Lucerne University of Applied Sciences and Arts
MSE Industrial Technologies

Master Thesis

Academic Year 2021-22

Validation of the real-time capability of ROS 2 in mobile robotics

Industrial Partner: Lucerne University of Applied Sciences and Arts
Competence Center for Mechanical Systems
Advisor: Prof. Ralf Legrand
Co-Advisor: Prof. Dr. Björn Jensen
Expert: Dipl. Ing. ETH Ruedi Haller
Author: Marco Grossmann

June 17, 2022

Master-Thesis an der Hochschule Luzern - Technik & Architektur

Titel	**Validierung der Echtzeitfähigkeit von ROS 2 in der mobilen Robotik**
Diplomandin/Diplomand	**Grossmann Marco**
Master-Studiengang	**Master in Engineering**
Semester	**FS22**
Dozentin/Dozent	**Prof. Legrand Ralf, Prof. Dr. Jensen Björn**
Expertin/Experte	**Dipl. Ing. ETH Haller Ruedi**

Abstract Deutsch
In den letzten Jahren ist die Bedeutung von mobilen Robotern schnell gewachsen. Mit dieser neuen Art von Roboter entstehen neue Anforderungen bei der Entwicklung. Ein Tool, das sich in diesem Bereich für die Softwareentwicklung durchgesetzt hat, ist das Robot Operating System (ROS). Anfänglich von und für Hochschulen entwickelt, kommt ROS nach und nach auch mehr in industriellen Anwendungen zum Einsatz. Diese bringen jedoch auch neue Herausforderungen mit sich. Um die damit einhergehenden Ansprüche zu erfüllen ist die erste Version von ROS, ROS 1, komplett überarbeitet worden. Daraus entstanden ist ROS 2. Ziel dieser neuen Version ist es, zuverlässige und deterministische Applikationen entwickeln zu können, die beispielsweise auch in sicherheitskritischen Anwendungen zum Einsatz kommen können.
Ziel dieser Master-Thesis ist es, sowohl die Leistungsfähigkeit als auch die Limitationen von ROS 2 für Echtzeitanwendungen mittels Versuche an zwei verschiedenen Versuchsaufbauten aufzuzeigen und einzuordnen. Hierzu wird auch eine Speicher Programmierbare Steuerung (SPS), ein industriell weit verbreitetes System, beigezogen.
Basierend auf den Resultaten der Experimente kann aufgezeigt werden, dass das Verhalten bezüglich zeitlicher Vorgaben stark durch den/die Entwickler:in beeinflusst wird. Dies beginnt mit dem Aufsetzen des Betriebssystems und geht weiter über diverse Einstellmöglichkeiten im Quellcode der zu entwickelnden Anwendung. Ebenfalls muss der Einbindung von externer Hardware (z.B. Sensoren) grosse Beachtung beigemessen werden.
Zum jetzigen Entwicklungsstand von ROS 2 eignet sich das Tool für Anwendungen mit weichen Echtzeitanforderungen. Dies sind Anwendungen, bei der eine gestellte zeitliche Anforderung an das System bis zu einem gewissen Grad resp. zu einer gewissen Anzahl überschritten werden darf. Für Anwendungen mit harten Echtzeitanforderungen, bei denen die zeitlich gestellten Anforderungen jeder Zeit eingehalten werden müssen, sind andere Systeme wie beispielsweise eine SPS zu bevorzugen. Dies kann sich jedoch mit der laufenden Weiterentwicklung von ROS 2 in der Zukunft ändern.

Abstract Englisch
In recent years, the importance of mobile robots has grown rapidly. With this new type of robot, new development requirements arise. One tool that has gained acceptance in software development is the Robot Operating System (ROS). Initially developed by and for universities, ROS is gradually being used increasingly in industrial applications. However, this also brings new challenges. The first version of ROS, ROS 1, was completely revised in order to meet the demands. The goal of this new

version is to be able to develop reliable and deterministic applications that can also be used in safety-critical applications, for example.
This master thesis aims to demonstrate and classify the performance and the limitations of ROS 2 for real-time applications through tests on two different test setups. For this purpose, a programmable logic controller (PLC), a widely used industrial system, is used for comparison.
Based on the results of the experiments, it can be shown that the developer strongly influences the behaviour concerning time specifications. This starts with setting up the operating system and continues with various setting options in the application's source code to be developed. Likewise, integrating external hardware (e.g. sensors) must be given great attention.
At the current stage of the development of ROS 2, the tool is suitable for applications with soft real-time constraints. These are applications where a set time constraint on the system may be exceeded up to a certain degree or a certain number of times. For applications with hard real-time constraints, where the time requirements must always be met, other systems such as PLCs are preferable. However, this may change in the future with the ongoing development of ROS 2.

Ort, Datum Horw, 17.06.2022

Acknowledgement

At this point, I would like to express my gratitude to the people involved in this project.
I would like to thank my advisor, Prof. Ralf Legrand, not only for his support and advice in my Master's thesis but throughout my whole Master's studies.
I would also like to thank Prof. Dr. Björn Jensen for his advice during the project and for giving me the opportunity to write my Master's thesis in the area of mobile robotics.
Last but not least, I would like to thank Audrey Queudet, Associate Professor at the University of Nantes, for providing me with the paper «Bringing Real-Time Capabilities to ROS», which helped me to get started with the topic.

Contents

List of Figures

List of Tables

Nomenclature

Symbols

Symbol	Definition	Unit
θ	Deflection angle of the pendulum bar	[rad]
θ_{init}	Initial deflection angle of the pendulum bar for the simulation	[rad]
ϑ	Deflection angle of the ballbot body	[rad]
μ_{j}	Viscous friction coefficient of the pendulum bar joint	[Nm/(rad/s)]
μ_{m}	Viscous friction coefficient of motor	[Nm/(rad/s)]
φ	Angle of the ball	[rad]
ω_{inertial}	Angular velocity of the reaction wheel	[rad/s]
ω_{w}	Relative angular velocity of the reaction wheel wrt the pendulum bar	[rad/s]
$\omega_{\text{w, init}}$	Initial relative angular velocity of the reaction wheel wrt the pendulum bar for the simulation	[rad/s]
$\mathbf{A}_c$	System matrix of the state-space model	[-]
$\mathbf{B}_c$	Input matrix of the state-space model	[-]
$\mathbf{C}_c$	Output matrix of the state-space model	[-]
$\mathbf{D}_c$	Feedthrough matrix of the state-space model	[-]
d	Point in time when a task has to be accomplished	[-]
Δe	Time span needed to perform a task	[s]
g	Gravitational constant	[m/s^2]
$I_{\text{m, j}}$	Moment of inertia of the motor wrt the pendulum bar joint	[kgm^2]
$I_{\text{p, j}}$	Moment of inertia of the pendulum bar wrt the pendulum bar joint	[kgm^2]

I_{w}	Moment of inertia of the reaction wheel wrt its centre of mass	[kgm^2]
$I_{\mathrm{w,\,j}}$	Moment of inertia of the reaction wheel wrt the pendulum bar joint	[kgm^2]
I_{tot}	Sum of all moments of inertias with wrt the pendulum bar joint	[kgm^2]
i	Electrical current	[A]
i_{gearbox}	Transformation ratio of the gearbox	[-]
k_{e}	Motor back EMF constant	[Vs/rad]
k_{t}	Motor torque constant	[Nm/A]
L	Motor inductance	[H]
L_{p}	Length of the pendulum bar	[m]
m_{m}	Mass of the motor	[kg]
m_{p}	Mass of the pendulum bar	[kg]
m_{w}	Mass of the reaction wheel	[kg]
R	Motor resistance	[Ω]
r	Point in time when a task can start	[-]
r_{k}	Reference input	[-]
T_{w}	Torque applied on the reaction wheel	[Nm]
u	Electrical voltage	[V]
$\mathbf{u}$	Input vector of the state-space model	[-]
$\mathbf{x}$	State vector of the state-space model	[-]
$\mathbf{y}$	Output vector of the state-space model	[-]

Glossary

Jitter	Jitter is the deviation of the expected and the actual time of execution.
Kernel	Part of an operating system. Interface between software and hardware.
Latency	Time between an action and the corresponding respond.
Vanilla Kernel	Unmodified kernel version.

Acronyms and Abbreviations

DDS	Data Distribution Service
GPOS	General-Purpose Operating System
IMU	Inertial Measurement Unit
IP	Internet Protocol
LQR	Linear Quadratic Regulator
OS	Operating System
PLC	Programmable Logic Controller
PWM	Pulse-width Modulation
QoS	Quality of Service
ROS	Robot Operating System
RTOS	Real-Time Operating System
TCP	Transmission Control Protocol
UDP	User Datagram Protocol

Chapter 1

Introduction

Mobile robotics is a fast-growing and very demanding area of robotics. In contrast to, for example, industrial robots, mobile robots have to perceive their environment and react appropriately to each new situation. To do so, tasks such as mapping, localization, navigation or motion control are needed to name only a few of the tasks performed by mobile robots. Therefore, another, more modular software approach was demanded. One framework which allows for such a modular way of building software is the Robot Operating System (ROS). The paradigm of ROS is to split the software into smaller, self-contained units. By exchanging information between these units, the different units are able to fulfil the overall goal. The advantage of such a modular architecture is that it is easier to maintain an overview of complex software. Furthermore, it facilitates the reuse of existing code for new projects. The same applies to changing robot hardware. When for example, a camera is replaced by another model, only the software unit must be adapted, which interfaces with the hardware. These are the reasons why ROS is one of the most popular frameworks when it comes to programming mobile robots. However, the scope of ROS is not limited to mobile robots. It can be used to program any robot or any other application.

ROS 1 was initially developed at Stanford University and the robotics institute Willow Garage and is an open-source project. Industrial applications were not the primary target when ROS 1 was developed; therefore, real-time capability or reliability requirements were not among the primary considerations in the design process. While the popularity of ROS 1 in the research area of mobile robotics grew very fast, the expansion to other areas, especially into the industry, was limited by these design decisions.
To overcome these drawbacks, a new version of ROS, ROS 2, was developed instead of improving the existing one. The focus while developing ROS 2 was to maintain the advantages of ROS 1 while enhancing these parts, which limit the deployment of ROS 1 into new applications.

The goal of this master thesis is to show both the potential but also the limitations of ROS 2 for applications with real-time constraints in mind. Furthermore, the effort that is needed in order to make ROS 2 real-time capable should be shown. It starts with setting up the operating system and contains system settings as well as settings in the code.
In order to have a benchmark for the performance of ROS 2, a Programmable Logic Controller (PLC) will be used. These are the state-of-the-art controller for real-time applications with the highest demands and are

very often used in industrial applications. Therefore, they are very well suited to assess the current status of ROS 2.
All these analyses will be done through two different experimental setups in control theory. This is a domain where real-time constraints are often required, especially to control unstable systems.

This thesis is structured in the following way: Chapter 2 will introduce the general concepts for real-time systems as well as for ROS. In chapter 3, relevant related work is presented. Chapter 4 will focus on the setup of a real-time capable operating system (RTOS) and show the performance difference from a general-purpose operating system (GPOS). In chapter 5, the first experimental setup is presented, from the concept to the final setup. Then the experiments conducted are introduced, and the results are shown and discussed. Chapter 6 presents the second experimental setup in a similar way as the first setup is presented. Again this includes the experimental procedure, their results, and their classification. In chapter 7, the thesis and results are concluded, and a proposal for further investigations is given.

Chapter 2

Preliminaries

2.1 Real-time Systems

As stated in [1], real-time systems are often designed as follows: a technical process is supervised by a measuring system (e.g. sensor). Based on this measuring information, a computation system evaluates how to influence the technical system by utilising an actuating variable to get the desired behaviour of the technical system. Such a system is shown in figure 2.1 from [1].

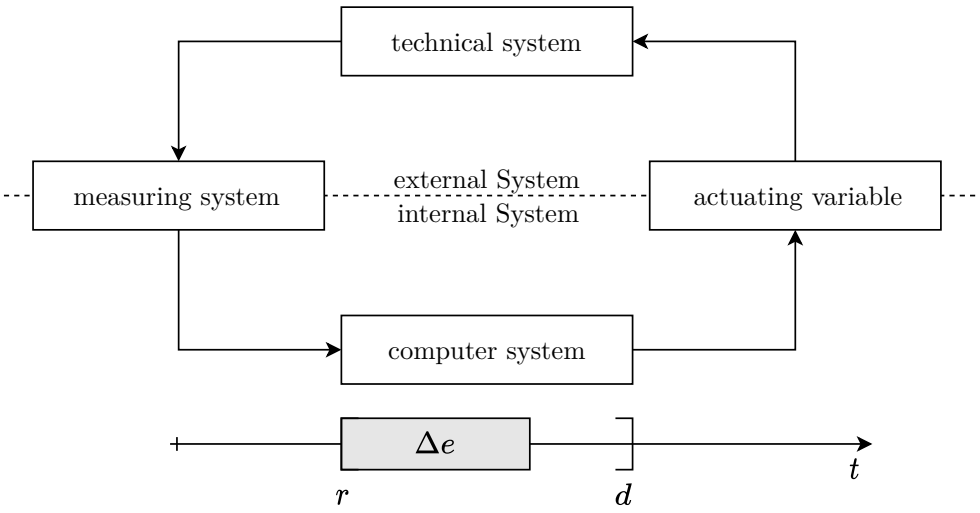


Figure 2.1: Basic model of a real-time system

It is called a real-time system if the incoming data from the measuring system is processed to an output by the computer system faster than a given time. Mathematical this constraint can be formulated as follows:

$$r + \Delta e \leq d \tag{2.1}$$

Where r is the time when the computation task can start, Δe is the time needed for the task, and d is the point in time when the task must be accomplished.
This leads to the following definition of real-time systems by [2]:

> «A real-time system is a system that is at any time ready to process incoming data in a predefined amount of time. Depending on the use-case, this data can accrue randomly distributed, or at predefined points in time.»

This definition shows that real-time is not primarily about short computation time. It is rather about the deterministic behaviour of the system. However, short computation time can help meet strict deadlines [1]. Depending on how strict equation 2.1 must be met, some sources as [1] or [3] distinguish between different real-time criteria:

hard real-time In hard real-time systems, equation 2.1 must hold at any time. Every miss out on this condition is treated as a system failure and is not tolerable. Of course, this can not be achieved under all circumstances, e.g. in case of a network failure. Therefore, other precautions as redundancy need to be taken to avoid this.

soft real-time In soft real-time systems, some miss-outs of the time condition are tolerated. It is enough if, e.g. the condition is met in 90% of the cases or the timeout is only small. This can lead to a quality reduction which is acceptable, however.

firm real-time Is similar to soft real-time. If a deadline is missed, the output becomes void and is therefore ignored.

Real-time systems are essential if unexpected time delays during information processing can lead to damage or unwanted system behaviour.
According to [1], this potential damage can be divided into the following three groups:

- Harm of life: e.g. pedestrian detection in self-driving cars
- Financial loss: e.g. collision of moving parts in a production process
- Quality loss: e.g. in multimedia systems

One common way to characterise the performance of real-time systems is to measure the jitter. Jitter is the deviation between a task's expected and actual execution time, and it indicates how deterministic the system is. Therefore, the maximal jitter is of particular interest for real-time systems and will be one of the key indicators used in this thesis to assess the performance of ROS 2. Figure 2.2 from [4] shows the relation between the desired cycle time and jitter for a task that is executed periodically. Such setups with periodical executions are widespread, for example, in closed-loop control systems, which are classic examples where time constraints often play an essential role. However, how much jitter is acceptable heavily depends on the application and can not be generalised.

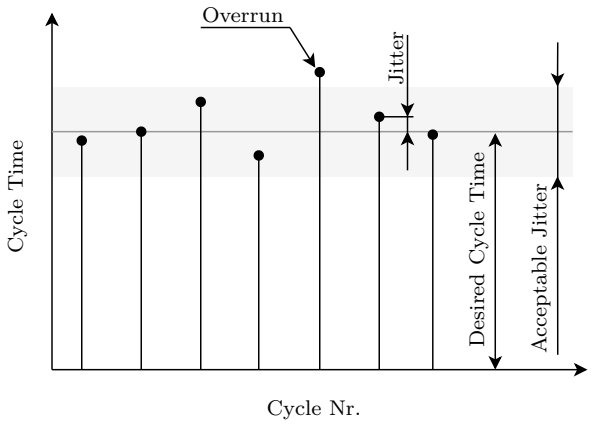


Figure 2.2: Visual representation of jitter in a cyclic task

2.2 ROS 1

The general idea and the advantages of ROS have already been pointed out in chapter 1. The focus of this section is more on the technical aspects of ROS.
This thesis uses the following terminology to distinguish between the two fundamental versions of ROS:

ROS The term *ROS* is used whenever both versions of ROS are meant.

ROS 1 When only the first version of ROS is meant, the term *ROS 1* is used.

ROS 2 When only the second version of ROS is meant, the term *ROS 2* is used.

Even though ROS stands for Robot Operating System, it isn't an operating system in a classical way like Windows or Linux. It can be seen as a framework or environment that can run several processes in parallel and allows for communication between these processes [5]. Therefore, ROS 1 is built on a set of concepts that provide the demanded functionalities [5, 6]. The most important ones are described here briefly:

Nodes A node is a process that performs a specific task. E.g. establish the connection to a camera or read the sensor data. Each node is a self-contained program. Therefore, systems like mobile robots often consist of many different nodes running simultaneously.

Messages If a node wants to send information to another node, this is done via messages. Depending on the type of information, a message can be a standard primitive type, e.g. an integer, floating-point, string, etc. Suppose a single standard primitive type is not enough to describe the information. In that case, a message can also consist of arrays of primitive types, or they can be composed of other messages.

Topic The transport of messages from one node to another is done via topics. A node which wants to send information publishes the messages on a certain topic. Each topic needs to have a name in order to identify it. A node interested in this information can subscribe to this topic and receive the messages published on this topic. It is possible to have multiple publisher and subscriber nodes on the same topic. Vice versa, one node can publish and subscribe to various topics.

Master The master enables nodes to find each other in the network. This is necessary if nodes want to send and receive messages. Therefore the master has a registration service. Each node registers with the master on which topic it wants to publish or subscribe to. The master then informs the respective nodes, and they can establish a connection for communication. The message can then be sent directly from one node to another and does not have to pass the master.

The network used for communication in ROS 1 is built on a peer-to-peer architecture. In contrast to a client/server architecture, in a peer-to-peer network, all stations have equal rights, and there is no central hub like a server [7]. Communication is directly between peers. To send messages over the network, ROS 1 uses TCPROS and UDPROS, based on the TCP/IP and the UDP/IP standard, respectively. Which of both to use is negotiated between the nodes, depending on what is preferred and what is supported by the nodes [8].
In this thesis, there will be no further investigation on the behaviour of ROS 1 since it is coming to its end of life. All future releases will be based on ROS 2.

2.3 ROS 2

The goal of ROS 1 was to develop a framework that can be reused for a variety of use cases instead of creating a new framework for each special case [5]. However, as with all other frameworks, also ROS 1 has its weaknesses. ROS 2 is to overcome these drawbacks and enable the use of ROS 2 in areas where this was not possible with previous versions. Two major weaknesses of ROS 1 are the limited variety of operating systems supported by ROS 1 and the lack of real-time capability [9]. In this thesis, mainly the real-time capability will be in focus.
The main concepts of ROS 2 were taken over from ROS 1. A ROS 2 system is still based on *nodes*, *messages*, and *topics*, while the *discovery* in ROS 2 has replaced the *master*. In ROS 2, nodes do not have to register with a master. All nodes which are in the same ROS-domain are discovered automatically. The replacement of the master is an improvement regarding reliability. The master approach used in ROS 1 is a single point of failure, which can cause the whole system to shut down. This improvement is due to the new communication middleware used in ROS 2. Instead of having its own transport protocol, ROS 2 uses an existing middleware, the Data Distribution Service (DDS). The advantage of using an existing middleware is that it is already developed and tested in many use cases. However, this results in a slightly other architecture compared to ROS 1. This difference is shown in figure 2.3, taken from [9].

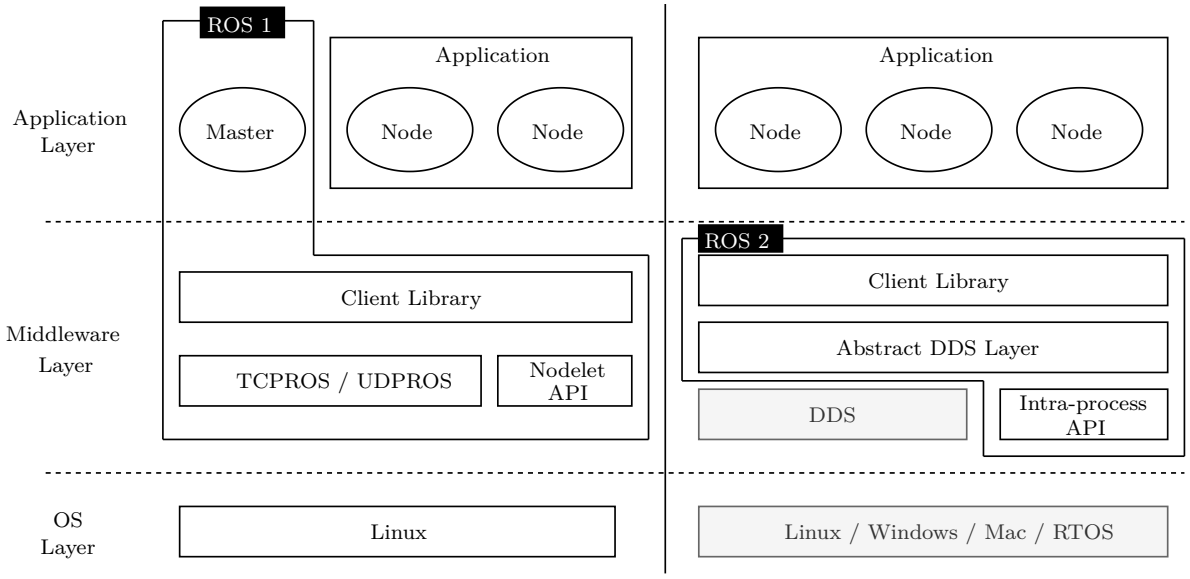


Figure 2.3: Differences in the architecture of a ROS 1 and a ROS 2 system

DDS is a standard that is used to build an efficient publish/subscribe model that is able to fulfil real-time requirements [10]. To do so, the user can define Quality of Service (QoS) parameters within the DDS standard. These QoS parameters help to use the limited resources optimally by having a stricter communication policy only where needed and hence not generating unnecessary overheads. An overview of the most important QoS parameters regarding real-time constraints can be found in section 5.6.1.
Because DDS is only a standard definition, many different implementations are available. ROS 2 allows the use of several different implementations from different vendors. To enable the interaction between the ROS 2 Client Library and the different DDS implementations, there is an «Abstract DDS Layer» in between. The ROS 2 data objects are converted to the respective DDS data objects and vice versa in this layer. Another advantage of this abstraction layer is that ROS 2 developers can focus on the application and don't have to handle DDS-specific code [11]. This is why DDS characteristics are not further discussed here.

Chapter 3

Related Work

The first version of ROS 2 (alpha version) was released in 2015. By now, several publications are available on the performance, latency and real-time behaviour of ROS 2. In this section, the most relevant work for this thesis is presented.

One of the first research was done by [9] in 2016. Their work is based on an alpha version of ROS 2, hence this version did not have all the features and was not as well developed as the current version of ROS 2. Therefore, some of their findings might not be relevant for newer ROS 2 versions. Nevertheless, this publication gives a good overview of what to expect from ROS 2 in different situations and compares the performance to ROS 1. The situations include different network layouts (local and remote), message sizes, number of subscriber nodes, QoS settings and DDS implementations. To characterise the performance in these different situations, they use latency measurements, throughput and the number of threads on each node.
The authors could show that another DDS implementation is preferable depending on the network layout (local or remote). In the local case, the latency is caused by overhead. In the remote case, however, the transmission time between the computers is the dominating part, and overhead latencies are negligible. Instead, the throughput of the DDS implementation should be considered. Furthermore, the publication states that ROS 2 shows superior behaviour over ROS 1 regarding multiple subscribers on a topic. In ROS 1, there is a big gap between the point in time when the first and the last subscriber gets a message. In ROS 2, all subscribers get the messages simultaneously except for some little jitter. Furthermore, the authors could show the benefits of using DDS as communication middleware over the communication protocols used in ROS 1, although additional latency is added to the communication by two data object conversions.

Another investigation on ROS 2 has been provided by [12]. In their work, they focus more on the scalability of a ROS 2 system. They examine the influence of the publisher frequency and the influence of the number of nodes in a data-processing pipeline on the end-to-end latency.
For the first investigation on the publisher frequency, they used a setup with three nodes and publishing frequencies between 1 Hz and 100 Hz. The evaluation shows that the latency decreases for higher frequencies. The authors state that this could be due to energy-saving features in particular hardware or to different priorities in thread scheduling.

Regarding the behaviour depending on the number of nodes, a linear relationship between the number of nodes and the latency can be observed as long as the system is not overloaded.
Similar to the work presented above, [12] also splits the latency into its divisions. Hence, one can see that most of the latency comes from the DDS middleware. On the ROS 2 side, the main part of the latency arises between the notification of DDS to ROS 2 that there is a new message and the actual message retrieval by ROS 2.
According to their research, the authors provide a few aspects to consider:

- If the payload is higher than 64 KB, latency increases with the payload size due to fragmentation size of UDP
- The higher the frequency, the lower the latency
- DDS middleware and the delay between message notification and message retrieval by ROS 2 contribute the most significant portions of the overall latency.

Chapter 4

System Setup

This chapter discusses the required characteristics of operating systems, which are needed to run real-time applications on it. The system setup is tested utilising a simulation provided by ROS 2, and the results are discussed.
For all the work that is conducted in this thesis, Foxy Fitzroy is the ROS 2 distribution used. This is the current ROS 2 distribution with long-term support by the time of writing. Support will last until May 2023. ROS 2 Foxy Fitzroy runs on Linux Ubuntu 20.04.

4.1 Real-time Operating Systems

In order to run a real-time system with ROS 2, it is not sufficient to only install ROS 2 on a standard Linux kernel. The Linux kernel itself is not preemptible. Therefore, low priority tasks cannot be interrupted to run high priority tasks first before completing the low priority task. As a consequence, the behaviour of the system is not deterministic because important tasks might have to wait too long before they are executed. There exist various solutions to prevent this. One way is to use a real-time operating system (RTOS) that has all the necessary properties by default, e.g. freeRTOS, VxWorks or QNX. However, ROS 2 is mainly targeted at Linux Ubuntu. This is where ROS 2 is tested and where the most resources are available. There are two popular kernel extensions to use Linux Ubuntu as a real-time capable OS. One is the so-called RT_PREEMPT patch which extends the Linux kernel with interruption possibility. Another one is the Xenomai framework.
Xenomai implements a dual kernel approach: The cobalt kernel, which comes with Xenomai, schedules the real-time tasks while the Linux kernel schedules the non-real-time tasks. The main advantage of Xenomai over RT_PREEMPT is that it is hard real-time save according to [4]. However, developing applications that run under the cobalt core must be considered when programming the tasks. Special tools and libraries must be learned and used. Hence, an already built application can not benefit from the real-time capabilities of the cobalt kernel. Furthermore, according to [13], there is no official ROS 2 support in Xenomai yet.
In contrast to this dual kernel approach, RT_PREEMPT is a single kernel approach. Consequently, each application can benefit from the real-time capabilities of the kernel extension. According to [14], RT_PREEMPT is nowadays the de-facto standard for real-time Linux. Therefore, this will be the kernel extension in use for this thesis.

