock support. Shotcrete is popular in tunnel construction for several reasons such as high compressive strength, low permeability, fast curing and no need for formwork, which greatly reduce cost. It is also the only practical construction material capable of handling the sheer scale and uneven surface topology of tunnel excavation.

The primary method of tunnelling in Norway is the so-called "drill and blast" method. This method is a sequential process which in short involves drilling holes for explosives, blasting 3-5 m sections, ventilating, mucking and scaling. After mucking and scaling of the newly blasted tunnel section, the site is still not secure for human personnel. This is because there is no guarantee that all loose debris has been dislodged and there is still a potential for cracking due to stresses in the rocks. Shotcrete is therefore applied to the tunnel walls in order to secure the area and to make the surface

impermeable. Shotcrete is typically applied using a layer thickness of $\approx 10\ cm$, depending on rock quality and specifications (AMV, 2018).

The main criteria for the shotcrete operation are as follows:

1. The minimum thickness must be fulfilled (usually $\approx 10\ cm$ depending on rock quality).
2. Minimise concrete usage to reduce cost and environmental impact.
    a. It is essential that shotcrete is applied perpendicular to the spraying surface at the correct distance in order to reduce rebound and overspray.
    b. It is also crucial that the thickness of the applied concrete layer is "thick enough" according to specifications, but not excessive.

Rebound of applied material is inevitable. The US Army Corps of Engineers (2005) estimated that under normal conditions, when spraying perpendicular to the surface, the rebound will be between 2-5 % for application to floors or slabs, up to 5-10% on sloping or vertical walls and as much as 10-15% for overhead work. In addition, if the spraying angle is not perpendicular to the surface and differs from 90°, the rebound may increase up to 50 % (Girmscheid and Moser, 2001). The spraying distance is also important. Optimal spraying distance is 1.2 to 1.5 metres, but in practice, this may be hard to achieve with human operators and between 1 to 2 meters is acceptable (Hofler and Schlumpf, 2004). If the spraying distance becomes shorter, the amount of rebound will increase, and if the spraying distance becomes too large, it may also increase rebound, affect layer compaction and degrade layer strength.

Quality of applied shotcrete also depends on factors such as the speed of application, types of aggregates, accelerators, admixtures, the machinery and more. However, the most detrimental factor is the skill-set of the nozzle operator and their ability to keep the correct spraying angle and distance. In practice, studies show that human operators tend to spray concrete until the finished layer has a smooth and even look, when in fact, they could have used less concrete. Using smoothness as a stopping condition for layer thickness also means that there is a risk of the concrete layer being too thin in certain places, making the strength of the layer insufficient.

For human operators and personnel, shotcrete has a severe drawback in terms of health, safety and environment. Sprayed concrete releases chemicals and airborne dust, which may prove harmful to the respiratory system and can lead to medical conditions such as chronic obstructive pulmonary disease. Furthermore, production of concrete has a large carbon footprint, and application of shotcrete may contribute to the release of chemical substances which are not beneficial for the environment. For these reasons, it is desirable to keep the operator at a safe distance and minimise excessive use of shotcrete.

1.3 Project Background

Andersen Mekaniske Verksted (AMV) AS is a Norwegian company founded in 1860. The company produces industrial machinery for mining and tunnelling operations, in addition to equipment for the oil and gas industry. As part of their research and technological development, the company wants to improve the shotcrete operation in the "Drill and Blast"-method of tunnelling. They aim to make the process safer for personnel and the environment, with better quality and cost-effectiveness. The platform for this research is the AMV 4200H, depicted in Figure 1.1. AMV wants to develop a framework for achieving automatic application of shotcrete. Automatic spraying gives more control over the operational parameters and may also contribute to reducing the required operator interaction.

Figure 1.1: 3D render of the AMV 4200H, courtesy of AMV.

The AMV 4200H was originally designed for manually controlled operations. The kinematic structure has 5 degrees of freedom (DOF) and is simple enough that manual control of individual joints is intuitive for the operator. The machine has a telescopic boom capable of extending up to 15 m, giving reach far into unsupported areas while keeping the operator at a safe distance. The main drawback is that the operator must have the nozzle within line of sight during the application process. Due to poor visibility caused by dust, spraying mist and distance to the nozzle, the operator may be forced to get close to potentially hazardous areas. Reducing the required input from the operator by automation is desirable for the health and safety of personnel as well as the possibility of increased operational efficiency and reduction in material cost.

On the AMV 4200H, ready mixed concrete is poured from a concrete transport vehicle into the receiving hopper at the front of the heavy-duty truck. The concrete mix is pumped through hoses to the nozzle at the end of the manipulator. When the wet-mix reaches the nozzle, hardener and pressurised air are added to accelerate the curing process and to increase the pressure and velocity of the shotcrete. Today, the manipulator is manually controlled by a nozzleman, either from a panel within the vehicle or over radio. Furthermore, the key specifications of the AMV 4200H are:

- Hybrid, both electric and diesel operation.
- Gross weight approximately 37 metric tonnes.
- Electric system; 3 1000VAC at 50Hz. Control system electronics: 24VDC.
- 2 pcs 90kW 1000VAC electric motors for operation of the concrete pump, boom and compressors.
- Danfoss ECU using CAN bus communication.

1.4 Project Scope and Objectives

The scope of this project is to produce a framework for automatic application of shotcrete to new tunnel sections using the AMV 4200H. Application of shotcrete will be performed with individual kinematic closed-loop control of each joint. External feedback from exteroceptive sensors, such as laser measurements are not considered in this thesis. The tunnel surface is parameterised by reference points provided by the nozzleman. The tunnel section is assumed to be relatively smooth with constant cross section. The spraying angle is set equal to the normal on the tunnel surface and uneven surface topologies are not explicitly considered. AMV has stated their expectations for the thesis, summarised as follows:

- Solve forward and inverse kinematics.
- Programming of a control system for automatic shotcrete application.
- Generation of a spraying plan on the tunnel surface, at a given velocity and surface distance.
- Development of a graphical user interface for automatic spraying.
- Development of a CANopen interface between a PC and the Danfoss ECU.
- Modelling of the physical system in Simulink for testing and control.

1.5 Limitations

This project is subject to some limitations. The main limiting factor is the scale of the project concerning time. Several simplifying assumptions have been made to ensure that a minimal working example of the proposed framework could be completed in time. This, for example, includes operation without exteroceptive sensing of the environment, not using laser scanning data and instead assuming that the spraying surface can be modelled as a surface using polynomials. Another limitation is related to project planning, as the project has several dependencies, accurate estimation of time usage can be difficult. Moreover, the authors started this thesis with little prior knowledge on many of the core concepts required for the execution, such as shotcrete technology, robotics, CAN bus communication and Python programming. Lastly, the physical machine is not at the UiA campus, and validation testing cannot be performed until the end of the project.

1.6 State-of-the-art

Using robotic manipulators for application of sprayed concrete is not a novel idea. The first robotic shotcrete manipulators appeared over 40 years ago (Kurth et al., 2010). Typically, any machine with the ability to manipulate a nozzle was retrofitted with shotcrete equipment. The resulting machines were rudimentary and only applied dry-mix concrete, and since these machines were not purpose-built for shotcrete operations, they often proved inefficient with little flexibility in terms of manipulability.

Over the next decades, along with an increasing demand for operational safety, accountability and efficiency, shotcrete manipulators became more specialised for the given application and became equipped with dedicated equipment on-board such as concrete pumps, compressors, tanks for accelerators, dosage units and more. Further improvements were achieved concerning materials, fibres and aggregates, as well as chemical products such as water reducing admixtures and pumping aids, plasticisers, set accelerators, hydration control and concrete improving and curing agents (Girmscheid and Moser, 2001).

By the 2000s, Girmscheid and Moser (2001) demonstrated the first fully automatic shotcrete robot. Their research was carried out for an existing 8-DOF manipulator called the MEYCO Robojet. The complex kinematic chain did not have a closed-form solution, and they relied on numerical methods for calculation of the inverse kinematics. Their work provided the Robojet with what they call "application intelligence", and refers to automatic correction and optimisation of operational parameters to achieve theoretical final lining thickness with high quality and minimal rebound. Application intelligence was obtained by combining empirical research on optimal shotcrete deposition with laser measurements and an application process control program to estimate optimum parameter settings. They provided three operational modes; manual, semi-automatic and fully automatic mode.

- **Manual mode**: The complex kinematic structure was simplified so that the Cartesian path of the nozzle could be controlled with a 6D-joystick. This mode is used when the presence and skill of the operator are required, for example on highly irregular surfaces or holes. The operator controls application parameters.

- **Semi-automatic**: The operator controls the translation and velocity of the nozzle along the tunnel surface. Other process variables such as wall distance and nozzle orientation are handled by the application process control and mechanical control system.
- **Fully automatic**: Takes complete control over the application process and determines necessary shotcrete capacity, air pressure, distance to tunnel surface, rotational and translational speed of nozzle. All based on the required layer thickness. This mode is suitable for sections with predictable surface geometry, such as smoothly blasted excavations and profiles drilled by tunnel boring machines.

For the semi-automatic and automatic modes, the operational area must be scanned with laser prior to the shotcrete application to calculate the operational parameters. In fully automatic mode, the tunnel profile scanned twice. First, it is divided into a grid structure, and the required deposition for each cell in the grid is calculated as the difference between measured and theoretical thickness. The process control system uses the required layer thickness to interpolate the nozzle velocity along each cell in the grid to provide the correct deposition. After shotcreting, the tunnel is scanned again to assess the layer deposition. Girmscheid and Moser conclude that no single operating mode is superior and that each operating mode serves a specific purpose depending on the boundary conditions.

Others have also performed similar research to that of Girmscheid and Moser but have, on the contrary not provided the same application intelligence. The research has in general been more focused on robotic modelling and automation of similar machines (Cheng et al., 1996), (Honegger et al., 2002), (Wang and Su, 2007), (Xuewen et al., 2010).

More recent work on fully automatic operations was performed by Nabulsi et al. (2010) on the Sika R-Putzmeister PM-407 shotcrete machine. This machine has a 5-DOF manipulator, and the inverse kinematics were solved using decoupling and iteration methods. They provide two modes, manual and semi-automatic, with the same principal functionality as Girmscheid and Moser. In their discussions, they discuss several drawbacks with the retrofitted machine such as complex kinematics, lack of hydraulic power to actuate all joints simultaneously and that the machine is directly operated on hydraulics without load-sensing.

The use of laser scanning is not new in the shotcrete industry. The effectiveness of laser scanning before and after shotcrete application is well proven. A trail project demonstrated a reduction in concrete expenditure by 20-30 % due to more accurate thickness control and operator training (Norwegian Tunnelling Society, 2017). Today, one of the most promising developments for the shotcrete industry comes from the field of 3D mapping and segmentation. New methods and techniques have made it feasible to achieve a fully autonomous shotcrete operation where all process parameters are monitored, optimised and corrected in real-time (RT) based on 3D data. On this basis, AMV's goal for the future is to take full advantage of the potential in modern 3D techniques. They aim to realise a new system with automated real-time concrete thickness measurement though 3D-vision. The system shall work in dusty low-light conditions with real-time transfer of 3D-data to the control system for automatic corrections of errors in shotcrete deposition and layer thickness.

Unlike Girmscheid and Moser (2001), the AMV 4200H has a similar kinematic structure to the machine used by Nabulsi et al. (2010) and is simple enough that manual joystick control is intuitive. Thus, the kinematic structure does not have to be simplified further. In comparison to Nabulsi et al. (2010), the hydraulic system of the AMV 4200H has more overhead and modern load-independent flow control valves, making the machine more suited for automatic control. However, as pointed out by Nabulsi et al. (2010), robotisation of a machine that is not purpose-built for automatic control usually present unexpected difficulties. As an example, most autonomous industrial robots are created with automatic control in mind, and considerable effort is given to provide a closed-form solution for the inverse kinematics. Most existing shotcrete machines are intended for manual

operation, and little thought has been given to the kinematic structure, often leading to unsolvable systems of equations.

In this thesis, "application intelligence", i.e. optimisation of shotcrete application parameters, have not been considered. However, a manual and fully automatic mode will be provided without the use of laser scanning, using closed-loop control with joint feedback. A polynomial representation of the tunnel surface is created from probed points provided by the operator. The wall distance will be determined from the position of the provided reference points, and the nozzle orientation is set equal to the normal on the surface. This approach assumes even cross sections and does not consider the optimal nozzle angle to the real and uneven tunnel topology. A trajectory will be generated on the polynomial tunnel surface, and the operator will have full control over the shape of the pattern and the nozzle velocity. Moreover, a graphical user interface is provided for simple planning and administration of the spraying procedure. In addition, the required software and interfacing with the on-board electronic control unit (ECU) is provided.

This thesis does not provide a solution for fully automatic shotcrete operations with real-time application process control but provides a prestudy and foundation for further work. The contributions include providing:

- A solution to the forward and inverse kinematics of the AMV 4200H.
- An algorithm for generating the spraying pattern and a method for mapping the spraying pattern onto the polynomial tunnel surface.
- An implementation of individual joint control with feedback.
- A method for automatic application of shotcrete without exteroceptive sensor feedback.
- A computer application with a graphical user interface, implementing the methods.
- A communication interface between the ECU and a PC.
- Verification of the developed model through hardware-in-the-loop testing.

Special effort is also given to prepare the concepts and ideas presented in this thesis for further development.

1.7 Document Structure

This thesis is structured into 8 chapters, a summary of each is given below:

- **Chapter 1** – *Introduction* – starts with setting the stage and giving a general introduction to the key concepts, methods and limitations of shotcrete technology. Moreover, the background for the project is given along with the project scope, objectives and limitations. Then, a state-of-the-art on shotcrete technology is presented.
- **Chapter 2** – *System Description* – describes the physical system and explains how the forward and inverse kinematics are solved. It also gives a discussion and analysis on some of the unique mechanical components such as the nozzle eccentric and the actuating geometry for boom elevation. In addition, the system Jacobian is derived along with an analysis of kinematic singularities.
- **Chapter 3** – *Trajectory Planning* – gives explanations for the whole process of generating the spraying trajectory and the nozzle orientation reference. The chapter starts with defining the coordinate systems used to parameterise the spraying surface and how the geometry of road tunnels are in general. Furthermore, it describes how reference points given by the operator are used for generating the tunnel surface, and how the planar trajectory is generated and mapped onto the surface.
- **Chapter 4** – *System Modelling* – starts with giving motivation for HIL-setup and dynamic modelling. Then a thorough explanation is given on how the HIL model works and why it can be simplified. Then, modelling of system dynamics are presented along with the different methods that were employed to fit a model to the measured validation data.
- **Chapter 5** – *Interface and Control* – presents the communication and setup between the physical system, ECU and user interface PC. It also describes the blocks, logic and communication created in the Danfoss Plus+1 software, and gives a brief introduction to CAN bus and CANopen. Furthermore, the proposed control structure and deadband compensator are presented. Lastly, the developed software and graphical user interface are illustrated and explained in detail.
- **Chapter 6** – *Results* – presents the results from research and development.
- **Chapter 7** – *Discussion and Further Work* – provides a discussion on the obtained results and recommendations for further work.
- **Chapter 8**– *Conclusion* – Lastly, a conclusion to the work and project is drawn.

1.8 Source Code Repository

The methods in this thesis are implemented in the form of a Python application. The source code is included in Appendix E.2 as well as a GitHub repository for easier reference and cloning.

`https://github.com/auen/auto_shotcrete`

Chapter 2

Kinematics

While applying shotcrete, the vehicle is stationary and partially sustained by support legs. The application is solely executed by the boom assembly depicted in Figure 2.1, accommodating five kinematic joints:

- Revolute joint for slewing, θ_1.
- Revolute joint for elevation, θ_2.
- Prismatic joint for the telescope, d_3.
- Revolute joint for nozzle roll, θ_4.
- Revolute joint for nozzle pitch, θ_5.

The slewing drive, which turns the boom about the base, is actuated by two hydraulically driven worm gears. The elevation is handled by two hydraulic cylinders, connected in parallel to form an inverted slider-crank mechanism with the revolute joint. The prismatic joint consists of a two-stage telescopic arm which is actuated by two hydraulic cylinders in series inside the boom. Unlike load-bearing cranes, the cylinders are not actuated in sequence. This means that, while the total extension is measured, the individual extension of the boom elements is unknown under partial extension. The two last joints form a wrist mechanism, controlling roll and pitch of the nozzle.

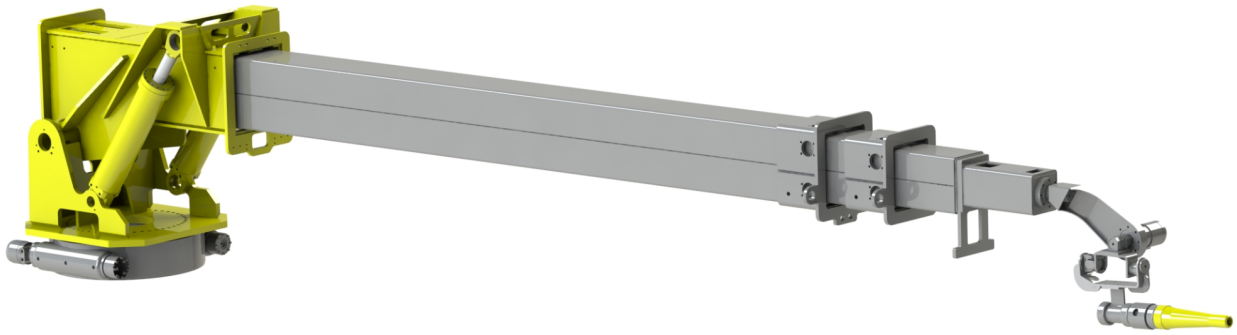


Figure 2.1: CAD model of boom assembly with reduced detailing.

The nozzle is connected to the machine via an eccentric mechanism which rotates to create a spraying cone. This mechanism is a closed kinematic loop itself and serves the purpose of increasing the spread of the nozzle jet. Note that the eccentric mechanism is not treated as a part of the kinematic analysis. Only the centre point of rotation is considered. The eccentric is discussed further in Section 2.1.3.

2.1 Forward Kinematics

Forward kinematics refers to the use of kinematic equations to calculate the pose of the end effector given a set of joint variables, represented as a block-diagram in Figure 2.2. In practical terms, it

describes how to apply data from sensors to determine the position and orientation of the nozzle.

Figure 2.2: Forward kinematics: From Joint Space to Cartesian coordinates.

The forward kinematics is analysed following the joint frame assignment technique of Denavit-Hartenberg, (Spong et al., 2005). Using the DH-convention, each joint frame can be described by a coordinate transformation from the previous joint frame $i-1$ to the current frame i. All motion is described relative to a non-moving frame, referred to as the base frame. Each joint frame i may be described by a certain set of successive translations and rotations from the previous joint frame $i-1$. This relation is given by the transformation

$$\mathbf{T}_{i}^{i-1} = \left[\begin{array}{ccc|c} & \mathbf{R} & & \mathbf{T} \\ \hline 0 & 0 & 0 & 1 \end{array}\right] = \mathbf{Rot}_{z_{i-1}}(\theta_i) \cdot \mathbf{Trans}_{z_{i-1}}(d_i) \cdot \mathbf{Trans}_{x_{i-1}}(a_i) \cdot \mathbf{Rot}_{x_{i-1}}(\alpha_i) \tag{2.1}$$

where

θ_i : Joint angle of Joint i $[rad]$

d_i : Offset along the previous z to the common normal $[m]$

a_i : Length of the common normal $[m]$

α_i : Angle about common normal from z_{i-1} to z_i axis $[rad]$

$\mathbf{R}$: Rotation matrix containing the column vectors $\mathbf{r}_x, \mathbf{r}_y, \mathbf{r}_z$ $[-]$

$\mathbf{T}$: Translational vector containing the coordinates (x, y, z) $[m]$

The matrices in Equation (2.1); $\mathbf{Rot}_z$, $\mathbf{Trans}_z$, $\mathbf{Trans}_x$, $\mathbf{Rot}_x$ are given in Appendix B.1.

Equation (2.1) describes how to get from one joint to the next by multiplication of two screw displacements. First a rotation θ_i is performed around the previous z-axis, z_{i-1}, followed by a translation d_i along the direction of z_{i-1} to the common normal of axes of z_{i-1} and z_i. Next, the coordinate system $i-1$ is translated along the common normal a distance a_i. Lastly, the coordinate system is rotated around the previous x-axis, x_{i-1} such that the new z-axis, z_i aligns with the axis of revolution or translation for Joint i. Note that the y-axis is obtained by completing the right-handed coordinate frame.

For a serial manipulator, the transformation from the base frame to the end effector is simply the ordered matrix product of the individual joint transformations:

$$\mathbf{T}_n^1 = \mathbf{T}_2^1 \cdot \mathbf{T}_3^2 \ ... \ \cdot \mathbf{T}_n^{n-1} \tag{2.2}$$

Applying the above-stated method to the AMV 4200H shown in Figure 2.1, one obtains the kinematic map in Figure 2.3. From the figure, it can be seen that the construction has two joint offsets. One at the base between Joint 1 and Joint 2, the other at the wrist frame between Joint 4 and the end effector. In general, joint offsets can increase the difficulty of finding closed-form solutions for the inverse kinematics. Furthermore, Joint 4 is moved such that it is perpendicular to Joint 5 by adding the distance from Joint 4 to joint 5 to the length d_3. The angle of Joint 2, θ_2 is offset by $90°$ such that the telescopic boom is horizontal in the home position. The coordinate system of Joint 6 does not represent an actual joint. The sixth coordinate system is included to orient the z-axis of the end effector such that it points in the spraying direction of the nozzle. Moreover, the only change from Joint 6 to the end effector is a translation of length d_6.

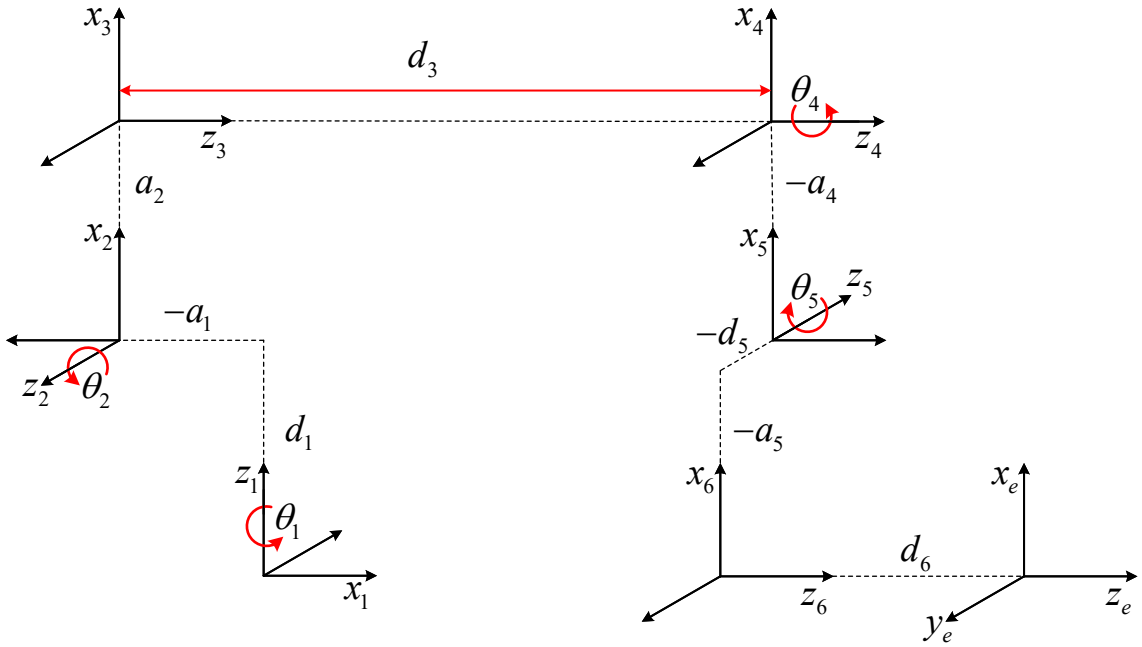


Figure 2.3: Kinematic model of the boom assembly on AMV 4200H. Note that joint motions are indicated in red.

Using Equation (2.2), the complete transformation from the base frame to the end effector is found from ordered matrix multiplication of all intermediate joint transformations:

$$\mathbf{T}_e^1 = \mathbf{T}_2^1 \cdot \mathbf{T}_3^2 \cdot \mathbf{T}_4^3 \cdot \mathbf{T}_5^4 \cdot \mathbf{T}_6^5 \cdot \mathbf{T}_e^6 \tag{2.3}$$

Interpreting the rotation matrix, $\mathbf{R}$ itself is not particularly intuitive. Furthermore, since the machine is 5-DOF, it is under-actuated in terms of a 6-DOF space. Two rotations can therefore describe the functional rotation. The nozzle is directed along the z_e-axis. Consequently, the yaw (z-rotation) becomes insignificant since it does not affect the task, and we are left with roll and pitch denoted as β and γ, respectively. Expressing the orientation by Euler angles using the Tait-Bryan convention in the $x - y - (z)$ sequence gives:

$$\beta = \arctan 2(-\mathbf{R}(2,3), \mathbf{R}(3,3)) \tag{2.4}$$

$$\gamma = \arcsin(\mathbf{R}(1,3)) \tag{2.5}$$

Note that arcsin has infinite periodic solutions at $n \cdot 2\pi$ and a complimentary solution at $\pi - \gamma$. However, for spraying purposes it is only its principal value that is of interest. For the case of zero roll, both solutions yield the same x-vector, while the z-vector is flipped.

2.1.1 Denavit- Hartenberg Parameters

Table 2.1: Denavit–Hartenberg parameters for AMV 4200H. The ranges of the definition are given in degrees.

i	θ_i	d_i	a_i	α_i	Range
1	θ_1	d_1	$-a_1$	$\frac{\pi}{2}$	$\theta_1 \in [-65, 65]^\circ$
2	$\theta_2 + \frac{\pi}{2}$	0	a_2	$\frac{\pi}{2}$	$\theta_2 \in [-16, 57]^\circ$
3	0	d_3	0	0	$d_3 \in [7.012, 15.012]\ m$
4	θ_4	0	$-a_4$	$\frac{\pi}{2}$	$\theta_4 \in [-180, 180]^\circ$
5	θ_5	$-d_5$	$-a_5$	$-\frac{\pi}{2}$	$\theta_5 \in [-118, 62]^\circ$
6	0	d_6	0	0	–

The dimensions in Table. 2.1 is found in Appendix B.1.

2.1.2 Reachable Workspace

The reachable workspace of the end effector is plotted by looping the forward kinematics through the ranges of definition given in Table 2.1. The two last joints, (θ_4, θ_5) are excluded for simplicity due to little contribution to the reachable workspace. Note that only the reachable workspace, i.e. the volume reachable in at least one orientation is illustrated. It does not represent the dexterous workspace, which is the volume reachable in all orientations.

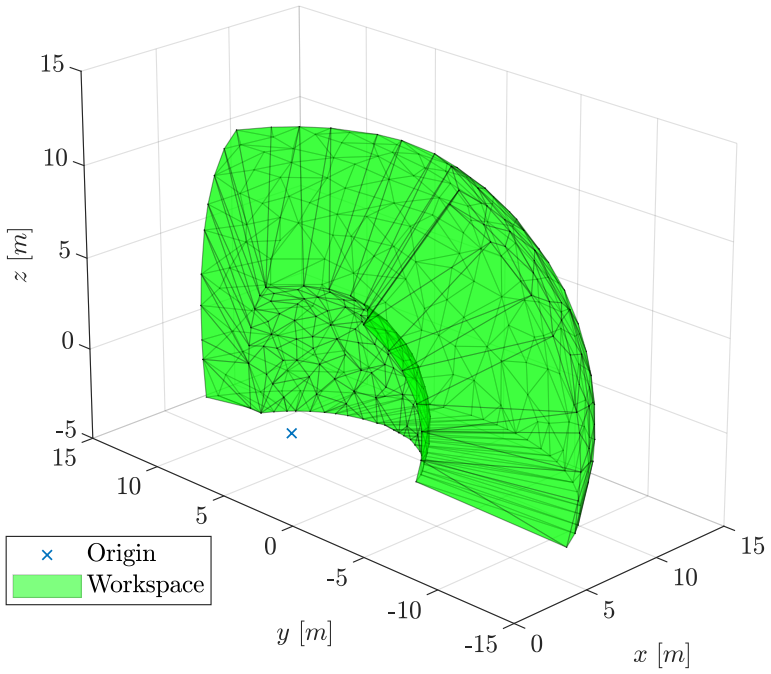


Figure 2.4: Reachable workspace for the boom assembly of AMV 4200H, excluding the nozzle.

2.1.3 Nozzle and Eccentric

During application of shotcrete, the nozzle is rotated in small overlapping circles. This is done to improve surface finish, facilitate mixing and improve the overall structural integrity of the shotcrete. The nozzle rotates on an eccentric, i.e. a disk with an axle offset from the centre of rotation, see Figure 2.5. The eccentric is driven by a hydraulic motor and is connected to the nozzle body, hinged to a universal coupling on the connecting bracket. As the motor rotates, the nozzle will follow in a circular motion within a cone whose origin is located directly below the universal coupling. The distance L_{offset} is the constant offset between the centre of rotation (CR) for the nozzle and the coordinate system of Joint 6. Rotation of the nozzle creates a cone with an angular opening of $\varphi \approx 4,5$ degrees. Note that the nozzle tip follows a skewed circle within the cone due to the mechanical constraints of the universal coupling, resulting in the nozzle moving back and forth in the z-direction as it rotates about the CR.

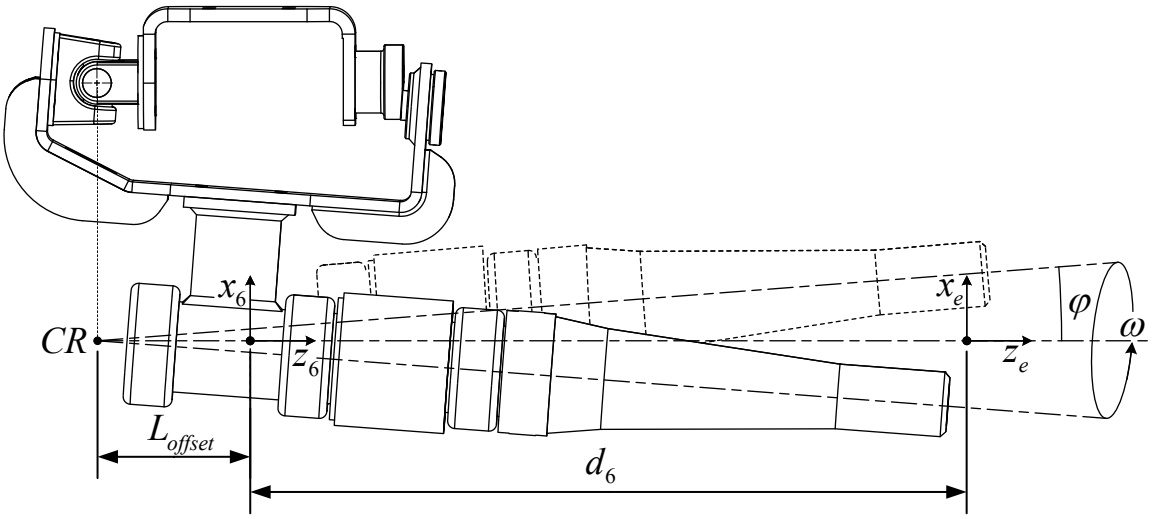


Figure 2.5: Nozzle and eccentric; dotted line shows nozzle in the upper position. The solid line shows the nozzle in the lower position. Note that nozzle rotation is out-of-plane.

The nozzle nutation is not considered in the kinematic analysis since its purpose is to disperse the shotcrete jet. As a simplification for the kinematics, the nozzle centre point (NCP) is defined as lying on a straight line drawn from the CR at a distance d_6 from coordinate system 6. The value of d_6 is considered to be the middle value of the nozzle tip translation in the z-direction due to the eccentric. The simplification contributes to an inaccuracy of the nozzle position by $\approx \pm 19$ mm in the z_e-direction.

The nutation velocity ω, can be measured by an encoder or estimated, since the displacement of the hydraulic motor is known and the incoming flow is measured. By estimation, the nozzle rotational velocity is

$$\omega = \frac{Q_m}{v_m} \quad [rad/s] \tag{2.6}$$

where v_m is the motor displacement and Q_m is the volumetric flow rate. Assuming steady-state operation and no radial acceleration, the mean spread radius of shotcrete on the tunnel surface can be approximated as

$$r_w = \tan(\varphi) \cdot (L_{offset} + d_6 + D_s) \quad [m] \tag{2.7}$$

where

r_w : Spread on surface $[m]$
D_s : Distance to surface $[m]$
φ : Nozzle spread angle $[rad]$

2.1.4 Nonlinear Actuation of Joint 2

All rotational links with the exception of θ_2 are driven by hydraulic motors. The angular displacements can be assumed directly proportional to the volumetric flow through the motors. Consequently, a linear conversion between the actuator space and joint space can be assumed. However, Joint 2, with angle θ_2 is manipulated by means of two hydraulic cylinders, attached symmetrically about the rotation axis. The geometry – where the cylinders are not tangent to the arc of the driven element – is illustrated in Figure 2.6. This geometric condition prevents a direct variation between the volumetric flow in the actuators and the angular displacement θ_2. The relation between cylinder length L_C and joint angle θ_2 is solvable by examining the geometry of the mechanism. L_C is

the distance between the upper and lower attachment points. Applying the Pythagorean theorem for the cylinder length yields:

$$L_C(\theta_2) = \sqrt{(L_{y1} + L_{y2}\cos\theta_2 - L_{x2}\sin\theta_2)^2 + (L_{x2}\cos\theta_2 - L_{x1} + L_{y2}\sin\theta_2)^2} \quad [m] \tag{2.8}$$

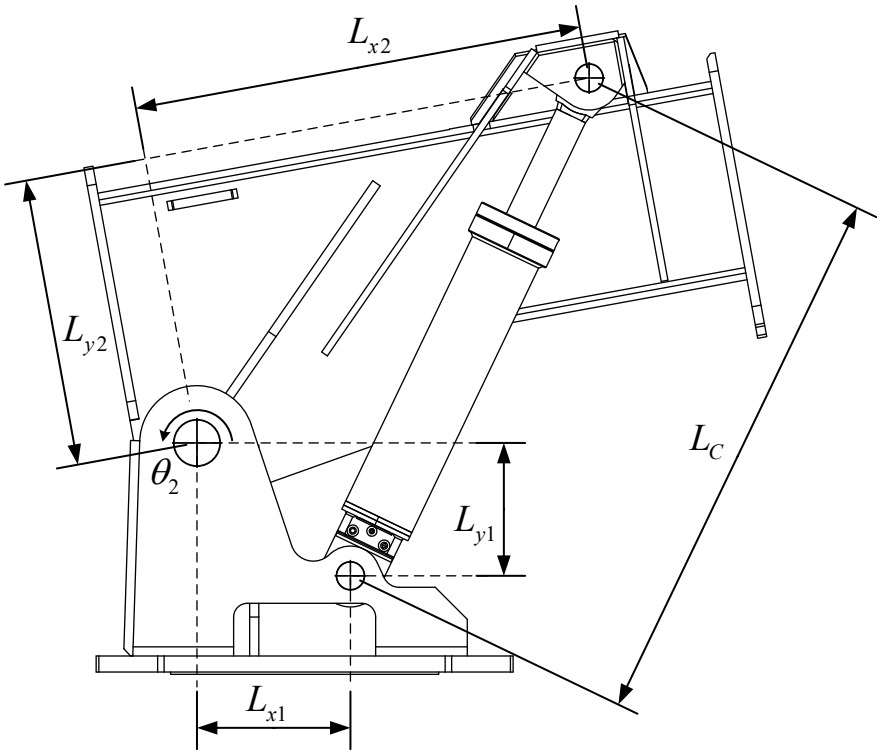


Figure 2.6: Actuating mechanism for joint θ_2.

The function is evaluated within the operating range from Table 2.1. In Figure 2.7 the real relation is plotted and compared to a proportional relation between θ_2 and L_C. The dimensions are presented in Appendix B in Table B.2.

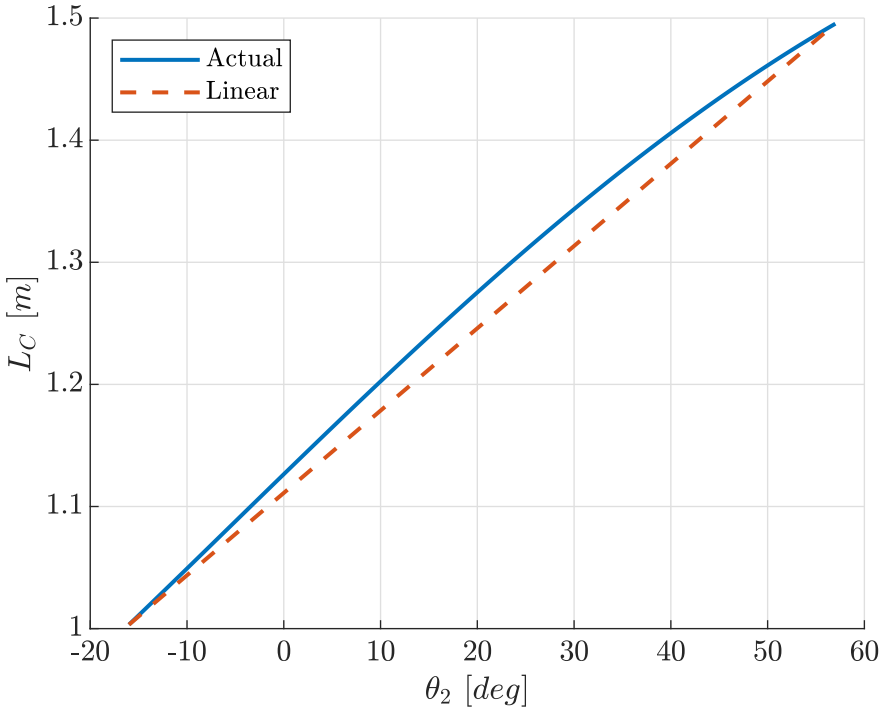


Figure 2.7: Cylinder length to angle relation for θ_2.

