Hochschule Karlsruhe
University of
Applied Sciences

Faculty of Mechanical Engineering and Mechatronics
Degree Program: Mechatronics

Balancieren eines einrädrigen Fahrzeugs mit Hilfe des Gyroskopischen Effekts

Balancing a One-Wheeled Vehicle Using the Gyroscopic Effect

Bachelor Thesis (B.Eng.)

by
Ruben Egle
Date of Birth: 09.11.2001
Student ID: 79733

University Supervisor (Karlsruhe University of Applied Sciences)
Prof. Dr.-Ing. Joachim Wietzke

Second Supervisor (Karlsruhe University of Applied Sciences)
Prof. Dr.-Ing. Tobias Baas

Period of Work
May 22, 2025 – October 20, 2025

Hochschule Karlsruhe
University of
Applied Sciences

Fakultät für
Maschinenbau und
Mechatronik

HKA

Bachelor-Thesis:	**Ruben Egle** **Mechatronik**
Arbeitsplatz:	Hochschule Karlsruhe Fakultät für Maschinenbau und Mechatronik 76133 Karlsruhe
Betreuer am Arbeitsplatz:	Prof. Dr. Joachim Wietzke
Betreuender Dozent:	Prof. Dr. Joachim Wietzke
Datum der Ausgabe: 22.05.2025	**Abgabetermin:** 22.09.2025

Kurztitel / Subject:

Balancieren eines einrädrigen Fahrzeugs mit Hilfe des Gyroskopischen Effekts
Balancing a One-Wheeled Vehicle Using the Gyroscopic Effect

Beschreibung des Themas:

In den 60er Jahren hat der amerikanischer Ingenieur Charles F. Taylor ein einrädriges Fahrzeug, genannt One-Wheeled Vehicle, entworfen und einen Prototyp gebaut. Das Fahrzeug besteht aus einer Plattform, welche an einem Rad aufgehängt ist. Es balanciert in Fahrtrichtung über eine vom Motordrehmoment abhängige Verschiebung der Plattform relativ zum Rad und seitlich über den Einsatz eines Gyroskops. Dieses wird auch zum Steuern des Fahrzeugs verwendet. In einem vorangehenden Forschungsprojekt wurde ein Prototyp, genannt Monowheeler, zur Umsetzung eines solchen Fahrzeugs mit moderner Technik entworfen und die Stabilisierung in Fahrtrichtung in Betrieb genommen.

Das Ziel dieser Arbeit ist die vollständige Inbetriebnahme des Monowheelers und die Umsetzung der seitlichen Stabilisierung mit einem Gyroskop.

Im Einzelnen sind die folgenden Punkte zu bearbeiten:

- Aufbau, Anpassung und Inbetriebnahme der Hardware
- Erarbeitung und Aufbau eines Sensorkonzepts
- Vergleich und Entwurf von klassischen und modellbasierten Reglerkonzepten
- Umsetzung des Reglers auf einem Mikrokontroller
- Dokumentation und Präsentation der Ergebnisse

Optionales Ziel:

- Konzeption und Entwurf einer Steuerung zum Kurvenfahren

Vorsitzende des Prüfungsausschusses

Weygand

Prof. Dr.mont. Sabine Weygand

Verlängerung der Abschlussarbeit

Neuer Abgabetermin: 20.10.2025

Weygand

Prof. Dr.mont. Sabine Weygand

Declaration of Authorship

I hereby truthfully declare that I have written this thesis independently, that I have fully and accurately cited all sources used, and that I have clearly indicated everything that has been taken from the work of others, either unchanged or with modifications.

Karlsruhe, October 16, 2025

Signature:

Acknowledgements

I would like to take this opportunity to thank everyone who accompanied and supported me in the preparation of this bachelor's thesis. First and foremost, I would like to thank Prof. Dr. Joachim Wietzke, who supervised this thesis. He accompanied the project from the very beginning with outstanding technical and organizational support. Special thanks also go to Prof. Dr. Tobias Baas for supervising the thesis as co-examiner, providing the EML's laboratory facilities and resources, and contributing his technical expertise, which was instrumental to the success of this thesis.

I would also like to thank Prof. Ansgar Blessing for his valuable technical input and contributions.

Finally, I would like to thank my family and my girlfriend for their mental support, their help with proofreading, and their assistance in obtaining literature. Without their support, this work would not have been possible in this form.

Abstract

The *Monowheeler* is an actively stabilized, single-wheeled vehicle that uses the gyroscopic effect for balancing. It is an underactuated, unstable system with highly coupled, nonlinear dynamics of roll and yaw motion. The *Monowheeler* serves as an experimental platform for testing the novel concept presented in this thesis for stabilizing and controlling a single-wheeled vehicle, which was inspired by Charles Taylor's *One-Wheeled Vehicle* from 1964. The unstable pitch dynamics are controlled by shifting the wheel relative to the rest of the vehicle. The gyroscopic effect is used to control the equally unstable rolling dynamics. Through the targeted use of the coupled degrees of freedom, maneuvers for cornering can be performed.

To implement the control system on the real vehicle, the existing mechanical platform is expanded to include a suitable sensor system and a circuit board for connecting all electrical components. The vehicle states are estimated using data from an inertial measurement unit through sensor fusion with a complementary filter. The sensors and actuators are controlled and the discrete control loop is implemented on a Linux-based microcontroller.

The control system is designed based on the mathematical model of the vehicle and optimized in a simulation. The results are then validated experimentally and adapted to the real system in an iterative process. Both classical and model-based control concepts are investigated. The pitch movement is controlled using an LQR controller with gain scheduling. The roll movement is controlled by a PID controller. By specifying controller-based trajectories for the roll movement, the behavior of the vehicle can be controlled in a targeted manner. This ensures that disturbances are compensated for by actively tilting the vehicle, or the coupling of the roll and yaw movements is used for cornering. In the process, the limitations of classic control concepts can be demonstrated and suggestions for the use of complex control algorithms can be made.

Various experiments have shown that the *Monowheeler* can both balance and corner in a targeted and controlled manner. This is the first time that an actively stabilized unicycle of this size has been realized that uses the gyroscopic effect to move freely on a plane.

Kurzfassung

Der *Monowheeler* ist ein aktiv stabilisiertes, einrädriges Fahrzeug, dass den gyroskopischen Effekt zum Balancieren nutzt. Es handelt sich um ein unteraktuiertes, instabiles System mit hochgradig gekoppelter, nichtlinearer Dynamik der Roll- und Gierbewegung. Der *Monowheeler* dient als experimentelle Plattform zur Erprobung des in dieser Arbeit vorgestellten, neuartigen Konzept zur Stabilisierung und Steuerung eines einrädrigen Fahrzeugs, welches von Charles Taylors *One-Wheeled-Vehicle* aus dem Jahr 1964 inspiriert wurde. Die instabile Nickdynamik wird durch eine Verschiebung des Rads relativ zum Rest des Fahrzeugs kontrolliert. Zur Steuerung der ebenso instabilen Rolldynamik wird der gyroskopische Effekt genutzt. Durch die gezielte Nutzung der gekoppelten Freiheitsgrade können Manöver zum Kurvenfahren durchgeführt werden.

Zur Realisierung der Regelung auf dem realen Fahrzeug wird die vorhandene mechanische Plattform um ein geeignetes Sensorsystem sowie eine Platine zur Anbindung aller elektrischer Komponenten erweitert. Die Fahrzeugzustände werden anhand der Daten einer inertialen Messeinheit durch eine Sensorfusion mit einem Komplementärfilter geschätzt. Die Ansteuerung der Sensorik und Aktorik sowie die Umsetzung des diskreten Regelkreises erfolgt auf einem Linux-basierten Mikrocontroller.

Die Regelung wird auf Basis des mathematischen Modells des Fahrzeugs ausgelegt und in einer Simulation optimiert. Anschließend werden die Ergebnisse experimentell validiert und in einem iterativen Prozess auf das reale System angepasst. Dabei werden sowohl klassische als auch modellbasierte Reglerkonzepte untersucht. Die Nickbewegung wird mit einem LQR-Regler mit Gain-Scheduling kontrolliert. Die Rollbewegung wird durch einen PID-Regler gesteuert. Durch die Vorgabe von reglerbasierten Trajektorien für die Rollbewegung kann das Verhalten des Fahrzeugs gezielt gesteuert werden. Dadurch kann entweder sichergestellt werden, dass Störungen durch aktives Schrägstellen des Fahrzeugs kompensiert werden, oder die Kopplung der Roll- und Gierbewegung wird zum Kurvenfahren genutzt. Dabei können die Grenzen klassischer Reglerkonzepte aufgezeigt und Vorschläge für den Einsatz komplexer Kontrollalgorithmen unterbreitet werden.

In verschiedenen Experimenten konnte gezeigt werden, dass der *Monowheeler* sowohl balancieren als auch gezielt und kontrolliert Kurvenfahren kann. Damit wird erstmals ein aktiv stabilisiertes Einrad dieser Größenordnung realisiert, dass den gyroskopischen Effekt nutzt um sich frei in der Ebene bewegen zu können.

