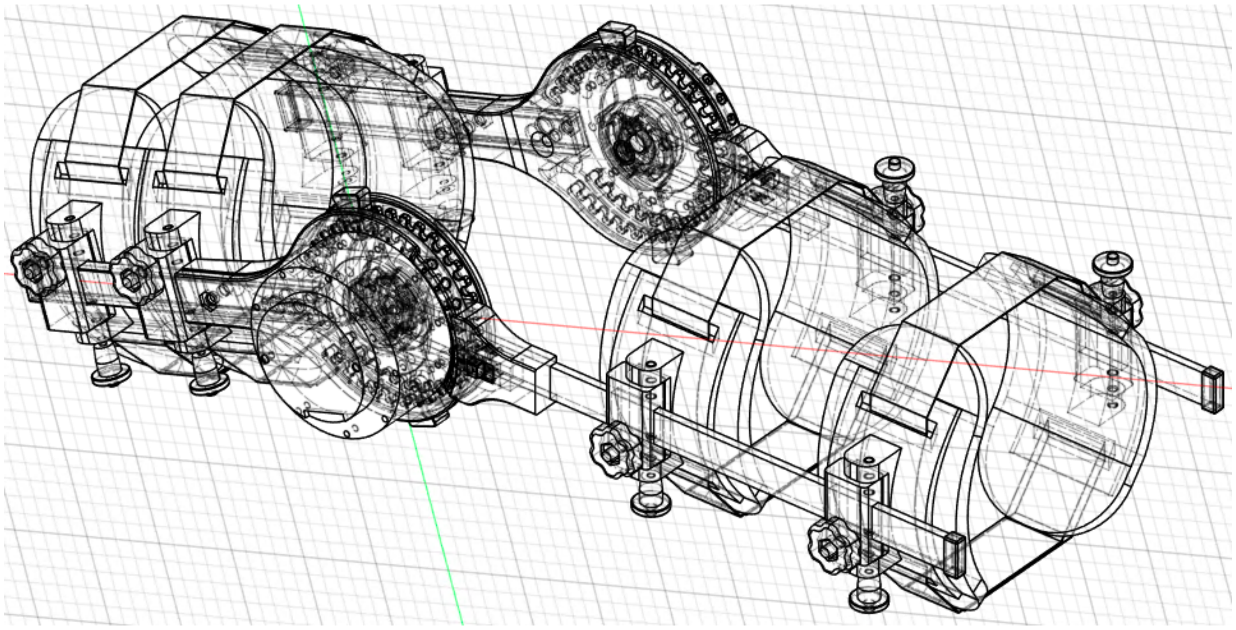


Adaptive Exoskeleton for Knee Injury Rehabilitation

Further Development of Control and Sensor System Integration in a Single-Joint Exoskeleton Concept

Albin Backman, Olle Bukowski, Theo Gustavsson,
Erika Ivarsson, David Ljungqvist, Olivia Sare

Department of Electrical Engineering

Chalmers University of Technology
Gothenburg 2025
www.chalmers.se

Bachelor Thesis 2025

Adaptive Exoskeleton for Knee Injury Rehabilitation

Further Development of Control and Sensor System Integration in a Single-Joint Exoskeleton Concept

Albin Backman, Olle Bukowski, Theo Gustavsson,
Erika Ivarsson, David Ljungqvist, Olivia Sare

Department of Electrical Engineering
Chalmers University of Technology
Gothenburg 2025

Adaptive Exoskeleton for Knee Injury Rehabilitation
Further Development of Control and Sensor System Integration
in a Single-Joint Exoskeleton Concept

Albin Backman, Olle Bukowski, Theo Gustavsson,
Erika Ivarsson, David Ljungqvist, Olivia Sare

Supervisor: Fabian Just
Examiner: Emanuel Dean

Bachelor Thesis 2025
Department of Electrical Engineering
Chalmers University of Technology
SE-412 96 Gothenburg

Cover Picture: Computer Aided Design of the projects Exoskeleton.

Written in LaTeX
Gothenburg 2025

Acknowledgment

We would like to extend our gratitude to our supervisor, Fabian Just, as well as his master's students Yuhong Zhou, Domenico Caliandro, and Pengcheng Shen. We are also thankful to our examiner, Emanuel Dean. Without their guidance, this project would not have been possible, and without their positive and enthusiastic attitude, it would not have been nearly as enjoyable to carry out.

We also want to thank research engineer Rikard Karlsson and students Gustav Bardun, Johan Baumann, Filip Djäknegren and many others active in the C.A.S.E association, who have always met us with a smile and pointed us in the right direction when we either have needed a specific cable or have felt lost.

Abstract

This project presents the continued development of a single-joint exoskeleton intended for knee rehabilitation. The overall goal was to contribute to more effective, individualized rehabilitation by enabling an exoskeleton system that is able to provide transparent, torque-based assistance and track limb motion in real time. Such a system could ultimately reduce recovery time, support at-home therapy and improve the quality of care.

Moving towards this goal, the project focused on introducing sensing and control capabilities. A ROS2-based software architecture was implemented to coordinate real-time communication between hardware components. The actuator was successfully integrated via CAN-bus and operated under open-loop control. However, torque feedback had to be estimated indirectly using current measurements as no dedicated torque sensor was available. The IMU delivered usable quaternion data, but filtering and sensor fusion were not fully completed due to time constraints.

While a fully functional control system was not achieved, the system now provides real-time data flow and basic motion control. The final framework offers a solid technical foundation for future work on further IMU- and actuator-integration, feedback control and clinical applications.

Sammanfattning

Detta projekt behandlar vidareutvecklingen av ett exoskelett för knärehabilitering i en frihetsgrad. Att möjliggöra individualiserad och effektiv terapi genom ett system som kan ge transparent, vridmomentsbaserat stöd och följa benets rörelser i realtid är det slutgiltiga målet. Ett sådant system kan i framtiden bidra till kortare återhämtningstider och förbättrad vårdkvalitet.

Projektet fokuserar på att introducera sensor- och styrfunktioner. En ROS2-baserad programvaruarkitektur implementerades för att koordinera realtidskommunikation mellan implementerad hårdvara. Aktuatorn integrerades framgångsrikt och drivs genom öppen loopstyrning. Motorns moment uppskattades dock med hjälp av strömmätningar då aktuatorn inte hade en inbyggd momentsensor. Kvaterniondata togs fram ur IMU-sensorn, men filtrering och sensorfusion slutfördes inte helt på grund av tidsbegränsningar.

Även om ett fullt fungerande styrsystem inte uppnåddes, finns en struktur för realtidskontroll. Slutstrukturen ger en solid teknisk grund för framtida arbete med IMU- och aktuatorkontroll samt framtida kliniska tillämpningar.

Keywords: Exoskelett, Rehabilitering, ROS2, Momentkontroll, IMU, Transparens, Realtidskontroll

Contents

1 Introduction

Rehabilitation plays a crucial role in societies with advanced healthcare systems, such as Sweden, serving as a cornerstone for improving the quality of life for individuals with physical impairments and chronic conditions. According to the *World Health Organization* (WHO), rehabilitation is defined as "*a set of interventions designed to optimize functioning and reduce disability in individuals with health conditions in interaction with their environment*" [1]. In Sweden, the net cost for healthcare in 2021 was approximately 352 billion SEK (Swedish Krona) [2]. Of this, physiotherapy and occupational therapy account for 481 SEK per inhabitant, totaling just over 5 billion SEK for the year 2021.

One approach to reducing these costs is to streamline rehabilitation, for example, by enabling training at home, logging results digitally, and sending them to physiotherapists or treating physicians. Or to optimize and individualize physical training for each patient by adjusting weights and repetitions based on measured results and therefore reducing the rehabilitation time [3]. This requires accurately measuring values from the exercises, such as forces exerted during a specific exercise. To enhance rehabilitation effectiveness and patient outcomes, innovative technologies like exoskeletons are increasingly being explored [4]. These devices offer the potential to provide targeted and measurable support during exercises, thereby optimizing the rehabilitation process and potentially lowering overall healthcare costs. In recent years, robotic systems have advanced rapidly in rehabilitation therapy and diagnostics, evolving and exploring the usage of modern systems [5].

An adaptive exoskeleton is a robotic exoskeleton designed to provide adjustable and personalized support to a user's joint during physical therapy or rehabilitation exercises [6]. By adapting its assistance based on the user's specific movements, capabilities, and therapeutic needs, it ensures optimal support and comfort. By providing controlled assistance, exoskeletons can improve mobility and reduce rehabilitation time [4].

This thesis specifically focuses on enhancing an adaptive exoskeleton for rehabilitation purposes by integrating torque-controlled actuation and an IMU-based sensor. Torque-controlled actuation functions as a motor and is essential as it allows precise matching of the exoskeleton's movements to the user's natural joint dynamics. The IMU-based sensor provides accurate spatial awareness, enabling real-time monitoring and critical adjustments for effective therapy. By focusing on these aspects, this thesis will explore practical approaches to achieving seamless interaction between the user and the exoskeleton, also known as transparency. Transparency can be defined as the exoskeleton's ability to assist without causing noticeable resistance or interference with the user's natural movements [7]. Achieving this requires overcoming several engineering challenges, including sensor inaccuracies, latency in real-time signal processing, mechanical friction, and effective real-time disturbance management. Through these efforts, the project seeks to optimize rehabilitation outcomes, ultimately paving the way for more efficient, personalized, and cost-effective patient care.

1.1 Purpose

This project aims to further develop a single-joint adaptive exoskeleton specifically designed for knee rehabilitation [8]. A previous project of this exoskeleton introduced essential features such as adjustable range of motion (ROM), modularity that allows adaptation to different joints (right or left legs and elbows), tactile sensors, and motorized assistance, see figure 1.

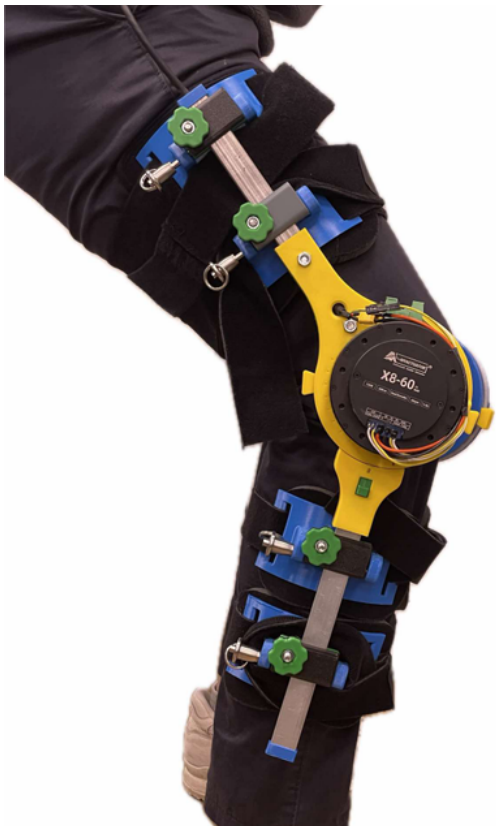


Figure 1: The prototype exoskeleton which was developed in a previous project. The exoskeleton featured adjustable straps as well as a torque motor and was developed to be used on both arms and legs [8].

In this project, preexisting the exoskeleton will be enhanced by integrating a *Torque Control System* utilizing the existing actuator, along with an *Inertial Measurement Unit* (IMU). The primary goal of the project is to develop a fully transparent system in which the wearability of the exoskeleton is seamlessly integrated into the user's natural movement allowing for effective rehabilitation.

1.2 Challenges

Achieving transparency in robotic exoskeletons presents significant engineering challenges. Transparency may be defined by the eq. $\tau_{dis} - \hat{\tau}_{dis} = 0$ where $\hat{\tau}_{dis}$ is the estimated disturbance torque and τ_{dis} is the effective disturbance torque [7]. In theory, the estimated disturbance $\hat{\tau}_{dis}$ is able to compensate for the effective disturbances τ_{dis}, but a system will never be fully compensated due to noise, communication delay, inertia and limited update rate [7]. Also, in a system with full compensation of these examples, it is still causal, meaning that disturbance compensation will only be applied after the disturbance has been measured. This limitation introduces a time lag that affects system responsiveness. To build a system with high transparency, with above limitations in mind, these challenges must be addressed:

- **Sensor limitations:** Torque sensing and IMUs exhibit noise and drift, reducing measurement accuracy and affecting feedback reliability.
- **Signal processing latency:** The system must execute real-time computations for control adjustments. Processing time introduces delays that reduce the effectiveness of disturbance compensation.
- **Mechanical dynamics:** Friction, backlash and inertia in mechanical linkages introduce additional forces that must be accounted for in the control algorithm.
- **Communication bottlenecks:** Data transfer between IMUs, microcontrollers, and actuators is subject to transmission delays, reducing closed-loop control efficiency.

Since the system involves human interaction, the desired torque may be difficult to estimate, resulting in a faulty desired compensation such as torque compensation, creating too little or too much support for the user [7]. Sudden user-initiated movements and environmental factors introduce disturbances that the control system must compensate for in real time.

1.3 Limitations

This project is limited by several technical, practical, and methodological constraints which shapes the scope and outcomes of the work.

Firstly, the exoskeleton is restricted to a single degree of freedom (1-DOF), focusing solely on knee movement. This decision is based on the current state of rehabilitation for knee injuries, where single-plane movement is often sufficient and safer during early recovery stages [9]. Additionally, the complexity and time required to implement multi-joint functionality exceeded the available time frame.

Another key limitation is the hardware. The project builds on an existing prototype and does not involve designing the mechanical structure from scratch. While components may be purchased, the budget of 10,000 SEK restricts the choice of materials, sensors, and circuit boards. Consequently, more advanced or custom-designed hardware solutions could not be considered, which might affect system precision and adaptability. This also limits the system computational power and input/output capability, which may constrain real-time performance and sensor integration.

The system's sensing capabilities are limited as well. The IMU and actuator used are subject to noise, drift, and calibration errors, which may impact the accuracy of spatial awareness and feedback. No advanced sensor fusion or external referencing systems are implemented.

From a methodological standpoint, the system is not clinically tested or evaluated in a real-world rehabilitation context. No direct collaboration with healthcare professionals or patients was conducted, meaning the design and testing are only theoretical or technical.

Lastly, the focus of the project is limited to achieving transparency in control, meaning minimal resistance or delay in user-exoskeleton interaction. Other important rehabilitation metrics such as endurance, long-term wearability, or haptic feedback have not been addressed within the scope of this thesis.

2 Theory

The construction of a transparent exoskeleton involves complex hardware and software solutions. This chapter means to give a background of the components and the control theory base used to create a working system.

2.1 Hardware

The hardware of the exoskeleton consists of a combination of mechanical, electronic and computational components that together enable controlled motion and assistive rehabilitation. This section provides an overview of the physical structure and key devices used, both in the previous prototype and the current system, with a focus on their intended function and theoretical relevance.

The mechanical frame, including joints and braces, is designed to be modular and adjustable to fit different limb sizes and applications. These parts were initially produced using 3D printing techniques, specifically Fused Deposition Modeling (FDM), which allowed for rapid prototyping and design iteration. Actuation is handled by a torque-controlled motor capable of assisting joint movement under controlled conditions. To process data and control the system, a compact single-board computer (Raspberry Pi) is used as the main controller.

Sensing is another critical part of the hardware system. Devices such as torque actuator and an IMU are used to monitor joint position and movement. A more detailed explanation of both actuator and IMU and their function in motion tracking will be presented later in this chapter and in subchapter 2.4.3.

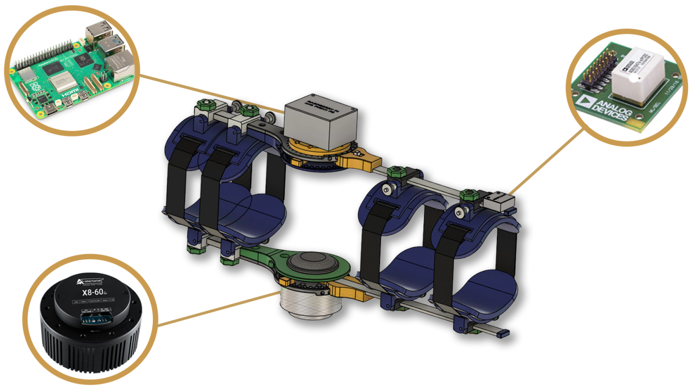


Figure 2: Highlight of hardware used in the exoskeleton. Raspberry to the top left, IMU on the right and motor at the bottom left.

2.1.1 Orthosis

The orthosis refers to the the mechanical structure of the exoskeleton, physically coupling the actuator to the user's limb and enabling force transmission between the motor and the limb. This orthosis is made up of two segments. one attached to the thigh and the other attached to the shank. In this project, the orthosis is fully fabricated using FDM 3D printing.

2.1.2 Actuator

An actuator is a device that converts energy into mechanical motion or force and works as a motor to drive joint rotations [10]. The actuator used in this project is a brushless servo motor designed to deliver precise torque current, velocity, and position control [11]. The torque current can later be calculated into pure torque through eq. 1:

$$\tau = K_t \cdot I \tag{1}$$

Brushless actuators are commonly used in robotic and medical applications due to their silent operation, low internal friction, and high accuracy [12]. The selected actuator integrates an internal encoder which allows real-time tracking of joint angle and rotational speed.

Importantly, the actuator is also backdrivable, meaning it is able to be rotated by external forces even when not actively powered [13]. This allows users or external loads to influence the system. Backdrive results in a change in motor current that is able to be measured and used to measure externally applied torque in real time.

The actuator communicates with the system's onboard computer, later described in 2.1.4, via a CAN-bus interface, described in 2.1.5, [11]. It receives velocity or torque commands and responds with structured data packets that include current joint angle, angular velocity, and torque-related current. These status messages are crucial for feedback control and are discussed in greater detail in subchapter 2.1.5.

2.1.3 Inertial Measurement Unit

An IMU, (inertial measurement unit), is a sensor that measures and reports a body's specific force, angular velocity and sometimes magnetic field [14]. It typically consists of a combination of accelerometers and gyroscopes, and in some cases magnetometers. These measurements enable estimation of the sensor's orientation in space.

In this exoskeleton system, the IMU is mounted on one of the rigid segments of the orthosis to track its spatial orientation. The angular velocity from the gyroscope and the linear acceleration from the accelerometer are used to estimate the rotation of the segment relative to a fixed world frame [14]. This information is crucial for determining the posture and movement of the leg, which in turn feeds into the control system and dynamics model.

Orientation is often represented using quaternions due to its robustness against singularities and drift in 3D space [14]. The process of computing and using quaternions from IMU data is described in detail in subchapter 2.4.

IMU data is affected by multiple sources of noise and drift, which makes raw orientation estimates increasingly unreliable over time [14]. The gyroscope delivers high-frequency angular velocity measurements but is prone to drift due to integration bias. The accelerometer, in contrast, is sensitive to external vibrations and transient linear accelerations, which may lead to distorted orientation readings if used in isolation. To mitigate these issues, filtering and sensor fusion techniques are applied. These are described in more detail in subchapter 2.6.3.

2.1.4 Onboard Computer: Raspberry Pi 5

An on-board computer is a small computer mounted on a system to handle computations. A Raspberry Pi 5 is a compact single-board computer capable of handling real-time data acquisition, signal processing, and control computations [15]. With a processor running at 2.4 GHz and up to 8 GB of RAM, it offers sufficient performance for executing control algorithms and managing sensor data streams simultaneously [16].

The device supports various interface protocols, making it suitable for communication with external sensors such as IMU's and motor drivers. Its GPIO-pins (General Purpose Input/Output) enable direct hardware interaction [8]. This is essential for applications requiring critical timing. The Raspberry Pi 5's small form factor and low power consumption further contribute to its suitability for embedded systems in wearable robotics.

2.1.5 Motor Communication Interface: CAN-bus

A controller area network (CAN) bus is a robust communication protocol designed for real-time communication between different components [17]. It is primarily used in automotive and industrial applications and enables efficient communication between multiple components. The protocol is a form of message-based broadcasting network that works by specifying unit-IDs with each message. Where in theory, multiple units are able use the same bus while only listening and responding to messages with their unique IDs. This ID system also enables prioritization, although in this application there is only one unit, the motor, listening for commands with a specific ID and responds using another ID containing the aforementioned information, see subchapter 2.1.2.

Another property of using a CAN system, is it's use of dual-channels resulting in robust transmission with lower vulnerability to electrical disturbances [17]. In a CAN-connection, two cables are used, a CAN-H and CAN-L, representing CAN-high and CAN-low. The idea here being that the signal is encoded in *differential mode.* Meaning that any disturbance in the physical channel affects both cables the same amount, and in turn, since the difference between the voltage on the channels decide the output, the signal remains unchanged.

To interface the motor with the onboard computer, a CAN communication interface is required. This is handled by the PiCAN-FD extension board, mounted through GPIO-pins on top of the Raspberry Pi 5 and through CAN-cables to the actuator. Thus, the PiCAN FD board acts as a bridge between the Raspberry Pi 5 and the motor, enabling the sending and receiving of messages.

CAN protocols can differ in how the frames are expected to be constructed. In the case of the MyActuator the frames are expected to follow this ”template” in order to get interpreted correctly:

$$\underbrace{\texttt{141}}_{\text{Sender ID (11 bits)}} \qquad \# \qquad \underbrace{\texttt{A2}}_{\text{Command (1 byte)}} \qquad \underbrace{\texttt{00 00 00 10 27 00 00}}_{\text{Data (7 Bytes)}}$$

Example: A CAN-frame command sent from the PICAN-FD, targeting the unit with ID 141, Sending the Speed Closed-Loop Control command A2, With the encoded data frames specifying a speed of 100 Degrees per second.

Due to the nature of the CAN protocol a response is needed from the receiving unit in order to continue the communication. Which in this case could look like this:

$$\underbrace{\texttt{241}}_{\text{Sender ID (11 bits)}} \qquad \# \qquad \underbrace{\texttt{A2}}_{\text{Command (1 byte)}} \qquad \underbrace{\texttt{32 64 00 F4 01 2D 00}}_{\text{Data (7 Bytes)}}$$

Example: A CAN-frame reply sent from the Motor, Indicated by the changed bit in the ID, Replying with the encoded Command and the data bytes with internal sensor information such as current motor- temperature, speed and torque current.