2.2 Inverse Kinematics

Whereas the forward kinematics describes the pose of the end effector in Euclidean space, the inverse kinematics defines the joint variables as a function of the end effector pose. A block-diagram representation is illustrated in Figure 2.8. Given that the robot in question is a 5-DOF machine, the reachable workspace lies in a 5-DOF subspace (Craig, 2014). An example could be to control a line segment in space, where the rotation about the line is not considered. Similarly, for shotcrete spraying, the nozzle rotation about its own axis does not affect the task.

Figure 2.8: Inverse kinematics: From Cartesian coordinates to Joint Space.

No exact closed-form solution has been found for the manipulator and as such the inverse kinematics is divided into two analytical segments using kinematic decoupling. The first segment is defined as the transformation from the base to the wrist frame, illustrated in Figure 2.3 as frame 4. This transformation incorporates the three first joints $(\theta_1, \theta_2, d_3)$, and is solved in terms of translation. The second segment is specified as the remaining transformation, called the wrist. It is comprised of the last two joints (θ_4, θ_5) and the nozzle. This transformation is solved in terms of orientation. Both segments are solved analytically and coupled to obtain a complete solution. Because the wrist is offset and translates the nozzle, it affects the final position, rendering the calculation incorrect with respect to position. However, the calculation serves as an initial guess for an iterative solver, where the final values are resolved numerically. A flow chart representation is presented in Figure 2.9 comprising the following operations:

- **Start:** Position error magnitude is initialised to infinity in order to trigger the while condition.
- **Inverse Kinematics Boom:** Inverse kinematics for the first transformation is calculated analytically such that the wrist frame coincides with the goal position.
- **Inverse Kinematics Wrist:** The wrist mechanism inverse kinematics is calculated to obtain the correct nozzle orientation.
- **Forward Kinematics:** Forward kinematics is applied to the proposed solution to acquire the pose of the proposed solution.
- **Offset Boom Reference:** The position error is calculated and used as the offset for the first transformation.
- **Calculate Position Error Magnitude:** Position error magnitude is updated and evaluated against the tolerance.

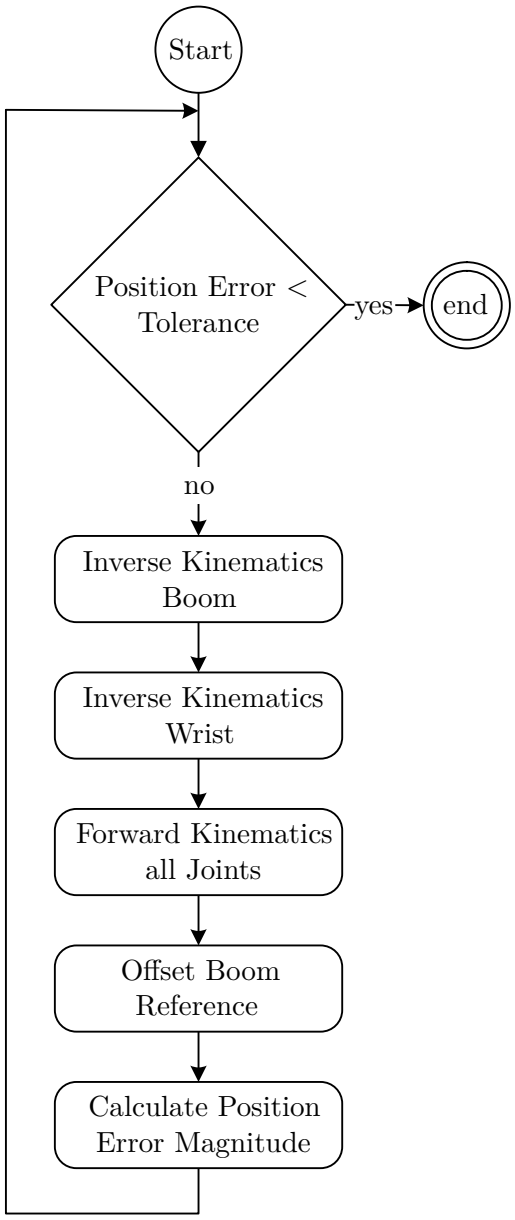


Figure 2.9: Flow chart representation of inverse kinematics algorithm.

2.2.1 Solution for Boom

Wrist frame position is controlled using the first three joints θ_1, θ_2, d_3 in a similar configuration to a polar robot. The first joint θ_1 controls the slewing motion of the boom about the global yaw axis, meaning that the joint only affects the x-y-position of the wrist frame. The second joint θ_2 controls the boom inclination, and the prismatic joint d_3 controls the boom length. Extracting the relevant transformations from Eq. (2.3) gives:

$$\mathbf{T}_4^1 = \mathbf{T}_2^1 \cdot \mathbf{T}_3^2 \cdot \mathbf{T}_4^3 \tag{2.9}$$

Expressions for the wrist frame coordinates are extracted

$$x_w = \mathbf{T}_4^1(1,4) = -\cos(\theta_1)\left(a_1 + a_2\,\sin(\theta_2) - d_3\cos(\theta_2)\right) \quad [m] \tag{2.10}$$

$$y_w = \mathbf{T}_4^1(2,4) = -\sin(\theta_1)\left(a_1 + a_2\,\sin(\theta_2) - d_3\cos(\theta_2)\right) \quad [m] \tag{2.11}$$

$$z_w = \mathbf{T}_4^1(3,4) = d_1 + a_2\,\cos\left(\theta_2\right) + d_3\,\sin\left(\theta_2\right) \quad [m] \tag{2.12}$$

and solved for the joints

$$\theta_1 = \arctan\left(\frac{y_w}{x_w}\right) \quad [rad] \tag{2.13}$$

$$\theta_2 = 2\arctan\left(\frac{\sqrt{2a_1x_w\mathrm{c}(\theta_1) + (a_1^2 - a_2^2 - 2d_1z_w + z_w^2)\mathrm{c}(\theta_1)^2 + x_w^2} + \mathrm{c}(\theta_1)(z - d_1)}{x_w + (a_1 + a_2)\mathrm{c}(\theta_1)}\right) - \frac{\pi}{2} \quad [rad] \tag{2.14}$$

$$d_3 = \frac{-d_1 + z_w - a_2 \cdot \mathrm{c}(\theta_2)}{\mathrm{s}(\theta_2)} \quad [m] \tag{2.15}$$

providing a closed-form solution to the inverse kinematics of the wrist frame position. Note that s, c are abbreviations for the sine and cosine functions, respectively.

2.2.2 Solution for Wrist

The principal functionality for the wrist transformation is to manipulate the nozzle orientation, and thus the inverse kinematics are solved in terms of roll and pitch. To define the pitch and roll, we start by examining the product of the nozzle spatial rotations about the global x and y-axes, and the orientation obtained through the forward kinematic transformations

$$\mathbf{T}_e^1(1{:}3, 1{:}3) = \mathbf{R}_e = \mathbf{Rot}_x(\beta) \cdot \mathbf{Rot}_y(\gamma) \tag{2.16}$$

The matrix $\mathbf{R}_e$ has dimension $(3, 3)$. However, since a 5-DOF machine can only guide a line segment in space, the complete pose cannot be determined arbitrarily. The kinematics is configured such that the nozzle aligns with the z-axis of the end frame, meaning that the desired nozzle orientation is described using the third column in the rotation matrix, producing the following equation:

$$\mathbf{R}_e(1{:}3, 3) = \mathbf{T}_e^1(1{:}3, 3) \tag{2.17}$$

Which, when expanded becomes:

$$\mathrm{s}\left(\gamma\right) = \mathrm{c}\left(\theta_1\right)\,\mathrm{c}\left(\theta_5\right)\,\mathrm{s}\left(\theta_2 + \frac{\pi}{2}\right) - \mathrm{s}\left(\theta_5\right)\,\left(\mathrm{s}\left(q_1\right)\,\mathrm{s}\left(\theta_4\right) + \mathrm{c}\left(q_1\right)\,\mathrm{c}\left(\theta_4\right)\,\mathrm{c}\left(\theta_2 + \frac{\pi}{2}\right)\right) \tag{2.18}$$

$$-\mathrm{c}\left(\gamma\right)\,\mathrm{s}\left(\beta\right) = \mathrm{s}\left(\theta_5\right)\,\left(\mathrm{c}\left(q_1\right)\,\mathrm{s}\left(\theta_4\right) - \mathrm{c}\left(\theta_4\right)\,\mathrm{c}\left(\theta_2 + \frac{\pi}{2}\right)\,\mathrm{s}\left(q_1\right)\right) + \mathrm{c}\left(\theta_5\right)\,\mathrm{s}\left(q_1\right)\,\mathrm{s}\left(\theta_2 + \frac{\pi}{2}\right) \tag{2.19}$$

$$\mathrm{c}\left(\beta\right)\,\mathrm{c}\left(\gamma\right) = -\mathrm{c}\left(\theta_5\right)\,\mathrm{c}\left(\theta_2 + \frac{\pi}{2}\right) - \mathrm{c}\left(\theta_4\right)\,\mathrm{s}\left(\theta_5\right)\,\mathrm{s}\left(\theta_2 + \frac{\pi}{2}\right) \tag{2.20}$$

Note that, also here, s, c are abbreviations for the sine and cosine. Assuming that we know the values for the first rotational joints (θ_1, θ_2) and orientation (β, γ), the set of equations is solved using MATLAB. The solution yields extensive expressions for the wrist joints (θ_4, θ_5) and is intentionally not included in the text. The source code is presented in Appendix E.1.1.

2.3 The Jacobian Matrix

The Jacobian matrix has many useful applications in robotics. It provides a relationship between the system description in operational space and joint space. The Jacobian can be used to describe and deduce joint space velocities, accelerations and forces, as well as determine Cartesian forces, velocities and accelerations based on values in joint space. In addition, it is a useful tool in the analysis of singularities, and can be used for numerical computation of the inverse kinematics. In this thesis, the Jacobian is not used explicitly but is provided for future work and as a part of the system description.

The Jacobian matrix defines a linear mapping that relates joint space velocities, $\dot{\mathbf{q}}$ to the Cartesian velocity of the end effector, $\dot{\mathbf{x}}_\mathbf{e}$:

$$\dot{\mathbf{x}}_\mathbf{e} = \mathbf{J}(\mathbf{q})\dot{\mathbf{q}} \tag{2.21}$$

Applying the chain rule to Eq. (2.21), the expression for end effector acceleration, $\ddot{\mathbf{x}}_\mathbf{e}$ becomes:

$$\ddot{\mathbf{x}}_\mathbf{e} = \mathbf{J}(\mathbf{q})\ddot{\mathbf{q}} + \dot{\mathbf{J}}(\mathbf{q})\dot{\mathbf{q}} \tag{2.22}$$

From Eq. (2.21) and Eq. (2.22) it is also possible to obtain the expressions for joint space velocities and accelerations. The equations are derived using algebra and the inverse Jacobian

$$\dot{\mathbf{q}} = \mathbf{J}^{-\mathbf{1}}(\mathbf{q})\dot{\mathbf{x}} \tag{2.23}$$

$$\ddot{\mathbf{q}} = \mathbf{J}^{-\mathbf{1}}(\mathbf{q})(\ddot{\mathbf{x}} - \dot{\mathbf{J}}(\mathbf{q})\dot{\mathbf{q}}) \tag{2.24}$$

Using the Jacobian to calculate joint space velocities and accelerations does however require that the matrix is non-singular, i.e. that there exists an inverse. For the Jacobian to be invertible, it must be square (n-by-n), and the determinant must be different from zero. The geometric Jacobian of the AMV 4200H is non-square with dimension $(6, 5)$, this is because the workspace has six DOF, and the machine is underactuated with only five joint variables. The geometric Jacobian contains the derivatives of nozzle translation (x, y, z) and orientation column vectors $(\mathbf{r}_x, \mathbf{r}_y, \mathbf{r}_z)$, with respect to all five joint variables. Since there are only five joint variables and the geometric Jacobian has rank six, matrix multiplication is not defined. Therefore, we require that the Jacobian is square with dimensions 5-by-5 such that it is possible to associate the number of reference variables to the number of DOFs on the machine. This is achieved by exchanging the derivatives of $(\mathbf{r}_x, \mathbf{r}_y, \mathbf{r}_z)$ with the derivatives of the expressions for β (roll) and γ (pitch) from Eq.(2.4) and Eq.(2.5), discussed in Section 2.1. A relation between the actual and controllable DOFs is obtained

$$\dot{\mathbf{x}}_\mathbf{e} = \begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{z} \\ \dot{\beta} \\ \dot{\gamma} \end{bmatrix} = \mathbf{J}(\mathbf{q}) \cdot \begin{bmatrix} \dot{q_1} \\ \dot{q_2} \\ \dot{q_3} \\ \dot{q_4} \\ \dot{q_5} \end{bmatrix} \tag{2.25}$$

where β, γ represents roll and pitch of the nozzle respectively. This formulation enables the matrix product to be defined as well as the inverse of the Jacobian, given $\det(\mathbf{J}) \neq 0$. On this basis, an analytical Jacobian matrix can be formed as the partial derivative of the Cartesian variables with

respect to all joint variables:

$$\mathbf{J}(\mathbf{q}) = \frac{\partial f_i}{\partial q_j} = \begin{bmatrix} \frac{\partial x}{\partial q_1} & \frac{\partial x}{\partial q_2} & \frac{\partial x}{\partial q_3} & \frac{\partial x}{\partial q_4} & \frac{\partial x}{\partial q_5} \\ \frac{\partial y}{\partial q_1} & \frac{\partial y}{\partial q_2} & \frac{\partial y}{\partial q_3} & \frac{\partial y}{\partial q_4} & \frac{\partial y}{\partial q_5} \\ \frac{\partial z}{\partial q_1} & \frac{\partial z}{\partial q_2} & \frac{\partial z}{\partial q_3} & \frac{\partial z}{\partial q_4} & \frac{\partial z}{\partial q_5} \\ \frac{\partial \beta}{\partial q_1} & \frac{\partial \beta}{\partial q_2} & \frac{\partial \beta}{\partial q_3} & \frac{\partial \beta}{\partial q_4} & \frac{\partial \beta}{\partial q_5} \\ \frac{\partial \gamma}{\partial q_1} & \frac{\partial \gamma}{\partial q_2} & \frac{\partial \gamma}{\partial q_3} & \frac{\partial \gamma}{\partial q_4} & \frac{\partial \gamma}{\partial q_5} \end{bmatrix} \tag{2.26}$$

For more details on obtaining the Jacobian matrix, the reader is referred to Appendix E.1.2.

2.4 Singular Configurations

When controlling a manipulator in Cartesian space it is desirable to know the limitations. The limitations of a manipulator are referred to as singularities and typically represent configurations at which a machine loses one or more degrees of freedom. The effects of kinematic singularities are of interest for the following reasons summarised by Siciliano et al. (2009, p. 116):

(a) *Singularities represent configurations at which mobility of the structure is reduced, i.e., it is not possible to impose an arbitrary motion to the end effector.*

(b) *When the structure is at a singularity, infinite solutions to the inverse kinematics problem may exist.*

(c) *In the neighbourhood of a singularity, small velocities in the operational space may cause large velocities in the joint space.*

Furthermore, it is common to classify singularities as either boundary singularities, i.e. singularities that occur at the boundary of the reach of the manipulator, or as internal singularities, which are caused by geometric relations in joint space.

The singularities can be identified by examining the Jacobian matrix. Singularities are found where the Jacobian matrix is rank-deficient. If the Jacobian is rank deficient, it means that one or more mathematical couplings are missing between the description in joint space and Cartesian space, hence the controllability is degraded. Furthermore, the geometric locations and analytical expressions for the singularities can be identified by solving the determinant of the Jacobian equal to zero, $\det(\mathbf{J}) = 0$. However, the Jacobian of the AMV 4200H is very complex, and there are no analytical solutions to the expression $\det(\mathbf{J}) = 0$.

Since the singular configurations are hard to obtain analytically, they may be found empirically. By empirical testing and analysis. No joint space singularities were found for the AMV 4200H. However, it is subject to one representational singularity, when the nozzle z_e-axis is collinear to the global x-axis, i.e. when in the home position.

2.5 Alternative Approach for Inverse Kinematics

As an alternative approach to the inverse kinematics problem, Conformal Geometric Algebra (CGA) can be used to obtain solutions to problems that are not solvable with traditional methods. As an example, Kleppe and Egeland (2016) showed that it is possible to obtain a closed-form solution for the Universal Robots UR5. In addition, Tørdal et al. (2016) showed that by using GCA, the inverse kinematics of the COMAU Smart-5 NJ-110 can be solved 45 times faster than the software provided by the manufacturer. CGA is a method for assessing geometric shapes from a higher-dimensional space. In the case of using CGA to evaluate shapes in 3D-space, the problem is projected into a 5D vector space. The benefit of CGA is that operations such as reflections, rotations and translations become effortless to solve in conformal space. For this reason, an attempt was made to solve the inverse kinematic problem by CGA using the visualisation software developed by Perwass (2004). However, no solution was identified as a consequence of the joint offsets between $\theta_1 - \theta_2$ and $\theta_4 - \theta_5$. The authors were not able to form a set of shapes and intersections that unambiguously produced an exact solution.

Chapter 3

Trajectory Planning

Trajectory planning is the procedure of determining the desired motion of an end frame. The motion is described as a timeseries of the desired position and orientation at given points in time. For this particular application, the process is three-fold. First, the workspace needs to be determined in terms of a surface. The surface dimensions, in combination with the desired shape and velocity parameters, lay the foundation for creating a planar spraying pattern. Finally, the pattern is mapped onto the surface to obtain position references in terms of Cartesian coordinates and Euler rotations. The sequence is illustrated in Figure 3.1, where each block is described further in their respective sections.

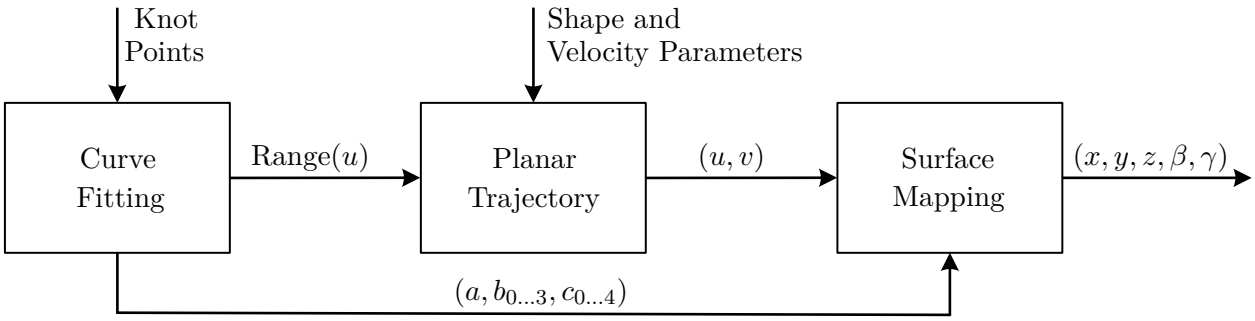


Figure 3.1: Block diagram representation of trajectory planning strategy.

3.1 Surface Mapping

As a simplification, the tunnel is parametrised as a cylinder-like, developable surface, i.e. a surface with zero Gaussian curvature that can be flattened without any distortion. The advantage of this simplification is that the spraying pattern can be designed in planar coordinates while the distances and velocities are preserved when transforming to 3D-space. However, it assumes that the tunnel wall topology is smooth and that the cross section is constant. The resulting coordinate systems are illustrated in Figure 3.2.

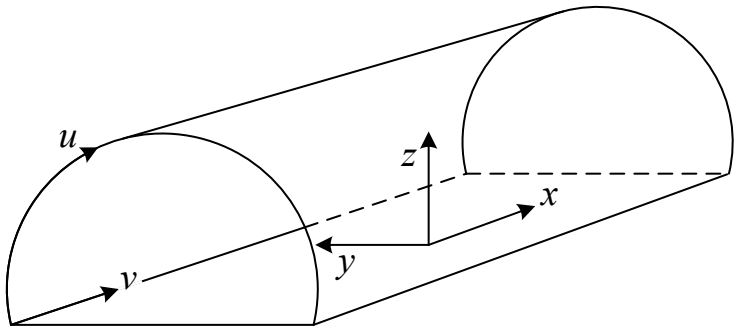


Figure 3.2: Planar and Cartesian coordinate systems in relation to tunnel geometry.

3.1.1 Tunnel Geometry

The Norwegian Public Roads Administration has issued standard cross-sectional profiles for road tunnels (Norwegian Public Roads Administration, 2004). There are two main profile types, where the dimensions are dependent on the average annual daily traffic (AADT). Both profiles are illustrated in Figure 3.3.

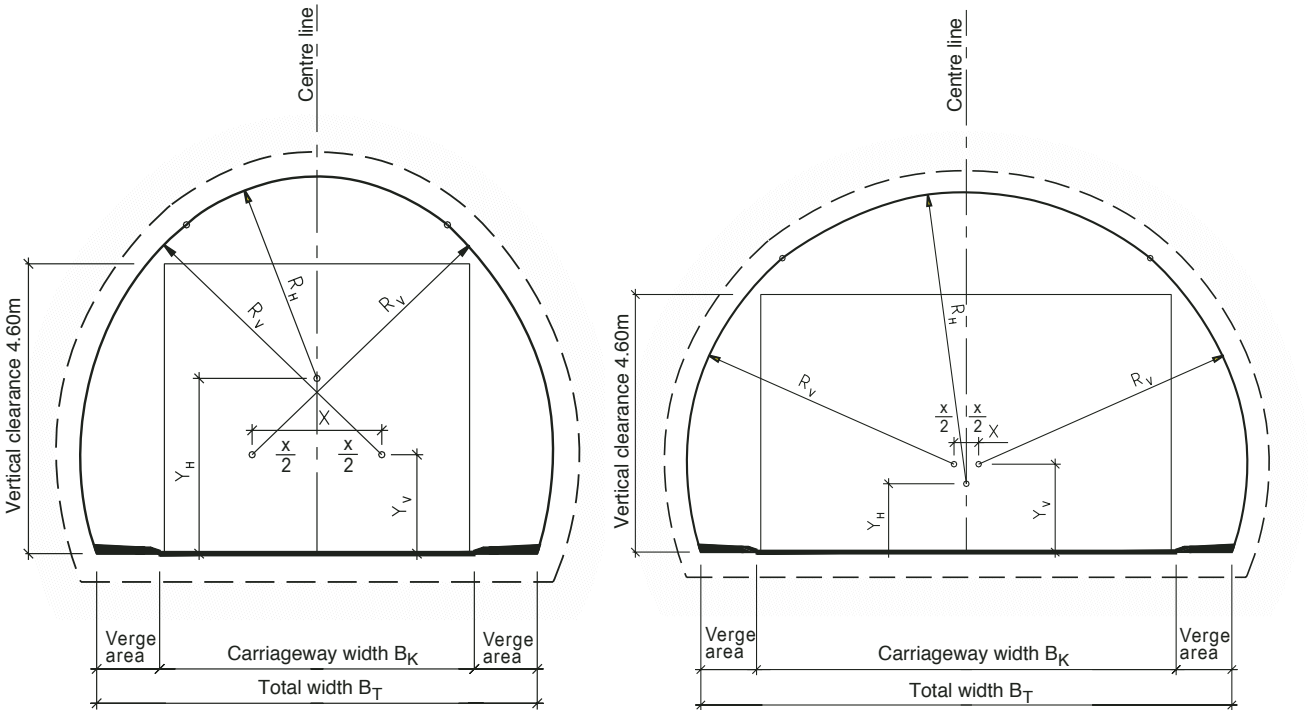


Figure 3.3: Standard cross sections for Norwegian road tunnels. Courtesy of the Norwegian Public Roads Administration.

Parametric polynomial functions are commonly used to describe arcs and as such, the (y, z)-coordinates are parameterised with respect to u. Increasing the polynomial degree enhances flexibility in terms of shaping, by increasing the maximum number of inflexion points. However, a higher degree polynomial can cause unwanted oscillatory behaviour. Therefore, the parametric polynomial degrees should be kept as low as possible, but high enough to capture the tunnel geometry. Exact replicas of the major Norwegian road tunnel profiles were used as references to evaluate the necessary polynomial degree. The different polynomial degrees were increased and compared to the reference tunnel until a significant drop in mean square error (MSE) was achieved. Results for the T8.5 profile is presented in Table 3.1. Results for all major Norwegian road tunnel profiles are presented in Appendix C.1.

Table 3.1: Mean square error for best fit of y and z parametric functions by polynomial degree for T8.5 profile.

Degree		1	2	3	4
MSE	y $[m]$	0.5048	0.5048	0.0020	-
	z $[m]$	4.1432	0.0376	0.0376	0.0001

Results for the parametric curves are illustrated in Figure 3.4. An overview of all the common tunnel profiles is presented in Appendix C.

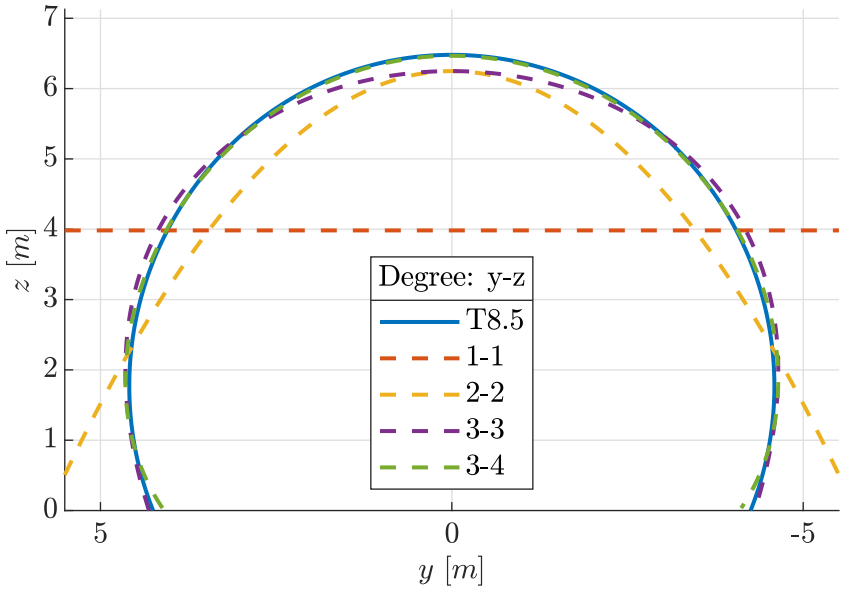


Figure 3.4: Parametric tunnel approximation for profile T8.5 by polynomial degrees of y and z functions.

The analysis shows that cubic- and quartic functions are sufficient to describe the y and z-coordinates respectively. Parametric equations describing the Cartesian coordinates in terms of u, v coordinates become

$$x(v) = v + a \quad [m] \tag{3.1}$$

$$y(u) = b_3 \cdot u^3 + b_2 \cdot u^2 + b_1 \cdot u + b_0 \quad [m] \tag{3.2}$$

$$z(u) = c_4 \cdot u^4 + c_3 \cdot u^3 + c_2 \cdot u^2 + c_1 \cdot u + c_0 \quad [m] \tag{3.3}$$

where coefficients a, b_i and c_i are determined from the tunnel dimensions.

3.1.2 Normal Vector

Keeping the nozzle relatively normal to the surface is instrumental to reduce rebound and overspray and as such an expression for the normal vector is required. The normal vector to a parametric surface at some point is defined by Marsden and Tromba (2012) as

$$\mathbf{N}_{u,v} = \mathbf{T}_u \times \mathbf{T}_v \tag{3.4}$$

$$\mathbf{T}_u = \left[\frac{\partial x}{\partial u}, \frac{\partial y}{\partial u}, \frac{\partial z}{\partial u}\right] \tag{3.5}$$

$$\mathbf{T}_v = \left[\frac{\partial x}{\partial v}, \frac{\partial y}{\partial v}, \frac{\partial z}{\partial v}\right] \tag{3.6}$$

where

$\mathbf{N}_{u,v}$: Normal vector in point (u, v)
$\mathbf{T}_u$: Tangent vector in u direction
$\mathbf{T}_v$: Tangent vector in v direction

Substituting in the equations for the surface model provides an expression for the normal vector in terms of u:

$$\mathbf{T}_u = \Big[0,\, 3b_3 \cdot u^2 + 2b_2 \cdot u + b_1,\, 4c_4 \cdot u^3 + 3c_3 \cdot u^2 + 2c_2 \cdot u + c_1\Big] \tag{3.7}$$

$$\mathbf{T}_v = \Big[1,\, 0,\, 0\Big] \tag{3.8}$$

$$\mathbf{N}_{u,v} = \Big[0,\, 4c_4 \cdot u^3 + 3c_3 \cdot u^2 + 2c_2 \cdot u + c_1,\, -(3b_3 \cdot u^2 + 2b_2 \cdot u + b_1)\Big] \tag{3.9}$$

The normal vector $\mathbf{N}_{u,v}$ is subsequently normalised to unit representation:

$$\hat{\mathbf{N}}_{\mathbf{u,v}} = \frac{\mathbf{N}_{u,v}}{|\mathbf{N}_{u,v}|} \tag{3.10}$$

Note that the direction of the normal vector depends on the direction of spraying. When spraying from positive y-values to negative y-values (see Figure 3.2 for reference), the right-handed coordinate system ensures that the normal points out from the tunnel surface, i.e. pointing outwards from the cross-sectional centre. In the opposite case, spraying from negative y-values to positive y-values, the sign of u changes, and hence the direction of the normal vector is flipped. Therefore the direction of the normal is adjusted accordingly in the software.

3.2 Curve Fitting

With the surface model in place, the coefficients (a, $b_{0..3}$, $c_{0..4}$) need to be approximated according to the tunnel geometry. Coefficients ($b_{0..3}$, $c_{0..4}$) define the cross sectional shape of the tunnel, while a specifies the depth offset between (u, v) and (x, y, z). The approximation is subject to two goals; the resulting surface needs to be as close to the actual tunnel geometry as possible, and should provide a developable surface approximation. The requirements are satisfied using a two-step procedure. First, the shape is created with an initial parameterisation, using the parameter $s = [0 \dots 1]$. From this shape, the arc length is estimated, and the curve is reparameterised to the parameter $u = [0 \dots L_{arc}]$ such that line integrals in the plane and surface are equal.

Values for ($b_{0..3}$, $c_{0..4}$) are approximated by curve fitting using the least squares method. Observations are provided by the operator, where the NCP is navigated manually to collect knot points. Therefore, the number of observations will be sparse compared to e.g. a LIDAR-generated point cloud. For a parametric equation, the number of observations must be at least be equal to the number of coefficients to be determined. Therefore, an n-th degree polynomial requires $n + 1$ observations. In this case, the expression for z is a 4th-degree polynomial, and as such requires at least five observations.

As the observations are collected manually by an operator, they cannot be assumed to be uniformly spaced along the parametric curve. Determining the correct parameter value for each knot point is a complex issue which can be solved by numerical methods (Grossman, 1971). In this thesis, the parameter values are approximated linearly by connecting the knot points with chords. The sum of all chords is regarded as a downscale of the true arc length and is the denominator in the

following equation

$$s_k = \frac{\sum_{i=2}^{k} \sqrt{(z_i - z_{i-1})^2 + (y_i - y_{i-1})^2}}{\sum_{i=2}^{n} \sqrt{(z_i - z_{i-1})^2 + (y_i - y_{i-1})^2}} \quad [-] \tag{3.11}$$

where

s_k : Estimated parameter value for knot point k

n : Total number of knot points

The chord-length approximation is demonstrated as a comparison to a regular approximation, assuming even spacing in Figure 3.5. The first and last knot points are placed at the start and end of the curve, while the other points are placed randomly within $(\frac{1}{4}, \frac{2}{4}, \frac{3}{4}) \pm 1\,[m]$ respectively. The chord length approximation always gives a more accurate result than the uniform spacing-method.

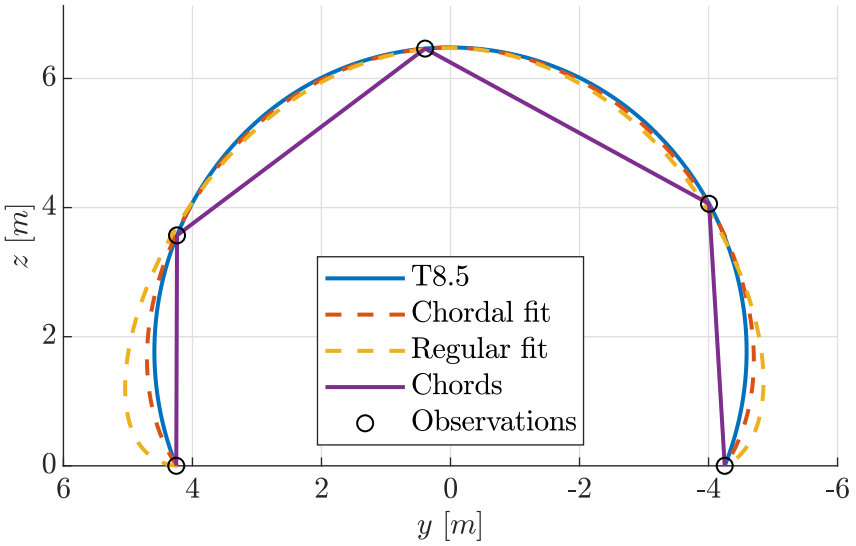


Figure 3.5: Curve fitting from five pseudo-random knot points for T8.5 profile.

While the shape is determined, the surface is still not conforming with the requirement of developability. Arc length, L_{arc} is assessed by dividing the curve into segments and summing the Euclidean distances, i.e. the chords. Due to the discretisation, the precision is directly affected by the number of knot points. All of the six major tunnel profiles were evaluated to investigate a feasible resolution. Approximated arc lengths by the number of line segments are presented in Figure 3.6. The approximations converge within a resolution of 20 chords.

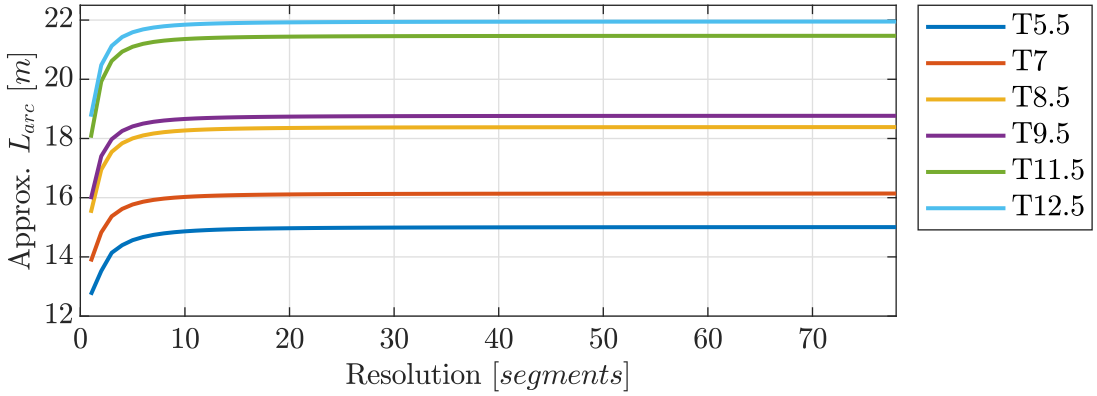


Figure 3.6: Arc length approximation by discretisation resolution.

The parametric curve is reparameterised to attain the parameter range $u = [0 \dots L_{arc}]$. Parameter values for each observation are approximated using the same formula as before, where the arc length is known:

$$u_k = \frac{1}{L_{arc}} \cdot \sum_{i=2}^{k} \sqrt{(z_i - z_{i-1})^2 + (y_i - y_{i-1})^2} \quad [m] \tag{3.12}$$

The Python implementation of the methods above is found in Appendix E.2.3.

3.3 Planar Trajectory

The objective of the planar trajectory is to allow the nozzle to spray an area as large as possible with continuous motion while the keeping movement of the manipulator to a minimum. Traditionally, the pattern resembles a square wave, providing an impractical trajectory for the corners. Constant velocity through 90 degree turns requires unbounded acceleration, which is undesirable. Firstly it is not possible to achieve in reality. Secondly, it contributes to vibrations and strain on the mechanical system. Therefore a new pattern is proposed, where the transitions between strokes are semicircular arcs. Both patterns are illustrated in Figure 3.7. Strokes are chosen to be horizontal (x-direction in Figure 3.2) to achieve minimum energy expenditure during operation, since the boom elevation stays near constant during horizontal strokes. The long boom construction of the AMV 4200H is prone to vibrations and resonance. Therefore, attention is given to make the transition between strokes smoother by rounding the transitions.

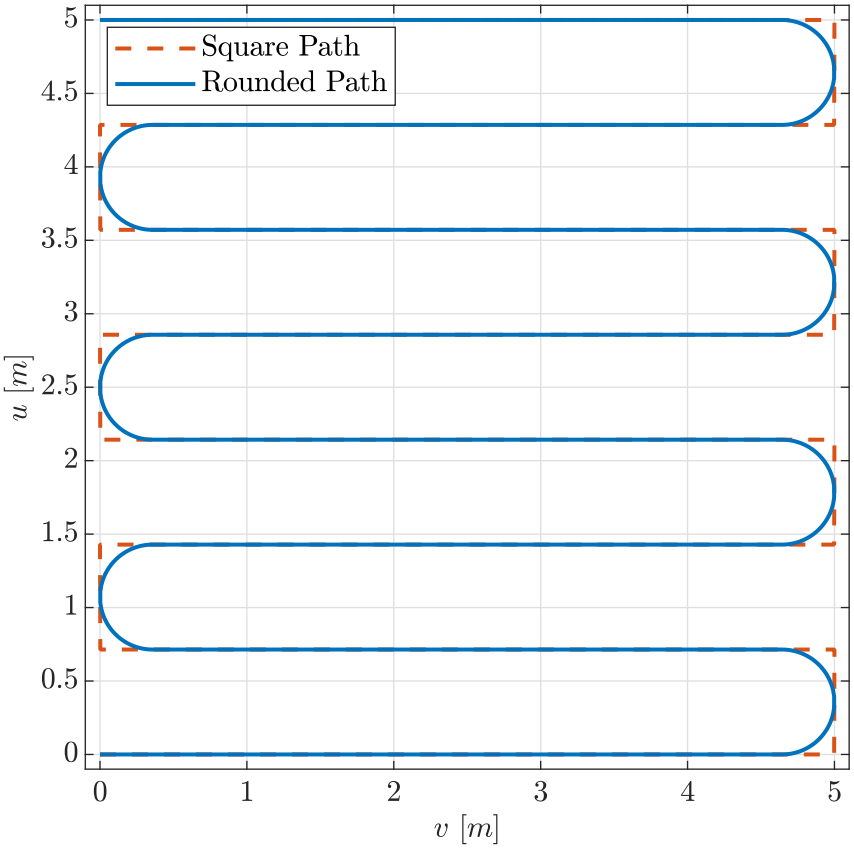


Figure 3.7: Planar trajectory curve with semicircular transitions between strokes. The square trajectory curve is superimposed with a dotted line.