4.2 Test Setup

In order to show the effect of using Linux with the RT_PREEMPT patch, both the vanilla Linux kernel and the patched kernel are tested with the real-time demo included in the ROS 2 installation [4, 15]. This demo is a simulation of an inverted pendulum and consists of two nodes. While one node simulates the behaviour of the pendulum, the other one acts as a controller. A third node which is not part of the real-time environment logs the real-time performance (i.e. jitter). This demo is used because it is part of the ROS 2 installation, and there are already results available for comparison.
All the experiments have been executed on the same platform with the following specification:

Table 4.1: Setup for the operating system comparison

	Vanilla Linux kernel	**RT_PREEMPT**
Platform	Dell Latitude 5580 Notebook	
Basic operating system	Ubuntu 20.04 LTS	
Kernel version	5.8.0-43	5.4.78
Kernel patch	-	5.4.78-rt44
CPU model	Intel Core i7-7820HQ	
CPU freq	2.90 GHz	
Nr. Cores	4	
RAM	16 GB	
ROS 2 distro	Foxy Fitzroy	

The update rate of the nodes is set to the default frequency of 1 kHz. To assess the performance of the respective kernel version, the time is measured between the expected update and the actual update. The observation includes one million cycles.
Each kernel Version is tested under two different conditions: In the first step, the experiment is performed without any additional system load. In the second step, the system is loaded with an additional load on the CPUs to simulate processes that run simultaneously. In this case, two CPU cores are utilized with the additional load. A real-time system must be able to schedule processes according to their priority which means it must prefer the processes of the inverted pendulum over the additional load. Therefore, a real-time capable system should not show higher maximal jitter under load.

4.3 Results

The results of the first run (shown in table 4.2) show unexpected behaviour. The first thing to notice is that when the system is loaded with additional stress, the results for the vanilla Linux kernel and for the RT_PREEMPT kernel are significantly better than in the unstressed case. The second noticeable

characteristic is the overall performance of the RT_PREEMPT kernel. Compared to the results of [4], both mean and maximal jitter values are significantly higher. Even though neither hardware nor software used in [4] is known, better results were expected.

Table 4.2: Resulting jitter for the inverted pendulum demo

	Vanilla Linux kernel		RT_PREEMPT	
	unstressed	stressed	unstressed	stressed
Min	2.48 μs	2.30 μs	3.44 μs	2.28 μs
Mean	69.68 μs	30.60 μs	51.11 μs	15.59 μs
Max	937.43 μs	695.97 μs	138.30 μs	87.15 μs

In order to rule out an installation problem as the reason for the bad RT_PREEMPT kernel performance, the installation was tested by the cyclictest. According to [16], cyclictest measures the time between the thread's expected wake-up time and the actual wake-up time. This measurement can be used to benchmark real-time systems. In this setup, the maximal delay of the wake-up time under the RT_PREEMPT kernel is around 30 μs for both cases, unstressed and stressed. Hence, the bad RT_PREEMPT performance is not caused by a faulty installation.
By further examining the system behaviour, it turned out that different CPU modes (so-called c-states) could be one possible cause for the strange behaviour (less jitter under additional stress). C-states are used to save energy when CPU load is low [17]. They can be interpreted as a standby mode. Every time they have to be reactivated from energy-saving to working mode, it takes some time. This explains why a stressed system shows better results in terms of jitter: With the additional CPU load, they can less often change to a power-saving mode and hence have to be reactivated less often, which saves time.
Therefore, for systems with strict requirements regarding jitter, c-states should be disabled to keep the cores of the CPU active. Of course, this comes at the cost of higher power consumption of the device. How to disable the c-states is described in the Appendix B.5

The results of the inverted pendulum demo with disabled c-states can be found in table 4.3. From there, one can see that the mean jitter for both the default Linux kernel as well as for the RT_PREEMPT kernel shows a much better result. The maximum jitter is even more important than the mean for a real-time system. Under the RT_PREEMPT kernel, there is also a significant improvement. However, this is not valid for the vanilla Linux kernel because this one is not preemptible and therefore not suitable for real-time systems.

Table 4.3: Resulting jitter for the inverted pendulum demo with disabled c-states

	Vanilla Linux kernel		RT_PREEMPT	
	unstressed	stressed	unstressed	stressed
Min	2.40 μs	2.46 μs	2.46 μs	2.50 μs
Mean	3.58 μs	3.70 μs	2.76 μs	2.85 μs
Max	930.71 μs	839.23 μs	37.01 μs	42.59 μs

One can see that the main advantage of a real-time capable operating system does not lie in a better overall performance. As seen in table 4.3, minimal jitter is for both kernel versions about the same. The same applies to the mean jitter. The RT_PREEMPT patched kernel only has a slightly lower mean jitter than the default Linux kernel. However, what could be dramatically decreased for the preemptible kernel is the maximal jitter. This better behaviour in the worst case makes a system deterministic, and therefore, real-time capable.

Each kernel's histogram is created to overview better how jitter values are distributed. Both show the case with additional stress since this is the most critical case.

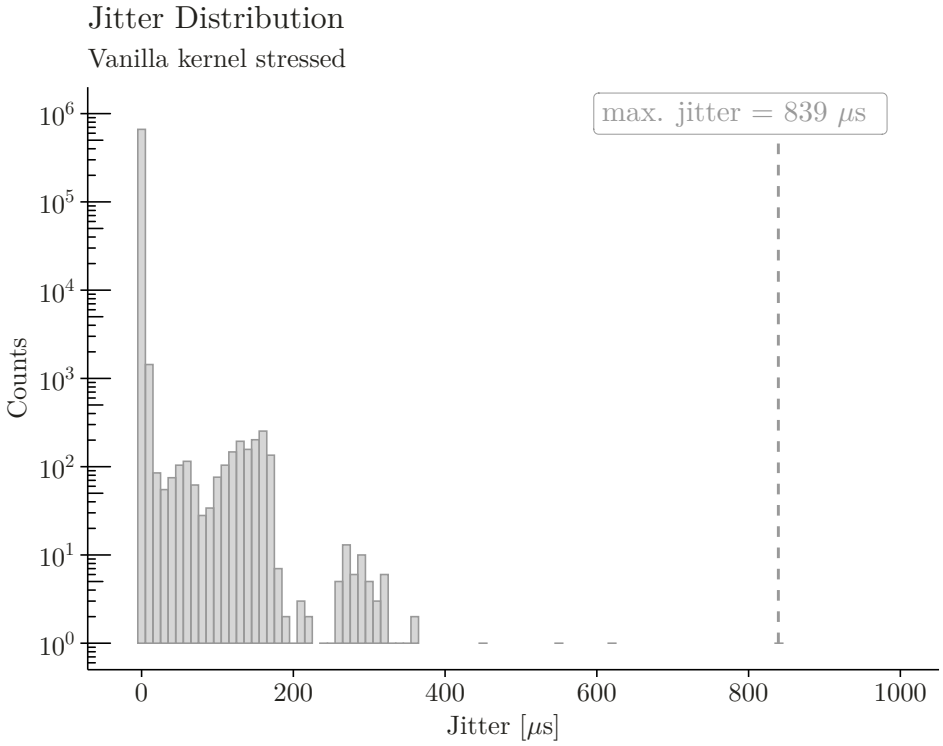


Figure 4.1: Jitter histogram for the unmodified vanilla Linux kernel

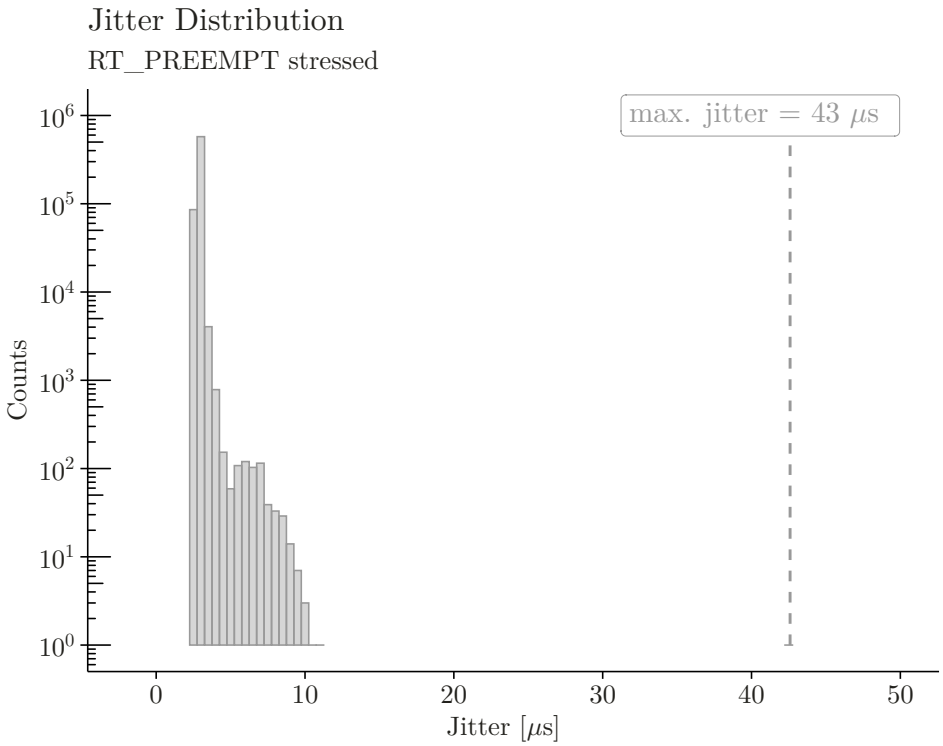


Figure 4.2: Jitter histogram for the patched Linux kernel

Chapter 5

Experimental Setup: Inverted Pendulum

As already stated in chapter 1, the goal of this thesis is to show and validate the real-time capabilities of ROS 2. This will be done as an actual world experimental setup with sensors and actuators communicating with ROS 2 instead of a virtual simulation setup. The performance and behaviour of ROS 2 will be determined through different measurements, and these measurements and the results are explained in section 5.6.
To put the performance results of ROS 2 in the proper context, they will be compared to a Programmable Logic Controller (PLC), an industrial solution for real-time systems.
As an experimental setup, the inverted pendulum case is chosen. This is a widespread and well-studied problem in control theory and is therefore well suited to show and compare the real-time performance of ROS 2. Summarised, the following points argue for the chosen setup:

- It is an unstable control system. Therefore reliable communication between sensors and actuator is needed to keep the pendulum stable.
- It is a commonly used example in research. Hence, the results of this thesis will be comparable to other similar research projects.
- It has only one degree of freedom and is, therefore, a good starting point.

The experimental setup and implementation will be explained in the following sections.

5.1 Conceptual Setup

In this thesis, the inverted pendulum is implemented with a reaction wheel to balance the pendulum bar at the stable point. In this case, the setup consists of the following main parts:

Base The base is fixed to the ground and can not move in any direction. The base is the mounting point for the pendulum bar.

Pendulum bar The pendulum bar is that part that should be balanced. It is mounted with a rotary joint to the base, and it, therefore, has one degree of freedom.

Reaction wheel The reaction wheel is responsible for keeping the pendulum bar at the stable point. By applying a torque onto the wheel and the law of angular momentum conservation, the pendulum bar can be balanced.

Motor The motor applies the torque onto the reaction wheel depending on the current state of the pendulum bar and the reaction wheel.

Sensors To measure the current state of the pendulum bar and the reaction wheel, different sensors are needed. The types of sensors used in this case will be determined in section 5.4.1.

All the mentioned parts except the sensors can be found in figure 5.1.

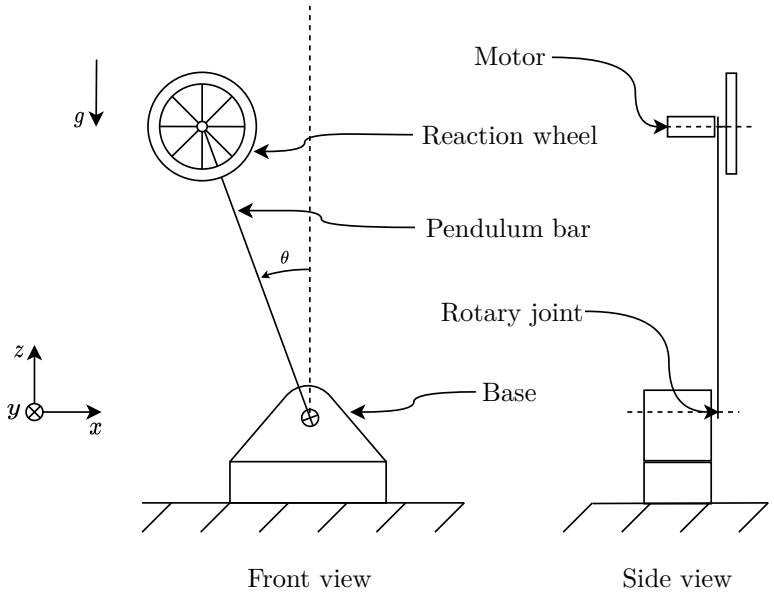


Figure 5.1: Conceptual draft including the main components of the inverted pendulum with reaction wheel

The system requirements must be determined before a more detailed version of the inverted pendulum can be elaborated. Appropriate components for the setup can only be defined if the requirements are known. The complete system specification can be found in appendix B.1. Here, only the most essential points are shown:

- The system shall be able to balance the pendulum bar at the $\theta = 0$ rad position.
- The starting position will be at $\theta = 0$ rad.
- A brushed DC motor shall drive the reaction wheel.

Furthermore, a risk assessment has been conducted to evaluate risks that could have a negative influence on the development and the operation of the inverted pendulum.
According to this risk assessment, the most significant risks during the development of the experimental setup are a supply shortage for the components needed and wrong-dimensioned components, which will lead to the collapse of the pendulum. To minimise or even prevent these risks, the following actions are taken:

To be as independent from delivery times as possible, the components used should, whenever possible, be taken from existing laboratory equipment. Secondly, simulation should be used to support the dimensioning process to prevent wrong dimensioned components. More on the simulation of the inverted pendulum can be found in chapter 5.3.
The complete risk assessment, including a risk matrix, can be found in appendix B.2.

5.2 Control System

The first step in controlling the pendulum is formulating the system's dynamic behaviour using differential equations. When the dynamic behaviour is known, the regulator can be chosen and adapted to this particular system.
In figure 5.2, taken from [18] and adapted, a schematic representation of the inverted pendulum can be found.

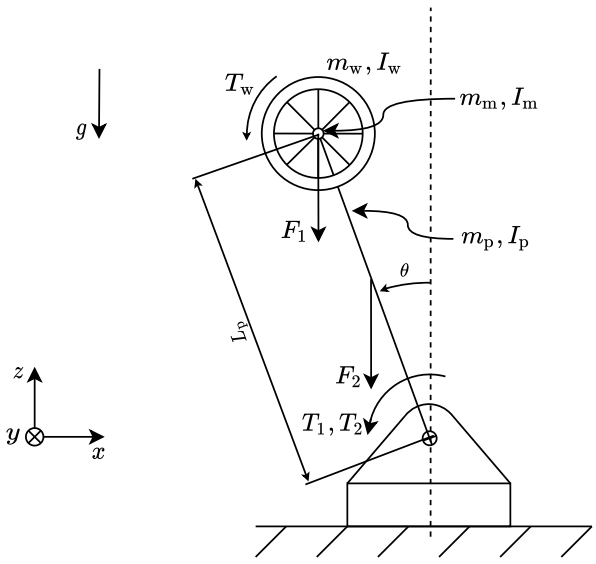


Figure 5.2: Schematic representation of the inverted pendulum with important variables for the control system

From figure 5.2, one can see that two torques generated by F_1 and F_2 will lead to the destabilisation of the pendulum. These torques result from the various masses on the pendulum and the deflection of the pendulum bar θ. The two forces are made up of the following variables:

$$F_1 = (m_\mathrm{w} + m_\mathrm{m}) \cdot g \tag{5.1}$$

$$F_2 = m_\mathrm{p} \cdot g \tag{5.2}$$

By taking the deflection angle θ into account, the generated torques around the lower joint of the pendulum bar can be written as follows:

$$T_1 = F_1 \cdot L_\mathrm{p} \cdot \sin(\theta) \tag{5.3}$$

$$T_2 = F_2 \cdot \frac{1}{2} L_\mathrm{p} \cdot \sin(\theta) \tag{5.4}$$

Together this leads to:

$$\begin{aligned} T_\mathrm{tot} &= T_1 + T_2 \\ &= (F_1 \cdot L_\mathrm{p} + F_2 \cdot \frac{1}{2} L_\mathrm{p}) \cdot \sin(\theta) \end{aligned} \tag{5.5}$$

To compensate for these two torques and stabilise the pendulum at $\theta = 0$, the motor can apply a torque T_w on the reaction wheel.
The resulting movement of the pendulum depends on these three torques and the total inertia of the pendulum I_tot and the viscose friction μ_j of the pendulum joint. According to [18], the dynamical system of the inverted pendulum with reaction wheel can be formulated as follows:

$$I_\mathrm{tot} \cdot \ddot{\theta} = T_\mathrm{tot} - \mu_\mathrm{j} \cdot \dot{\theta} - T_\mathrm{w} \tag{5.6}$$

$$I_\mathrm{tot} \cdot \ddot{\theta} = \underbrace{(F_1 \cdot L_\mathrm{p} + F_2 \cdot \frac{1}{2} L_\mathrm{p}) \cdot \sin(\theta)}_{T_\mathrm{tot}} - \mu_\mathrm{j} \cdot \dot{\theta} - T_\mathrm{w} \tag{5.7}$$

with:

$$I_\mathrm{tot} = I_{\mathrm{w,\,j}} + I_{\mathrm{m,\,j}} + I_{\mathrm{p,\,j}} \tag{5.8}$$

For simplicity reasons, this non-linear dynamical system will be linearised at the stable point $\theta = 0$ rad. According to the small-angle approximations, $\sin(\theta) = \theta$. This approximation leads to the following second-order linear differential equation:

$$I_\mathrm{tot} \cdot \ddot{\theta} = (F_1 \cdot L_\mathrm{p} + F_2 \cdot \frac{1}{2} L_\mathrm{p}) \cdot \theta - \mu_\mathrm{j} \cdot \dot{\theta} - T_\mathrm{w} \tag{5.9}$$

Starting with the dynamical system of equation 5.9, the model for the control system is built.
For the inverted pendulum case, a state-space model is used. Especially for multi-input or multi-output control systems, this kind of model is easier to use than other approaches [19]. Equations 5.10 and 5.11 show the general formulation of the model:

$$\dot{\mathbf{x}}(t) = \mathbf{A}_c \mathbf{x}(t) + \mathbf{B}_c \mathbf{u}(t) \qquad \text{State equation} \tag{5.10}$$

$$\mathbf{y}(t) = \mathbf{C}_c \mathbf{x}(t) + \mathbf{D}_c \mathbf{u}(t) \qquad \text{Output equation} \tag{5.11}$$

The system matrix $\mathbf{A}_c$ describes the system's dynamical behaviour, in this case, the inverted pendulum. In

contrast, the input matrix $\mathbf{B}_c$ describes how the inputs $\mathbf{u}(t)$ are linked to the states. The states can be seen as storage elements in the dynamical system.
On the other hand, the output matrix $\mathbf{C}_c$ and the feedthrough matrix $\mathbf{D}_c$ describe how the output $\mathbf{y}(t)$ depends on the input $\mathbf{u}(t)$ and the states $\mathbf{x}(t)$. The model is represented in a block diagram in figure 5.3 taken from [19].

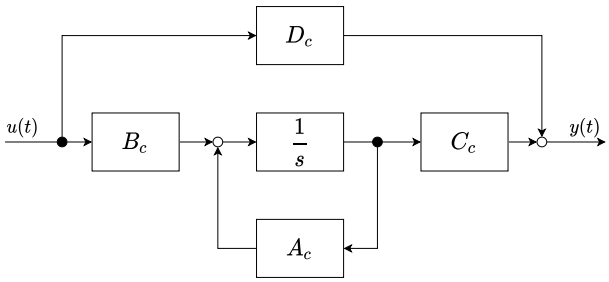


Figure 5.3: Block diagram of the state space model approach

Using the equation of the dynamical system (equation 5.9), the elements of the matrices $\mathbf{A}_c$ and $\mathbf{B}_c$ can be determined. To do so, some conversions need to be done in between. These can either be found in appendix B.3 or the original source [18]. In the main part of this thesis, only the results are presented:

$$\mathbf{A}_c = \begin{bmatrix} 0 & 1 & 0 \\ \frac{F_1 \cdot L_\mathrm{p} + F_2 \cdot \frac{1}{2} L_\mathrm{p}}{I_\mathrm{tot}} & -\frac{\mu_\mathrm{j}}{I_\mathrm{tot}} & \frac{k_\mathrm{t} k_\mathrm{e}}{R I_\mathrm{tot}} + \frac{\mu_\mathrm{m}}{I_\mathrm{tot}} \\ -\frac{F_1 \cdot L_\mathrm{p} + F_2 \cdot \frac{1}{2} L_\mathrm{p}}{I_\mathrm{tot}} & \frac{\mu_\mathrm{j}}{I_\mathrm{tot}} & -\frac{I_\mathrm{tot} + I_\mathrm{w}}{I_\mathrm{tot} I_\mathrm{w}} (\mu_\mathrm{m} + \frac{k_e k_t}{R}) \end{bmatrix} \qquad \mathbf{B}_c = \begin{bmatrix} 0 \\ -\frac{k_\mathrm{t}}{R I_\mathrm{tot}} \\ \frac{I_\mathrm{tot} + I_\mathrm{w}}{I_\mathrm{tot} I_\mathrm{w}} \frac{k_\mathrm{t}}{R} \end{bmatrix} \tag{5.12}$$

$$\mathbf{C}_c = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \qquad \mathbf{D}_c = \begin{bmatrix} 0 \end{bmatrix} \tag{5.13}$$

Further, the state space-model is extended to a state feedback control structure. This extension can be found in figure 5.4 from [19].

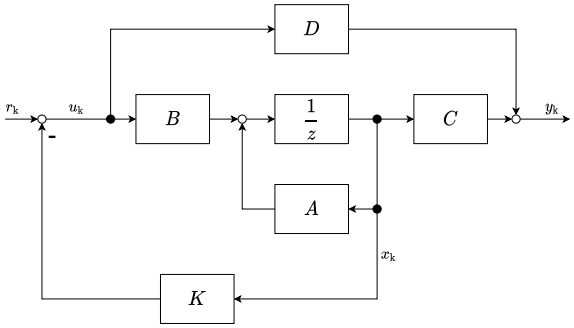


Figure 5.4: State feedback control structure

The so-called *linear quadratic regulator* is one method to compute the feedback parameters $\mathbf{K}$. A regulator is a system where the reference input r_k is equal to zero. The feedback parameters are determined by minimising a performance index. The formulation of this optimisation problem and an approach to solving the problem can be found in [19]. Solvers are among others implemented in Matlab.
With known feedback parameter $\mathbf{K}$, the input vector $\mathbf{u}_\mathrm{k}$ at a given state $\mathbf{x}_\mathrm{k}$ can be calculated with:

$$\mathbf{u}_\mathrm{k} = -\mathbf{K}\mathbf{x}_\mathrm{k} \tag{5.14}$$

5.3 Simulation

As mentioned in chapter 5.1, a rigid body simulation shall help design and dimension a setup that can balance the pendulum bar. Rather than setting up the simulation in CAD software, the simulation will be done within the ROS 2 environment. This has the advantage that not only the components can be tested, but also the control system presented above can be integrated and evaluated. Furthermore, the ROS 2 setup has a similar architecture to the one of the final real-world setup.
In this thesis, Gazebo is used to perform the simulations. Gazebo is a simulation tool designed to simulate the interaction of robots in indoor and outdoor environments. Included are physics engines which make Gazebo suitable for rigid body simulations like this one.

The simulation setup looks as follows: All the hardware needed is simulated. This comprises the pendulum parts like the pendulum bar or the reaction wheel but also all the sensors and the actuator. The controller is set up as a ROS 2 node as it will be later for the real-world setup. The controller needs the current state variables of the pendulum as inputs in order to calculate the actuating variable, in this case, the voltage applied on the DC-motor. Gazebo simulations can be extended by so-called plug-ins, which allow for the connection between the simulation and ROS 2 nodes. Hence, it needs two plug-ins for this case:

state_reader_plugin This plugin reads the states of the pendulum in the simulation and publishes these onto the *joint_states* topic. This includes the angle and angular velocity of the pendulum bar and the angular velocity of the reaction wheel.

reaction_wheel_torque_plugin This plugin feeds the generated torque back to the simulation and applies it onto the reaction wheel. Therefore, this plugin subscribes to the *wheel_torque* topic.

Figure 5.5 shows an overview of the simulation architecture. The *controller* node computes the motor voltage depending on the input variables coming from the Gazebo simulation. Because the physical behaviour of the DC-motor is not directly represented in the simulation environment, there is another ROS 2 node in-between the controller node and the simulation. In this *motor_simulator* node, the incoming voltage is converted into a torque, which will be applied to the reaction wheel in the simulation. The torque generated by a DC motor depends not only on the voltage applied to the motor but also on the current angular velocity of the motor. Therefore the *motor_controller* node also needs the information on the current angular velocity of the motor. Finally, the calculated torque is fed back to the simulation via the *reaction_wheel_torque_plugin*.
For the simulation, the inverted pendulum has been modelled in CAD software only with the main components shown in figure 5.1. The initial dimensions are chosen so that the setup can be easily carried around

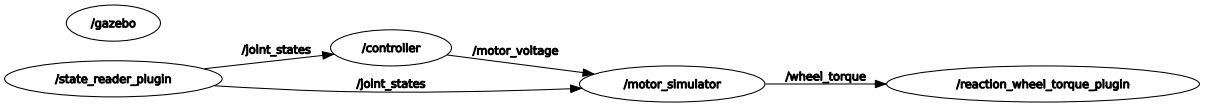


Figure 5.5: Architecture of the simulation with the two plugins at the begin and the end and the «normal» ROS 2 nodes in-between

and placed on a regular table. After modelling the parts in the CAD software, the model must be set up in Gazebo by the process shown in figure 5.6. This has been applied to each part.

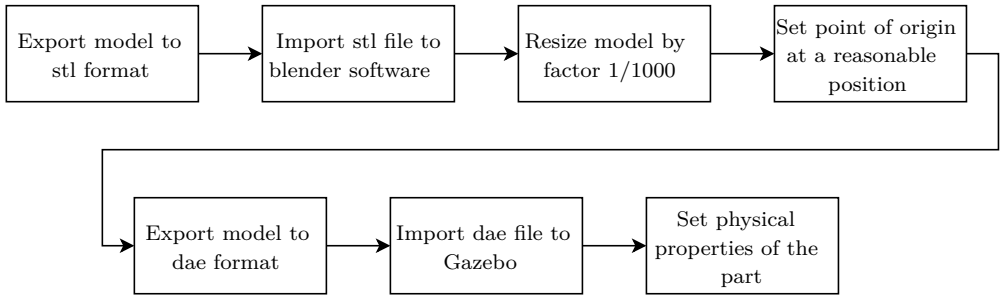


Figure 5.6: Process from CAD model to Gazebo model

The prepared parts can then be connected via joints. The final model will then be stored as an sdf-file. When the model is set up once this way, further changes to the model can directly be applied in the sdf-file, which is an XML format.