To extract meaningful values such as torque current, speed and angle from the actuator's CAN response frames, the raw data must be decoded from hexadecimal format into a readable format. Each frame contains 8 bytes of data, where multiple 16-bit (2-byte) values are packed using little-endian byte ordering [18]. This means that the least significant byte (LSB) comes first, followed by the most significant byte (MSB).

An example of a CAN response is shown in table 1 below.

Table 1: Example of a CAN response frame from the actuator.

Byte Index	**Data Field (Hex)**	**Description**
0	`0xA2`	Command ID
1	`0x2D`	Temperature (int8_t)
2–3	`0x01 0xF4`	Torque current (LSB / MSB)
4–5	`0x64 0x00`	Speed (LSB / MSB)
6–7	`0x00 0x32`	Angle (LSB / MSB)

2.2 Software

In order for the hardware components to operate cohesively, a robust software framework is required. This chapter provides an overview of the software tools, programming languages, and middleware used in the system. The software architecture is primarily implemented in C++, with Python reserved for visualization and data analysis tasks. Communication and modular control are managed through ROS2, which provides a flexible and scalable framework for real-time operation and component communication.

2.2.1 ROS2 Middleware

ROS2, short for Robot Operating System 2, is an open source framework designed to support the development of robotic systems [19]. It acts as a middleware, handling communication between different parts of the system through a decentralized architecture. ROS2 is based on a node structure, where each node performs a specific task, such as reading sensor data or sending motor commands. An example of a simple publisher-subscriber structure is shown in figure 3. The publisher-node sends messages with a set frequency that the subscriber reads and interprets. Independently of this, the same subscriber node is also a client to the service meaning that the client sends a request to the service and receives a response with useful data. The figure highlights these connections between 3 nodes but the framework needed for an exoskeleton is much larger and may easily be built upon.

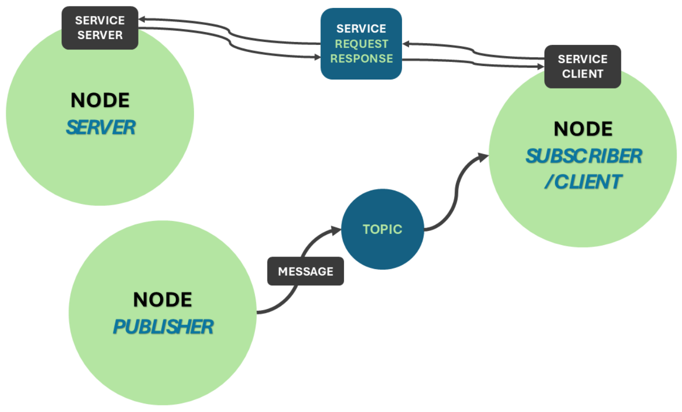


Figure 3: Example of publisher-subscriber structure that sends and receives messages using a service client.

ROS2 is used to coordinate the interaction between hardware components and the control algorithms [19]. It enables the exchange of data using a publish-subscribe model. This makes systems modular and easy to manage. ROS2 also supports real-time performance and deterministic execution, which are important features in robotic systems that require precise timing, such as exoskeletons.

2.2.2 Programming Languages

Supported programming languages in ROS2 applications are C++ and Python [19]. C++ is used for all time-critical operations due to its high execution speed and direct memory management [20]. In applications such as real-time control of an exoskeleton, these properties are essential to ensure low-latency responses and precise actuator behavior.

Python is used in a supplementary role for data visualization and post-processing tasks that do not require real-time performance [20]. While Python offers ease of use and a large ecosystem of scientific libraries, it is not well-suited for low-level control tasks due to its interpreted nature and lower execution speed. By assigning C++ to core control logic and Python to auxiliary functions, the system maintains both performance and development flexibility.

2.2.3 CAD - Computer Aided Design

Computer Aided Design (CAD) refers to the use of software tools to create detailed digital models of mechanical components and assemblies [21]. These 3D models allow for precise definition of geometry, material properties, and spatial relationships between parts. Complex systems may be constructed by assembling individual components and defining how they interact with one another [22]. In the context of an exoskeleton, CAD is used to design the mechanical structure and extract key physical parameters such as mass, center of mass, and moments of inertia, which are essential for dynamic modeling and control discussed in later subchapters.

2.3 Kinematics of an Exoskeleton

This subchapter outlines how joint movement in a one degree of freedom exoskeleton is modeled and measured. Here, the limb is modeled as two rigid segments, and motion is described in terms of joint angles. Orientation and positions are estimated using sensor data, including motor encoders and IMU, described in earlier chapters. These kinematic principles form the basis for the dynamic modeling and control strategies presented later on.

The system is simplified to a single degree of freedom, representing a knee joint. Considering the natural movement of a knee joint, it may be assumed that the joint is rigid, thus simplifying calculations [23]. As illustrated in figure 4, the angle α is defined as the angle between thigh and calve. It follows a local coordinate system, which means that the angle is always measured between two rigid segments, regardless of the orientation in a 3D space. The change of angle by the user is what later acts as an input in a control system loop as it results in a change of torque.

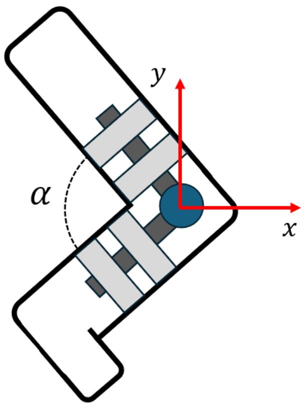


Figure 4: Representation of a 1-DOF exoskeleton and how the angle α is defined.

2.4 Quaternion Orientation

To accurately track the orientation of the segments of an exoskeleton's limb in three-dimensional space, quaternions may be used. Quaternions are four-dimensional numbers that represent rotations without the singularities or ambiguities associated with Euler angles [24]. An IMU mounted on the exoskeleton limb provides a quaternion that describes its orientation relative to a global reference frame. By interpreting them relative to the gravity vector, it is possible to estimate joint angles and monitor limb posture. This section introduces the basic principles of quaternions, discusses their advantages for motion tracking, and explains how they are applied within the exoskeleton system to support joint angle estimation and control.

2.4.1 Euler Angles

Euler angles represent rotations in three-dimensional space by defining three successive rotations around the principal axes x, y, z, as shown in figure 5 [24]. They require little computational effort and are therefore generally easier to understand mathematically. However, they suffer from several known downsides. Most notably, Euler angles are able to experience *gimbal lock*, a situation where two of the three rotation axes align, causing a loss of one degree of freedom (figure 6). For physical systems operating in three dimensions, this leads to unpredictable and uncontrolled behavior. Additionally, Euler angles interpolate orientations poorly, resulting in slower and less smooth real-time system performance compared to quaternions, which inherently interpolate the shortest rotational path.

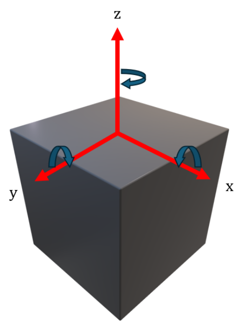


Figure 5: Illustration of how an object rotates along the three axes, giving the parameters of an Euler angle vector.

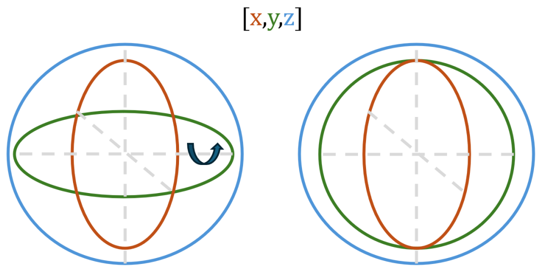


Figure 6: Gimbal lock occurring between the y and z axes when rotating 90 degrees around the x-axis.

2.4.2 Quaternions

Quaternions provide a robust alternative for representing rotations in three-dimensional space [25]. Unlike Euler angles, quaternions do not suffer from gimbal lock and allow smooth interpolation of orientations, making them well-suited for applications like exoskeleton control.

A quaternion is a hyper-complex number composed of four elements, eq. 2:

$$q = w + xi + yj + zk \tag{2}$$

where w is the scalar (real) part, and (x, y, z) form the vector (imaginary) part. It is often represented as a four-dimensional vector, eq. 3:

$$q = \begin{bmatrix} w \\ x \\ y \\ z \end{bmatrix} \tag{3}$$

Quaternions used for rotations are typically *unit quaternions*, meaning they have a norm of one, eq. 4:

$$\|\mathbf{q}\| = \sqrt{w^2 + x^2 + y^2 + z^2} = 1 \tag{4}$$

Utilizing this makes it easier to understand how a quaternion is mathematically structured. A rotation in 3D space may be represented by a quaternion constructed from a rotation axis $\vec{u}$ (a unit vector) and a rotation angle θ:

$$q = \cos\left(\frac{\theta}{2}\right) + (u_x i + u_y j + u_z k) \sin\left(\frac{\theta}{2}\right) \tag{5}$$

where:

- θ is the rotation angle,
- (u_x, u_y, u_z) are the components of the unit vector $\vec{u}$ defining the rotation axis.

The rotation angle θ ranges from 0° to 360°, while the scalar component w varies between -1 and 1, depending on the cosine of half the rotation angle. In figure 7 below, the rotation angle θ is shown in combination with the unit vector to better visualize the components. Quaternions enable smooth updates to an object's orientation in real time, avoiding the discontinuities that Euler angles may introduce [14].

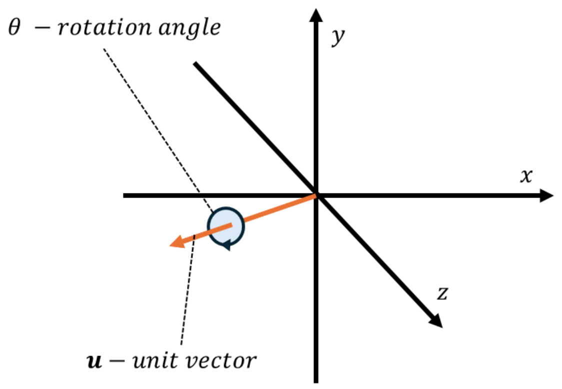


Figure 7: An illustration of a unit vector (u_x, u_y, u_z) and the rotation angle θ in a 3D space.

2.4.3 IMUs and Quaternions

An IMU is able to provide orientation data which may be transformed into a quaternion relative to a global (inertial) frame [25]. This orientation can then be used to calculate the angle, θ, between a system and the global vertical axis, which can be used to compute gravitational compensation in dynamic models.

An IMU's gyroscope measures angular velocity in three axes: $(\omega_x, \omega_y, \omega_z)$. These measurements are able to convert into a quaternion that represents the rotation of the segment over time. For small angular displacements and high sampling frequencies, this quaternion may be approximated as, eq. 6:

$$q_{\text{gyro}} = \left[1, \frac{\omega_x}{2}, \frac{\omega_y}{2}, \frac{\omega_z}{2}\right] \tag{6}$$

2.4.4 Orientation Angle Relative to Gravity

The orientation data from an IMU is provided as a quaternion, defined by the four components represented in eq. 3. This quaternion expresses the rotation of a body-fixed coordinate system with respect to a global reference frame.

To analyze segment orientation in space, a unit vector is selected from the local coordinate system. If the segment aligns with the local z-axis, the reference vector may be written as eq. 7:

$$\vec{v}_{\text{local}} = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} \tag{7}$$

This local vector is embedded into quaternion form by setting the scalar part to zero, eq. 8:

$$v_q = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} \tag{8}$$

To convert this vector into global coordinates, quaternion multiplication is applied, eq. 9 [14]:

$$\vec{v}_{\text{global}} = q \cdot v_q \cdot q^{-1} \tag{9}$$

For comparison, gravity in the global coordinate system is typically represented by eq. 10:

$$\vec{g} = \begin{bmatrix} 0 \\ 0 \\ -1 \end{bmatrix} \tag{10}$$

An angle θ is then calculated to describe the deviation between the segment orientation and the vertical direction. This is obtained using the dot product, eq. 11:

$$\theta = \cos^{-1}\left(\frac{\vec{v}_{\text{global}} \cdot \vec{g}}{\|\vec{v}_{\text{global}}\| \cdot \|\vec{g}\|}\right) \tag{11}$$

The resulting angle serves as an intermediate variable in further modeling, particularly in dynamic calculations presented in chapter 2.5.

2.5 Dynamics of an Exoskeleton

This chapter presents the dynamic modeling of the exoskeleton system, focusing on the relationship between joint motion, segment parameters, and the resulting torques. Dynamic equations are essential for designing control strategies that accurately respond to the physical behavior of the system. The model incorporates inertial effects, gravitational load and, if relevant, frictional components [23]. Particular emphasis is placed on how joint torque requirements are influenced by the orientation of the limb relative to gravity, as calculated in the previous chapter. The resulting expressions form the basis for implementing gravity compensation and feedback control, as discussed in subchapter 2.4.4.

2.5.1 Equation of Motion

The dynamic behavior of the exoskeleton joint is described by the standard equation of motion for a rotational joint [23]. This relationship defines how torques applied at the joint result in angular acceleration, while accounting for the system's inertia, damping, and gravitational effects. This equation serves as a base input for the control system later on.

To avoid ambiguity, the variable α is used to denote the joint angle throughout this chapter. The symbol θ, previously introduced in chapter 2.5, refers to the orientation angle of the segment relative to the gravity vector and is used specifically in the context of the estimation of gravitational torque. In robotics, the variable q is often used to denote this angle but has not been chosen considering quaternions also use q as their standard notation. The general form of the equation is given by eq. 12.

$$\tau = M(\alpha)\ddot{\alpha} + C(\alpha, \dot{\alpha}) + G(\alpha) \tag{12}$$

Where:

- τ is the total torque applied at the joint [Nm],
- α is the joint angle [rad],
- $\dot{\alpha}$ and $\ddot{\alpha}$ are the angular velocity and angular acceleration [rad/s] respectively [rad/s^2],
- $M(\alpha)$ is the inertia term (mass matrix) [kg·m^2],
- $C(\alpha, \dot{\alpha})$ represents Coriolis and damping effects [Nm],
- $G(\alpha)$ is the gravitational torque acting on the joint [Nm].

In the context of a single-DOF exoskeleton, this equation simplifies to scalar terms, yielding eq. 13:

$$\tau = M\ddot{\alpha} + C\dot{\alpha} + G \tag{13}$$

The torque τ corresponds to the output of the actuator. The scalar M represents the effective moment of inertia of the limb segment about the joint axis. The damping term $C\dot{\alpha}$ may include contributions from passive resistance in the braces or structural joints. The gravitational torque G depends on the orientation of the limb relative to the direction of gravity and is further discussed in section 2.5.3.

This equation serves as the foundation for the remaining sections, where each dynamic component is defined and quantified for the current exoskeleton configuration.

2.5.2 Mass matrix and Inertia

The mass matrix describes the resistance of a mechanical system to angular acceleration. In a multi-joint robotic system, this matrix is typically a function of joint positions and includes coupling terms between links [26]. However, in this project where the system is only 1-DOF the mass matrix will become a scalar. This effective inertia is denoted by M, and it contributes to the inertial torque $\tau_{inertial}$ of the dynamic eq. 14 where $\ddot{\alpha}$ is the angular acceleration of the joint.

$$\tau_{\text{inertial}} = M\ddot{\alpha} \tag{14}$$

The value of M is obtained from the physical properties of the segment mounted below the knee joint. These properties are extracted from the CAD model, which provides the segment's mass m, the distance from the joint axis to its center of mass l_C, and its moment of inertia at the center of mass I_{COM}. If the moment of inertia is not already computed about the joint axis, the parallel axis theorem is applied to shift it from the center of mass, eq. 15:

$$M = I_{\text{COM}} + m \cdot l_C^2 \tag{15}$$

Here, l_C is the perpendicular distance from the knee joint axis to the center of mass of the segment. This formulation assumes the segment behaves as a rigid body and rotates about a single fixed axis. The resulting scalar mass matrix M is later used in the full equation of motion and plays a key role in determining the torque required to accelerate the limb segment.

2.5.3 Gravitational Torque

The exoskeleton consists of two rigid segments: one fixed to the thigh and one attached to the shank. As only the knee joint is actuated and analyzed, and the thigh segment lies proximal to the joint, it does not contribute to the torque at the knee. Consequently, the gravitational torque is computed solely based on the shank segment [23].

For a single rotating link with mass m, the gravitational constant g, and the distance from the joint axis to the segment's center of mass l_C, the gravitational torque τ_g is expressed as eq. 16:

$$\tau_g = m \cdot g \cdot l_C \cdot \cos(\theta) \tag{16}$$

Here, θ represents the angle between the users leg and the vertical axis in a 3D-space. This angle is obtained from the IMU-derived orientation, converted from quaternions as described in subchapter 2.4.4. As previously stated, the gravitational torque is an important input for the control system for which the theory is presented in the upcoming chapter.

2.6 Control Theory

In dynamic robotic systems, control systems are used to develop a model for driving the system to a desired state, with the objective of minimizing delay, overshoot, or steady-state error [27]. The following sections will present the mathematical and theoretical foundations on which the desired control system for the knee orthosis exoskeleton will be designed.

2.6.1 Control Loops

Different control systems can be classified based on their feedback paths, open-loop or closed-loop systems [27]. Closed-loop systems have an output signal that is fed back to the input whereas open-loop systems do not, which is illustrated in figure 8 and 9. Closed-loop systems are furthermore divided into negative feedback loops and positive feedback loops, where the output is either added or subtracted to the reference input signal.

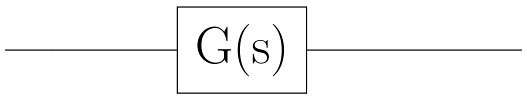


Figure 8: Illustration of an open loop system with transfer function, plant, G(s).

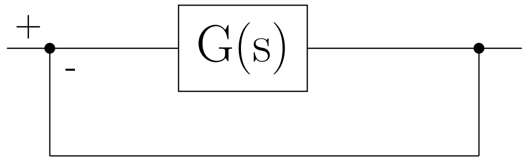


Figure 9: Illustration of a negative feedback loop system with transfer function, plant, G(s).

2.6.2 Regulators

Regulating a process is more than necessary when working with almost all control systems and there are many ways of doing this [27]. A controller is used in closed-loop systems as it monitors the signal of the system and measures the error between input and output. The error is related to the actual signal using individual methods or a combination of multiple controls. The most common methods are more thoroughly explained below. In order to move on with more advanced methods later, the theory behind the most common ones act as a necessary base.

Proportional control (P)
Being the most simple action, the proportional control takes the error between the desired and real output and creates a signal that is directly proportional [28]. The control signal,

$c(t)$ is amplified by the gain of the P-controller, K_p which leads to a control signal that relates to the magnitude of the error, $e(t)$ as seen in equation 17.

$$c(t) = K_p e(t) \tag{17}$$

While the P-controller operates fairly quickly it may come with an offset from the desired set point which is rarely wished for. Since the control method is based upon an amplification of the error, the offset can leave a steady-state error. Once the system reaches a point where the proportional correction equals the required control action, the error stops decreasing, even if it is not zero.

Integral control (I)
To account for the offset error that may happen with a P-controller, an integral regulator may be implemented in combination. It eliminates steady-state error by reacting to the accumulated error over time and adjusts the signal until the error is fully corrected [28]. This is essential for systems in need of fine control like this project but it makes the controller operate slower. The error as seen in eq. 18 is integrated and multiplied with a chosen gain, K_i.

$$c(t) = K_i \int_0^t e(\tau)\, d\tau \tag{18}$$

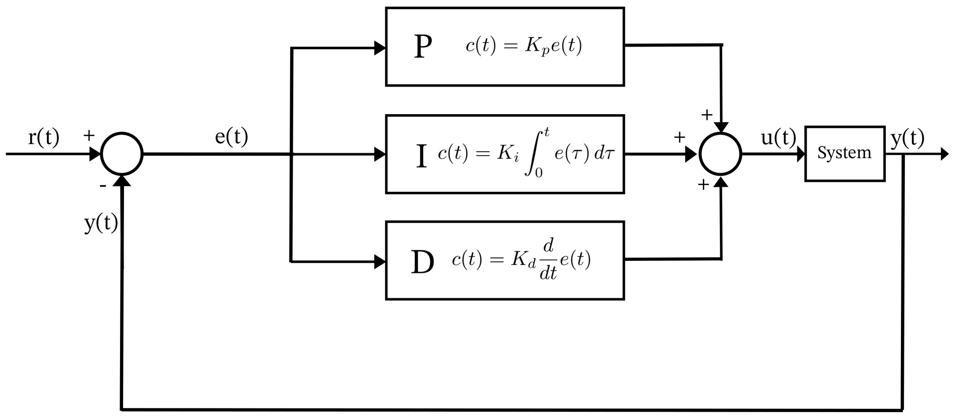
Derivative control (D)
Unlike the other methods, a D-regulator works more like a feed forward control. By primarily analyzing the change in error, it may be used to prevent larger changes and thus make the system converge faster and smoother if the algorithms are more complex [27]. The larger the change in error over time, the more it will respond. What differentiates it from other controllers is the fact that it does not guide the system to a steady state, which means that it has to be combined with another controller. The equation for a D-controller is presented below in eq. 19.

$$c(t) = K_d \frac{d}{dt} e(t) \tag{19}$$

PID regulator
Combining these methods may, as earlier stated, result in a better output by utilizing the strengths of each controller. PID, which stands for *Proportional-Integral-Derivative*, uses a combination of the 3 control methods presented above to regulate the process and achieve the desired output [27]. Tuning or weighing the different parameters of a PID regulator results in different effects on the process. It is the most common controller and, with the right tuning, the most accurate and efficient for more complex processes. Combining the eq. 17, 18 and 19 gives the full equation:

$$c(t) = K_p e(t) + K_i \int_0^t e(\tau)\, d\tau + K_d \frac{d}{dt} e(t) \tag{20}$$

The structure of a PID-regulator is illustrated in figure 10 below, which shows how the different regulators are paired together as seen in eq. 20. The PID-regulator will then act as a block in the larger control system used for this project.



r(t)
+
-
e(t)
y(t)
P $c(t) = K_p e(t)$
I $c(t) = K_i \int_0^t e(\tau)\, d\tau$
D $c(t) = K_d \frac{d}{dt} e(t)$
+
+
+
u(t)
System
y(t)

Figure 10: Illustration of a PID-controller and the methods used in a feedback loop.