Nomenclature

List of Symbols

F_F	Static friction force at the vehicle contact point	N	8, 10, 11
F_G	Weight force of the vehicle	N	8, 10, 11, 56, 57, 58, 74, 76, 83, 84
F_N	Normal force at the vehicle contact point	N	8, 10
F_{CF}	Centrifugal force due to vehicle cornering	N	10, 11, 12
F_{GP}	Weight force of the vehicle platform	N	8, 56, 57
J_G	Mass moment of inertia of the gyroscope	$\mathrm{kg\,m^2}$	10, 11, 74, 76, 83, 84
J_z	Mass moment of inertia about the z-axis	$\mathrm{kg\,m^2}$	11, 74, 83, 84
$J_{x,\tau}$	Mass moment of inertia about the x-axis relative to the contact point	$\mathrm{kg\,m^2}$	10, 11, 74, 76, 83, 84
$J_{y,\tau}$	Mass moment of inertia about the y-axis relative to the contact point	$\mathrm{kg\,m^2}$	8, 56, 57, 58
$M_{G\psi}$	Moment generated by the gyroscope about the z-axis	$\mathrm{N\,m}$	11, 12
$M_{G\varphi}$	Moment generated by the gyroscope about the x-axis	$\mathrm{N\,m}$	10, 12
S_G	Center of gravity of the vehicle		8, 10, 11
S_P	Center of gravity of the vehicle platform		8
T_a	Sampling time	s	33, 34
$\dot{\psi}_G$	Angular velocity of the gyroscope about its vertical axis	rad	10, 11, 16, 74, 76, 83
ψ	Yaw – rotation about the z-axis according to DIN 70000	rad	10, 11, 12, 74, 83, 84, 91
ψ_G	Angle of the gyroscope about its vertical axis	rad	11, 74, 76, 84
φ	Roll – rotation about the x-axis according to DIN 70000	rad	10, 11, 12, 32, 33, 34, 46, 71, 74, 76, 83, 84, 91
ϑ	Pitch – rotation about the y-axis according to DIN 70000	rad	8, 32, 33, 34, 46, 50, 56, 57, 59, 63
g	Gravitational constant	$\mathrm{m\,s^{-2}}$	32, 34, 37
h_0	Height of the center of gravity S_G at rest	m	8, 10, 11, 56, 57, 58, 74, 76, 83, 84
m_{total}	Total mass of the vehicle	kg	11, 83, 84

p	Displacement of the contact point relative to the vehicle	m	8, 14, 56, 59
r_{curve}	Instantaneous curve radius	m	11
v	Velocity in the vehicle's x-direction	$\mathrm{m\,s^{-1}}$	11, 12, 83, 84, 86, 92
w_G	Constant angular velocity of the gyroscope	rad	10, 11, 19, 20, 74, 76, 83, 84

Abbreviations

ADC	Analog to Digital Converter	20, 36, 81, 117, 118
BBB	BeagleBone Black	23, 24, 25, 28, 93, 94, 95, 98, 99, 110, 112, 113, 114, 116, 117, 119, 121, 124, 125, 137
BLDC Motor	Brushless DC Motor	13
CEMF	Counter Electromotive Force	20
CMG	Control Moment Gyroscope	3
CSV	Comma Separated Values	36, 93, 125
DLP	Digital Lowpass	40, 41, 42, 43, 44, 45, 46, 135, 136, 7, 8, 9, 10, 11, 12
E-Cap	Electrolytic Capacitor	26
EMC	Electromagnetic Compatibility	25, 127
EML	Embedded Mechatronics Laboratory	2, 93, 98, 113, 116, 117, 119
FFT	Fast Fourier Transformation	36, 39, 40, 42, 43, 45, 135, 136, 8, 10, 12
FIFO	First In First Out	117, 118
GPIO	General Purpose Input Output	23, 24, 93, 95, 117
HKA	Karlsruhe University of Applied Sciences	2
IDE	Integrated Development Environment	93
IIR	Infinite Impulse Response	103, 104, 139

IMU	Inertial Measurement Unit	23, 31, 32, 33, 34, 35, 36, 38, 39, 40, 41, 42, 43, 44, 45, 46, 50, 113, 114, 115, 125, 127, 135, 136, 137, 7, 8, 9, 10, 11, 12
LiPo	Lithium Polymer Battery	24, 25, 26, 27
LQR	Linear Quadratic Regulator	55, 56, 58, 66, 73, 76, 78, 83, 84, 127, 128, 139
mDNS	Multicast DNS	94, 121, 124
MIMO	Multiple Input Multiple Output	66
MMAP	Memory Map	110, 113, 116, 117
MPC	Model Predictive Control	4, 55, 130
PWM	Pulse-Width Modulation	20, 116, 117
RMS	Root Mean Square	35, 36
SCP	Secure Copy	94
SIMO	Single Input Multiple Output	66, 70, 71, 73, 79, 83
SISO	Single Input Single Output	66
SNR	Signal to Noise Ratio	34, 35, 36, 37, 40, 42, 43, 45, 46, 127, 137, 139
SOS	Second-Order-Section	103, 104, 139
SPI	Serial Peripheral Interface	24, 27, 93, 95, 112, 113, 114, 115, 136
SSH	Secure Shell	93, 94, 95
ToF	Time of Flight	31
TTL	Transistor-Transistor Logic	27, 28, 135
UART	Universal Asynchronous Receiver/Transmitter	23, 24, 93, 95, 112, 119
UDP	User Datagram Protocol	93, 98, 120, 121, 123, 124
ufw	Uncomplicated Firewall	95

Contents

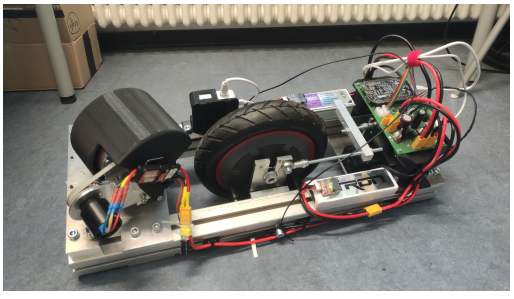
1 Introduction

With the increasing autonomy and miniaturization of mobile systems, the control of unstable, dynamic, and tightly coupled systems in robotics is becoming increasingly important. Unstable systems are found in numerous technical applications and can be stabilized and controlled using many different approaches. Systems with multiple degrees of freedom, such as the inverse 3D pendulum, are particularly complex. The challenge of controlling such systems takes on an additional practical dimension when they are used not only in stationary but also in mobile applications. To balance, single-wheeled vehicles must actively control the two degrees of freedom of roll and pitch and passively control the degree of freedom of yaw. For free movement in the plane, it must also be possible to actively control yaw, whereby the degrees of freedom are often highly coupled with each other. In most cases, such systems are underactuated, so that the coupling of the degrees of freedom must be used in a targeted manner to achieve the desired goal. This places particularly high demands on the control and system design of such vehicles.

While many mobile robot systems are inherently statically stable and capable of performing complex dynamic maneuvers, they do not require active stabilization. Conversely, classic balancing systems usually focus on actively stabilizing an unstable state, but are severely limited in their mobility. Although there are numerous concepts and demonstrators for self-balancing vehicles, there is currently only one electronically stabilized vehicle of this size that can achieve completely free movement on a plane with only one wheel. This vehicle uses a combination of a flywheel and a drive wheel, which are common concepts for stabilizing and controlling unstable systems. The aim of this work is to balance a single-wheeled vehicle, the so-called *Monowheeler*, using the gyroscopic effect and to investigate its cornering capabilities. This involves developing, testing, and presenting a novel concept for stabilizing and controlling an underactuated, unstable system with highly coupled dynamics.

The *Monowheeler* is a single-wheeled vehicle whose drive wheel is suspended from a platform. The vehicle is inspired by Charles Taylor's *One-Wheeled Vehicle* [36], who stabilizes his unicycle, which is about the size of a car, by hand. Figure 1.1b and Figure 1.1a show the Monowheeler and Charles Taylor's historical model.

(a) *One-Wheeled Vehicle* by Charles F. Taylor [22]



(b) Monowheeler

To control the unstable pitching dynamics of the *monowheeler*, the wheel can be moved relative to the platform. The equally unstable rolling motion is controlled by a gyroscope in the front of the vehicle. The concept and design of the *Monowheeler* were developed in a previous research and development project [33] at the Karlsruhe University of Applied Sciences (HKA) as part of the Embedded Mechatronics Laboratory (EML). In this thesis, the existing mechanical platform will be expanded to include a suitable sensor system, and based on this, a control system for stabilizing and steering the vehicle will be developed and implemented.

The thesis is structured as follows: Chapter 2 summarizes the current state of the art and the previous project. Chapter 3 describes and interprets the mathematical model of the *Monowheeler*. Subsequently, in Chapter 4, the existing hardware is put into operation and measured in order to incorporate the dynamic properties of the actuators into the modeling. A suitable sensor concept is developed in Chapter 5 to record the vehicle data. Chapter 6 documents the development of a suitable concept for stabilizing the vehicle. The vehicle's ability to perform dynamic maneuvers such as cornering is investigated in Chapter 7. Finally, Chapter 8 describes the integration of the hardware and the control loop on the microcontroller used.

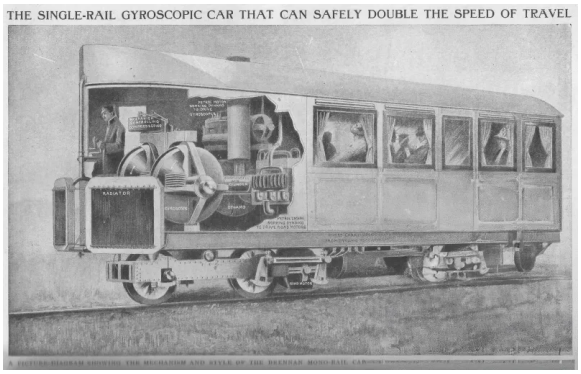
2 State of the art

Single-wheeled vehicles can be controlled in many ways. A distinction must be made between the number of degrees of freedom and the concepts for controlling these degrees of freedom. The following chapter describes the current state of the art and provides an overview of theoretical concepts and practical implementations of single-wheeled vehicles. Particular attention is paid to the use of the gyroscopic effect, as this is a central component of the concept pursued in this work.

2.1 Gyroscopic effect

The gyroscopic effect describes the behavior of a gyroscope whose axis alignment is changed. This creates a torque perpendicular to the axis around which the change occurs. This effect is used in various technical applications to actively and passively stabilize systems. One example is Louis Brennan's *monorail* from 1905, in which a rail vehicle is balanced on a rail with the aid of two gyroscopes [5]. A similar approach is taken by Pyotr Shilovsky's *Gyrocar* [35] and its further development, the *Gyro-X*. However, none of the concepts caught on.