The first simulation run serves as the initial fix point. From this point, the initial dimensions can be adjusted to get a system that can balance the pendulum. To evaluate the performance of the pendulum in the simulation, the simulation is started with a pendulum bar deflection θ_{init}. The simulation will then show if the setup is able to bring the pendulum bar to the $\theta = 0$ rad position. For the first run of the simulation, the parameters in table 5.1 are used.

Table 5.1: Parameters for the first simulation run

Parameter	Value	Parameter	Value
θ_{init}	-0.05 rad	I_{w}	0.00019715 kgm^2
$\dot{\theta}_{\text{init}}$	0 rad/s	μ_{j}	0 Nm/(rad/s)
$\omega_{\text{w, init}}$	0 rad/s	R	21.5 Ω
L_{p}	0.18 m	L	0.00137 H
m_{p}	0.3731 kg	k_{t}	0.4028 Nm/A
m_{m}	0.2 kg	k_{e}	0.4028 Vs
m_{w}	0.1188 kg	μ_{m}	0 Nm/(rad/s)
I_{p}	0.006821 kgm^2	K	[-2414.8, -331.0, -1.4]
I_{m}	0.00001819 kgm^2		

The motor parameters are according to the data sheet of the chosen DC motor. The motor and gearbox data sheet can be found in appendix B.4. The gear box has a ratio of $i_{\text{gearbox}} = 19 : 1$. Therefore, the parameters k_{t} and k_{e} are multiplied by the factor 19 compared to the data sheet. The mass of the motor m_{m} is made up of the motor and the gearbox (ca. 0.11 kg) plus a margin (0.09 kg) for additional components of the joint, which are not defined by the time of the simulation.
Because the viscous friction coefficients μ_{j} and μ_{m} are unknown, they are set to zero in this simulation. However, the rotation joints will be implemented with ball bearings; therefore, friction is expected to have a minor impact on the dynamical system.
The feedback parameters $\mathbf{K}$ are calculated using a Matlab script (see electronic appendix in folder «10_Experimental_Setup_Inverted_Pendulum/30_Code/pendulum_control_system-main»).

The data recorded during the first simulation run shows that the setup is unable to stabilise the pendulum when starting at $\theta_{\text{init}} = -0.05$ rad. This behaviour is shown in figure 5.7. The deflection increases until the reaction wheel hits the floor short before 0.75 seconds.
In order to find possible actions to prevent the pendulum from falling, further data that has been recorded during simulation is analysed. The main factor for stabilising the pendulum bar is the torque applied on the reaction wheel, plotted in figure 5.8 (a). From there, one can see that the torque initially is high but starts to decrease fast and stays low until the reaction wheel hits the floor. As for all plots in figure 5.8, the magnitude is the value to consider. The sign only indicates the direction. The torque T generated by the motor depends beside the given motor constants on the one hand on the voltage U applied on the motor and on the other hand on the angular velocity ω of the motor shaft (equation 5.15).

$$T = \frac{k_{\text{t}}}{R} \cdot U - \frac{k_{\text{t}}}{R} \cdot k_{\text{e}} \cdot \omega \tag{5.15}$$

The recorded motor voltage can be found in figure 5.8 (b). It remains at the maximum for the motor in use. Hence, the motor voltage cannot be the reason for the drop in torque because the first term of equation 5.15 is constant over the simulation time. Figure 5.8 (c) shows the evolution of the angular velocity of the

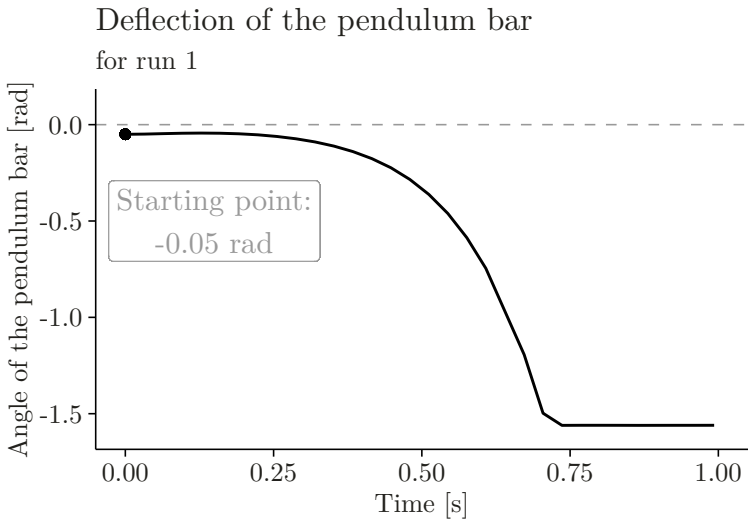
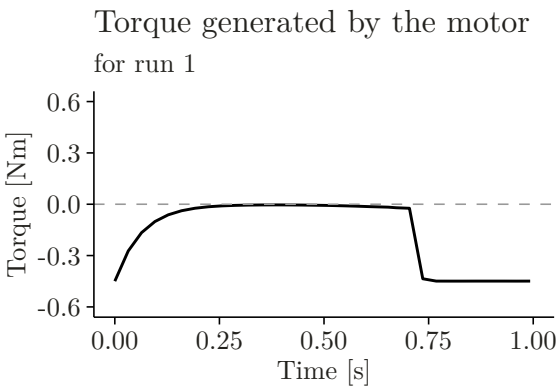
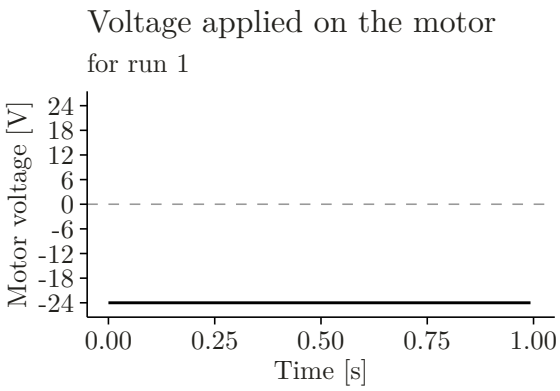


Figure 5.7: Recorded deflection angle of the pendulum bar for the first simulation run

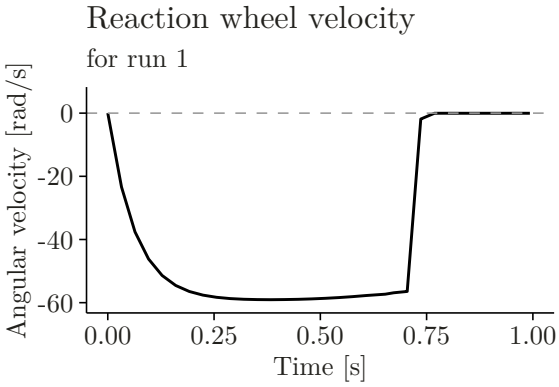
reaction wheel. One can see that the reaction wheel accelerates fast to its maximum angular velocity, which is the reason for the decreasing torque, despite the high motor voltage. The angular velocity of the reaction wheel starts to decrease slightly towards the point of impact. This is due to the position of the measurement: The angular velocity is measured relative between the reaction wheel and the upper joint of the pendulum bar. With the increasing angular velocity of the pendulum bar, the relative velocity between the pendulum bar and the reaction wheel slightly decreases.



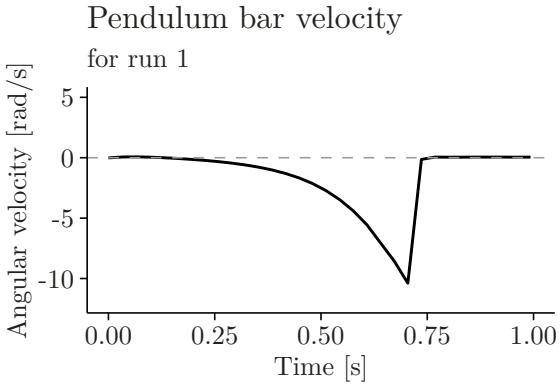
(a) Torque that has been generated by the motor



(b) Voltage that has been applied on the motor during simulation



(c) Rotation velocity of the reaction wheel



(d) Rotation velocity of the pendulum bar

Figure 5.8: Recorded data for the first simulation run

One way to get a high torque for longer is to prevent the reaction wheel from accelerating too fast. This can be done by increasing the inertia of the reaction wheel I_{w}. The most effective way to do so is to increase the diameter. In this case, the diameter increased from 100 mm to 140 mm for the second simulation run. This comes at a little more weight of the reaction wheel m_{w}. With the updated parameters of the reaction wheel, the feedback parameters K must be adapted. Table 5.2 shows the updated parameters compared to the first simulation run.

Table 5.2: Updated parameters for the second simulation run

Parameter	**Value**
m_{w}	0.1688 kg
I_{w}	0.0005901 kgm^2
K	[-546.4, -76.5, -1.3]

The second simulation run shows a different behaviour compared to the first one. With the reaction wheel's increased inertia, the setup can now stabilise the pendulum bar from the initial deflection of $\theta_{\text{init}} = -0.05$ rad. This is shown in figure 5.9. In the beginning, the pendulum oscillates around the stable point and with progress in time, the pendulum converges to the stable point. This behaviour could be changed by tuning the feedback parameters K. However, this lies not within the scope of this simulation.

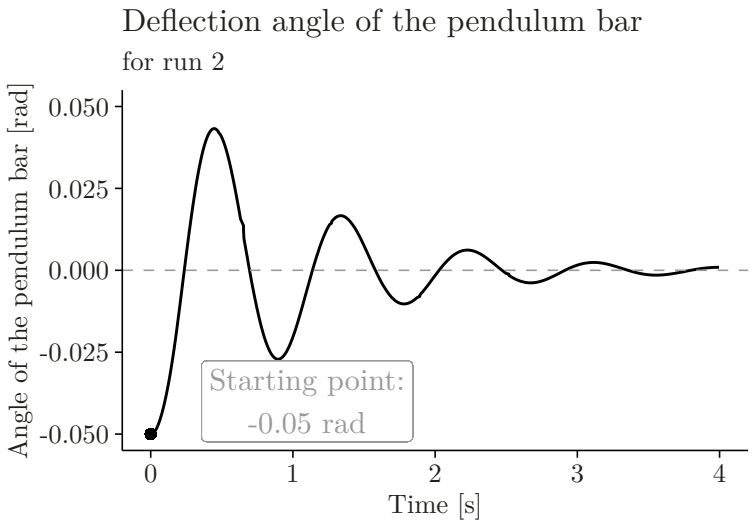
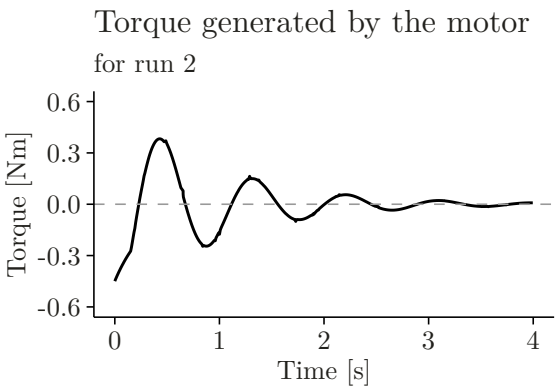
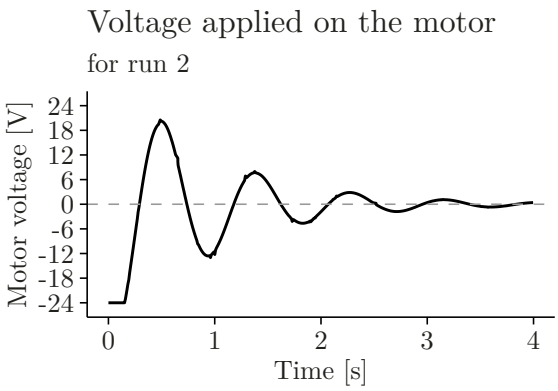


Figure 5.9: Recorded deflection angle of the pendulum bar for the second simulation run

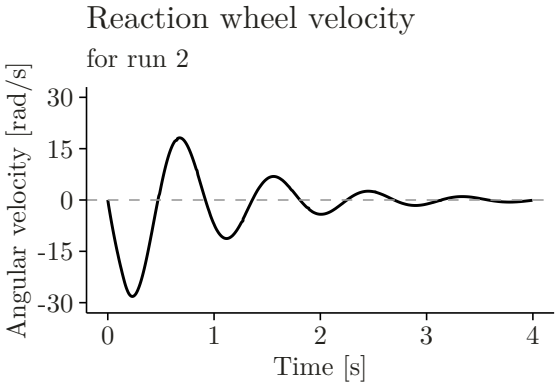
For completeness, the rest of the recorded data is shown in the same way as for the first simulation run (figure 5.10). From there, one can see differences from the first run coming from the increased inertia of the reaction wheel. Especially in figure 5.10 (c), one can see that the maximum angular velocity of the reaction wheel is only about one-half of the maxima in the first run (compare figure 5.8 (c)).



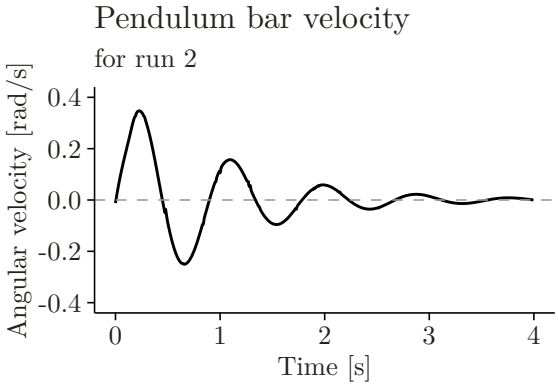
(a) Torque that has been generated by the motor



(b) Voltage that has been applied on the motor during simulation



(c) Rotation velocity of the reaction wheel



(d) Rotation velocity of the pendulum bar

Figure 5.10: Recorded data for the second simulation run

The second run proves that the setup is able to balance the pendulum bar, even though θ_{init} is chosen small with -0.05 rad. The small recovery angle is chosen on purpose, because then reliable information flow gets more important.

5.4 Final Setup

5.4.1 Mechanical Setup

Based on the findings of the simulation, the final mechanical setup is elaborated. The final setup differs most from the conceptual setup used in the simulation in terms of joints and mounting points. The main components of the final setup, also shown in figure 5.11, are:

Frame The aluminium frame is the mounting point for the base of the pendulum as well as for the stopper. For safety reasons, the frame shall offer the ability to be extended by a closed cage according to the requirements specification in appendix B.1.

Base The base is the fixed part of the lower joint.

Stopper The stopper prevents the pendulum bar from a too high deflection angle. Otherwise, the reaction wheel or the distance sensor could get damaged. The stopper allows the pendulum bar to move ± 15 deg.

Axle The axle is the pivot for the pendulum bar and is fixed to the two bases.

Distance sensor The laser distance sensor measures the distance to the pendulum bar, and this linear displacement is then converted into the deflection angle θ. Furthermore, also the angular velocity $\dot{\theta}$ is determined by this sensor. The simulation in section 5.3 shows that θ and $\dot{\theta}$ are the most critical states for the control system (the first two values of feedback parameters K are far bigger than the third value). Therefore a precise measurement of the two states is crucial. The laser distance sensor guarantees this high accuracy. The data sheet of the sensor can be found in appendix B.4.

Pendulum bar The pendulum bar is pivoted on the axle with two ball bearings. At the free end, the motor with the gear box is mounted to the pendulum bar.

Pendulum bar counter support The pendulum bar counter support is mounted on the pendulum bar and serves as a mounting point for the encoder. The encoder is connected to the reaction wheel via the encoder flange. This is shown in figure 5.12.

Encoder The incremental encoder is responsible for measuring of the angular velocity of the reaction wheel ω_{w}. The encoder has a resolution of 1000 pulses per revolution which results in one pulse per 0.36 deg. Because ω_{w} is the least important state to control the system, the coarse resolution of the encoder has only a minor influence on the control system. The data sheet of the encoder can be found in appendix B.4.

Motor + Gear box The gear box motor applies torque onto the reaction wheel. A motor with gear box has been chosen because high torque is of interest for this application. The step-down ratio of the gear box supports this demand. The motor and gear box data sheet can be found in appendix B.4.

Reaction wheel The reaction wheel gets accelerated and decelerated by the motor. From the simulation in chapter 5.3 is known that the greater the moment of inertia of the reaction wheel, the longer the motor can generate high torque.

In figure 5.12, the detailed implementation of the upper revolute joint can be seen as a section view. The reaction wheel is supported on both sides by ball bearings. This guarantees that neither the axle of the gear box nor the encoder is loaded with radial forces. The motor flange, the connection between the gear box shaft and the reaction wheel, is fixed with a stud bolt to the gear box shaft.

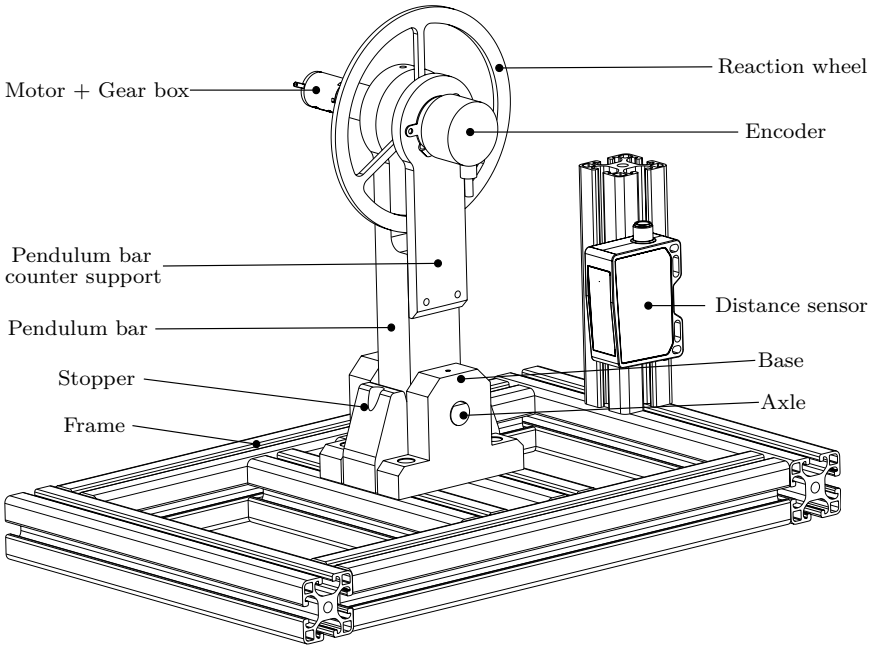


Figure 5.11: Overview of the final setup of the inverted pendulum with reaction wheel

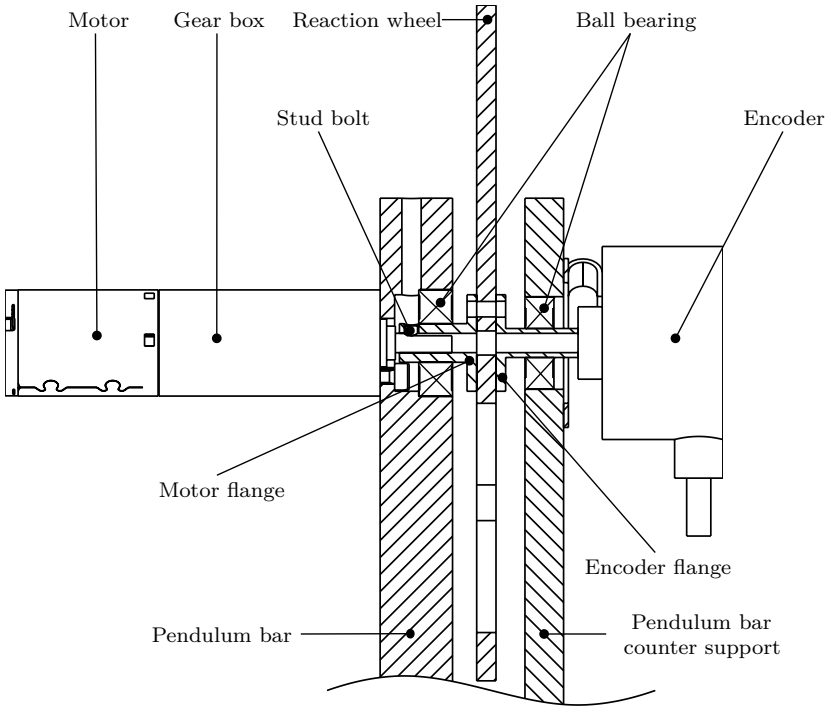


Figure 5.12: Section view through the upper revolute joint

5.4.2 Electrical Setup

This section considers the components of the experimental setup from the electrical point of view. This comprises mainly how the components are supplied with an accurate voltage and how the signals are handled. Figure 5.13 shows a general overview of the electrical setup. The arrows for the Signal / Data connections indicate the direction of the data transmission.
The notebook runs the controller to stabilise the inverted pendulum. In order to compute the actuating variable, the information from the sensors is needed. Because many computers do not have hardware inputs/outputs, additional hardware is needed. In this case, two microcontrollers are used to interact with the sensors and the motor. The main requirements for this additional hardware are analogue voltage input for the distance sensor, hardware interruptibility for the encoder and pulse-width modulation control for the H-bridge. Therefore, a microcontroller is well-suited hardware to fulfil these requirements. In order to communicate with a microcontroller, ROS offers new possibilities with micro-ROS. This can be seen as the successor of the *rosserial* library in ROS 1. It brings all the core functionalities of ROS 2 to microcontrollers. This offers the possibility to exchange messages between a notebook and a microcontroller over an USB port via the common publisher/subscriber model in ROS 2. More detailed information about the tasks performed on the various hardware can be found in section 5.4.3. By the time of writing, only a limited amount of microcontrollers are supported by micro-ROS. An Arduino Due has been chosen first because it is widely spread and, therefore, many resources exist. However, the first experiments with micro-ROS showed that the Arduino Due could not publish messages on three different topics at a rate of 100 Hz. Therefore, a more powerful microcontroller shall be used. According to their website, the CPU of the Teensy 4.1 is many times faster than other typical 32-bit microcontrollers [20]. Furthermore, the development can be integrated into the Arduino IDE and programs written for an Arduino microcontroller can also be executed on a Teensy 4.1. A first test showed that the Teensy 4.1 is able to reach a publishing rate of 100 Hz for three topics. However, both microcontrollers are used during operations to split the workload. The encoder reading is done on the Arduino Due. Therefore the interrupts can not affect the measurement or publishing of the messages related to the distance sensor, and the workload can be shared.
Both microcontrollers operate at a voltage of 3.3 V. However, the sensors in use have higher operation voltages. The encoder operates at 5 V and the analogue output of the laser distance sensor ranges from 0 V to 10 V. Feeding these voltages directly to the microcontrollers could damage them. Therefore, for both sensors, a voltage divider is introduced. This ensures that the voltage the microcontrollers face never exceeds 3.3 V. A schematic representation of a voltage divider can be found in figure 5.14. In table 5.3 the corresponding values for the chosen resistors are listed. Because the encoder and the laser sensor have different operating voltages, each sensor needs a specific voltage divider.

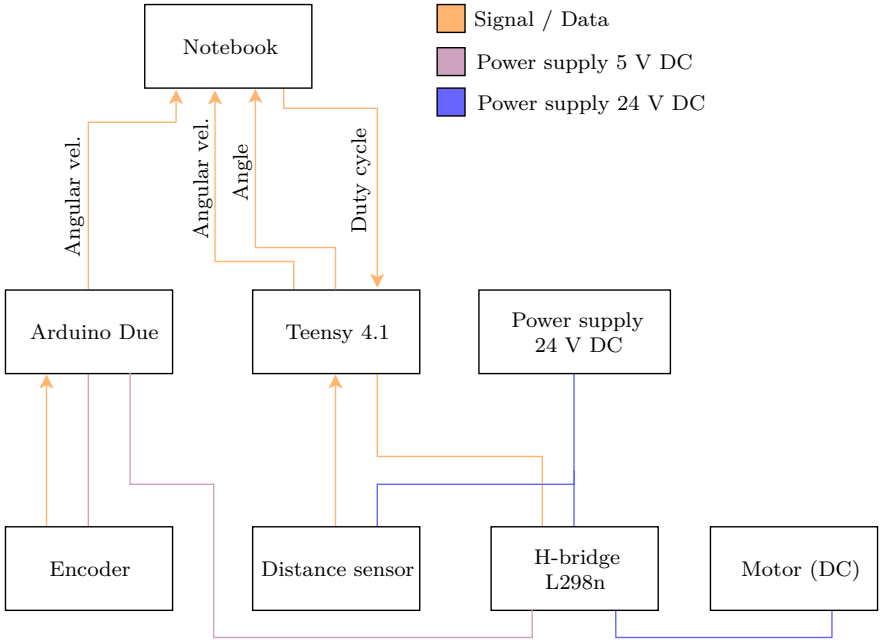


Figure 5.13: General overview of the electrical setup. Arrows indicate the direction of data transmission

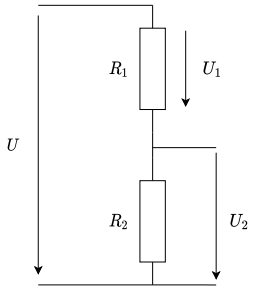


Figure 5.14: Schematic representation of a voltage divider

Table 5.3: Used resistors for two individual voltage dividers

	R_1	R_2
Encoder	10000 Ω	6800 Ω
Laser Sensor	1000 Ω	470 Ω

In figure 5.14, the voltage U corresponds to the output voltage of the respective sensor. The voltage fed to the microcontrollers is then U_2 with:

$$U_2 = \frac{R_2}{R_1} \cdot U_1 \tag{5.16}$$

$$U = U_1 + U_2 \tag{5.17}$$

Furthermore, the laser sensor's signal must be filtered to reduce the measurement noise. Noisy measurements can lead to instability of the pendulum. Therefore, a moving average filter is applied to the signal reading. The difficulty in signal filtering is to get sufficient noise reduction while not adding too much delay to the system. This can be tuned by changing the number of past measurements that are taken into account for the average computation. In this case, this has been done by testing different setups and evaluating their influence. A size of three has been evaluated as a reasonable trade-off for this system.

5.4.3 ROS 2 Architecture

The ROS network consists of three nodes. One on each microcontroller and one on the notebook. A graphical representation of the network can be found in figure 5.15. The main components are:

arduino_node The *arduino_node* is a micro-ROS node running on the Arduino Due. It is responsible for the measurement of the reaction wheel speed. Therefore, it measures the time elapsed between two subsequent encoder pulses and computes the number of pulses per second from this information. This is then published on the *encoder* topic.

teensy_node The *teensy_node* is another micro-ROS node running on the Teensy 4.1. On this node, the current distance to the pendulum bar is measured. Furthermore, the difference between two subsequent distance measurements is determined and divided by the elapsed time. This yields the velocity information of the pendulum bar. This information is published on the *laser_sensor* and *laser_sensor_vel* topic, respectively. Another duty of the *teensy_node* is to set the PWM signal, which is then fed to the H-bridge in order to drive the motor. Therefore, the node subscribes to the *duty_cycle* topic.

controller The *controller* is a normal ROS 2 node. It is running on the notebook. It subscribes to the *encoder*, *laser_sensor* and the *laser_sensor_vel* topic to receive all the needed sensor information. With this information converted to the suitable unit, the *controller* node is able to calculate the actuating variable to stabilise the pendulum. This is then converted into a duty cycle and sent to the *teensy_node* via the *duty_cycle* topic.