2.6.3 Filtering

In control systems, sensor signals are often corrupted by noise, which can degrade the performance of the controller or even cause instabilities in the system [27]. Filtering techniques are therefore used to process measured signals, remove high and low frequency noise, and improve the quality of the feedback used in the aforementioned control loops, subchapter 2.6.1. Some of the more commonly used filters are described below:

Low-pass filter
A low-pass filter attenuates high-frequency components in a signal, allowing low-frequency components (such as steady-state or slowly varying signals) to pass through [29]. It is useful for removing high-frequency noise.

High-pass filter
A high-pass filter does the opposite: it attenuates low-frequency components and allows for high-frequency components to pass through [29]. The high-pass filter is useful for isolating rapidly changing signals, such as detecting abrupt motion or vibrations, while ignoring slow drifts.

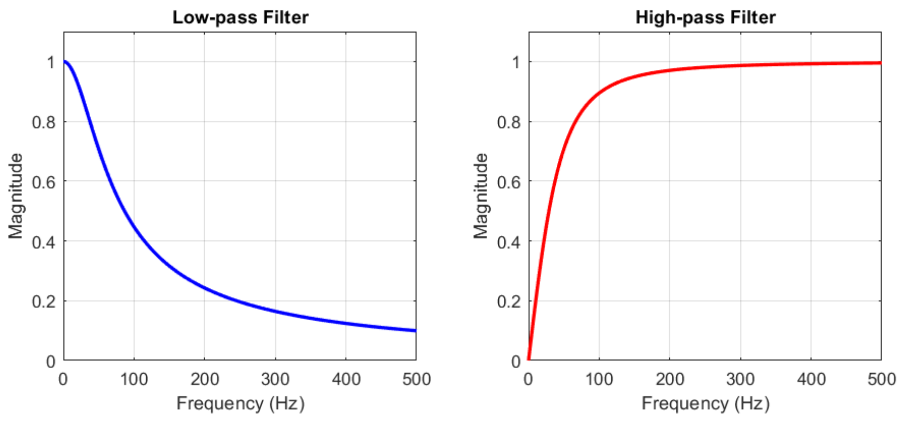


Figure 11: Figure showing the frequency response of a low-pass filter, that attenuates high frequencies and a high-pass filter, which functions by attenuating smaller frequencies.

The frequency responses of before mentioned filters are shown in figure 11, MatLab code for these graphs can be seen in appendix B. Different combinations of high-pass and low-pass filters create specific filters that target certain frequency ranges [29]. These combinations can be used for passing through a specific frequency band (band-pass filter) or targeting a specific frequency that is particularly noisy (band-stop filter).

Kalman filter
The Kalman Filter is a time-domain estimator that is used for estimating the state of a dynamic system [30]. It works recursively, by using the prior state estimate as well as the current sensor measurements to produce a new state estimate, see figure 12. This is achieved by computing a weighted average between the prediction and the measurement, where the weights reflect their respective uncertainties.

The equation for the Kalman filter looks like eq. 21:

$$\hat{x}_k = (1 - K)\hat{x}_{k|k-1} + K z_k \tag{21}$$

Where $\hat{x}_{k|k-1}$ is the predicted state from the previous estimate, $z_k$is the current measurement, and K is the weighted Kalman gain, $0 \leq K \leq 1$. As seen in the equation, a higher K puts more weight in the measurement, while a lower K favors the prediction.

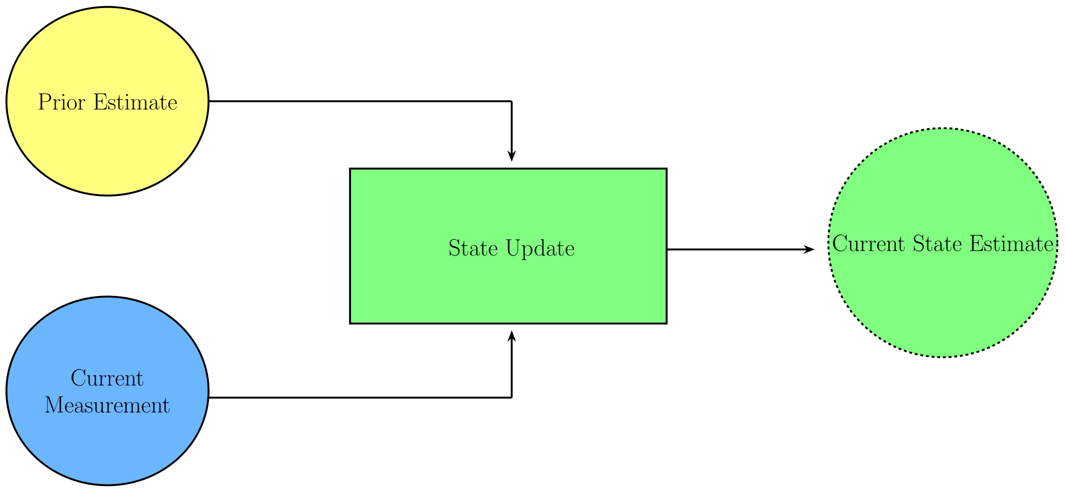


Figure 12: Illustration showing how the Kalman filter updates the current state estimation by using both the prior estimate and the current sensor measurements.

Madgwick filter
Another type of time-domain estimator is the *Madgwick filter*. The Madgwick filters is a *sensor fusion algorithm* used for translating raw orientation data provided by an IMU into quaternions [31]. It uses the measurements from an IMU's magnetometer and accelerometer to calculate the gyroscopes measurement error and corrects this through a gradient-descent algorithm [32].

2.6.4 Disturbance Observer

Observers are systems that use available measurements to provide an estimate of the systems internal states [27]. Some state variables, such as torque and velocity, can be measured directly. However, in many applications, systems are also affected by external disturbances (extra plant inputs), such as unknown forces or model uncertainties, that are different from the control inputs, as seen as the signal ***d*** in figure 13. The disturbances can thus have a negative effect on the behavior of the system.

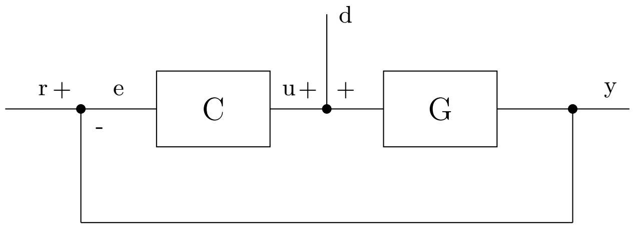


Figure 13: Negative feedback control system with input signal r, error signal e, control signal u and disturbance signal d [33].

While a constant, steady state disturbance, is able to be suppressed by, for example the integral term of a classical *PID* controller, as explained in subchapter 2.6.2, a harmonic disturbance cannot be effectively compensated for in the same way. A harmonic disturbance being a periodic disturbance of sinusoidal nature, (denoted as d_h in figure 14) [34]. If tried, the pure PID control will result in oscillations in the output signal, as seen in the figure 15 below.

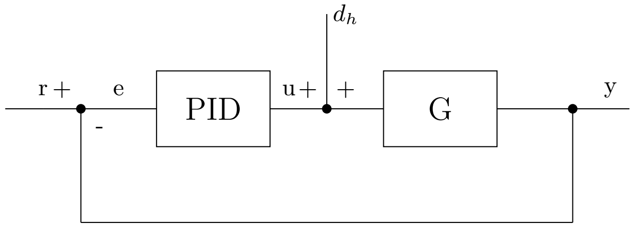


Figure 14: A block diagram illustrating the control scheme of a PID controller with an harmonic disturbance signal d_h.

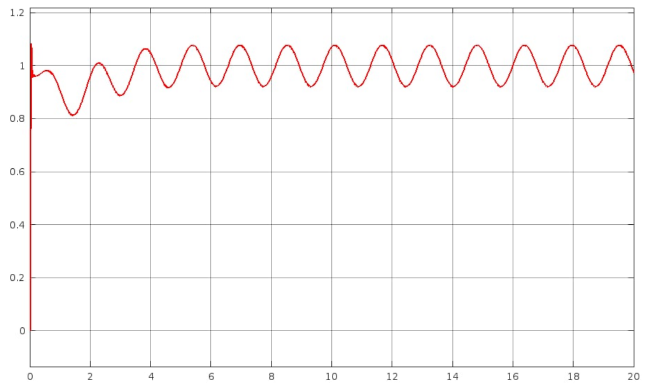


Figure 15: The control results of trying to suppress harmonic disturbances using a PID controller.

This limitation motivates the introduction of an alternate suppression model; the *Disturbance observer* (DOB), which estimates the plant inputs of disturbances and compensates them through a negative inner-loop output-feedback [35].

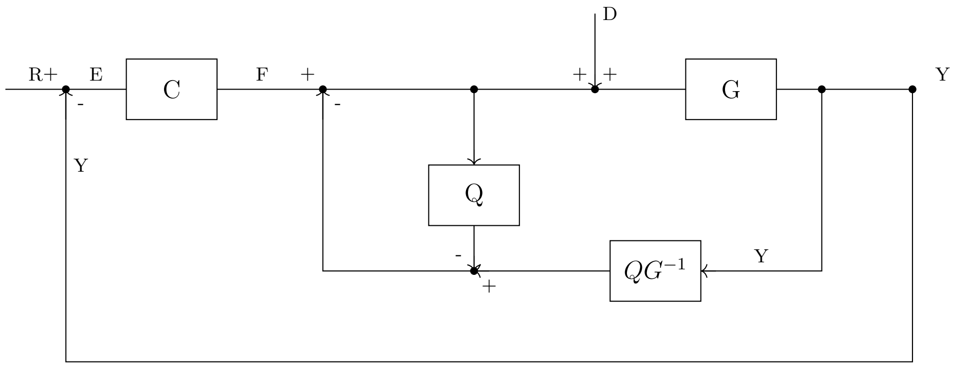


Figure 16: A block diagram illustrating a control system with a controller and an inner-loop outer feedback disturbance observer.

Mathematically, the effects of such a DOB is able to be proven by simple calculations of the separate input to output plants, G_{dy} and G_{ry}. The signals shown in figure 16 are listed below as well as the calculation for the disturbance estimate signal $\hat{D}$ in eq. 22:

$$\begin{aligned}
\text{Reference Signal:}\quad & R \\
\text{Disturbance Signal:}\quad & D \\
\text{Output Signal:}\quad & Y = G(D + F - \hat{D}) \\
\text{Error Signal:}\quad & E = R - Y \\
\text{Control Signal:}\quad & F = CE = C(R - Y) \\
\text{Estimated Disturbance Signal:}\quad & \hat{D} = \frac{QG^{-1}}{1-Q}Y - \frac{Q}{1-Q}F
\end{aligned} \tag{22}$$

The output signal, Y, is given by the sum of all transfer function contributions from input to output, as shown in eq. 23, [27].

$$Y = G_{ry}(s)R(s) + G_{dy}(s)D(s) \tag{23}$$

The separate input-output transfer functions are given by the equations below, eq. 24:

$$\begin{aligned}
G_{dy} &= \frac{G_n G(1-Q)}{G_n(1+GC) + Q(G-G_n)} \\
G_{ry} &= \frac{G_n GC}{G_n(1+GC) + Q(G-G_n)}
\end{aligned} \tag{24}$$

And with Q selected as a low-pass filter with the following property, eq. 25:

$$\lim_{\omega \to 0} Q(j\omega) = 1 \tag{25}$$

By now looking at asymptotically small frequencies, It may be shown that the output signal, defined by eq. 23, approaches a specific form. First the separate transfer functions for small frequencies are evaluated in equations 26 and 27:

$$\lim_{\omega \to 0} G_{dy}(j\omega) = \lim_{\omega \to 0} \frac{G_n G(1-Q)}{G_n(1+GC)+Q(G-G_n)} = 0 \tag{26}$$

$$\begin{aligned} &\lim_{\omega \to 0} G_{ry}(j\omega) = \\ &\lim_{\omega \to 0} \frac{G_n GC}{G_n(1+GC)+Q(G-G_n)} = \\ &\frac{G_n GC}{G_n + G_n GC + G - G_n} = \\ &\lim_{\omega \to 0} \frac{G_n C}{1+G_n C} \end{aligned} \tag{27}$$

It is now shown that the following expression for Y holds true:

$$\begin{gathered} \lim_{\omega \to 0} Y = \lim_{\omega \to 0} G_{ry}(j\omega)R(j\omega) + G_{dy}(j\omega)D(j\omega) = \\ \lim_{\omega \to 0} \frac{G_n C}{1+G_n C} R(j\omega) \end{gathered} \tag{28}$$

Thus, from eq. 28, it follows that for small frequencies, the closed-loop system with the DOB, behaves like a closed looped system of the nominal plant. The DOB actively rejects these frequencies if they are in the range of the bandwidth of the low-pass filter, Q [35].

3 Method

This section outlines the methodology and workflow adopted throughout the thesis project, detailing the key steps taken from the initial hardware design decisions to the final software implementation. Some parts of the workflow had to be done sequentially, while others could be done concurrently, but most parts of the project development followed an iterative procedure, where certain parts had to be solved temporarily to be revisited, refined, and restructured later.

3.1 Hardware Design

This subchapter outlines the practical steps taken to select, integrate and configure the hardware components used in the exoskeleton system. It covers the key design decisions, including component selection, physical wiring and testing setup.

3.1.1 Component Selection

The hardware components were selected based on performance requirements, flexible compatibility, and integration within the project's limited timeline and budget.

Selection of motor
The brushless servo actuator used in this project was inherited from a previous iteration of the exoskeleton concept and therefore not subject to re-selection. However, it was suitable for continued development as it provided sufficient torque for knee joint actuation and featured an integrated encoder for real-time joint angle estimation. These characteristics aligned well with the project's goals of enabling torque-based control and sensor-driven feedback.

Selection of computers
Raspberry Pi 5 was chosen due to it's superior processing power and GPIO support, which was necessary for the application. It is also widely used across various levels of expertise with extensive documentation and support available online.

A computer with an operating system was chosen over a microcontroller with microROS because of the need for ensured compatibility and greater software flexibility. This choice enabled the usage of ROS2, a more documented and accessible environment, while also aligning with the project's software requirements. Given the project's technical scope and available time, a full OS also enabled direct access to higher level programming interfaces and libraries, avoiding the hardware-specific complexities associated with microcontrollers.

During the selection process, alternative platforms including NVIDIA Jetson, ESP32, and

Arduino were identified and assessed. The evaluation criteria included processing capabilities, GPIO support, availability of documentation, and access to broad community resources. Based on these criterias, Raspberry Pi 5 was selected to ensure compatibility with ROS2 and maintain flexibility in software development.

Selection of IMU
The ADIS16470 was chosen due to its industry-level performance, and as it is widely popular in industry applications [36–38]. Judged by it's datasheet, it also had great performance [39]. This enabled the group not to have to focus on common signal reliability issues as previously mentioned in chapter 2. The group could now instead focus on implementing the part and future groups could also have a great reliable product that does not necessarily need recalibration.

Alternative products were of course evaluated, judging by their performance, price and widespread usage. Those considered were either too noisy, unreliable, too niche and generally lacked widespread documentation.

Selection of CAN-bus card
To enable communication between the Raspberry Pi and the actuator, the PiCAN FD board was decided. It supports both CAN 2.0B and CAN FD protocols, and connects directly to the Raspberry Pi's 40-pin GPIO header [40]. This flexibility enabled the group to have a choice between data speeds, if later proven that it was supported by the other components. An illustration of the communication is presented below in figure 20b.

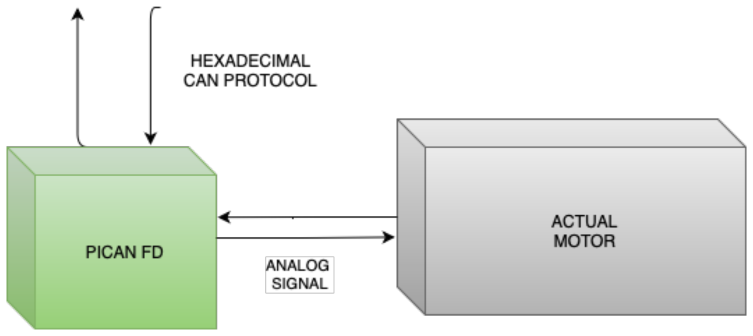


Figure 17: PICAN FD to motor connection.

3.1.2 Integrating hardware

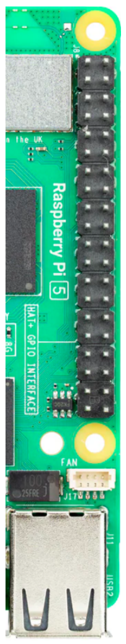
Most information on how to physically connect components either came from manufacturers' or resellers' official product data sheet. Some connections were chosen to get software running as intended, as mentioned later in detail in subchapter 5.1.5. This especially applied to the IMU and was characterized by more troubleshooting, trial and error, using forums, in lack of documentation of the niche problems.

In short: The actuator was connected to the PiCAN FD board via a standard *4-wire CAN interface*. The PiCAN FD board, in turn, was with selected pins connected to the Raspberry Pi 5's GPIO header. The IMU was connected via SPI, requiring manual wiring from the IMU to the GPIO pins. The connections are further illustrated in the figures presented below.

This connection is best illustrated by table 18a, visualizing the Raspberry Pi 5's GPIO-pins:

(a) A table showing an overview of the Raspberry Pi 5's pinouts with relevant highlights.

Raspberry Pi Pinouts			
GPIO	**Pin No.**	**Pin No.**	**GPIO**
3v3 POWER	1	2	5V POWER
GPIO2	3	4	5V POWER
GPIO3	5	6	GROUND
GPIO4	7	8	GPIO14
GROUND	9	10	GPIO15
GPIO17 — SPI 1	11	12	GPIO18 SPI1
GPIO27	13	14	GROUND
GPIO22	15	16	GPIO23
3v3 POWER	17	18	GPIO24
GPIO10 — SPI 0	19	20	GROUND
GPIO9 — SPI 0	21	22	GPIO25
GPIO11 — SPI 0	23	24	GPIO8 — SPI 0
GROUND	25	26	GPIO7 — SPI 0
GPIO0	27	28	GPIO1
GPIO5	29	30	GROUND
GPIO6	31	32	GPIO12
GPIO13	33	34	GROUND
GPIO19 — SPI 1	35	36	GPIO16 — SPI 1
GPIO26	37	38	GPIO20 — SPI 1
GROUND	39	40	GPIO21 — SPI 1



(b) Corresponding pins in real life [41].

Figure 18: Figure showing theoretical coupling in a table (a), and it's correspondence in real life (b).

In the configuration both the Raspberry PI and SPI-buses were used. This enabled two separated and clear data streams, for the IMU and PiCAN FD respectively. In this configuration, the IMU used the SPI 1-bus (marked in red), and it's remaining needed pins (marked in light red). The PiCAN FD used the SPI 0-bus (marked in blue), and because of uncertainty the remaining pins were all connected to the PiCAN FD, as seen in the table 18a).

Computer - PiCAN connection
The computer to PiCAN connection was connected as seen above in table 18a. It's simplified connection is shown in figure 19 below.

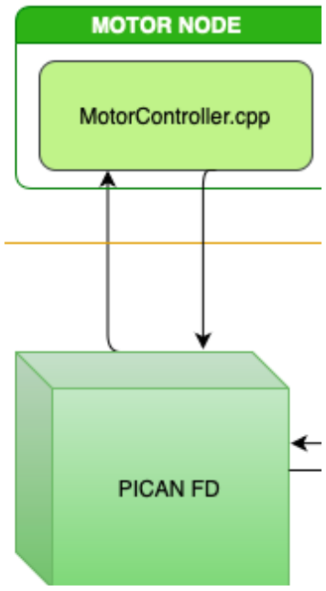
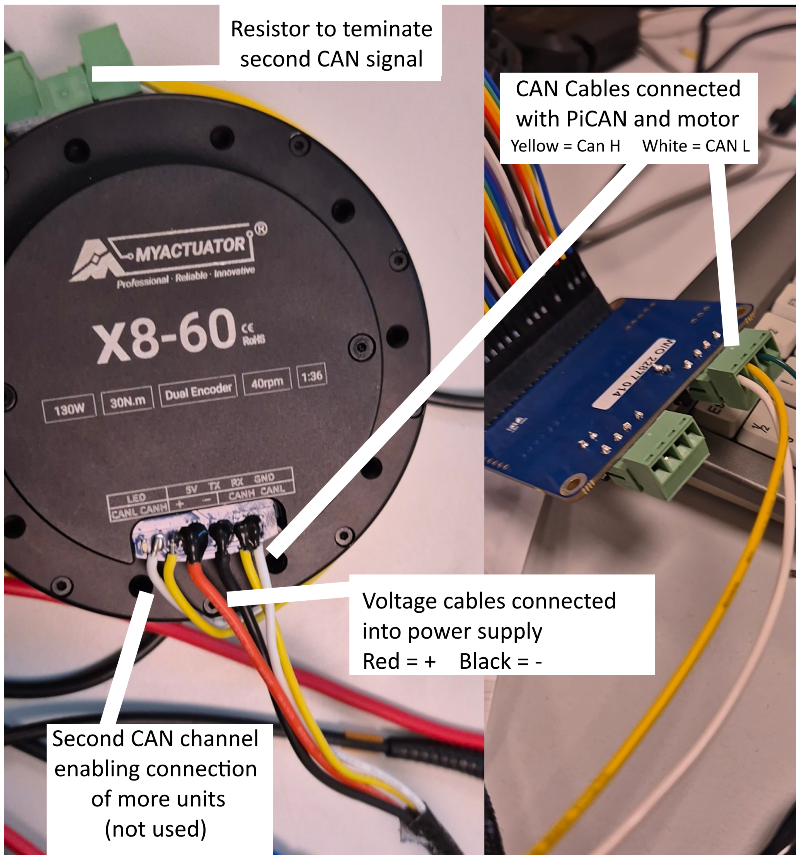


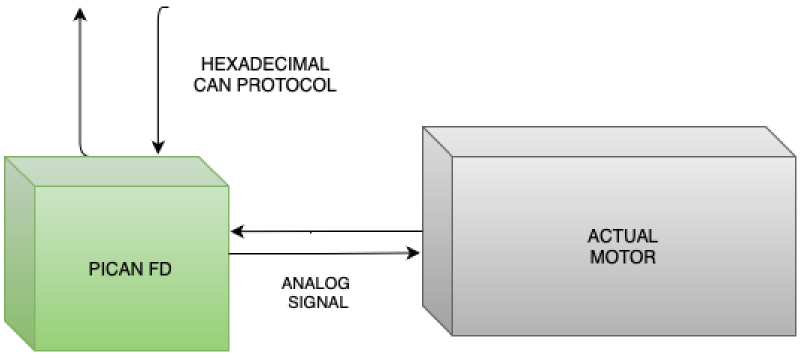
Figure 19: A figure simplifying the PiCAN FD connection to computer node.