(a) *Gyro-X* by Alex Tremulis [10]



(b) *Monorail* by Louis Brennan [37]

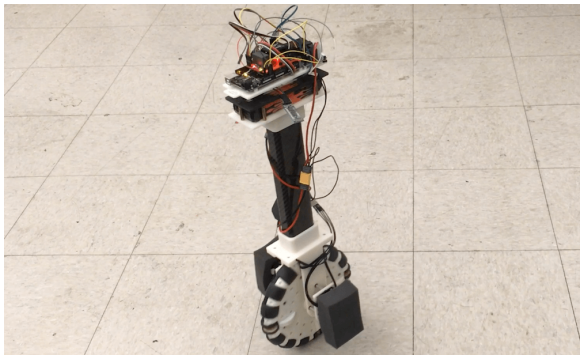
In space travel, the gyroscopic effect is successfully used in the form of Control Moment Gyroscopes (CMGs) to stabilize the position of satellites and space stations in space. The most prominent system of this type is used on the ISS and helps to control the position of the space station in an energy-efficient manner [17]. But smaller satellites also use the effect to influence their position and altitude [21][18].

In shipping, the gyroscopic effect is used for passive stabilization of ships against wind and waves. A large gyroscope is suspended in a gimbal so that the natural tendency of a gyroscope to maintain its orientation stabilizes the ship and minimizes unwanted rolling movements [30].

2.2 Single-wheel vehicles

In recent years, many concepts for stabilizing single-wheel vehicles have been implemented. Most of these use flywheels, active center of gravity shifting through drive and braking forces, or so-called omniwheels. Omniwheels consist of a large wheel whose treads can be actuated orthogonally to the alignment of the wheel. This allows such a unicycle to balance independently and move freely in a translational manner. However, the alignment of the vehicle cannot be influenced. An example of this concept was presented with the *OmBuro* in [34].

The most advanced unicycle of this type is the *Mini Wheelbot*, which was introduced in early 2025. The robot consists of two flywheels, which are used as a drive wheel and a flywheel depending on the orientation. With the help of a nonlinear Model Predictive Control (MPC) algorithm, the *Wheelbot* is able to balance and turn corners. This is the first time that a single-wheeled robot of this type has been able to fully control all degrees of freedom [14].



(a) *OmBuro* by Junjie Shen and Dennis Hong [34]

(b) *Mini Wheelbot* [27]

2.3 Single-wheeled vehicles with the gyroscopic effect

Analysis of existing concepts shows that the gyroscopic effect is rarely used as the primary means of stabilizing and controlling single-wheeled vehicles. Instead, this principle is mainly used in other vehicle classes, or completely different concepts are used.

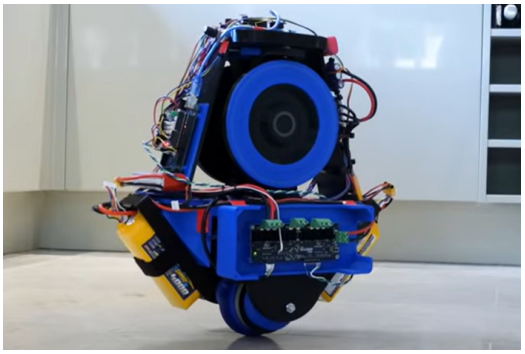


Figure 2.3: Unicycle with gyroscopes by James Bruton [19]

One of the few applications of the gyroscopic effect is in James Bruton's unicycle, which uses two gyroscopes to control the robot's rolling motion. However, this design specifically relies on a concept that eliminates the coupling between rolling and yawing by means of the gyroscope. The gyroscopes rotate in opposite directions so that their moments cancel each other out through the movements of the vehicle. The vehicle balances itself in the direction of travel through drive and braking moments. The unicycle therefore has no control authority over the direction of travel. [19].

The vehicle developed in this work is based on Charles F. Taylor's *One-Wheeled Vehicle*.

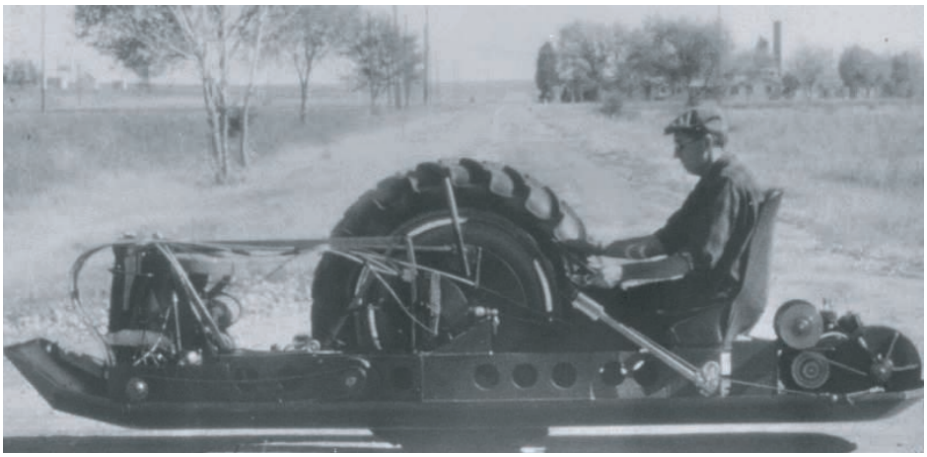


Figure 2.4: *One-Wheeled Vehicle* by Charles F. Taylor [22]

The vehicle consists of a platform with a large drive wheel in the center. A mechanism that shifts the wheel relative to the platform is used to maintain balance in the direction of travel. Rolling movements and cornering are primarily stabilized and controlled by a large gyroscope in the front part of the platform. Control is purely mechanical and requires human intervention. This vehicle can also control all degrees of freedom. However, Taylor's vehicle also highlights the limitations of the concept. Stabilization by a large gyroscope works best when driving at a constant speed, but loses its effectiveness at low speeds because the vehicle relies on the additional dynamic coupling between lateral tilt and cornering. Tight turns and dynamic maneuvers are only possible to a limited extent due to the system's high inertia, which is due to physical constraints, and the limited control authority over the direction of travel.
The basis for the approach pursued here is [33], in which Taylor's patent [36] was analyzed and a model-sized prototype was designed using modern components. While this preliminary work outlined initial control concepts for balancing the vehicle, the main objective of this work is the complete commissioning of the prototype as well as the detailed elaboration,

implementation, and redevelopment of the control strategies. In addition, the vehicle's ability to drive curves will be investigated.

3 System description

In order to stabilize a single-wheeled vehicle, the two degrees of freedom, pitch and roll, must be controlled. For free movement in the plane, control over the yaw degree of freedom is also required:

- **Pitch**: Longitudinal inclination, describes the forward-backward balance
- **Roll**: Lateral inclination, describes the left-right balance
- **Yaw**: Rotation around the vertical axis, determines the direction of travel

Since the primary goal of this work is to stabilize the *monowheeler*, a simplified model is deliberately used for the analysis and controller design, which reduces the dynamics to the motion of a 3D inverted pendulum. This model makes it possible to consider and control the essential degrees of freedom of pitch, roll, and yaw separately. With this simplification, the overall system can be divided into two independent subsystems due to the orthogonality of the directions of motion and the decoupling of the acting forces and moments in the longitudinal and transverse planes: The first describes the pitch, the second the roll and yaw. The simplified pendulum model is particularly well suited for analyzing balancing and developing control strategies, as the system dynamics remain manageable and can be interpreted directly. At the same time, it should be noted that this model does not represent the complete vehicle dynamics. Especially when cornering, additional couplings between the degrees of freedom occur that are not taken into account here. However, the dynamics of balancing are well represented and will be validated with the real system in the course of the work. The detailed description and interpretation of the two subsystems of the *Monowheelers* is provided in [33].

3.1 Pitch subsystem

The subsystem for modeling pitch can be reduced to a 2D inverse pendulum using the simplifications described above. The vehicle is assumed to be a rigid body. To stabilize the system in its unstable resting position, the contact point can be shifted relative to the vehicle's platform. It is assumed that the dynamics of the displacement process (motor torque, inertia, etc.) and the resulting geometric changes (change in mass moment of inertia, vertical displacement of the center of gravity due to mechanics, etc.) are negligible [33, vgl. S. 19f]. The movement situation can be represented as shown in Figure 3.1:

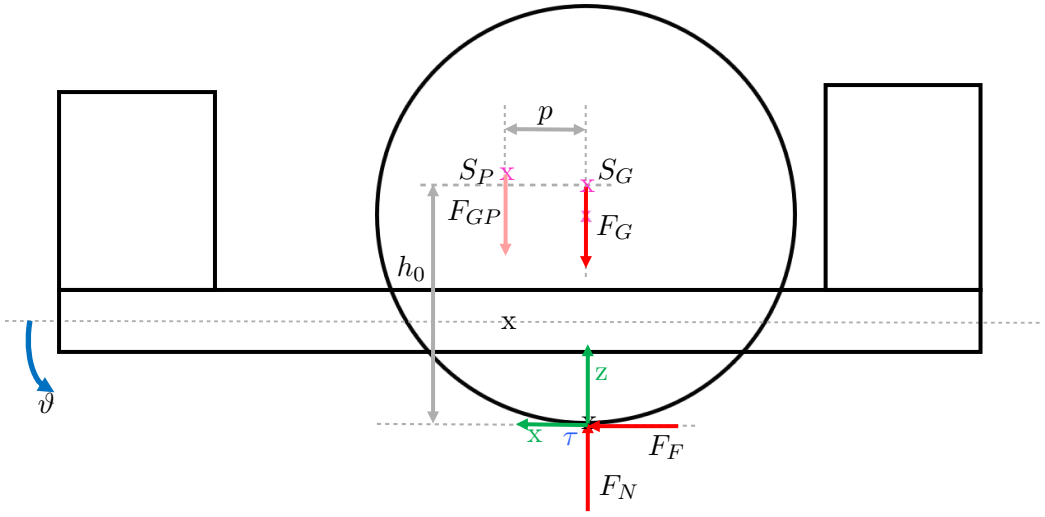


Figure 3.1: Free body diagram pitch [33, S.21]

The moment equilibrium around the moment pole τ results in the simplified differential equation [33, S. 21]:

$$J_{y,\tau} \cdot \ddot{\vartheta} = F_{GP} \cdot p + F_G \cdot h_0 \cdot \sin \vartheta \tag{3.1}$$

The pitch angle ϑ can therefore be directly influenced by the linear displacement of the contact point relative to the platform. This means that neither the platform needs to be tilted nor the vehicle accelerated to compensate for disturbances. In addition, permanent disturbances can be easily compensated for, in contrast to other concepts such as balancing with a flywheel.