The pattern is created using the driving parameters u and v, where u describes the translation along the arc of the tunnel surface and v the depth in the x-direction in Figure. 3.2. The parameters

that define the planar pattern is specified by the operator and includes:

ss_{des} : Desired stroke spacing between each horizontal segment $[m]$
d : Depth of the spraying pattern in the x-direction of the tunnel $[m]$
v_e : Velocity of the end effector along the trajectory $[m/s]$

The operator sets the parameters. Subsequently, they are corrected to fit an integer number of strokes over the given arc length of the tunnel surface. The correction is determined by finding the nearest integer to the ratio of arc length to desired stroke spacing

$$n_{seg} = \text{round}\left(\frac{L_{arc}}{ss_{des}}\right) \quad [-] \tag{3.13}$$

where L_{arc} is the arc length of the tunnel. If the number of segments, n_{seg} is an odd number, the direction of the return stroke is opposite to the starting stroke, and if the number is even, the return stroke will have the same direction as the starting segment.

When the number of segments has been calculated, the desired stroke spacing must be recalculated to accommodate the number of segments without a remainder. This is done by taking the ratio of total arc length to number of segments

$$ss_{cor} = \frac{L_{arc}}{n_{seg}} \quad [m] \tag{3.14}$$

where ss_{cor} is the corrected stroke spacing. The next step in creating the planar trajectory is to calculate the total length of the spraying path. The total path length is calculated as the sum of all segments:

$$L_{path} = (d - ss_{cor}) \cdot (n_{seg} + 1) + ss_{cor} + \pi \cdot \frac{ss_{cor}}{2} \cdot n_{seg} \quad [m] \tag{3.15}$$

From Eq. (3.15), we have an expression for the total distance travelled by the nozzle. In addition, the specified velocity, v is known, and hence the total runtime, t_{end} can be calculated:

$$t_{end} = \frac{L_{path}}{v_e} \quad [s] \tag{3.16}$$

The number of points used to describe the planar trajectory is an independent parameter, and can be chosen freely depending on the requirements for resolution. The default value is set to one point per centimetre. Once the number of points and runtime is calculated, the sample time of the trajectory can be calculated

$$dt = \frac{t_{end}}{n_p} \quad [s] \tag{3.17}$$

where dt is the sample time and n_p is the number of points for the trajectory. From the pattern parameters, the time intervals at which the pattern is horizontal or rounded can be calculated. This is done simply by dividing the total travel distances by the specified velocity

$$t_{horz} = \frac{d - ss_{cor}}{v_e} \quad [s] \tag{3.18}$$

$$t_{round} = \frac{\pi d}{2v_e} \quad [s] \tag{3.19}$$

where t_{horz} is the time for one horizontal stroke and t_{round} is the transition time between strokes. The period of the driving parameter u is equal to the time of one horizontal stroke plus the time of one transition:

$$P_u = t_{horz} + t_{round} \quad [s] \tag{3.20}$$

The algorithm for generating the planar pattern takes the above-calculated constants as input and returns the values for u, v as functions of time. The algorithm for pattern generating is shown below in Algorithm 1.

Algorithm 1: Generation of planar trajectory

input : stroke spacing, depth, velocity, number of segments, number of points, end time, sample time, stroke time, transition time, period u

output: driving parameters u and v as functions of time

for *i in number of points* **do**

time = (sample time) $\cdot i$;

period number of u = floor$\left(\dfrac{\text{time}}{\text{period } u}\right)$;

shifted time = time - (period number of u) $\cdot$ (period u);

if *shifted time* $<$ *transition time* **then**

`// u stays constant and v increases/decreases linearly`

u=(period number of u)$\cdot$ depth;

if *period number of u is even* **then**

v=velocity$\cdot$ (shifted time)+$\dfrac{\text{stroke spacing}}{2}$;

else

v=$-$velocity$\cdot$ (shifted time)+depth$-\dfrac{\text{stroke spacing}}{2}$;

else

`// create argument to drive trigonometric function`

$$\text{arg} = \frac{\pi}{\text{stroke time}} \cdot (\text{shifted time} - \text{transition time}) + \frac{3\pi}{2};$$

`// u, v in semicircle transition`

$$u = \frac{\text{stroke spacing}}{2} \cdot \sin(\text{arg}) + (\text{period number of } u) \cdot (\text{stroke spacing}) + \frac{\text{stroke spacing}}{2};$$

if *period number of u is even* **then**

$$v = \frac{\text{stroke spacing}}{2} \cdot \cos(\text{arg}) + \text{depth} - \frac{\text{stroke spacing}}{2};$$

else

$$v = \frac{\text{stroke spacing}}{2} \cdot \cos(\text{arg} + \pi) + \frac{\text{stroke spacing}}{2};$$

return u, v

Algorithm 1 does, however, leave the starting and ending segments incomplete by a distance equal to the radius of the transition, which corresponds to half the stroke spacing. Therefore, the lists for u, v are padded with the additional distances to complete the first and last strokes.

Below in Figure 3.8, the u, v coordinates used to create Figure 3.7 are presented as functions of time. This example was plotted for a tunnel with a depth of $5\,[m]$, an arc length of $5\,[m]$ and a stroke spacing of $0.714\,[m]$. The complete script for generating the planar trajectory can be found in Appendix E.2.2.

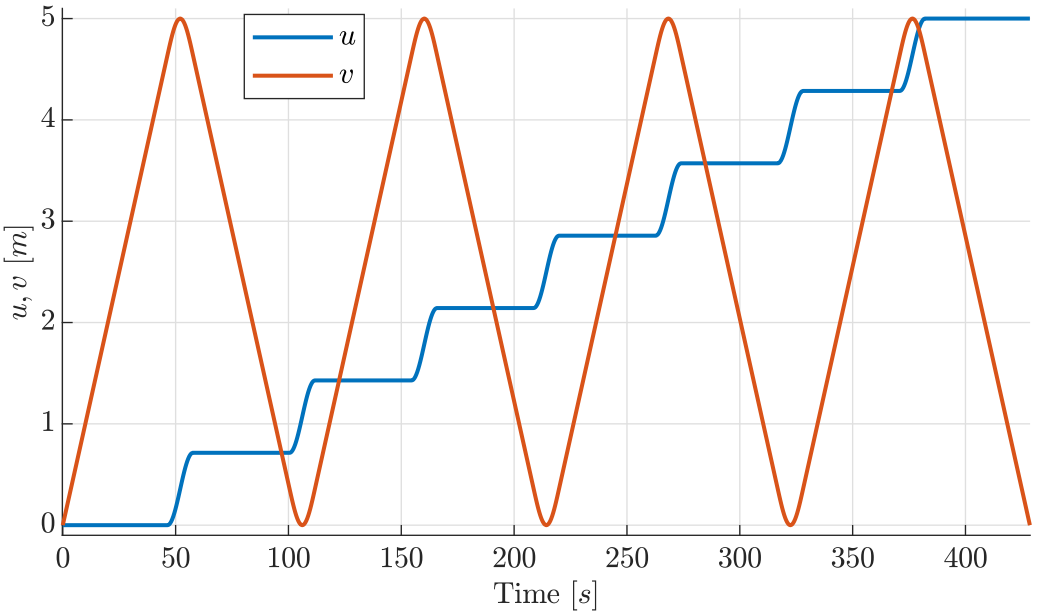


Figure 3.8: Rounded trajectory in u, v-coordinates plotted as functions of time.

3.4 Spraying Trajectory

Combining the methods discussed in this chapter; using five semi-random points to describe a T 9.5 tunnel profile, and using the aforementioned curve fitting techniques along with a planar pattern with a depth of 5 m and a stroke spacing of 0.7 m. The spraying trajectory example is presented in Figure 3.9.

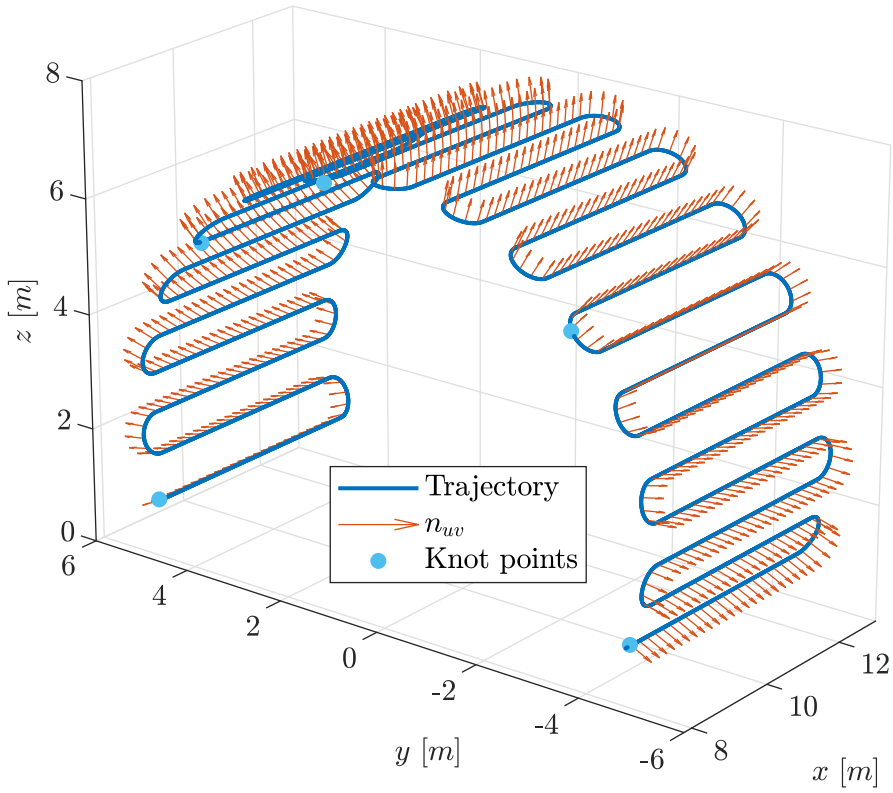


Figure 3.9: Spraying trajectory mapped onto T 9.5 surface. Knot points are indicated in blue and the normal vectors are indicated with arrows.

Chapter 4

System Modelling

The goal of system modelling is two-fold; create a model capable of emulating the actual system such that the developed framework and ECU can be tested, and to develop a rigorous understanding of system dynamics, which in turn may be used to improve the controller and framework. For the Hardware-In-the-Loop (HIL)- simulation, dynamics were simplified. In this chapter, the HIL-model is presented, subsequently, the measured validation data from the machine is used to derive a model of the system dynamics. The intention is to estimate the vibration and deflection characteristics of the boom and to evaluate its impact on the kinematics.

4.1 HIL Model

To test and verify that the Danfoss ECU can control the machine, a simulation model of the plant was developed. The model is simplified and assumes that valve commands directly affect the states of the hydraulic actuators and hence, the joint variables. The hydraulic valves on the AMV 4200H have load-independent flow control, meaning that the flow stays near constant, independent of the pressure differential. This internal valve regulation gives rise to dynamics resembling a free integrator, where the valve command is directly proportional to the velocity of the actuator. Based on these assumptions, the dynamic plant model was created as illustrated in Figure 4.1.

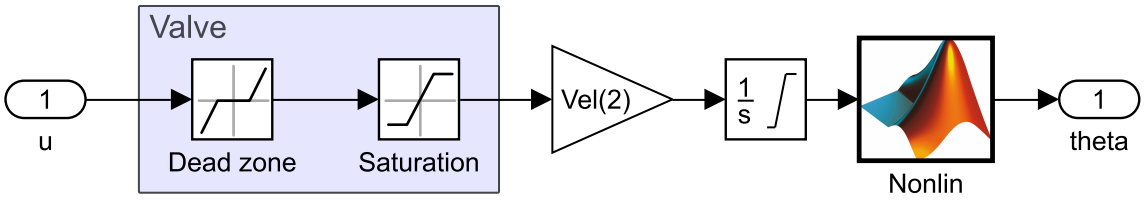


Figure 4.1: Example of a dynamic model. Note that the block "Nonlin" holds Eq. (2.8) and that it only applies to θ_2.

Below, a list is presented with explanations for each block.

- **u**: Input valve command.
- **Dead zone**: Block to simulate a range of input values where the output of the physical system is zero due to the valve deadband.
- **Saturation**: Block to limit valve signal to the upper and lower saturation values of ± 1.
- **Gain**: Scales input of **u** to actuator velocity. Maximum valve command corresponds to maximum actuator velocity.
- **Integrator**: Sums the input velocity over time and outputs actuator position. The integrator has internal saturation which ensures that joint variables are within their respective ranges of definition.
- **Nonlin***: Nonlinear relation between actuator length and joint angle θ_2. *Applies only to θ_2.

- **theta**: Output joint angle. Note that for the prismatic joint the output is d_3.

The complete HIL-setup is found in Appendix D.1.

4.2 Deadband

The deadband represents a range of valve openings that are below the threshold for achieving response in the hydromechanical system. The deadband may change dynamically based on the load condition and can, in general, be hard to model. As a simplification, the deadband is set equal to the mechanical deadband in the spool. The valves on the AMV 4200H is of the type Sauer Danfoss PVG 32, and according to the technical specification of the valves (Danfoss, 2015), the mechanical deadband of the spool under normal load conditions is:

$$\% \text{ Deadband Spool} = \pm\frac{\text{Deadband Distance}}{\text{Spool Travel Distance}} \cdot 100\% = \pm\frac{1.5\ mm}{7\ mm} \cdot 100\% \approx \pm 21.5\% \tag{4.1}$$

Note, that in practice the mechanical deadband is always present, but the total deadband will be larger depending on the load condition.

4.3 Dynamic Model

The boom assembly is long and slender, rendering a rigid approximation incorrect. Furthermore, the telescopic elements are non-uniform and slides on nylon bushings during extension and retraction. A traditional cantilever bending model is insufficient to describe the bending dynamics. In this section, the flexibility dynamics are further investigated to create a system model. The principle behind system identification is to induce an exciting input to the system and monitor the outputs. Afterwards, a model is constructed to produce a similar output signal when excited by the same input. A block diagram is illustrated in Figure 4.2, where the excitation and response data are known.

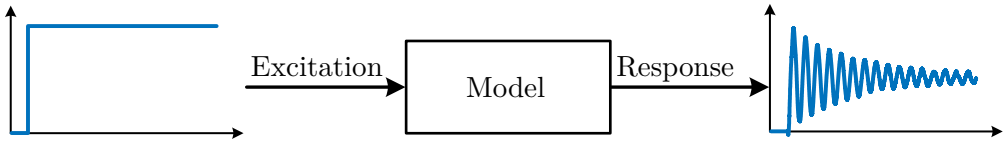


Figure 4.2: System identification block diagram.

Real validation data was measured on the AMV 4200H, and three different approaches to system identification were investigated; grey-box modelling, black-box modelling and manual vibration calculations.

4.3.1 Measurements

The physical system was measured to collect validation data of flexibility, using a Leica AT960 absolute tracker. This device tracks the position of a cat's eye-reflector at $1000\,Hz$ with an angular accuracy of $\pm 15\,\mu m + 6\,\mu m/m$ and distance accuracy of $\pm 10\,\mu m$ (Hexagon Manufacturing Intellegence, 2016). Note that the accuracy values are given in mean percentage error. The tracker was connected to a Beckhoff CX2040-series CPU module in conjunction with a load cell to monitor position and force simultaneously. Capturing the dynamics requires multiple measurements, while the tracker can only follow one reflector at a time, calling for a repeatable test. A mass m was tied to the end of the boom via the load cell; the string was cut while the force and position were recorded. The experiment manifests as underdamped free vibrations with an initial position and no initial velocity. The test was repeated four times, while positions $[1 \ldots 4]$ indicated in Figure 4.3 were measured. Likewise, this procedure was performed for four different boom lengths, ranging from fully extended to completely retracted pose. A second set of measurements were also taken

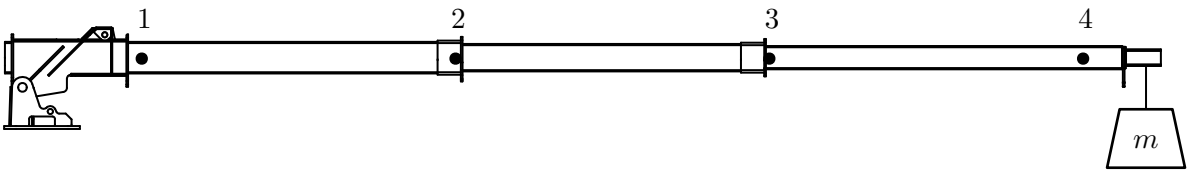


Figure 4.3: Test setup for flexibility measurement.

to capture the boom's transverse dynamics. However, these measurements were corrupted and are therefore omitted. The transverse measurements will not be discussed further in this thesis.

The measurements were converted to angles $\psi_{1...4}$, and the applied load was converted into a torque about point 4. All tests for a given boom length were synchronised to the instant where the string was cut. Results for the fully extended boom are presented in Figure 4.4.

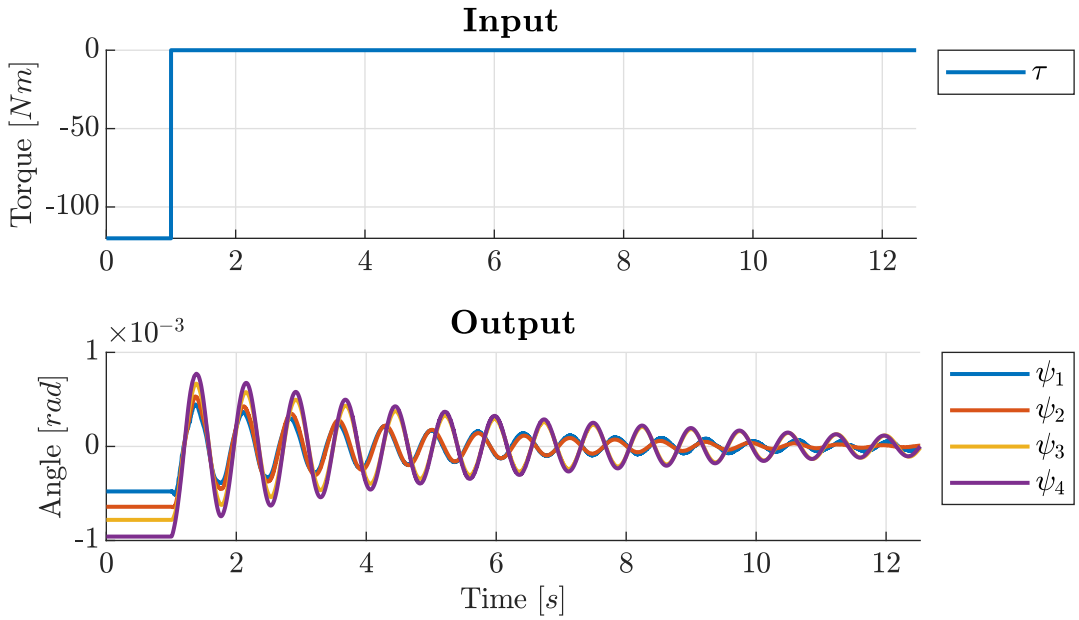


Figure 4.4: Input and output measurements for the flexible model of the fully extended boom.

4.3.2 Grey-Box Model

Grey-box modelling is an identification technique where coefficients of a mathematical structure are fitted using measured data to create a model of the system. The `greyest`-function in MATLAB requires a state-space representation of the system and a data-set of input and output data. The state-space representation is a mathematical model used to describe a system of first-order differential equations (Nise, 2011). The system is represented using two equations; a state equation and an output equation, presented in Eq. (4.2) and Eq. (4.3) respectively. The presented system is time-invariant, meaning that the coefficients are constant. Simulation and response to stimuli are administered in the state equation, where each of the internal states are calculated. The output equation provides an expression of the output in terms of the inputs and internal states.

$$\dot{\mathbf{x}}(t) = \mathbf{A}\mathbf{x}(t) + \mathbf{B}\mathbf{u}(t) \tag{4.2}$$

$$\mathbf{y}(t) = \mathbf{C}\mathbf{x}(t) + \mathbf{D}\mathbf{u}(t) \tag{4.3}$$

Where

$\mathbf{x}$: State vector

$\dot{\mathbf{x}}$: Derivative of state vector

$\mathbf{y}$: Output vector

$\mathbf{A}$: System matrix

$\mathbf{B}$: Input matrix

$\mathbf{C}$: Output matrix

$\mathbf{D}$: Feedforward matrix

$\mathbf{u}$: Input vector

State vector $\mathbf{x}$ contains the internal states of the system, where the system matrix $\mathbf{A}$ describes their mathematical relation. Input matrix $\mathbf{B}$ describes how the inputs excite the system. For the case of a single-input model, it is reduced to a vector. The output matrix $\mathbf{C}$ characterises relations between internal states and system outputs. Similarly to the input matrix, it is reduced to a vector for a single-output model.

For the physical interpretation of the deflection, a serialised four-body system was designed, illustrated in Figure 4.5. Note that gravity is not explicitly included in the model, as a simplification. The effects of gravity are instead identified as part of the inertia. The boom is considered as four rigid bodies interconnected via revolute joints, rotational springs and viscous dampers, composing a single-input, multiple-output (SIMO) system. The four bodies are assumed to align with the points in Figure 4.3, where the mechanical rotational reference is connected to θ_2.

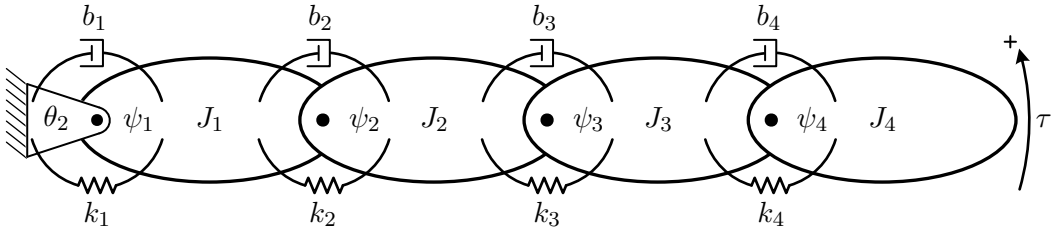


Figure 4.5: Simplified physical deflection model. Positive rotation anticlockwise. Note that θ_2 is considered as the ground reference and that gravity is identified as part of the inertia.

Where

J_i : Moment of inertia for body i $[kgm^2]$

b_i : Viscous damping for damper i $[Nms/rad]$

k_i : Spring stiffness for spring i $[Nm/rad]$

ψ_i : Angle of rotation for body i with respect to inertial frame $[rad]$

τ : External torque imposed on body 4 $[Nm]$

From this model, Newton's second law of motion for rotation, presented in Eq. (4.4), is applied to derive a set of differential equations:

$$\mathbf{J}_n\ddot{\boldsymbol{\psi}}_n = \sum \boldsymbol{\tau} \ [Nm] \tag{4.4}$$

$$J_1\ddot{\psi}_1 = k_1(\theta_2 - \psi_1) + b_1(\dot{\theta}_2 - \dot{\psi}_1) + k_2(\psi_2 - \psi_1) + b_2(\dot{\psi}_2 - \dot{\psi}_1) \tag{4.5}$$

$$J_2\ddot{\psi}_2 = k_2(\psi_1 - \psi_2) + b_2(\dot{\psi}_1 - \dot{\psi}_2) + k_3(\psi_3 - \psi_2) + b_3(\dot{\psi}_3 - \dot{\psi}_2) \tag{4.6}$$

$$J_3\ddot{\psi}_3 = k_3(\psi_2 - \psi_3) + b_3(\dot{\psi}_2 - \dot{\psi}_3) + k_4(\psi_4 - \psi_3) + b_4(\dot{\psi}_4 - \dot{\psi}_3) \tag{4.7}$$

$$J_4\ddot{\psi}_4 = k_4(\psi_3 - \psi_4) + b_4(\dot{\psi}_3 - \dot{\psi}_4) + \tau \tag{4.8}$$

A linear time-invariant (LTI) state-space model is constructed, adopting the angular positions and their time derivatives as internal states together with the angular position for θ_2. The state vector is assumed:

$$\mathbf{x} = [\psi_1,\ \dot{\psi_1},\ \psi_2,\ \dot{\psi_2},\ \psi_3,\ \dot{\psi_3},\ \psi_4,\ \dot{\psi_4},\ \theta_2] \tag{4.9}$$

Whereas the system matrix becomes:

$$\mathbf{A} = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ -\frac{k_1+k_2}{J_1} & -\frac{b_1+b_2}{J_1} & \frac{k_2}{J_1} & \frac{b_2}{J_1} & 0 & 0 & 0 & 0 & \frac{k_1}{J_1} \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ \frac{k_2}{J_2} & \frac{b_2}{J_2} & -\frac{k_2+k_3}{J_2} & -\frac{b_2+b_3}{J_2} & \frac{k_3}{J_2} & \frac{b_3}{J_2} & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & \frac{k_3}{J_3} & \frac{b_3}{J_3} & -\frac{k_3+k_4}{J_3} & -\frac{b_3+b_4}{J_3} & \frac{k_4}{J_3} & \frac{b_4}{J_3} & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & \frac{k_4}{J_4} & \frac{b_4}{J_4} & -\frac{k_4}{J_4} & -\frac{b_4}{J_4} & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \tag{4.10}$$

Depending on the context, the system is subjected to different inputs. The validation data incorporates a torque input at the tip of the boom. However, when simulating, the system is excited by a proportional hydraulic valve operating with overhead, rotating θ_2. Consequently, it is convenient to use $\dot{\theta_2}$ as input for simulation purposes. In either case, the model has a single input, and $\mathbf{B}$ becomes a vector where each index defines how the input excites the corresponding system state. The input configurations for validation and simulation are presented in Eq. (4.11) and Eq. (4.12) respectively.

$$\mathbf{B}_1 = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & -\frac{1}{J_4} & 0 \end{bmatrix}^T \tag{4.11}$$

$$\mathbf{B}_2 = \begin{bmatrix} 0 & \frac{b_2}{J_1} & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}^T \tag{4.12}$$

Correspondingly, the output matrix is context dependent. The number of rows coincides with the number of outputs, while the columns specify linear relations between states and outputs. The validation data includes outputs for all the angular positions(ψ_1, ψ_2, ψ_3, ψ_4), establishing the output matrix presented in Eq. (4.13). In the simulation, only ψ_4 is considered, as presented in Eq. (4.14).

$$\mathbf{C}_1 = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \end{bmatrix} \tag{4.13}$$

$$\mathbf{C}_2 = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \end{bmatrix} \tag{4.14}$$

Applying the grey-box method to this model, makes the system identification a matter of fitting values for $(J_{1\ldots4}, b_{1\ldots4}, k_{1\ldots4})$. However, the deflection is highly dependent on the extension of the boom. Validation data were collected for four different boom lengths $d_3 = (7,\ 9.7,\ 12.3,\ 15)\,[m]$, imposing a different set of coefficients for each case. The selected approach is to create the system matrix $\mathbf{A}$ for each length and interpolate the coefficients, creating a variable system matrix as a function of boom length d_3. The output is not directly affected by any inputs and therefore, the feedforward matrix is empty in both cases. The state- and output equations are adapted:

$$\dot{\mathbf{x}}(t) = \mathbf{A}(d_3)\mathbf{x}(t) + \mathbf{B}\mathbf{u}(t) \tag{4.15}$$

$$\mathbf{y}(t) = \mathbf{C}\mathbf{x}(t) \tag{4.16}$$

4.3.3 Verification of State-Space Model

To verify that the proposed state-space system behaves as expected, the impulse response was simulated using θ_2 as input. Inertia, damping and stiffness were all approximated as an informed guess. The MATLAB implementation is found in Appendix E.1.4.

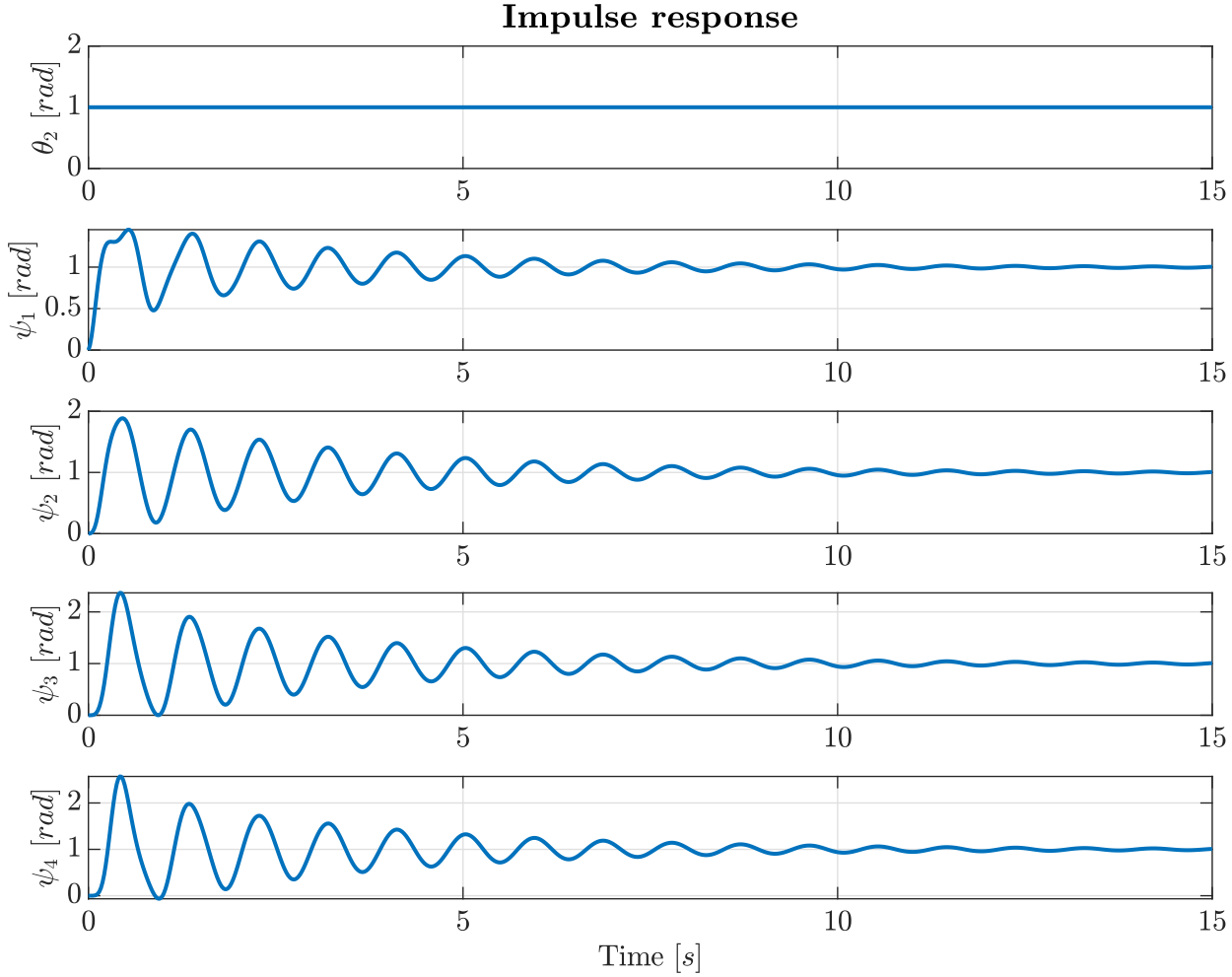


Figure 4.6: Impulse response showing all position states, with $\dot{\theta}_2$ as input.

The response uncovers that the model behaves as expected, where the excitation propagates from the mechanical joint towards the tip of the boom. In addition, it is observed that the ψ_1-state is affected by the first peak in ψ_2, meaning that the states influence each other in both directions.

4.3.4 Black-Box Model

System identification by black-box modelling assumes that the structure of the system is unknown. The model is constructed purely by evaluating exciting inputs and the corresponding output reactions. This method is useful if the system dynamics are problematic to model analytically. Given its flexible nature and preferred disturbance dynamics handling, the autoregressive-moving-average model with exogenous inputs (ARMAX) is popular for black-box identification (National Instruments, 2018). As a benchmark to test if a mathematical model is obtainable for the acquired validation data, the ARMAX model was employed with varying number of coefficients.

4.3.5 Manual Calculations

As a control, a manual calculation was undertaken using traditional vibration analysis. The calculations interpret the boom as a single spring-damper-inertia system where the tests are free underdamped vibrations with the static deflection as initial position and no initial velocity. All of the following formulas originate from (Rao, 1995). Starting with the static deflection to determine

spring stiffness, the torque is calculated considering the hanging load as force and boom length as the effective arm.

$$k_{eq} = \frac{\tau_{stat}}{\psi_{4,stat}} \quad [Nm/rad] \tag{4.17}$$

Where

k_{eq} : Equivalent spring stiffness $[Nm/rad]$
τ_{stat} : Static torque $[Nm]$
$\psi_{4,stat}$: Static deflection $[rad]$

Damping is approximated by investigating at which rate the oscillations decrease. The logarithmic decrement δ is calculated by assessing the change of amplitude between two adjacent peaks.

$$\delta_i = \ln\left(\frac{A_i}{A_{i+1}}\right) \quad [-] \tag{4.18}$$

Where

δ_i : Logarithmic decrement of peak pair $[-]$
A_i : Amplitude of peak i $[deg]$
A_{i+1} : Amplitude of peak $i+1$ $[deg]$

Decrement is calculated for all observed peak pairs and averaged. Damping ratio, ζ is a distinction of the logarithmic decrement and is found solving the following formula

$$\zeta = \frac{\delta}{\sqrt{(2\pi)^2 + \delta^2}} \quad [-] \tag{4.19}$$

which is useful, in combination with the damped frequency, ω_d to find the natural frequency, ω_n. Damped frequency is approximated by measuring the period between adjacent peaks in the validation data and converting to frequency:

$$\omega_n = \frac{\omega_d}{\sqrt{1-\zeta^2}} = \frac{2\pi}{\overline{P_d} \cdot \sqrt{1-\zeta^2}} \quad [rad/s] \tag{4.20}$$

Where

ω_n : Natural frequency $[rad/s]$
ω_d : Damped frequency $[rad/s]$
$\overline{P_d}$: Average damped period $[s]$

From which the inertia of the boom is derived:

$$J_{eq} = \frac{k_{eq}}{\omega_n^2} \quad [kg\,m^2] \tag{4.21}$$

And finally, the damping b_{eq} is calculated:

$$b_{eq} = \zeta \cdot 2J_{eq} \cdot \omega_n \quad [Ns/m] \tag{4.22}$$

Simulating using the differential equation:

$$\ddot{\psi}_4 = -\frac{1}{J_{eq}} \cdot (k_{eq}\psi_4 + b_{eq}\dot{\psi}_4) \quad [rad/s^2] \tag{4.23}$$

The calculations were completed using MATLAB, where the source code is available in Appendix E.1.3. For real-time simulation, the estimated properties can be implemented in state-space representation using a simplified form of the system matrix in Eq. (4.10). The traditional vibration analysis only exposes the first mode of vibration. However, the properties are a direct estimation for the physical properties of the system.

Chapter 5

Interface and Control

For this project, it was decided to create a virtual representation of the AMV 4200H to be used for hardware-in-the-loop simulation. Using an HIL-model serves several benefits compared to direct implementation and testing on a physical system. Firstly, the physical machine that will be used for final validation testing is still under construction at the time of writing, besides, the physical location of the machine is not at campus. By using a virtual model, construction may continue uninterrupted while the project is carried out in parallel.

Using HIL-simulation allows testing without danger of harm to personnel or equipment. Moreover, it provides a way of developing control strategies early in the design process and allows for rapid synthesis and testing which would otherwise be difficult or impractical. Furthermore, using HIL-simulations allows testing of edge cases that would not be tested on the real system, for example testing the controller to instability.

5.1 Danfoss Plus+1

The programmable logic controller on the AMV 4200H is an electric control unit (ECU) from the Danfoss MC024-series. This device is programmed through function blocks in the proprietary software, Danfoss plus+1. The communication is via CAN bus. The hardware supports CANopen, which is a higher-level protocol built upon the CAN bus standard and is used for in-vehicle communication on the machine. The CANopen protocol standardises communication, device definitions and interfacing between devices and applications from different manufacturers. The top layer of the Danfoss application discussed in the following sections is found in Appendix D.14.

5.1.1 CAN bus

Controller Area Network (CAN) is the industry standard for in-vehicle communication. The standard was developed by Bosch in 1985 (National Instruments, 2019) to reduce wiring and copper use in vehicle communication systems. Since 1985, several higher-level protocols have been developed, among them the CANopen protocol.

By using CAN bus, the devices on the network can communicate over one single bus, significantly reducing the number of input/output (I/O) connections. The network is low cost, low weight and provides all devices with intelligence. Each CAN device has a built-in CAN-controller, making the devices able to read/write and filter messages on the network. This functionality also makes it possible to modify the CAN network with minimal impact. Each CAN bus message has a priority which is determined from the sending ID, called the arbitration ID. If two nodes on the network transmit simultaneously, the message with higher priority will be sent first, and the lower priority message will be postponed. In addition, the CAN bus standard has a built-in error checking mechanism. In each CAN message, there is a field called the Cyclic Redundancy Code. This field determines whether a frame has an error or not. If there is an error, the message is discarded by the whole network, and an error frame can be transmitted to signal the error to the network.