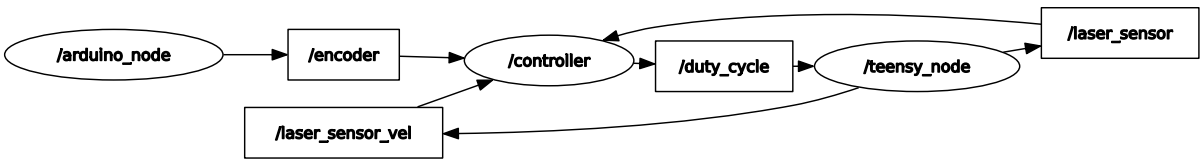


Figure 5.15: ROS graph for the distributed system of the inverted pendulum

The ROS 2 code is written in C++. This is a programming language that is considered efficient and hardware-related programming. For real-time applications, it is essential to be in complete control over memory, CPUs etc. which is why this programming language is preferred over Python. The code can be found in the electronic appendix under «10_Experimental_Setup_Inverted_Pendulum/30_Code/pendulum_controller-main».

5.5 Initialisation Phase

This section describes the difficulties encountered during the initialisation phase and the subsequent actions.

The dynamical model of the inverted pendulum expects linear behaviour between the motor voltage and the torque generated by the motor. This is the case if a steady voltage source drives the motor. However, in this case, the voltage must be variable in order to control the inverted pendulum. Hence, the motor is driven

with PWM forcing from the H-bridge. During one period of the PWM signal, there are two different states: Low and high. Depending on the behaviour of the H-bridge during the low state, one distinguishes two operation modes of the H-bridge: Drive/coast and drive/brake. When using drive/coast mode, the motor can spin without resistance during the PWM signal's low phase, which leads to highly non-linear system behaviour. In contrast, in drive/brake mode, the motor is slowed down during the low phase of the PWM signal by its back EMF and therefore, linear assumption holds [21]. As a result, the H-bridge must be used in drive/brake mode rather than drive/coast mode.

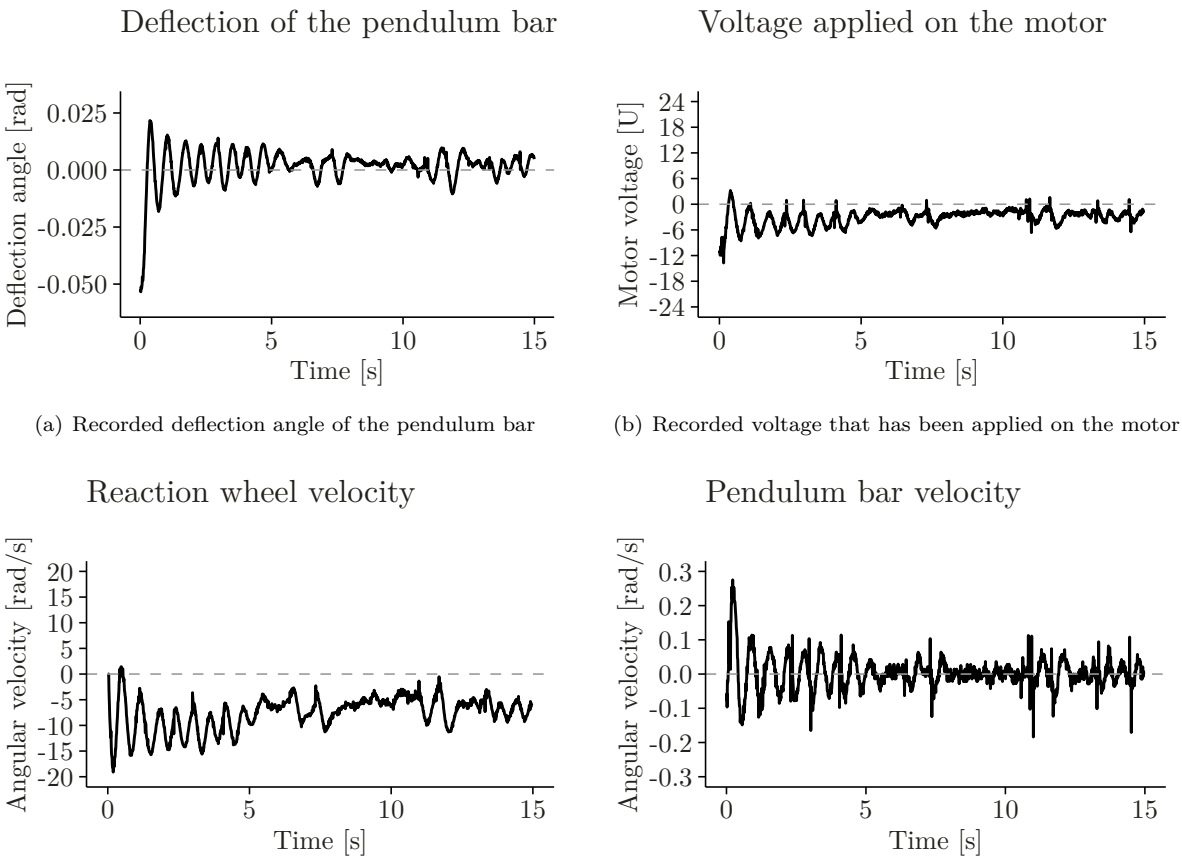
The PWM frequency is another thing to take care of related to the PWM signal. Because the motor is modelled as an RL-circuit, the current rises exponentially when the PWM signal is high. It takes about five times the L/R constant until the current reaches the steady-state. If the PWM frequency is too high and therefore, the high phase of the signal is too short, the current is not able to reach its steady-state. In this case, the motor is not able to generate the torque which is expected. This problem mainly occurs for low duty cycles, namely when the high phase of the signal is short compared to the period time. To prevent this, the PWM frequency should be chosen so that the high phase of the signal is longer than five times the L/R constant of the motor, also for low duty cycles.

The initial parameters for the controller are first derived by the dynamical model of the inverted pendulum in Matlab and are then tuned by hand on the real experimental setup. The final control parameters for the closed-loop system are:

$$K = [-180, -25, -0.45] \tag{5.18}$$

The update rate of the ROS 2 nodes is set to 100 Hz for all the nodes. According to the dynamical model, the fastest pole is located at -2.45 Hz and therefore, 100 Hz should be a sufficiently high sampling frequency.

The recorded pendulum behaviour with these settings in place is shown in figure 5.16. Similar to the simulations above, the inverted pendulum is started at an initial deflection angle. From there, the pendulum is able to stabilise. However, compared to the simulation, the recorded data shows more noise which is due to the sensor noise of the laser distance sensor. Furthermore, different from the simulations, where all the values converged to zero, in the real experimental setup, the values oscillate around a steady offset (except for the pendulum bar velocity). In contrast to the ideal simulation, the real pendulum differs form the ideal condition. Wiring and fabrication tolerances can, for example, lead to a displacement of the centre of mass. In addition, the table on which the pendulum is located might not be perfectly horizontal. This can lead to the observed behaviour of the pendulum.



(a) Recorded deflection angle of the pendulum bar

(b) Recorded voltage that has been applied on the motor

(c) Recorded reaction wheel velocity

(d) Recorded pendulum bar velocity

Figure 5.16: Recorded sensor data for a real world run of the inverted pendulum

5.6 Experimental Procedure

A set of experiments are executed to assess the performance of ROS 2. The experiments focus on the *controller* node, which is a ROS 2 node and runs on the notebook. The behaviour of the micro-ROS nodes is only considered marginally.
The primary performance indicator for the assessment is the maximal jitter (see section 2.1) that occurs for the publishing of the duty cycle messages.
The experiments conducted can be summarised into the following five groups:

1. **Influence of various system settings**. The influence of different system settings shall be evaluated in the first sequence of experiments. The result will show which setting to focus on while developing new applications.

2. **Evaluation of the system limits.** Based on the results of the first sequence of experiments, a setting shall be chosen which minimises jitter. The limitation of the system shall be evaluated in an extended test run.

3. **Influence of the cycle time.** This experiment shall evaluate how the system can cope with shorter cycle times.

4. **Jitter on micro-ROS nodes.** As stated above, for completeness, the jitter on micro-ROS nodes is also evaluated.

5. **Comparison to PLC.** In the last experiment, the performance of the ROS 2 node will be compared to a PLC. Therefore, the program to balance the pendulum is implemented on a Beckhoff PLC. The jitter of the PLC cycle is compared to the jitter of the *controller* node.

All the described experiments are done on the same notebook described in section 4.1. The specifications can be found in table 4.1 in the column of the RT_PREEMPT kernel.
To measure jitter, the elapsed time between two subsequent calls of the publisher callback function is measured. Subtracting the ideal cycle time yields the jitter.
The timestamps of the integrated logging possibilities of ROS 2 were not sufficiently accurate to measure the jitter. One drawback of the ROS 2 integrated logging is that it saves the data to file during the program execution, which is not desirable for real-time processes. In order to not disturb the real-time tasks, the measurements are done by means of a standard C++ function and the results are saved to file after the experiment is done. From the second experiment on, the tests were conducted with the pendulum at rest in order to not keep the pendulum running for multiple hours.

5.6.1 Influence of Various System Settings

There are different setting options to reduce the jitter in a ROS 2 network. Some of them are ROS 2 settings, e.g. QoS parameters. Others are settings related to the operating system. In this first sequence of experiments, the goal is to show which settings are the most important in order to reduce the maximal jitter. In the scope of this thesis, the following settings are evaluated (all the commands needed are listed in Appendix B.5 and B.6):

Reliability The QoS parameter reliability is set to *best effort*. This setting does not ensure that the message is received on the other side. However, the message is sent with less overhead. With the default setting (reliable), a message is sent multiple times if it is not received on the other side. This can lead to an unwanted blockage of the network. Default: reliable. [22, 23]

History This QoS parameter defines how many messages are stored in a buffer. For performance reasons, in this case, only one message is stored. Default: 10. [22, 23]

Deadline The deadline Qos parameter specifies the maximal acceptable timespan between two subsequent messages published. In this case, the deadline is set equal to the publishing period, which is ten milliseconds Default: No deadline. [23]

Static Single-Threaded Executor The executor is part of a ROS 2 node responsible for invoking the timers and the subscription callbacks. In contrast to the other executors, the *static single-threaded executor* does only search one time which timer and callbacks exist in the node. This increases the performance. On the other hand, this executor can only work correctly if all timers and callbacks are built during the initialisation period [23]. Default: Single-Threaded Executor.

Multi-Threaded Executor The multi-threaded executor can create different threads in order to process messages in parallel [24]. Default: Single-Threaded Executor.

Priority The priority tells the operating system which tasks are essential and should be preferred over others. The priority of the ROS 2 node is set to 98. Default: 20.

Memlock This setting ensures that no memory is allocated during runtime. Default: Off.

CPU Affinity The CPU affinity setting allocates the ROS 2 node to a specific CPU core. Default: Off, the CPU core on which the task is performed can change during runtime.

CPU Affinity + Isolation With this setting, not only the CPU core is defined on which the task is performed. Moreover, no other task is allowed to use this core. Default: Off.

To compare the influence of the different settings, all the settings explained above are set to their default value in the first run. This result serves as a baseline. After the baseline run, each of the above settings is tested individually. That means that only one setting is turned on at the time while all other settings are set to their default value. Each run is done twice, once with normal system load (referred to as unstressed) and once with additional, simulated system load (referred to as stressed). The additional load is created with the *stress debian package* [25]. In the stressed case, three of the four available CPU cores are fully used to capacity (find the shell command in Appendix B.5). All the test runs are done throughout 60'000 messages. Cycle time is set to ten milliseconds. Combined, this results in a test duration of ten minutes.

Figure 5.17 and table 5.4 show the results for the first sequence of experiments. Represented are the percental reduction of the maximal measured jitter compared to the baseline test (unstressed and stressed, respectively). The figure shows that the two settings with the heaviest influence are *priority* and *CPU affinity + isolation*. In contrast to other settings, these two show substantial performance increases in both cases, unstressed and stressed. On trend, many settings tend to have relatively small effects when additional load is applied. In the stressed case, the high load is likely dominating the jitter. However, with high *priority*, the operating system is able to preempt the low priority system load. With *CPU affinity + isolation* in place, the additional load does not influence the core running the ROS 2 tasks. Therefore, these two settings are able to handle systems with high loads, and *cpu affinity + isolation* should be preferred over stand-alone *cpu affinity*.
Another interesting point is the behaviour of the two executors: While the *static single-threaded executor* shows superior performance for the unstressed case, the *multi-threaded executor* shows slightly better behaviour in the stressed case. Because in the case of the executor, only one or the other can be used, this will be further examined in a longer test run in section 5.6.2.

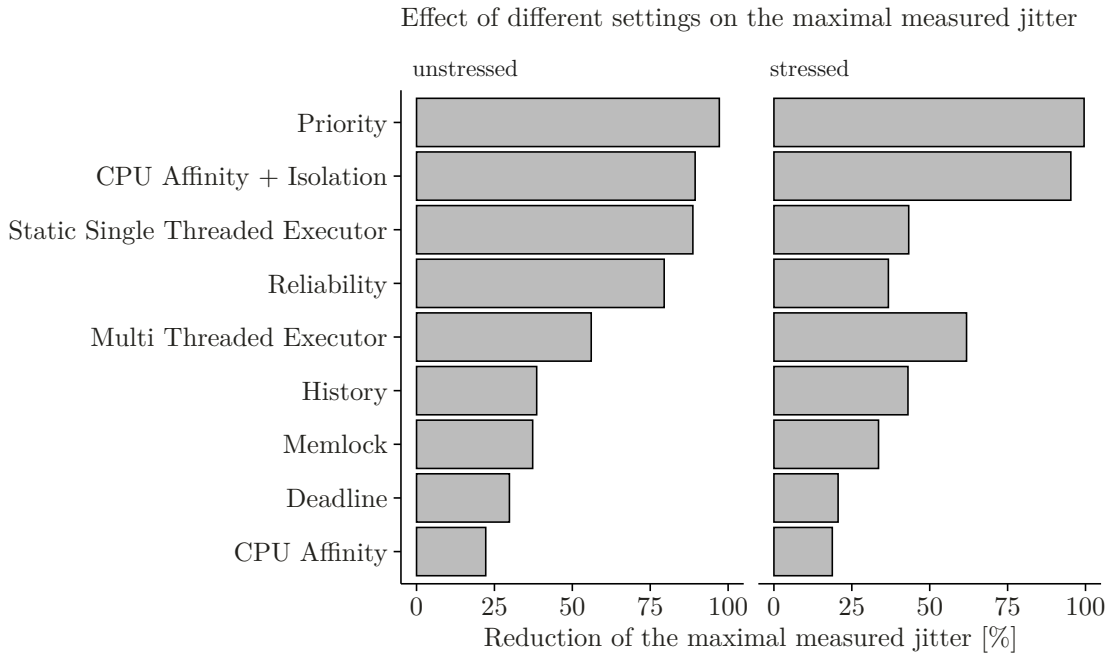


Figure 5.17: Effect of different jitter reduction settings compared to a baseline measurement

Table 5.4: Numeric results of the system settings comparison

	unstressed		stressed	
	max. Jitter	Jitter reduction	max. Jitter	Jitter reduction
Baseline	8633 μs	-	17'764 μs	-
Reliability	1772 μs	79.47 %	11'238 μs	36.74 %
History	5305 μs	38.56 %	10'120 μs	43.03 %
Deadline	6060 μs	29.81 %	14105 μs	20.60 %
Static Single-Threaded Ex.	977 μs	88.69 %	10'079 μs	43.26 %
Multi-Threaded Ex.	3794 μs	56.06 %	6782 μs	61.82 %
Priority	244 μs	97.18 %	80 μs	99.55 %
Memlock	6716 μs	37.27 %	11801 μs	33.57 %
CPU Affinity	6716 μs	22.21 %	14435 μs	18.74 %
CPU Affinity + Isolation	914 μs	89.41 %	836 μs	95.30 %

5.6.2 System Limits

In the section above, the different settings are examined separately regarding their effect on the maximal measured jitter. In this section, however, the goal is to find the limits of a real-time system built with ROS 2 when all settings are activated. Because either the *static single-threaded executor* or the *multi-threaded*

executor can be in use, the first step is to further evaluate the performance of the two executors through longer test durations. In this setup, the test duration is set to two hours.
Because a real-time system must meet the constraints in worst-case conditions, further evaluation is done only for the stressed system. However, this time all the settings from section 5.6.1 are applied and only the executor is changed for the two test runs.

The jitter histogram for the two different runs in figure 5.18 shows similar results for both executors with slightly better results for the *static single-threaded executor*. Not only the maximal measured jitter is smaller for the *static single-threaded executor*, but also the mean of the absolute jitter values and the standard deviation (see table 5.5). Only for the minimal measured jitter, which refers to how much to early a message is sent, the *multi-threaded executer* shows the better result. However, a message sent too early is less critical for the stability of a pendulum or similar control systems than massages sent too late.

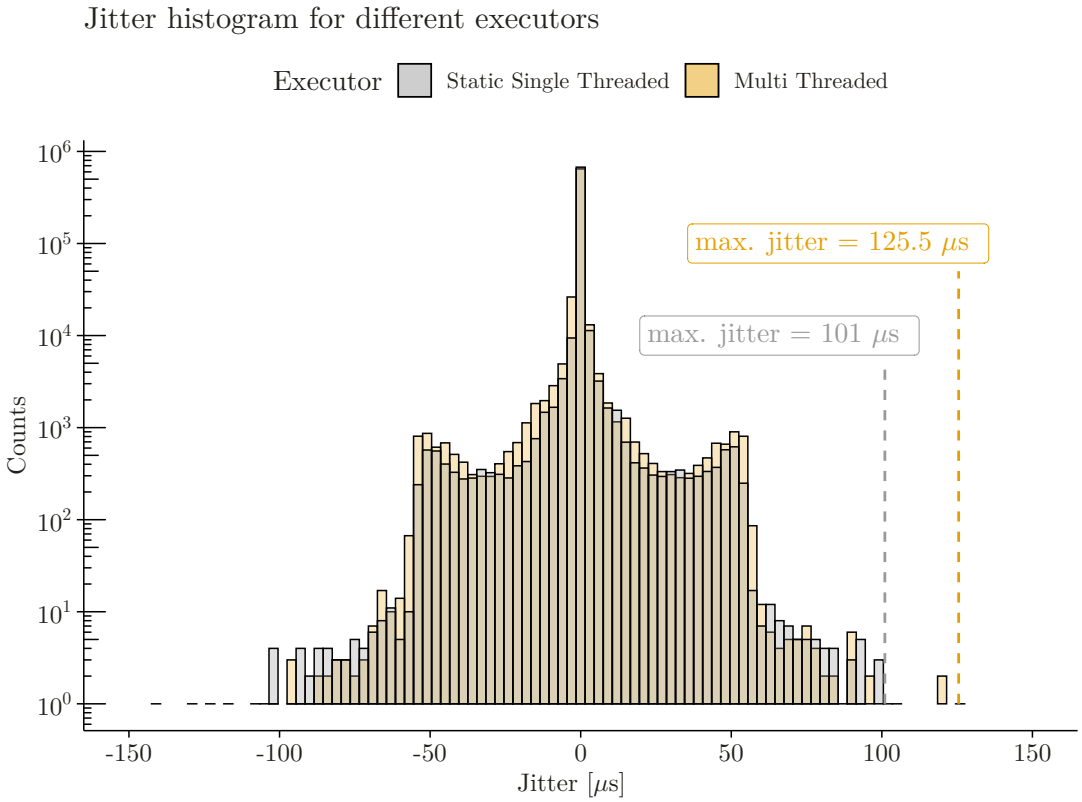


Figure 5.18: Comparison of the jitter distribution for the static single-threaded executor and the multi-threaded executor with all other settings enabled.

This evaluation shows that the *static single-threaded executor* is better suited to reduce jitter than the *multi-threaded executor*. From the first sequence of experiments in section 5.6.1, one could expect the inverse result for a stressed system. But with *CPU affinity + isolation* enabled, the system behaves more like an unstressed system because the simulated stress can not influence the CPU core on which the ROS 2 node is running. Therefore, the *static single-threaded executor* is preferred over the multi-threaded executor.

Table 5.5: Results of the two test runs with different executors

Executor	**Max**	**Mean (absolute)**	**Min**	**Standard Deviation**
Static Single Threaded	101 μs	0.97 μs	-141.6 μs	4.79 μs
Multi Threaded	125.5 μs	1.43 μs	-128.6 μs	5.96 μs

In order to get a benchmark in which magnitude the maximal jitter for a ROS 2 publisher is, two four-hour tests are performed. Together with the two-hour test from above (*static single-threaded executor* in figure 5.18 and table 5.5), this leads to a total test time of ten hours.
The evaluation shows that the maximal jitter from the two-hour test run is not exceeded in the two four-hour runs. This indicates that the maximal jitter for a ROS 2 publisher with all the described settings can be expected to be around 100 μs. Compared to the cycle time, the jitter is around one per cent of the cycle time, which should be acceptable for many applications in mobile robotics. A graphical representation of the jitter distribution over the long-term test can be found in figure 5.19, while the numerical results are shown in table 5.6.

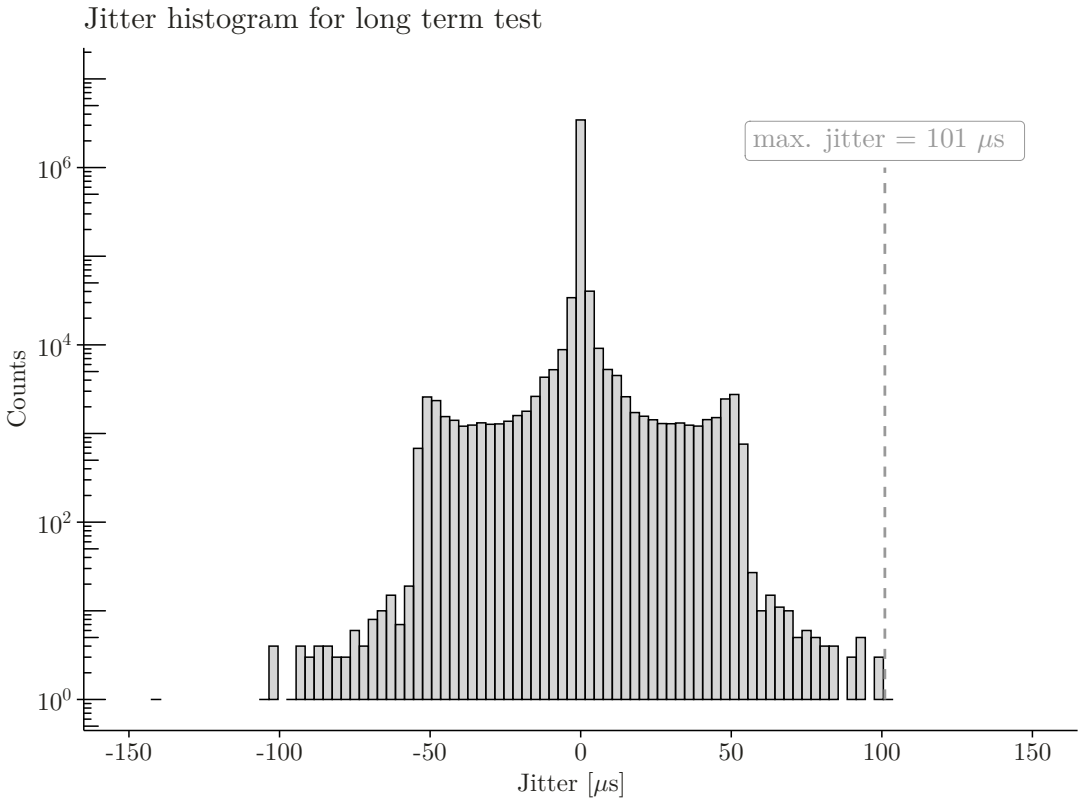


Figure 5.19: Jitter distribution for long-term test with all settings enabled

Table 5.6: Result of the long-term test

	Max	**Mean (absolute)**	**Min**	**Standard Deviation**
Long-term test	101 μs	0.78 μs	-141.6 μs	4.22 μs

5.6.3 Cycle Time

All the tests described to this point are conducted with a cycle time of ten milliseconds. In this test, the goal is to determine how a decreased cycle time to one millisecond impacts the jitter. As a reference, the two-hour test run of section 5.6.2 with the *static single-threaded executor* is taken. For the shorter cycle time test run, same number of messages are sent, but with shorter cycle time the test duration decreased.
At first glance, the ROS 2 node seems to perform better for the shorter cycle time (see figure 5.20). However, with a closer look, one can determine that this is not totally true. On the one hand, the maximal measured jitter in the test run with one millisecond cycle time is drastically reduced to 62.2 μs. On the other hand, if the jitter is evaluated in relation to the respective cycle time, the reduction in cycle time leads to an increase of the relative jitter to 6.22 %. Not only the relative jitter increase but also the mean jitter of the absolute values as well as the standard deviation. This might be due to the higher system load coming from the increased frequency. In future studies, this could be further investigated.

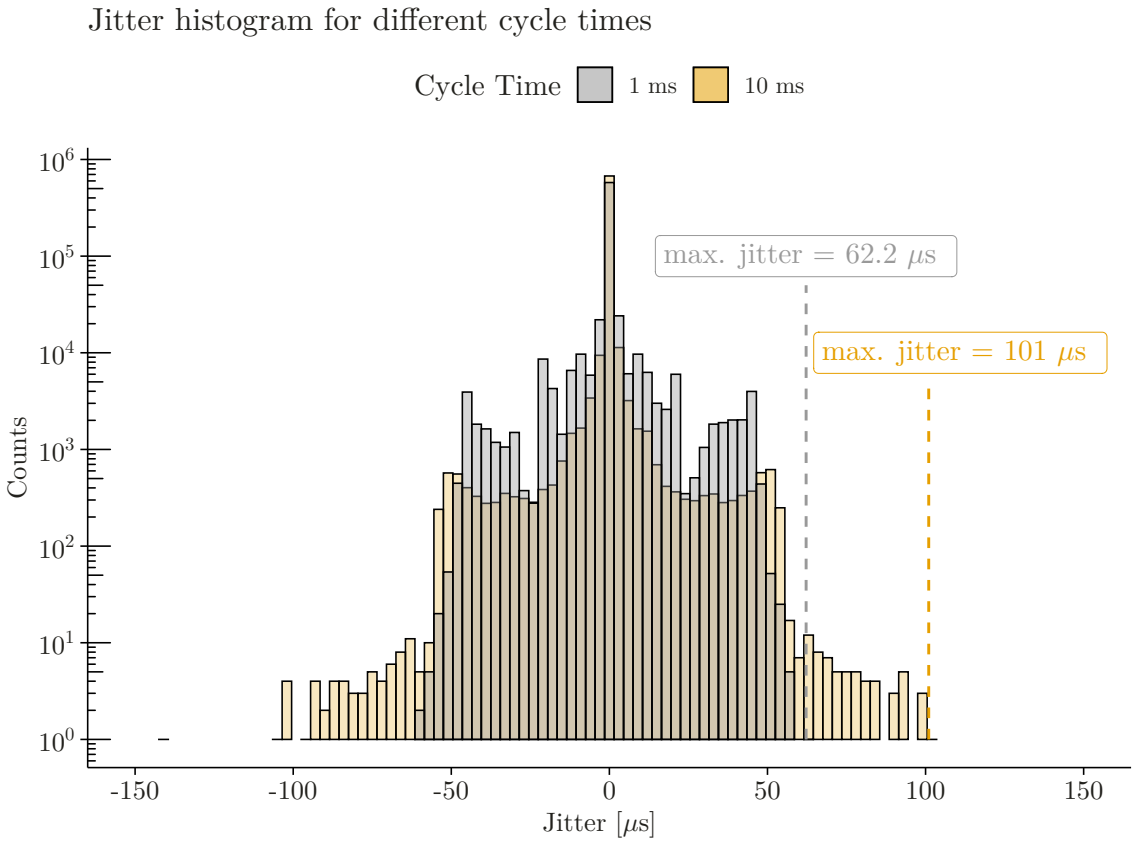


Figure 5.20: Comparison of the jitter histograms for different cycle times

Table 5.7: Numeric results for the cycle time comparison

Cycle Time	Max	% of Cycle Time	Mean (absolute)	Min	Standard Deviation
1 ms	62.22 μs	6.22 %	3.31 μs	-61.36 μs	8.80μs
10 ms	101 μs	1.01 %	0.97 μs	-141.6 μs	4.79 μs

5.6.4 micro-ROS performance

As stated in the introduction of section 5.6, the micro-ROS nodes' behaviour and performance are only examined marginally. Nevertheless, micro-ROS is an exciting tool for communication with hardware inputs and outputs.