3.2 Roll and yaw subsystem

The focus when stabilizing the vehicle is on controlling the roll movement. Without active countersteering, the vehicle tips over sideways. In contrast, yawing can be described as an indifferent equilibrium. Although yawing determines the direction of travel, it is not decisive for the stability of the vehicle and remains constant without external disturbance. Yawing is therefore only controlled secondarily. According to the task description, the gyroscopic effect should be used to stabilize and control the roll and yaw movements.
The subsystem for modeling roll and yaw cannot be reduced to a 2D problem, as the two movements are coupled due to the gyroscopic effect and the dynamic coupling between roll and yaw when cornering. The simplified pendulum model continues to be used, in which the *Monowheeler* is considered a rigid body. This allows the roll motion, similar to the pitch motion, to be simplified as a 2D inverse pendulum with an unstable equilibrium position. The direct influences of the gyroscope used for control and the forces acting on a rigid body when cornering are taken into account by considering yaw. However, this is still a simplified model, as other effects when cornering are not taken into account. For example, the lateral tilting of a wheel causes the wheel to follow a curved path due to the rolling constraint [43][4, S, 16ff]. Furthermore, additional moments are generated by the gyroscopic effect of the wheel. However, the dynamics relevant to balancing are also represented here and are to be validated using the real system.

3.2.1 Gyroscopic effect

The basics of the gyroscopic effect are described in detail in [33, S. 11ff]. If a rotating, rotationally symmetric gyroscope with the moment of inertia J_K and the angular velocity $\vec{\omega_K}$ is additionally rotated with an angular velocity $\vec{\omega_R}$ about another axis, a torque is generated about the axis perpendicular to both angular velocities:

$$\vec{M} = J \cdot \vec{\omega}_K \times \vec{\omega}_R \tag{3.2}$$

This principle gives rotating gyroscopes a self-stabilizing property.

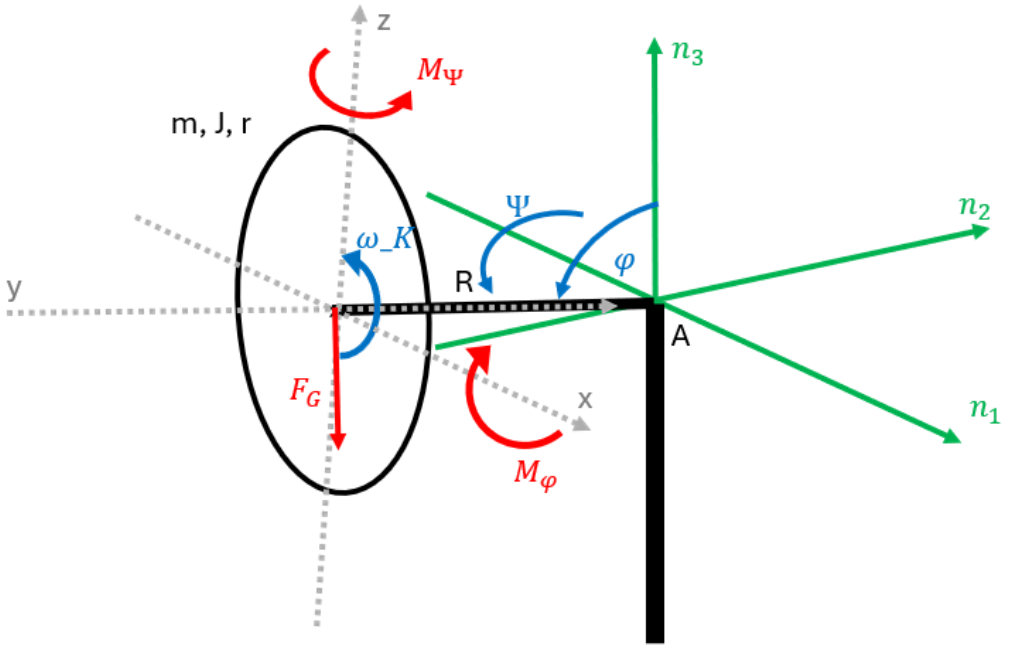


Figure 3.2: Free body diagram gyroscope as an inverse pendulum [33, S.16]

If a gyroscope is mounted on a structure resembling an inverse pendulum, it begins to tilt sideways under the influence of gravity at an angular velocity $\dot{\varphi}$. This corresponds to an additional rotation of the gyroscope and, according to Equation 3.2, generates a moment M_ψ that accelerates the gyroscope around its vertical axis. This in turn leads to a further moment M_φ, which counteracts the original tilt $\dot{\varphi}$ and, with the appropriate choice of parameters, even reverses it. In this case, the gyroscope does not fall over, but performs a combination of precession (rotation around the vertical axis) and nutation (periodic tilting movement) from [33, S. 16].

3.2.2 Application of the gyroscopic effect

The gyroscopic effect is applied to the *Monowheeler* to control and stabilize the roll and yaw movements. In order to actively influence the roll movement, the gyroscope is actuated in such a way that the targeted rotation of the gyroscope generates a moment that acts directly

on the roll movement. This allows the coupled differential equations for the simplified roll and pitch subsystem to be set up. For the time being, the system is assumed to be frictionless. Figure 3.3 shows the free-body diagram of the roll motion with influences from the yaw motion:

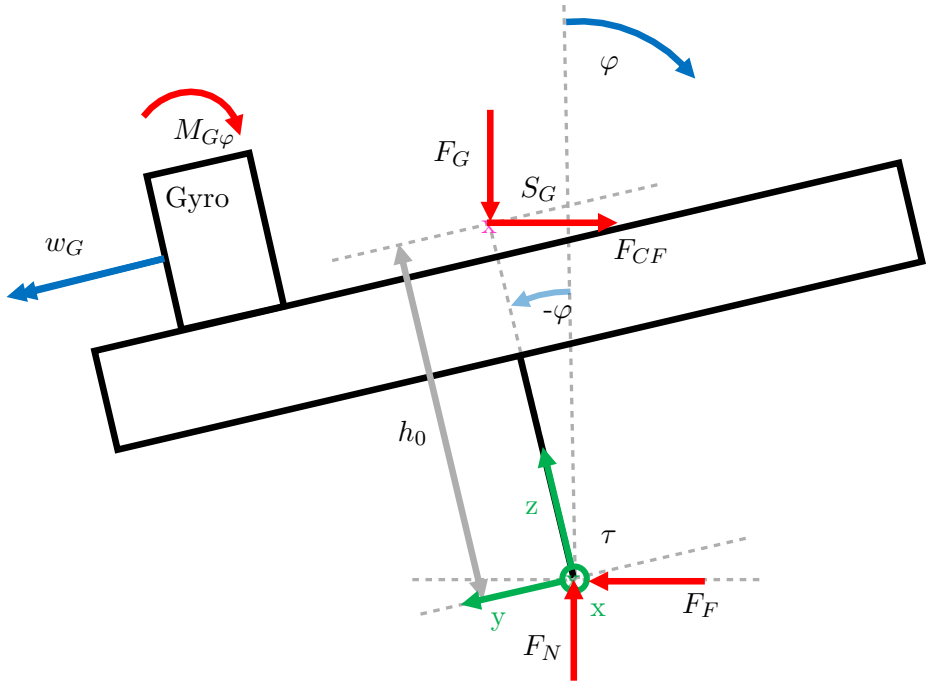


Figure 3.3: Free body diagram angle of roll [33, S.28]

The translational forces play a minor role, as the translational movement of the vehicle can be calculated based on the speed and direction of travel. Therefore, the moment equilibrium is formed around the moment pole:

$$J_{x,\tau} \cdot \ddot{\varphi} = M_{G\varphi} + F_G \cdot h_0 \cdot \sin\varphi + F_{CF} \cdot h_0 \cdot \cos\varphi \tag{3.3}$$

The gyroscopic moment $M_{G\varphi}$ can be described according to Equation 3.2:

$$M_{G\varphi} = J_G \cdot w_G \cdot (\dot{\psi} + \dot{\psi}_G) \tag{3.4}$$

Figure 3.4 shows the free body diagram of the yaw movement with influences from the roll movement. It can also be seen that the position of the gyroscope on the vehicle has no influence on the equations of motion and can therefore be freely selected in the design.