Before integrating the IMU, since the PiCAN is a HAT (Hardware Attached on Top), it was mounted directly on the Raspberry Pi 5 pins. In combination with a lack of time, the exact pin configuration was therefore not researched. When introducing the IMU this configuration was changed, and remaining pins all connected to the PiCAN FD as previously explained.

PiCAN - Motor
The motor was connected to the PiCAN module with two cables, one yellow and one white, as seen in 20a. As mentioned in subchapter 2.1.5 these are the two cables used in CAN to get a clean signal.


Resistor to teminate second CAN signal
CAN Cables connected with PiCAN and motor
Yellow = Can H White = CAN L
MYACTUATOR
Professional · Reliable · Innovative
X8-60
130W
30N.m
Dual Encoder
40rpm
1:36
Voltage cables connected into power supply
Red = + Black = -
Second CAN channel enabling connection of more units (not used)

(a) The connections in real life.

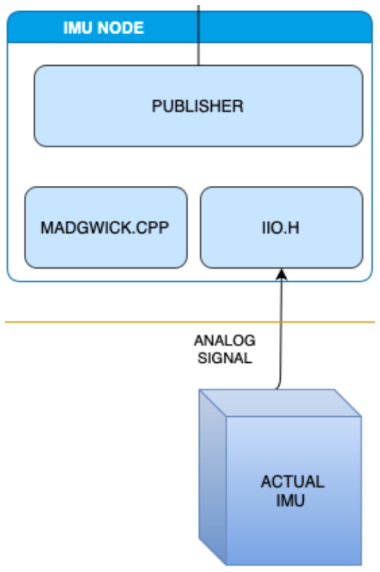


(b) The connections in theory.

Figure 20: Pictures that shows the connections between the motor and PiCAN FD.

Also seen in figure 20a, the remaining pair of yellow and white cables, were connected to a resistor, terminating the remaining pair. This prevented the signal from bouncing, and was done because no further module needed to be connected. The black and red cable was connected to a satisfactory power supply, provided by CASE. Furthermore, there was a UART-plug below the motor's connection plate. This was used for initial debugging and configuration of the unit, made with MyActuators native debugging software [42].

Computer - IMU connection
Lastly, the IMU was connected to the computer as illustrated in 21b.



(a) IMU.

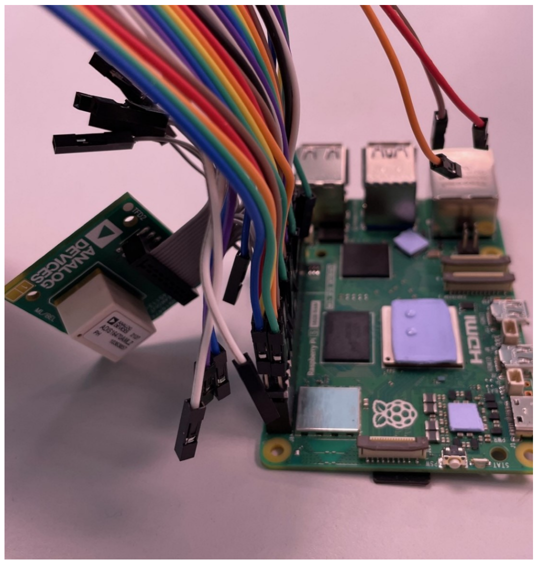

(b) RPi connections.

Figure 21: The image shows how the IMU and RPi are connected digitally (a) and physically (b).

In detail, the connection was made according to table 2.

Table 2: Tabular showing how the computer-IMU connections are connected through GPIO-pins.

RPi 5 Name	**Function**	**RPi Pin No.**	**IMU Signal**	**IMU Pin No.**
GPIO 18	SPI 1 Chip Select	12	CS	3
GPIO 21	SPI 1 Serial Clock Signal	40	SCLK	2
GPIO 20	SPI 1 MOSI	38	DIN	6
GPIO 19	SPI 1 MISO	35	DOUT	4
GPIO 4	Data Ready	7	DR	13
3V3	Power	17	VDD	10
GPIO 12	Reset Signal	32	RST	1
GROUND	Reset Signal	9	GROUND	9

To connect the computer with the IMU, there were initally two options.
Either:
1. Connecting the IMU on top of the PiCAN FD
or
2. Connecting it directly on the computers GPIO pins, together with the Pi-CAN.

The second option was chosen because of uncertainty whether the PiCAN pins would be pass through, even if the pins were successfully extended. This option is also what is visualized in figure 21b. The cells marked in blue are GPIOs that are decided software wise. This is further explained in subchapter 5.1.5

3.2 Software Architecture

This section outlines how the software system was structured to enable communication between the systems key components. It describes the setup of the operating system, the implementation of ROS2 nodes for the actuator and IMU and the overall control logic.

3.2.1 System Architecture

The fundamental start software-wise was to install the correct operating system (OS). This was installed on to an SD card, which was then inserted into the Raspberry Pi.

For Raspberry Pi, there are many compatible operating systems, where some are adapted and therefore preferable for specific tasks. The required system criteria for this project comes from different needs, as listed in the projects challenges in subchapter 1.2. The system on a software level needs to be:

- Responsive.
- Fast.
- Implementable with ROS2 - a directive from examiners.
- Relatively well documented - facilitates any troubleshooting.

Since ROS2 Jazzy was used, Ubuntu was chosen over Windows 10 due being more efficient, higher performing, and greater flexibility for development.

CAN interface setup
An essential step was getting the PICAN-FD setup as a visible network interface in the OS, this was done by appending the following lines:

```
dtparam=spi=on
dtoverlay=mcp2515 -can0,oscillator=16000000,interrupt=25
```

to the boot configuration at

```
/boot/firmware/config.txt
```

And later activated before usage with the correct bitrate using the following terminal commands:

```
sudo ip link set can0 up type can bitrate 1000000
sudo ifconfig can0 up
```

This specifies to the OS, what type of device it is, how it operates and that its communicating through the SPI-channel of the Raspberry PI's pins with the bitrate specified by the manufacturer.

IMU Node
The IMU Node consists of three components: The Industrial Input Output subsystem (IIO), which is a subsystem in the Linux kernel used for SPI-communication [43]. Here it is used to read the raw IMU data from the sensor, in the form of accelerometer and gyroscopic values and pass it into the next step of the chain. These variables are then processed by the next component, a cross compiled Madgwick filter, originally developed for Arduino, which converts the raw sensor data into a quaternion (see subchapter 2.4.3) [44].

The final component, the publisher, then publishes these four quaternion values onto the quaternion topic at a rate of 1000 Hz. This rate was chosen to be higher than the rate of the control loop polling the quaternion topic, to ensure that its getting the most up-to-date data because these processes execute asynchronously from each other. Meaning that the control node does not need to read from the topic every time the topic gets updated with a new value. At this rate, the IMU is running at well below its theoretical limit and could be pushed further, the decision was made not to do proceed further because of a lack of knowledge on potential delays/overheads caused by OS/program execution times/ROS, being out of scope of our project. The general view of the imu node is illustrated in figure 22.

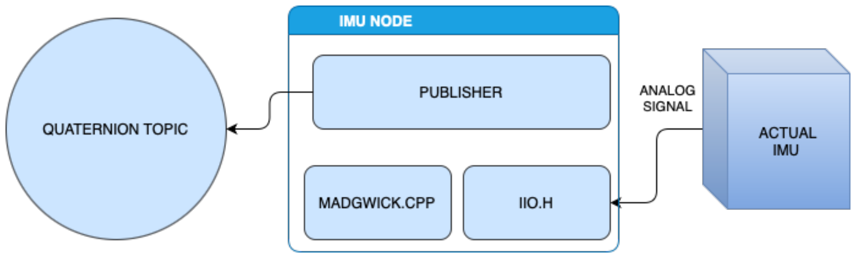


Figure 22: Diagram of IMU node

Motor Node
The motor node consists of three components: MotorController.cpp and two ROS2 Servers. The ROS2 Servers being the speed server and the stop server. Whose only purpose is to respond to their respective speed-service or stop-service with the relevant data when they get a request, in this case from the control node. The relevant data in this application being: motor encoder angle, motor speed, motor torque current and motor temperature for the speed service. While the response for the stop service contains no data, acting only as an acknowledgment that the stop-command has been sent to the motor.

MotorController is essentially only a class that does all the CAN-interfacing with the motor. This includes CAN-related functions, a constructor and a destructor. When this class is constructed it calls the constructor, and vice versa for the destructor, both of these being called once at startup and shutdown of the motor node. The constructor opens a raw CAN socket and binds it to the can0 interface, which is set up in the the OS to be our PICAN FD. This is done using the SocketCAN library from the Linux kernel [45] and allows us to send and receive CAN-frames on the CAN bus. The destructor simply sends a stop command and closes the socket once its called at the termination of the motor node process. Resulting in a safe shutdown without the motor repeating the last given command, which in worst case, could be a move command resulting in the motor moving even though the software is shut down.

The aforementioned functions are best explained through following the process of how a CAN message is sent on the socket. The controller is capable of sending two different types commands to the motor, specified in the X8-60 Motor Motion Protocol as *Speed Closed-Loop Control Command (0xA2)* and *Motor Stop Command (0x81)* [46]. The speed command uses an internal closed loop system to draw the current needed for the motor to spin with a set speed. This speed is determined through the CAN frames data bytes. as discussed in section 2.1.5 this needs to be encoded through an encoder function with takes in the desired speed and outputs a constructed CAN frame to be sent on the can-bus. The same is done for the stop command, except that the stop-commands data frame does not contain any information. Once the command has been ”translated” to a CAN frame its ready to be sent. Here two functions are working in tandem: one that sends the frame and another that receives the reply. After the frame is sent, the thread halts, and waits for the reply from the motor. As soon as this frame has been received its output is written to the reply field of the appropriate service. Now the program is ready to receive another request and the cycle repeats.

The general view of the motor node is illustrated below in figure 23.

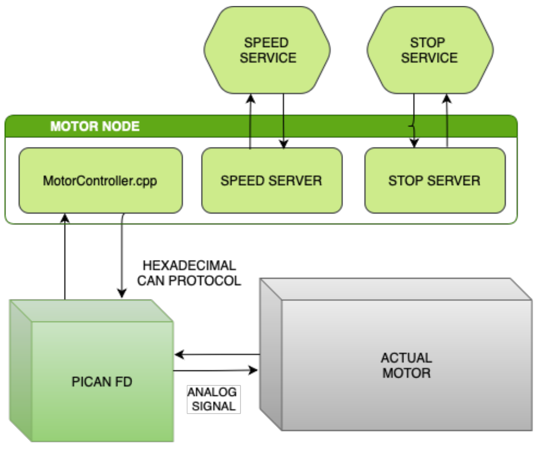
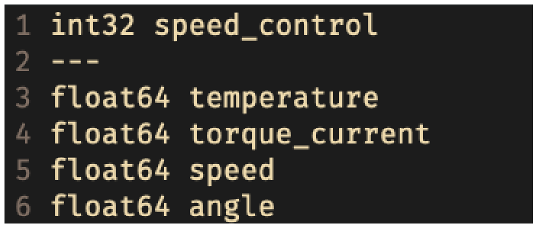


Figure 23: Diagram of motor node.

Control Node

The control node consists of three essential components and houses the *decision-making engine* of the application, called the control logic. The control logic component takes in all the relevant data to the control system from the two information sources: motor joint angle, motor speed and motor torque current from the motor as reported by the relevant ROS service. In addition to motor information, quaternion information from the IMU is also subscribed to through the ROS Quaternion topic.

ROS2 did have a preset topic message template for quaternions, containing the required variables. However, for the motor services two custom service messages had to be created and compiled in an additional package, to then later be imported and used in the control node and motor node in order to send messages in the same format. The format of a .srv file is Request data, dashed line, Reply data, in that order as seen in figure 24.



```
int32 speed_control
---
float64 temperature
float64 torque_current
float64 speed
float64 angle
```

Figure 24: Custom Speed service .srv message.

The control logic unit should then take the quaternion and motor data and pass it through the actual control logic/control system. This proper control system had not successfully been implemented to a satisfactory degree and was therefore omitted from the code. In its place, we made three example ”demo” control modes as placeholders and demonstration purposes. These being: ”Simple torque control”, ”Advanced torque control attempt with filtering” and ”keyboard control”.

This processing then returns a speed value which is then sent as a request to the speed service to tell the motor to actuate in that direction with a set speed. This processing happens at a clock rate set 500 Hz as an arbitrary number, high enough to feel responsive and lower than the IMU and actuators rate limit while still reserving a lot of time to potential delays/overhead discussed earlier. The general view of the control node is illustrated below in figure 25.

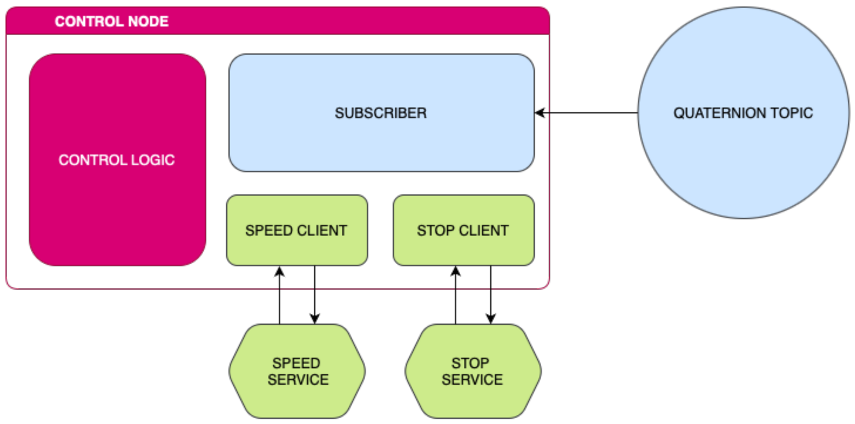


Figure 25: Diagram of control node

3.3 CAD

As stated in earlier, to determine the gravitational torque and understand the influence of gravity on the exoskeleton, a mass matrix is required. To obtain the mass matrix of the exoskeleton, computer-aided design (CAD) software was used. In this project, Autodesk Fusion 360 was the software chosen mostly because of the previous work on the already existing CAD model. To save time, their CAD parts were used for this project as well. Some of their CAD parts were missing or did not match with the parts in reality, and therefore some of them were created or adjusted.

Once all components were modeled and ready to be assembled, a mass was assigned to each part to reflect the properties of the existing physical components. In addition to the mechanical structure, Raspberry Pi, PiCAN FD, and ADIS16470 (IMU) were intended to be mounted on the exoskeleton. Custom cases for these circuit boards were therefore designed in CAD. One case housing both the Raspberry Pi and the PiCAN FD, and another case for the ADIS16470.

4 Results

This section presents the outcomes of the project, covering both the hardware and software aspects of the exoskeleton's final system. The results include the finalized CAD model, the integration and setup of key hardware components along with the software architecture and the control logic.

4.1 CAD

The result of the assembled exoskeleton in CAD is shown in figure 26. The including components now have assigned correct masses.

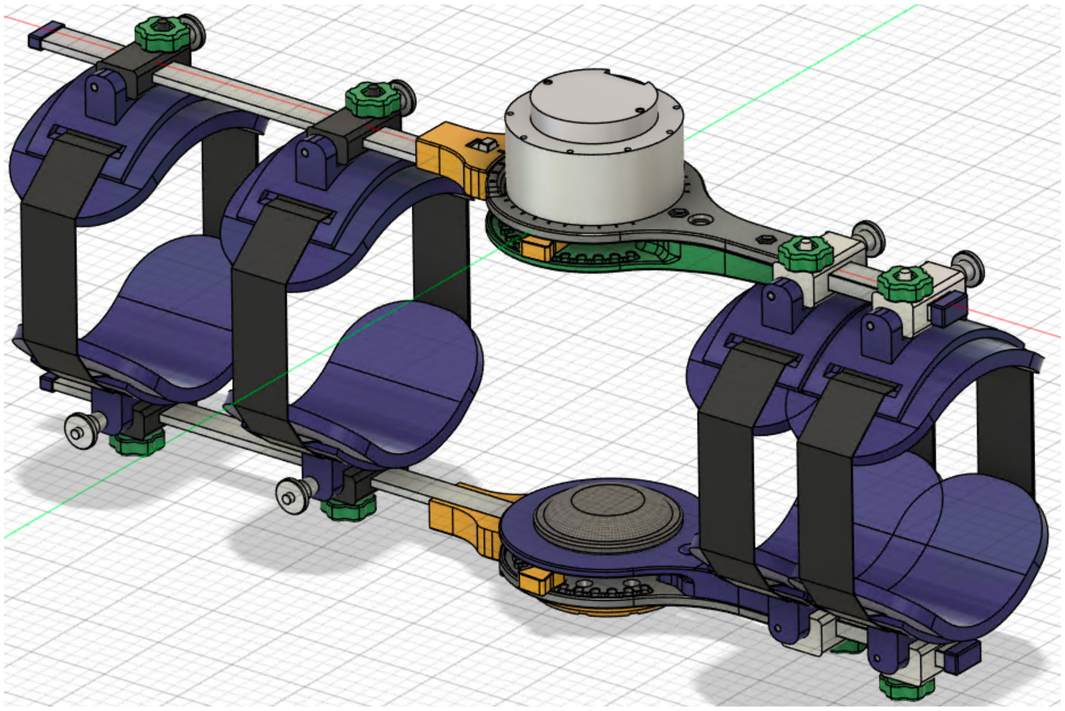


Figure 26: Picture of the assembled CAD model of the exoskeleton.

A table with the separate parts and its weight is shown in figure 27.

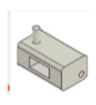
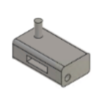
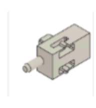
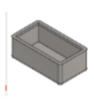
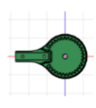
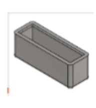
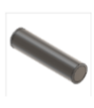
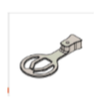
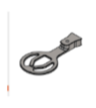
	85,8 g		23,1 g		39,2 g		125,5 g
	937 g		21,5 g		31,4 g		28,2 g
	4,9 g		5,9 g		1,4 g		51,1 g
	1,3 g		24,7 g		2,0 g		119,2 g
	5,4 g		67,2 g		1,0 g		
	3,1 g		8,6 g		8,5 g		

Figure 27: A table with the parts and its weight.

The circuit boards on the assembled exoskeleton is shown in figure 28.

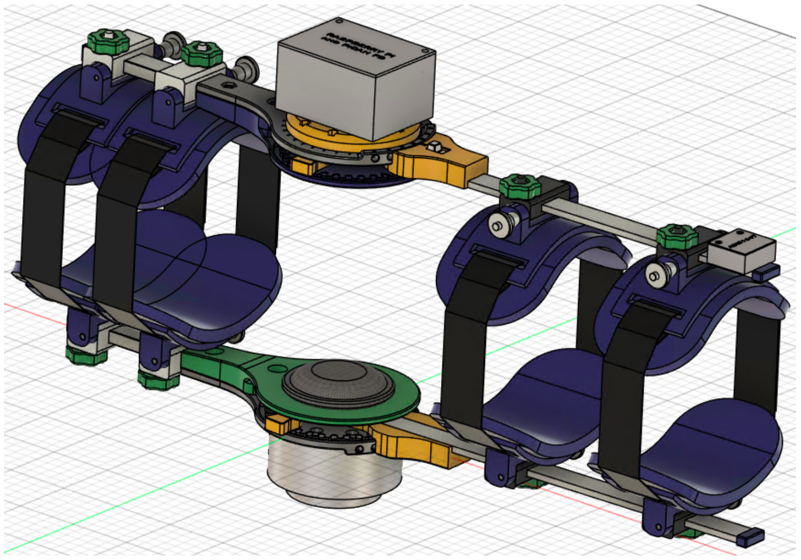


Figure 28: Exoskeleton with Raspberry Pi and PiCAN FD on the opposite side of the motor, and ADIS on the end of the long bar.

An exploded view, and the CAD models of the circuit boards and their cases is shown in appendix C. The boards are highly simplified compare to the existing physical circuit boards.

4.1.1 Calculation of Mass Matrix

With the values taken from the CAD assembly of the exoskeleton, the calculations for the mass matrix was done using eq. 15 resulting in the values presented in table 3 below.

Table 3: Physical parameters of the exoskeleton prototype.

Quantity	**Value**
Moment of Inertia at center of mass (I_{com})	0.0897 $\text{kg} \cdot \text{m}^2$
Distance from joint to center of mass (l_c)	0.00666 m
Mass of exoskeleton (m)	2.67 kg
Mass matrix (M)	0.0898 $\text{kg} \cdot \text{m}^2$

4.2 Setup Results

The diagram illustrated in figure 29 provides an overview of the software architecture of the final motor control system, along with its hardware components and system logic. The corresponding system is shown with it's connections.