The tests are conducted in the same way as for the standard ROS 2 node. The cycle time ten milliseconds, and the test duration is two hours. The evaluation is again done on the notebook. Hence, this time, the time is logged when the messages arrive at their destination, not the time the publisher sends them. This is in contrast to the tests above. This is done for two reasons: First, it is easier to save the logged data to a csv-file on a notebook than on a microcontroller. Second, for the *controller* node, which computes the

actuating variable, it matters when the messages arrive and not when they are sent.
The jitter histogram in figure 5.21 shows a rather lousy result compared to the results above. The main reason for this high jitter can be expected in the way the communication between the micro-ROS node and the standard ROS 2 node is established. This is done by the so-called micro-ROS agent. This ROS 2 package builds the bridge between the DDS network and the micro-ROS nodes. It is provided by micro-ROS and in this case, this package was used as it is without any changes. Therefore, the micro-ROS agent that handles the incoming messages and forwards them to the *controller* node is neither treated with high priority nor does it run on the isolated core. To get a performance similar to the tests above, the micro-ROS agent must be analysed and adapted with the settings presented in sections 5.6.1 wherever possible. However, this is not within the scope of this thesis.

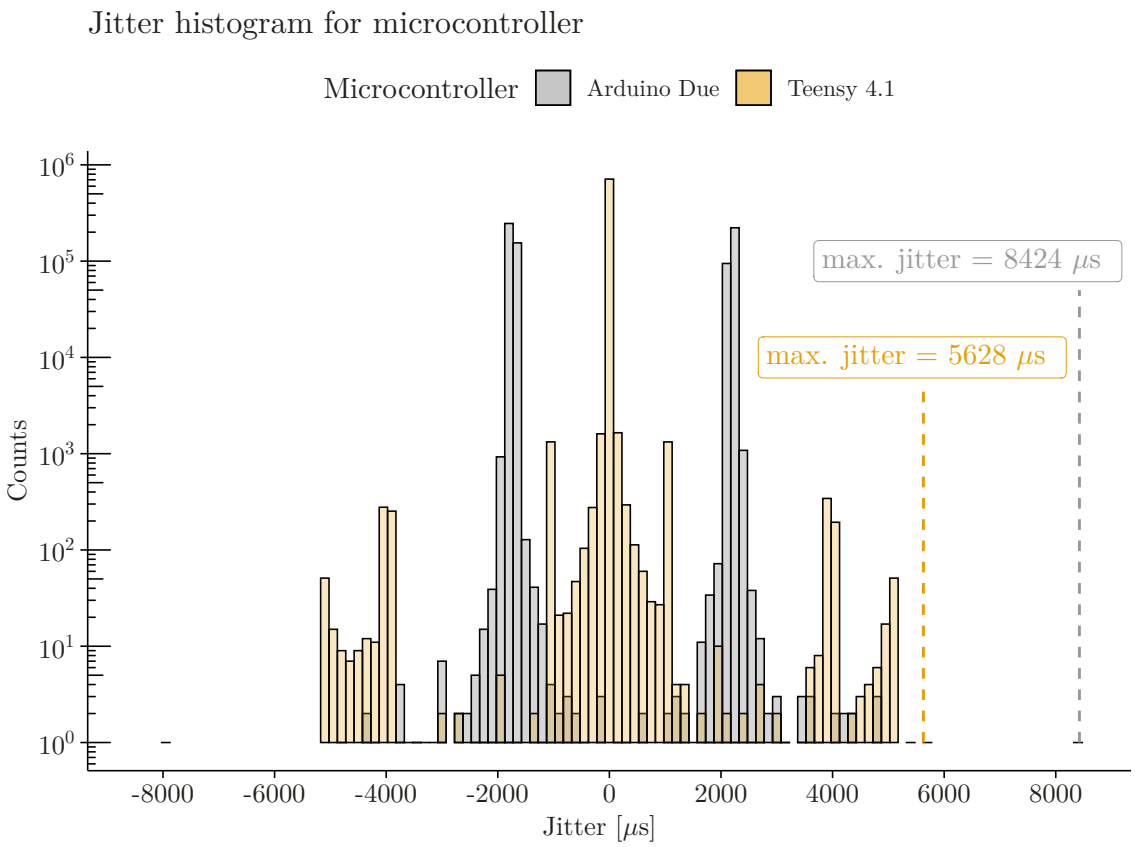


Figure 5.21: Jitter histogram for the incoming messages of the two microcontrollers

To get an idea what is possible with micro-ROS on a microcontroller, the test is repeated, but this time, the timestamps are logged on the microcontroller itself (publisher side) and then sent to the notebook to save it. This test aims to show the possible performance of micro-ROS without the delays added by the micro-ROS agent.
The jitter histogram for the on-board logging on the microcontroller shows good performance for both. Especially the Teensy 4.1 exhibits very low jitter with only four microseconds at maximum. This supports the statement from section 5.4.2 that the Teensy 4.1 is a more powerful microcontroller than the Arduino Due.

However, both microcontrollers show superior performance to the notebook, which could be expected. While the notebook has to run an operating system and handle different tasks at the same time, a microcontroller is running only one single program. Therefore all the resources are available only for this task, and no other task can disturb it.

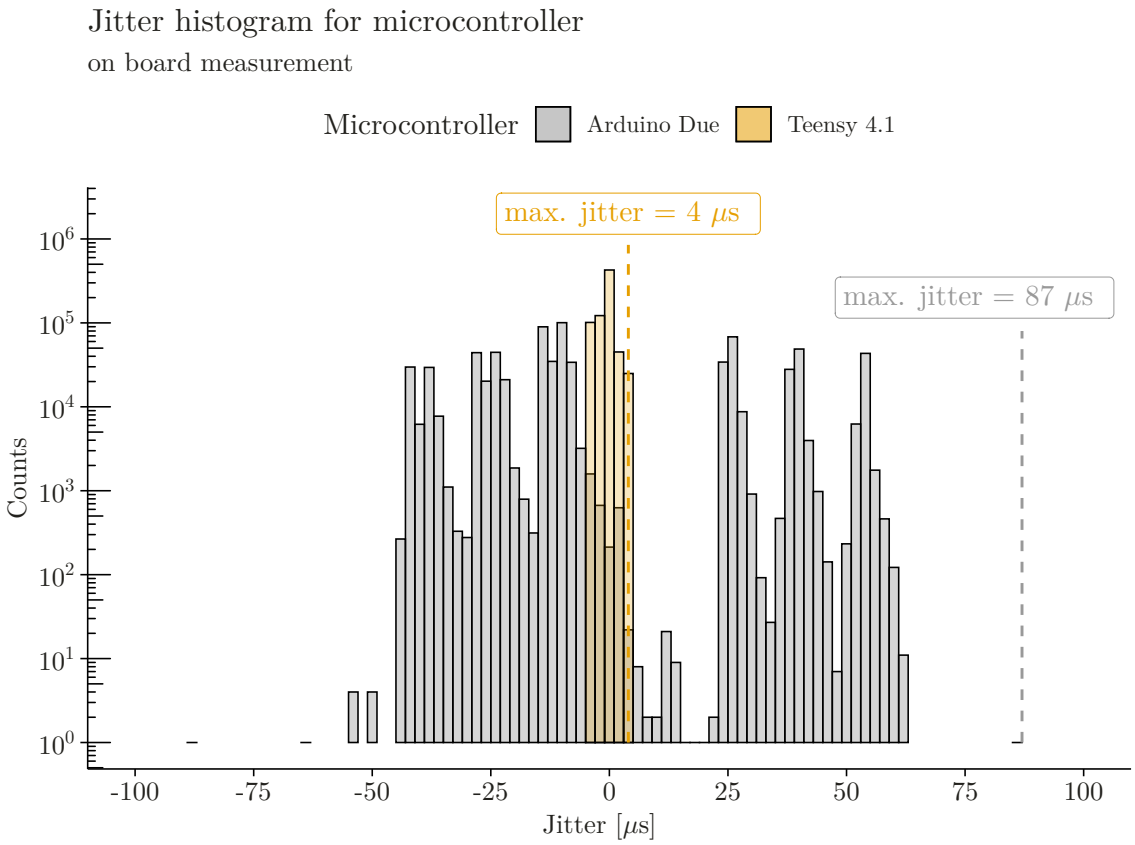


Figure 5.22: Jitter histogram for the microcontroller on the publisher side

Likely, the performance with an optimised micro-ROS agent lies somewhere in between the two cases showed in this section.

5.6.5 PLC Comparison

In the last test scenario, the jitter of a ROS 2 node is compared to a PLC. PLCs are widely used for industrial applications as for example factory automation machineries or computerised numerical control machinery. They are considered hard real-time save. The goal of this test is to find out if ROS 2, with its improvements, can be an alternative for using PLCs.

The structure of a PLC program is entirely different from a ROS 2 application. While in ROS 2, the tasks are separated into different nodes, in a PLC program, the whole task is in one main program, executed in a cyclic manner. Therefore, the controller for the inverted pendulum is rewritten for the PLC. At the start of each cycle, the sensor data is read. Then the actuating variable is calculated based on the sensor information, and at the end of each cycle, the actuating variable is set on the hardware output. As a limitation, PLCs

can only use dedicated input/output terminals. In the case of Beckhoff PLCs, they do not offer an analogue voltage output between -24 V and +24 V with a sufficiently high current supply to drive a DC-Motor. Furthermore, there is no PWM terminal available to regulate the voltage. Hence, in this setup, the inverted pendulum could not be balanced by the PLC. For that purpose, the dynamical model for the control system would have to be adapted. Because to measure the jitter, the pendulum does not need to be balanced, the test has been conducted with the pendulum at rest in order to save time. Nevertheless, the PLC reads the sensor information, and the actuating variable is computed. The only step skipped is that the actuating variable is not set on a hardware output.
In table 5.8, the specifications of the PLC and the notebook are listed. The notebook specifications are the same as in table 5.4 but are listed here again for easier comparison. From there, one can see that the notebook generally has better specifications than the PLC. Beckhoff states that the PLC used for this comparison is in the «P60 Mid Performance» class. This means that there exists both more and less performant PLCs.

Table 5.8: Comparison of the PLC and the notebook specifications

	PLC	**Notebook**
Type	CX2030	Dell Latitude 5580
CPU model	Intel Core i7-2610UE	Intel Core i7-7820HQ
CPU freq.	1.50 GHz	2.90 GHz
Nr. Cores	2	4
RAM	2GB	16 GB

The result of the two-hour test for the PLC shows that the maximal jitter that occurred is only half the magnitude of the maximal jitter for the ROS 2 node. The Same applies to the standard deviation of the jitter. Only the mean absolute jitter is better for the ROS 2 node. However, in real-time applications, not the mean but the worst-case performance is of interest. Accordingly, ROS 2 can not keep up with a PLC even though the specification of the PLC are slightly worse than the ones of the notebook. In addition, every step in the examined PLC cycle is included, from acquiring sensor information to the computation of the actuating variable. In ROS 2, due to its modularity, these steps are performed in separated nodes which all exhibit jitter.

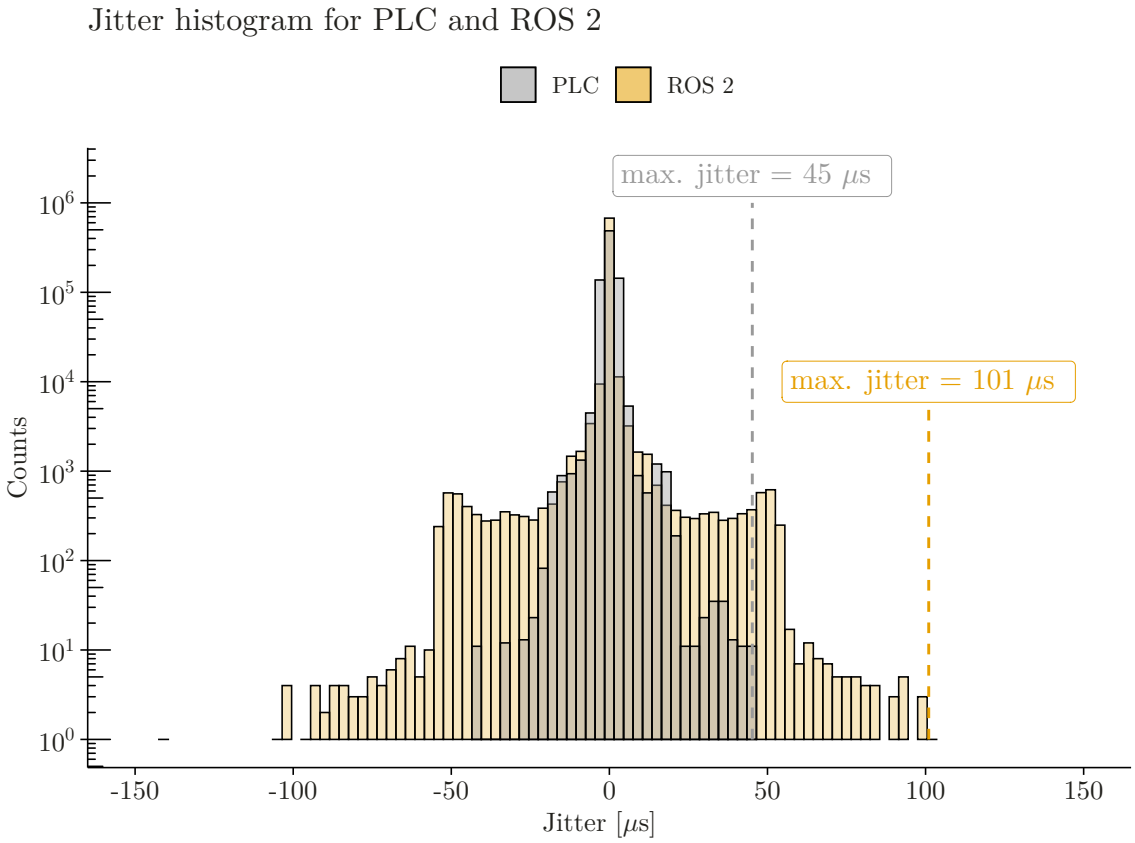


Figure 5.23: Jitter histograms for the execution of the PLC program and the ROS 2 node

Table 5.9: Numeric results for the PLC and the ROS 2 jitter

	Max	**Mean (absolute)**	**Min**	**Standard Deviation**
PLC	45 μs	1.43 μs	-42 μs	2.25 μs
ROS 2	101 μs	0.97 μs	-141.6 μs	4.79 μs

5.7 Discussion

The first series of experiments could show the importance of having the proper settings in place. Only then ROS 2 is able to limit jitter to an acceptable level for mobile robotics applications. This, of course, includes that ROS 2 runs on an operating system that allows for preemption. Especially important is to set the right priorities for each node and to have one or more CPU cores reserved only for the real-time critical tasks. But since all the tested settings had a positive influence, the lowest jitter can be expected when all the settings are turned on. This setup was tested in a long-term test to get a statistically more meaningful result. Three identical tests with a total test duration of ten hours showed that the maximal expected jitter is around 100

μs, about one per cent of the update rate. This result is comparable to other research on this topic, e.g. in [26].
The question if this is suitable for real-time applications can not be answered in general. As it is known from section 2.1, real-time systems must be deterministic and hold deadlines. However, the deadlines depend on the applications. In any case, ROS 2 is not hard real-time capable as long as it runs on a RT_PREEMPT kernel because this kernel is only claimed to be soft real-time capable [27]. This makes the system as a whole less suitable for safety-critical applications and applications with the highest timing constraints, such as motion control tasks in CNC machines.
For applications which are not safety-critical but still have some requirements regarding timing to maintain, for example, quality (soft or firm real-time), ROS 2 shows the potential to be an alternative to existing systems after these first tests. For a more detailed analysis, the second set of tests is presented in section 6.5. However, what is essential is that in order to get a real-time capable system (whether it is a soft or hard real-time system), every single component of the system must fulfil these requirements. It is not enough to focus only on the settings of ROS 2. If additional hard- and software is in use (e.g. micro-ROS) it must be ensured that these components do not violate the real-time requirements.

In the last experiment, the jitter of a ROS 2 publisher is compared to a PLC, which is one of the state of the art component for hard real-time systems. Comparing these two systems is not straightforward because they follow different paradigms. While ROS follows a modular and distributed architecture with different, independent nodes which run in parallel, a PLC executes only one main program periodically. The two systems are compared by the jitter they exhibit. For ROS 2 this is the jitter of only one executed function. For the PLC it is the jitter for the execution of the main program which includes all the tasks of the PLC. Even though the hardware of the PLC is less performant than the notebook, which runs ROS 2, it shows clearly superior performance. Furthermore, in this jitter, the execution of the whole program is included and not only a single function. Also, from the perspective of latency, which is the time it takes for a message to travel from A to B, the PLC paradigm seems to be better suited for the highest timing demands. If, for example, sensor information has to travel over several nodes, with each node, additional latency and jitter are added.
Therefore, applications with hard real-time constraints and the highest requirements regarding timing and jitter ROS 2 is currently unable to replace a PLC.

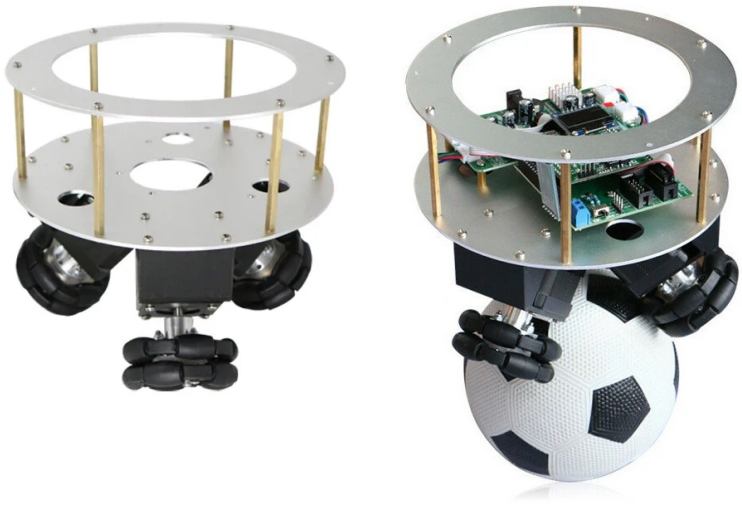
Chapter 6

Experimental Setup: Ballbot

To further evaluate ROS 2, a second experimental setup is elaborated. The second setup shall be a mobile robot because this is one of the most common areas of application of ROS 2. Furthermore, it should have more degrees of freedom than the first experimental setup. The chosen setup which fits these requirements is a ballbot. A ballbot is a robot that is balanced on a single sphere. It can be seen as an advancement of the inverted pendulum introduced in chapter 5 to more degrees of freedom which is one of the main reasons for choosing this type of mobile robot. Like the inverted pendulum, it is an unstable system and therefore well suited to show the performance of a real-time system.
The first ballbots were developed in the first decade of the new millennium. Well-known examples are [28], [29] and [30]. In the meantime, many more ballbots arose; therefore, this mobile robot's design and control system is a well-studied problem with lots of resources available. Nevertheless, due to its five degrees of freedom, it is a challenging task to balance a ballbot.
In this chapter, the design of the ballbot, as well as the implementation of the control system and the software, are explained. Then a set of performance tests and their results are presented.

6.1 Make or Buy

Because of the project's advanced time, the first decision was to develop and build the ballbot from scratch or buy an existing «ballbot kit». This section elaborates the pros and cons of the two approaches and the corresponding risks.
The «ballbot kit» considered is shown in figure 6.1. It can be purchased from different vendors. The kit is available in three different versions, which differ in electronics. The basis kit does not contain any electronics. It only comprises the mechanical parts and the motors. The two other versions come with the electronics needed to control the ballbot (microcontroller, motor driver, power adapter etc.). They differ in the type of microcontroller used. One uses an Arduino Mega, and the other uses an STM32. However, from chapter 5, it is known that for microcontrollers, micro-ROS is needed. Since ROS 2 is the focus of this thesis and not micro-ROS, neither of the two versions with electronic components come into consideration.



(a) Basis version (b) Version with electronics

Figure 6.1: «Ballbot kit» for sale in different versions

In order to have a basis for the decision of making or buying a ballbot chassis, pro and contra points for the bought option are listed below:

Pros:

- Parts are already dimensioned and tested in this setup
- Price
- Time savings

Cons:

- All electronic parts and sensors must be chosen and placed on the chassis anyway
- No CAD model available (for placement of the additional hardware or to find important parameters as e.g. center of mass)
- No experience with stepper motors
- Forces applied on the omni wheels are introduced directly to the motor shaft
- Incomplete documentation
- Unknown quality

Similar to the inverted pendulum in chapter 5, a risk assessment has been done for the development of a ballbot. The risks elaborated are very similar to those for the inverted pendulum, with a slightly higher probability of occurrence due to the higher complexity of the experimental setup. The complete risk assessment can be found in appendix C.1.

Taking all these points into account, there is no clear advantage when buying the ballbot chassis. On the contrary, the development of an own chassis offers more flexibility, and this is why the chassis of the ballbot is self-developed.

6.2 Ballbot Control System

Even though the same type of feedback control loop is used as for the inverted pendulum (linear-quadratic regulator, see section 5.2), the description of the dynamical system is far more complex due to the many degrees of freedom. The modelling of the system dynamics of such a ball balancing robot has been studied extensively, and many publications exist on that topic. Because the development or improvement of the control system is not the main goal of this thesis, in this section, only the main ideas of the ballbot control system are explained. The control system has been taken from [30], and all the details can be found there. The dynamical model has been elaborated using the Lagrangian dynamic formulation. This approach is based on the energies of the system. This comprises potential and kinetic energies for all the parts of the system and formulates the following Lagrange equation:

$$\frac{d}{dt}\left(\frac{\partial T}{\partial \dot{\overrightarrow{q}}}\right)^T - \left(\frac{\partial T}{\partial \overrightarrow{q}}\right)^T + \left(\frac{\partial V}{\partial \overrightarrow{q}}\right)^T - f_{NP} = 0 \tag{6.1}$$

Where T is the sum of all the kinetic energies and V is the sum of all the potential energies. $\overrightarrow{q}$ are the so-called minimal variables which correspond to the degrees of freedom of the system. In this case this is:

$$\overrightarrow{q} = [\vartheta_x, \vartheta_y, \vartheta_z, \varphi_x, \varphi_y] \tag{6.2}$$

The non-potential forces f_{NP} are the forces acting on the system in order to stabilise it. In this case, these are the three torques acting on the wheels of the ballbot.
Solving equation 6.1 then leads to the equations of motion for the system. This is done by means of a Matlab script and will not be further explained here because this would exceed the scope of this thesis. The parameters of the ballbot needed for the dynamical system are are taken from the CAD model and are also apparent in the Matlab script. The Matlab script can be found in the electronic appendix under «20_Experimental_Setup_Ballbot/30_Code/ ballbot_control_system-main». The resulting feedback parameters **K** are in the form of a 3 x 10 matrix, which shows the increased complexity compared to the inverted pendulum showcase. Furthermore, the fastest pole of the closed loop system is located at -6.9 Hz. Therefore, the closed loop system is more dynamical compared to the inverted pendulum.

6.3 Design of the Ballbot

In this section, the development of the ballbot is introduced. This comprises the mechanical part, the electronics as well as the software.

6.3.1 Mechanical Design

Due to the demanding schedule, no simulation has been conducted this time to find appropriate dimensions for all the parts of the ballbot. In order to minimise the risk of having wrong dimensioned parts, which could prevent the ballbot from balancing, the mechanical design is firmly based on existing examples of ballbots. Therefore, the main components of the developed ballbot are the same as for all the other ballbots (the components can also be found in figure 6.2, data sheets are in the electronic appendix in folder «*20_Experimental_Setup_Ballbot/10_Data_Sheets/*»).

Body The body of the ballbot consists of two circular sheet metals. They are mounted together by distance bolts. The two sheet metals serve as mounting points for various other components. The three drive units are mounted on the bottom of the lower sheet metal. The batteries are placed and mounted on the top side of the lower sheet metal using hook-and-loop fasteners for flexible positioning. On the upper sheet metal, all the electronic components are mounted. This includes the Raspberry Pi 4, the IMU, the PWM extension as well as the H-bridges.

Drive unit The drive unit is responsible for balancing the ballbot. Therefore, several components are needed:

Motor + Gear box The central part of the drive unit is the brushed DC-Motor with a gear box. The gear box has a reduction ratio of 75:1 and ensures sufficient torque. The motor is equipped with an encoder needed for the control system. The encoder has a relatively low resolution of 371 counts per revolution. However, since the encoder is mounted on the motor side and the gear box reduces the motor shaft speed, one turn of the shaft has a resolution of 371 cpr $\cdot\, 75 = 27'825$ counts, which is reasonably high.

Omni wheel Because the ballbot has three rotational degrees of freedom, the wheels that drive the ballbot must allow for relative movements about all three axes between the ball and the ballbot. For a typical three-wheel arrangement, one can not use standard wheels because they would block the relative motion. Therefore, one has to use so-called omni wheels. These are wheels with little rolls on the circumference, allowing movements perpendicular to the wheel.

Wheel support The wheel support ensures that the forces acting on the wheels are not transmitted to the motor shaft. The omni wheels are mounted on a separate shaft supported by two ball bearings. The end of this shaft is then connected to the gear box shaft by a coupling which can compensate for angular and parallel offset.

Battery For mobile robots it is crucial to have the voltage supply on board to be independent of a power supply. In this case, not one battery is used but three. It is easier to place three small batteries on the lower sheet metal between the motors than one large battery. By connecting the three batteries in parallel, the capacity can be increased.

Raspberry Pi 4b For this setup, it is not suitable to use a notebook to control the system. The processor must be on the ballbot itself. Therefore, a Raspberry Pi 4b is used, which is a single-board computer in the size of a credit card. It is powerful enough to run Ubuntu on it, which is the target platform for ROS 2. Furthermore, it offers a wide variety of hardware inputs and outputs as well as possibilities for serial communication.

IMU The inertial measurement unit measures the deflection angle and the angular velocity of the ballbot body. An IMU is a combination of different sensors. In this case, this comprises an accelerometer, a gyroscope and a geomagnetic sensor. With a fusion algorithm these sensor informations are combined and yields, for example, the absolute orientation. The IMU used is the BNO055 from Bosch.

PWM extension PWM signals are used to drive the DC motors. The Raspberry Pi 4 has two independent PWM outputs. However, the ballbot has three drive units which all must be regulated individually.

Hence an extension board must be used, which offers 16 PWM channels. The PWM extension used is the PCA9685.

H-bridge The H-bridges are used to amplify the PWM signal of the extension board. One H-bridge per motor is in use. The H-bridge used is the DRV8871 which has a current limiter on board that protects the H-bridge in case the motor draws too much power.