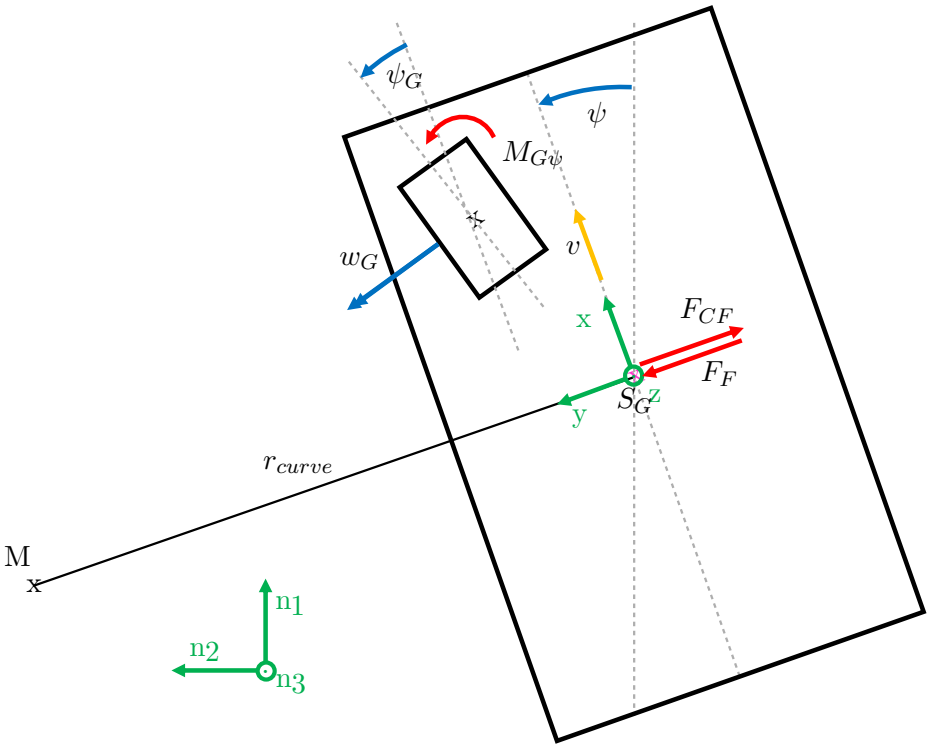


Figure 3.4: Free body diagram yaw [33, S.28]

The moment equilibrium for the yaw movement results in the following differential equation:

$$J_z \cdot \ddot{\psi} = M_{G\psi} \tag{3.5}$$

The gyroscope moment $M_{G\psi}$ can be described according to Equation 3.2:

$$M_{G\psi} = -J_G \cdot w_G \cdot \dot{\varphi} \tag{3.6}$$

The inertial force F_{CF}, also known as centrifugal force, which arises as a reaction to centripetal acceleration when cornering, can be described as follows:

$$F_{CF} = m_{total} \cdot r_{curve} \cdot \dot{\psi}^2 = m_{total} \cdot \frac{v}{\dot{\psi} \cdot \cos\varphi} \cdot \dot{\psi}^2 \tag{3.7}$$

This results in the following coupled differential equations [33, S. 28 ff]:

$$J_{x,\tau} \cdot \ddot{\varphi} = J_G \cdot w_G \cdot (\dot{\psi} + \dot{\psi}_G) + F_G \cdot h_0 \cdot \sin\varphi + m_{total} \cdot v \cdot \dot{\psi} \cdot h_0 \tag{3.8}$$

$$J_z \cdot \ddot{\psi} = -J_G \cdot w_G \cdot \dot{\varphi} \tag{3.9}$$

3.2.3 Interpretation and real system

The theoretical model developed above describes the dynamics of the system under idealized conditions. In particular, the actual contact behavior of the wheel with the ground is not taken into account, which means that in practice the model only provides a limited representation of the actual movement. The theoretical model continues to oscillate indefinitely after

the slightest excitation, and the orientation of the gyroscope cannot be changed sustainably when viewed from the world coordinate system. Due to the coupling of the differential equations, any manipulation of the roll angle has a direct reaction on the yaw movement, which in turn excites the roll. This makes it impossible to stabilize the vehicle and drive it around corners in a targeted manner. The influences of the interaction between the tire and the ground must therefore be taken into account. Three effects are particularly relevant:

- **Static friction during yawing:** Occurs when the vehicle experiences a moment about the vertical axis but does not turn due to sufficient adhesion. In this area, the coupling between roll and yaw motion is eliminated, so that the roll motion can be manipulated without affecting the yaw motion as long as the roll speed is so low that the resulting gyroscopic moment $M_{G\psi}$ is smaller than the static friction.
- **Sliding friction during yawing:** Occurs when the applied yaw moments exceed the adhesive force and the wheel begins to rotate around the vertical axis. This effect leads to energy losses and has a damping effect on the yawing motion, preventing the vehicle from oscillating.
- **Sliding friction during rolling:** Lateral relative movement in the contact zone during rolling, caused by friction with the ground and internal deformation work in the tire. Acts as a damping effect on the rolling motion and reduces vibrations.

These effects are partially included in the modeling. The parameters are selected so that the behavior of the model approximates the expected behavior of the prototype as closely as possible. Factors such as the ground surface, the type of tire, temperature, humidity, dust and dirt, tire pressure, etc. make a theoretical determination of the friction parameters impractical within the scope of this work. Therefore, the model is used to validate and develop concepts for regulation and control, but these must then be adapted to the real system, or the controller must be robust enough to compensate for the modeling inaccuracies. Combining the findings from theoretical modeling with the influences of the interaction between tires and the ground results in the following system behavior: If the speed v of the vehicle is zero or if the vehicle is traveling at a speed v other than zero and is close to the rest position ($\varphi \approx 0$ und $\dot{\varphi} \approx 0$), the static friction around the vertical axis is so high that normal operation of $\dot{\psi} = 0$ can be assumed. This means that the rolling motion depends only on the tilting moment caused by falling over and the actuated gyroscopic moment.
If the vehicle is traveling at a speed v and is not in the rest position, the self-stabilizing effect of the gyroscope and the dynamics of cornering come into play. The vehicle begins to tilt, causing the gyroscopic moment $M_{G\psi}$, in combination with the unmodeled effects of cornering, to trigger a rotation around the vertical axis. The vehicle is cornering and both the centrifugal force F_{CF} and the gyroscopic moment $M_{G\varphi}$, both of which are caused by the yawing motion, counteract the original tilting.

4 Hardware Integration

The existing hardware comprises a prototype with reduced functionality and all components required to build a fully functional vehicle. The mechanical design of the *Monowheelers* is fully documented in [33]. This chapter describes how the individual components are combined to form a complete overall system.

4.1 Actuators

The actuators of the *Monowheeler* comprise two *Dynamixel XH540-W150-T/R* servo motors, which are used to control the pitch, roll, and yaw movements, an Brushless DC Motor (BLDC Motor), which drives the gyroscope at a constant speed, and a drive motor, which accelerates and brakes the vehicle in the direction of travel.

4.1.1 Dynamixel servo motors

The two Dynamixel motors used are each responsible for controlling and regulating one of the two subsystems defined in Chapter 3. Both Dynamixel motors are controlled via the *RS485* protocol. Four mechanisms are used to prevent damage to the motors and hardware in the event of a fault:

- **Soft stops**: TPU stops are used for both Dynamixel to limit the angular deflection in order to cushion the force peaks in the event of a collision and protect the remaining hardware.
- **Internal overload protection**: The Dynamixel motors have internal overload protection. If the required current becomes too high, for example because the Dynamixel motor hits the TPU stop, the Dynamixel motor switches off automatically.
- **Software check**: In position mode, maximum and minimum positions can be specified which the Dynamixel motor cannot exceed. In Velocity Control Mode, the software on the microcontroller checks in each iteration whether the maximum position is exceeded.
- **Internal Watchdog**: If the Dynamixel does not receive any new commands for too long (100 ms), it switches itself off and can only be reactivated by restarting. This also deactivates the motors in the event of a software crash (e.g., due to a segmentation fault).

In order to model the dynamics of the control variable, the Dynamixels are characterized in the following section and approximated as PT1 elements with the following form:

$$G(s) = \frac{Y(s)}{U(s)} = \frac{K}{T \cdot s + 1} \tag{4.1}$$

All PT1 fits are performed using the Python script *actuator_modelling.py*.

Wheel Dynamixel

The Dynamixel for the pitch movement is used in position control mode, since the control variable is the position of the contact point relative to the platform. The displacement of the contact point is achieved by means of a four-bar linkage mechanism. Since the mechanism does not have a linear relationship between servo angle and displacement due to design constraints, a third-degree polynomial is used to convert the variables. The polynomial is calculated using the Python script *4barlinkage.py* or *4barlinkageinverse.py*. The calculation rule is documented in [33, S. 59ff]. The signs are chosen according to the convention of the direction of rotation of the servo motor and the definition of the displacement p.
The Dynamixel has an internal PID controller that can be parameterized. The best results for the proportional and differential components can be achieved with $K_P = 800$, $K_D = 50$. Higher values lead to constant oscillations and overshoots, while lower values impair the dynamics. To measure the control variable, a displacement p is specified and the actual behavior is recorded. The setpoint is based on the realistic working range in controller operation, estimated from the simulation. The results of a measurement can be seen in Figure 4.1.