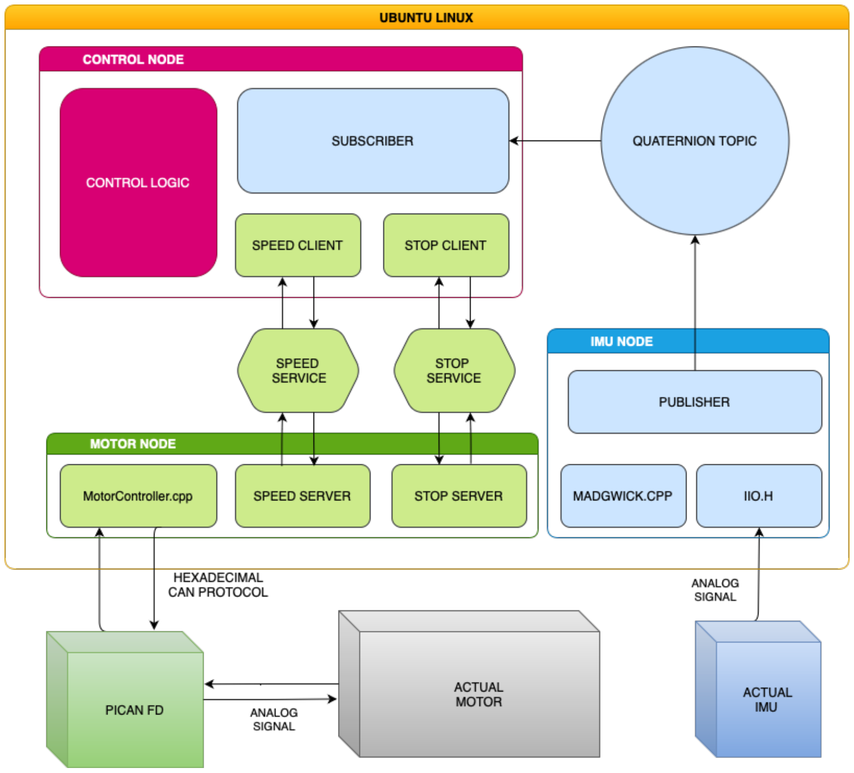


Figure 29: Overview of the motor control system built in ROS2 Linux, along with its hardware components.

- The IMU node reads analog signal data, puts it through a Madgwick filter, and publishes it as quaternions on to the quaternion topic.
- The control node subscribes to this topic, processes it, and prints it to the terminal.
- The control node also sends velocity commands through a client-server protocol to the motor via the PiCAN FD.
- The actuator receives the analog signals and responds by activating and sending response messages back all the way to the control node.

4.2.1 Control Logic

As of the final iteration of the code, the control logic in the control node may be described as a *Conditional Logistic Control System.*

The system operates as follows: The control node receives the motor response through its services, processes the incoming CAN message, and if conditions are met, a velocity message is sent based on the received torque current. The code is set up so that the speed is able to be adjusted to a certain degree per second (DPS) and sent to the actuator. However, an implementation for different thresholds corresponding to different velocity messages, or for a linear increase, is not implemented.

The graph below illustrates the control logistic present in the final control node, figure 30.

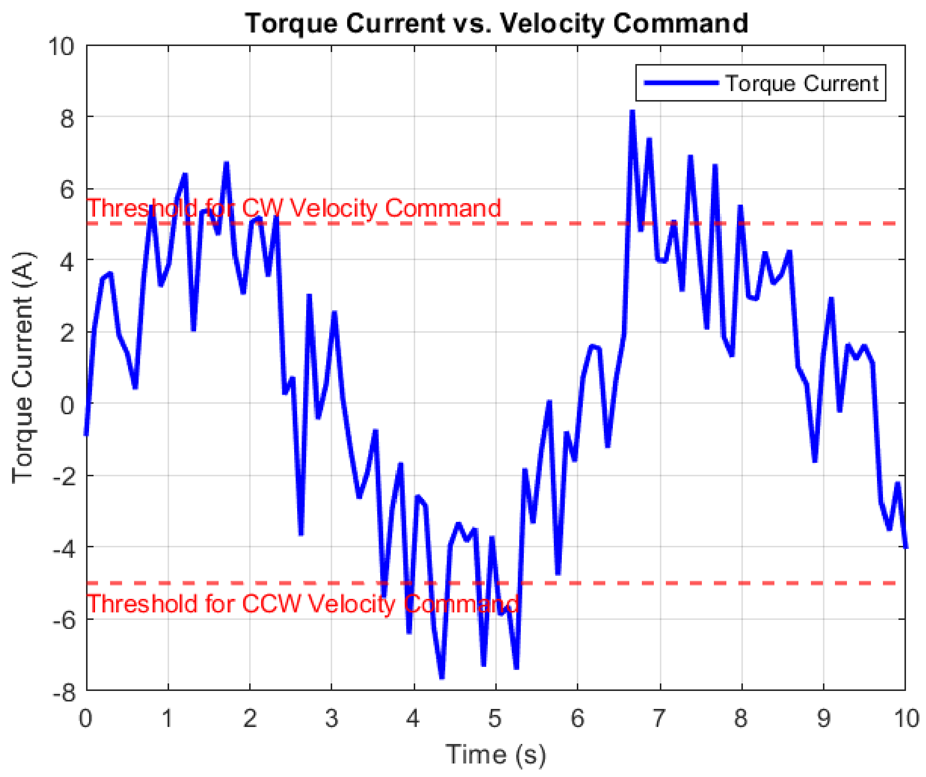


Figure 30: Matlab graph showing arbitrary thresholds for sending clockwise (CW) or counterclockwise (CCW) velocity commands to the actuator. See the Matlab code in appendix B.

A different approach was also conducted in an earlier iteration, which had the exoskeleton system respond to keyboard inputs in a way that the actuator could be controlled in real time. The software was constructed such that the velocity could be controlled linearly, meaning repeatedly sending the same velocity message would increase the DPS by that amount.

The code-files used to build the ROS2 architecture are provided in appendix D at the end of the report.

4.3 Visualization

Figure 31 displays real-time readings decoded from the low-byte and high-byte segments of the CAN response messages. The first plot shows the torque current, the second the actuator's angle with a wraparound at 360 degrees, and the third plots the angular velocity. On the x-axis all three graphs are plotting against the sample numbers. Note that this particular recording was done on simulated data, see appendix B.

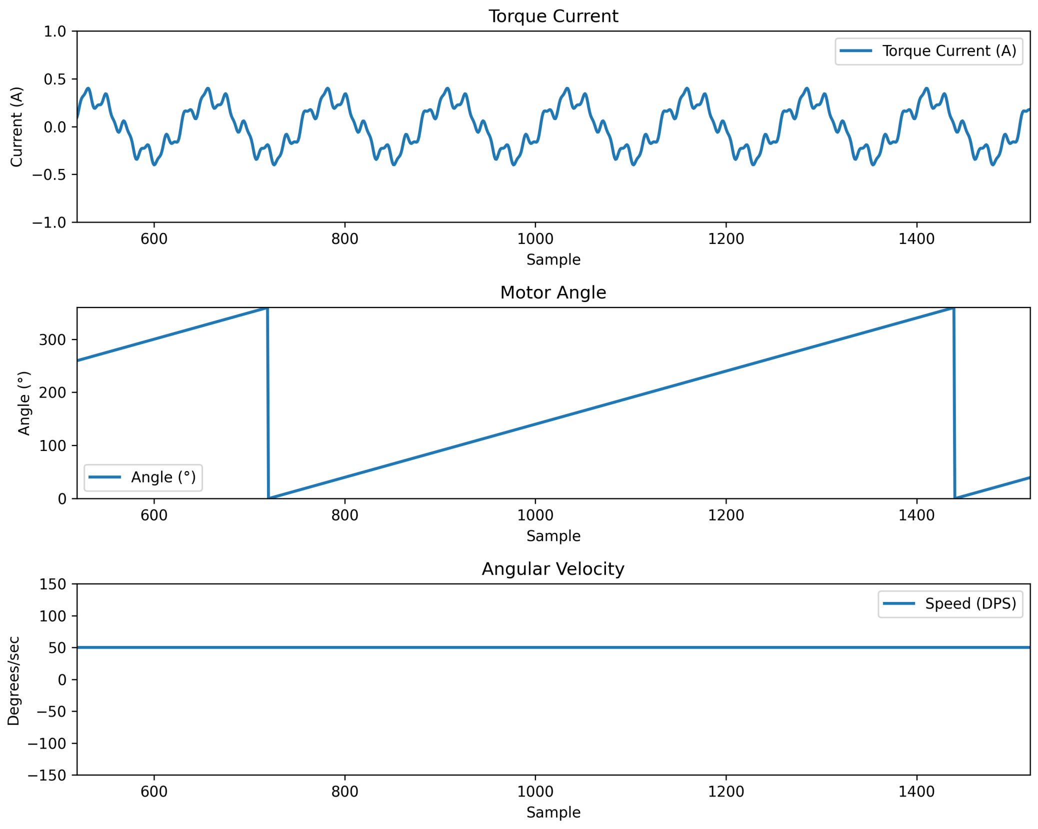


Figure 31: Visualization in real time

5 Discussion

This section discusses the results presented in chapter 4 and reflects on the final system performance along with encountered challenges. The discussion aims to evaluate the extent to which the project's purpose was achieved and to bring up the groups' key critical insights gained from the project. In the end, some ideas for future development of the exoskeleton are discussed.

5.1 Technical Insights of System and Lessons Learned

Given the prior knowledge of the group, a couple of misconceptions occurred during the project. This section highlights a couple of the problems and how they were solved, as well as the key inputs taken from the process.

5.1.1 Actuator Torque Sensor Misconception

During early integration, we operated under the assumption that the actuator featured a built-in torque sensor, allowing direct measurement of applied torque for closed-loop control. We spent considerable time exploring documentation and driver commands in search of a "read torque" command, only to discover that no such direct measurement exists. Instead, the applied torque must be estimated by calculating it from the motor's drawn current using the motor's torque constant, which was explained in subchapter 2.1.2. This approach requires precise current sensing and an accurate value for K_t which can be difficult to achieve due of noise from other electronics.

5.1.2 Choice of Operating System

ROS2 only supports Linux Ubuntu, Windows, and Red Hat Enterprise Linux. Because of the projects requirements requiring a fast response, but relatively well known and well documented, Linux Ubuntu was chosen. Windows is to slow and Red Hat Enterprise Linux is rather unknown to the public. Another viable option would have been Linux Raspbian (Raspberry Pi's official OS) because it is preconfigured with the Raspberry PI's IO Pins/-Ports in mind, allowing easier use. However this OS is not supported by ROS2, Meaning that we would have to use a docker image to use ROS2, introducing further delays, which ultimately removed it as a viable alternative.

5.1.3 Use of Microcontrollers

Microcontrollers were often mentioned in the project. Using a microcontroller was initially considered, but due to the software criteria and uncertainty of every aspect of compatibility in the project, a computer with OS was chosen. Not only was this selection a low-risk choice, it also provided flexibility and eliminated the immediate need to learn about microcontrollers. To preserve alternative options, this option was considered as a possible addition or reconfiguration of the system, provided there was time left. For example, a microcontroller could act as a middle ground, calculating data to relieve a master computer.

Given the project's time constraint, this approach minimized the risk of unexpected delays given the steeper learning curve of microcontrollers and the limited documentation of micro-ROS. The use of a full OS allowed the team to focus on several core aspects of the project without being immediately constrained by hardware-specific challenges.

5.1.4 PiCAN FD Misconceptions

After extensive research, several technical aspects of the PiCAN FD and the overall use of expansion cards on the Raspberry Pi remained uncertain. Details regarding various types of expansion cards, such as HATs (Hardware Attached on Top), were usually inconsistently documented. This lack of information meant that it was difficult to understand how these cards interacted with the GPIO header and other system components. Raspberry Pi itself provided limited official documentation about the general functionality of these expansion cards, leaving users dependent on fragmented information from third-party sources.

The PiCAN FD, being a niche product from a specialized manufacturer, was no exception. Despite extensive efforts to get clarifications from companies, online forums, and input from experienced people at Chalmers, it was eventually realized that the pin headers might not support pass-through connectivity as initially assumed and intended when selecting the part. The original plan being a compact solution of coupling the IMU on top of the PiCAN FD pass-through pins, allowing the relevant Raspberry Pi 5 GPIO signals to pass through.

However, further investigation suggested that this might be impossible. It was explained that the pins could potentially be replaced with longer pins, making them protrude from the pin-covers, enabling connection above the PiCAN. However, recognizing the significant risk of damaging the PiCAN module in combination with this possibly being a misconception, and might not even pass through the signals correctly or simply be redirected into the PiCAN itself, it was disregarded.

As a result, a new alternative wiring configuration was needed. This led to a less streamlined setup, where instead of the IMU resting on top of the PiCAN FD expansion board, they both had to share the Raspberry Pi 5's 40-pin header using additional cabling. This configuration likely compromised signal integrity and made the system more cumbersome to prototype with. The use of breadboards was considered but ultimately rejected due to concerns about reliability and the need for a stable presentable for a stable final configuration within the project's time frame.

Ultimately, the final configuration was a compromise - a functional but less elegant solutions than reflected the trade-offs between stability, signal integrity and ease of assembly.

5.1.5 IMU Connection

After successfully implementing the motor node connection, it felt natural to continue with integrating the IMU. The prior directive to use C++ in combination with ROS2 instead of Python, however, came with challenges. Learning C++ and ROS2 in addition to developing proved to be time consuming, thus the integration of the IMU was again de-prioritized in favor of developing a more refined but basic C++ based control system, integrated into ROS2 for the motor.

When attempting to finally connect the IMU, challenges related to the Raspberry Pi 5 configuration of GPIO pins were encountered. Unlike what was expected from extra research, the additional pins required precise placement due to preset software driver configurations. Further investigation revealed that the compatible IMU driver, originally designed for the ADIS16475, required matching pin configurations.

5.1.6 IMU Software Related Troubleshooting

Further issues occurred during IMU integration. Initial configuration attempts failed to produce data despite the correct setup of the IIO subsystem for SPI communication, and device tree overlay. After troubleshooting, we discovered that the Analog Devices driver for the ADIS16470 was installed in an incorrect directory, preventing GPIO initialization beyond the chip select (CS) line. After relocating the driver, we realized it was designed for a custom Linux kernel from Analog Devices, incompatible with the Ubuntu kernel used for ROS2. Since switching kernels would have required rebuilding the entire ROS2 environment, we opted to retain the Ubuntu kernel and adapt the existing setup.

By extracting necessary files from the Analog Devices kernel and modified the **adis16470-overlay.dts** file to resolve compilation errors by replacing macro definitions with direct values. Additional modifications to the Linux boot configuration allowed the IMU to initialize correctly.

5.2 CAD Development and Mechanical Design

A consequence of not finishing the addition of a fully functional control system, is that a lot of the theoretical research regarding the motion of the exoskeleton could not be implemented.

An example being the information presented in chapter 2.3 to 2.5.3. We wouldn't say the research of kinematics and dynamics of the system were wasted, even though the knowledge was not directly used. For future development, the implementation of the equations should be clear and easy to understand. The quaternion math is rather simple and should provide good calculations of the gravitational vector in real time.

Throughout the project, a significant amount of time was dedicated to developing a complete CAD model of the exoskeleton. The model, is in the end, a very important part of the mechanical foundation for all subsequent theoretical work. By designing and iterating on the geometry in CAD, we were able to determine the exact segment lengths, masses, center of mass positions, and moments of inertia. These physical parameters are essential for constructing accurate dynamic equations and simulating joint behavior. With the equations presented in chapter 2 this can easily be implemented in future work.

Although we did not reach the stage of physically assembling and testing the exoskeleton, it is capable of being used for future simulations and for finding mechanical limitations. The model also made it possible to extract data that would have been difficult to measure directly, such as the distance from the joint to the center of mass of the lower leg segment. Having these values makes it possible to formulate a realistic and mathematically grounded dynamic model.

The CAD design process is able to identify practical challenges early on, such as ensuring sufficient clearance for joint rotation, aligning motor and sensor placements, and maintaining structural integrity while minimizing weight. These considerations will be valuable when moving into the prototyping and testing phase in future iterations of the project. With the theoretical information provided in the report, the inputs in the control system will be calculated fairly easily.

5.3 Safety and Risk Management

In the course of integrating the actuator and power electronics, several safety-critical issues emerged that must be addressed before further development or human-testing can proceed. Below, we summarize the main hazards identified, illustrate a critical incident that underscored the system's risk potential and propose strategies for hardware, software, and mechanical safety improvements.

5.3.1 Voltage Discrepancy

Initially, the information given to the group was that the actuator operates with an input voltage of 24 V. Taking a closer look at the manufacturer's specifications revealed that it actually operates at 48 V. This not only increases the risk of a high voltage related accident, but also exceeds the permitted voltage level outside of the CASE laboratory at Chalmers. The original power supply given to us was also taken away as it was went against the regulations of Chalmers. Furthermore, the 48 V requirement effectively rules out using standard battery packs, as suitable 48 V battery solutions are not appropriate for our application due to their weight and size.

5.3.2 Mechanical and Kinetic Hazards

The kinetic hazards of a prototype such as the current exoskeleton, with the selected motor, are quite large. Mounting the actuator on the orthosis creates pinch points and unguarded rotating masses which could lead to severe crushing injuries. This, due to the fact that the actuator's rated torque capacity at 48 V is 30 Nm and therefore more than sufficient for rehabilitation tasks [11]. What generates an even larger risk is the peak torque of the motor at 60 Nm. With the measured shank length of the exoskeleton at $l = 0.31$ m, a peak torque would generate a force of 190 - 200 N, eq. 29:

$$\frac{\tau_{\text{peak}}}{l_{\text{shank}}} = \frac{60 \text{ Nm}}{0.31 \text{ m}} \approx 194 \text{ N} \approx 19.75 \text{ kg} \tag{29}$$

This means that at a peak torque, the motor applies almost 20 kg of force at the end of the bar. For a patient with knee injuries, this amount of force is dangerously high and could lead to further injuries if the system malfunctions. During the testing phase of the motor this actually happened and an error lead to the motor snapping one of the plastic safety pins. Not only did it show that the safety pins are not nearly sufficient enough, but also that some sort of software based safety mechanism is needed. In subchapter 5.3.3, we will further discuss possible strategies to reduce the risk concerning both the voltage and torque.

5.3.3 Risk Reduction Strategies

The system is far from safe enough for the medical field and many precautions will be needed before even a test prototype can be implemented. As the prototype is now, actions need to be done just to ensure the safety of future developers as well. We recommend the following measures before any user trials:

- **Power regulations:** The 48 V motor is designed to run at this voltage level and lowering the voltage would lead to an increase in electricity. Instead, putting a limit on the ampere in the power supply would indirectly limit the torque. Another alternative would be to change actuator to a model with a lower voltage input. The software architecture makes this option fairly easy.
- **Software safeguard:** Right now, the system does not have some sort of software safety stop. If more advanced software is developed, some sort of kill switch would need to be implemented. Alternately implement default limits in the control software with configurable parameters to prevent runaway commands.

- **Emergency stop and safety pins:** Implement a hard-wired emergency stop that cuts motor power instantly in case of software malfunction. Also mount safety pins of a material that is able to withstand full peak torque in case of emergency stop malfunction.

Implementing these recommendations will greatly reduce both the probability and severity of electrical or mechanical failures, establishing a safer baseline for the next phase of control system development and eventual human prototype testing.

5.4 Reflection on Achievements

Looking back, our most significant milestone was the successful integration of ROS2 into the system. We managed to build a working node architecture in which key hardware components, including an actuator and an IMU, could communicate reliably through the ROS2 publish-subscribe structure, which future teams may build upon.

Although not all planned features were achieved, we established the fundamental building blocks necessary for a functioning and expandable exoskeleton system. These include ROS2 communication, actuator interfacing, and real-time orientation sensing.

For us, this was the first time we worked with systems of this complexity, including ROS2 and embedded sensing. A significant portion of the project was therefore spent understanding the capabilities and limitations of the hardware, clarifying what functionality we actually needed, and determining how different components should interact. This exploration phase was time-consuming but essential as it shaped both the architecture and the scope of what became technically possible within the project timeline.

As this project involved many tools and technologies that were new to us, we quickly realized that extensive research and preparation were crucial. Despite our efforts to plan carefully, misunderstandings and misinterpretations still occurred along the way. While these challenges were tough, the group managed to solve the problems and the learning outcome from this project has been huge.

Finding compatible components proved to be particularly time-consuming under a tight schedule. We often chose safer and better documented alternatives to reduce the risk of time-consuming troubleshooting. Throughout the project, we consulted various sources and experts, but since we were constantly learning, it was to overlook important details or formulate the wrong questions. In the end, we learned that certain risks are unavoidable even with careful planning.

The key results of this project, as well as the gap between vision and current implementation, are summarized in the conclusion in chapter 6.

5.4.1 Reflection on Project Scope

Originally, the project's primary goal was to develop a control system, but as challenges with software integration and hardware compatibility emerged, the focus shifted towards solving embedded systems problems. This was largely driven by the complexity of transitioning to C++ and ROS2, as well as learning the unforeseen issues with IMU and GPIO configurations. This shift highlights how embedded system challenges is able to dominate a project intended for control system development, especially when integrating diverse hardware components during time constraints. In the last month of the project, the scope had changed to developing a working ROS2 architecture for future work. In that sense, we succeeded in reaching the scope.

5.5 Future Work

While the project achieved some strong results, it mainly lays a solid foundation for future improvements as previously stated. Below, we outline some of the next steps that could be worked on to further develop the exoskeleton system.

5.5.1 Control Development

As presented in the result chapter 4, the current control system is explained as a conditional logistic control system, meaning there is no feedback control in any way. While this logic has proven functional for basic movement demonstration, it needs further control development to become a transparent medical device in the future.

As the primary goal of the project was to achieve a fully transparent and user-supportive system, the next step would be to implement a self-regulating control system as well as pivot over to sending only torque messages instead of velocity. By choosing the users input y, as the desired torque and by estimating the disturbances, $\hat{D}$ through the IMU, encoder, measured torque current and the mass matrix, a disturbance observer as explained by in 2, could be implemented within a feedback loop.