An example of a standard CAN frame with 8-bit data length is presented in Figure 5.1. Note that a CAN message can carry up to 8 bytes of data.

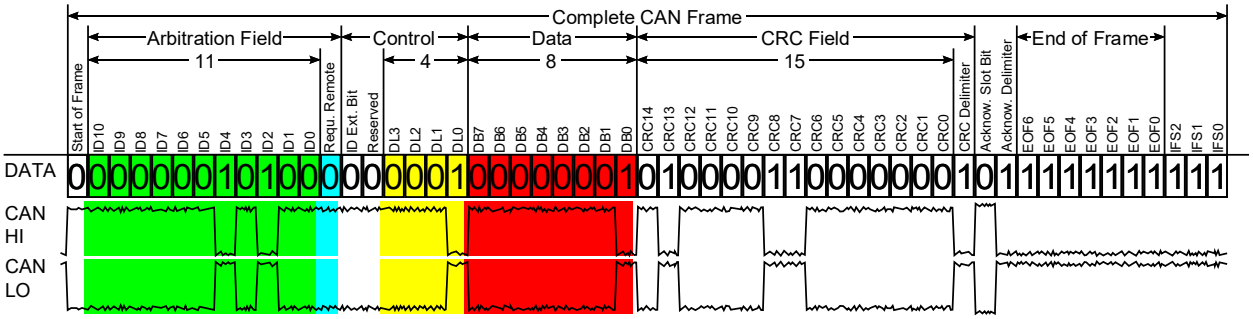


Figure 5.1: CAN-Bus-frame in base format without stuffbits. Used under licence of Wikimedia Commons (Wikimedia Commons, 2014).

5.1.2 CANopen

In this project, the CANopen protocol is not explicitly used, but instead it is ensured that the definitions of messages do not conflict with the CANopen protocol. In CANopen, the arbitration ID is referred to as the Communication Object Identifier (COB-ID). The COB-ID is the 11-bit arbitration field split into two parts. First, a 4-bit function code defining the transmission type of the message. The 7 last bits represents the node-ID. For simple CANopen networks, CAN in Automation (CiA) 301 specifies some predefined message identifiers for Process Data Objects (PDOs):

Table 5.1: Pre-defined connection set for PDOs in CANopen. CiA 301 (CAN in Automation, 2002).

Communication Object	COB-ID(s) hex	Slave Nodes
PDO	0x180 + NodeID	1. Transmit PDO
	0x200 + NodeID	1. Receive PDO
	0x280 + NodeID	2. Transmit PDO
	0x300 + NodeID	2. Receive PDO
	0x380 + NodeID	3. Transmit PDO
	0x400 + NodeID	3. Receive PDO
	0x480 + NodeID	4. Transmit PDO
	0x500 + NodeID	4. Receive PDO

A Process Data Object (PDO) can represent any process variable that changes over time, and the PDO protocol is used to process and distribute the real-time data among nodes. For this thesis, only the three first PDO transmits are used. The PDOs are used for transmitting sensor values and status flags from the ECU to the human-machine interface HMI PC, and valve commands from the ECU to the real-time target.

5.2 Setup

To achieve a working HIL-simulation and to test the system, HMI interface and computer application, communication must be established between the real-time target, ECU and the HMI PC. The Danfoss ECU has two CAN bus-channels, channel one is used at the maximum baud rate of 1 Mbit/s, to give as much overhead as possible for data transfer between the ECU and HMI PC. The second CAN bus-channel is used for inter-vehicular communication as well as sending data to the physical system, and runs at 250 kbit/s. The physical system is in this case simulated by

the real-time target Speedgoat Baseline-S (Speedgoat GmbH, 2019). The real-time target runs Simulink Real-Time, which makes it possible to compile models directly from MATLAB/Simulink. The problem, however, is that the Speedgoat does not support CAN bus without an additional and expensive CAN I/O module. It does, however, support communication over UDP. The full communication layout is presented in Figure 5.2.

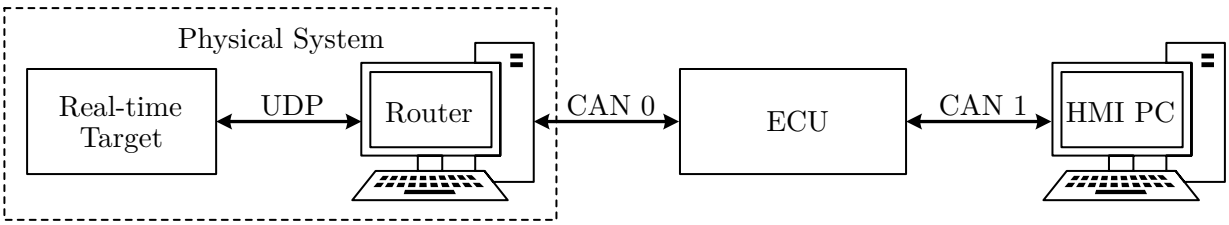


Figure 5.2: Communication layout between Real-time target and HMI PC.

An extra computer is added to the physical system. In practice, this computer only functions as a router. The PC runs Simulink with MATLAB's Vehicle Network Toolbox and supports CAN bus communication. The PC receives incoming CAN bus signals over CAN 0 from the ECU and forwards the data over UDP to the Speedgoat, the Speedgoat replies over UDP which is subsequently translated back to CAN bus, and forwarded back to the ECU. UDP provides a throughput tenfold higher than CAN bus, capable of speeds up to 1 Gbit/s. However, by using UDP, one cannot guarantee transmission without packet loss. For the CAN bus network, packets may get lost if an error occurs, which in turn depends highly on the signal to noise ratio, length of cables and bus load. For this particular setup, conditions are ideal; the UDP Ethernet cable and CAN bus connections have lengths of $\approx 1\ m$, there are few nodes in the experimental network and the bus loads are relatively low; not higher than 3 %. In addition, all reference signals have high resolution and should function even in the case of packet loss.

5.3 Communication

There are four nodes in the communication network, as shown in Figure 5.2. The actual sensors on the machine provide a 16-bit resolution when used as single turn encoder (Pepperl+Fuchs, 2018). All sensor values and joint space references are therefore created with 16-bit precision. Since one CAN frame can carry up 8-bytes of data, one frame can hold up to four sensor values. Joint space references and sensor values therefore need two PDOs. Below, a short explanation is given for each send/receive operation:

- **HMI PC to ECU**: PC sends joint space reference over CAN 1, using PDOs 1 and 2, and the node ID of the PC is 0x2. Resulting in COB-IDs 0x182, 0x282 respectively. The implementation is found in the script *main.py* in Appendix E.2.4 under the module *CANInterface*.
- **ECU to HMI PC**: The ECU relays the sensor values from the RT-target over CAN 1 to the HMI PC to update the nozzle position. In addition, it also sends the value of a joystick button used for collecting knot points. The ECU sends on PDO 1,2,3, and the node ID of the ECU is 0x3. Therefore, the HMI PC listens for messages with COB-ID 0x183 and 0x283, representing the sensor values, as well as COB-ID 0x383 for the joystick button. The implementation of packing and sending the data in Danfoss Plus+1 can be found in Appendix D.9 and D.8, respectively.
- **ECU to RT-target**: The ECU sends valve commands over CAN 0 to the RT-target. The signals originate either directly from the joysticks or the control system depending on a status flag indicating manual or regulating mode. The data is sent with COB-IDs 0x183 and 0x283. The packing and sending of data to the RT-target can be found in D.10 and D.11, respectively.
- **RT-target to ECU**: The RT-target sends the simulated sensor values back to the ECU over CAN 0. The node ID of the RT-target is 4, and PDO 1,2 are used. Thus the ECU listens for

COB-IDs 0x184 and 0x284. The implementation for receiving data from the RT-target can be found in Appendix D.7.

- **Router PC**: All information to and from the RT-target is translated and forwarded by the Routing PC. CAN bus messages are translated to UDP and vice versa. The Routing PC block-diagrams are found in Appendix D.2.

5.4 Control

For individual joint control of the AMV 4200H, five proportional-integral (PI)-controllers were proposed. The PI-controller is a robust and reliable variant of the three-term controller, the PID-controller. In the industry, the Derivative-term is often omitted, because the term is sensitive to high-frequency noise due to the derivative action on the error signal. The D-term may e.g. become a problem in applications prone to vibrations and other disturbances. Fast changes in setpoint or disturbances can cause large outputs from the D-term, which in turn may cause unwanted effects and lead to instability. In industrial settings the PID-controllers constitutes more than 95% of all control loops, and most are PI-controllers (Åström and Hägglund, 2006). The PI-controller may be slower than a full three term controller but typically provides better steady-state stability in the presence of noise. The drawback to the I-term is a phenomenon called integral windup, which refers to the integral of the error accumulating faster than the controller can compensate. In such cases, it is common to reset or deactivate the I-term. Below, a description of the proposed controller is given.

5.4.1 Deadband Compensation

Two types of deadband compensation are used in this thesis, namely deadband tolerance and output deadband. The deadband tolerance is integrated in the Danfoss PI-controller-block, and operates on the control error such that the output valve command

$$u_{out} = 0 \qquad \text{if } |e(t)| \leq \text{Tol} \tag{5.1}$$

where $e(t)$ is the current error and Tol is the tolerance at which the controller output is set to zero. The deadband tolerance prevents "hunting" of the setpoint by turning off the controller when the machine is "close enough". This provides smoother operation and helps reduce jitter and integral windup.

The output deadband, on the other hand, prevents unnecessary wear on the valves by avoiding repeated activation-deactivation cycles when the valve commands are less than required for achieving response in the mechanical system. As discussed in Section 4.2, dead zone dynamics has been included in the HIL-setup to simulate the deadband in the valves. Therefore a compensator is made to compensate for the effect of the deadband. The compensator works by evaluating the incoming valve signals, such that the output

$$u_{out} = \begin{cases} \text{DB}^+ + (1 - \text{DB}^+) \cdot u_{in} & \text{if } u_{in} \geq \text{DB}^+ \\ 0 & \text{if } \text{DB}^- < u_{in} < \text{DB}^+ \\ \text{DB}^- + (1 + \text{DB}^-) \cdot u_{in} & \text{if } u_{in} \leq \text{DB}^- \end{cases} \tag{5.2}$$

where $(\text{DB}^+, \text{DB}^-)$ represents the positive and negative deadband values, respectively. The implementation of the output deadband compensator in Danfoss plus+1 can be found in Appendix D.13.

5.4.2 Transfer Function

The chosen controller type is the PI-controller. This controller has the mathematical representation

$$u(t) = K_p \left(e(t) + \frac{1}{T_i} \int_0^t e(\tau)\, d\tau\right) \tag{5.3}$$

where

$u(t)$: Control signal
$e(t)$: Current error
K_p : Proportional gain
T_i : Integration time $[s]$

In the Laplace domain, the the transfer function of the PI-controller is as follows

$$G_c(s) = K_p \left(1 + \frac{1}{T_i\, s}\right) \tag{5.4}$$

where $G_c(s)$ is the transfer function of the controller. The transfer function of the simplified plant model is found by dividing the output and input of the HIL-model

$$G_p(s) = \frac{K_{vel}}{s} \tag{5.5}$$

where $G_p(s)$ is the plant transfer function and K_{vel} is the plant open-loop gain that scales the valve commands u_i to joint velocity.

Since the HIL-model is a type 1 system, it is guaranteed that there will be no overshoot, and hence proportional control is sufficient. Therefore, the I-term is turned off in the HIL-simulation since the introduction of the integral-term will introduce overshoot. For the real system, dynamics will be of a higher order, and the integral action will contribute to the elimination of steady-state error.

5.4.3 Controller

An illustration of the proposed controller is presented in Figure 5.3. The implementation in Danfoss Plus+1 is found in Appendix D.12.

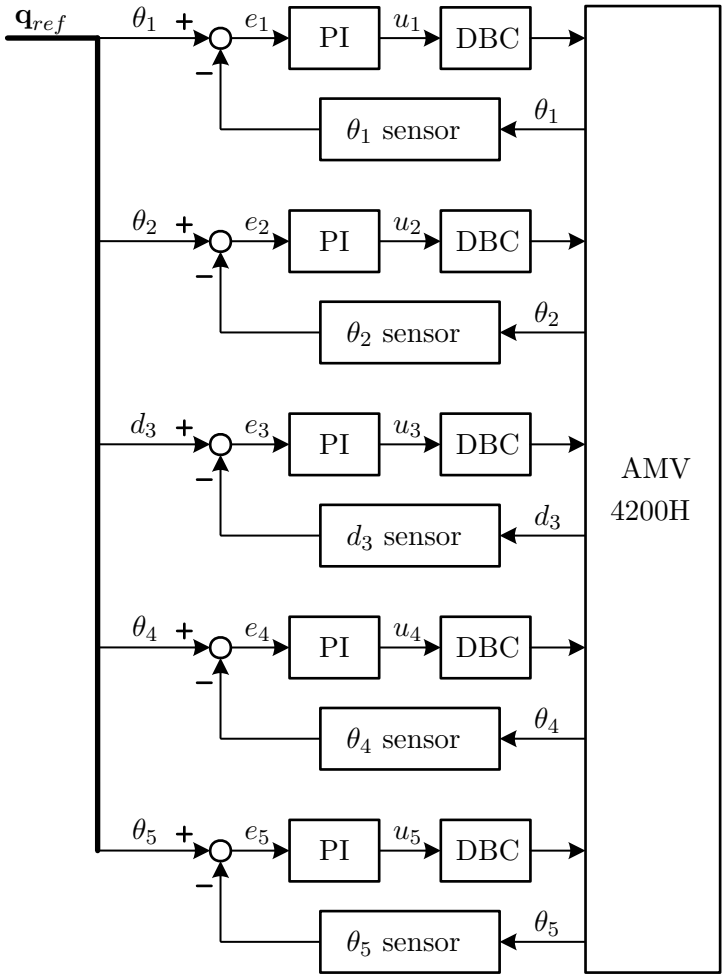


Figure 5.3: Control structure; five proportional-integral (PI) controllers with deadband compensation (DBC). Note that the I-term is turned off in the HIL-simulation as discussed in Section 5.4.2.

5.5 PC Program

The developed software is created in Python 3.7 and consists of three main modules and a main script importing the three modules. The first module contains all the kinematic equations and the inverse kinematic algorithm discussed in Chapter 2. This module is called kin.py and can be found in Appendix E.2.1. Module two contains the functionality for parameterisation and generating of the tunnel surface, as well as mapping of the planar trajectory onto the tunnel surface. The material for module two is covered in Chapter 3, under Sections 3.1 and 3.2. The module is called surf_trans.py and can be found in Appendix E.2.3. The third module handles the correction of user input parameters and the generating of the planar trajectory discussed in Chapter 3 under Section 3.3. This module is called pattern_generator.py and can be found in Appendix E.2.2. The main script imports the previously presented modules and runs the backend of the user interface. The main script contains instructions for all interaction between the user and interface, as well as CAN bus-communication, the inverse kinematics and Cartesian references. The script main.py is found in Appendix E.2.4.

5.5.1 HMI

The Human-Machine Interface was developed using a combination of Python and the Qt-framework. The frontend of the HMI was created using Qt-designer, which is a tool for designing and building graphical user interfaces. Whereas the backend was created using Python 3.7 and the PyQt5 framework. The appearance of the user interface is presented in Figure 5.4 and is made to emulate the existing style theme used in the other interfaces used by AMV. Below, an explanation is given for each of the different buttons, fields and their functionality:

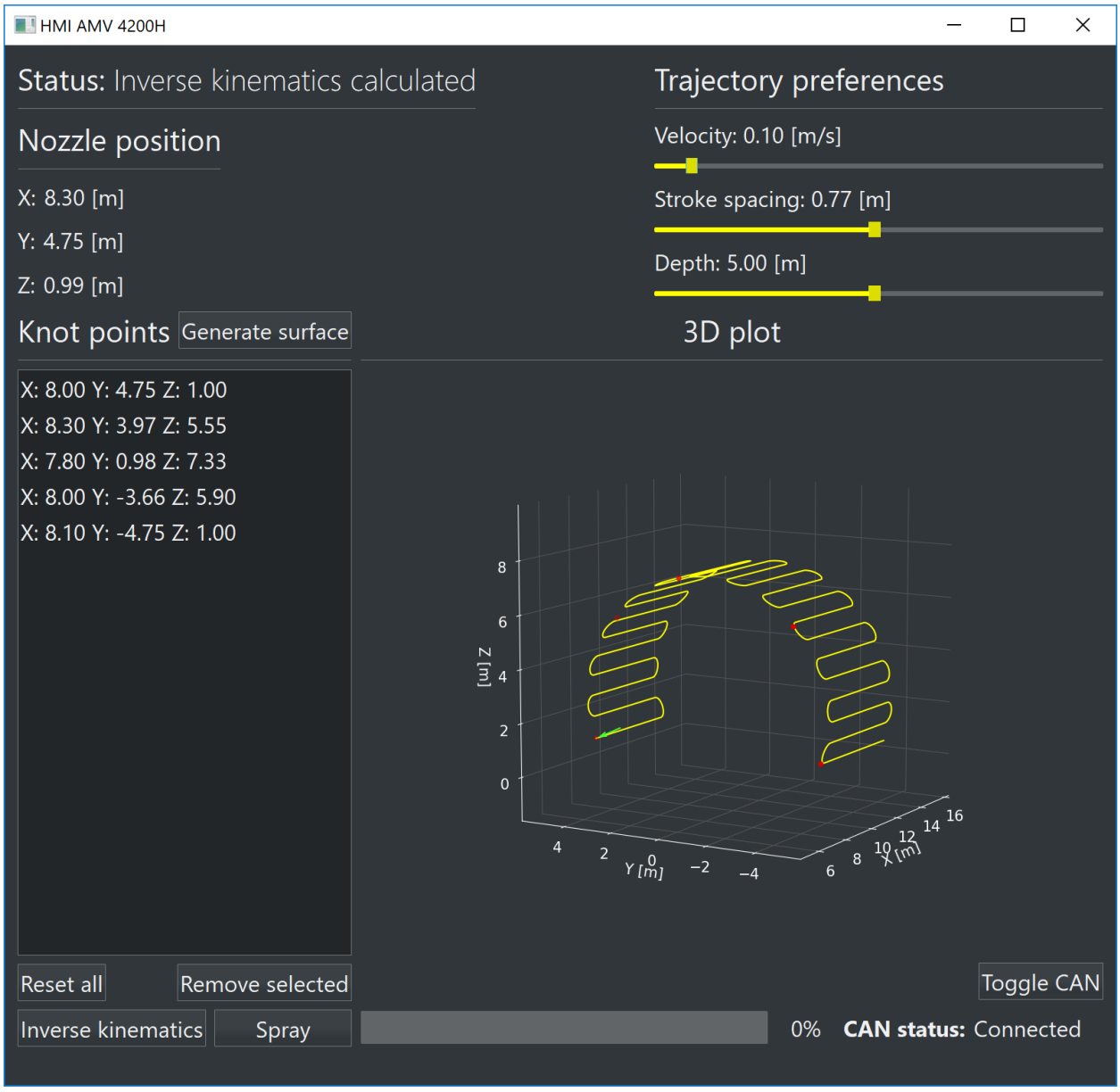


Figure 5.4: Human-Machine Interface for application of shotcrete.

- **Status**: Field indicating the current status of the program. Grey text indicates either "idle-mode" or holds the name of the previously completed process. Yellow text indicates ongoing processes and red text indicates an error.
- **Trajectory preferences**: Sliders controlling the parameters; depth, stroke spacing and velocity of the trajectory.
- **Nozzle position**: Indicates the Cartesian position of the end effector and is updated in real-time based on sensor values from the ECU.
- **Knot points**: List containing a minimum of five knot points for approximation of the tunnel surface.

- **Generate surface**: Creates the nozzle trajectory with the parameters specified in "Trajectory preferences". It calls a multiprocessing method to calculate the spraying trajectory.
- **3D plot**: A matplotlib-widget that plots the trajectory and position of the nozzle, indicated by a yellow line and a green vector, respectively. The trajectory is updated each time a new surface is generated, and the nozzle position is updated in real-time based on sensor values from the ECU. The plotting method is inspired by Tosi (2009).
- **Reset all**: Removes all existing data.
- **Remove selected**: Removes individual knot points.
- **Toggle CAN**: Connect or disconnect CAN bus.
- **Inverse kinematics**: Calculates joint space references. The button initiates a multiprocessing pool, taking advantage of the maximum number of processor threads available on the given computer to reduce computation time.
- **Spray**: Initiates spraying procedure. This operation will not start until the operator provides confirmation from the joysticks. Once confirmation is provided, the joint space reference is transferred over CAN bus to the ECU for the execution of the spraying procedure. The Spray-button changes to a Pause button once spraying is initiated.
- **Progress bar**: Indicates working progress for large tasks. Used during calculation of the inverse kinematics and the spraying procedure.
- **CAN status**: Indicates the status of CAN bus-communication. Red text indicates a CAN bus error.

5.5.2 HMI and Safety Features

There are two main safety features in the developed software. Firstly, to make the program stable, independent of user interaction; it is ensured that any button can be pressed at any time without causing the program to crash. In addition, clear instructions are prompted to the user if improper operations are attempted or if operations are initiated in the wrong order. The second safety feature is related to the actual operation of the machine; before initialisation of the spraying procedure, the operator must be present at the manual controller to initialise spraying. As such, the operator can pause, abort or take manual control over the machine at any time. Pressing the joystick button will pause the machine from spraying, and any movement of the joysticks will immediately give the operator manual control and stop the controller from regulating.

Chapter 6

Results

Before the proposed methods can be used confidently for real-world applications, the system performance should be considered. This chapter presents key results to highlight the benefits and drawbacks of the implementation.

6.1 Kinematics

While the inverse kinematics solver developed in this project provides an analytical solution in terms of the nozzle orientation, it does not place the NCP exactly at the reference position. Therefore, it is of particular interest to evaluate the accuracy and computing time of the solver. A typical scenario was created in the PC program; fitting a surface to a T9.5 tunnel profile, using a depth of 5 m and stroke spacing $ss_{des} = 0.65\ m$ resulting in 16075 points to evaluate. The inverse kinematics was evaluated both in terms of accuracy and computing time.

6.1.1 Precision of Inverse Kinematics

While the NCP is never placed precisely at the reference, it will approach the reference position, increasing the precision for each iteration. Before the proposed inverse kinematic method can be applied, the accuracy should be considered. The aforementioned scenario was solved and the magnitude of position error e was logged for each iteration of each point. The data-set was analysed by evaluating the maximum error, e_{max} after 1, 2, and 3 iterations as well as the average value, $\overline{e}$ and standard deviation, σ_e. The results are presented in Table 6.1. It becomes evident that the solver is within millimetre precision after two iterations and micrometre precision after three iterations. Thus, demonstrating that the inverse kinematic solver can reach a sufficient working accuracy for shotcrete application.

Table 6.1: Maximum, average, and standard deviation for inverse kinematics error magnitude.

Iteration	$\boldsymbol{e_{max}}$ $[m]$	$\overline{\boldsymbol{e}}$ $[m]$	$\boldsymbol{\sigma_e}$ $[m]$
1	0.711	0.680	0.015
2	6.00 e-3	4.184 e-3	0.746 e-3
3	43.389 e-6	16.565 e-6	8.181 e-6

Note that at the first iteration, the error is equal to the distance from the wrist frame, i.e. θ_4, to the NCP and that the magnitude depends on the current angle of θ_5.

6.1.2 Computing Time for Inverse Kinematics

In addition to increasing the spraying uniformity, the automation is intended to save time over a manual operation. Therefore, the computing time is decisive in determining the feasibility of the inverse kinematics solver. For the given scenario, the inverse kinematics was evaluated in terms of computing time. The code was run in the following environment:

Table 6.2: Test environment for the inverse kinematics performance test.

	Designation	**Specification**
Language	Python	3.7 (32-bit)
CPU	Intel i7 4810MQ	4x2.8 GHz
OS	Windows 10	64-bit

Where the solver was run using multiprocessing. This escapes Python's global interpreter lock (GIL) and runs the code in parallel, maximising all the available CPU threads. Source code is available in Appendix E.2.4. The points were processed in 11.06 s, implying an average time of 0.688 ms processing time per set of five joint angles.

6.2 Trajectory Planning

A well-planned trajectory lays the foundation for a good overall result. Both in terms of spraying as the operator intends and taking into account the attributes of the machine. The AMV 4200H is a long slender machine, susceptible to resonance and vibrations. Furthermore, the stroke spacing is adapted to fit the surface model, meaning that the resulting spacing is not necessarily the same as requested by the operator. These two aspects are investigated further in this section.

6.2.1 Corrected Stroke Spacing

Adapting the stroke spacing to achieve a common divisor with the estimated arc length causes a difference between the desired spacing requested by the operator and the actual spacing used for the trajectory. The mismatch was investigated to evaluate its extent by evaluating the ratio Δ_{ss}, and recalling Eq. (3.13) and Eq. (3.14):

$$\Delta_{ss}(ss_{des}) = \frac{ss_{des} - ss_{cor}}{ss_{des}} = 1 - \frac{L_{arc}}{\text{round}\left(\frac{L_{arc}}{ss_{des}}\right) \cdot ss_{des}} \; [-] \tag{6.1}$$

The equation indicates that the highest deviations occur in the smallest tunnels. Therefore, the formula was evaluated using the arc length of a T5.5 tunnel profile, $L_{arc} = 14.9\,m$. The stroke spacing was evaluated over the selectable range in the HMI. The result is presented in Figure 6.1.

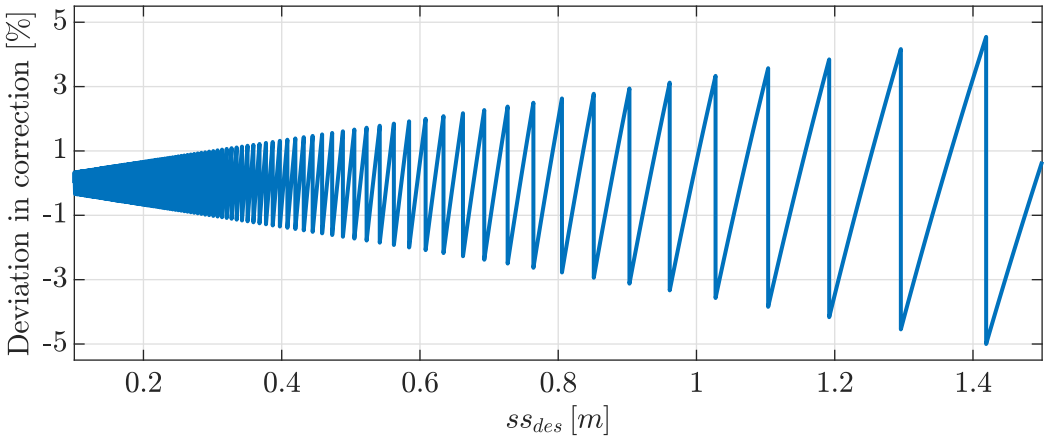
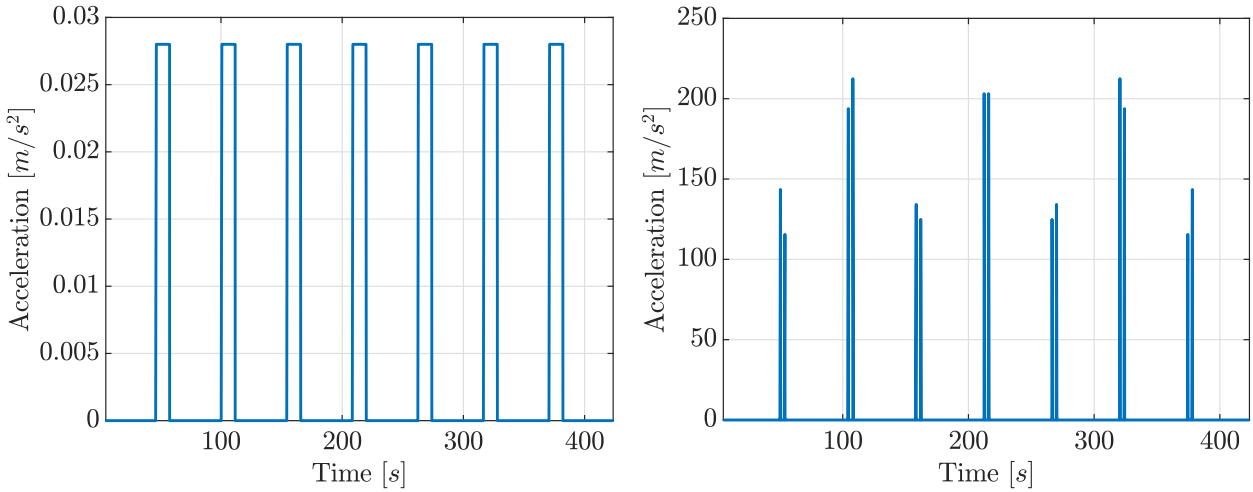


Figure 6.1: Deviation of corrected stroke spacing as a percentage of the desired stroke spacing for the T5.5 tunnel profile.

The highest deviation from requested stroke spacing ss_{des} is 5 %, meaning that the corrected parameters will be close to the operator's request; within 5 %, depending on the tunnel shape.

6.2.2 Acceleration

In Figure 3.7, both a traditional square path and the proposed semicircular-transition path are illustrated. Both paths were evaluated at a constant trajectory velocity $v_e = 0.1\,m/s$, where the planar acceleration magnitudes are presented in Figure 6.2. The calculations were computed numerically with $dt = 1\,ms$. The solution without rounded transitions has acceleration characteristics resembling impulse functions in the corners. Smaller values for dt were tested, where the impulses approach infinity as dt approaches zero. Acceleration in the semicircular transitions is constant, which agrees with the formula for centripetal acceleration $a = v^2/r$. Therefore, the acceleration is bounded, indicating that the trajectory is physically possible to follow.



(a) Semicircular transitions.

(b) Without rounded transitions.

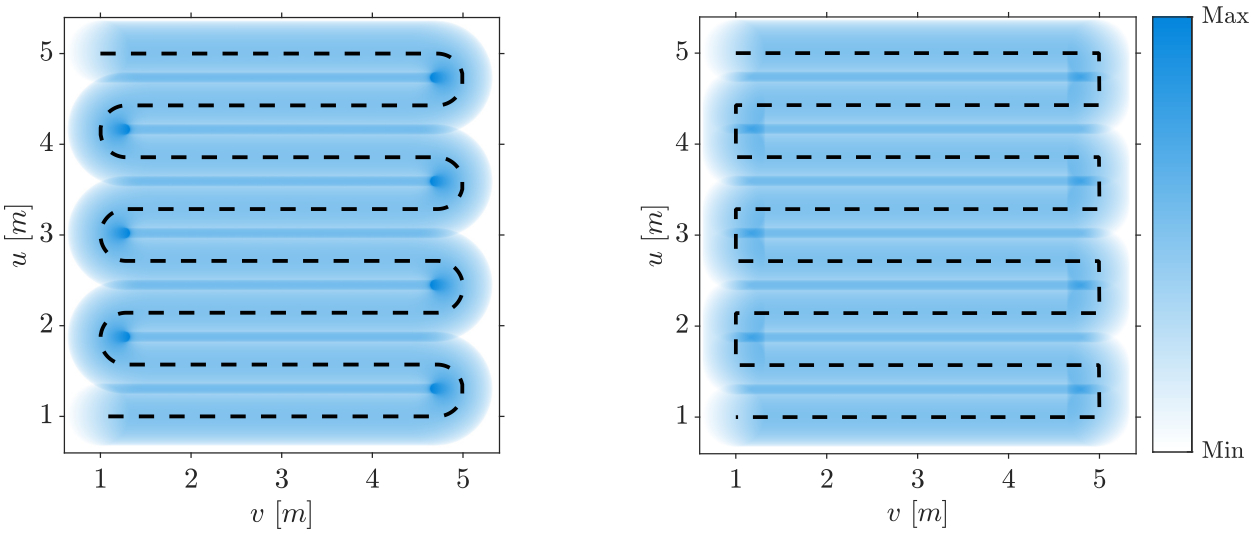
Figure 6.2: Comparison of acceleration in the planar trajectory, using square and semicircular transitions.

On the contrary, the third derivative is not bounded in either case. This is because the transition from horizontal stroke to circular motion requires an instant change from zero to constant centripetal acceleration. In robotics, it is common to consider the third derivative of position, referred to as jerk. Jerk is a measure of the rate of change of acceleration and has the SI-unit m/s^3. The phenomenon is often perceived as smoothness of motion during acceleration, e.g. the difference between a smooth push of the brake pedal in a car and an abrupt push to the pedal. Since acceleration determines the force and forces are the cause of vibrations, jerk in the trajectory can be considered as a measure of the induced disturbances. It is especially important to consider jerk in situations where the equipment is prone to resonance and vibrations, as is the case for the AMV 4200H. For these reasons, it is desirable to minimise jerk to achieve smooth motion; minimising strain and disturbances in the machinery.

6.2.3 Shotcrete Distribution

Under ideal circumstances, the shotcrete is distributed as a uniform layer over the working surface. When the volumetric flow and nozzle trajectory velocity is constant, the distribution is dictated only by the spraying pattern. The distribution characteristics for semicircular and square transitions were studied by simulating shotcrete deposits along the trajectory. The simulation assumes that the distribution within the cone described in Section 2.1.3 is uniform. In reality, it is affected by the spread angle of the jet as well as the nozzle nutation rate ω. The simulation results are illustrated in Figure 6.3. Dashed lines indicate the trajectories.

Both patterns are identical except for transitions between the strokes. The figure indicates that the semicircular trajectory seems more susceptible to material buildup in the centre of rotation. Furthermore, the semicircular trajectory boundary fluctuates more than the square trajectory. Consequently, the footprint is less square-like.



(a) Semicircular transitions.

(b) Without rounded transitions.

Figure 6.3: Comparison of shotcrete distribution for square and rounded transitions in the planar trajectory.

It should be noted that the shotcrete deposition in Figure 6.3b is not entirely representative of a real scenario. This is because the deposition is plotted with a constant velocity through the 90-degree corners. In reality, the actual machine must either slow down or overshoot to complete these corners, and deposition will deviate from the simulated deposition.

6.3 System Modelling

Results for the system modelling are presented in two categories; estimation using the MATLAB System Identification Toolbox, incorporating grey-box and black-box results, as well as manual calculations as a control. As described in Section 4.3.1, four different boom lengths were evaluated, ranging from fully retracted to fully extended, i.e. for $d_3 = [7.012, 15.012]\ m$. Normalised root mean square error (NRMSE) is used as the key performance index. For the grey-box and black-box approaches, the validation data was packed in a timeseries in MATLAB and fed into the estimation function.

6.3.1 Grey-box

Employing the `greyest`-function in MATLAB did not yield any useful results as the algorithm did not converge towards any particular solution. The implications are further assessed in the discussion.

6.3.2 Black-box

Using the `armax`-function for fitting, a model was identified for all boom lengths. Afterwards, the identified model was compared to the validation data using the `compare`-function. The best fit was found using 5th-degree polynomials. Results for all states are separated into individual figures by boom length in Figure 6.4 through 6.7.

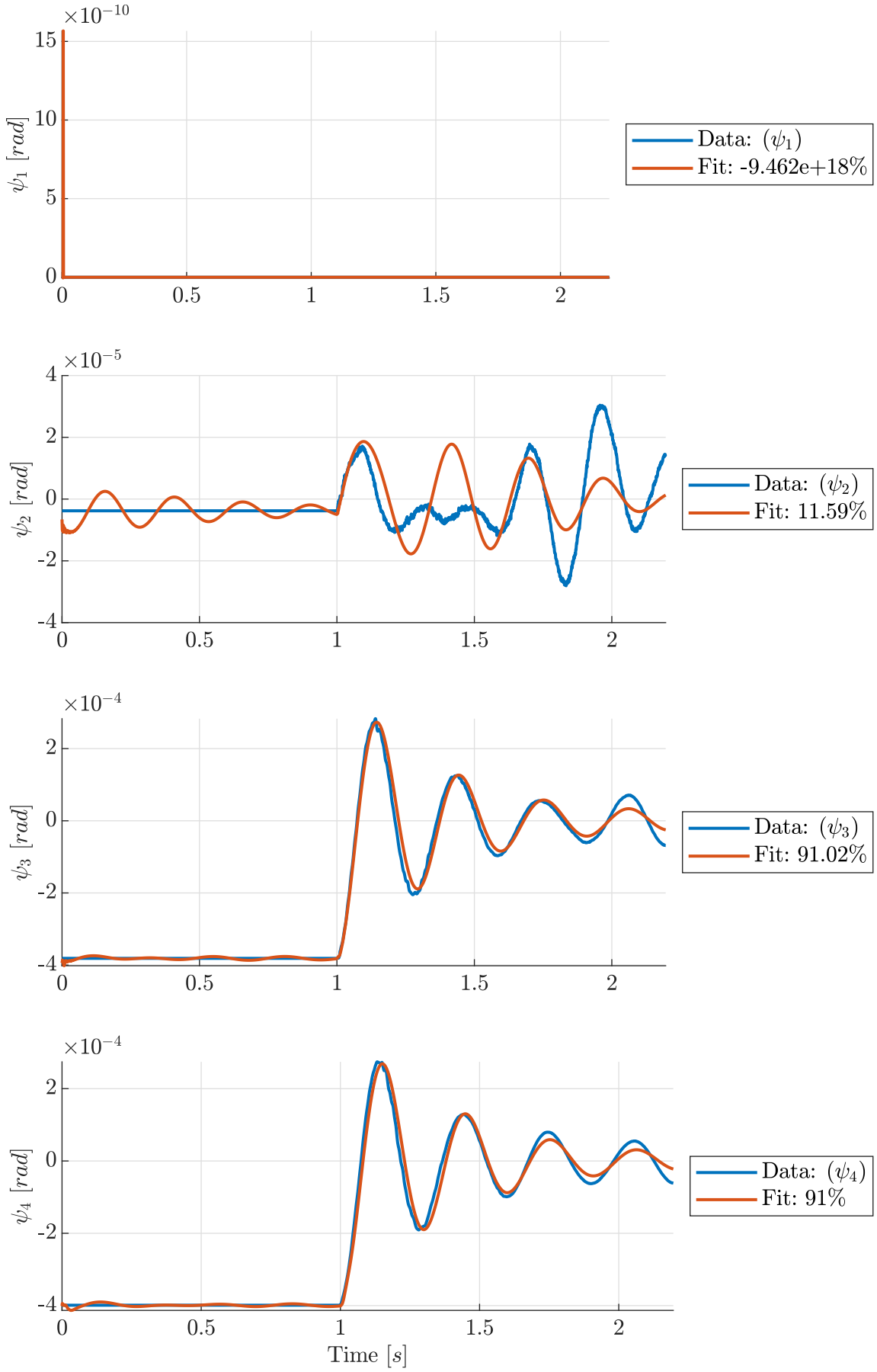


Figure 6.4: Comparison between validation data from the fully retracted pose and associated ARMAX fit for SIMO-model.