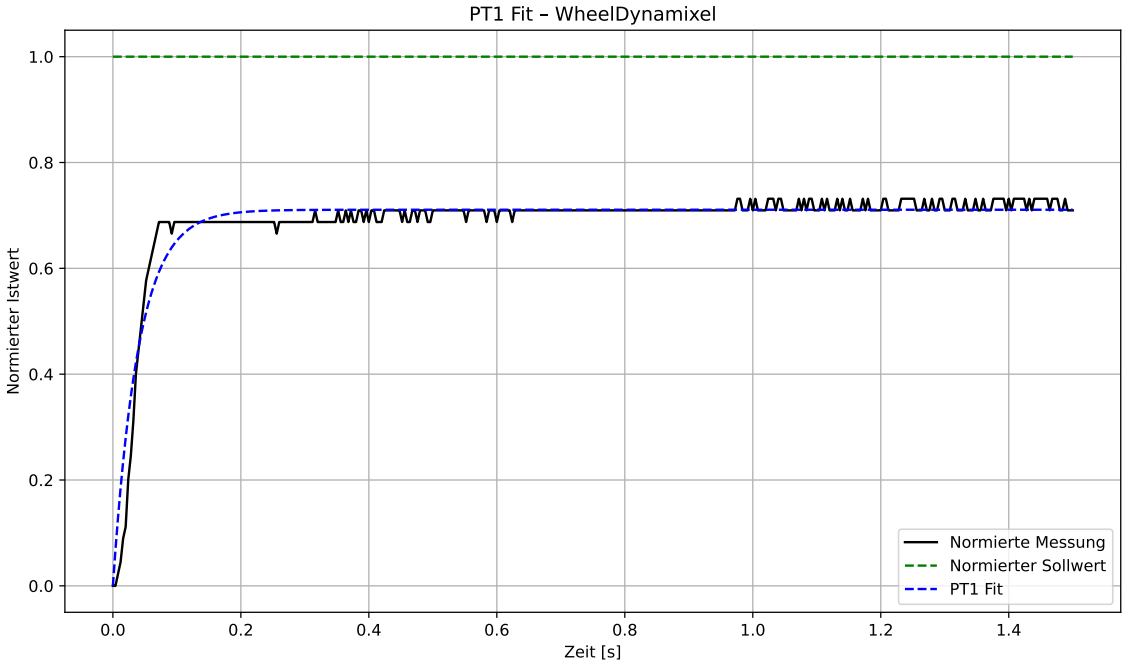


Figure 4.1: Wheel Dynamixel PT1 Fit 10% setpoint jump, $K_P = 800$, $K_D = 50$, $K_I = 0$

This results in the following values for the PT1 fit (Table 4.1):

Table 4.1: PT1-Fit Wheel Dynamixel with $K_P = 800$, $K_D = 50$, $K_I = 0$

Set-point step	K	T
2 %	0.429	0.048
10 %	0.711	0.040
30 %	0.889	0.044

The behavior shown in Figure 4.1 corresponds more to a PT2 element with a dead time in the order of 5 ms. Table 4.1 also shows that the parameters of the PT1 fit vary greatly depending on the setpoint. However, a PT1 fit with averaged values for K and T adequately represents the essential dynamics and enables a simple and robust description of the dynamics of the control variable. The low gain is also noticeable, especially with small setpoint jumps. To remedy this, an integral component is introduced (see Figure 4.2):

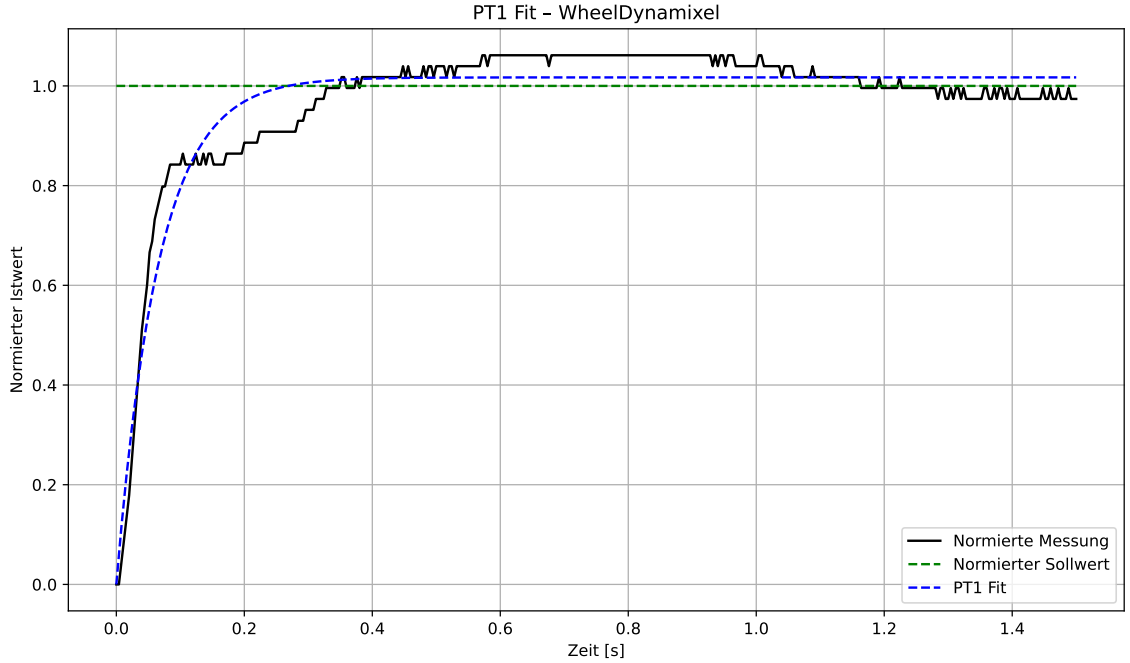


Figure 4.2: Wheel Dynamixel PT1 Fit 10% setpoint jump, $K_P = 800$, $K_D = 50$, $K_I = 1000$

This results in the following values for the PT1 fit (Table 4.2):

Table 4.2: PT1-Fit Wheel Dynamixel with $K_P = 800$, $K_D = 50$, $K_I = 1000$

Set-point step	K	T
2 %	1.127	0.243
10 %	1.017	0.066
30 %	0.994	0.042

Basically, the gain has increased for small setpoint jumps of less than 0.5 to approx. 1. However, in Figure 4.2 it can be seen that the real system first jumps to the same final

value as without the integral component, then remains there until the integral component begins to take effect, before the actual value moves to its final position after an overshoot. In reality, however, the setpoint changes constantly, so that the integral component hardly improves accuracy because the time constant is too large. In favor of dynamics, the integral component is therefore omitted and the parameters $K_P = 800$, $K_D = 50$, $K_I = 0$ are used. The parameters $K = 0.85$ and $T = 0.045$ are selected as PT1-Fit.

Gyro Dynamixel

The Dynamixel for the roll and yaw movement is used in Velcoity Control Mode, as the control variable is the angular velocity of the gyroscope $\dot{\psi}_G$. Here, too, a four-bar linkage mechanism is used, but the control variable corresponds exactly to the negative angular velocity of the Dynamixel. The sign is taken into account in the software. Internally, the Dynamixel controls the speed with a PI controller. First, the control system is measured with the preset PI parameters of the Dynamixel. To do this, several jumps are made in the range of the expected control variable on the system. An example of a step response can be seen in Figure 4.3.

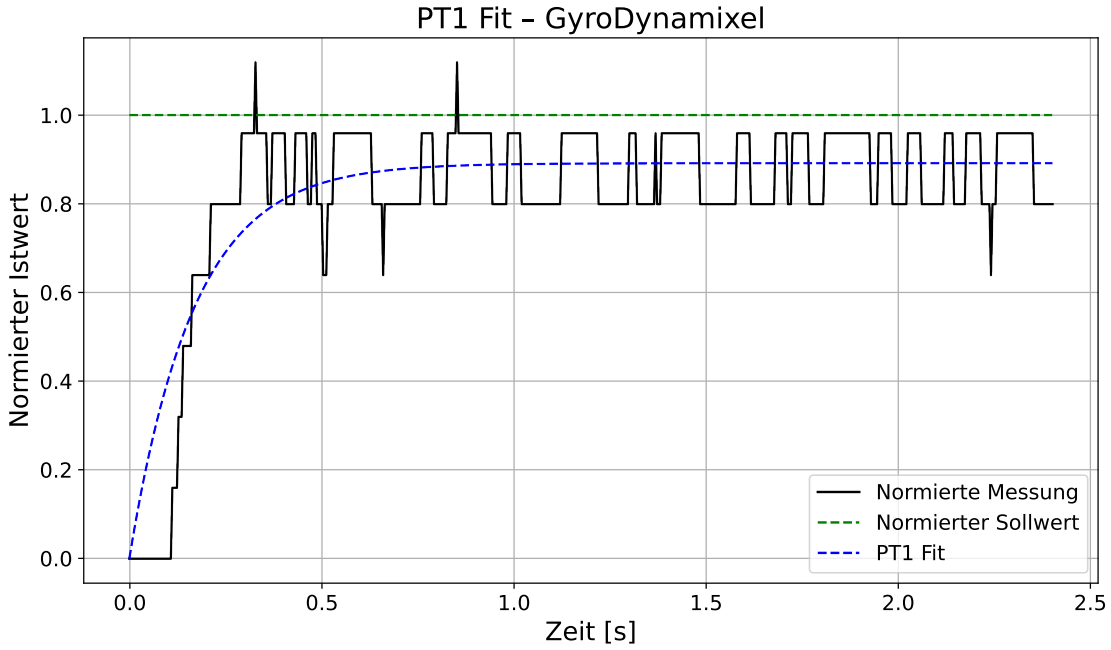


Figure 4.3: Gyro Dynamixel PT1 Fit 3% setpoint jump, $K_P = 100$, $K_I = 1980$

Table 4.3 shows the PT1-Fit results of the gyro Dynamixel for $K_P = 100$, $K_I = 1980$:

Table 4.3: PT1-Fit Gyro Dynamixel with $K_P = 100$, $K_I = 1980$

Set-point step	K	T
0,5 %	0.537	0.585
1 %	0.743	0.414
2 %	0.860	0.223
3 %	0.891	0.168
5 %	0.946	0.139
10 %	0.970	0.111

Table 4.3 shows that the Dynamixel becomes significantly more accurate (K closer to 1) and faster (T smaller) with larger jumps. However, since most of the control takes place in the lower range of the speed spectrum, the controller parameters are increased to $K_P = 500$, $K_I = 5000$ (example step response in Figure 4.4):