The theoretical work in the report acts as a very good foundation for the above mentioned implementations. Creating an algorithm for reading and converting the IMU readings to quaternions, and then implementing simple quaternion math to find θ could be a good staring point, subchapter 2.4.4.

5.5.2 Design Development

While our results provide a solid foundation in terms of software and hardware integration, it largely overlooks the overall design aspects of the intended system. To ensure the sensor data can be used in a meaningful way, the sensors must be mounted on the exoskeleton, preferably like in our CAD drawings and connected using longer and more robust wiring.

This way, the published IMU topic and encoder data would no longer be redundant, but rather give meaningful information about how our system behaves. The current hardware connections are not sufficient either and an implementation of a custom PCB board could greatly reduce the complex wiring.

Lastly, as discussed in chapter 5.3.3, an implementation of better hardware safety pins is absolutely necessary. Even though this was not a part of the project scope, future project should take these safety precautions into consideration

6 Conclusion

At the beginning of this project, our goal was to take a pre-existing exoskeleton prototype and advance it with torque-controlled actuation and real-time motion sensing using an IMU. The long-term vision was to lay the groundwork for a transparent control system used for rehabilitation. One of the most important achievements was the successful integration of a working ROS2 architecture, connecting the system. We established a functioning node-based system that enabled key components, including the actuator and IMU, to communicate through a reliable and modular publish-subscribe framework.

We also made progress with the actuator, managing to establish open-loop control and send motion commands via CAN bus. These commands could also be sent manually by the press of a button. Although we initially had received information that the actuator had a built-in torque sensor, we later discovered this was not the case. Instead, the torque data needs to be based on the generated current. This discovery led to a shift in our approach, requiring the implementation of current sensing and the use of a torque constant to estimate torque. However, due to time constraints, we were not able to complete this implementation further than receiving an accurate current read.

Regarding IMU-integration we partially succeeded to integrate the IMU into the system, providing quaternion data at high frequency. While we did not get to the point that the IMU could be used to simulate the leg orientation, a Madgwick filter was successfully installed to convert the data to quaternions. The theoretical work needed for the quaternion math has been done and can as previously said, easily be incorporated.

Due to the limited time, a fully functioning feedback-based control system was not implemented by the end of the project. While we did establish a basic conditional logic system that could command the actuator based on IMU readings, no PID or disturbance observer-based control loop was finalized. Still, the structure is in place for these systems to be added.

In summary, we didn't reach full transparency or complete control functionality, but we did set up a working framework of software and hardware where actuator commands and IMU data flow in real time. As it will be easy for the next group to continue with the development, the result was a great success in the end. Not to mention, the amount of experience the project has given us a group. It will be a very useful experience for us and for future work.

References

[1] “Rehabilitation,” Available at https://www.who.int/news-room/fact-sheets/detail/rehabilitation (2025-04-17).

[2] “Statistik om hälso- och sjukvård samt regional utveckling 2021,” Available at https://skr.se/download/18.5e71e97518692446121b5588/1678982887387/Statistik%20om-halso%20och-sjukvard-samt-regional-utveckling_2021_2023-03-16.pdf (2025-04-17).

[3] P. Mutombo, “Impact of personalized rehabilitation programs on post-surgical recovery,” *Journal of Physical Therapy and Sports Medicine*, vol. 8, no. 4, 2024.

[4] B. J. C. G. P. K. G. J. L. J. -S. A. S. B. K. B. Wiśniowska-Szurlej A, Wołoszyn N, “Enhanced rehabilitation outcomes of robotic-assisted gait training with eksonr lower extremity exoskeleton in 19 stroke patients,” *Med Sci Monit.*, vol. 29, no. e940511, 2023.

[5] O.-C. M. Al-Tashi Mohammed, Lennartson Bengt and J. Fabian, “Classroom-ready open-source educational exoskeleton for biomedical and control engineering,” *at - Automatisierungstechnik*, vol. 72, no. 5, pp. 460–475, 2024.

[6] R. W. O Ott, L Ralfs, “Framework for qualifying exoskeletons as adaptive support technology,” *Frontiers in Robotics and AI*, vol. 9, 2022.

[7] H. B. V. K.-M. R. R. Fabian Just, Özhan Özen Philipp Bösch and G. Rauter, “Exoskeleton transparency: feed-forward compensation vs. disturbance observer,” *Automatisierungstechnik*, vol. 66, no. 12, 2018.

[8] S. M. A. N. H Björk, J Kurusakpadong, “Mechanical design improvement of exoskeleton for knee and elbow injury rehabilitatione,” 2025.

[9] “Film: Träning efter knäprotesoperation — 1177.se,” https://www.1177.se/Jonkopings-lan/undersokning-behandling/smartbehandlingar-och-rehabilitering/rehabilitering-i-jonkopings-lan/film-traning-efter-knaprotesoperation/, [Accessed 06-02-2025].

[10] R. Rao, “Robot actuators: A comprehensive guide to types, design, and emerging trends,” 2023.

[11] MyActuator, “Product details,” Available at https://www.myactuator.com/x8-60-details (2025-05-12).

[12] P. Yedamale, “Brushless dc (bldc) motor fundamentals,” Chandler, Arizona, USA, Tech. Rep., 2003.

[13] S. C. A. V. D. L. T. V. Pablo Lopez Garcia, Elias Saerens, “Factors influencing actuator’s backdrivability in human-centered robotics,” *MATEC Web Conf.*, vol. 366, no. 8, 2022.

[14] “Imu and quaternions,” https://www.steppeschool.com/pages/blog/imu-and-quaternions, [Accessed 01-05-2025].

[15] "Raspberry Pi 5," https://www.raspberrypi.com/products/raspberry-pi-5/, [Accessed 03-05-2025].

[16] D. C. H. L. J. C. C. C. S. L. G Yin, X Zhang, "Processing surface emg signals for exoskeleton motion control," *Frontiers in Neurorobotics*, vol. 14, 2020.

[17] N. Liang and P. Dobrivoje, "The CAN bus," in *Intelligent Vehicle Technologies*. Elsevier, 2001, pp. 21–64. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/B9780750650939500049

[18] LeCroy, "Canbus trigger, decode, and measure," 2008, unpublished.

[19] "Ros 2 documentation: Jazzy," https://docs.ros.org/en/jazzy/Concepts.html, [Accessed 02-05-2025].

[20] "C++ Vs Python: Overview, uses & key differences," https://www.simplilearn.com/tutorials/cpp-tutorial/cpp-vs-python, [Accessed 08-05-2025].

[21] "CAD Software Solutions," https://www.ptc.com/en/technologies/cad, [Accessed 04-05-2025].

[22] "Assembly Modeling: Meaning, Constraints & amp; CAD — Vaia — vaia.com," https://www.vaia.com/en-us/explanations/engineering/design-engineering/assembly-modeling/, [Accessed 04-05-2025].

[23] J. Nassour, "Robot dynamics," https://www.tu-chemnitz.de/informatik/KI/edu/robotik/ws2017/Dyn.pdf, 2018, [Accessed 09-05-2025].

[24] W. Dillingham, "Euler and Quaternion Angles: Differences and Why it Matters," https://resources.inertiallabs.com/en-us/knowledge-base/euler-and-quaternion-angles-differences-and-why-it-matters, [Accessed 02-05-2025].

[25] G. S. B Eater, "Quaternions," https://eater.net/quaternions, [Accessed 20-04-2025].

[26] "Mass matrix," https://www.sciencedirect.com/topics/engineering/mass-matrix, [Accessed 08-05-2025].

[27] L. Bengt, *Reglerteknikens grunder*, 4th ed. Lund: Studentlitteratur, 2002.

[28] "Quaternions," https://eng.libretexts.org/Bookshelves/Industrial_and_Systems_Engineering/Chemical_Process_Dynamics_and_Controls_(Woolf)/09%3A_Proportional-Integral-Derivative_(PID)_Control/9.02%3A_P_I_D_PI_PD_and_PID_control, [Accessed 16-04-2025].

[29] D. C. Lay, L. S. R., and J. J. McDonald, *Linear algebra and its applications*, sixth edition, global edition ed. Boston: Pearson, 2022.

[30] w. A. Becker, "Online Kalman Filter Tutorial." [Online]. Available: https://www.kalmanfilter.net/

[31] S. O. Madgwick, "An efficient orientation filter for inertial and inertial/magnetic sensor arrays," 2010.

[32] T. Lai, "IMU Madgwick filter explanation," Apr. 2024. [Online]. Available: https://medium.com/@k66115704/imu-madgwick-filter-explanation-556fbe7f02e3

[33] MatLab, "tune-pid-controller-to-favor-reference-tracking-or-disturbance-rejection," https://se.mathworks.com/help/control/getstart/.html, [Accessed 06-05-2025].

[34] Can Kutlu Yuksel, Tomáš Vyhlídal, Jaroslav Bušek, Milan Anderle Silviu-Iulian Niculescu, "Harmonic disturbance compensation of a system with long dead-time, design and experimental validation," https://hal.science/hal-04071873v1/file/CKY_TMech_2023.pdf, 2023, [Accessed 06-05-2025].

[35] H. Alexander, "Introduction to disturbance observers and disturbance rejection using disturbance observers," https://aleksandarhaber.com/introduction-to-disturbance-observers-and-controllers-based-disturbance-observers/, 2023, [Accessed 06-05-2025].

[36] C. Mathas, "Analog rankings: Top 10 suppliers own 68% market share," Planet Analog [Online]. Available at https://www.planetanalog.com/analog-rankings-top-10-suppliers-own-68-market-share/, 2022, [Accessed 13-05-2025].

[37] P. Clarke, "ADI edges closer to TI atop analog IC supplier ranking," eeNews Europe [Online]. Available at https://www.eenews-europe.com/adi-edges-closer-to-ti-atop-analog-ic-supplier-ranking/, 2022, [Accessed 13-05-2025].

[38] A. M. Phan, M. B. Joplin, and M. J. Campola, "Single-Event Effect Test Report: Analog Devices LTC1604AIG#90117 ADC," NASA, NASA/TM–20210024585, nov 2021, [Online]. Available at https://ntrs.nasa.gov/archive/nasa/casi.ntrs.nasa.gov/20210024585.pdf (Accessed 13-05-2025).

[39] https://www.analog.com/en/products/adis16470.html#documentation, [Accessed 13-05-2025].

[40] "Pican fd med lin-bus," https://www.electrokit.com/pican-fd-med-lin-bus, [Accessed 03-05-2025].

[41] "Raspberry Pi 5 Board ONLY — argon40.com," https://argon40.com/en-au/products/raspberry-pi-5-board, [Accessed 13-05-2025].

[42] "Downloads-X Series — MyActuator — myactuator.com," https://www.myactuator.com/downloads-xseries, [Accessed 13-05-2025].

[43] "Industrial i/o," https://www.kernel.org/doc/html/v4.14/driver-api/iio/index.html, [Accessed 09-05-2025].

[44] Arduino, "Madgwickahrs," https://github.com/arduino-libraries/MadgwickAHRSl, 2024, [Accessed 09-05-2025].

[45] "Socketcan," https://docs.kernel.org/networking/can.html, [Accessed 13-05-2025].

[46] "Products widely used," https://www.myactuator.com/_files/archives/cab28a_5b42a01f1a4a44c3a4e0f1ac6ecb22ff.zip?dn=(X)%20Protocol%20and%20manual%20of%20V2-250213.zip, 2021, [Accessed 10-05-2025].

[47] "ChatGPT — chatgpt.com," https://chatgpt.com/, [Accessed 13-05-2025].

[48] "Microsoft Copilot: Your AI companion — copilot.microsoft.com," https://copilot.microsoft.com, [Accessed 13-05-2025].

[49] "DeepL Translate: The world's most accurate translator — deepl.com," https://www.deepl.com/en/translator, [Accessed 13-05-2025].

[50] B. Granström, "LibGuides: Search and evaluate information: Artificial intelligence (AI)." [Online]. Available: https://guides.lib.chalmers.se/sokaochutvarderainformation/EN/ai

[51] L. S. Lo, "The CLEAR path: A framework for enhancing information literacy through prompt engineering," *The Journal of Academic Librarianship*, vol. 49, no. 4, p. 102720, Jul. 2023. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S0099133323000599

A Appendix

A: Usage of Artificial Intelligence and Smart Software

Throughout this project, various intelligent tools such as generative AI (artificial intelligence), translation and smart writing applications have been used to increase productivity, understanding and learning. It should also be disclosed that our supervisor, Fabian Just, highly suggested this. The most commonly used have been OpenAI's ChatGpt, Microsoft's Copilot for generative AI and deepL for translations and academic writing [47–49].

The use of smart generative programs can be a great at automating tasks that were previously time-consuming or labor-intensive. In our code it has for example been used for understanding the CmakeList and package.xml files and for rewriting pseudo-code into actual code.

However, when using generative AI, it is important to always be vigilant about the data provided or the methods derived from it. AI bias, hallucinations, and reliance on fake sources are all challenges associated with generative AI [50].

To try and derive more useful prompts from generative AI, the CLEAR (Concise, Logical, Explicit, Adaptive, Reflective) framework, have been used throughout the project [51]. The CLEAR framework is "a concise and user-friendly method tailored to optimize interactions with generative AI language models". Using CLEAR, an engineered prompt for code used in our project could look something like this:

"Rewrite the Python method that creates the three separate plots in Live-Plot.py so that each plot is a subplot."

When using smart writing tools like *deepL*, a prompt would not necessarily adhere to the CLEAR framework. Instead, a prompt would simply be a paragraph or a string of text that the smart application would rewrite in the style of either casual, academic, business or other chosen writing style. For example:

" *As said before, using a combination of these methods can improve the results by making the most of each controller's strengths.* "

$\Longrightarrow$ **Chosen style: *Academic*** $\Longrightarrow$

" *Combining these methods may, as earlier stated, result in a better output by utilizing the strengths of each controller.* "

B Appendix

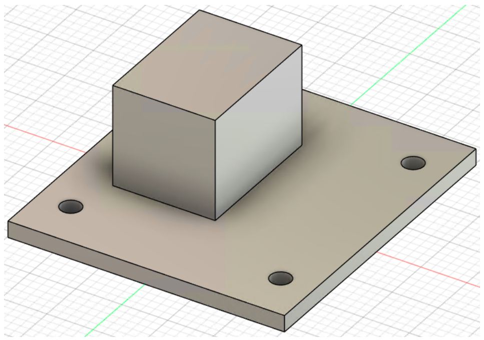
B: Matlab Code

```matlab


t = linspace(0, 10, 100);
torque_current = 5 * sin(t) + 2 * randn(size(t));

threshold_upper = 5;
threshold_lower = -5;

    % Plot
figure;
plot(t, torque_current, 'b-', 'LineWidth', 2);
hold on;

yline(threshold_upper, '--r', 'LineWidth', 1.5);
yline(threshold_lower, '--r', 'LineWidth', 1.5);


text(0, threshold_upper + 0.5, 'Threshold for CW Velocity Command'
, 'Color', 'r', 'FontSize', 10, 'HorizontalAlignment', 'left');
text(0, threshold_lower - 0.5, 'Threshold for CCW Velocity
Command',
'Color', 'r', 'FontSize', 10, 'HorizontalAlignment', 'left');


xlabel('Time (s)');
ylabel('Torque Current (A)');
title('Torque Current vs. Velocity Command');
grid on;
legend('Torque Current');
hold off;

%exportgraphics(gcf, 'tvc.pdf', 'ContentType', 'vector');
```

```matlab
% Easy Plot of  lp and hp filter
fs = 1000;
f = linspace(0, fs/2, 1000);
fc = 50;
omega = 2*pi*f/fs;
omega_c = 2*pi*fc/fs;

H_lp = 1 ./ sqrt(1 + (f/fc).^2);
H_hp = (f/fc) ./ sqrt(1 + (f/fc).^2);

    % Plot
figure;
subplot(1,2,1);
plot(f, H_lp, 'b', 'LineWidth', 2);
title('Low-pass Filter'); xlabel('Frequency (Hz)'); label('
    Magnitude');
grid on;
ylim([0 1.1]);

subplot(1,2,2);
plot(f, H_hp, 'r', 'LineWidth', 2);
title('High-pass Filter'); xlabel('Frequency (Hz)'); label('
    Magnitude');
grid on;
ylim([0 1.1]);
```

C Appendix

C: CAD


(a) CAD of ADIS16470.

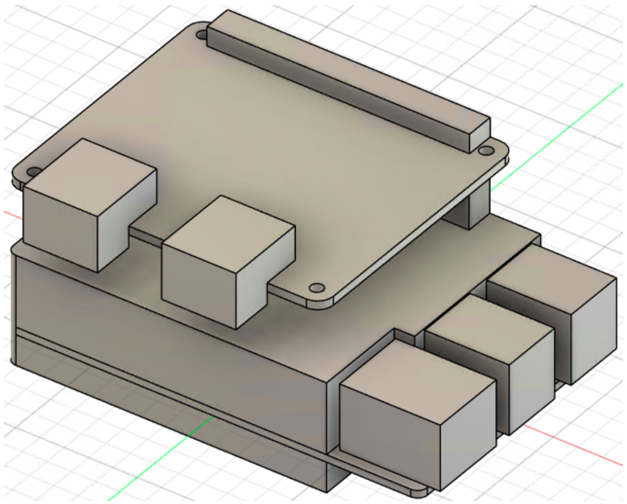
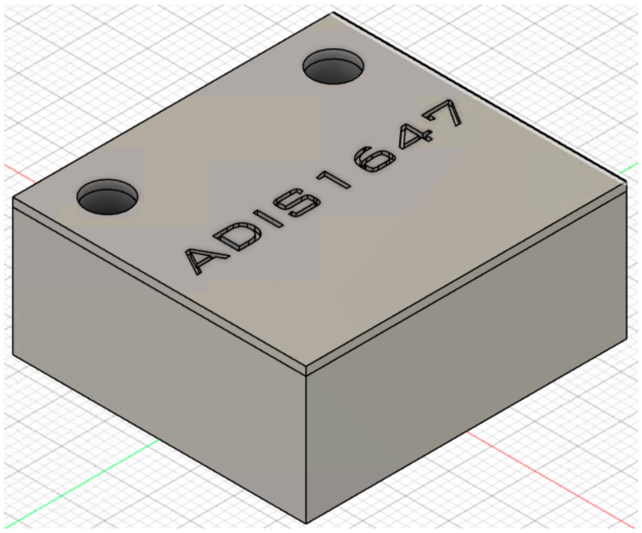
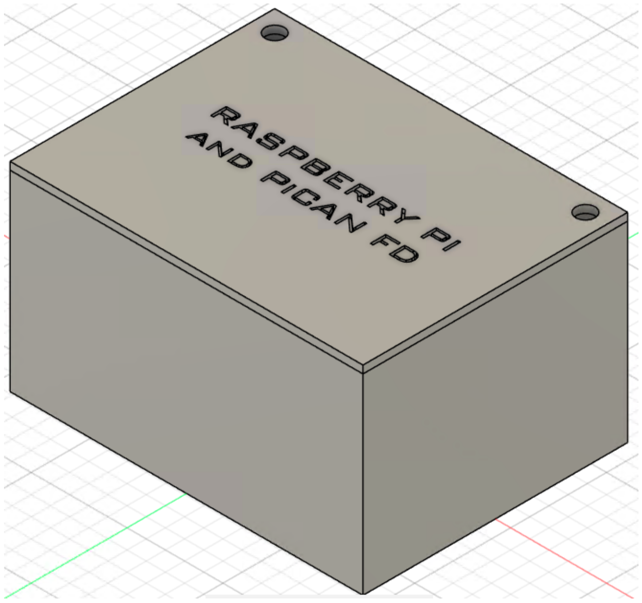

(b) CAD of Raspberry Pi with a fan and Pi-CAN FD on top on them, with pins in between.

Figure C.1: The circuit boards in CAD.



(a) Case with the ADIS inside.



(b) Case with Raspberry Pi and PiCAN FD inside.

Figure C.2: CAD of the cases with the circuit boards inside.

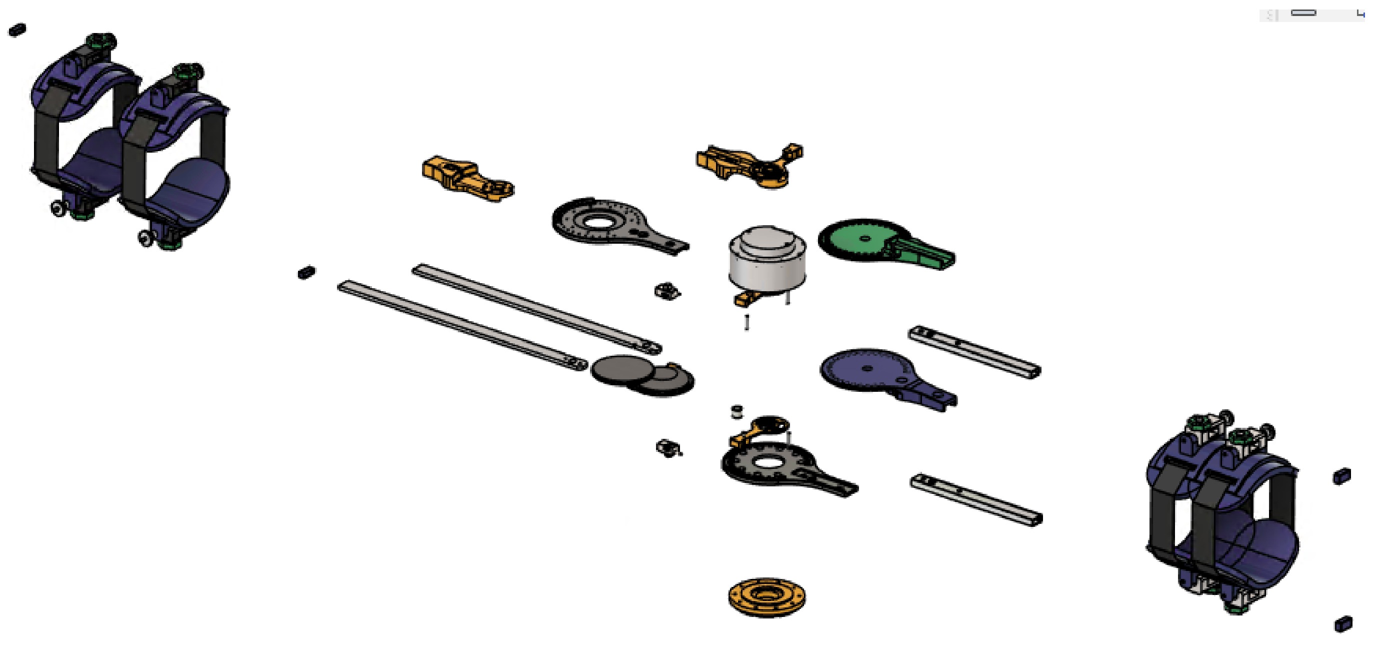


Figure C.3: Exploded view of the exoskeleton.