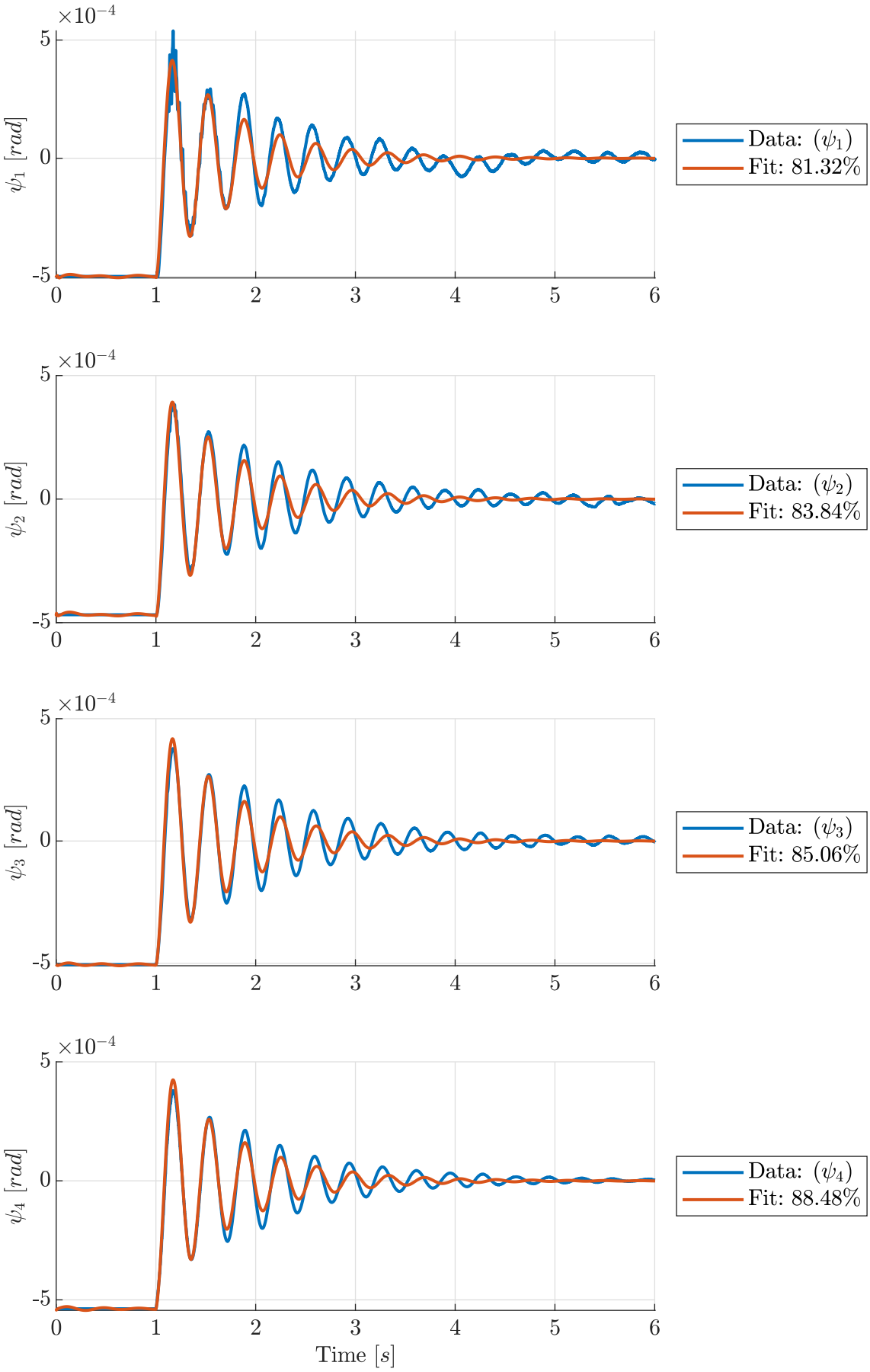


Figure 6.5: Comparison between validation data from the second most retracted pose and associated ARMAX fit for SIMO-model.

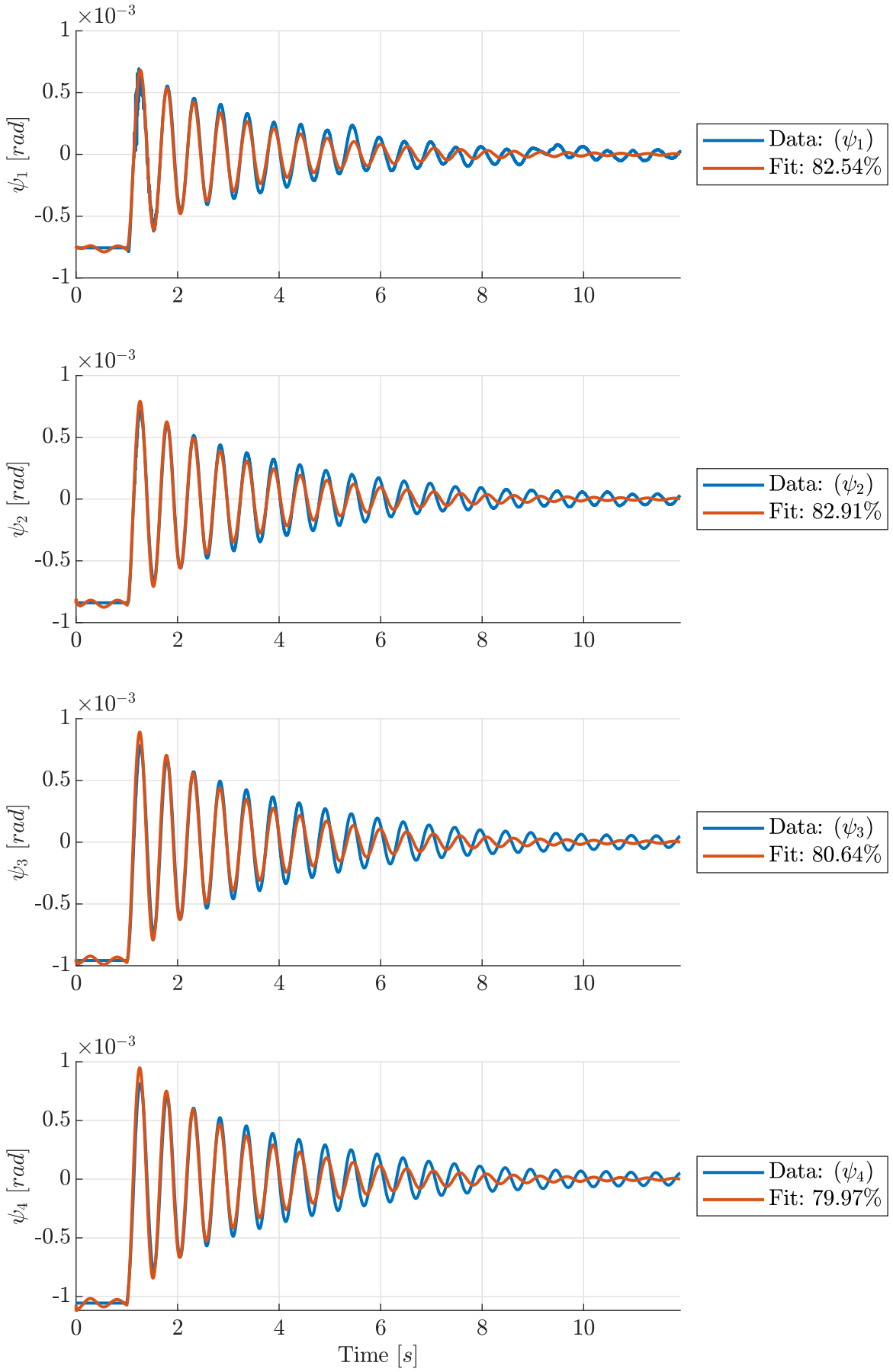


Figure 6.6: Comparison between validation data from the second most extended pose and associated ARMAX fit for SIMO-model.

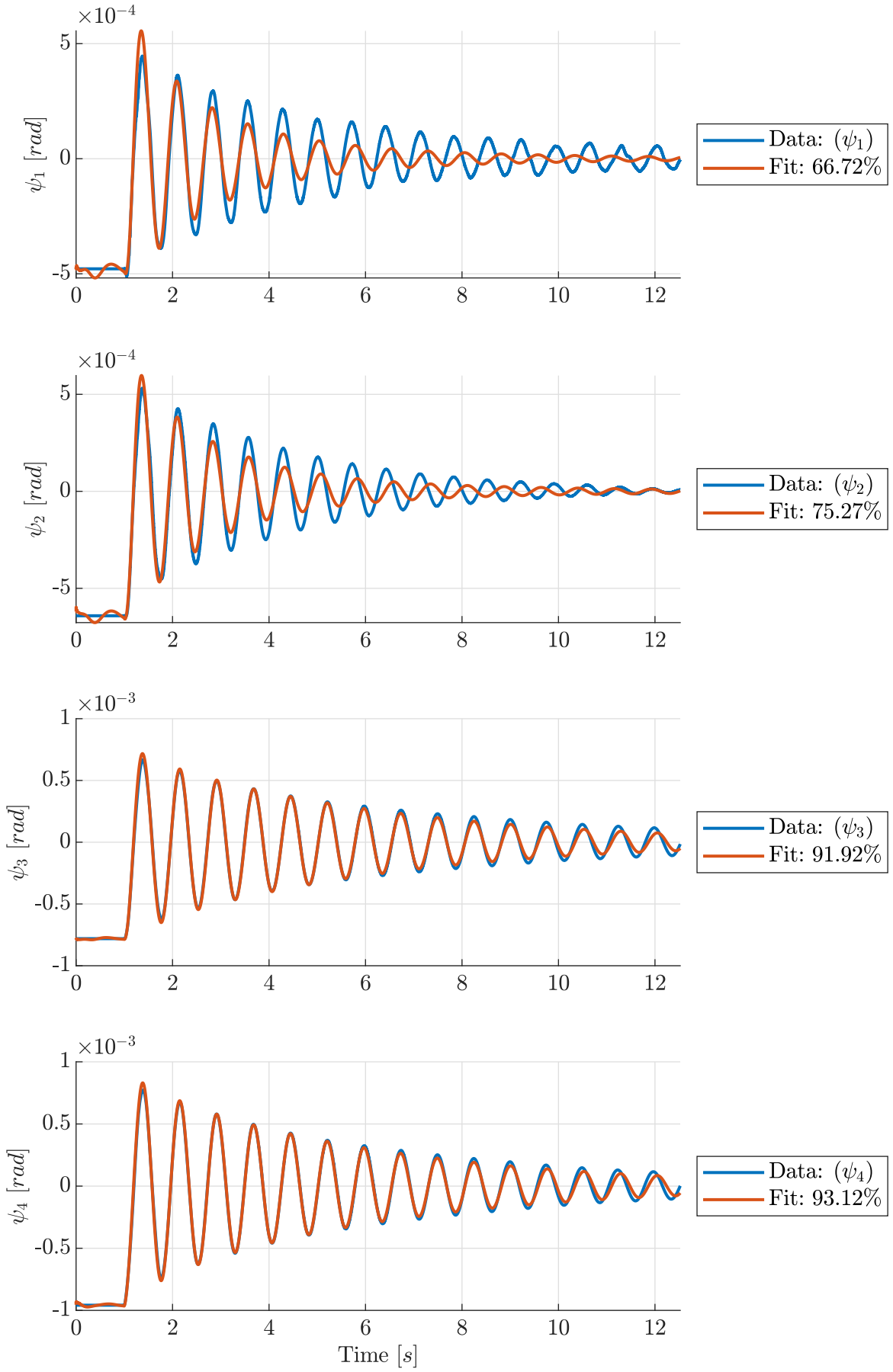


Figure 6.7: Comparison between validation data from the fully extended pose and associated ARMAX fit for SIMO-model.

6.3.3 Manual Calculations

The traditional evaluation of the validation data yielded approximations for inertia J_{eq}, spring stiffness k_{eq}, viscous damping b_{eq} and natural frequency ω_n, depending on the boom length d_3. The relevant validation data is presented in Figure 6.8, where the peaks used for calculations are indicated. The figure also presents a simulation of the free vibrations using the calculated properties and initial conditions from the validation data.

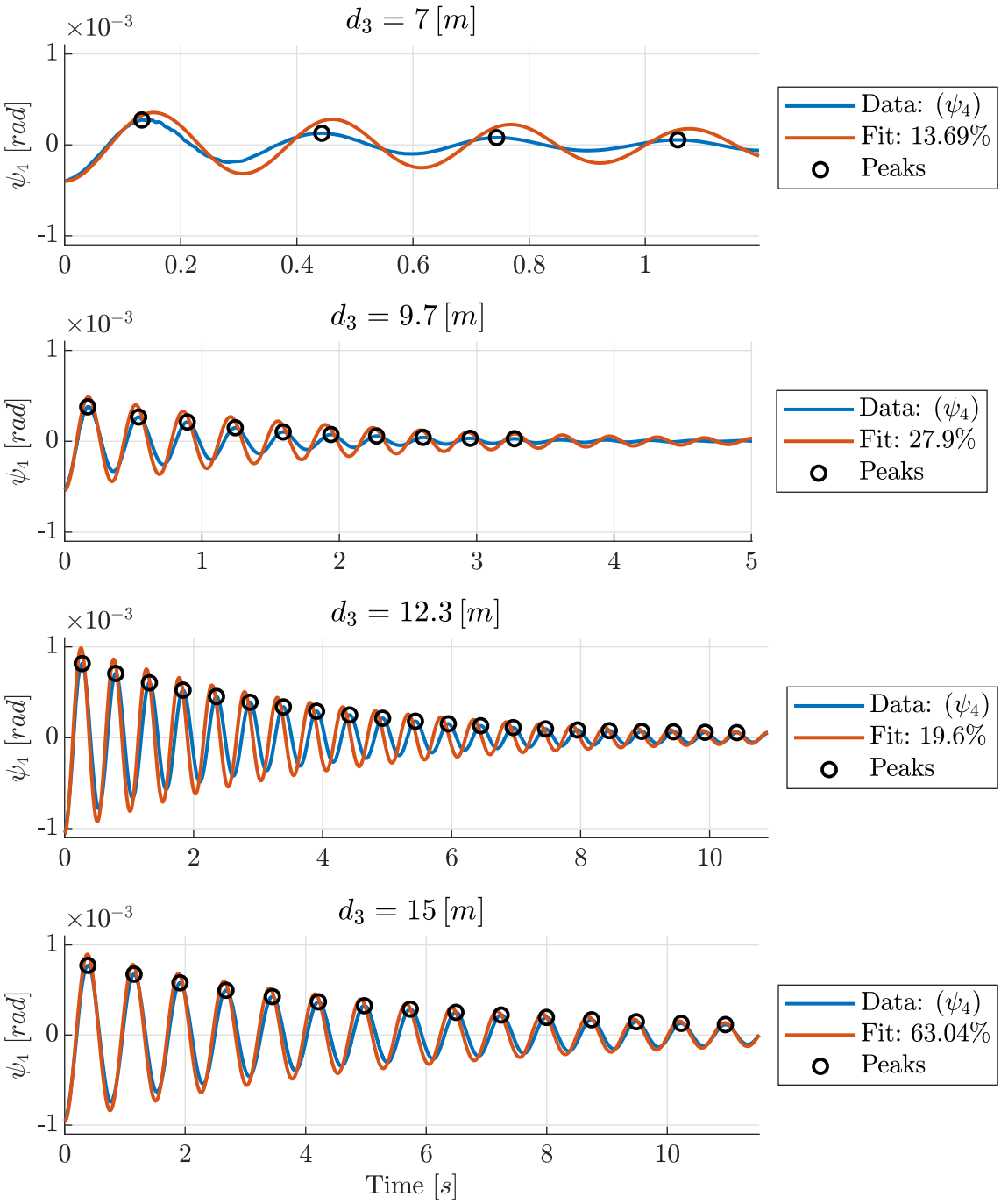


Figure 6.8: Comparison between validation data and manual simulation for different boom lengths.

The diagram reveals that the system behaves less and less like a damped oscillator as the boom is retracted. A possible source of errors is that the boom sustains a hose for carrying shotcrete. This hose weighs in at over $3000N$ at most and is anchored at three points on the boom. In the fully extended pose, it is relatively uniformly distributed, however when retracting the boom, the hose folds and eventually makes contact with the ground.

This characteristic is apparent in the calculated values as well, presented in Table 6.3. While the effective moments of inertia follow a decreasing trend along with the reduction in boom length, the spring stiffness breaks the trend for the innermost pose.

Table 6.3: Manually estimated flexibility properties by boom length.

$\boldsymbol{d_3}\ [m]$	$\boldsymbol{J_{eq}}\ [10^4 \cdot kgm^2]$	$\boldsymbol{k_{eq}}\ [10^6 \cdot Nm/rad]$	$\boldsymbol{b_{eq}}\ [10^4 \cdot Ns/rad]$	$\boldsymbol{\omega_n}\ [rad/s]$
7	1.4684	6.6925	1.8056	21.3491
9.7	2.9854	9.8988	3.4036	18.2093
12.3	4.7770	7.3270	2.5215	12.3847
15	8.0473	5.5675	2.8875	8.3177

6.4 Interface and Control

Results for interface and control were obtained through HIL-simulation. As long as the deflection dynamics are not implemented in real-time, the overall purpose of the simulation is to verify that the interfacing, control structure and planning algorithm works as intended. Considering that the dynamics in the HIL model are simplified and that the controller gains are not directly applicable to the physical system, the controller was tuned rudimentary. The physical system is simulated on the Speedgoat, controlled by the Danfoss ECU, based on setpoints provided by the HMI PC. Trajectory preferences were adjusted as presented in Table 6.4, producing the reference and simulated trajectory illustrated in Figure 6.9.

Table 6.4: Trajectory preferences for HIL testing.

Preference	**Value**
Velocity	0.1 $[m/s]$
Spacing	0.7 $[m]$
Depth	5.0 $[m]$

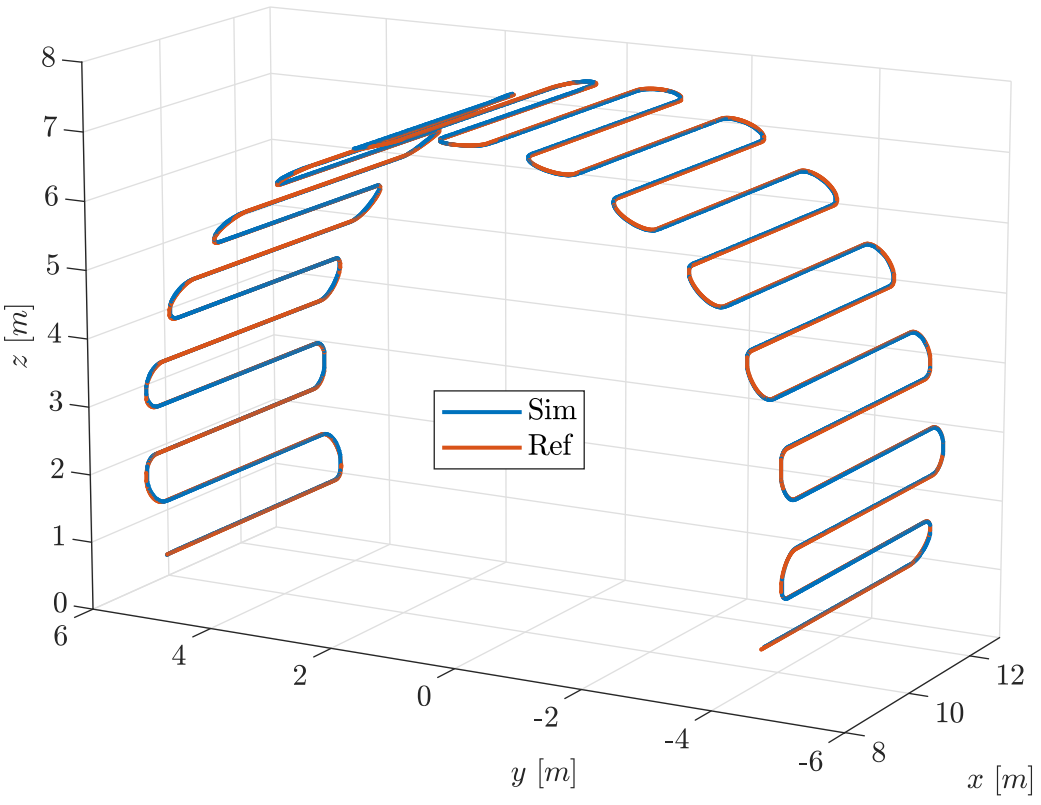


Figure 6.9: Comparison between reference and simulated trajectory in Cartesian coordinates.

The magnitude of the error between the trajectories is presented in Figure 6.10, while the NCP velocity is shown in Figure 6.11 and valve commands for a single stroke and two transitions are demonstrated in Figure 6.12.

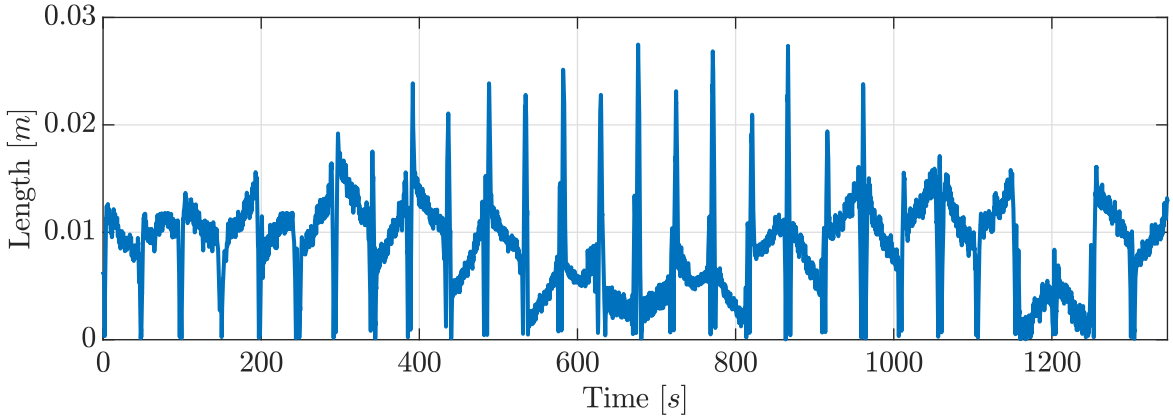


Figure 6.10: Simulated trajectory error magnitude.

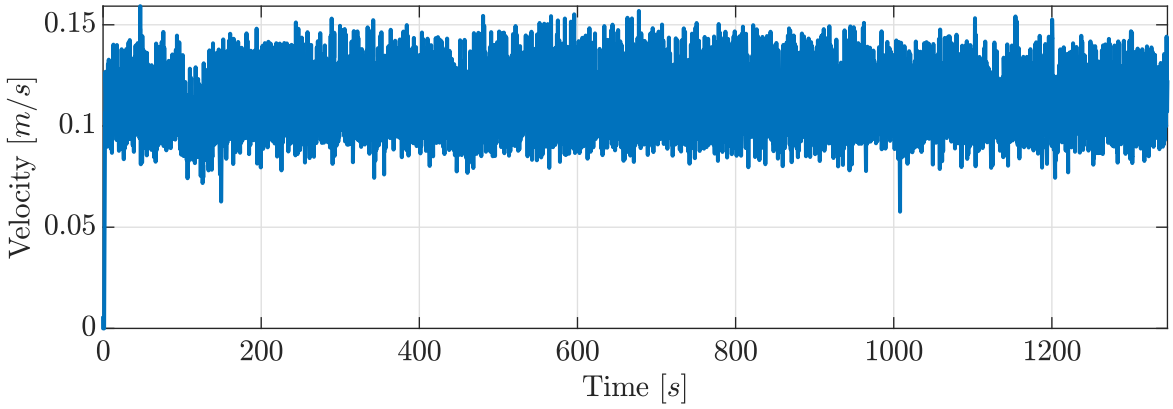


Figure 6.11: Simulated trajectory velocity.

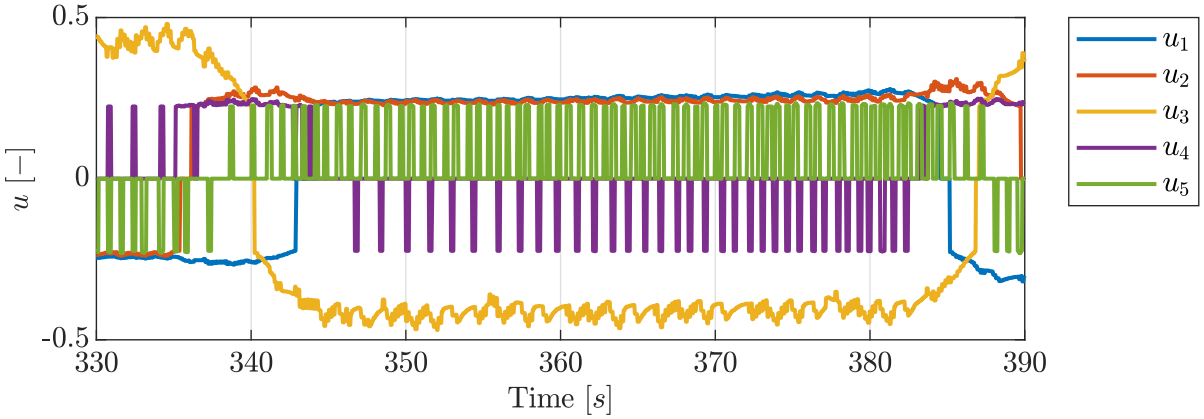


Figure 6.12: Simulated valve commands for one stroke.

Chapter 7

Discussion and Further Work

This chapter gives a discussion on the obtained results and the developed framework in general. Recommendations for further work and improvement are also given, indicated by indented sections under each discussed topic.

7.1 Kinematics

For the kinematic structure of the AMV 4200H, no closed-from solution has been found. Analytical expressions have been established for the boom position and the nozzle orientation separately, using kinematic decoupling. The proposed method requires iterations, as previously explained. The drawback of iterative methods is that the time of convergence cannot be guaranteed. In addition, it is computationally intensive, rendering the method unsuited to run in real-time on the ECU. Eliminating the iterations from the calculations would render the computing time more predictable since an exact solution would be acquired at the first evaluation. Furthermore, it would make the inverse kinematics faster, opening for the possibility of running the calculations in real-time on the ECU. It may also simplify the Jacobian matrix, requiring less computational effort to calculate the joint velocity, acceleration and reaction forces. In other words, obtaining a closed-form solution to the inverse kinematics is beneficial in all robotic applications.

> **Further work:** The current kinematic configuration of the AMV 4200H is cumbersome from an inverse kinematics perspective. The mechanical design should be redesigned to provide a closed-form solution. For a 6-DOF manipulator, a sufficient condition for guaranteeing a closed-form solution is that three consecutive joint axes intersect at a point (Craig, 2014). On 6-DOF industrial robots, it is typically the three last joints that intersect, making it possible to decouple the position and orientation, hence obtaining a closed-form solution. By the same reasoning, a closed-form solution should exist for the AMV 4200H if the last three joints are reconfigured to intersect at a single point.

7.2 Trajectory Planning

Surface: The proposed surface model is suited to the standard tunnel profiles used in Norwegian road tunnels. However, other tunnel geometries, such as railway tunnels and tunnels used in other countries, are not assessed in this thesis. This may render the surface model insufficient to describe certain tunnels. While the chord-length method provides a better fit than uniform parameterisation, the approximation error can be minimised further using numerical methods, as proposed by Grossman (1971). The methods used in the thesis are based on AMV's request to use a minimum of knot points to create a surface model, meaning that the operator dictates the measurement accuracy. We considered the alignment of the vehicle as a better approximation to the v-axis than the operator measurements. Therefore, the surface coordinate system is positioned such that the v-axis is parallel to the x-axis on the vehicle.

Further work: In a practical situation, the positioning of the vehicle should be taken into account, such that a spraying procedure can be initiated independent of the orientation of the vehicle. This can e.g. be achieved by positioning the vehicle within a virtual model of the environment and calculating the transform between the global tunnel coordinates and local coordinates on the vehicle. The authors consider the next natural progression for surface mapping to replace the manually collected knot points with laser measurements. This will increase point density and reduce measuring uncertainties.

Pattern: The traditional square pattern was evaluated as well as a pattern with semicircular transitions. Both patterns revealed strengths and disadvantages. The square pattern provides a more uniform distribution of shotcrete, while the rounded pattern provides a smoother operation. The rounded pattern imposes a sudden jump in acceleration when entering transition mode. This may induce oscillations in the boom, and a transition with continuous acceleration profile, such as Bezier curves may be considered to achieve a trajectory with higher continuity. The authors have not reviewed the optimal trajectory preferences, as adjusted in the HMI, with regard to layer thickness and distribution.

Further work: A thorough investigation of the effects on shotcrete distribution and nozzle accuracy from different patterns should be backed by field-testing. An optimal pattern is possibly a blend of the trajectories mentioned above, where the corners are rounded with a radius determined as a function of the nozzle velocity.

7.3 System Modelling

Dynamics: The AMV 4200H holds a multitude of system dynamics. This includes flexible bodies, hydraulics, rotating equipment and the concrete pump which creates pulsation effects. The impact of different system dynamics should be considered before relying on simulation results. The authors made an effort to model the complete hydraulic system in Simscape Multibody. However, this model proved too CPU intensive for the RT target. The hydraulic system is load-sensing and operates with overhead. Furthermore, the valves incorporate closed-loop control to ensure a linear relation between the valve input and hydraulic flow. Therefore, the hydraulic dynamics were considered negligible in relation to mechanical deflection.

Measurements: Measurements of the physical system was subject to temporal limitations. Both since the timing of the field-trip needed to adhere to the production cycle of the machine, and since the allotted time for the visit was limited. This means that the measurements were taken at an early stage in the project and that a "safe approach", by measuring free vibrations, was selected instead of frequency response testing. Measurements were taken; both for vertical and transverse motion, to create a 2-DOF deflection model. However, the transverse measurements were corrupted due to complications with the load-cell, causing inconsistencies in the magnitude of the input. The authors did observe that the transverse movements were subject to backlash in the connections between telescopic elements. This phenomenon should possibly be investigated further in future measurements.

Toolbox identification: The attempts to create a full system approximation using the proposed grey-box model did not yield any useful results. This suggests that the proposed state-space model of the physical system is not sufficient to capture all the system dynamics from the validation data. It is theorised that a more complex state-space model may have yielded a higher fit. The grey-box method assumes that the mathematical structure is known, and tries to estimate the unknown system coefficients based on the validation data. The black-box model, on the other hand, does not require a system model and as such, gives higher flexibility in terms of modelling errors. However, while the ARMAX-model provides high flexibility, the lack of a physical model means that it is not possible to select different inputs or outputs to a model after identification. This is the case for the measurements, where the validation data input is torque at the tip, while in simulation the

input is $\dot{\theta_2}$. Not being able to change the inputs and outputs from the model renders the black-box model of little use for simulation purposes. The black-box model provides confirmation that it is possible to obtain a fit to the measurements.

Manual calculations: While the toolbox approaches produced little useful results, the manual calculations provided useful approximations to physical properties of the system. On the other hand, it is evident that the resulting deflection behaviour is a superposition of multiple dynamics. The manual calculations only capture the first and slowest vibrational mode. This is the most dominant mode, and it is unknown to the authors if the remaining vibrational modes appear prominent during operation. The manual calculations have provided several system parameters such as stiffness, damping and eigenfrequencies that may be of interest in future work.

> **Further work:** Creating a dynamic model of the system is essential for achieving a representative simulation and for further insight into control optimisation. The authors suggest conducting a field-test to validate if the damped oscillator model provides a sufficient approximation to estimate the NCP position. If this is not the case, the state-space model can be expanded upon. An alternative approach could be to test the system by means of frequency response.

7.4 Interface and Control

Communication: The communication setup between the Speedgoat, Danfoss ECU, Router PC and HMI PC is not an optimal setup with regard to concerning data transmission. The Router PC contributes to unnecessary latency and has no practical function for the overall system other than forwarding and translating incoming/outgoing data. However, since the system response is relatively slow and HIL-simulation was performed under ideal circumstances with relatively low bus loads, latency in the communication has not been an issue.

> **Further work:** For future hardware simulations, dedicated hardware capable of both direct CAN bus communication and real-time simulation should be used to eliminate the Router PC and ensure more reliable communication.

Automatic spaying: When running the automatic spraying mode, the setpoint of the nozzle is updated in real-time based on periodic reference messages sent by the HMI PC. However, these messages are sent from an environment under the GIL, which means, depending on the CPU-load for that particular process, that the periodic messages may in some cases be postponed and sent in bulk as soon as the processor becomes available. The velocity deviations have been measured and were found to be relatively small, but the rate of setpoint updates, and therefore the nozzle velocity, cannot be guaranteed.

> **Further work:** Ensuring that the joint space references are updated at the correct time instant is instrumental for precise control of the machine. The ECU is purpose-built for controlling industrial systems, capable of running tasks in proper real-time with high reliability. Therefore, a possible solution is to build an array on the ECU before initiation of the spraying procedure, subsequently reading out the references using a timed process in the ECU. However, taking into account AMV's plans of achieving fully automatic shotcrete application with real-time process control and correction, it may prove impractical to upload static values. Another, and a possibly better solution would be to move calculation of inverse kinematics to the ECU; having a central processing computer that interprets the environment, monitors the application process and corrects the nozzle Cartesian position and orientation. In this case, only the Cartesian position and orientation references are updated, and all inverse kinematics, which is specific to the machine, is run on the ECU.

PC application: The Python application provides a working implementation of the proposed methods in accordance with modular programming techniques. Kinematics, surface transformation and pattern generation are divided into separate modules. This is part of an effort to simplify testing of different trajectory concepts or kinematic configurations and reduce dependencies. Each module can be developed separately without requiring particular knowledge of the other modules. Moreover, the computer application is equipped with safety features and protection against invalid user interactions to establish it as a platform for field-testing.

Control: The proposed control structure should, with proper tuning, be sufficient for controlling the nozzle in automatic spraying mode. However, the controller has not been tested in the presence of complex dynamics and disturbances. Therefore, optimal gain settings have not been investigated. The primary purpose of control in this thesis has been to verify that the developed system works, i.e. that the HMI PC correctly calculates and transfers data to the ECU, and that the ECU tracks the setpoint accordingly, albeit with simplified system dynamics. As a measure to protect the sliding joints from debris, the AMV 4200H is equipped with a suspended cover over the boom assembly. The cover retracts onto a spring-loaded roller. Boom extension, d_3 is measured using a rotary encoder embedded in the roller. This mechanism is potentially sensitive to disturbances, such as shotcrete debris. It is not known to which extent this may affect the accuracy of the d_3 measurements.

Further work: The next natural step of development is to finalise the dynamic model and provide the control system with disturbances such that controller performance can be assessed. It may also be possible to reduce mechanical disturbances by using the identified eigenfrequencies of the boom to filter the input signal. Using a lookup table of the eigenfrequencies and applying the correct cutoff frequency based on the extension of the boom could reduce resonance.

Chapter 8

Conclusion

This research aimed to provide a framework for automatic application of shotcrete for the AMV 4200H. An automatic spraying mode was developed that relies on interoceptive sensing, i.e. on joint feedback for the execution of the spraying procedure. First, an operator must provide a minimum of five probed reference points of the tunnel geometry. These points are subsequently used to generate the tunnel surface and spraying trajectory. Through an intuitive graphical user interface, the operator manages the spraying procedure and has full control over the process parameters such as the shape of the spraying pattern and nozzle velocity. Once a satisfactory trajectory is selected, the inverse kinematics are calculated and transferred to the ECU with ease. The proposed methods and framework have been tested and verified through HIL-simulation with the ECU of the AMV 4200H.

Throughout the thesis, several subjects have been investigated. This includes; forward and inverse kinematics, surface transformations and pattern generation, dynamic modelling and control, HIL-simulation and the development of a PC application with a functional HMI.

Forward kinematics were solved in a straightforward manner using the DH-convention. The inverse kinematics, on the other hand, were more challenging as no closed-form solution was found. The proposed method utilises a combination of kinematic decoupling and iteration to obtain the final solution. This method has shown promise in regard to computing time and convergence. An analytic solution would be preferable for real-time application. Unsolvable kinematic structures are a reoccurring problem with retrofitting automatic control to machines intended for manual operation. This may be considered somewhat analogous to how early robotic manipulators were not initially designed for shotcrete equipment, as most shotcrete manipulators of today are not purpose-built for automatic control.

The spraying trajectory was created based on simplified assumptions. These assumptions include assuming zero Gaussian curvature, an approximation of tunnel cross section by polynomials and the creation of a spraying pattern resembling a square pulse wave. The developed spraying trajectory serves as intended, but no further investigation has been conducted in regard to optimal trajectory for application of shotcrete. For industrial implementation, the trajectory should be developed in collaboration with experts in the field and should be subject to rigorous testing for identifying optimal process settings.

Identification of flexible-body dynamics was conducted using validation data from the physical system. Attempts were made using traditional free vibration theory, as well as grey-box and black-box identification, where only the manual calculations yielded credible results. For accurate simulation, it is necessary to consider the system dynamics. The identified model has not been compared to the physical system in terms of tracking the NCP.