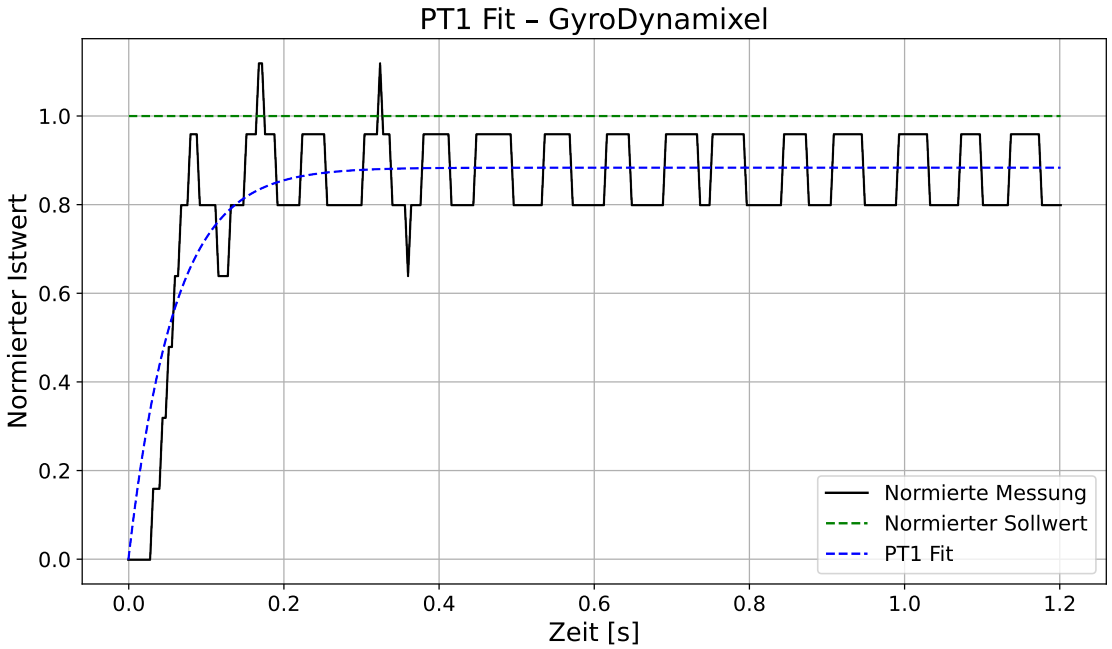


Figure 4.4: Gyro Dynamixel PT1 Fit 3% setpoint step, $K_P = 500$, $K_I = 5000$

In Table 4.4, the PT1-Fit results of the gyro Dynamixel for $K_P = 500$, $K_I = 5000$ can be seen:

Table 4.4: PT1-Fit Gyro Dynamixel with $K_P = 500$, $K_I = 5000$

Set-point step	K	T
0,5 %	0.471	0.168
1 %	0.702	0.122
2 %	0.883	0.079
3 %	0.884	0.058
5 %	0.911	0.051
10 %	0.939	0.045

The results in Table 4.4 show a significant improvement in the time constants, especially for smaller jumps. However, the dynamics deteriorate for large jumps. In Figure 4.5, it can be seen that the Dynamixel begins to oscillate with larger setpoint jumps:

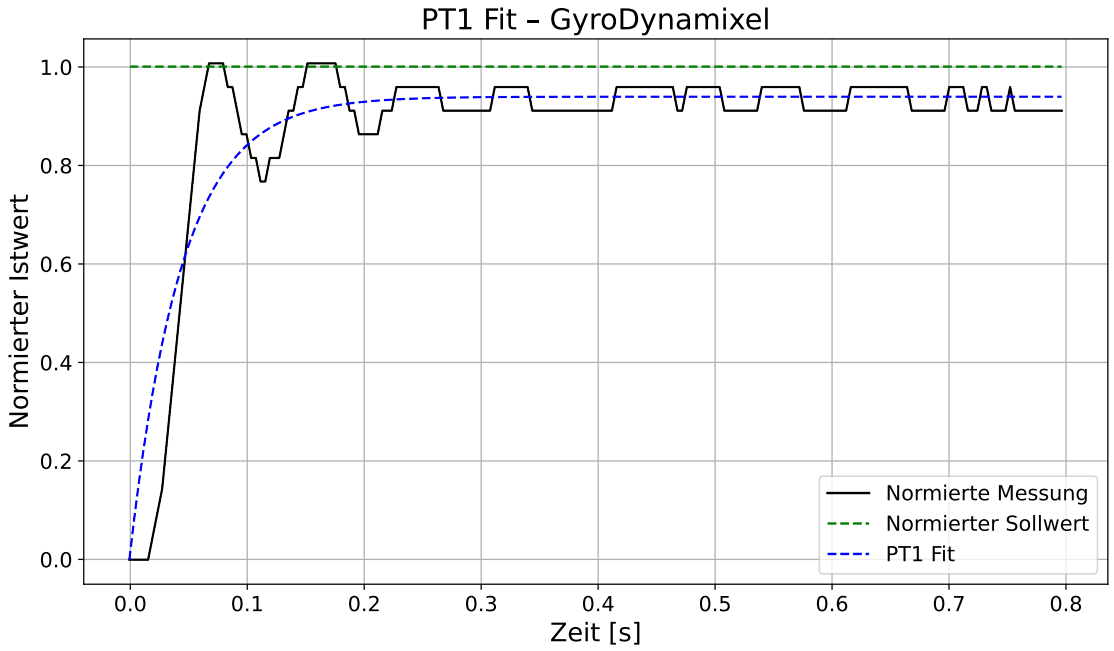


Figure 4.5: Gyro Dynamixel PT1 Fit 10% setpoint jump, $K_P = 500$, $K_I = 5000$

Therefore, the control parameters are reduced until a compromise between fast dynamics and a stable system is achieved at $K_P = 300$, $K_I = 3000$. As in Figure 4.6, the system does not oscillate even with the largest jump tested:

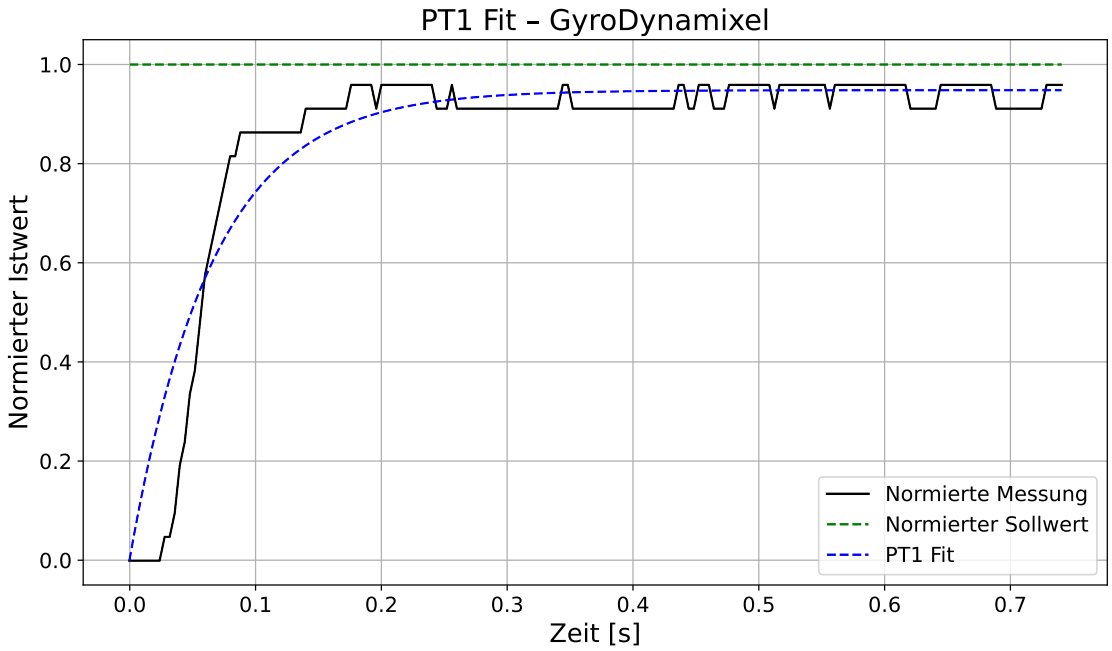


Figure 4.6: Gyro Dynamixel PT1 Fit 10% setpoint jump, $K_P = 300$, $K_I = 3000$

In Table 4.5, the PT1-Fit results of the Gyro Dynamixel for $K_P = 300$, $K_I = 3000$ can be seen:

Table 4.5: PT1-Fit Gyro Dynamixel with $K_P = 300$, $K_I = 3000$

Set-point step	K	T
0,5 %	0.481	0.379
1 %	0.742	0.217
2 %	0.852	0.133
3 %	0.886	0.100
5 %	0.916	0.078
10 %	0.948	0.065

With the adjusted control parameters, the track is now significantly faster than with the default settings. Similar to the Wheel Dynamixel, the measurements show that the behavior corresponds more to a PT2 element with dead time and that the parameters vary greatly depending on the setpoint specification. However, a PT1 fit with averaged values for K and T adequately represents the essential dynamics and enables a simple and robust description of the dynamics of the control variable. For simulation and modeling, a gain of $K = 0.85$ and a time constant of $T = 0.130$ are selected.

4.1.2 Gyro Motor

In order to provide sufficient torque for regulating and controlling the roll and yaw movement, the gyroscope must rotate at least at a constant angular velocity $w_G = 4000\,\text{rpm}$ [33, S. 6].

For this purpose, an *MEGA Motor ACn 22/20/3* [24] with a belt drive is used [33, S. 73]. An *Jeti Spin 66 pro opto* [20] is used as the motor driver [33, S. 73], which is configured with the appropriate *Jeti Box*.

The driver separates the control signal from the motor currents using an optocoupler and requires its own 5 V power supply in addition to the power supply for the motor. The driver is used as a sensorless speed controller that determines the speed of the motor based on the Counter Electromotive Force (CEMF), which only works reliably when a load is applied and above a minimum speed.