D Appendix

D: ROS2 Code

Control Node

Listing 1: Control Node

```cpp
    #include "tutorial_interfaces/srv/stop_motor.hpp"
#include "tutorial_interfaces/srv/speed_control.hpp"
#include "rclcpp/rclcpp.hpp"
#include "std_msgs/msg/string.hpp"
#include "geometry_msgs/msg/quaternion.hpp"

#include <chrono>
#include <cstdlib>
#include <memory>
#include <termios.h>
#include <unistd.h>


using namespace std::chrono_literals;

using speed_srv = tutorial_interfaces::srv::SpeedControl;
using stop_srv = tutorial_interfaces::srv::StopMotor;

geometry_msgs::msg::Quaternion latest_imu_data;

// #Function to wait for input, return character
int getch(){
  struct termios oldt, newt;
    int ch;
    tcgetattr(STDIN_FILENO, &oldt);
    newt = oldt;
    newt.c_lflag &= ~(ICANON | ECHO);
    tcsetattr(STDIN_FILENO, TCSANOW, &newt);
    ch = getchar();
    tcsetattr(STDIN_FILENO, TCSANOW, &oldt);
    return ch;
 }

/// Sends a speed command to motor node and returns a result
    containing torque and other stats
std::shared_ptr<speed_srv::Response> node_send_speed_command(const std
    ::shared_ptr<rclcpp::Node>& node, const rclcpp::Client<speed_srv
    >::SharedPtr& client, int speed)
{
  auto request = std::make_shared<speed_srv::Request>();
  request->speed_control = speed;

  auto result = client->async_send_request(request);
  if (rclcpp::spin_until_future_complete(node, result) ==
      rclcpp::FutureReturnCode::SUCCESS)
  {
    auto res = result.get();
      return res;
  } else {
    printf("Failed to call service\n");
    return nullptr;
  }
}

/// sends a stop command to motor node, returns nothing.
void node_send_stop_command(const std::shared_ptr<rclcpp::Node>& node,
     const rclcpp::Client<stop_srv>::SharedPtr& client)
{
  auto request = std::make_shared<stop_srv::Request>();
  auto result = client->async_send_request(request);

  if (rclcpp::spin_until_future_complete(node, result) != rclcpp::
      FutureReturnCode::SUCCESS)
  printf("Failed to call STOP service\n");
  else
  printf("Stop command sent!\n");
}

//--- CLASS: Simple 1D-Kalman for Torque
class Kalman1D {
  double x_est;  // Estimated torque
  double P;      // Estimeted kovarians
  const double Q, R;  // process- and read-noise
public:
  Kalman1D(double Q_, double R_, double P0 = 1.0, double x0 = 0.0)
    : x_est(x0), P(P0), Q(Q_), R(R_) {}
  double update(double z) {
    // 1) Predicted
    P += Q;
    // 2) Calculate Kalman Gain
    double K = P / (P + R);
    // 3) Correct the prediction
    x_est += K * (z - x_est);
    // 4) Update co-variance
    P *= (1.0 - K);
    return x_est;
  }
};

// Saves the recieved imu data to global variable in current file. (
    Quaternion)
void imu_callback(const geometry_msgs::msg::Quaternion::SharedPtr msg)
{
    latest_imu_data = *msg;
```

```cpp
}

int main(int argc, char **argv)
{
  rclcpp::init(argc, argv);
  rclcpp::Rate loop_rate(500);

  auto node = rclcpp::Node::make_shared("control_node");
  auto client = node->create_client<speed_srv>("motor_speed_service");
  auto stop_client = node->create_client<stop_srv>("motor_stop_service");
  auto imu_sub = node->create_subscription<geometry_msgs::msg::Quaternion>("imu/quaternion", 10, imu_callback);

  while (!client->wait_for_service(1s)  ) {
    if (!rclcpp::ok()) {
      RCLCPP_ERROR(rclcpp::get_logger("rclcpp"), "Interrupted while waiting for the service. Exiting.");
      return 0;
    }
    % RCLCPP_INFO(rclcpp::get_logger("rclcpp"), "service not available, waiting again..."); //kolla p   can stop i denna situation
  }

  RCLCPP_INFO(node->get_logger(), "Control node is running. Subscribed to IMU data.");

  // ---------------
  // Rudamentary 1Dkalman filter, hysteres and lowpass (attempt)
  //
  // ###Explained:
  // Q = Expected process-noise | Lower val = slower response of filter |
  // R = Torque sensor - Noise  | Higher val = flattens signal more | Note: The motor amplitude is noisy!
  //
  //
  //-------------------------------------------------

  int speed = 0;
  Kalman1D torqueKF(/*Q=*/1e-1, /*R=*/2e-2);
  bool engaged = false;
  const double thr_hi = 0.15;  // For what current amplitude should "engage" be activated | Starts to react to user hand over thr_hi | hi must be over low!
  const double thr_lo = 0.13;  // For what current amplitude should it not react
  const double gain     = 25000.0;  // Gain dps per Nm | Small torque val (usually 0.002) must have this multiple to become a high enough speed | (5000-25000 range is good)
  const int    max_spd   = 100;    // A maximum speed for safety precautions.
  int fall_count = 0;               //
  const int max_fall = 150;         // 150 * 2 ms = 300 ms  grace period

  //----------------Exponential lowpass filter-----
  // Use: Supposed to filter out the noise that is not human
  //
  //--------------------
  double torque_lp = 0.0;
  const double alpha_lp = 0.05;    // cutoff  3 Hz at 500Hz sampling

  // ---Simple 500Hz loop with Kalman + lowpass + hysteres---
  while (rclcpp::ok()) {
    auto res = node_send_speed_command(node, client, speed);
    if (!res) {
      loop_rate.sleep();
      continue;
    }
//############################### START MODE 1: KALMAN FILTERING CONTROL ####

  //   double torque_raw = res->torque_current;
  //   double torque_f   = torqueKF.update(torque_raw);

  //   // --- APPLY LOWPASS ---
  //   torque_lp = alpha_lp * torque_f + (1.0 - alpha_lp) * torque_lp;

  // // --- HYSTERES & PROPORTIONAL ---
  // //
  // // Essentially: Logic behind what will distinguish human from noise
  // //
  // // Engaged loop is supposed to trigger when a human pushes
  // // Current problem: Wont disengage and therefore takes noise input as 'Human input'
  // // ---> Therefore it "wobbles"
  // //

  //   if (!engaged) {                  //If it is not in engaged mode (Which is always, with the exception of human input)
  //     if (fabs(torque_lp) > thr_hi) {            //If the absolute value of lowpass-current amplitude...
  //       engaged    = true;                     //...is over hi-threshold value: engage (We have human input)
  //       fall_count = 0;                        //Counter sets to zero (Remember, if it reaches 150 (300ms), it will count it as 'disengaged') - See fall_count++ below
  //       int s = std::min(max_spd, int(fabs(torque_lp) * gain));   //s is speed, has gain as a multiple
  //       speed = (torque_lp > 0 ? -s : +s);     //speed is speed with directions taken into consideration based
```

```
    on torque sign
//      } else {
    // Not engaged and not enough input, make sure speed is zero
//        speed = 0;
//      }
//    } else {
    //If it IS engaged...
//      if (fabs(torque_lp) > thr_lo) {
    //and absolute value of lowpassed-current amplitude is over low
    threshold:
//        fall_count = 0;
    //(Start counter)
//        int s = std::min(max_spd, int(fabs(torque_lp) * gain));
    //^
//        speed = (torque_lp > 0 ? -s : +s);
    //^

//      } else {
//       //Signal is too weak: Start fall count
//        fall_count++;
//        if (fall_count >= max_fall) {
    //If fall count reaches max_fall, disengage, set speed to 0
//          engaged    = false;
//          speed      = 0;
//          node_send_stop_command(node, stop_client);
    //Really sets speed to zero, we wont risk

//          fall_count = 0;
//        }
//      }
//    }
//    RCLCPP_INFO(node->get_logger(), "Quaternion: [w=%+8.4f, x=%+8.4
    f, y=%+8.4f, z=%+8.4f]   |  Raw Torque: [t=%+8.4f], Kfilt: %+8.4
    f   ,  LP: %+8.4f   ,  eng=%d        spd=%d\n"
//    , latest_imu_data.w, latest_imu_data.x, latest_imu_data.y,
    latest_imu_data.z, res->torque_current, torque_f, torque_lp, (
    int)engaged, speed);

//############### END MODE 1: KALMAN FILTERING CONTROL

//##################  START MODE 2: CONSTANT SPEED TORQUE CONTROL
//---Before uncommenting this, comment the kalman-filter-loop ----
      if (res->torque_current > 0.2) {
        speed = -30;
      }
      else if (res->torque_current < -0.2) {
        speed = 30;
      }
      RCLCPP_INFO(node->get_logger(), "Quaternion: [w=%+8.4f, x
          =%+8.4f, y=%+8.4f, z=%+8.4f]   |  Raw Torque: [t=%+8.4f]"
      , latest_imu_data.w, latest_imu_data.x, latest_imu_data.y,
          latest_imu_data.z, res->torque_current);
```

```
  //##################END MODE 2: CONSTANT SPEED TORQUE CONTROL
  }

  //################ ALWAYS KEEP THIS #####################
    rclcpp::spin_some(node);

    node_send_stop_command(node, stop_client);
    rclcpp::shutdown();
    return 0;
  }
  //
// }

// Simplest possible - keyboard driven control
// printf("Press 'd' for positive speed, 'a' for negative speed, 's'
    for stop, 'q' to quit...\n");

    // int c = getch();

    // if (c == 'd') {
    //   speed = 10; // add positive speed
    //   node_send_speed_command(node, client, speed);
    // } else if (c == 'a') {
    //   speed = -10; // add negative speed
    //   node_send_speed_command(node, client, speed);
    // }else if (c == 's') { // s to stop
    //   node_send_stop_command(node, stop_client);
    //   speed = 0; // set speed to 0
    // }else if (c == 'q') {
    //   node_send_stop_command(node, stop_client);
    //   speed = 0; // Set speed to 0
    //   break;
    // } else {
    //   continue;
    // }
  //}
```

Motor Node

Listing 2: Motor Node

```cpp
    #include "rclcpp/rclcpp.hpp"
#include "tutorial_interfaces/srv/speed_control.hpp"
#include "tutorial_interfaces/srv/stop_motor.hpp"
#include "MotorController.hpp"

#include <memory>

using speed_srv = tutorial_interfaces::srv::SpeedControl;
using stop_srv = tutorial_interfaces::srv::StopMotor;

MotorController motor_controller;

void handle_speed_control(const std::shared_ptr<speed_srv::Request> request, std::shared_ptr<speed_srv::Response> response)
    {
        motor_controller.send_speed_command(request->speed_control);
        auto frame = motor_controller.recieve_ack();
        auto result = motor_controller.decode_ack(frame);

        response->angle = result.angle;
        response->speed = result.speed;
        response->torque_current = result.torque_current;
        response->temperature = result.temperature;
    }

void handle_stop_motor(const std::shared_ptr<stop_srv::Request>, std::shared_ptr<stop_srv::Response>)
    {
        motor_controller.send_stop_command();
    }

int main(int argc, char **argv) //int argc  r typ antalet eller len(argv), argv  r en vektor med alla argument
{                               //**anv nds bara f r att det  r en vektor. F rsta * menar att hela vektorn  r fylld av pointers. den andra "*" g r att pointers kan peka till str ngen/inten
    rclcpp::init(argc, argv);   //init startar ros, nu kan den lyssna efter enbart argc och argv, den beh ver inte parsea

    std::shared_ptr<rclcpp::Node> node = rclcpp::Node::make_shared("motor_node");
    // rclcpp::Service<my_srv_file>::SharedPtr service =
    //   node->create_service<my_srv_file>("motor_service", &add); // Den tar node som  r en shared pointer

    rclcpp::Service<speed_srv>::SharedPtr service = node->create_service<speed_srv>("motor_speed_service", handle_speed_control);
    rclcpp::Service<stop_srv>::SharedPtr stop_service = node->
        create_service<stop_srv>("motor_stop_service",
        handle_stop_motor);

    RCLCPP_INFO(rclcpp::get_logger("rclcpp"), "rdy to send motor cmd")
        ;

    rclcpp::spin(node); /
    rclcpp::shutdown();
}
```

IMU Node

Listing 3: IMU Node

```
    #include <chrono>
#include <functional>
#include <memory>
#include <string>
#include <cmath>
#include <iio.h>
#include "rclcpp/rclcpp.hpp"
#include "geometry_msgs/msg/quaternion.hpp"
#include "MadgwickAHRS.h"

using namespace std::chrono_literals;

class IMUPublisher : public rclcpp::Node
{
public:
    IMUPublisher()
        : Node("imu_publisher"), madgwick()
    {
        publisher_ = this->create_publisher<geometry_msgs::msg::
            Quaternion>("imu/quaternion", 10);
        timer_ = this->create_wall_timer(2ms, std::bind(&IMUPublisher
            ::timer_callback, this));

        // Initialize libiio context for ADIS16470
        ctx = iio_create_default_context();
        if (!ctx) {
            RCLCPP_ERROR(this->get_logger(), "Failed to create IIO
                context");
            rclcpp::shutdown();
            return;
        }

        dev = iio_context_find_device(ctx, "adis16470");
        if (!dev) {
            RCLCPP_ERROR(this->get_logger(), "Failed to find ADIS16470
                device");
            rclcpp::shutdown();
            return;
        }

        gyro_x = iio_device_find_channel(dev, "anglvel_x", false);
        gyro_y = iio_device_find_channel(dev, "anglvel_y", false);
        gyro_z = iio_device_find_channel(dev, "anglvel_z", false);
        accel_x = iio_device_find_channel(dev, "accel_x", false);
        accel_y = iio_device_find_channel(dev, "accel_y", false);
        accel_z = iio_device_find_channel(dev, "accel_z", false);
    }

private:
    void timer_callback()
    {
        float gx, gy, gz, ax, ay, az;

        // Read IMU data
        if (!read_sensor_data(gx, gy, gz, ax, ay, az))
        {
            RCLCPP_ERROR(this->get_logger(), "Failed to read IMU data"
               );
            return;
        }

        // Update Madgwick filter
        madgwick.updateIMU(gx, gy, gz, ax, ay, az);

        // Publish quaternion
        auto quaternion_msg = geometry_msgs::msg::Quaternion();
        quaternion_msg.w = madgwick.q0;
        quaternion_msg.x = madgwick.q1;
        quaternion_msg.y = madgwick.q2;
        quaternion_msg.z = madgwick.q3;

        RCLCPP_INFO(this->get_logger(), "Publishing quaternion: [%.4f,
             %.4f, %.4f, %.4f]",
                    quaternion_msg.w, quaternion_msg.x, quaternion_msg
                        .y, quaternion_msg.z);
        publisher_->publish(quaternion_msg);
    }

    bool read_sensor_data(float &gx, float &gy, float &gz, float &ax,
        float &ay, float &az)
    {
        if (!gyro_x || !gyro_y || !gyro_z || !accel_x || !accel_y || !
            accel_z)
        {
            return false;
        }
        char buffer[32];
        int16_t raw_gx, raw_gy, raw_gz;
        int16_t raw_ax, raw_ay, raw_az;

        // Read gyroscope data
        iio_channel_attr_read(gyro_x, "raw", buffer, sizeof(buffer));
        raw_gx = static_cast<int16_t>(std::stoi(buffer));

        iio_channel_attr_read(gyro_y, "raw", buffer, sizeof(buffer));
        raw_gy = static_cast<int16_t>(std::stoi(buffer));

        iio_channel_attr_read(gyro_z, "raw", buffer, sizeof(buffer));
        raw_gz = static_cast<int16_t>(std::stoi(buffer));

        // Read accelerometer data
        iio_channel_attr_read(accel_x, "raw", buffer, sizeof(buffer));
        raw_ax = static_cast<int16_t>(std::stoi(buffer));

        iio_channel_attr_read(accel_y, "raw", buffer, sizeof(buffer));
        raw_ay = static_cast<int16_t>(std::stoi(buffer));

        iio_channel_attr_read(accel_z, "raw", buffer, sizeof(buffer));
        raw_az = static_cast<int16_t>(std::stoi(buffer));

        // Scaling values for IMU (depends on the sensor)
        gx = raw_gx * 0.000000026f; // Example scale
        gy = raw_gy * 0.000000026f;
        gz = raw_gz * 0.000000026f;
        ax = raw_ax * 0.000000187f;     // Example scale
        ay = raw_ay * 0.000000187f;
        az = raw_az * 0.000000187f;

        return true;
    }

    rclcpp::TimerBase::SharedPtr timer_;
    rclcpp::Publisher<geometry_msgs::msg::Quaternion>::SharedPtr
       publisher_;

    // IMU Variables
    struct iio_context *ctx;
    struct iio_device *dev;
    struct iio_channel *gyro_x, *gyro_y, *gyro_z;
    struct iio_channel *accel_x, *accel_y, *accel_z;
    Madgwick madgwick;
};

int main(int argc, char *argv[])
{
    rclcpp::init(argc, argv);
    rclcpp::spin(std::make_shared<IMUPublisher>());
    rclcpp::shutdown();
    return 0;
}
```

Motor Controller

Listing 4: Motor Controller

```
    #include "MotorController.hpp"
#include <iostream>
#include <cstring>
#include <sys/types.h>
#include <sys/socket.h>
#include <net/if.h>
#include <sys/ioctl.h>
#include <linux/can.h>
#include <linux/can/raw.h>
#include <unistd.h>
#include <cstdint>

///Constructor
MotorController::MotorController() {
    std::cout << "MotorController constructed, can socket open" << std
        ::endl;
    can_socket_ = socket(PF_CAN, SOCK_RAW, CAN_RAW);
    if (can_socket_ < 0) {
        std::cerr << "Error opening CAN socket" << std::endl;
        return;
    }
    struct ifreq ifr;
    std::strcpy(ifr.ifr_name, "can0"); // ai generated code: use can0
        interface
    if (ioctl(can_socket_, SIOCGIFINDEX, &ifr) < 0) {
        std::cerr << "Error getting CAN interface index" << std::endl;
        close(can_socket_);
        can_socket_ = -1;
        return;
    }
    struct sockaddr_can addr;
    addr.can_family = AF_CAN;
    addr.can_ifindex = ifr.ifr_ifindex;
    if (bind(can_socket_, (struct sockaddr *)&addr, sizeof(addr)) < 0)
         {
        std::cerr << "Error binding CAN socket" << std::endl;
        close(can_socket_);
        can_socket_ = -1;
        return;
    }
    std::cout << "CAN socket initialized on can0" << std::endl;
}

/// Destructor
MotorController::~MotorController() {
    std::cout << "MotorController destructed, can socket closed" <<
        std::endl;
    // ai generated code: close CAN socket if open
    if (can_socket_ >= 0) {
        send_stop_command();
        close(can_socket_);
        std::cout << "CAN socket closed" << std::endl;
    }
}

// method
void MotorController::send_speed_command(int speed) {
    std::cout << "Sending speed command: " << speed << std::endl;
    struct can_frame frame = encode_speed_command(speed);
    // ai generated code: send frame over CAN socket
    if (can_socket_ < 0) {
        std::cerr << "CAN socket not initialized" << std::endl;
        return;
    }
    int nbytes = write(can_socket_, &frame, sizeof(frame));
    if (nbytes != sizeof(frame)) {
        std::cerr << "Error sending CAN frame" << std::endl;
    } else {
        std::cout << "CAN frame sent" << std::endl;
    }
}

//method
struct can_frame MotorController::recieve_ack() {
    std::cout << "Receiving CAN ack..." << std::endl;
    struct can_frame frame;
    std::memset(&frame, 0, sizeof(frame));
    // ai generated code: read frame from CAN socket
    if (can_socket_ < 0) {
        std::cerr << "CAN socket not initialized" << std::endl;
        return frame;
    }
    int nbytes = read(can_socket_, &frame, sizeof(frame));
    if (nbytes < 0) {
        std::cerr << "Error reading CAN frame" << std::endl;
    } else if (nbytes < sizeof(frame)) {
        std::cerr << "Incomplete CAN frame received" << std::endl;
    } else {
        std::cout << "CAN frame received" << std::endl;
    }
    return frame;
}

struct can_frame MotorController::encode_speed_command(int32_t
    speed_dps) {
    struct can_frame frame;
    std::memset(&frame, 0, sizeof(frame));
    frame.can_id  = 0x141;   // 0x140 + motor ID = 0x141 f r ID=1
    frame.can_dlc = 8;

    frame.data[0] = 0xA2;    // Speed closed-loop command
```

```cpp
    // bytes 1   3 = 0
    // Skala till raw-enhet 0.01 dps/LSB
    int32_t speed_raw = speed_dps * 100;
    uint32_t u = static_cast<uint32_t>(speed_raw);

    frame.data[1] = 0x00;
    frame.data[2] = 0x00;
    frame.data[3] = 0x00;

    // L gg in alla fyra bajts, little endian
    frame.data[4] = static_cast<uint8_t>( u        & 0xFF);
    frame.data[5] = static_cast<uint8_t>((u >>  8) & 0xFF);
    frame.data[6] = static_cast<uint8_t>((u >> 16) & 0xFF);
    frame.data[7] = static_cast<uint8_t>((u >> 24) & 0xFF);

    return frame;
}

SpeedControlResult MotorController::decode_ack(const struct can_frame& frame) {
    SpeedControlResult result;
    // Protocol: see user prompt for details
    // Data[0] = 0xA2 (command byte)
    // Data[1] = temperature (int8_t)
    // Data[2,3] = iq (int16_t, little endian)
    // Data[4,5] = speed (int16_t, little endian)
    // Data[6,7] = angle (int16_t, little endian)
    result.angle = static_cast<int16_t>((frame.data[7] << 8) | frame.data[6]);
    result.speed = static_cast<int16_t>((frame.data[5] << 8) | frame.data[4]);
    result.torque_current = static_cast<int16_t>((frame.data[3] << 8) | frame.data[2]) * 0.01f;
    result.temperature = static_cast<int8_t>(frame.data[1]);
    return result;
}
void MotorController::send_stop_command() {
    std::cout << "Sending STOP command" << std::endl;
    struct can_frame frame;
    std::memset(&frame, 0, sizeof(frame));
    frame.can_id = 0x141;
    frame.can_dlc = 8;
    frame.data[0] = 0x81; // Stop command
    // All other bytes are already zero from memset

    if (can_socket_ < 0) {
        std::cerr << "CAN socket not initialized" << std::endl;
        return;
    }

    int nbytes = write(can_socket_, &frame, sizeof(frame));
    if (nbytes != sizeof(frame)) {
        std::cerr << "Error sending STOP frame" << std::endl;
    } else {
        std::cout << "STOP frame sent" << std::endl;
    }
    struct can_frame ack_frame = recieve_ack();
    (void)ack_frame; // Ignore the ack frame for now
}
```