Teensy 4.1 A Teensy 4.1 microcontroller is used to read the signals of the three encoders. Due to the high number of counts per revolution of the gear box shaft, it turned out that the Raspberry Pi 4b is not able to determine the direction of turning in a reliable way.

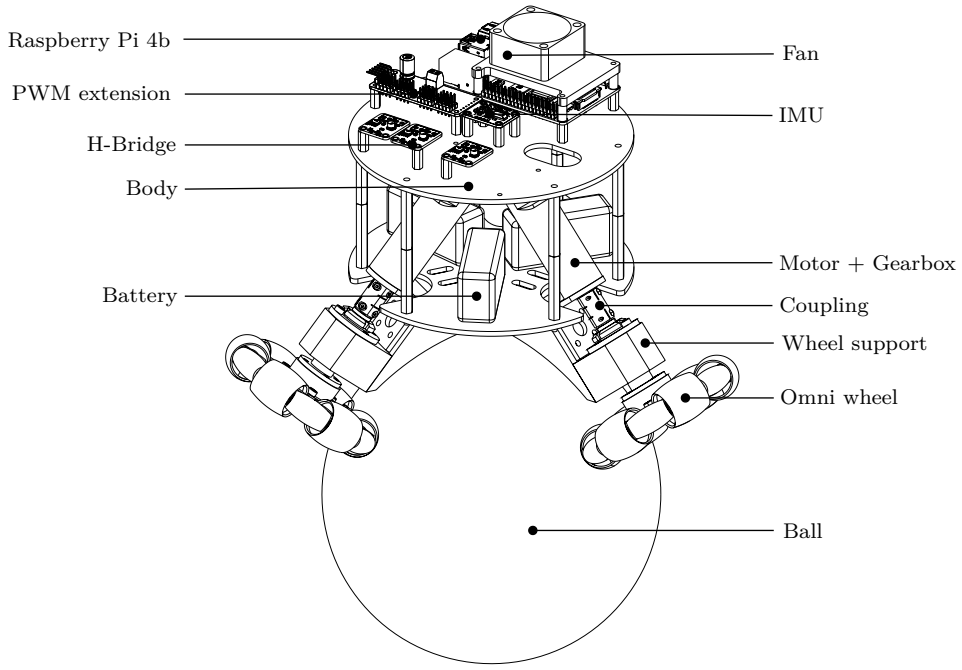


Figure 6.2: Overview of the final ballbot setup

6.3.2 Electrical Design

The electrical components described in section 6.3.1 must be connected in order to get a working system. The electric diagram, which shows all the connections between the different components, is shown in figure 6.3. For simplicity reasons, the diagram only comprises one instead of three motors, batteries and H-bridges. However, the missing components are connected analogue.
The tasks performed by the various components are already described in section 6.3.1. This section focuses more on the power supply and the communication between the components.
The core of the electrical components is the Raspberry Pi 4b. The ROS 2 nodes will run on it; therefore, it contains all the logic needed to stabilise the ballbot. To do so, it needs, on the one hand, all the necessary sensor information (IMU and encoders), and on the other hand, it must be able to set the actuating variable.

Starting on the sensor side, the IMU gets its 5 V power supply from the Raspberry Pi 4b. The IMU provides different information, e.g. the angular velocity about all three axes or a gravity vector. A communication protocol is needed to get all this information from the IMU to the Raspberry Pi 4b. One supported by both the Raspberry Pi 4b and many of the compatible hardware is I^2C. It is a serial data bus based on two wires, the data line (SDA) and the clock line (SCL). This bus uses the primary/secondary bus access procedure. I^2C is used to transfer the IMU information to the Raspberry Pi.
The angular velocity of the motors is measured by optical encoders, which generate a square wave signal when turning. In order to get not only the angular velocity but also the direction, two channels are needed. If the edge of the square wave signal is rising on the first channel, one has to determine if the signal on the second channel, which has a phase offset, is high or low. The first implementation of this algorithm on the Raspberry Pi 4b has shown that it is not accurate enough to determine the direction of turning. Despite turning in the same direction, the output was once a forward turning and the next time a backwards turning motor shaft. It is likely that the tested libraries (pigpio and wiringpi) to access the hardware pins of the Raspberry Pi 4b were not fast enough for the high frequency of the square wave signal. Therefore, a Teensy 4.1 microcontroller is used, which can accurately determine the turning direction for three motors at high speeds. However, this time the Teensy 4.1 does not run a micro-ROS node but instead uses the USB interface to send the angular velocities to the Raspberry Pi.
On the actuation side, a PWM extension board is used. The power supply is coming from the Raspberry Pi 4b. The desired duty cycles to drive the motors are sent over the same data bus used by the IMU. The duty cycle is then set on the output pins connected to the H-bridge, which amplifies the PWM signal. Therefore the batteries are connected to the H-bridges. The amplified PWM signal is then directed to the motors.

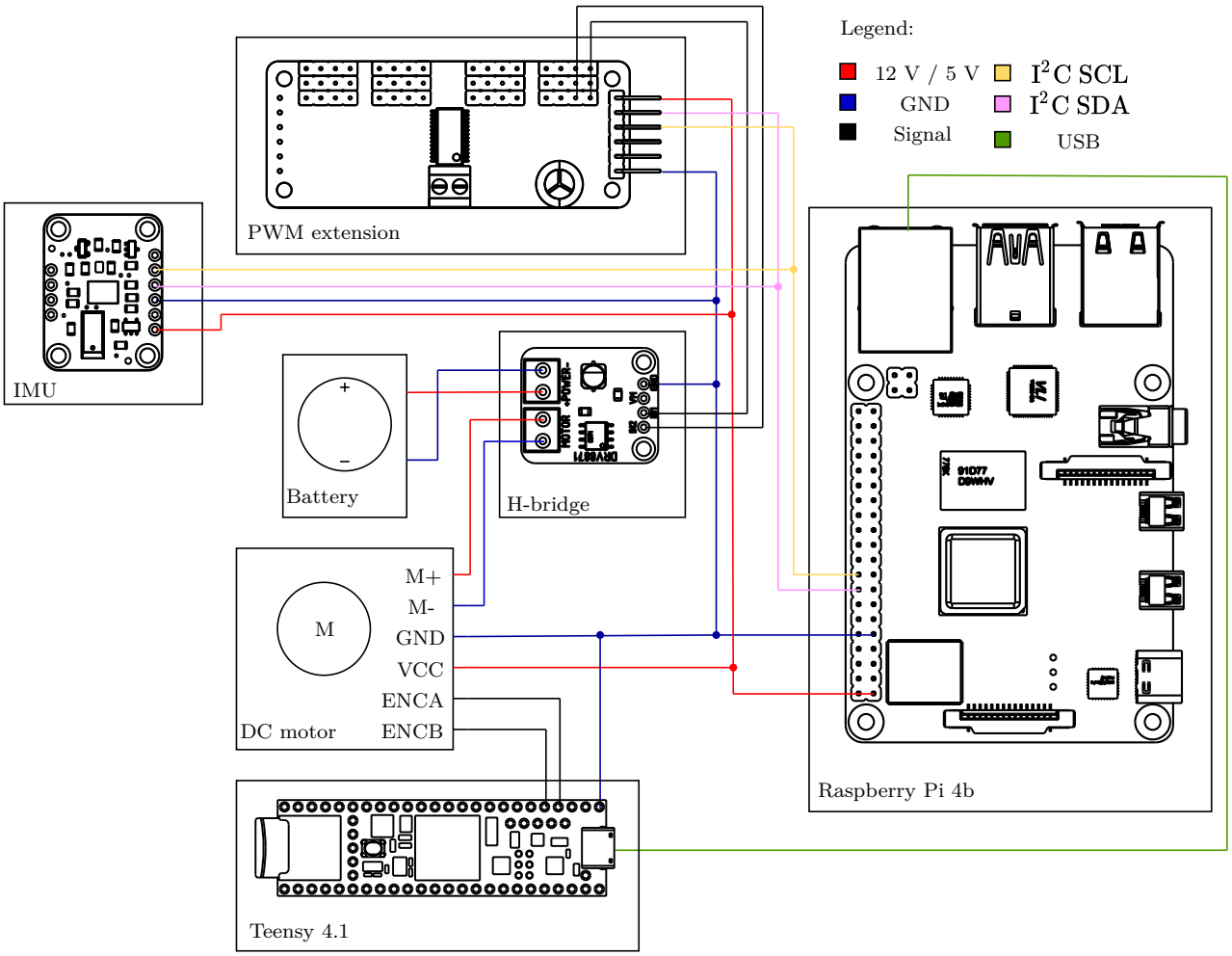


Figure 6.3: Simplified electrical diagram of the ballbot with only one motor, battery and H-bridge

6.3.3 ROS 2 Architecture

This section covers the implementation of the various ROS 2 nodes needed in order to control the ballbot. A graphical representation of the ROS 2 network can be found in figure 6.4. All the nodes run on the Raspberry Pi 4b (the setup of the Raspberry Pi 4b is explained in appendix C.2). The *controller* node is the core node where all the information for the control system comes together. This includes the angle and the angular velocity of the ballbot, which are transmitted via the *gravity* and the *imu/data* topics, respectively. On the other hand, the angular velocity of the motor shaft is sent to the *controller* node via the *motor_state* topic. The state of the ball is passed over via the *odometry* topic.

However, the information must be acquired before the various information can be sent to the *controller* node. This is the duty of the different nodes. The *bno055_sensor_node* is responsible for reading the IMU information via the I^2C bus. This node is based on an existing ROS 2 package. The package has been adapted to meet real-time requirements. Furthermore, the publication of data that is not needed has been removed. The other node that acquires sensor information is the *encoder_reader*. This node reads the incoming information on the USB port of the Raspberry Pi. This comprises the current angle as well as the

angular velocity for each of the three motor shafts. The data is sent in the form of a string data type. The *encoder_reader* node splits the string into the three angles and angular velocities and extracts the numbers out of these substrings. Both the angle and the angular velocity for each motor is then forwarded not only to the *controller* node but also to the *odometry* node. The latter converts the incoming information of each motor to the state of the ball. The state of the ball consists of the angle and the angular velocity about the x- and y-axis. The conversion is done by an equation provided by [31].
With all this information, the *controller* node can compute the actuating variables by means of the linear-quadratic regulator implemented on it. The computed actuating variable is then converted to a duty cycle for the H-bridges. This conversion not only depends on the motor constants but also on the current angular velocity of the motors (see equation 5.15). The computed duty cycle is then sent to the *motor_driver* node. This node is then connected to the hardware through the I^2C bus. The node can set the duty cycle on each of the 16 pins of the PWM extension. An existing C++ library is used and wrapped into a ROS 2 node for the communication to the extension board.

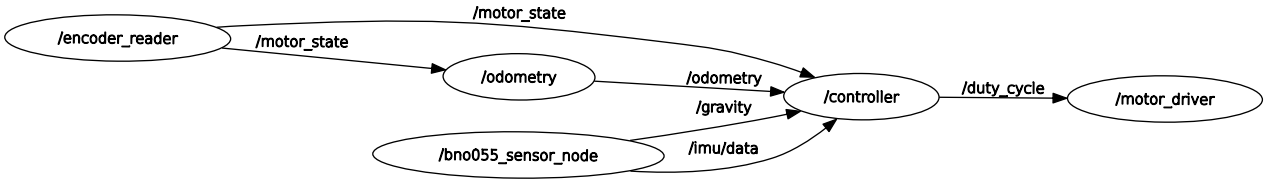


Figure 6.4: ROS 2 graph of the ballbot

The ROS 2 code can be found in the electronic appendix: «20_Experimental_Setup_Ballbot/30_Code/ballbot-main»

6.4 Initialisation Phase

During the initialisation phase of the ballbot, different difficulties have arisen. This section focuses on these difficulties and shows the counteractions taken.

After some difficulties with the linearity of the output signal of the H-bridges in the first experimental setup, the same H-bridges were intended for the ballbot. The goal was to use the gained knowledge during the initialisation phase of the first experimental setup. However, when the motor shaft is blocked and the ballbot is at a too high deflection angle, the motor draws too much current for the L298n H-bridge. Therefore another H-bridge has to be chosen. The DRV8871 H-bridge can handle higher currents and is able to limit the current. That prevents the H-bridge from being damaged.

Another problem that prevented the ballbot from balancing is the slippage between the omni wheels and the ball. According to [30], one of the assumptions for the dynamical model is no slippage between the ball and the omni wheels. The presence of slippage invalidates, therefore, the dynamical model. One possibility to minimize slippage to an acceptable level is to increase the friction coefficient between the ball and the omni wheels. Several liquid rubbers have been tested qualitatively on the ball, and the one with the best result has been chosen. It is a rubber used for non-slip socks for children, which illustrates that this is a reasonable

choice. Hence, the ball is coated with the rubber to increase friction between the ball and the omni wheels.

With the slippage reduced to a minimum, the feedback parameters **K** for the controller shall be fine-tuned until the ballbot is able to balance. Starting with the initial parameters calculated by means of the Maltlab script, the parameters shall be optimised. In a first step, the dynamical model and its implementation in ROS 2 is tested qualitative to make sure that there are no sign errors or similar mistakes. This is done by deflecting the ballbot around one axis and observing the reaction of the ballbot. When the angle is large enough, the ballbot starts to drive back toward the stable upright position. However, what stands out is that the ballbot is not able to react on small deflection angles. Only after a certain threshold does it starts to move. However, due to the large angle, the resulting torque must be high to straighten up the ballbot. The desired behaviour should be that the ballbot reacts already on small deflection angles and, therefore, can use lower motor torques. Analysis of the recorded data during these test runs showed that the problem does not originate from inaccurate sensor information. The IMU can also recognise small deflection angles; based on that, the duty cycle for the H-bridges is computed correctly. However, the ballbot does not start moving. From section 5.5, it is known that too high PWM frequencies can lead to a lower torque generated by the motor than expected. However, decreasing the PWM frequency did not help to solve the problem.
After an intense testing phase, a new problem was identified, which remained unrecognised until then. In some situations, not only the rolls on the circumference touch the ball but also fixed parts of the omni wheel. In these situations, the ballbot is not able to allow the relative movement between the ball and the wheel that is needed. This problem can occur because the omni wheels are double-row omni wheels with displaced rolls at the circumference. Since the geometry of the omni wheels did not allow for reworking them so that the gap between the ball and the fixed parts of the omni wheels increases, new single-row omni wheels are purchased. The advantage of single-row omni wheels is that there is no displacement of the rolls on the circumference, and therefore, there is no fixed part of the wheel that could get in touch with the ball. Another advantage of the new omni wheels is that the rolls have a structured surface, resulting in a good grip between the wheels and the ball. A rubber coat is therefore not needed any more for the new wheels. Because the single-row omni wheels have a bigger diameter, the ball must be changed to a smaller one.
Due to the delivery time of the wheels there was not enough time left in order to tune the control parameters properly. First quick tests showed improvement due to the new wheels, but without human input, the ballbot cannot stand upright for more than a few seconds. Hence, more time needs to be invested in finding appropriate control parameters. Furthermore, some data filtering might be needed to get a smoother behaviour of the ballbot.

6.5 Experimental Procedure

Similar to the inverted pendulum showcase, the ballbot is used to conduct a series of experiments to investigate the performance of ROS 2 further. Because the ballbot is not able to properly balance yet, the tests are conducted at rest. Nevertheless, all the sensor data is acquired, and the actuating variables are set, so the results are still valid. The same settings evaluated in section 5.6.1 have been used for the tests. As for the tests on the inverted pendulum, additional stress is applied to the CPUs.
The following experiments were conducted on the hardware of the ballbot:

1. **Jitter on the Raspberry Pi 4b** The first experiment is the same conducted on the inverted pendulum. The goal of this repetition is to show the potential performance decrease on hardware with limited resources compared to the first execution on the notebook.

2. **Latency** Another essential measurement to assess the system performance is latency, which is the time it takes a message to get from the sender to the receiver. In ROS terms, from the publisher to the subscriber.

3. **Callback Duration** Callbacks are these functions that are called upon certain events. In this test, the duration of the callback functions is determined with special attention on the deviation of the durations.

6.5.1 Jitter on the Raspberry Pi 4b

The jitter experiment on the Raspberry Pi 4b is done on the *controller* node. This node runs with the highest possible priority of 98. All the other nodes run with lower priorities descending from 90. These lower priorities are awarded arbitrarily. Different priorities are chosen because it does not make sense to have the same priorities for all nodes. Having the same priorities for all the nodes would have the same effect as having no priority for all the nodes.
For an easier contextualisation of the results presented in this section, the specification of the Raspberry Pi 4b and the notebook used for the inverted pendulum are compared in table 6.1. From there, one can see that the Raspberry Pi 4b is less powerful than the notebook.

Table 6.1: Comparison of the Raspberry Pi 4b and the notebook specifications

	Raspberry Pi	**Notebook**
Type	Raspberry Pi 4b	Dell Latitude 5580
CPU model	Cortex-A72	Intel Core i7-7820HQ
CPU Architecture	aarch64	x86_64
CPU freq.	1.8 GHz	2.90 GHz
Nr. Cores	4	4
RAM	4 GB	16 GB

The resulting jitter histogram in figure 6.5 shows what could be expected from the specifications comparison in table 6.1: The maximal jitter for a node running on a Raspberry Pi 4b is far higher than on more powerful hardware (max. jitter on the notebook is 101 μs).
To verify this result, the inverted pendulum demonstration from section 4.2 (provided by the ROS 2 installation) has been executed on the Raspberry Pi 4b. The maximal measured jitter is 276.5 μs, about 6.5 times higher than on the notebook (43 μs). For the *controller* node of the ballbot on the Raspberry Pi, the jitter increased by a factor of about 8.5 compared to the *controller* node of the real inverted pendulum on the notebook. Both factors are of the same magnitude; therefore, it could be verified that the main reason for the increased jitter is the limited hardware.

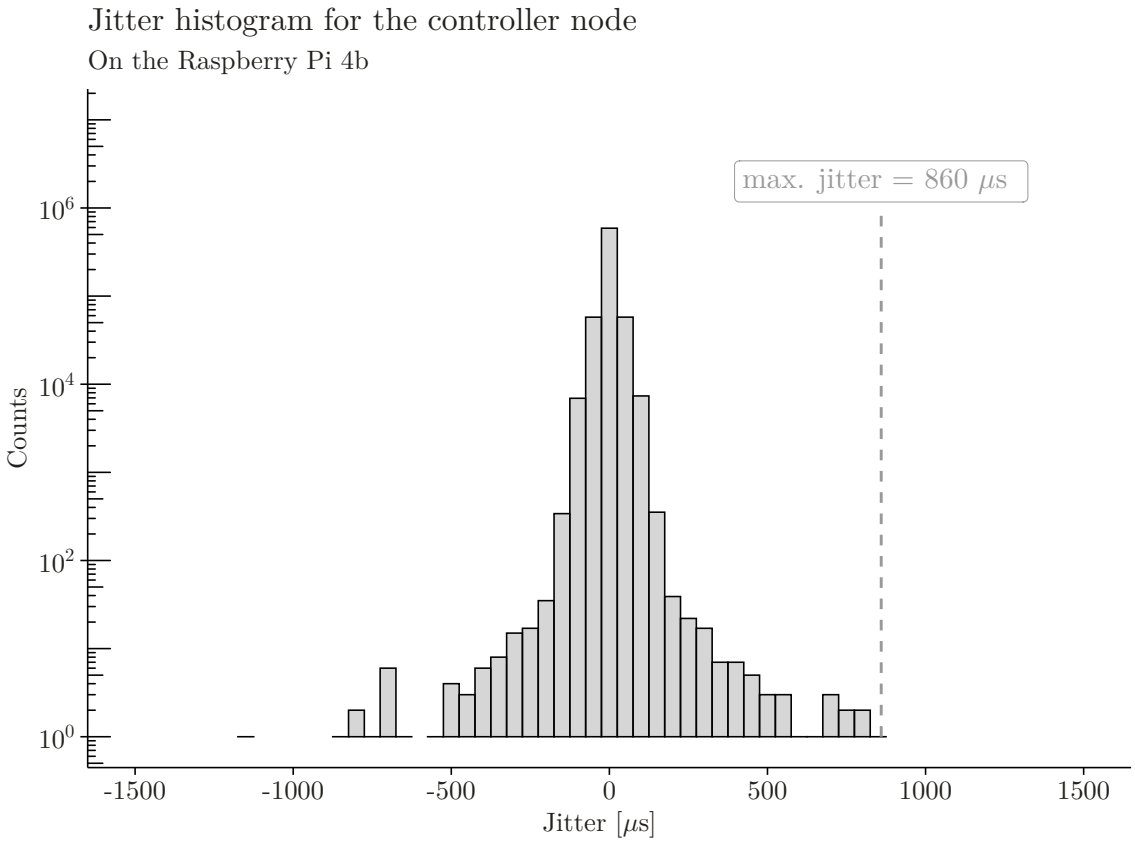


Figure 6.5: Jitter histogram for the controller node on the Raspberry Pi 4b

Table 6.2: Numeric result for the jitter analysis on the Raspberry Pi

	Max	**Mean (absolute)**	**Min**	**Standard Deviation**
Jitter	860 μs	15.88 μs	-1152 μs	25.08 μs

6.5.2 Latency

So far, the jitter has been used to assess the performance of ROS 2. This is the deviation between the time a message should be published and the actual publishing time. Apart from that, there are more indicators of interest to get an overview of the real-time performance. One of these indicators is the latency of the messages, i.e. how long it takes until the subscribing node receives a message. One common way to determine this is to measure the round trip latency. This means that the time is measured, that it takes to send a message from one node to another and then send it back directly to the node of origin. This has the advantage that all the time measurement actions can be conducted in the same node. If the round trip latency is divided by two, one can get an estimation for the latency of the one-way latency.
This experiment is conducted on an adapted subsystem of the ballbot. It only uses the *odometry* and the

controller node. Furthermore, the *controller* node is extended in order to return the incoming message to the *odometry* node directly. The corresponding rosgraph can be found in figure 6.6. These two nodes have been chosen because these are the two which do not depend on other communication protocols than on the ones of ROS 2. Furthermore, there is no connection to hardware for these nodes. This enables to conduct this test not only on the Raspberry Pi 4b but also on the notebook.

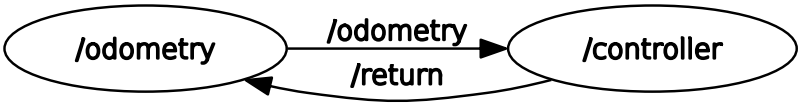


Figure 6.6: rosgraph for the adapted subsystem of the ballbot

The test to assess the round trip latency was performed in a two-hour test with a cycle time of ten milliseconds. Both nodes run at the highest possible priority of 98. The previous test in section 6.5.1 shows that the hardware platform on which the test is executed dramatically influences the performance outcome. Therefore, the round trip latency test has been performed on both platforms used so far in this thesis. The notebook as well as the Raspberry Pi 4b.
The analysis of the results shown in figure 6.7 confirms this assumption. Similar to the jitter tests, the maximal measured latency for the Raspberry Pi 4b is about seven times higher than for the notebook. However, not only the maximal latency but also the other indicators are by far better for the notebook (see table 6.3). However, the main problem for the Raspberry Pi 4b is not the high mean latency. If this is known, it can be considered when designing the system. But especially the high standard deviation on the Raspberry Pi 4b makes the system less deterministic and, therefore, less suited for time-critical applications than the notebook.

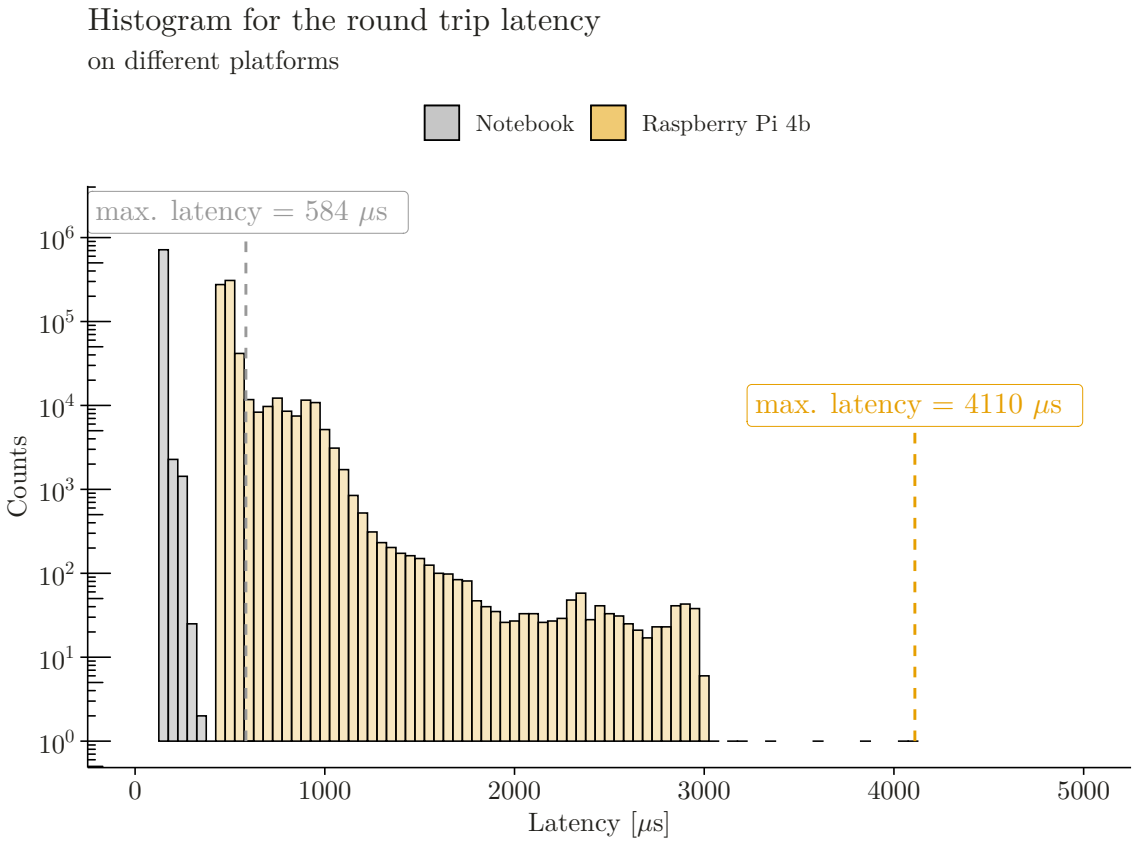


Figure 6.7: Histogram for the round trip latency on the notebook and the Raspberry Pi 4b

Table 6.3: Numeric results for the round trip latency

	Max	**Mean**	**Min**	**Standard Deviation**
Notebook	584 μs	162 μs	159 μs	4.65 μs
Raspberry Pi 4b	4110 μs	524 μs	429 μs	146.56 μs

6.5.3 Callback Duration

Callbacks are special functions whose execution is triggered by certain events. In the case of the communication paradigm in ROS, the publisher/subscriber model, these callbacks are subscription or timer callbacks. Subscription callbacks are called whenever a message is coming in on the *topic* the callback corresponds to. Timer callbacks, on the other hand, are called cyclic depending on the timer respectively the cycle time. In ROS, timer callbacks are used to publish messages.
The time that is needed in order to execute a callback function depends clearly on the number of computations that are performed within the callback. To get a deterministic system, a low callback duration has not the highest priority. Of course, the duration of a timer callback has to be lower than the publishing period

in order to reach the demanded frequency. However, the deviation, and associated with it, the maximal duration of the callback durations is what makes a system deterministic. Therefore, similar to the jitter tests, in this test, the focus will be on the maximal callback durations as well as on the standard deviations. The test scenario presented in this section is based on the same setup as the one in the previous section. This is for comparison reasons. On the notebook, only the nodes without hardware interactions can be tested. Therefore, the time of three callbacks is measured. The *odometry publisher* first computes the current angle and angular velocity of the ball and publishes this information then on the *odometry* topic. The *odometry subscription* callback is called whenever a new message arrives at the *controller* node. This callback function takes the incoming message and returns it back to the *odometry* node. There the *return subscription* callback is called on incoming returned messages. The only thing done in this callback is to check whether the predefined number of messages is reached or if the test has to continue. The test duration is set to two hours which corresponds to 720'000 messages at a publishing rate of 100 Hz.
As already stated above, the different mean callback duration for the three different callbacks arise from varying workloads. The *return subscription* callback needs, of course, less time to check one *if*-statement than the *odometry publisher*, which needs to compute the current state of the ball and publish the message (see figure 6.8 and table 6.4). What is also noticeable is that callbacks with higher mean duration also have a higher standard deviation. No matter whether the callback runs on the notebook or the Raspberry Pi 4b. This is likely to originate from the single computation steps. If each computation step has a mean duration and a specific deviation, the more steps performed in a row, the higher the deviation from the total mean. However, generally, the standard deviations for the callback durations are relatively small. This helps to get a deterministic system that is vital to meet real-time requirements. Measurement of the callback durations also confirms what is already known from the two previous experiments: The notebook, which offers more resources than a Raspberry Pi shows superior behaviour in all four categories, maximal, mean and minimal callback durations, as well as for the standard deviation. This emphasises once more the importance of the hardware in order to meet timing requirements in ROS 2.