Fundamental closed-loop control, incorporating PI-controllers and deadband compensation was developed. A testing platform, consisting of a computer application, Danfoss ECU and an HIL setup with simplified dynamics was constructed. The simulation provided confirmation of interfacing and

verification of the ECU-implemented controller, while the computer application incorporated the methods for kinematics and trajectory planning. The setup was composed such that the simulated system can be replaced with the physical machine for field-testing. Preferences in the computer application are managed from a simple and intuitive HMI, however, it has not been assessed by an operator.

In this thesis, many subjects have been investigated; some more than others. Due to the scale of the research, not all subjects could be studied in-depth as much as the authors desired to, and work remains for a commercially viable solution. However, methods, results and discussions are divided into separate categories, formulating distinct research areas. This facilitates future development by allowing many researchers to collaborate on the same project.

This work serves as a foundation for further development towards AMV's goal of realising a fully autonomous shotcrete operation with real-time process control and correction.

Bibliography

AMV. Application for An innovative machine for digitalization of shotcrete operations in tunnelling with real-time process control and 3D mapping. 2018.

Åström, K. J. and Hägglund, T. *Advanced PID Control*. ISA - Instrumentation, Systems, and Automation Society, 2006.

CAN in Automation. CiA 301 - CANopen Application Layer and Communication Profile. 2002.

Cheng, M., Liang, Y., Wey, C., Chou, J., and Chen, J. Development of a New Dimension and Computer-Aided Construction System for Shotcreting Robot. *Proceedings of the 13th International Symposium on Automation and Robotics in Construction*, 1996.

Craig, J. J. *Introduction to robotics : mechanics and control*. Springer, 2014.

Danfoss. Technical Information, PVG 32. 2015.

Dhir, R. K. *Advances in ready mixed concrete technology : proceedings of the first International Conference on Ready-Mixed Concrete*. Pergamon Press, 1976.

Girmscheid, G. and Moser, S. Fully automated shotcrete robot for rock support. *Computer-Aided Civil and Infrastructure Engineering*, 2001. 16(3). doi:10.1111/0885-9507.00226.

Grossman, M. Parametric Curve Fitting. *The Computer Journal*, 1971.

Hexagon Manufacturing Intellegence. Leica Absolute Tracker AT960 ASME Specifications. 2016. URL `https://www.hexagonmi.com/-/media/Files/Hexagon/HexagonMI/Brochures/LaserTrackerSystems/HexagonMIAT960DatasheetA4_en.ashx`.

Hofler, J. and Schlumpf, J. *Shotcrete in Shotcrete in Tunnel Construction*. 2004.

Honegger, M., Schweitzer, G., Tschumi, O., and Amberg, F. Vision supported operation of a concrete spraying robot for tunneling work. 2002. pages 230–233. doi:10.1109/mmvip.1997.625333.

Kleppe, A. L. and Egeland, O. Inverse Kinematics for Industrial Robots using Conformal Geometric Algebra. *Modeling, Identification and Control*, 2016. 37(1).

Kurth, T., Gause, C., and Rispin, M. Robotic shotcrete applications for mining and tunneling. *North American Tunneling 2004*, 2010. doi:10.1201/9781439833759.ch27.

Marsden, J. E. and Tromba, A. *Vector Calculus*. W.H. Freeman and Company Publishers, 6. ed edition, 2012.

Nabulsi, S., Rodriguez, A., and Rio, O. Robotic Machine for High-Quality Shotcreting Process. *ISR/Robotik 2010*, 2010. pages 1137–1144.

National Instruments. Selecting a Model Structure in the System Identification Process. 2018. URL `http://www.ni.com/product-documentation/4028/en/`.

National Instruments. Controller Area Network (CAN) Overview. 2019. URL `http://www.ni.com/en-no/innovations/white-papers/06/controller-area-network--can--overview.html`.

Nise, N. S. *Control Systems Engineering*. Wiley, 2011.

Norwegian Public Roads Administration. *Road Tunnel Design Manual*. Norwegian Public Roads Administration, 2004. URL `https://www.vegvesen.no/_attachment/61416/binary/14123`.

Norwegian Tunnelling Society. *The Principles of Norwegian Tunnelling*. Norwegian Tunnelling Society, 2017.

Pepperl+Fuchs. Datasheet - ENA42HD-S***-CANopen. 2018. URL `https://www.pepperl-fuchs.com/norway/no/classid_362.htm?view=productdetails&prodid=76995`.

Perwass, C. Teaching Geometric Algebra with CLUCalc. 2004.

Rao, S. S. *Mechanical Vibrations*. Prentice Hall, third ed. edition, 1995.

Siciliano, B., Sciavicco, L., Villani, L., and Oriolo, G. *Robotics - Modelling, Planning and Control*. Springer, 2009.

Speedgoat GmbH. Speedgoat Baseline real-time target machine. 2019. URL `https://www.speedgoat.com/products-services/real-time-target-machines/baseline/capabilities`.

Spong, M. W., Hutchinson, S., and Vidyasagar, M. *Robot Modeling and Control*. Wiley, 2005.

Teichert, P. Carl Akeley, A tribute to the founder of shotcrete. *Shotcrete*, 2002.

Tørdal, S. S., Hovland, G. E., and Tyapin, I. Efficient Implementation of Inverse Kinematics on a 6-DOF Industrial Robot using Conformal Geometric Algebra. *Advances in Applied Clifford Algebras*, 2016. 27(3).

Tosi, S. *Matplotlib for Python Developers*. Packt Publishing, 2009.

US Army Corps of Engineers. Standard Practice for Shotcrete. 2005.

Wang, X. and Su, X. Modeling and sliding mode control of the upper arm of a shotcrete robot with hydraulic actuator. *IEEE ICIT 2007 - 2007 IEEE International Conference on Integration Technology*, 2007. pages 714–718. doi:10.1109/ICITECHNOLOGY.2007.4290413.

Wikimedia Commons. CAN-bus-frame in base format without stuffbits. 2014. URL `https://en.wikipedia.org/wiki/File:CAN-Bus-frame_in_base_format_without_stuffbits.svg`.

Xuewen, R., Yibin, L., and Rui, S. Kinematic analysis of a shotcreting robot. *2010 International Conference on Mechanic Automation and Control Engineering, MACE2010*, 2010. (1):2640–2643. doi:10.1109/MACE.2010.5536789.

List of Figures

List of Tables

Appendices

Appendix A

Transformation Matrices

$$\mathbf{T}_i^{i-1} = \left[\begin{array}{ccc|c} & \mathbf{R} & & \mathbf{T} \\ \hline 0 & 0 & 0 & 1 \end{array}\right] = \mathbf{Rot}_{z_{i-1}}(\theta_i) \cdot \mathbf{Trans}_{z_{i-1}}(d_i) \cdot \mathbf{Trans}_{x_{i-1}}(a_i) \cdot \mathbf{Rot}_{x_{i-1}}(\alpha_i) \tag{A.1}$$

$$\mathbf{Rot}_{z_{i-1}}(\theta_i) = \left[\begin{array}{ccc|c} \cos\theta_i & -\sin\theta_i & 0 & 0 \\ \sin\theta_i & \cos\theta_i & 0 & 0 \\ 0 & 0 & 1 & 0 \\ \hline 0 & 0 & 0 & 1 \end{array}\right] \tag{A.2}$$

$$\mathbf{Trans}_{z_{i-1}}(d_i) = \left[\begin{array}{ccc|c} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & d_n \\ \hline 0 & 0 & 0 & 1 \end{array}\right] \tag{A.3}$$

$$\mathbf{Trans}_{x_{i-1}}(a_i) = \left[\begin{array}{ccc|c} 1 & 0 & 0 & a_i \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ \hline 0 & 0 & 0 & 1 \end{array}\right] \tag{A.4}$$

$$\mathbf{Rot}_{x_{i-1}}(\alpha_i) = \left[\begin{array}{ccc|c} 1 & 0 & 0 & 0 \\ 0 & \cos\alpha_i & -\sin\alpha_i & 0 \\ 0 & \sin\alpha_i & \cos\alpha_i & 0 \\ \hline 0 & 0 & 0 & 1 \end{array}\right] \tag{A.5}$$

Note that line separation in the matrices are given to separate rotation and translation.

Rotational Matrices for Nozzle Roll and Pitch

$$\mathbf{Rot}_x(\alpha) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos(\alpha) & -\sin(\alpha) \\ 0 & \sin(\alpha) & \cos(\alpha) \end{bmatrix} \tag{A.6}$$

$$\mathbf{Rot}_y(\gamma) = \begin{bmatrix} \cos(\gamma) & 0 & \sin(\gamma) \\ 0 & 1 & 0 \\ -\sin(\gamma) & 0 & \cos(\gamma) \end{bmatrix} \tag{A.7}$$

Appendix B

Dimensions

Table B.1: Kinematic dimensions.

	Length $[m]$
a_1	0.235
a_2	0.355
a_4	0.201
a_5	0.355
d_1	0.505
d_5	0.008
d_6	0.678

Table B.2: Actuating geometry for joint q_2.

	Length $[m]$
L_{x1}	0.335
L_{x2}	0.985
L_{y1}	0.290
L_{y2}	0.630

Appendix C

Tunnel Geometry

C.1 Parametric Functions

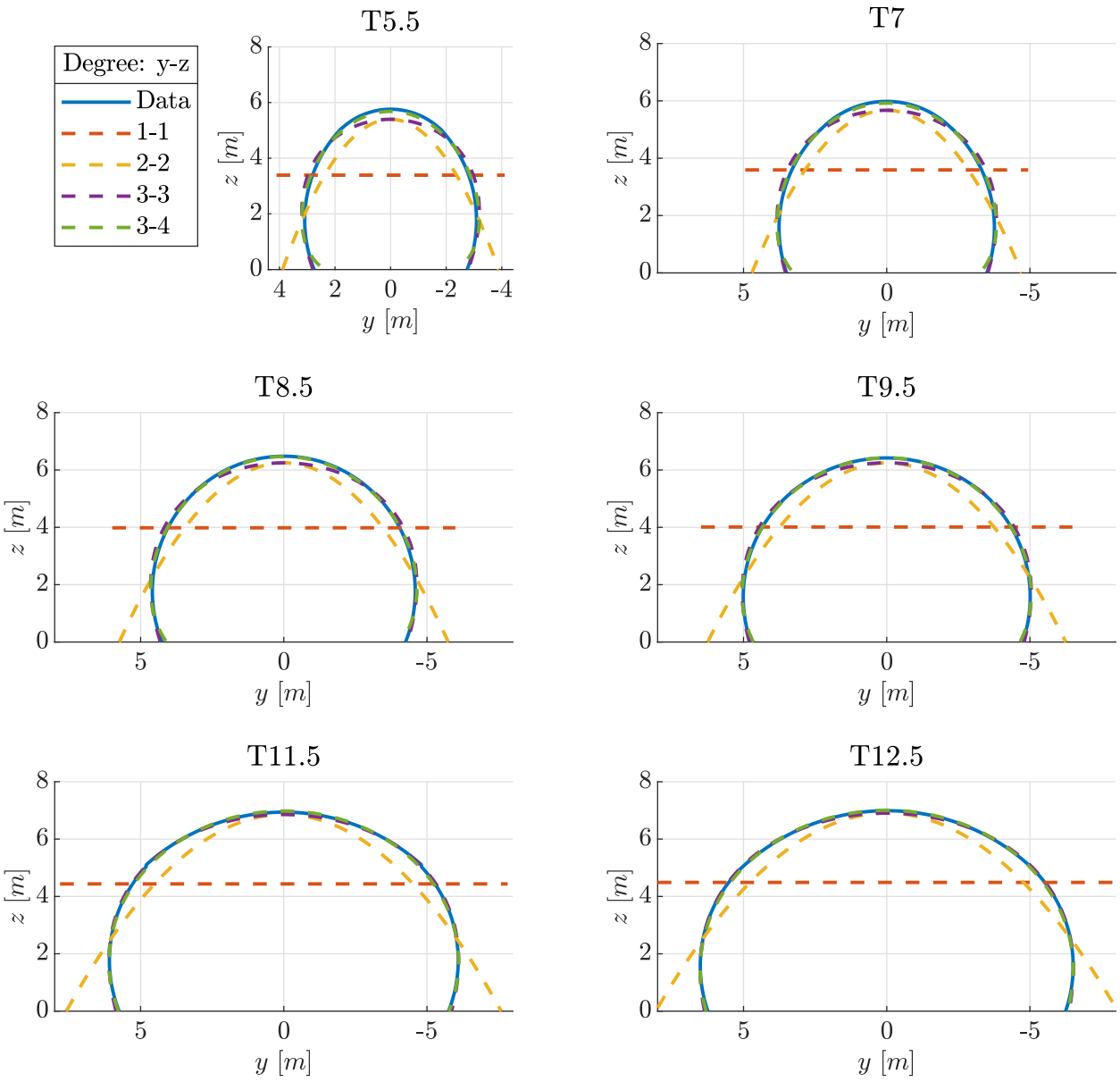


Figure C.1: Comparison of parametric tunnel approximation by polynomial degree of y and z functions.

Table C.1: Mean square error for best fit of y and z parametric functions by polynomial degree for major Norwegian road tunnel profiles.

MSE	y $[m]$			z $[m]$			
Degree	1	2	3	1	2	3	4
T5.5	0.3575	0.3575	0.0083	3.2876	0.0672	0.0672	0.0030
T7	0.3734	0.3734	0.0043	3.5125	0.0510	0.0510	0.0016
T8.5	0.5048	0.5048	0.0020	4.1432	0.0376	0.0376	0.0001
T9.5	0.4806	0.4806	0.0007	4.0366	0.0241	0.0241	0.0000
T11.5	0.6185	0.6185	0.0008	4.7071	0.0126	0.0126	0.0014
T12.5	0.5650	0.5650	0.0011	4.6832	0.0084	0.0084	0.0006

Appendix D

Block Diagrams

D.1 Simulink Blocks Speedgoat

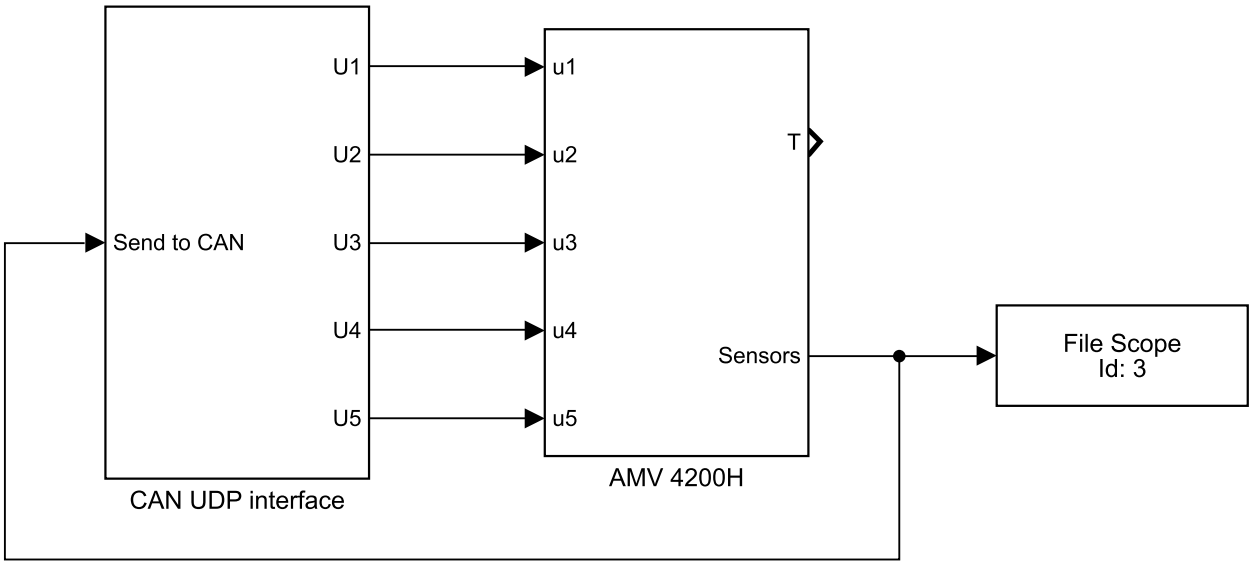


Figure D.1: Top layer block in RT-target. Contains CAN/UDP communication and the physical system AMV 4200H-

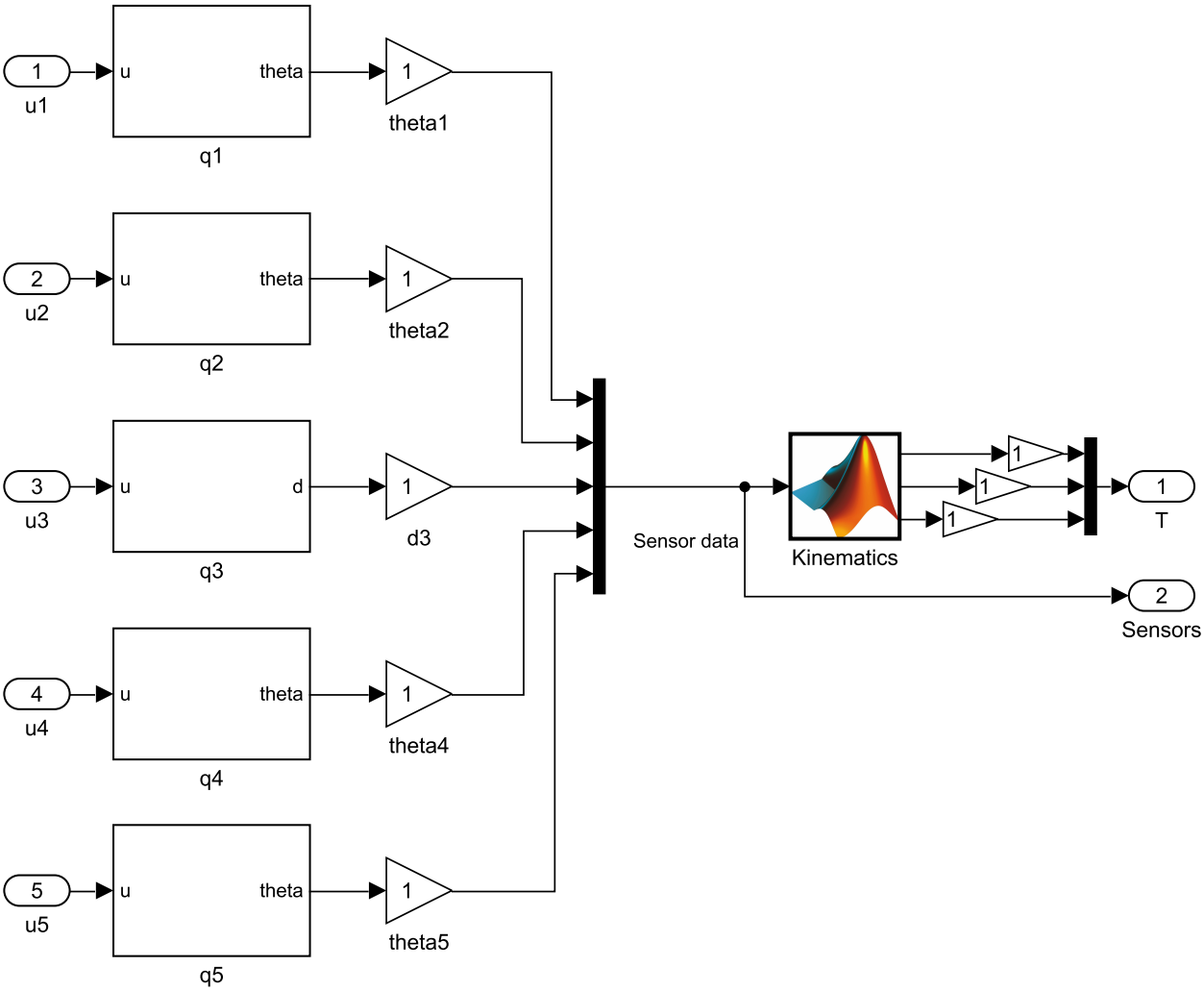


Figure D.2: This block is the AMV 4200H from Figure D.1. It represents the physical system in the HIL-setup.

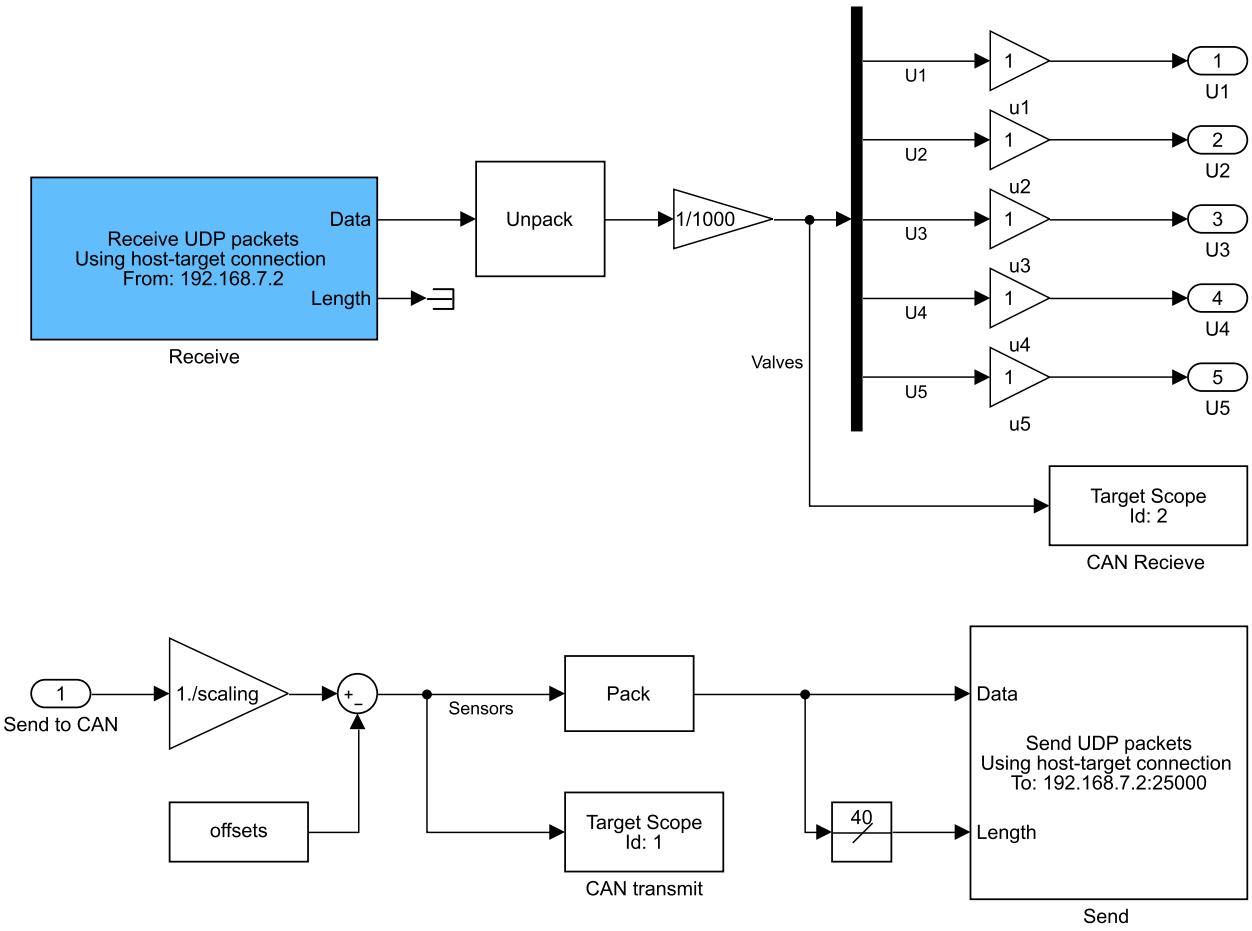


Figure D.3: Block diagram for Speedgoat receiving valve commands over UDP from Router PC, and for sending the sensor values back to the Router PC.

D.2 Simulink Blocks Router PC

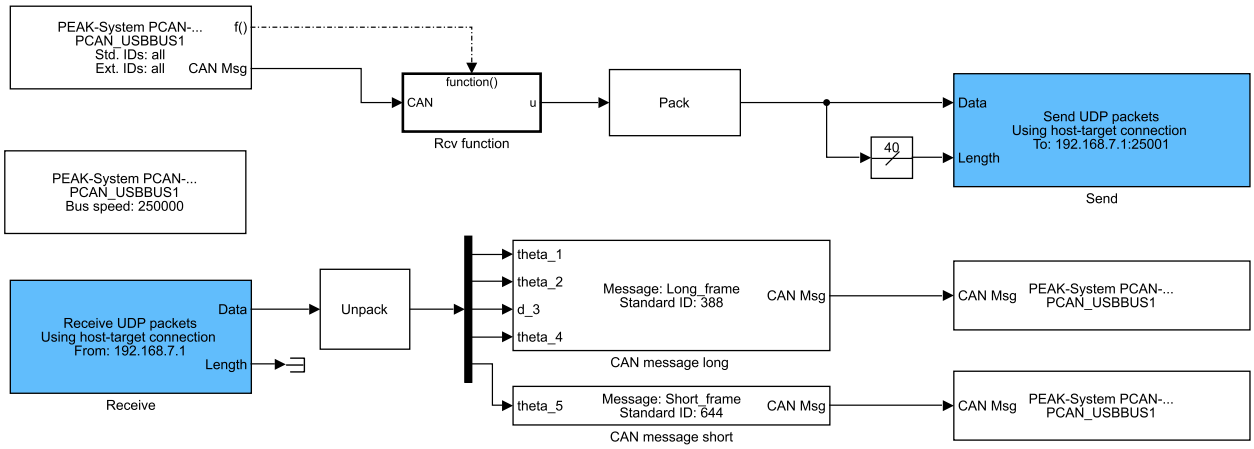


Figure D.4: Block diagram for receiving CAN messages from microcontroller and receiving incoming sensor values from the Speedgoat over UDP and routing back to the microcontroller over CAN bus.

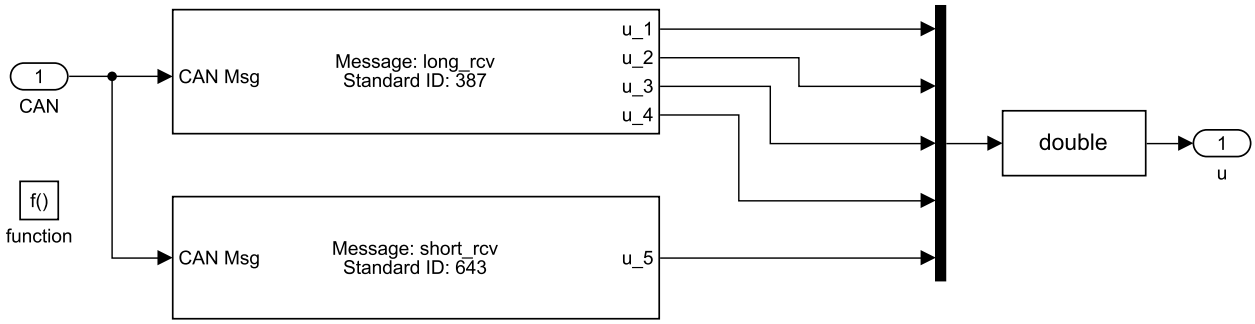


Figure D.5: Block diagram representing the "RCV function" from Figure D.4. Subsystem active on message recieved.

D.3 Danfoss Plus+1

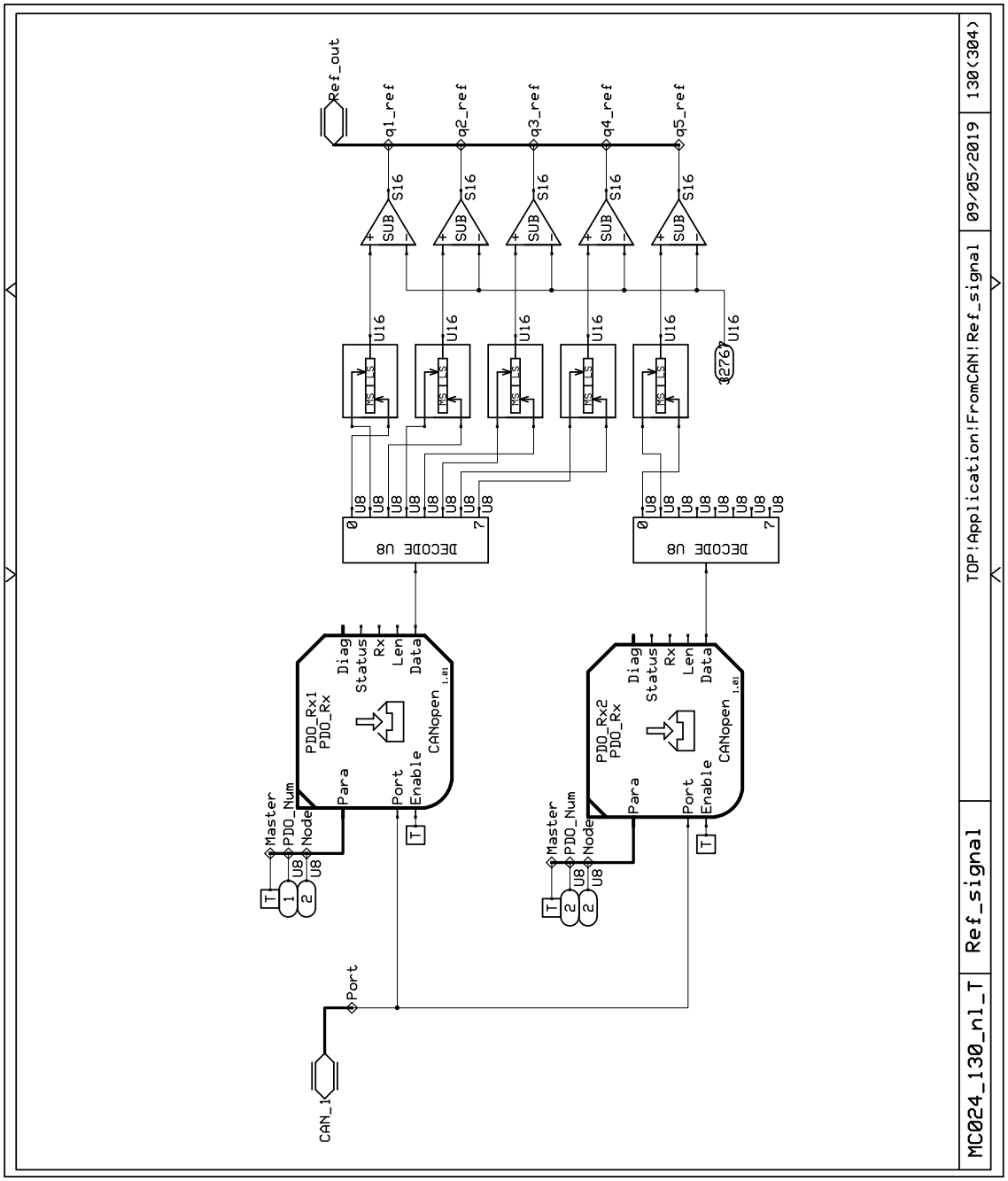


Figure D.6: Page for receiving reference values from HMI PC. Contains CANopen Rx block and conversion from U16 to S16.

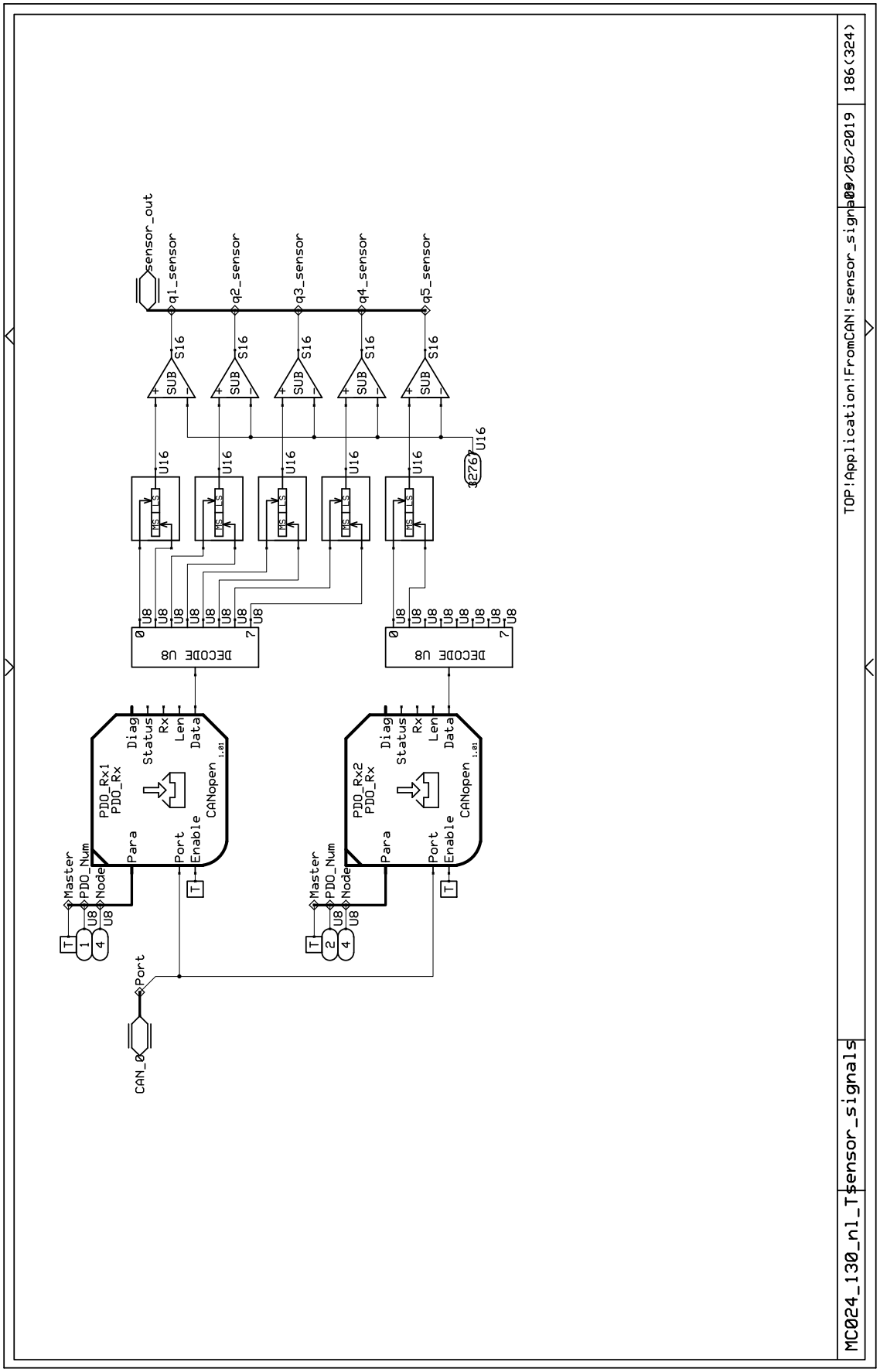


Figure D.7: Page for receiving sensor values from RT-target. Contains CANopen Rx block and conversion from U16 to S16.

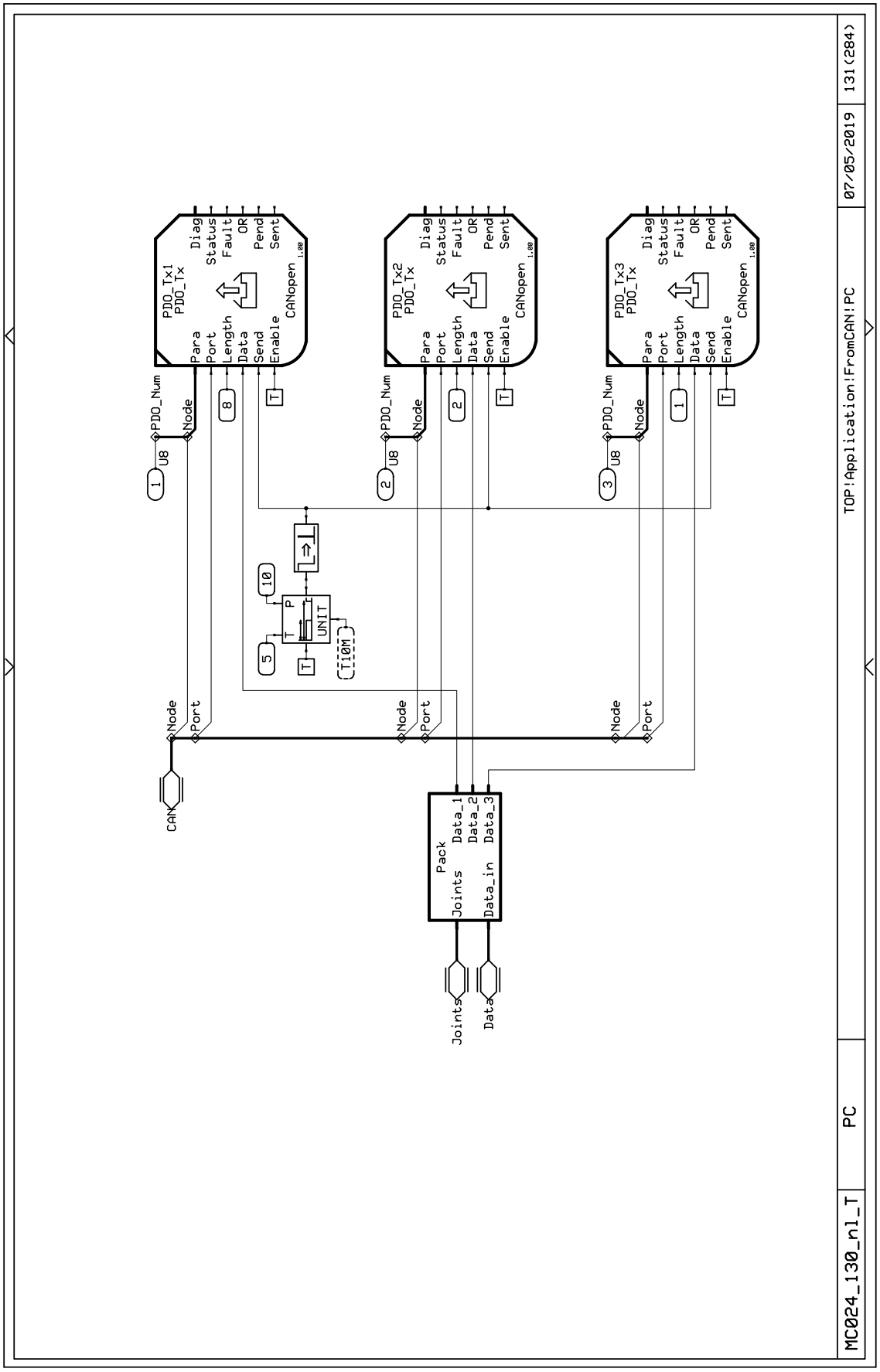


Figure D.8: Page for transmitting joystick button press used for knot points, controller on/off Boolean and sensor values to HMI PC. Using PDOs 1,2 and 3, from sending node 3.