Motor Controller Hpp

Listing 5: Motor Controller Hpp

```
    #pragma once
#include <linux/can.h>
#include <linux/can/raw.h>


struct SpeedControlResult {
    float angle;
    float speed;
    float torque_current;
    float temperature;
};

class MotorController{
public:
    MotorController();
    ~MotorController();

    struct can_frame recieve_ack();
    int can_socket_; // CAN socket file descriptor

    void send_stop_command();
    void send_speed_command(int speed);

    struct can_frame encode_speed_command(int speed);
    SpeedControlResult decode_ack(const struct can_frame& frame);

private:
};
```

Madgwick

Listing 6: MagdWick

```
//===========================
// MadgwickAHRS.c
//===========================
//
// Implementation of Madgwick's IMU and AHRS algorithms.
// See: http://www.x-io.co.uk/open-source-imu-and-ahrs-algorithms/
//
// From the x-io website "Open-source resources available on this
    website are
// provided under the GNU General Public Licence unless an alternative
     licence
// is provided in source."
//
// Date      Author          Notes
// 29/09/2011 SOH Madgwick    Initial release
// 02/10/2011 SOH Madgwick  Optimised for reduced CPU load
// 19/02/2012 SOH Madgwick  Magnetometer measurement is normalised
//
//==============================

//---------------------------
// Header files

#include "MadgwickAHRS.h"
#include <math.h>

//---------------------------
// Definitions

#define sampleFreqDef   512.0f          // sample frequency in Hz
#define betaDef         0.1f            // 2 * proportional gain


//===========================
// Functions

//---------------------
// AHRS algorithm update

Madgwick::Madgwick() {
  beta = betaDef;
  q0 = 1.0f;
  q1 = 0.0f;
  q2 = 0.0f;
  q3 = 0.0f;
  invSampleFreq = 1.0f / sampleFreqDef;
  anglesComputed = 0;
}

```

```
void Madgwick::update(float gx, float gy, float gz, float ax, float ay
    , float az, float mx, float my, float mz) {
  float recipNorm;
  float s0, s1, s2, s3;
  float qDot1, qDot2, qDot3, qDot4;
  float hx, hy;
  float _2q0mx, _2q0my, _2q0mz, _2q1mx, _2bx, _2bz, _4bx, _4bz, _2q0,
      _2q1, _2q2, _2q3, _2q0q2, _2q2q3, q0q0, q0q1, q0q2, q0q3, q1q1,
      q1q2, q1q3, q2q2, q2q3, q3q3;

  // Use IMU algorithm if magnetometer measurement invalid (avoids NaN
       in magnetometer normalisation)
  if((mx == 0.0f) && (my == 0.0f) && (mz == 0.0f)) {
    updateIMU(gx, gy, gz, ax, ay, az);
    return;
  }

  // Convert gyroscope degrees/sec to radians/sec
  gx *= 0.0174533f;
  gy *= 0.0174533f;
  gz *= 0.0174533f;

  // Rate of change of quaternion from gyroscope
  qDot1 = 0.5f * (-q1 * gx - q2 * gy - q3 * gz);
  qDot2 = 0.5f * (q0 * gx + q2 * gz - q3 * gy);
  qDot3 = 0.5f * (q0 * gy - q1 * gz + q3 * gx);
  qDot4 = 0.5f * (q0 * gz + q1 * gy - q2 * gx);

  // Compute feedback only if accelerometer measurement valid (avoids
      NaN in accelerometer normalisation)
  if(!((ax == 0.0f) && (ay == 0.0f) && (az == 0.0f))) {

    // Normalise accelerometer measurement
    recipNorm = invSqrt(ax * ax + ay * ay + az * az);
    ax *= recipNorm;
    ay *= recipNorm;
    az *= recipNorm;

    // Normalise magnetometer measurement
    recipNorm = invSqrt(mx * mx + my * my + mz * mz);
    mx *= recipNorm;
    my *= recipNorm;
    mz *= recipNorm;

    // Auxiliary variables to avoid repeated arithmetic
    _2q0mx = 2.0f * q0 * mx;
    _2q0my = 2.0f * q0 * my;
    _2q0mz = 2.0f * q0 * mz;
    _2q1mx = 2.0f * q1 * mx;
    _2q0 = 2.0f * q0;
    _2q1 = 2.0f * q1;
    _2q2 = 2.0f * q2;
    _2q3 = 2.0f * q3;
    _2q0q2 = 2.0f * q0 * q2;
    _2q2q3 = 2.0f * q2 * q3;
    q0q0 = q0 * q0;
    q0q1 = q0 * q1;
    q0q2 = q0 * q2;
    q0q3 = q0 * q3;
    q1q1 = q1 * q1;
    q1q2 = q1 * q2;
    q1q3 = q1 * q3;
    q2q2 = q2 * q2;
    q2q3 = q2 * q3;
    q3q3 = q3 * q3;

    // Reference direction of Earth's magnetic field
    hx = mx * q0q0 - _2q0my * q3 + _2q0mz * q2 + mx * q1q1 + _2q1 * my
         * q2 + _2q1 * mz * q3 - mx * q2q2 - mx * q3q3;
    hy = _2q0mx * q3 + my * q0q0 - _2q0mz * q1 + _2q1mx * q2 - my *
        q1q1 + my * q2q2 + _2q2 * mz * q3 - my * q3q3;
    _2bx = sqrtf(hx * hx + hy * hy);
    _2bz = -_2q0mx * q2 + _2q0my * q1 + mz * q0q0 + _2q1mx * q3 - mz *
         q1q1 + _2q2 * my * q3 - mz * q2q2 + mz * q3q3;
    _4bx = 2.0f * _2bx;
    _4bz = 2.0f * _2bz;

    // Gradient decent algorithm corrective step
    s0 = -_2q2 * (2.0f * q1q3 - _2q0q2 - ax) + _2q1 * (2.0f * q0q1 +
        _2q2q3 - ay) - _2bz * q2 * (_2bx * (0.5f - q2q2 - q3q3) + _2bz
         * (q1q3 - q0q2) - mx) + (-_2bx * q3 + _2bz * q1) * (_2bx * (
        q1q2 - q0q3) + _2bz * (q0q1 + q2q3) - my) + _2bx * q2 * (_2bx
        * (q0q2 + q1q3) + _2bz * (0.5f - q1q1 - q2q2) - mz);
    s1 = _2q3 * (2.0f * q1q3 - _2q0q2 - ax) + _2q0 * (2.0f * q0q1 +
        _2q2q3 - ay) - 4.0f * q1 * (1 - 2.0f * q1q1 - 2.0f * q2q2 - az
        ) + _2bz * q3 * (_2bx * (0.5f - q2q2 - q3q3) + _2bz * (q1q3 -
        q0q2) - mx) + (_2bx * q2 + _2bz * q0) * (_2bx * (q1q2 - q0q3)
        + _2bz * (q0q1 + q2q3) - my) + (_2bx * q3 - _4bz * q1) * (_2bx
         * (q0q2 + q1q3) + _2bz * (0.5f - q1q1 - q2q2) - mz);
    s2 = -_2q0 * (2.0f * q1q3 - _2q0q2 - ax) + _2q3 * (2.0f * q0q1 +
        _2q2q3 - ay) - 4.0f * q2 * (1 - 2.0f * q1q1 - 2.0f * q2q2 - az
        ) + (-_4bx * q2 - _2bz * q0) * (_2bx * (0.5f - q2q2 - q3q3) +
        _2bz * (q1q3 - q0q2) - mx) + (_2bx * q1 + _2bz * q3) * (_2bx *
         (q1q2 - q0q3) + _2bz * (q0q1 + q2q3) - my) + (_2bx * q0 -
        _4bz * q2) * (_2bx * (q0q2 + q1q3) + _2bz * (0.5f - q1q1 -
        q2q2) - mz);
    s3 = _2q1 * (2.0f * q1q3 - _2q0q2 - ax) + _2q2 * (2.0f * q0q1 +
        _2q2q3 - ay) + (-_4bx * q3 + _2bz * q1) * (_2bx * (0.5f - q2q2
         - q3q3) + _2bz * (q1q3 - q0q2) - mx) + (-_2bx * q0 + _2bz *
        q2) * (_2bx * (q1q2 - q0q3) + _2bz * (q0q1 + q2q3) - my) +
        _2bx * q1 * (_2bx * (q0q2 + q1q3) + _2bz * (0.5f - q1q1 - q2q2
        ) - mz);
    recipNorm = invSqrt(s0 * s0 + s1 * s1 + s2 * s2 + s3 * s3); //
        normalise step magnitude
    s0 *= recipNorm;
    s1 *= recipNorm;
    s2 *= recipNorm;
    s3 *= recipNorm;

    // Apply feedback step
    qDot1 -= beta * s0;
    qDot2 -= beta * s1;
    qDot3 -= beta * s2;
    qDot4 -= beta * s3;
  }

  // Integrate rate of change of quaternion to yield quaternion
  q0 += qDot1 * invSampleFreq;
  q1 += qDot2 * invSampleFreq;
  q2 += qDot3 * invSampleFreq;
  q3 += qDot4 * invSampleFreq;

  // Normalise quaternion
  recipNorm = invSqrt(q0 * q0 + q1 * q1 + q2 * q2 + q3 * q3);
  q0 *= recipNorm;
  q1 *= recipNorm;
  q2 *= recipNorm;
  q3 *= recipNorm;
  anglesComputed = 0;
}

//-----------------------
// IMU algorithm update

void Madgwick::updateIMU(float gx, float gy, float gz, float ax, float
    ay, float az) {
  float recipNorm;
  float s0, s1, s2, s3;
  float qDot1, qDot2, qDot3, qDot4;
  float _2q0, _2q1, _2q2, _2q3, _4q0, _4q1, _4q2 ,_8q1, _8q2, q0q0,
      q1q1, q2q2, q3q3;

  // Convert gyroscope degrees/sec to radians/sec
  gx *= 0.0174533f;
  gy *= 0.0174533f;
  gz *= 0.0174533f;

  // Rate of change of quaternion from gyroscope
  qDot1 = 0.5f * (-q1 * gx - q2 * gy - q3 * gz);
  qDot2 = 0.5f * (q0 * gx + q2 * gz - q3 * gy);
  qDot3 = 0.5f * (q0 * gy - q1 * gz + q3 * gx);
  qDot4 = 0.5f * (q0 * gz + q1 * gy - q2 * gx);

  // Compute feedback only if accelerometer measurement valid (avoids
      NaN in accelerometer normalisation)
  if(!((ax == 0.0f) && (ay == 0.0f) && (az == 0.0f))) {

    // Normalise accelerometer measurement
    recipNorm = invSqrt(ax * ax + ay * ay + az * az);
    ax *= recipNorm;
    ay *= recipNorm;
    az *= recipNorm;

    // Auxiliary variables to avoid repeated arithmetic
    _2q0 = 2.0f * q0;
    _2q1 = 2.0f * q1;
    _2q2 = 2.0f * q2;
    _2q3 = 2.0f * q3;
    _4q0 = 4.0f * q0;
    _4q1 = 4.0f * q1;
    _4q2 = 4.0f * q2;
    _8q1 = 8.0f * q1;
    _8q2 = 8.0f * q2;
    q0q0 = q0 * q0;
    q1q1 = q1 * q1;
    q2q2 = q2 * q2;
    q3q3 = q3 * q3;

    // Gradient decent algorithm corrective step
    s0 = _4q0 * q2q2 + _2q2 * ax + _4q0 * q1q1 - _2q1 * ay;
    s1 = _4q1 * q3q3 - _2q3 * ax + 4.0f * q0q0 * q1 - _2q0 * ay - _4q1
         + _8q1 * q1q1 + _8q1 * q2q2 + _4q1 * az;
    s2 = 4.0f * q0q0 * q2 + _2q0 * ax + _4q2 * q3q3 - _2q3 * ay - _4q2
         + _8q2 * q1q1 + _8q2 * q2q2 + _4q2 * az;
    s3 = 4.0f * q1q1 * q3 - _2q1 * ax + 4.0f * q2q2 * q3 - _2q2 * ay;
    recipNorm = invSqrt(s0 * s0 + s1 * s1 + s2 * s2 + s3 * s3); //
        normalise step magnitude
    s0 *= recipNorm;
    s1 *= recipNorm;
    s2 *= recipNorm;
    s3 *= recipNorm;

    // Apply feedback step
    qDot1 -= beta * s0;
    qDot2 -= beta * s1;
    qDot3 -= beta * s2;
    qDot4 -= beta * s3;
  }

  // Integrate rate of change of quaternion to yield quaternion
  q0 += qDot1 * invSampleFreq;
  q1 += qDot2 * invSampleFreq;
  q2 += qDot3 * invSampleFreq;
  q3 += qDot4 * invSampleFreq;

  // Normalise quaternion
  recipNorm = invSqrt(q0 * q0 + q1 * q1 + q2 * q2 + q3 * q3);
  q0 *= recipNorm;
  q1 *= recipNorm;
  q2 *= recipNorm;
  q3 *= recipNorm;
  anglesComputed = 0;
}

//-----------------------
// Fast inverse square-root
// See: http://en.wikipedia.org/wiki/Fast_inverse_square_root

float Madgwick::invSqrt(float x) {
  float halfx = 0.5f * x;
  float y = x;
  long i = *(long*)&y;
  i = 0x5f3759df - (i>>1);
  y = *(float*)&i;
  y = y * (1.5f - (halfx * y * y));
  y = y * (1.5f - (halfx * y * y));
  return y;
}

//--------------------------

void Madgwick::computeAngles()
{
  roll = atan2f(q0*q1 + q2*q3, 0.5f - q1*q1 - q2*q2);
  pitch = asinf(-2.0f * (q1*q3 - q0*q2));
  yaw = atan2f(q1*q2 + q0*q3, 0.5f - q2*q2 - q3*q3);
  anglesComputed = 1;
}
```

MagdWick h

Listing 7: MagdWick h

```
//==============================
// MadgwickAHRS.h
//=============================
//
// Implementation of Madgwick's IMU and AHRS algorithms.
// See: http://www.x-io.co.uk/open-source-imu-and-ahrs-algorithms/
//
// From the x-io website "Open-source resources available on this
    website are
// provided under the GNU General Public Licence unless an
    alternative licence
// is provided in source."
//
// Date     Author          Notes
// 29/09/2011 SOH Madgwick    Initial release
// 02/10/2011 SOH Madgwick  Optimised for reduced CPU load
//
//===========================
#ifndef MadgwickAHRS_h
#define MadgwickAHRS_h
#include <math.h>

//-------------------------
// Variable declaration
class Madgwick{
private:
    static float invSqrt(float x);
    float beta;       // algorithm gain
  // quaternion of sensor frame relative to auxiliary frame
    float invSampleFreq;
    float roll;
    float pitch;
    float yaw;
    char anglesComputed;
    void computeAngles();

//--------------------------
// Function declarations
public:
    Madgwick(void);
    void begin(float sampleFrequency) { invSampleFreq = 1.0f /
        sampleFrequency; }
    void update(float gx, float gy, float gz, float ax, float ay,
        float az, float mx, float my, float mz);
    void updateIMU(float gx, float gy, float gz, float ax, float
        ay, float az);
    //float getPitch(){return atan2f(2.0f * q2 * q3 - 2.0f * q0 *
        q1, 2.0f * q0 * q0 + 2.0f * q3 * q3 - 1.0f);};
    //float getRoll(){return -1.0f * asinf(2.0f * q1 * q3 + 2.0f *
         q0 * q2);};
    //float getYaw(){return atan2f(2.0f * q1 * q2 - 2.0f * q0 * q3
         , 2.0f * q0 * q0 + 2.0f * q1 * q1 - 1.0f);};
    float getRoll() {
        if (!anglesComputed) computeAngles();
        return roll * 57.29578f;
    }
    float getPitch() {
        if (!anglesComputed) computeAngles();
        return pitch * 57.29578f;
    }
    float getYaw() {
        if (!anglesComputed) computeAngles();
        return yaw * 57.29578f + 180.0f;
    }
    float getRollRadians() {
        if (!anglesComputed) computeAngles();
        return roll;
    }
    float getPitchRadians() {
        if (!anglesComputed) computeAngles();
        return pitch;
    }
    float getYawRadians() {
        if (!anglesComputed) computeAngles();
        return yaw;
    }
    float q0;
    float q1;
    float q2;
    float q3;
};
#endif
```

Live Plotting

Listing 8: Live$_{P}$*lot.py*

```
    #!/usr/bin/env python3
import can
import threading
import time
import math
import matplotlib.pyplot as plt
import matplotlib.animation as animation


    # Beh ver  ndras utifr n vilken HZ man  k r med. Kan bli
        v ldigt otydligt om window   r litet men samplingen (HZ)   r
        mycket stor

#Listorna som plottas
stop_thread = False
velocity_list = []
current_list = []
angle_list = []
data_lock = threading.Lock()

    # filtrera bort allt utom 241
can_filters = [{"can_id": 0x241, "can_mask": 0x7FF, "extended": False
    }]
bus = can.Bus(interface="socketcan", channel="can0", bitrate=100000,
    can_filters=can_filters)

# Funktion som lyssnar
def can_listener():
    global stop_thread
    while not stop_thread:
        try:
            msg = bus.recv(timeout=1)
            if msg is not None:
                interpret(msg)
        except can.CanError as e:
            print("CAN error:", e)
            break

# Funktion som tolkar svarsmeddelanden av typ A2
def interpret(msg):
    if msg.data[0] == 0xA2:
        data = msg.data
        torque_current = (data[3] << 8) | data[2]
        if torque_current >= 0x8000:
            torque_current -= 0x10000
        actual_current = torque_current * 0.01

            # Inverterade spikar kommer ibland
        velocity_raw = (data[5] << 8) | data[4]
        if velocity_raw >= 0x8000:
            velocity_raw -= 0x10000
        velocity = velocity_raw

            #  g  r  en wraparound vid 360 grader
        encoder_raw = (data[7] << 8) | data[6]
        angle_deg = encoder_raw % 360

        with data_lock:
            current_list.append(actual_current)
            angle_list.append(angle_deg)
            velocity_list.append(velocity)

            # H ller listorna rimligt  s m
            # if len(current_list) > 20000:
            #     current_list.pop(0)
            #     angle_list.pop(0)
            #     velocity_list.pop(0)

# Plottning
fig, (ax1, ax2, ax3) = plt.subplots(3, 1, figsize=(10, 8))
line1, = ax1.plot([], [], lw=2, label="Torque Current (A)")
line2, = ax2.plot([], [], lw=2, label="Angle (sine)")
line3, = ax3.plot([], [], lw=2, label="Speed")

ax1.set_title("Torque Current")
ax1.set_ylabel("Current (A)")
ax1.set_xlabel("Sample")
ax1.set_ylim(-1, 1)
ax1.legend()

    # Angle   r korrekt wrappar runt efter ett varv
ax2.set_title("Motor Angle")
ax2.set_ylabel("Degrees)")
ax2.set_xlabel("Sample")
ax2.set_ylim(0, 360)
ax2.legend()

    # Velocityn som visas nu   r liksom spikarna  omv nt
ax3.set_title("Degrees per Second")
ax3.set_ylabel("DPS")
ax3.set_xlabel("Sample")
ax3.set_ylim(-150, 150)
ax3.legend()

def init():
    line1.set_data([], [])
    line2.set_data([], [])
    line3.set_data([], [])
    return line1, line2, line3

def animate(frame):
    with data_lock:
        x = list(range(len(current_list)))
        y1 = current_list.copy()
        y2 = angle_list.copy()
        y3 = velocity_list.copy()

    line1.set_data(x, y1)
    line2.set_data(x, y2)
    line3.set_data(x, y3)

    window = 1000
    if len(x) > window:
        ax1.set_xlim(x[-window], x[-1])
        ax2.set_xlim(x[-window], x[-1])
        ax3.set_xlim(x[-window], x[-1])
    else:
        ax1.set_xlim(0, window)
        ax2.set_xlim(0, window)
        ax3.set_xlim(0, window)

    return line1, line2, line3

ani = animation.FuncAnimation(fig, animate, init_func=init, interval
    =100, blit=False)

# F r b ttre layout
plt.tight_layout()

# Starta CAN-lyssnaren i en separat tr d
listener_thread = threading.Thread(target=can_listener, daemon=True)
listener_thread.start()

# Visa plottf nstret
plt.show()

# N r plottf nstret st ngs:
stop_thread = True
bus.shutdown()
```