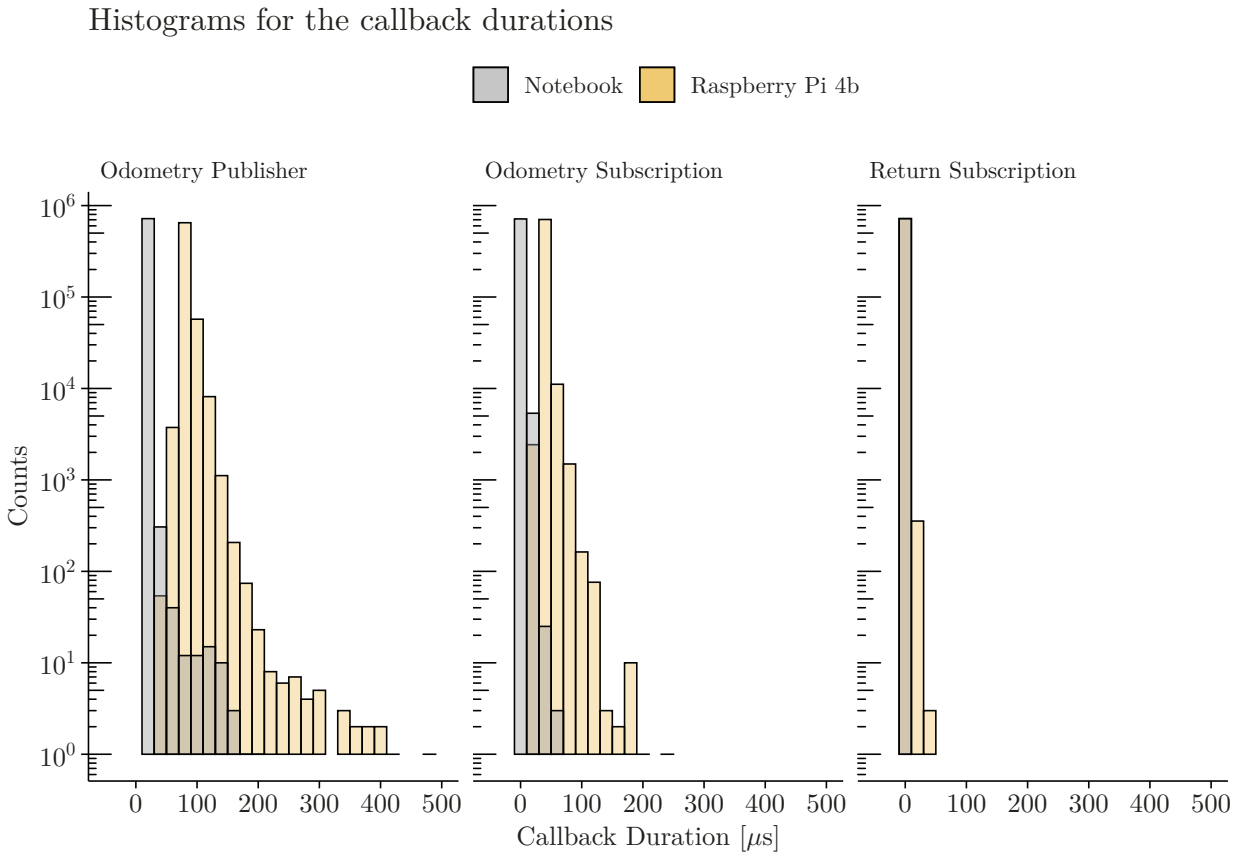


Figure 6.8: Histogram for the duration of three different callbacks

Table 6.4: Numeric results for the callback durations on the notebook and on the Raspberry Pi 4b (RPi 4)

	Max		**Mean**		**Min**		**Standard Deviation**	
	Notebook	RPi 4	Notebook	RPi 4	Notebook	RPi 4	Notebook	RPi 4
Odometry Pub.	199 μs	479 μs	17 μs	84 μs	14 μs	41 μs	1.15 μs	7.34 μs
Odometry Sub.	70 μs	238 μs	7 μs	39 μs	6 μs	19 μs	0.45 μs	4.1 μs
Return Sub.	7 μs	39 μs	0.04 μs	0.48 μs	0.03 μs	0.1 μs	0.03 μs	0.39 μs

6.6 Discussion

This second set of experiments focuses on the performance difference of ROS 2 for varying hardware. On the one hand, the Raspberry Pi 4b is used to drive the ballbot. This is a small single-board computer which offers different interfaces for communication but is limited in computational power. On the other hand, there is the notebook with fewer possibilities to interact with external hardware but more computational power

on board in exchange. The two hardware platforms are compared based on three different indicators (jitter, round trip latency and callback duration). The results show significant performance differences between the two platforms for all three test scenarios. Therefore, choosing the appropriate hardware for the task and the consequent timing requirements is crucial.
The results of the round trip latency are comparable to other publications in [12] and [26]. The former uses the 95% percentile as a measurement metric instead of the maximal latency. Therefore the results can not be compared directly. Computing the 95 % percentile for the results of this experiment leads to 166 μs and 876 μs for the notebook and the Raspberry Pi 4b, respectively, which is in the same magnitude as the results presented in the paper.
The analysis of the callbacks shows a relatively good result with low standard deviations, which is important for a deterministic system. Due to the lack of time, analysis is restricted to three callbacks. In future tests, the leftover ones could be analysed as well. Especially the callbacks communicating to hardware might be of interest. This could then be also an indicator for the determinism of the communication protocol outside of ROS 2, which was not taken into account in this thesis.

Chapter 7

Conclusion

With the growing popularity of mobile robots and their potential industrial applications, new requirements for these systems arose that were less critical under laboratory conditions. These are, among others, the ability to work in a deterministic manner and satisfy timing requirements. This is important for the reliable and safe service of robots. Therefore a completely new version of the popular Robot Operating System (ROS) has been introduced. Based on various experiments, this thesis shows what can be expected from this new version and where other solutions might be reasonable. Furthermore, a set of system settings has been elaborated in order to optimise the performance of ROS 2.

In the first set of experiments, the focus lies on the various settings that can be changed. This includes settings related to the operating system, to the communication protocol of ROS 2 and to the soft- and hardware. These experiments point out the importance of a correctly set up overall system. Even though there have been many improvements from ROS 1 to ROS 2 regarding reliability and real-time capability, the installation of ROS 2 is not sufficient in order to get a real-time capable system. This is associated with an initial effort to determine how all these settings work and where they can be set. However, since this is done and documented in this thesis, these settings can now be reused for subsequent applications.
The second set of experiments focuses more on the influence of the hardware used to run a ROS 2 application. Comparing two different hardware platforms showed clearly superior behaviour for the more powerful hardware. Both maximal jitter and round trip latency differentiate by a factor of about seven. Therefore, for a ROS 2 application that shall be able to hold time constraints, the hardware must be chosen according to the requirements.

In conclusion, one can say that ROS 2, running on powerful hardware and with proper settings enabled, can fulfil soft or firm real-time constraints. However, it is crucial that if external hardware is used, which is often the case in robotics; the communication with these components must also satisfy real-time requirements. If only one component in a chain of actions is not real-time safe, the hole chain is not real-time safe!
For real-time systems with more stringent timing requirements, ROS 2 is not suitable in its current state. Jitter comparison to a PLC shows the exemplary behaviour of the PLC regarding jitter even though running on less powerful hardware. Therefore, ROS 2 is currently unable to replace PLCs in applications such as motion control in a CNC machine. Furthermore, the paradigm of a PLC to cyclical execute one main

program facilitates meeting real-time requirements. The distributed architecture of ROS 2 results in jitter and latency for each component of the ROS 2 application, and one has to ensure that this does not add up to an unacceptable amount. However, this distributed and modular architecture is what makes ROS 2 so flexible and applicable to many use cases. Moreover, ROS 2 offers handy tools for simulations (e.g. gazebo, used for the inverted pendulum simulation) and debugging applications that can be integrated. Due to the open-source policy of ROS 2, there is a vast community which helps get started and many ROS 2 packages are already provided and do not have to be rewritten. This was also used in this thesis and can save a lot of time.
In conclusion, there will probably always be cases where other solutions than ROS 2 are more suitable. Nevertheless, with the progress from ROS 1 to ROS 2, the potential areas where ROS 2 can be used could definitely be expanded.

A measure of interest for future ROS 2 investigations might be the so-called end-to-end latency. This characterises the time it takes from the occurrence of new information to the actual consequence. In terms of the ballbot, this might be the time it takes until a new measurement of the IMU leads to a new duty cycle for the motors. This depends, of course, on the number of nodes in-between but is also an essential measure for real-time systems.
Another improvement from this work might be to investigate the performance of micro-ROS together with ROS 2 further when the micro-ROS agent has to ability to run at higher priorities. Having the possibility of a real-time save communication between a ROS 2 network and a microcontroller allows, for example, to use powerful hardware and extend it with hardware inputs and outputs via micro-ROS and a microcontroller instead of, for example, using a Raspberry Pi or similar.
Furthermore, creating a template ROS 2 package with all the necessary real-time settings elaborated in this thesis might help to save time for future projects.

Bibliography

[1] D. Zöbel, *Echtzeitsysteme: Grundlagen der Planung*, 2nd ed., ser. Lehrbuch. Berlin [Heidelberg]: Springer Vieweg, 2020.

[2] *ISO/IEC 2382:2015: Information technology - Vocabulary*, International Organization for Standardization Std., May 2015.

[3] K. Erciyes, *Distributed Real-Time Systems*. Springer-Verlag GmbH, Jul. 2019. [Online]. Available: https://www.ebook.de/de/product/37330036/k_erciyes_distributed_real_time_systems.html

[4] J. Kay and A. R. Tsouroukdissian, "Real-time control in ROS and ROS 2.0," Online, 2015, retrieved 17.11.2021. [Online]. Available: https://roscon.ros.org/2015/presentations/RealtimeROS2.pdf

[5] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Ng, "ROS: an open-source Robot Operating System," vol. 3, 01 2009.

[6] "ROS Wiki Concepts," Online, Jun. 2014, retrieved 06.10.2021. [Online]. Available: http://wiki.ros.org/ROS/Concepts

[7] A. Luntovskyy and D. Gütter, *Moderne Rechnernetze: Protokolle, Standards und Apps in kombinierten drahtgebundenen, mobilen und drahtlosen Netzwerken*, ser. Lehrbuch. Wiesbaden [Heidelberg]: Springer Vieweg, 2020.

[8] "ROS Wiki Technical Overview," Online, Jun. 2014, retrieved 06.10.2021. [Online]. Available: http://wiki.ros.org/ROS/TechnicalOverview

[9] Y. Maruyama, S. Kato, and T. Azumi, "Exploring the performance of ROS2," in *2016 International Conference on Embedded Software (EMSOFT)*, 2016, pp. 1 – 10.

[10] *OMG Data Distribution Service*, Online, Object Management Group Std. 1.4, Apr. 2015, retrieved 13.10.2021. [Online]. Available: https://www.omg.org/spec/DDS/1.4/PDF

[11] T. Dirk, "ROS 2 Middleware Interface," Online, Sep. 2017, retrieved 14.10.2021. [Online]. Available: http://design.ros2.org/articles/ros_middleware_interface.html

[12] T. Kronauer, J. Pohlmann, M. Matthe, T. Smejkal, and G. Fettweis, "Latency Analysis of ROS2 Multi-Node Systems," 2021.

[13] S. Barut, M. Boneberger, P. Mohammadi, and J. J. Steil, "Benchmarking Real-Time Capabilities of ROS 2 and OROCOS for Robotics Applications," in *2021 IEEE International Conference on Robotics and Automation (ICRA)*, 2021, pp. 708–714.

[14] J. Altenberg, "Introduction to Realtime Linux," Online, Oct. 2016, retrieved 24.11.2021.

[15] Open Robotics, "Real-time programming in ROS 2," Online, 2021, retrieved 17.11.2021. [Online]. Available: https://docs.ros.org/en/galactic/Tutorials/Real-Time-Programming.html#real-time-programming-in-ros-2

[16] The Linux Foundation, "Cyclictest," Aug. 2018, retrived 17.11.2021. [Online]. Available: https://wiki.linuxfoundation.org/realtime/documentation/howto/tools/cyclictest/start

[17] Dell Technologies, "What is the C-State?" Online, retrieved 17.11.2021. [Online]. Available: https://www.dell.com/support/kbdoc/en-uk/000060621/what-is-the-c-state?lwp=rt

[18] G. Belascuen and N. Aguilar, "Design, Modeling and Control of a Reaction Wheel Balanced Inverted Pendulum," in *2018 IEEE Biennial Congress of Argentina (ARGENCON)*, 2018, pp. 1–9.

[19] K. Stadler, "Model Predictive Control," Lecture notes, Oct. 2020.

[20] "Teensy 4.1," Online, retrived 10.05.2022. [Online]. Available: https://www.pjrc.com/store/teensy41.html

[21] E. N. Sihite, D. J. Yang, and T. R. Bewley, "Derivation of a new drive/coast motor driver model for real-time brushed DC motor control, and validation on a MIP robot," in *2019 IEEE 15th International Conference on Automation Science and Engineering (CASE)*, 2019, pp. 1099–1105.

[22] "ros2/demos," Feb. 2022, original-date: 2015-07-17T17:24:00Z. [Online]. Available: https://github.com/ros2/demos/blob/f008dbbeefd7da30c4133ad67655554d9f7f8519/pendulum_control/src/pendulum_demo.cpp

[23] Open Robotics, "About quality of service settings — ROS 2 documentation: Foxy documentation," Online, retrieved 12.03.2022. [Online]. Available: https://docs.ros.org/en/foxy/Concepts/About-Quality-of-Service-Settings.html?highlight=quality%20service

[24] ——, "Executors — ROS 2 documentation: Foxy documentation," retrieved 12.03.2022. [Online]. Available: https://docs.ros.org/en/foxy/Concepts/About-Executors.html

[25] "Debian – informationen über paket stress in buster." [Online]. Available: https://packages.debian.org/buster/stress

[26] J. Park, R. Delgado, and B. W. Choi, "Real-Time Characteristics of ROS 2.0 in Multiagent Robot Systems: An Empirical Study," *IEEE Access*, vol. 8, pp. 154 637–154 651, 2020.

[27] F. Reghenzani, G. Massari, and W. Fornaciari, "The Real-Time Linux Kernel," *ACM Computing Surveys*, vol. 52, no. 1, pp. 1–36, jan 2020.

[28] T. Lauwers, G. A. Kantor, and R. L. Hollis, "A dynamically stable single-wheeled mobile robot with inverse mouse-ball drive," *Proceedings 2006 IEEE International Conference on Robotics and Automation, 2006. ICRA 2006.*, pp. 2884–2889, 2006.

[29] M. Kumaga and T. Ochiai, "Development of a robot balanced on a ball — Application of passive motion to transport —," in *2009 IEEE International Conference on Robotics and Automation*, 2009, pp. 4106–4111.

[30] P. Fankhauser and C. Gwerder, "Modeling and Control of a Ballbot," Ph.D. dissertation, 06 2010.

[31] P. D. Ba, S.-G. Lee, S. Back, J. Kim, and M. K. Lee, "Balancing and translation control of a ball segway that a human can ride," in *2016 16th International Conference on Control, Automation and Systems (ICCAS)*. IEEE, oct 2016.

Appendix A

Project Management

A.1 Task

MSE | MASTER OF SCIENCE IN ENGINEERING

Lucerne University of Applied Sciences and Arts
HOCHSCHULE LUZERN
Technik & Architektur

MSE – Master Thesis

Horw, 20. September 2021
Seite 1/3

Aufgabenstellung für:

Marco Grossmann_____ (Masterstudent/-in)

Industrial Technologies_____ (Fachgebiet/Profil)

von

Prof. Ralf Legrand_____(Advisor/Advisorin)
Prof. Dr. Björn Jensen_____(Co-Advisor/Co-Advisorin)

Dipl. Ing. ETH Ruedi Haller_____(Experte/Expertin)

1. Arbeitstitel

Validierung der Echtzeitfähigkeit von ROS 2 in der mobilen Robotik

2. Fremdmittelfinanziertes Forschungs-/Entwicklungsprojekt

3. Industrie-/Wirtschaftspartner

Hochschule Luzern

CC Mechanische Systeme

4. Fachliche Schwerpunkte

1. Industrielle Robotik / mobile Robotik
2. Vision Technologie
3. Echtzeit Verarbeitung

FH Zentralschweiz

5. Inhalt

Kontext

Die Datenverarbeitung in Echtzeit ist nicht nur im industriellen Umfeld von Bedeutung. Insbesondere in der mobilen Robotik findet die Echtzeitdatenverarbeitung immer mehr Anwendung. Sicherheitskritische Funktionen sind hier von wesentlicher Bedeutung für die Mensch-Maschinen-Interaktion. Informationen müssen zeitnah und wenn nötig mit Priorität verarbeitet werden können. Deshalb verfügen neu entwickelte Soft- und Hardwarekomponenten im Bereich der mobilen Robotik vermehrt über Echtzeitfähigkeit.

Ziele

Ziel dieser Master Thesis ist es, das Potential von ROS 2 (Robot Operating System) hinsichtlich des echtzeitfähigen Verhaltens der Datenverarbeitung und der Kommunikation zu evaluieren und mit anderen Systemen zu vergleichen. Dies soll mit zwei Demonstratoren anhand einer Regelungstechnik Problemstellung aufgezeigt werden. Es soll sowohl ein Demonstrator mit einem als auch einer mit zwei Freiheitsgraden erstellt werden. Zudem soll das System durch einen weiteren Sensor, beispielsweise eine Kamera, ergänzt werden. Daran sollen die Möglichkeiten und Herausforderungen von ROS 2 bei der Integration von unterschiedlichen Hardwarekomponenten in ein System aufgezeigt werden.

Vorgehen

- Recherche zum Aufbau und zur Funktionsweise von ROS (1 und 2)
- Recherche zur Umsetzung der Echtzeitfähigkeit in ROS 2
- Einarbeitung in ROS
- Konzept für die Messung und Bewertung der Echtzeitfähigkeit
- Einarbeitung und Aufsetzung eines eichtzeitfähigen Betriebssystem unter Linux
- Analyse und Gegenüberstellung von ROS 2 zu ROS 1 und anderen echtzeitfähigen Systemen bezüglich Echtzeitfähigkeit und Integrationsaufwand
- Definition des Demonstrators mit einem Freiheitsgrad
- Planung und Umsetzung des Demonstrators mit einem Freiheitsgrad
- Analyse und Bewertung des Demonstrators mit einem Freiheitsgrad
- Definiton des Demonstrators mit zwei Freiheitsgraden
- Planung und Umsetzung des Demonstrators mit zwei Freiheitsgraden
- Auswertung und Verifizierung der Echtzeitfähigkeit
- Aufzeigen der Möglichkeiten und Herausforderungen
- Demonstration der beiden Versuchsaufbauten

6. Fachliteratur/Web-Links/Hilfsmittel

- ROS Dokumentation

7. Durchführung der Arbeit

Termine

Start der Arbeit:	Mo, 20.09.2021 (Semesterbeginn HS2021)
Abgabe der Aufgabenstellung:	bis spätestens Fr, 24.09.2021, 17.00 Uhr
Master Thesis Kolloquium:	Anfang Mai
Zwischenbesprechung:	während des Semesters

Abgabe Prüfungsexemplar:	bis spätestens Fr, 17.06.2022 um 17.00 Uhr als Upload auf ILIAS und direkt an Advisor/-in und Experte/-in
Verteidigung:	bis spätestens Mi, 29.06.2022
Meldung Grade:	bis spätestens Do, 07.07.2022 um 12.00 Uhr
Abgabe def. Master Thesis:	bis spätestens Fr, 15.07.2022 um 17.00 Uhr als Upload auf ILIAS und direkt an Advisor/-in und Experte/-in
Diplomposterausstellung:	Sa, 08.07.2022

→ Weitere Termine gemäss Ablauf Master Thesis

8. Form der Abgabe der Master Thesis

Das Prüfungsexemplar der Master Thesis wird ausschliesslich online eingereicht (für Advisor/Advisorin und Experte/Expertin). Nach der Verteidigung wird die finale Version der Master Thesis im pdf_Format auf ILIAS hochgeladen und via Sekretariat BA&MA an die Bibliothek zur Archivierung gegeben. Die Angabe zur Sicherheitsstufe ist zwingend erforderlich: Öffentlich mit oder ohne Internet/intern/vertraulich, damit diese korrekt durch die Bibliothek archiviert werden kann.

9. Titelblatt

Für die Arbeit muss zwingend das von der Bibliothek vorgegebene Titelblatt verwendet werden. Dieses ist auf MyCampus abrufbar. Es besteht die Möglichkeit neben diesem noch ein eigenes Titelblatt einzufügen.

10. Selbstständigkeits- und Redlichkeitserklärung

Die Selbstständigkeits- und Redlichkeitserklärung muss zwingend zusammen mit der Master Thesis abgegeben werden. Dieses Dokument darf jedoch nicht in die Master Thesis eingebunden werden, sondern muss als separates Dokument mitabgeben werden. Es ist die Vorlage der Bibliothek zu verwenden. Diese ist auf MyCampus abrufbar.

11. Zusätzliche Bemerkungen

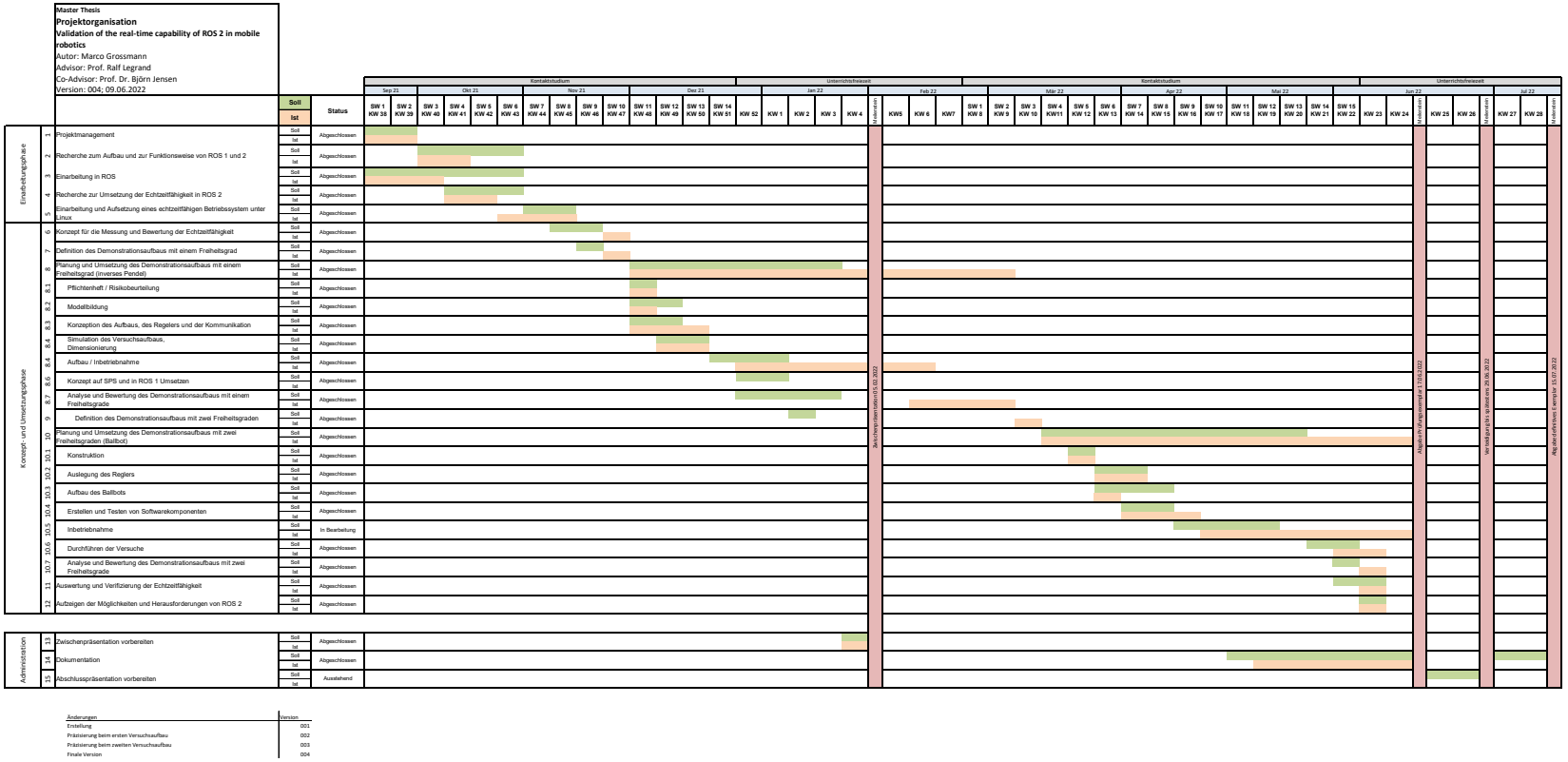
Betreffend Geheimhaltung und Rechte am Geistigen Eigentum ist die Vereinbarung zwischen dem Student bzw. der Studentin, der HSLU und dem Industrie-/Wirtschaftspartner massgeblich. Eine Vorlage hierfür ist auf MyCampus abrufbar.

Horw, 20.09.2021

Advisor/Advisorin	Experte/Expertin	Student/Studentin

Die Aufgabenstellung der Master Thesis muss mit allen Unterschriften bis **spätestens Ende der 1. Woche des Kontaktstudiums dem Sekretariat BA&MA via mse@hslu.ch** abgeben werden. Änderungen können danach jeweils per E-Mail übermittelt werden.

A.2 Project Plan



Änderungen	Version
Erstellung	001
Präzisierung beim ersten Versuchsaufbau	002
Präzisierung beim zweiten Versuchsaufbau	003
Finale Version	004

Appendix B

Experimental Setup: Inverted Pendulum

B.1 System Specification

Horw, 6. Februar 2022
Page 1/5

Experimental Setup: Inverted Pendulum with Reaction Wheel

	Name	Date	Signature
Prepared by:	Marco Grossmann	27.11.2021	m.g.