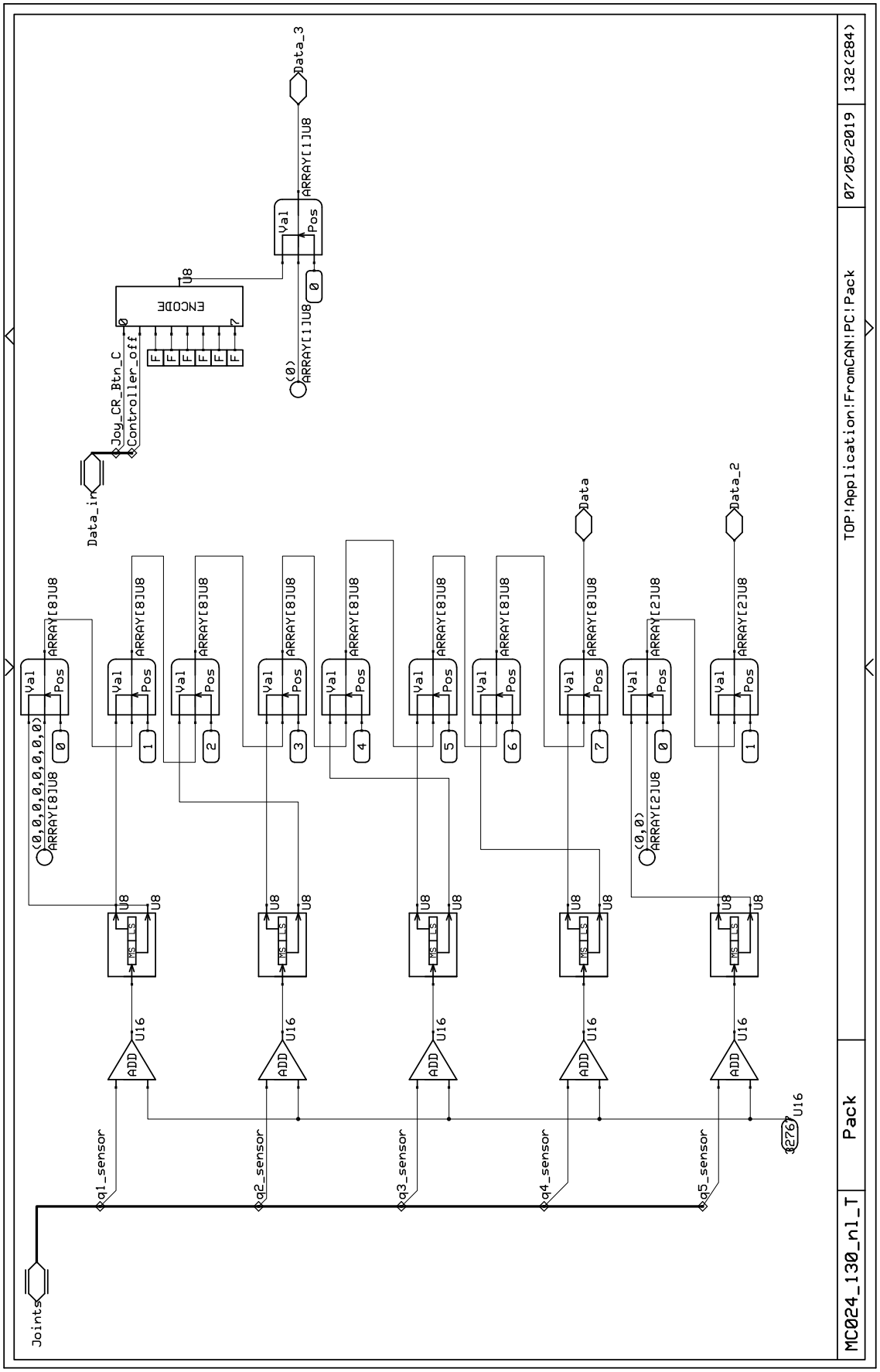


Figure D.9: Block for packing joystick button press, controller on/off Boolean and joint angles. Packs data to arrays of U8.

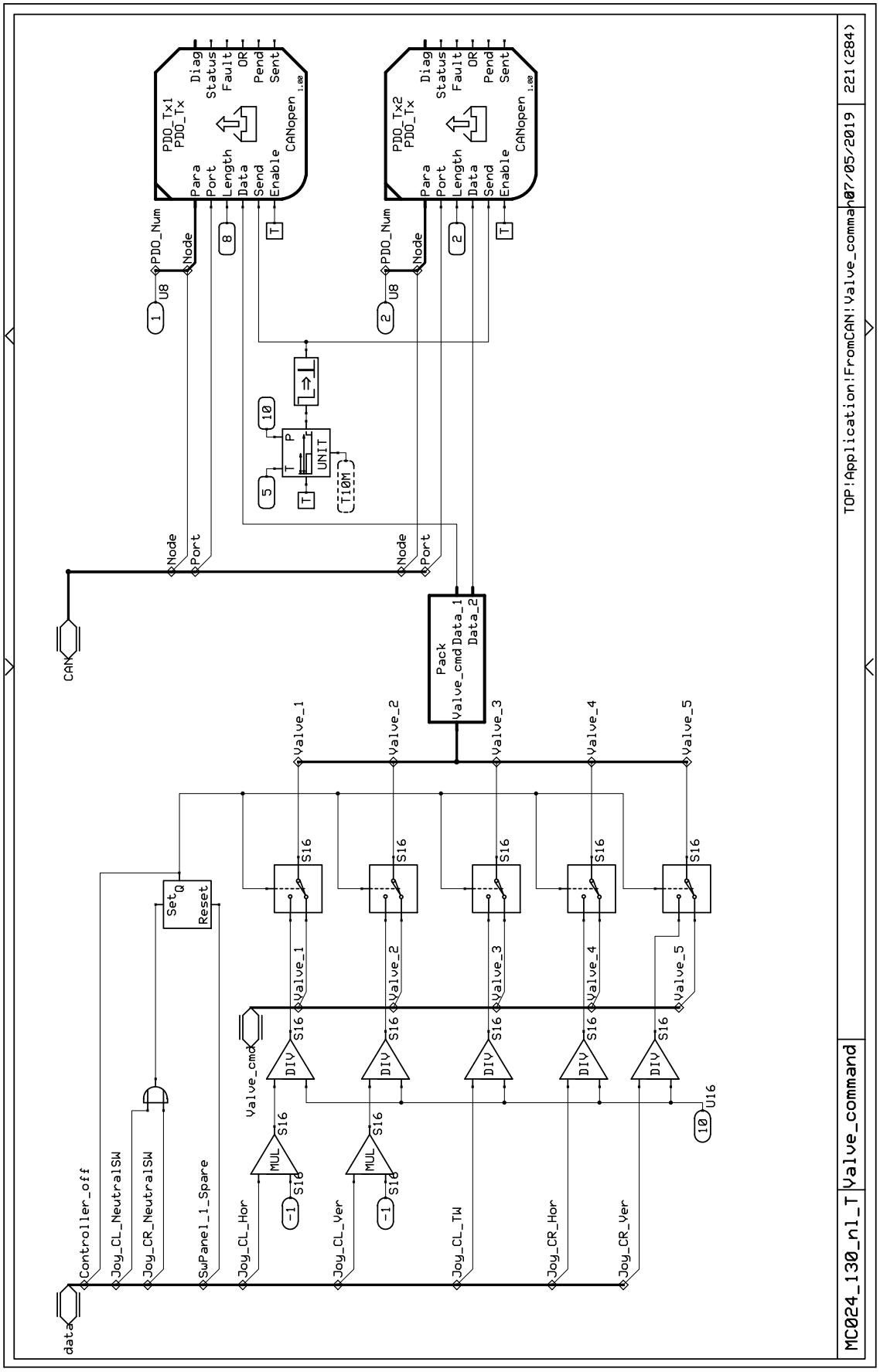


Figure D.10: Block for sending data joint angles to RT target. Block contains a switch mechanism to toggle manual and PI control.

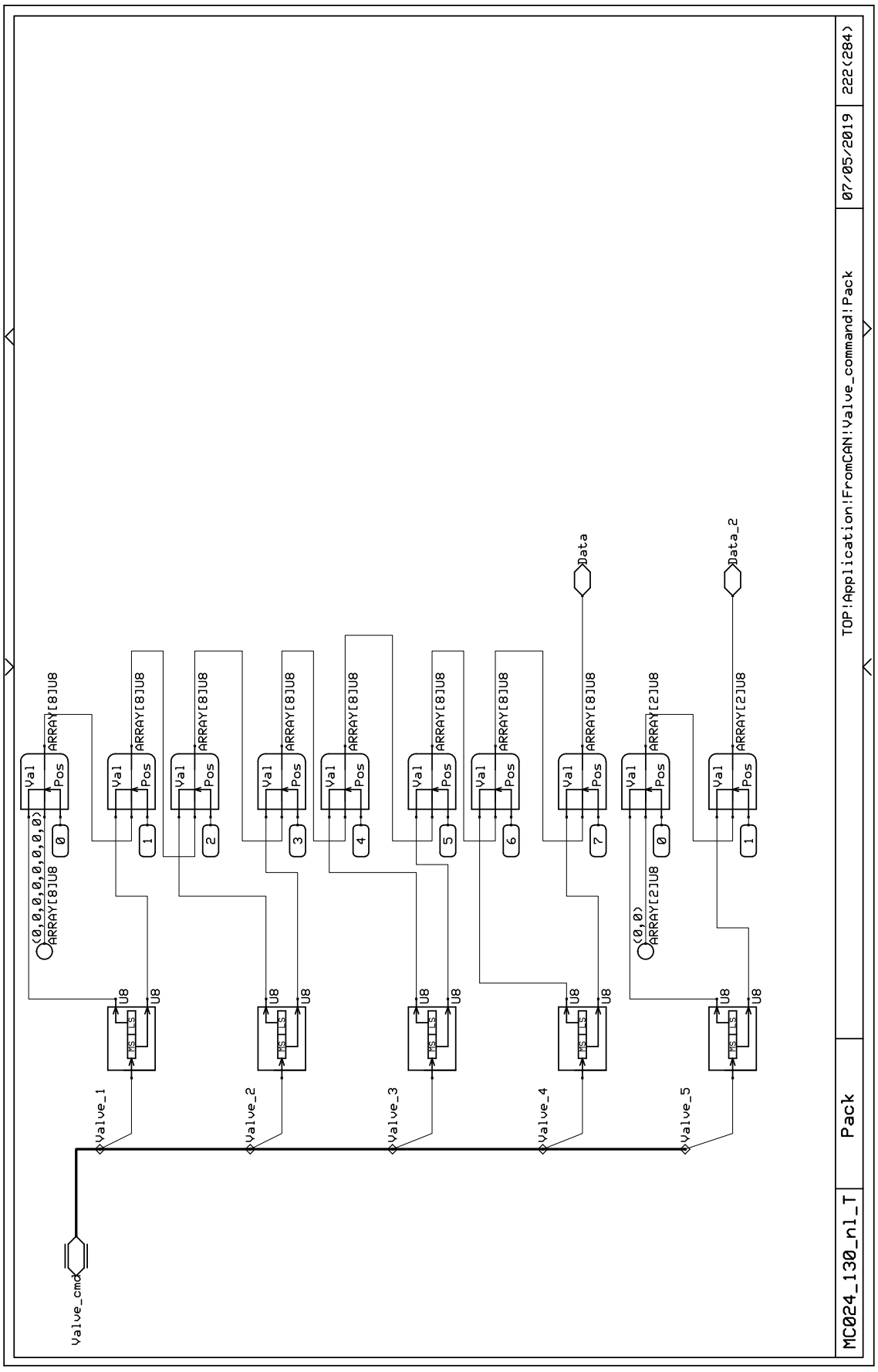


Figure D.11: Block for packing valve commands from U16 to arrays of U8.

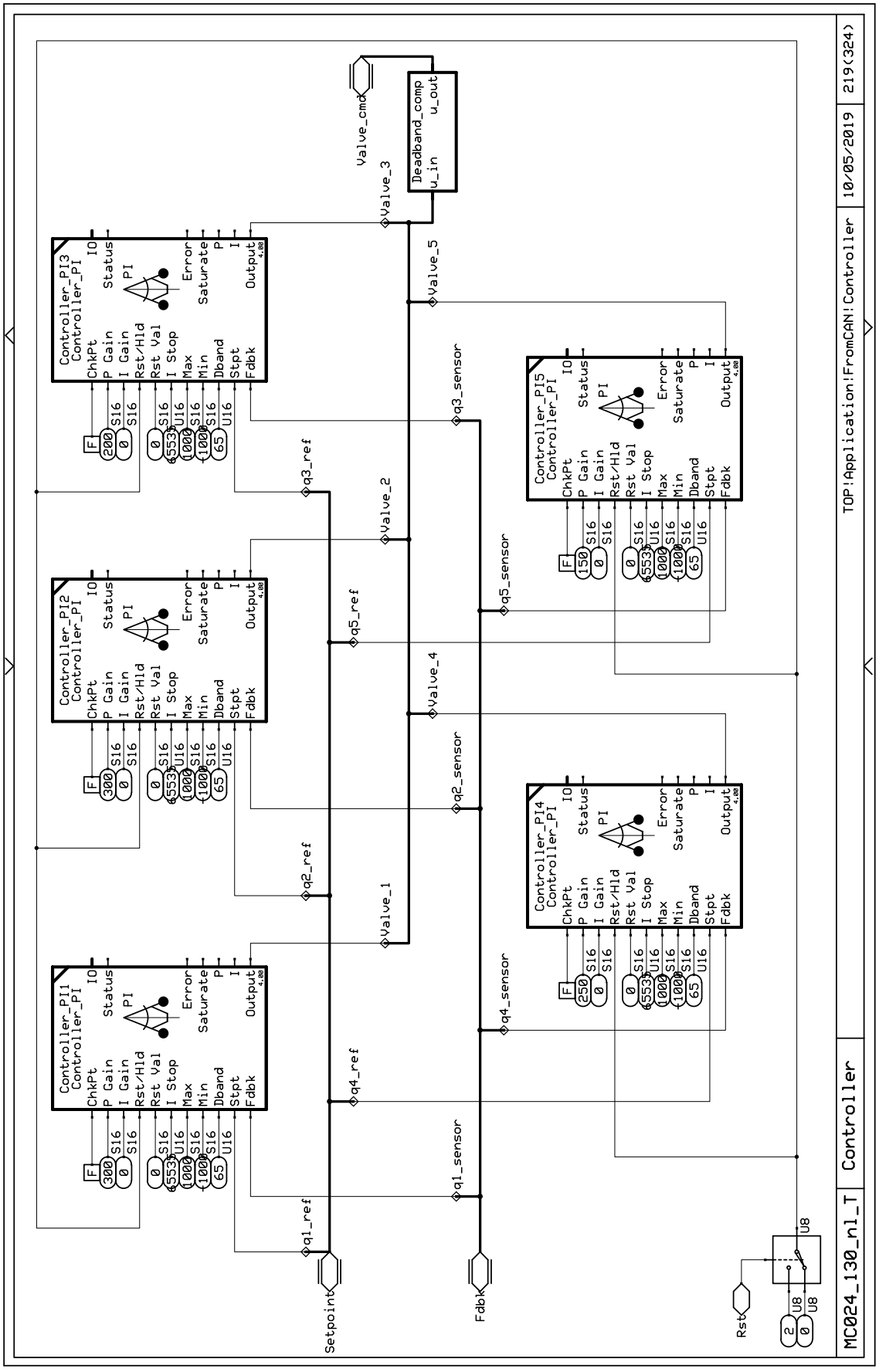


Figure D.12: PI-controller implementation.

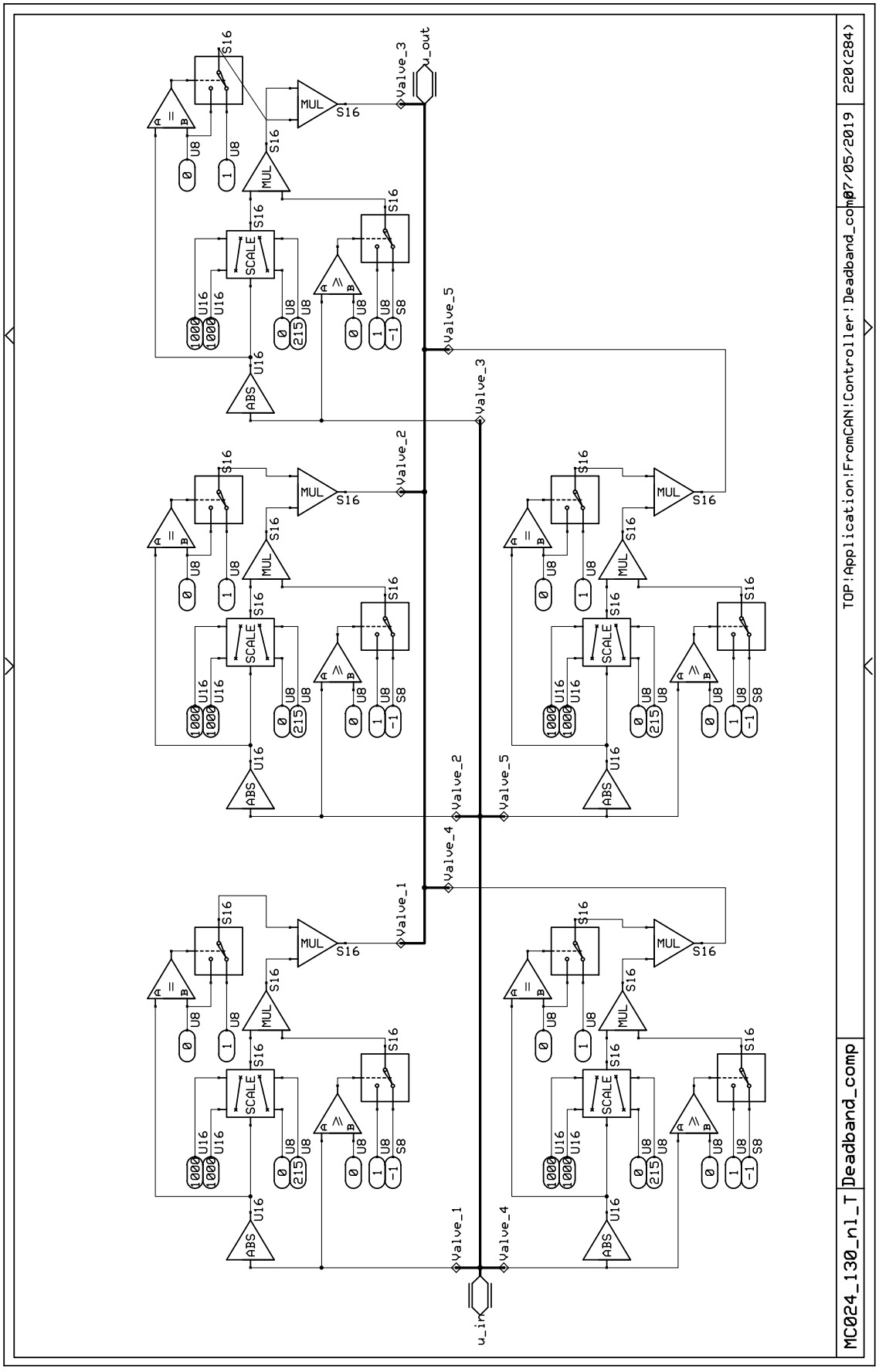


Figure D.13: Deadband compensator.

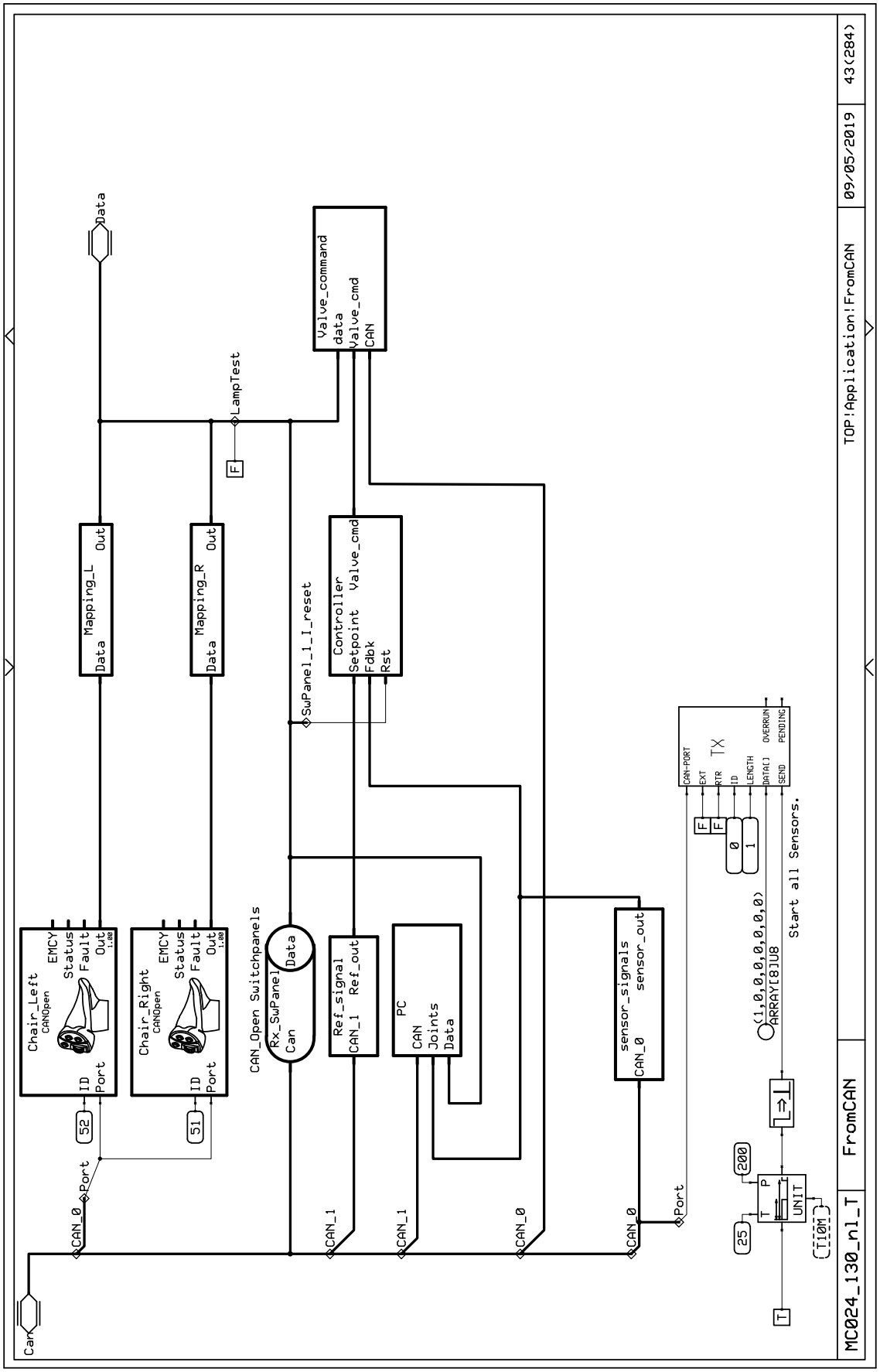


Figure D.14: Main page in Danfoss application. Contains communication, controller, data conversion and joysticks.

Appendix E

Source Code

E.1 MATLAB

E.1.1 InverseKinematicsWrist.m

This script finds a solution for (θ_4, θ_5) when $(\theta_1, \theta_2, \beta, \gamma)$ is known.

```
%% Initialise and declare
clear all
close all
clc
syms q1 q2 q3 q4 q5 beta gamma x y z real

%% Solve
q = [q1 q2 q3 q4 q5];
T = fk(q); % Forward kinematics transformation matrix
eq = T(1:3,1:3) == rotx(beta)*roty(gamma); % Rotational sequence: 'X Y(Z)'
sol = solve(eq(1:3,3),[q4 q5],"Real", true);

q4 = sol.q4(2)
q5 = sol.q5(2)

%% Functions
function T = fk(q)
% Geometry
a1 = 0.235;
a2 = 0.355;
a4 = 0.20098;
a5 = 0.345;
d1 = 0.505;
d5 = 0.00837;
d6 = 0.6928;

% DH table
DH = [
    q(1)         d1      -a1       pi/2;
    q(2)+pi/2    0        a2       pi/2;
    0            q(3)     0        0;
    q(4)         0       -a4       pi/2;
    q(5)         -d5       -a5      -pi/2;
    0            d6       0        0];

% Transformation matrices
T1 = T_DH(DH(1,:));
T2 = T_DH(DH(2,:));
T3 = T_DH(DH(3,:));
T4 = T_DH(DH(4,:));
T5 = T_DH(DH(5,:));
T6 = T_DH(DH(6,:));

T = T1*T2*T3*T4*T5*T6;
end
```

```
46
47 function T = T_DH(u)
48 T = Rot_z(u(1))*Trans_z(u(2))*Trans_x(u(3))*Rot_x(u(4));
49 end
50
51 function u = Rot_z(theta)
52 u=[
53     cos(theta) -sin(theta) 0 0;
54     sin(theta) cos(theta) 0 0;
55     0 0 1 0;
56     0 0 0 1];
57 end
58
59 function u=Rot_x(alpha)
60 u=[ 1 0 0 0
61     0 cos(alpha) -sin(alpha) 0;
62     0 sin(alpha) cos(alpha) 0;
63     0 0 0 1];
64 end
65
66 function u=Trans_x(r_n)
67 u=[
68     1 0 0 r_n;
69     0 1 0 0;
70     0 0 1 0;
71     0 0 0 1];
72 end
73
74 function u = Trans_z(d_n)
75 u=[
76     1 0 0 0;
77     0 1 0 0;
78     0 0 1 d_n;
79     0 0 0 1];
80 end
```

E.1.2 Jacobian.m

This script calculates the Jacobian matrix by decoupling translation and rotation.

```
1  %% Initialise and declare
2  clear all
3  close all
4  clc
5  syms q1 q2 q3 q4 q5 roll pitch real
6  q = [q1 q2 q3 q4 q5];
7
8  %% Forward kinematic transformation
9  Tsym = fk(q); % Symbolic transformation matrix
10 Re = Tsym(1:3,1:3); % Rotational transformation
11 pe = Tsym(1:3,4);   % Translational transformation
12
13 %% Orientation reference
14 Rref = rotx(roll)*roty(pitch); % Rotational reference, rotation='xyz'
15 beta = atan2(-Re(2,3),Re(3,3)); % Calculate euler angle beta(roll)
16 gamma = asin(Re(1,3));  % Calculate euler angle gamma(pitch)
17 Orient = [beta; gamma];
18
19 %% Create Jacobian
20 Jp = jacobian(pe,q); % Translational part of Jacobian matrix
21 Jo = jacobian(Orient,q); % Rotational part of Jacobian matrix
22 J = vertcat(Jp,Jo); % Jacobian
```

E.1.3 VibrationAnalysis.m

This script calculates physical properties of the boom when evaluated as a single spring-mass-inertia model under free underdamped vibrations.

```
%% Initialise and declare
clear all
close all
clc
run = 1;
transverse = false;


%% Calculate properties the different boom lengths
for boom_pos = 1:4
    clear input output initial_states origin

    % Load data (parsing of data is handled in another function)
    [input, output, initial_states, origin] = cleanData(run, transverse, boom_pos);
    time = [1:length(input)]./1000;

    % Locate peaks
    [pks,locs]= findpeaks(output(:,5),'MinPeakDistance',3,'MinPeakProminence',(max(output(:,5))-min(output(:,5)))/25);

    % Calculate applied torque
    arm = pdist([origin(1,:);origin(end,:)])+0.3; %Mass tied 30cm farther out than reflector
    if boom_pos == 1
        T = -41*9.81*arm; % 41kg mass for fully extended boom
    else
        T = -82*9.81*arm; % 82kg mass for remaining boom lengths
    end

    % Stiffness
    k_eq(boom_pos) = T/output(1,5);

    % Frequency
    for i = 1:length(pks)-1
        log_dec_data(i) = log(pks(i)/pks(i+1));
    end
    log_dec = mean(log_dec_data);
    damping_ratio = log_dec/sqrt((2*pi)^2+log_dec^2);
    period = mean(diff(locs/1000));
    omega_d(boom_pos) = 1/period*2*pi;
    omega_n(boom_pos) = omega_d(boom_pos)/sqrt(1-damping_ratio^2);

    % Inertia
    J_eq(boom_pos) = k_eq(boom_pos)/omega_n(boom_pos)^2;

    % Damping
    critical_damping = 2*J_eq(boom_pos)*omega_n(boom_pos);
    b_eq(boom_pos) = critical_damping*damping_ratio;

    % Simulate
    [sim_pos,sim_time] = springInertiaDamping(J_eq(boom_pos),k_eq(boom_pos),b_eq(boom_pos),output(1,5),time(end)-1);

end


%% Differential equation
function [pos,time] = springInertiaDamping(J, k, c, theta_0, t_stop)

% Parameters
dt = 0.001;
```

```
N = round(t_stop/dt);
time = linspace(0,t_stop,N);

% Initialise
pos = zeros(N,1);
theta = theta_0;
theta_dot = 0;

% Simulation
for n = 1:N
    theta_ddot = 1/J*(-k*theta-c*theta_dot);
    theta_dot = theta_dot+theta_ddot*dt;
    theta = theta+theta_dot*dt;
    pos(n) = theta;
end
end
```

E.1.4 ImpulseResponse.m

This script tests the state-space model for an impulse response using θ_2 as input.

```
%% Initialise
clear all
close all
clc

%% Specs
J = [10000, 5000, 3000, 1000];
b = [1, 1, 1, 1]*15000;
k = [6, 5, 4, 3]*220000;

%% State-space representation
% System matrix
A = zeros(9,9);
A(1,2) = 1;
A(2,:) = [-(k(1)+k(2))/J(1), -(b(1)+b(2))/J(1), k(2)/J(1),b(2)/J(1),0,0,0,0,k(1)/J
    (1)];
A(3,4) = 1;
A(4,1:6) = [k(2)/J(2),b(2)/J(2),-(k(2)+k(3))/J(2), -(b(2)+b(3))/J(2), k(3)/J(2), b
    (3)/J(2)];
A(5,6) = 1;
A(6,3:8) = [k(3)/J(3), b(3)/J(3),-(k(3)+k(4))/J(3),-(b(3)+b(4))/J(3), k(4)/J(3), b
    (4)/J(3)];
A(7,8) = 1;
A(8,5:8) = [k(4)/J(4), b(4)/J(4), -(k(4))/J(4), -b(4)/J(4)];

% Input
B = [0, b(2)/J(1), 0, 0, 0, 0, 0, 0, 1]'; % For q velocity input
% B = [0, 0, 0, 0, 0, 0, 0, -1/J(4), 0]'; % For torque input

% Output
C = zeros(5,9);
C(1,9) = 1;
C(2,1) = 1;
C(3,3) = 1;
C(4,5) = 1;
C(5,7) = 1;

% Feedthrough
D = zeros(5,1);

sys = ss(A,B,C,D);

%% Simulate response
[Y,T,X] = impulse(sys,15);

```

```
%% Plot
Fig = figure;
for n = 1:5
    subplot(5,1,n)
    plot(T,Y(:,n))
    grid
    if n == 1
        title('\textbf{Impulse response}')
        ylabel('$\theta_2\,\, [rad]$')
    else
        ylabel(strcat('$\psi_{',string(n-1),'}\,\, [rad]$'))
    end
    late(Fig)
    xlim([0 T(end)])
end
xlabel('Time [$s$]')

% Save figure
FigPrint(Fig, 'Impulse_SS', 500, 1.18, 0.97)
```

E.2 Python

E.2.1 kin.py

```
from copy import deepcopy
from numpy import sin, cos, pi, tan, arctan, array, arctan2, square, arcsin,
                                                    savetxt
from math import pi, inf, sqrt, radians


def fk(q):
    # Geometry
    a1 = 0.235
    a2 = 0.355
    a4 = 0.20098
    a5 = 0.345
    d1 = 0.505
    d5 = 0.00837
    d6 = 0.6928

    # DH table
    dh = array([[q[0], d1, -a1, pi / 2],
                [(q[1] + pi / 2), 0, a2, pi / 2],
                [0, q[2], 0, 0],
                [q[3], 0, -a4, pi / 2],
                [q[4], -d5, -a5, -pi / 2],
                [0, d6, 0, 0]])

    # Transformation matrices
    t1 = t_dh(dh[0, :])
    t2 = t_dh(dh[1, :])
    t3 = t_dh(dh[2, :])
    t4 = t_dh(dh[3, :])
    t5 = t_dh(dh[4, :])
    t6 = t_dh(dh[5, :])

    t = t1 @ t2 @ t3 @ t4 @ t5 @ t6

    return t


def t_dh(u):
    a = rot_z(u[0])
    b = trans_z(u[1])
    c = trans_x(u[2])
```

```
    d = rot_x(u[3])

    t = a @ b @ c @ d

    return t


def rot_z(theta):
    u = array([[cos(theta), -sin(theta), 0, 0],
               [sin(theta),  cos(theta), 0, 0],
               [0,                       0, 1, 0],
               [0,                       0, 0, 1]])

    return u


def rot_x(alpha):
    u = array([[1,          0,           0, 0],
               [0, cos(alpha), -sin(alpha), 0],
               [0, sin(alpha),  cos(alpha), 0],
               [0,          0,           0, 1]])

    return u


def trans_x(a_n):
    u = array([[1, 0, 0, a_n],
               [0, 1, 0, 0],
               [0, 0, 1, 0],
               [0, 0, 0, 1]])

    return u


def trans_z(d_n):
    u = array([[1, 0, 0,    0],
               [0, 1, 0,    0],
               [0, 0, 1,  d_n],
               [0, 0, 0,    1]])

    return u


def fk_wrist(q):
    # Geometry
    a4 = 0.20098
    a5 = 0.345
    d5 = 0.00837
    d6 = 0.6928

    # DH table
    dh = array([[q[0],       0, -a4,  pi/2],
               [q[1],      -d5, -a5, -pi/2],
               [0,          d6,   0,     0]])

    # Transformation matrices
    t1 = t_dh(dh[0, :])
    t2 = t_dh(dh[1, :])
    t3 = t_dh(dh[2, :])

    t = t1 @ t2 @ t3

    return t


def fk_boom(q):
    #  Geometry
```

```
    a1 = 0.235
    a2 = 0.355
    d1 = 0.505

    #  DH table
    dh = array([[q[0],                d1,  -a1, pi / 2],
                [(q[1] + pi / 2),      0,   a2, pi / 2],
                [0,                 q[2],    0,      0]])

    # Transformation matrices
    t1 = t_dh(dh[0, :])
    t2 = t_dh(dh[1, :])
    t3 = t_dh(dh[2, :])
    t = t1 @ t2 @ t3

    return t


def ik_wrist(qb, rx, ry):
    t2 = tan(ry / 2)
    t3 = t2 ** 2
    t5 = tan(qb[0] / 2)
    t6 = t5 ** 2
    t7 = t3 * t6
    t11 = tan(qb[1] / 2 + pi / 4)
    t14 = tan(rx / 2)
    t15 = t14 * t5
    t16 = t11 ** 2
    t19 = t14 ** 2
    t20 = t19 * t6
    t21 = t19 * t3
    t24 = t2 * t11
    t25 = 2 * t24
    t28 = 4 * t15 * t11
    t31 = t19 * t2
    t33 = 2 * t31 * t11
    t34 = t2 * t6
    t36 = 2 * t34 * t11
    t37 = t6 * t11
    t39 = 2 * t31 * t37
    t40 = t14 * t3
    t41 = t5 * t11
    t43 = 4 * t40 * t41
    t44 = t19 * t16 + t20 * t16 + t3 * t16 + t7 * t16 + t21 * t6 + t21 - t25 + t28
                                                 - t33 + t36 + t39 - t43 + t6 + 1
    t45 = t6 * t16
    t48 = t21 * t16 + t21 * t45 + t16 + t19 + t20 + t25 - t28 + t3 + t33 - t36 -
                                                t39 + t43 + t45 + t7
    t51 = sqrt(t44 / t48)
    t55 = t45 * t51
    t65 = t16 * t51
    t67 = t19 * t11 + t20 * t11 - t21 * t11 + t7 * t11 - 2 * t15 * t16 + t31 * t16
                                                 - t34 * t16 + t20 * t51 + 2 * t24 *
                                                t51 + t31 * t6 - 2 * t40 * t5 + t7 *
                                                t51 + 2 * t15 - t31 + t34 + t55 + t65
    t79 = t11 * t51
    t88 = t5 * t16
    t92 = -2 * t31 * t37 * t51 + 4 * t40 * t41 * t51 + t3 * t11 - 4 * t15 * t79 +
                                                t2 * t16 + t19 * t51 - t21 * t37 +
                                                t21 * t55 + t21 * t65 + t3 * t51 -
                                                t31 * t45 + 2 * t31 * t79 - 2 * t34 *
                                                 t79 + 2 * t40 * t88 - t11 - t2 - t37
    t94 = t14 * t6
    t96 = t2 * t5
    t108 = t14 * t16 - t40 * t16 - t94 * t16 + 2 * t96 * t16 + 2 * t31 * t5 + 2 *
                                                t31 * t88 + t40 * t45 + t40 * t6 +
                                                t14 - t40 - t94 + 2 * t96
```

```
    t111 = arctan((t67 + t92) / t108)
    q_4 = 2 * t111

    t1 = sin(rx)
    t2 = cos(ry)
    t6 = qb[1] / 2 + pi / 4
    t7 = sin(t6)
    t8 = cos(t6)
    t10 = sin(qb[0])
    t13 = 2 * t1 * t2 * t7 * t8 * t10
    t14 = cos(rx)
    t15 = t14 * t2
    t16 = t8 ** 2
    t18 = 2 * t15 * t16
    t19 = sin(ry)
    t21 = cos(qb[0])
    t24 = 2 * t19 * t7 * t8 * t21
    t29 = sqrt(-(t13 + t18 - t24 - t15 + 1) / (t13 + t18 - t24 - t15 - 1))
    t30 = arctan(t29)
    q_5 = -2 * t30

    q = [q_4, q_5]

    return q


def ik_boom(x, y, z):

    q_1 = arctan2(y, x)
    q_2 = 2*arctan2((200*z*cos(q_1)-101*cos(q_1)+(7369*cos(q_1)**2-40400*z*cos(q_1
                                                  )**2+40000*z**2*cos(
        q_1)**2+18800*x*cos(q_1)+40000*x**2)**(1/2)), (2*(100*x+59*cos(q_1))))-pi/
                                                    2
    q_3 = -(71*cos(q_2)-200*z+101)/(200*sin(q_2))

    q = [q_1, q_2, q_3]

    return q


def ik_iter(cart_ref):
    tol = 1e-4
    max_iter = 10

    x = cart_ref[0]
    y = cart_ref[1]
    z = cart_ref[2]
    nx = cart_ref[3]
    ny = cart_ref[4]
    nz = cart_ref[5]
    counter = 0

    ry = arcsin(nx)
    rx = -arctan2(ny, nz)

    t = deepcopy([x, y, z])
    pos_err_magnitude = inf
    q = []

    while pos_err_magnitude > tol:

        q = ik_boom(t[0], t[1], t[2])
        q[3:] = ik_wrist(q[0:3], rx, ry)
        tf = fk(q)

        pos_err = array(tf[:3, 3] - [x, y, z])
        pos_err_magnitude = sqrt(sum(square(pos_err)))
```

```
        # print("Error after ", counter, " iterations: ", pos_err_magnitude)
        t = t - pos_err
        counter += 1
        if counter > max_iter:
            raise Exception('Failed to converge inverse kinematics after {}
                                                    attempts'.format(max_iter))

    return q


def lim_check(q_ref):
    lim = array([[radians(-65), radians(65)],
                 [radians(-16), radians(57)],
                 [7.012, 15.012],
                 [-pi, pi],
                 [radians(-118), radians(62)]])

    for i in range(len(q_ref)):
        q = q_ref[i]

        if not lim[0, 0] <= q[0] <= lim[0, 1]:
            raise Exception("q1 out of range.")

        if not lim[1, 0] <= q[1] <= lim[1, 1]:
            raise Exception("q2 out of range.")

        if not lim[2, 0] <= q[2] <= lim[2, 1]:
            raise Exception("q3 out of range.")

        if not lim[3, 0] <= q[3] <= lim[3, 1]:
            raise Exception("q4 out of range.")

        if not lim[4, 0] <= q[4] <= lim[4, 1]:
            raise Exception("q5 out of range.")
```

E.2.2 pattern_generator.py

```
from math import floor, pi, sin, cos
from numpy import linspace


def correct_params(arc_length, depth, spacing_desired, vel):
    res = 100

    segments = round(arc_length / spacing_desired)  # return stroke is opposite
                                                    direction if segments is even
    spacing = arc_length / segments
    path_length = (depth - spacing) * (segments+1) + spacing + pi * spacing/2 *
                                                    segments
    n = round(path_length*res)
    t_end = path_length/vel

    return segments, t_end, spacing, n


def path(depth, spacing, vel, t_end, n, segments):

    dt = t_end / n
    t_rise = (depth - spacing) / vel
    t_adv = (spacing * pi) / (2 * vel)
    p_u = t_rise + t_adv

    t_pad = spacing / (2 * vel)
    n_pad = round(t_pad / dt)
    u = n_pad * [0]
    v = list(linspace(0, spacing/2, n_pad))

```

```
    for i in range(n_pad, n - n_pad):
        t = dt * i - t_pad
        period_number_u = floor(t / p_u)
        t_shift = t - period_number_u * p_u

        if t_shift < t_rise:
            u.append(period_number_u * spacing)
            if period_number_u % 2 == 0:
                v.append(vel * t_shift + spacing / 2)
            else:
                v.append(- vel * t_shift + depth - spacing / 2)

        else:
            p = (pi / t_adv) * (t_shift - t_rise) + 3 / 2 * pi
            u.append(spacing / 2 * sin(p) + period_number_u * spacing + spacing /
                                                    2)

            if period_number_u % 2 == 0:
                v.append(spacing / 2 * cos(p) + depth - spacing / 2)
            else:
                v.append(spacing / 2 * cos(p + pi) + spacing / 2)

    for i in range(n_pad):
        u.append(u[-1])

    if segments % 2 == 0:
        a = linspace(v[-1], v[-1] + spacing/2, n_pad)
        for i in range(n_pad):
            v.append(a[i])
    else:
        a = linspace(v[-1], 0, n_pad)
        for i in range(n_pad):
            v.append(a[i])

    return [u, v]
```

E.2.3 surf_trans.py

```
from math import sqrt
from numpy import linspace
from scipy import optimize
from scipy.linalg import norm


def fit(p, *n):
    if n == ():
        n = 1000
    n_reparam = 1000
    x_data = []
    y_data = []
    z_data = []
    d = []
    y_reparam = []
    z_reparam = []
    u_reparam = []
    u_coeffs = []
    u_scaled = linspace(0, 1, n_reparam)
    ut = linspace(0, 1, n)
    chord = []
    du = ut[1] - ut[0]
    parameter = []

    for i in range(len(p)):
        pt = p[i]
        x_data.append(pt[0])
        y_data.append(pt[1])
        z_data.append(pt[2])
```

```
30
31      for i in range(1, len(p)):
32          chord.append(sqrt((z_data[i]-z_data[i-1])**2+(y_data[i]-y_data[i-1])**2))
33
34      chord_arc = sum(chord)
35
36      for i in range(len(chord)):
37          parameter.append(sum(chord[:i])/chord_arc)
38      parameter.append(1)
39
40      y_coeffs, y_cov = optimize.curve_fit(y_func, parameter, y_data)
41      z_coeffs, z_cov = optimize.curve_fit(z_func, parameter, z_data)
42
43      for i in range(n):
44          par = ut[i]
45
46          yen = y_func(par + du, *y_coeffs)
47          ye = y_func(par, *y_coeffs)
48          zen = z_func(par + du, *z_coeffs)
49          ze = z_func(par, *z_coeffs)
50
51          d.append(sqrt((yen-ye)**2+(zen-ze)**2))
52      perimeter = sum(d)
53
54      if y_data[0] < y_data[-1]:
55          direction = 1
56      else:
57          direction = -1
58
59      for i in range(n_reparam):
60          y_reparam.append(y_func(u_scaled[i], *y_coeffs))
61          z_reparam.append(z_func(u_scaled[i], *z_coeffs))
62
63      for i in range(1, n_reparam):
64          u_reparam.append(sqrt((z_reparam[i]-z_reparam[i-1])**2+(y_reparam[i]-
                                                    y_reparam[i-1])**2))
65
66      u_chord_arc = sum(u_reparam)
67
68      for i in range(len(u_reparam)):
69          u_coeffs.append(sum(u_reparam[:i]))
70      u_coeffs.append(u_chord_arc)
71
72      u_coeffs, u_cov = optimize.curve_fit(u_func, u_coeffs, u_scaled)
73
74      return perimeter, x_data[0], [y_coeffs, z_coeffs, u_coeffs, direction]
75
76
77  def y_func(x, a, b, c, d):
78      return a * x**3 + b * x**2 + c * x + d
79
80
81  def y_func_diff(x, a, b, c, d):
82      return 3 * a * x**2 + 2 * b * x + c
83
84
85  def z_func(x, a, b, c, d, e):
86      return a * x**4 + b * x**3 + c * x**2 + d * x + e
87
88
89  def z_func_diff(x, a, b, c, d, e):
90      return 4 * a * x**3 + 3 * b * x**2 + 2 * c * x + d
91
92
93  def u_func(x, a, b, c, d, e, f):
94      return a * x**5 + b * x**4 + c * x**3 + d * x**2 + e * x + f
95
```

```
96
97  def surf(uv_list, params, offset):
98      u = uv_list[0]
99      v = uv_list[1]
100     y_coeffs = params[0]
101     z_coeffs = params[1]
102     u_coeffs = params[2]
103     direction = params[3]
104     x = []
105     y = []
106     z = []
107     nx = []
108     ny = []
109     nz = []
110
111     for i in range(len(u)):
112         par = u_func(u[i], *u_coeffs)
113         x.append(v[i] + offset)
114         y.append(y_func(par, *y_coeffs))
115         z.append(z_func(par, *z_coeffs))
116
117         x_diff = 0
118         y_diff = y_func_diff(par, *y_coeffs)
119         z_diff = z_func_diff(par, *z_coeffs)
120
121         scale = -direction*norm([x_diff, y_diff, z_diff])
122
123         nx.append(0)
124         ny.append(z_diff/scale)
125         nz.append(-y_diff/scale)
126
127     return [x, y, z, nx, ny, nz]
```

E.2.4 main.py

```
1   import sys, can, struct, traceback
2   from time import time, sleep
3   from math import pi, radians
4   import numpy as np
5   from numpy import savetxt
6   import main_window
7   import surf_trans as pp
8   import pattern_generator as pg
9   import kin
10  from PyQt5.QtWidgets import QMessageBox, QMainWindow, QApplication,
                                            QListWidgetItem
11  from PyQt5.QtCore import QThread, pyqtSignal, QTimer, QAbstractEventDispatcher, Qt
12  from multiprocessing import Queue, freeze_support, cpu_count, set_start_method,
                                            get_context
13  from queue import Empty
14
15
16  def generate_pattern(param, point_cloud, cart_queue, dt_queue):
17
18      print("Params", param)
19      depth = param[0]
20      stroke_spacing_des = param[1]
21      vel = param[2]
22
23      d, offset, params = pp.fit(point_cloud)  # (self.point_cloud)
24
25      segments, t_end, stroke_spacing, n = pg.correct_params(d, depth,
                                                stroke_spacing_des, vel)
26      uv_path = pg.path(depth, stroke_spacing, vel, t_end, n, segments)
27
28      cart_ref = pp.surf(uv_path, params, offset)
29
```

```
    dt = t_end/n
    print("t_end: ", t_end)
    return cart_queue.put(cart_ref), dt_queue.put(dt)


class Main(QMainWindow, main_window.Ui_MainWindow):

    # Static values
    vel_min = 0.05
    vel_max = 0.7
    stroke_spacing_min = 0.1
    stroke_spacing_max = 1.5
    depth_min = 2
    depth_max = 8

    def __init__(self, parent=None):
        super(Main, self).__init__(parent)
        self.setupUi(self)

        # Correct this later
        dummy_points = [[8.0, 4.7453950103830715, 1.0],
                        [8.3, 3.966736204580941, 5.5543035725747485],
                        [7.8, 0.9811185945264055, 7.326604185118965],
                        [8.0, -3.6612492782151276, 5.901951891186886],
                        [8.1, -4.7453950103830715, 1.0]]
        # dummy_points.reverse()

        self.knot_points = []
        for i in range(5):
            pos = dummy_points[i]
            self.knot_point_list.addItem(QListWidgetItem(str("X: {:.2f} Y: {:.2f} Z: "
                                                             "{:.2f}".format(pos[0],
            self.knot_points.append(pos)


        # Multiprocessing
        self.cart_ref_out = Queue()
        self.ikin_out = Queue()
        self.mode_flag = Queue()
        self.param_queue = Queue()
        self.point_queue = Queue()
        self.dt_queue = Queue()

        # Plot
        self.plot_widget.canvas.ax.mouse_init()

        # Methods
        self.mythread = MultiprocessThread(self.cart_ref_out, self.ikin_out, self.
                                           mode_flag, self.param_queue,
                                           self.point_queue, self.dt_queue)
        self.mythread.cart_done.connect(self.done_cart)
```

```
80          self.mythread.ik_done.connect(self.done_ik)
81          self.mythread.progress.connect(self.update_progressbar)
82          self.mythread.start()
83
84          # Update KPI
85          self.timer = QTimer()
86          self.timer.timeout.connect(self.update_kpi)
87          self.timer.start(200)
88
89          # Initialise variables
90          self.slider_velocity()
91          self.slider_stroke_spacing()
92          self.slider_depth()
93          self.knot_x = []
94          self.knot_y = []
95          self.knot_z = []
96          self.cart_ref = []
97          self.q_ref = []
98          self.status = "Idle"
99          self.can_status = "Not initialised"
100         self.dt = 0
101
102         # Buttons
103         self.ik_btn.clicked.connect(self.start_ik)
104         self.cart_btn.clicked.connect(self.start_pattern)
105         self.toggle_can_btn.clicked.connect(self.toggle_can)
106         self.remove_btn.clicked.connect(self.remove_knot_point)
107         self.reset_btn.clicked.connect(self.reset_all)
108         self.spray_btn.clicked.connect(self.init_spray)
109
110         # Sliders
111         self.velocitySlider.sliderMoved.connect(self.slider_velocity)
112         self.stroke_spacing_slider.sliderMoved.connect(self.slider_stroke_spacing)
113         self.depth_slider.sliderMoved.connect(self.slider_depth)
114
115         # CAN interface
116         self.connect_can()
117
118     def closeEvent(self, event):
119         reply = QMessageBox.question(self, 'Message', "Are you sure to quit?",
                                                 QMessageBox.Yes | QMessageBox.No,
120                                      QMessageBox.No)
121
122         if reply == QMessageBox.Yes:
123             self.can_bus.shutdown()
124         else:
125             event.ignore()
126
127     def toggle_can(self):
128         if self.can_status == "Connected":
129             self.disconnect_can()
130         else:
131             self.connect_can()
132
133     def connect_can(self):
134         self.can_bus = CANInterface()
135         try:
136             self.can_bus.start()
137             self.can_bus.joy_btn.connect(self.add_knot_point)
138             self.can_indicator.setStyleSheet('color: #eff0f1')
139             self.can_status = "Connected"
140             self.can_bus.progress.connect(self.update_progressbar)
141             self.can_bus.spray_error.connect(self.spray_error)
142             self.can_bus.spray_done.connect(self.spray_done)
143             self.can_bus.can_error.connect(self.can_error)
144
145         except:
```

```
146             self.can_indicator.setStyleSheet('color: #ff0000')
147             self.can_status = "Error"
148             QMessageBox.information(self, "Warning", "CAN interface cannot connect
                                                        ")
149
150     def disconnect_can(self):
151         self.can_bus.shutdown()
152         self.can_indicator.setStyleSheet('color: #eff0f1')
153         self.can_status = "Disconnected"
154
155     def plot_path(self):
156         self.plot_widget.canvas.ax.clear()
157         self.plot_widget.canvas.ax.set_xlabel('X [m]')
158         self.plot_widget.canvas.ax.set_ylabel('Y [m]')
159         self.plot_widget.canvas.ax.set_zlabel('Z [m]')
160         self.plot_widget.canvas.ax.set_aspect('equal')
161
162
163
164         # Aspect ratio correction
165         max_range = np.array(
166             [np.array(self.cart_ref[0]).max() - np.array(self.cart_ref[0]).min(),
167              np.array(self.cart_ref[1]).max() - np.array(self.cart_ref[1]).min(),
168              np.array(self.cart_ref[2]).max() - np.array(self.cart_ref[2]).min()])
                                                            .max()
169
170         xb = 0.5 * max_range * np.mgrid[-1:2:2, -1:2:2, -1:2:2][0].flatten() + 0.5
                                                            * \
171             (np.array(self.cart_ref[0]).max() + np.array(self.cart_ref[0]).min())
172
173         yb = 0.5 * max_range * np.mgrid[-1:2:2, -1:2:2, -1:2:2][1].flatten() + 0.5
                                                            * \
174             (np.array(self.cart_ref[1]).max() + np.array(self.cart_ref[1]).min())
175
176         zb = 0.5 * max_range * np.mgrid[-1:2:2, -1:2:2, -1:2:2][2].flatten() + 0.5
                                                            * \
177             (np.array(self.cart_ref[2]).max() + np.array(self.cart_ref[2]).min())
178
179         for a, b, c in zip(xb, yb, zb):
180             self.plot_widget.canvas.ax.plot([a], [b], [c], 'w')
181
182         self.plot_widget.canvas.ax.plot3D(self.cart_ref[0],
183                                           self.cart_ref[1],
184                                           self.cart_ref[2], c='yellow')
185         self.plot_widget.canvas.ax.scatter(self.knot_x, self.knot_y, self.knot_z,
                                                        c='#DD0000')
186
187         self.plot_widget.canvas.draw()
188
189     def start_pattern(self):
190         self.update_progressbar(0)
191         # Clear parameter queue
192         try:
193             while True:
194                 self.param_queue.get_nowait()
195         except Empty:
196             pass
197
198         self.param_queue.put([self.depth, self.stroke_spacing_des, self.velocity])
199
200         if len(self.knot_points) >= 5:  # Adjust to maximal number of coefficients
                                                        in parametric functions
201             self.ik_btn.setEnabled(False)
202             self.cart_btn.setEnabled(False)
203             self.reset_btn.setEnabled(False)
204             self.remove_btn.setEnabled(False)
205             self.spray_btn.setEnabled(False)
```

```
206
207             self.status = "Parametrising surface"
208             self.status_indicator.setStyleSheet('color: #ffff00')
209
210             self.q_ref = []
211             # Clear output queue
212             try:
213                 while True:
214                     self.cart_ref_out.get_nowait()
215             except Empty:
216                 pass
217
218             # Clear flag queue
219             try:
220                 while True:
221                     self.mode_flag.get_nowait()
222             except Empty:
223                 pass
224
225             self.point_queue.put(self.knot_points)
226             self.param_queue.put([self.depth, self.stroke_spacing_des, self.
                                                    velocity])
227             self.mode_flag.put("Path Gen")
228         else:
229             QMessageBox.information(self, "Warning", "Collect 5 or more knot
                                                    points")
230
231     def done_cart(self):
232         try:
233             self.cart_ref = self.cart_ref_out.get(True)
234             self.dt =  self.dt_queue.get(True)
235             self.ik_btn.setEnabled(True)
236             self.cart_btn.setEnabled(True)
237             self.reset_btn.setEnabled(True)
238             self.remove_btn.setEnabled(True)
239             self.spray_btn.setEnabled(True)
240             print("Number of points: ", len(self.cart_ref[1]))
241             self.progressBar.setMaximum(len(self.cart_ref[1]))
242             print("Cartesian reference stored in variable")
243             self.plot_path()
244             self.status = "Surface generated"
245             self.status_indicator.setStyleSheet('color: #eff0f1')
246
247             # savetxt('cart_ref.txt', np.array(self.cart_ref), delimiter='  ')
248
249         except Empty:
250             print("No Cartesian reference returned from thread!")
251
252     def start_ik(self):
253         if len(self.cart_ref) == 0:
254             QMessageBox.information(self, "Warning", "Create spraying plan before
                                                    calculating inverse
                                                    kinematics")
255         elif len(self.q_ref) > 0:
256             QMessageBox.information(self, "Warning", "Inverse kinematics already
                                                    calculated")
257         else:
258             self.ik_btn.setEnabled(False)
259             self.cart_btn.setEnabled(False)
260             self.reset_btn.setEnabled(False)
261             self.remove_btn.setEnabled(False)
262             self.spray_btn.setEnabled(False)
263
264             self.status = "Calculating inverse kinematics"
265             self.status_indicator.setStyleSheet('color: #ffff00')
266
267             # Clear output queue
```

```
            try:
                while True:
                    self.ikin_out.get_nowait()
            except Empty:
                pass

            # Clear flag queue
            try:
                while True:
                    self.mode_flag.get_nowait()
            except Empty:
                pass

            self.mode_flag.put("Inverse Kinematics")

    def done_ik(self):
        self.ik_btn.setEnabled(True)
        self.cart_btn.setEnabled(True)
        self.reset_btn.setEnabled(True)
        self.remove_btn.setEnabled(True)
        self.spray_btn.setEnabled(True)

        try:
            self.q_ref = self.ikin_out.get(True, 0.1)

            # print("Shape of ikin", np.shape(np.array(self.q_ref)))

            # savetxt('q_ref.txt', np.array(self.q_ref), delimiter='  ')

            print("Joint space reference stored in variable")
            self.status = "Inverse kinematics calculated"
            self.status_indicator.setStyleSheet('color: #eff0f1')

            try:
                kin.lim_check(self.q_ref)
            except Exception as e:
                QMessageBox.information(self, "Error", "{} Reposition vehicle!".
                                                        format(e))
                self.reset_all()

        except Empty:
            print("No joint space reference returned from thread!")

    def slider_velocity(self):
        self.velocity = self.vel_min + self.velocitySlider.value()/100 * (self.
                                                vel_max - self.vel_min)
        self.vel_disp.setText("Velocity: {:.2f} [m/s]".format(self.velocity))

    def slider_stroke_spacing(self):
        self.stroke_spacing_des = self.stroke_spacing_min + self.
                                                stroke_spacing_slider.value() /
                                                100 * \
                                  (self.stroke_spacing_max - self.
                                                                    stroke_spacing_mi
                                                                    )
        self.stroke_spacing_label.setText("Stroke spacing: {:.2f} [m]".format(self
                                                .stroke_spacing_des))

    def slider_depth(self):
        self.depth = self.depth_min + self.depth_slider.value()/100 * (self.
                                                depth_max - self.depth_min)
        self.depth_label.setText("Depth: {:.2f} [m]".format(self.depth))

    def add_knot_point(self):
        self.can_bus.joy_btn.disconnect(self.add_knot_point)
        pose = kin.fk(self.can_bus.sensor_values)
        pos = list(pose[:3, 3])
```

```
327             self.knot_points.append(pos)
328
329             # Plot
330             self.plot_widget.canvas.ax.scatter(pos[0], pos[1], pos[2], c='#DD0000')
331             self.plot_widget.canvas.draw()
332             self.plot_widget.canvas.ax.mouse_init()
333
334             # Add to list in GUI
335             self.knot_point_list.addItem(QListWidgetItem(str("X: {:.2f} Y: {:.2f} Z: "
336                                                              "{:.2f}".format(pos[0],
337
338             # Cool down
339             QTimer.singleShot(1000, self.connect_knot_signal)  # Increase timeout
                                                                   later
340
341     def connect_knot_signal(self):
342         self.can_bus.joy_btn.connect(self.add_knot_point)
343
344     def remove_knot_point(self):
345         row = self.knot_point_list.currentRow()
346         item = self.knot_point_list.item(row)
347
348         if item is not None:
349             self.knot_point_list.takeItem(row)
350             del self.knot_points[row]
351
352         else:
353             QMessageBox.information(self, "Info", "Select which knot point to
                                                       delete")
354
355     def update_kpi(self):
356         pose = kin.fk(self.can_bus.sensor_values)
357         pos = list(pose[:3, 3])
358
359         # print("Sensor values: {}".format(self.can_bus.sensor_values))
360
361         self.x_pos.setText("{:.2f} [m]".format(pos[0]))
362         self.y_pos.setText("{:.2f} [m]".format(pos[1]))
363         self.z_pos.setText("{:.2f} [m]".format(pos[2]))
364
365         self.status_indicator.setText(self.status)
366         self.can_indicator.setText(self.can_status)
367
368         if self.knot_points != []:
369             self.knot_x, self.knot_y, self.knot_z = zip(*self.knot_points)
370         else:
371             self.knot_x = []
372             self.knot_y = []
373             self.knot_z = []
374
375         if len(self.cart_ref) == 0:
376             scale = 2
```

```
            self.plot_widget.canvas.ax.clear()
            self.plot_widget.canvas.ax.set_xlabel('X [m]')
            self.plot_widget.canvas.ax.set_ylabel('Y [m]')
            self.plot_widget.canvas.ax.set_zlabel('Z [m]')
            self.plot_widget.canvas.ax.set_aspect('equal')
            self.plot_widget.canvas.ax.scatter(self.knot_x, self.knot_y, self.
                                                    knot_z, c='red')
            self.plot_widget.canvas.ax.set_xlim3d(-5.2, 21.54)
            self.plot_widget.canvas.ax.set_ylim3d(-13.37, 13.37)
            self.plot_widget.canvas.ax.set_zlim3d(-8.81, 17.94)
        else:
            scale = 1
            self.plot_path()

        self.plot_widget.canvas.ax.quiver(pose[0][3], pose[1][3], pose[2][3],
                                          pose[0][2] * scale, pose[1][2] * scale,
                                                                                  pose
                                                                                  [
                                                                                  2
                                                                                  ]
                                                                                  [
                                                                                  2
                                                                                  ]

                                                                                  *

                                                                                  scale
                                                                                  ,

                                          color=[0.3, 1, 0.2], pivot='tip')
        self.plot_widget.canvas.draw()

    def check_q(self):
        print("Length q:", len(self.mythread.q))

    def update_progressbar(self, value):
        self.progressBar.setValue(value)

    def reset_all(self):
        reply = QMessageBox.question(self, 'Message', "Do you want to reset all
                                                    parameters?", QMessageBox.Yes
                                     | QMessageBox.No, QMessageBox.No)
        if reply == QMessageBox.Yes:
            self.knot_points = []
            self.knot_point_list.clear()
            self.cart_ref = []
            self.q_ref = []
            self.depth_slider.setSliderPosition(50)
            self.stroke_spacing_slider.setSliderPosition(50)
            self.velocitySlider.setSliderPosition(50)
            self.slider_velocity()
            self.slider_stroke_spacing()
            self.slider_depth()
            self.status_indicator.setStyleSheet('color: #eff0f1')
            self.status = "Idle"
            self.update_progressbar(0)
        else:
            pass

    def init_spray(self):
        if len(self.q_ref) == 0:
            QMessageBox.information(self, "Warning", "Calculate inverse kinematics
                                                     before spraying")
        else:
            reply = QMessageBox.question(self, 'Message', "Move to initial
                                                    position?", QMessageBox.Yes
                                     | QMessageBox.No, QMessageBox.No)
```

```
            if reply == QMessageBox.Yes:
                self.ik_btn.setEnabled(False)
                self.cart_btn.setEnabled(False)
                self.reset_btn.setEnabled(False)
                self.toggle_can_btn.setEnabled(False)
                self.remove_btn.setEnabled(False)
                self.spray_btn.setEnabled(False)

                self.can_bus.joy_btn.disconnect(self.add_knot_point)
                self.spray_btn.clicked.disconnect(self.init_spray)
                self.can_bus.counter = 0

                print("q_ref", self.q_ref[0])

                self.can_bus.send_data(self.q_ref[0])
                self.can_bus.joy_btn.connect(self.run_machine)
                self.status = "Waiting for button press"
                self.status_indicator.setStyleSheet('color: #ffff00')
            else:
                pass

    def pause_spray(self):
        self.can_bus.stop_spray()
        self.can_bus.joy_btn.disconnect(self.pause_spray)
        self.can_bus.manual_mode.disconnect(self.pause_spray)

        reply = QMessageBox.question(self, 'Message', "Resume spraying? Remember
                                                    to enable automatic mode",
                                                    QMessageBox.Yes
                                        | QMessageBox.No, QMessageBox.No)
        if reply == QMessageBox.Yes:
            self.can_bus.timer(self.dt)
            self.can_bus.joy_btn.connect(self.pause_spray)
            self.can_bus.manual_mode.connect(self.pause_spray)
        else:
            self.spray_btn.clicked.disconnect(self.pause_spray)
            self.can_bus.joy_btn.connect(self.add_knot_point)
            self.spray_btn.setText("Spray")
            self.spray_btn.clicked.connect(self.init_spray)
            self.update_progressbar(0)
            self.can_bus.counter = 0
            self.status = "Inverse kinematics calculated"
            self.status_indicator.setStyleSheet('color: #eff0f1')
            self.ik_btn.setEnabled(True)
            self.cart_btn.setEnabled(True)
            self.reset_btn.setEnabled(True)
            self.toggle_can_btn.setEnabled(True)
            self.remove_btn.setEnabled(True)

    def run_machine(self):
        self.can_bus.joy_btn.disconnect(self.run_machine)
        self.spray_btn.clicked.connect(self.pause_spray)
        self.spray_btn.setEnabled(True)
        self.spray_btn.setText("Pause")
        self.status = "Spraying"
        self.can_bus.q_ref = self.q_ref
        self.can_bus.timer(self.dt)
        QTimer.singleShot(1000, self.joy_pause)

    def joy_pause(self):
        self.can_bus.joy_btn.connect(self.pause_spray)
        self.can_bus.manual_mode.connect(self.pause_spray)

    def spray_done(self):
        print("Done spraying!")
        QMessageBox.information(self, "Info", "Done spraying")
        self.status = "Done spraying"
```

```
492             self.can_bus.joy_btn.disconnect(self.pause_spray)
493             self.can_bus.joy_btn.connect(self.add_knot_point)
494             self.spray_btn.clicked.disconnect(self.pause_spray)
495             self.spray_btn.clicked.connect(self.init_spray)
496             self.spray_btn.setText("Spray")
497             self.ik_btn.setEnabled(True)
498             self.cart_btn.setEnabled(True)
499             self.reset_btn.setEnabled(True)
500             self.toggle_can_btn.setEnabled(True)
501             self.remove_btn.setEnabled(True)
502             self.reset_all()
503 
504         def spray_error(self):
505             self.status = "Spraying aborted"
506             self.status_indicator.setStyleSheet('color: #ff0000')
507             QMessageBox.information(self, "Warning", "Error in spraying!")
508             self.spray_btn.setText("Spray")
509             self.can_bus.joy_btn.disconnect(self.pause_spray)
510             self.can_bus.joy_btn.connect(self.add_knot_point)
511             self.spray_btn.clicked.disconnect(self.pause_spray)
512             self.spray_btn.clicked.connect(self.init_spray)
513             self.ik_btn.setEnabled(True)
514             self.cart_btn.setEnabled(True)
515             self.reset_btn.setEnabled(True)
516             self.toggle_can_btn.setEnabled(True)
517             self.remove_btn.setEnabled(True)
518 
519         def can_error(self):
520             self.can_indicator.setStyleSheet('color: #ff0000')
521             self.can_status = "Error"
522             QMessageBox.information(self, "Warning", "Error in CAN bus")
523 
524 
525 class MultiprocessThread(QThread):
526 
527     cart_done = pyqtSignal(bool)
528     ik_done = pyqtSignal(bool)
529     progress = pyqtSignal(int)
530 
531     def __init__(self, cart_ref_out, ikin_out, mode_flag, param_queue, point_q,
                                                    dt_queue):  # mode_flag
532         super(MultiprocessThread, self).__init__()
533         self.cartQ = cart_ref_out
534         self.ikinQ = ikin_out
535         self.flag = mode_flag
536         self.param = param_queue
537         self.flag.put("Idle Mode")
538         self.cart_ref = []
539         self.q_ref = []
540         self.n = cpu_count()
541         self.point_cloud_q = point_q
542         self.dt_queue = dt_queue
543         freeze_support()
544         print("Multiprocess object initialised")
545 
546     def run(self):
547         while True:
548             flag = self.flag.get(block=True)
549             print("Multiprocessing mode: {}".format(str(flag)))
550             if flag == "Path Gen":
551                 try:
552                     param = self.param.get(block=True)
553                 except Empty:
554                     print("Parameter q empty")
555                     break
556                 point_cloud = self.point_cloud_q.get(block=True)
557 
```

```
558                 self.pattern_process = get_context("spawn").Process(target=
                                                        generate_pattern, args=(
                                                        param, point_cloud, self.
                                                        cartQ, self.dt_queue,))
559                 self.pattern_process.start()
560                 try:
561                     self.cart_ref = self.cartQ.get(block=True)
562                 except Empty:
563                     print("CartQ is Empty!")
564                     break
565 
566                 self.cartQ.put(self.cart_ref)
567                 self.flag.put("Idle Mode")
568                 self.pattern_process.join()
569                 self.cart_done.emit(True)
570 
571             if flag == "Inverse Kinematics":
572                 if len(self.cart_ref) is 0:
573                     print("Provide Cartesian Path")
574                     pass
575                 else:
576                     prog = 0
577                     with get_context("spawn").Pool(processes=self.n) as self.pool:
                                                            # must be guarded
                                                        with self
578                         obj = zip(*self.cart_ref)
579                         self.cart_ref.clear()
580                         res = self.pool.imap(kin.ik_iter, obj)
581                         for i in res:
582                             self.q_ref.append(i)
583                             prog += 1
584                             self.progress.emit(prog)
585 
586                         self.ikinQ.put(self.q_ref)
587                         sleep(.1)  # need sleep, or queue not ready
588 
589                         self.q_ref.clear()
590                         self.flag.put("Idle Mode")
591                         self.pool.close()
592                         self.pool.join()
593                         self.ik_done.emit(True)
594                         self.progress.emit(0)
595 
596             if flag == "stop":
597                 break
598 
599 
600 class CANInterface(QThread):
601     id_long = 0x183
602     id_short = 0x283
603     id_button = 0x383
604     id_pc = 0x2
605 
606     # Offsets and multipliers
607     offsets = [-32768, -14364, 57442, -32678, -42962]
608     multipliers = [3.46210941710301e-05, 1.94410759575784e-05, 0.
                                            000122070312500000, 9.58737992428526e
                                            -05,
609                    4.79368996214263e-05]
610 
611     joy_btn = pyqtSignal(bool)
612     progress = pyqtSignal(int)
613     spray_error = pyqtSignal(bool)
614     spray_done = pyqtSignal(bool)
615     can_error = pyqtSignal(bool)
616     manual_mode = pyqtSignal(bool)
617 
```

```
618     def __init__(self):
619         QThread.__init__(self)
620         self.bus = can.Bus(bustype='kvaser', channel=0, bitrate=1000000)
621         print("CAN interface created")
622 
623         self.id_f1 = self.id_pc + 0x180
624         self.id_f2 = self.id_pc + 0x280
625 
626         self.sensor_values = [0, 0, 0, 0, 0]
627 
628         self.Btn = False
629         self.controller_off = False
630 
631         self.update_sensors = False
632         self.q_ref = []
633 
634         self.counter = 0
635         self.dt_timer = QTimer(singleShot=False, timerType=Qt.PreciseTimer)
636 
637     def __del__(self):
638         self.wait()
639 
640     def run(self):
641         self.update_sensors = True
642         self.start()
643         try:
644             while self.update_sensors:
645                 msg = self.bus.recv(0.1)
646                 # for msg in self.bus:
647                 if msg.arbitration_id == self.id_short:
648                     frame_2 = struct.unpack('!H', msg.data)
649                     self.sensor_values[4] = self.multipliers[4] * (frame_2[0] +
                                                                    self.offsets[4])
650 
651                 if msg.arbitration_id == self.id_long:
652                     frame_1 = struct.unpack('!HHHH', msg.data)
653                     self.sensor_values[0] = self.multipliers[0] * (frame_1[0] +
                                                                    self.offsets[0])
654                     self.sensor_values[1] = self.multipliers[1] * (frame_1[1] +
                                                                    self.offsets[1])
655                     self.sensor_values[2] = self.multipliers[2] * (frame_1[2] +
                                                                    self.offsets[2])
656                     self.sensor_values[3] = self.multipliers[3] * (frame_1[3] +
                                                                    self.offsets[3])
657 
658                 if msg.arbitration_id == self.id_button:
659                     frame_3 = struct.unpack('!B', msg.data)
660 
661                     if frame_3[0] == 1:
662                         self.joy_btn.emit(True)
663                     elif frame_3[0] == 2:
664                         self.manual_mode.emit(True)
665                     elif frame_3[0] == 3:
666                         self.joy_btn.emit(True)
667                         self.manual_mode.emit(True)
668 
669         except Exception:
670             print("Exception in CAN receive")
671             self.can_error.emit(True)
672 
673     def shutdown(self):
674         self.update_sensors = False
675         QTimer.singleShot(1000, self.bus.shutdown)
676         print("CAN interface disconnected")
677 
678     def timer(self, dt):
679         self.dt_timer.timeout.connect(self.msg_send)
```

```
680         self.dt_timer.start(dt*700)
681
682     def send_data(self, data):
683
684         data_int = []
685         for i in range(len(data)):
686             num = int(round(data[i] / self.multipliers[i] - self.offsets[i]))
687
688             if num < 0 or num > 2**16:
689                 raise Exception("Data conversion out of range")
690             data_int.append(num)
691
692         data_long = struct.pack('!HHHH', *data_int[0:4])
693         data_short = struct.pack('!H', data_int[4])
694
695         msg_long = can.Message(data=data_long, arbitration_id=self.id_f1,
                                                    is_extended_id=False)
696         msg_short = can.Message(data=data_short, arbitration_id=self.id_f2,
                                                    is_extended_id=False)
697
698         try:
699             self.bus.send(msg_long, timeout=0.1)
700             self.bus.send(msg_short, timeout=0.1)
701             # print("Message sent {}".format(msg_long))
702         except can.CanError:
703             self.dt_timer.timeout.disconnect(self.msg_send)
704             print("x0 message NOT sent at: ", self.counter)
705             self.spray_error.emit(True)
706
707     def msg_send(self):
708         if self.counter >= len(self.q_ref):
709             self.progress.emit(0)
710             self.dt_timer.timeout.disconnect(self.msg_send)
711             self.spray_done.emit(True)
712         else:
713             self.send_data(self.q_ref[self.counter])
714             self.progress.emit(self.counter)
715             self.counter += 1
716
717     def stop_spray(self):
718         self.dt_timer.timeout.disconnect(self.msg_send)
719
720 def main():
721     app = QApplication(sys.argv)
722     set_start_method('spawn')
723     form = Main()
724     # form.showFullScreen()
725     form.show()
726     sys.exit(app.exec_())
727
728
729 if __name__ == '__main__':
730     main()
```