Table of Revision

Version Nr.	Datum	Status	Änderungen und Bemerkungen	Bearbeitet von
1	27.11.2021	In work	Initial Issue	M. Grossmann
2	06.02.2022	Final	Minor changes	M. Grossmann

Table of Contents

1. Introduction

1.1. Scope

This document contains the Requirements Specification of the Experimental Setup: Inverted Pendulum with Reaction Wheel.

It contains the following:

- Functional & Non-Functional Requirements
- Design Constraints & Interface Requirements
- General Requirements

1.2. Background

This requirements specification is written in context of the master thesis “Validation of the real-time capabilities or ROS 2 in mobile robotics”. It describes the requirements for the first experimental setup, which will be used to show and validate the real-time capabilities of ROS 2 and compare it to other systems.

1.3. Numbering

Each requirement is given a unique number:

- SYS-xxx
  - SYS Overall System name: Reaction Wheel Pendulum
  - xxxx Unique number

2. Functional & Non-Functional Requirements

2.1. Functional Requiremens

Reaction Wheel Pendulum-1110
The system shall be able to keep the pendulum in the $\theta = 0$ rad position.

Reaction Wheel Pendulum-1115
The starting position shall be in the $\theta = 0$ rad position.

3. Design Constraints & Interface Requirements

3.1. Design Constraints

Reaction Wheel Pendulum-2110
The system shall be running with ROS 2 on a notebook with an Ubuntu OS

Reaction Wheel Pendulum-2115
The system shall be running on a Beckhoff PLC with TwinCAT 3.

Reaction Wheel Pendulum -2120
The components in uses shall be compatible to be used with ROS as well as with a PLC.

Reaction Wheel Pendulum -2125
Refers to Reaction Wheel Pendulum-2110: The notebook will be extendable by additional hardware (e.g., Arduino) to read sensor data or to run a motor.

Reaction Wheel Pendulum -2130
The reaction wheel shall be driven by a brushed DC motor.

3.2. Environmental Requirements

Reaction Wheel Pendulum-2210
The system shall run under all light conditions.

Reaction Wheel Pendulum-2215
The system shall operate in a temperature range between +10° C and + 30° C.

3.3. Interface Requirements

3.3.1. Mechanical Interfaces

Reaction Wheel Pendulum-2310
The pendulum will be mounted on a frame.

Reaction Wheel Pendulum-2315
The whole system will be portable.

3.3.2. Electrical Interface

Reaction Wheel Pendulum-2330
All external power sources except for the notebook and the PLC shall have a max. voltage of 24V DC.

3.3.3. Data Interface

Reaction Wheel Pendulum-2340
The system shall log statistical information (encoder information, jitter etc.) into a separate file.

4. General Requirements

4.1. Safety & Reliability

Reaction Wheel Pendulum-4110
The frame where the pendulum is mounted shall be ready to mount a safety cage.

B.2 Risk Assessment

Master Thesis

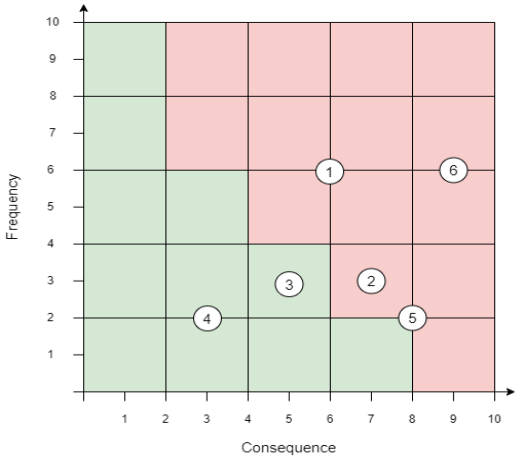
Risk assesment for the inverted Pendulum

Validation of the real-time capability of ROS 2 in mobile robotics

Autor: Marco Grossmann
Advisor: Prof. Ralf Legrand
Co-Advisor: Prof. Dr. Björn Jensen

Version: 002; 30.11.2021

Number	Failure Mode Description	Frequency / Probability	Consequence	Risk	Recommendations
	Development Risks				
1	Supply shortage	6	6	36	Use as many available parts from the lab as possible
2	Components wrong dimensioned	3	7	21	Use simulation to prevent desing faults
3	Difficulties in modeling the dynamics of the system	3	5	15	Search for publications on that topic
4	Dynamic Model oversimplificated	2	3	6	
	Operational Risks				
5	Hardware failure	2	8	16	
6	Injury of people	6	9	54	Plan security cage for long-term usage



Änderungen	Version
Initial version	001
Added graphical risk matrix	002

B.3 Control System

In this section the conversion from the linear differential equation of the dynamical system to the state-space representation is shown. It is based on [18].
Starting point is the dynamical system (see equation 5.9):

$$I_{\text{tot}} \cdot \ddot{\theta} = (F_1 \cdot L_{\text{p}} + F_2 \cdot \frac{1}{2} L_{\text{p}}) \cdot \theta - \mu_{\text{j}} \cdot \dot{\theta} - T_{\text{w}} \tag{B.1}$$

with:

$$I_{\text{tot}} = I_{\text{w, j}} + I_{\text{m, j}} + I_{\text{p}} \tag{B.2}$$

$$I_{\text{w, j}} = m_{\text{w}} \cdot L_{\text{p}}^2 \tag{B.3}$$

$$I_{\text{m, j}} = m_{\text{m}} \cdot L_{\text{p}}^2 \tag{B.4}$$

Further examination of the reaction wheel torque T_{w} leads to:

$$I_{\text{w}} \cdot \dot{\omega}_{\text{inertial}} = T_{\text{w}} \tag{B.5}$$

However, the inertial angular velocity ω_{inertial} of the reaction wheel is difficult to measure. Instead, the relative angular velocity ω_{w} between the pendulum bar and the reaction wheel is measured. Therefore:

$$\omega_{\text{inertial}} = \dot{\theta} + \omega_{\text{w}} \tag{B.6}$$

In the next step, the reaction wheel torque that is generated by a DC motor is expressed by means of the electromechanical model of the motor. For simplicity and because they have rather small influence on the model non linear friction terms are ignored in equation B.7.

$$T_{\text{w}} = T_{\text{e}} - \mu_{\text{m}} \cdot \omega_{\text{w}} \tag{B.7}$$

$$T_{\text{e}} = k_{\text{t}} \cdot i \tag{B.8a}$$

$$R \cdot i + L \cdot \frac{\text{d}i}{\text{d}t} = u(t) - k_{\text{e}} * \omega_{\text{w}} \tag{B.8b}$$

These further examinations of the dynamical system can be summarised into a system of equations:

$$I_{\text{tot}} \cdot \ddot{\theta} = (F_1 \cdot L_{\text{p}} + F_2 \cdot \frac{1}{2} L_{\text{p}}) \cdot \theta - \mu_{\text{j}} \cdot \dot{\theta} - T_{\text{w}} \tag{B.9a}$$

$$I_{\text{w}} \cdot \dot{\omega}_{\text{inertial}} = T_{\text{w}} \tag{B.9b}$$

$$T_{\text{w}} = T_{\text{e}} - \mu_{\text{m}} \cdot \omega_{\text{w}} \tag{B.9c}$$

$$T_{\text{e}} = k_{\text{t}} \cdot i \tag{B.9d}$$

$$R \cdot i + L \cdot \frac{\mathrm{d}i}{\mathrm{d}t} = u(t) - k_{\text{e}} * \omega_{\text{w}} \tag{B.9e}$$

$$\omega_{\text{inertial}} = \dot{\theta} + \omega_{\text{w}} \tag{B.9f}$$

As already stated in section 5.2, the number of states in the state-space model is equal to the number of storage elements. According to [18] there are three state variables of interest for the system of an inverted pendulum:

1. Deflection angle of the pendulum bar θ
2. Angular velocity of the pendulum bar $\dot{\theta}$
3. Angular velocity of the reaction wheel ω_{w}

Therefore, the state variables are:

$$x_1 = \theta \tag{B.10}$$

$$x_2 = \dot{\theta} \tag{B.11}$$

$$x_3 = \omega_{\text{w}} \tag{B.12}$$

The outcome of yields for the first state:

$$\dot{x}_1 = x_2 \tag{B.13}$$

From equations B.9c, B.9d and B.9e, T_{w} can be expressed as follows (ω_{w} is substituted by x_3):

$$T_{\text{w}} = \frac{k_{\text{t}}}{R} u(t) - (\frac{k_{\text{t}}}{R} k_{\text{e}} + \mu_{\text{m}}) x_3 \tag{B.14}$$

The equation for the second state can be found in equation B.9a. Reformulated and T_{w} substituted by equation B.14 this leads to:

$$\dot{x}_2 = \frac{F_1 \cdot L_{\text{p}} + F_2 \cdot \frac{1}{2} L_{\text{p}}}{I_{\text{tot}}} x_1 - \frac{\mu_{\text{m}}}{I_{\text{tot}}} x_2 + \frac{\frac{k_{\text{t}}}{R} k_{\text{e}} + \mu_{\text{m}}}{I_{\text{tot}}} x_3 - \frac{k_{\text{t}}}{R I_{\text{tot}}} u(t) \tag{B.15}$$

The third state comes form equation B.9b. $\dot{\omega}_{\text{inertial}}$ will be substituted by equation B.9f and T_{w} by equation B.14.

$$I_{\text{w}}(\dot{\omega}_{\text{w}} + \ddot{\theta}) = \frac{k_{\text{t}}}{R} u(t) - (\frac{k_{\text{t}}}{R} k_{\text{e}} + \mu_{\text{m}}) x_3 \tag{B.16}$$

Then $\dot{\omega}_{\text{w}}$ is substituted by $\dot{x}_3$ and $\ddot{\theta}$ by $\dot{x}_2$:

$$I_{\mathrm{w}}(\dot{x}_3 + \dot{x}_2) = \frac{k_{\mathrm{t}}}{R}u(t) - (\frac{k_{\mathrm{t}}}{R}k_{\mathrm{e}} + \mu_{\mathrm{m}})x_3 \tag{B.17}$$

In the last step, $\dot{x}_2$ is substituted by equation B.15 and $\dot{x}_3$ is taken to the left side of the equation:

$$\dot{x}_3 = -\frac{F_1 \cdot L_{\mathrm{p}} + F_2 \cdot \frac{1}{2}L_{\mathrm{p}}}{I_{\mathrm{tot}}}x_1 - \frac{\mu_{\mathrm{m}}}{I_{\mathrm{tot}}}x_2 - \frac{I_{\mathrm{tot}} + I_{\mathrm{w}}}{I_{\mathrm{tot}}I_{\mathrm{w}}}(\mu_{\mathrm{m}} + \frac{k_{\mathrm{t}}k_{\mathrm{e}}}{R})x_3 + \frac{I_{\mathrm{tot}} + I_{\mathrm{w}}}{I_{\mathrm{tot}}I_{\mathrm{w}}}\frac{k_{\mathrm{t}}}{R}u(t) \tag{B.18}$$

The three sate equations B.13, B.15 and B.18 can then be brought to matrix formulation. The formulation has already be seen in section 5.2 in equation 5.10 and 5.11.

$$\dot{x}(t) = A_c x(t) + B_c u(t) \qquad \text{State equation} \tag{B.19}$$

$$y(t) = C_c x(t) + D_c u(t) \qquad \text{Output equation} \tag{B.20}$$

with:

$$A_c = \begin{bmatrix} 0 & 1 & 0 \\ \frac{F_1 \cdot L_{\mathrm{p}} + F_2 \cdot \frac{1}{2}L_{\mathrm{p}}}{I_{\mathrm{tot}}} & -\frac{\mu_{\mathrm{j}}}{I_{\mathrm{tot}}} & \frac{k_{\mathrm{t}}k_{\mathrm{e}}}{RI_{\mathrm{tot}}} + \frac{\mu_{\mathrm{m}}}{I_{\mathrm{tot}}} \\ -\frac{F_1 \cdot L_{\mathrm{p}} + F_2 \cdot \frac{1}{2}L_{\mathrm{p}}}{I_{\mathrm{tot}}} & \frac{\mu_{\mathrm{j}}}{I_{\mathrm{tot}}} & -\frac{I_{\mathrm{tot}} + I_{\mathrm{w}}}{I_{\mathrm{tot}}I_{\mathrm{w}}}(\mu_{\mathrm{m}} + \frac{k_e k_t}{R}) \end{bmatrix} \qquad B_c = \begin{bmatrix} 0 \\ -\frac{k_{\mathrm{t}}}{RI_{\mathrm{tot}}} \\ \frac{I_{\mathrm{tot}} + I_{\mathrm{w}}}{I_{\mathrm{tot}}I_{\mathrm{w}}}\frac{k_{\mathrm{t}}}{R} \end{bmatrix} \tag{B.21}$$

The output matrix C_c is the identity matrix, because all the states are measured by means of sensors [18]. The feedthrough matrix D_c is a null matrix, because the system is strictly proper [19].

B.4 Data Sheets

All the data sheets for the components of the inverted pendulum are in the electronic version of the appendix in folder «*10_Experimental_Setup_Inverted_Pendulum/10_Data_Sheets/*».
If this document lies within the electronic appendix folder, the documents can be opened via the following links (press *ctrl* while klicking on the link in order to open the pdf in a new tab):

- Data sheet dc motor
- Data sheet gear box
- Data sheet laser distance sensor
- Data sheet encoder
- Data sheet H-bridge

B.5 General System Settings

C-States

The c-states are disabled under /etc/default/grub:

```
1 $ sudo nano /etc/default/grub
```

To disable c-states add the following line to the opened file:

```
1 GRUB_CMDLINE_LINUX="idle=poll"
```

i7z

i7z is a tool to monitor the c-states on intel cpu's.

```
1 $ sudo apt-get install i7z
2 $ sudo i7z
```

With c-states disabled, state C0 should be active all the time (100%).

CPU Isolation

CPU isolation is done in the grub menu under /etc/default/grub:

```
1 $ sudo nano /etc/default/grub
```

To isolate a cpu core, the same variable is used as for the disabling of the c-states. Therefore, this variable is appended with the cpu isolation information. The code below shows the isolation of the fourth cpu core (first core has number 0!).

```
1 GRUB_CMDLINE_LINUX="idle=poll isolcpus=3"
```

Stress

In order to simulate additional cpu usage, install and run the stress package.

```
1 $ sudo apt-get install stress
2 $ stress --cpu 3 # --cpu defines the number of cores that are fully used
      ↪ to capacity
```

B.6 ROS 2 Settings

History

```
#include "rclcpp/rclcpp.hpp"
auto qos = rclcpp::QoS(
rclcpp::KeepLast(1));
```

Reliability

Based on the qos instance from above, the reliability can be set:

```
// reliable
qos.reliable();

// best effort
qos.best_effort();
```

Deadline

Based on the qos instance from above, the deadline can be set:

```
std::chrono::milliseconds deadline_duration(10);
qos.deadline(deadline_duration);
```

Executor

The executor settings are done in the main function.

```
// Single Threaded
rclcpp::executors::SingleThreadedExecutor executor;

// Static Single Threaded
rclcpp::executors::StaticSingleThreadedExecutor executor;

// Multi Threaded
rclcpp::executors::MultiThreaded executor;

executor.add_node(node);
executor.spin();
```

Priority

The first step in order to set the priority of a task is to set the right permissions. This is done under /etc/security/limits.conf (the file must be edited with root privileges).

```
<your username>    -    rtprio    98
```

The priority can then be set in a ROS 2 node with the help of the *rttest package* provided by ROS 2. This is done in the main function.

```
1 #include <rttest/rttest.h>
2 if (rttest_set_sched_priority(98, SCHED_RR)) {
3   perror("Couldn't set scheduling priority and policy");
4 }
```

Memlock

Similar to the priority settings, the first step for the memory locking is to set the right permissions. This is done in the same file as for the priority settings (/etc/security/limits.conf)

```
1 <your username>    -    memlock    <limit in kB>
```

The the memory allocation can be done in the main function by the help of the *rttest package*.

```
1 #include <rttest/rttest.h>
2 if (rttest_lock_and_prefault_dynamic() != 0) {
3   fprintf(stderr, "Couldn't lock all cached virtual memory.\n");
4   fprintf(stderr, "Pagefaults from reading pages not yet mapped into RAM
    ↪ will be recorded.\n");
5 }
```

CPU Affinity

The settings for the CPU affinity are done in the main function of the ROS 2 node.

```
1 #include <sched.h>
2 cpu_set_t  mask;
3 CPU_ZERO(&mask);
4 CPU_SET(3, &mask);
5 int result = sched_setaffinity(0, sizeof(mask), &mask);
6 if (result != 0){
7   fprintf(stderr, "CPU NOT Available\n");
8 }
```

B.7 micro-ROS Installation

This section explains how to install the micro-ROS agent on a Ubuntu 20.04 system for ROS 2 Foxy Fitzroy. Therefore exchange `$ROS_DISTRO` with `foxy`. The instructions are based on a micro-ROS tutorial.
The first step is to source the ROS 2 installation:

```
1 $ source /opt/ros/$ROS_DISTRO/setup.bash
```

Create a separate workspace for micro-ROS:

```
$ mkdir microros_ws
$ cd microros_ws
$ git clone -b $ROS_DISTRO https://github.com/micro-ROS/micro_ros_setup.
    ↪ git src/micro_ros_setup
```

Then update the dependencies and install pip:

```
$ rosdep update
$ rosdep install --from-paths src --ignore-src -y
$ sudo apt-get install python3-pip
```

Build the micro-ROS tools and source them:

```
$ colcon build
$ source install/local_setup.bash
```

Then the micro-ROS agent can be installed and built:

```
$ ros2 run micro_ros_setup create_agent_ws.sh
```

```
$ ros2 run micro_ros_setup build_agent.sh
$ source install/local_setup.bash
```

To test if the installation was successful run:

```
$ ros2 run micro_ros_agent micro_ros_agent serial --dev /dev/ttyACM0
```

If the installations was successful messages should appear that no serial port was found.

If the port name is not known, one can search for all the available ports:

```
$ ls -l /dev/ttyACM*
```

Appendix C

Experimental Setup: Ballbot

C.1 Risk Assessment Ballbot

Master Thesis

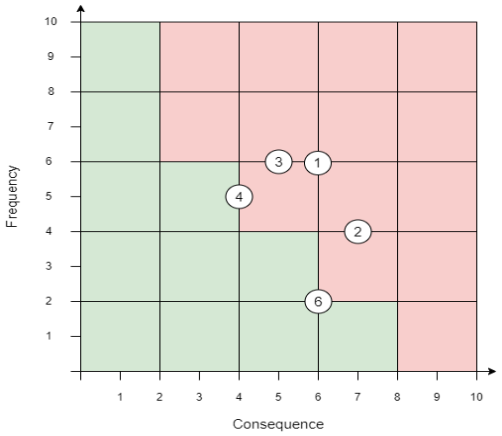
Risk assesment for the ballbot

Validation of the real-time capability of ROS 2 in mobile robotics

Autor: Marco Grossmann
Advisor: Prof. Ralf Legrand
Co-Advisor: Prof. Dr. Björn Jensen

Version: 002; 15.05.2022

Number	Failure Mode Description	Frequency/ Probability	Consequence	Risk	Recommendations
		Development Risks			
1	Supply shortage	6	6	36	Use as many available parts from the lab as possible, watch out for delivery time
2	Components wrong dimensioned or not appropriate	4	7	28	Use existing ballbots as reference / use knowledge gained from the inverted pendulum
3	Difficulties in modeling the dynamics of the system	6	5	30	Search for publications on that topic
4	Sensor noise leads to wrong informations	5	4	20	Use an appropriate filter if needed
		Operational Risks			
6	Hardware failure	2	8	16	



Änderungen	Version
Erstellung	001
Added new risks and graphical representation	002

C.2 Raspberry Pi Setup

This section describes how to setup a Raspberry Pi with Ubuntu, ROS 2 and the RT_PREEMPT patch. The first step is to install a current version of Ubuntu on the Raspberry Pi 4b. Download the image on a notebook running Ubuntu. Images can be found here. Copy the downloaded image onto an empty micro sd-card. To copy the file onto the sd-card, right click on the image file and choose «Open With Disk Image Writer», select the sd-card and click on «Start restoring...».
After that, the sd-card can be removed from the notebook an plugged in to the Raspberry Pi. After the first boot, one has to change the password. The standard username and password is for both ubuntu.
If needed, Ubuntu desktop environment can be installed. Therefore, internet access is required. This can be provided via an external Computer. Plug in an Ethernet cable to both, the Raspberry Pi and the notebook. On the notebook, choose «Shared to other computers» for the Ethernet port IPv4 settings.

```
1 $ sudo apt-get update
2 $ sudo apt-get upgrade
3 $ sudo apt-get install ubuntu-desktop
```

After rebooting, the Raspberry Pi should start up with the desktop environment. Now the internet can be accessed via a wifi connection.
The next step is to set up the ip-address of the Ethernet port on the Raspberry Pi in order to enable ssh connection. This can be edited under /etc/netplan/50-cloud-init.yaml.

```
1 network:
2   version: 2
3   renderer: NetworkManager
4   ethernets:
5     eth0:
6       dhcp4: no
7       addresses: [10.180.30.198/24]
```

After the file has been configured, the settings must be applied:

```
1 $ sudo netplan apply
```

To enable remote desktop connection from windows, install xrdp:

```
1 $ sudo apt-get install xrdp
```

The second step is to build the patched kernel: One way to do so is to use the *Build RT_PREEMPT kernel for Raspberry Pi 4* repository from the ROS 2 Real-Time Working Group. The kernel image is built on a notebook running Ubuntu 20.04:

```
1 $ git clone https://github.com/ros-realtime/linux-real-time-kernel-builder
2 $ cd linux-real-time-kernel-builder
```

Build the docker container:

```
1 $ docker build -t rtwg-image .
```

and run it:

```
1 $ docker run -t -i rtwg-image bash
```

The next step is to build the kernel inside the docker container. The default settings contains among others the following settigns:

- RT_PREEMPT kernel
- Fixed operation frequency at 1.0 GHz
- CPU1, CPU2 and CPU3 tickless
- No CPU frequency scaling

To change these settings, please have a closer look at the GitHub repository of the ROS 2 Real-Time Working Group. When the setup is properly configured, build the kernel:

```
1 $ cd linux-raspi-5.4.0/
2 $ make ARCH=arm64 CROSS_COMPILE=aarch64-linux-gnu- -j `nproc` deb-pkg
```

After the built has finished, check that the following four files exist:

```
1 user@3e9fd281ed2a:~/linux_build/linux-raspi-5.4.0 $ ls -la ../*.deb
2 ../linux-headers-5.4.101-rt53_5.4.101-rt53-1_arm64.deb
3 ../linux-image-5.4.101-rt53-dbg_5.4.101-rt53-1_arm64.deb
4 ../linux-image-5.4.101-rt53_5.4.101-rt53-1_arm64.deb
5 ../linux-libc-dev_5.4.101-rt53-1_arm64.deb
```

Now, the four files must be copied from the docker container onto the home directory of the Raspberry Pi. This can be done via ssh (command is executed from inside the running docker container on the notebook):

```
1 $ cd ~/linux_build
2 $ scp *.deb ubuntu@10.180.30.198:/home/ubuntu
```

Then the kernel can be installed on the Raspberry Pi:

```
1 $ cd ~
2 $ sudo dpkg -i *.deb
3 $ sudo reboot
```

The last step is to replace the old kernel with the new one:

```
1 $ cd /boot
2 $ sudo ln -s -f vmlinuz-5.4.140-rt64 vmlinuz
3 $ sudo ln -s -f vmlinuz-5.4.0-1052-raspi vmlinuz.old
4 $ sudo ln -s -f initrd.img-5.4.140-rt64 initrd.img
5 $ sudo ln -s -f initrd.img-5.4.0-1052-raspi initrd.img.old
6 $ sudo cp vmlinuz firmware/vmlinuz
```

```
$ sudo cp vmlinuz firmware/vmlinuz.bak
$ sudo cp initrd.img firmware/initrd.img
$ sudo cp initrd.img firmware/initrd.img.bak
$ sudo reboot
```

After the reboot, check that the new kernel is booted:

```
$ uname -a
Linux ubuntu 5.4.140-rt64 #1 SMP PREEMPT_RT Sat Oct 16 17:57:52 UTC 2021
    ↪ aarch64 aarch64 aarch64 GNU/Linux
```

In order to get a better real-time performance, three CPU cores of the Raspberry Pi will be isolated. Core isolation can be done under /boot/firmware/cmdline.txt: Append

```
isolcpus=1-3
```

at the end of the line to isolate the fourth core. The success of the isolation can be checked with:

```
$ cat /sys/devices/system/cpu/isolated
```

In order of success, the number of the isolated core should be printed to the terminal.
Another important setting is to allow the user memory allocation as well as setting a priority. Therefore set the permissions under /etc/security/limits.conf:

```
<your username>    -    rtprio    98
<your username>    -    memlock    -1
```

After editing, run:

```
$ ulimit -l unlimited
```

In order to activate the changed permissions, log out and log back in.

C.3 I2C

This section describes how to setup the i2c-bus for the communication between the sesnors and the Raspberry Pi.
The first step is to add the user to the i2c group:

```
$ sudo usermod -aG i2c ${USER}
```

The next step is to enable software i2c. For that, add the following line to /boot/firmware/usercfg.txt.

```
dtoverlay=i2c-gpio,bus=3
```

To check which devices are on the bus, install the i2c-tools and detect the participants:

```
$ sudo apt-get install i2c-tools
$ i2cdetect -y 3
```

C.4 IMU ROS 2 Package

The ROS 2 node that is used to read the sensor information from the BNO055 IMU is based on an existing ROS 2 package. In this section the installation of this package is described.
The first step is to instal the dependencies:

```
1 $ sudo apt-get install libi2c-dev
```

Then the package can be cloned from git. After that, a patch must be installed because the Bosch Sensortec API is written in C.

```
1 $ git clone --recurse-submodules https://github.com/bdholt1/
    ↪ ros2_bno055_sensor.git
2 $ cd ros2_bno055_sensor/thirdparty/BNO055_driver/
3 $ git apply ../../bno055.h.patch
```

After these steps, the package can be built. This package is used as a baseline. It is extended with the real-time settings explained in section 5.6.1.

C.5 PWM extension

To send duty cycles from a ROS 2 node to the PCA9685 an existing C++ library is wrapped into the ROS 2 node.

```
1 $ git clone https://github.com/TeraHz/PCA9685.git
```

The files from the source folder are then added to the third-party folder of the ballbot repository.

C.6 WiringPi

To read the serial messages from the microcontroller, wiringpi library is required:

```
1 $ git clone https://github.com/WiringPi/WiringPi.git
2 $ cd WiringPi
3 $ ./build
```

To check the installation type:

```
1 $ gpio -v
2 $ gpio readall
```

If *pgio readall* requires root permissitons, run:

```
1 $ sudo adduser "${USER}" dialout
2 $ sudo reboot
```

C.7 Eigen 3

The controller node requires the *Eigen* library:

```
$ cd ros_ws/src
$ git clone https://github.com/ros2/eigen3_cmake_module.git
$ cd ..
$ colcon build
```

In the CMakeLists.txt file of the ballbot use:

```
find_package(eigen3_cmake_module REQUIRED)
find_package(Eigen3)
ament_target_dependencies(controller Eigen3)
```

and add

```
<buildtool_depend>eigen3_cmake_module</buildtool_depend>
<build_depend>eigen</build_depend>
```

to the package.xml file.