Control is via an Pulse-Width Modulation (PWM) signal with 50 Hz. Before operation, the driver must be activated by setting an PWM duty cycle slightly below (approx. 0.1 ms − 0.2 ms) the value for the minimum speed. Acceleration is achieved by means of a ramp. When starting up for the first time after activation, this ramp is applied to the entire acceleration process. When restarting from a standstill, the ramp only takes effect from the set minimum speed, which can lead to overloading of the belt drive and the power supply. Therefore, the software must ensure that the driver is completely switched off after each run.

All experiments in this work are performed at a gyroscope speed of $w_G = 5000\,\mathrm{rpm}$. This speed is a compromise between energy efficiency and gyroscopic torque, estimated using the simulation study in [33, S. 37ff].

4.1.3 Drive Wheel

The drive wheel is an e-scooter wheel motor *Xiaomi M365 Essential 1S Pro*. The motor has 15 pole pairs, an approximate speed constant $K_V = 22\,\mathrm{min^{-1}\,V^{-1}}$, and a torque constant $K_M = 0.52\,\mathrm{N\,m\,A^{-1}}$ [33, S. 71].
The drive wheel is controlled by an *Maxon Escon 70/10*, which determines the position of the rotor using the built-in Hall sensors. The driver is configured so that the setpoint can be specified between 10 Hz and 5 kHz via an PWM signal [23, S. 21]. The motor also has an enable signal to switch the motor on and off and an analog output that outputs the current speed. It is important to note that the analog voltage must be configured to the maximum speed so that the signal never exceeds or falls below the valid voltage range of the microcontroller's Analog to Digital Converter (ADC) converter.
As already mentioned, the driver uses the motor's built-in Hall sensors to determine the rotor position. However, the motor's behavior is inconsistent and the motor jerks during operation. When the motor is powered, the sensor signals are so noisy that it is impossible to accurately determine the position of the rotor (see Figure 4.7):

Figure 4.7: Hall sensor signals drive wheel under load, unmodified

The cables for the motor phases and the Hall sensors are factory-twisted together in a single unshielded cable. To reduce electromagnetic interference caused by inductive coupling, the sensor lines are routed as far away from the motor phases as possible. The time-varying magnetic field in the motor phase induces a current in the signal lines, and the resulting interference voltage is proportional to the impedance of the conductor:

$$\vec{u} = \vec{Z} \cdot \vec{i} \tag{4.2}$$

To reduce the impedance and thus also the interfering voltage, external 1 kΩ pull-up resistors are installed in parallel with the internal 2.7 kΩ pull-up resistors in the [23, S. 16] driver. The impedance of the parallel resistors is purely real and can be determined as follows:

$$R_{ges} = \frac{R_1 \cdot R_2}{R_1 + R_2} = \frac{2.7\,\text{k}\Omega \cdot 1\,\text{k}\Omega}{2.7\,\text{k}\Omega + 1\,\text{k}\Omega} \approx 730\,\Omega \tag{4.3}$$

It should be noted that the voltage source of the Hall sensors can only supply a limited amount of current. The following current is required per sensor:

$$I = \frac{U}{R_{ges}} = \frac{5\,\text{V}}{730\,\Omega} \approx 7\,\text{mA} \tag{4.4}$$

The voltage source of the driver supplies a maximum of 30 mA and a maximum of two Hall sensors can switch simultaneously. This means that the maximum current required is less than the maximum current available. These measures reduce noise to such an extent that the motor control functions reliably (see Figure 4.8):

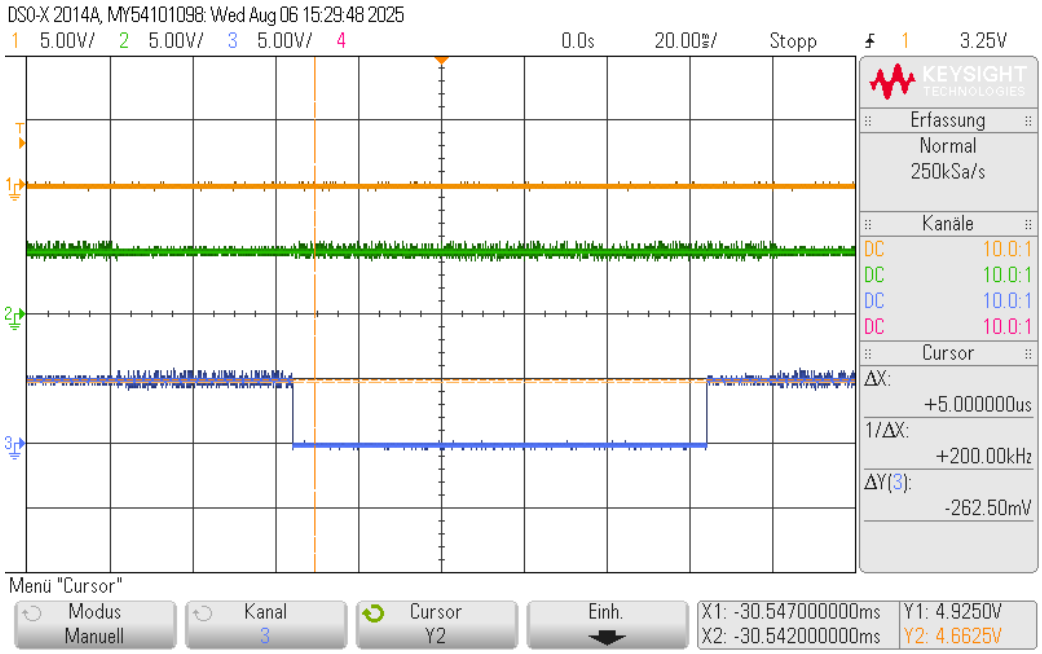


Figure 4.8: Hall sensor signals drive wheel under load, separate cable routing and impedance matching

Since the Hall sensors only provide six states per revolution, the internal speed control of the motor driver only works reliably from approx. 30 rpm (3 Pulse/s) onwards. Without an acceleration ramp, the torque during acceleration to the minimum speed is too high for the vehicle. However, if a ramp is used, the motor jerks when starting due to the low resolution of the sensors. The torque can be specified very precisely, even at very slow speeds. The motor driver also offers the option of outputting the current speed as an analog voltage. This voltage value is accurate and continuous even at very low speeds. Therefore, the motor is operated in current control mode with an external PID speed controller.

4.2 Electronics

The electronics perform two central tasks: the transmission and processing of measurement and control signals between sensors, actuators, and microcontrollers, and the power supply for all components. A revised version is being designed based on the circuit board developed in [33, S. 88ff]. This will correct the following errors from the first iteration of the circuit board:

- Control signal connection for gyro motor: Instead of an enable, PWM, and direction pin, the *Jeti Spin 66 pro opto* driver requires an 5 V voltage source, a ground connection, and a PWM pin.
- connection control signal drive wheel: Instead of an enable, PWM, and direction pin, the *Maxon Escon 70/10* driver requires a ground connection, a PWM pin, and an enable pin.

- power supply BeagleBone Black (BBB): Connect the 5 V supply voltage to the pins *VDD_5V* (P9.5 and P9.6) instead of the pins *SYS_5V* (P9.7 and P9.8).
- Addition of a USB port for supplying power to the WLAN module and adjustment of the DC voltage converter.
- Faulty Universal Asynchronous Receiver/Transmitter (UART)-RS485 protocol circuit: The selected IC for converting the UART and RS485 protocol signals operates with 5 V voltage levels, while the BBB operates with 3.3 V. The existing circuit board has a faulty circuit for converting the voltage levels, which can cause impermissible voltages at pins P9.3, P9.4, P9.11, P9.13, P9.24, and P9.26, leading to the destruction of an BBB.

While the components used—such as microcontrollers, WLAN modules, and actuators—remain unchanged, the internal connection is modified to correct the previously identified errors in the existing circuit board and improve the electrical properties.

4.2.1 Overview of Components

Table 4.6 lists all components and the associated interfaces that must be connected by the circuit board. The selection of components is described in [33].

Table 4.6: Overview of Electrical Components and Connections

Component	Power Supply	Control Signals
Maxon Escon 70/10	24 V, max. 15 A	Ground, PWM (3.3 V), Enable (3.3 V) Analog Ground, Power (1.8 V)
Jeti Spin 66 pro opto	24 V, max. 70 A	Ground, PWM (3.3 V), 5 V-Pin
TP-Link AC750-WLAN-Router	5 V, max. 2 A	-
BBB	5 V, max. 2 A	see pin reference in Table 4.6
ICM20948 (IMU)	-	Ground, 3.3 V-Pin, SPI (3.3 V)
2x*Dynamixel XH540-W150-T/R*	12 V, max. 5 A	RS485-Protocol

Table 4.7 documents the connections of the General Purpose Input Output (GPIO) of the BBB:

Table 4.7: Pinout BBB

Function	Hardware-Module	Pin - Num.	GPIO
Power Supply Input			
GND	-	P9.1 und P9.2	-
5 V	-	P9.5 und P9.6	-
Power Supply Output			
GND	-	P9.1 und P9.2	-
3.3 V	-	P9.3 und P9.4	-
Dynamixel XH540-W150-T/R 1			
RX	UART4	P9.11	30
TX	UART4	P9.13	31
Dynamixel XH540-W150-T/R 2			
TX	UART1	P9.24	15
RX	UART1	P9.26	14
Maxon Escon 70/10			
Enable	-	P9.15	48
PWM	PWM1_0	P9.14	50
Analog GND	ADC_GND	P9.34	-
Analog Out	ADC_AIN4	P9.33	-
Jeti Spin 66 pro opto			
PWM	PWM1_1	P9.16	51
ICM20948			
MISO (shared)	SPI1	P9.29	111
MOSI (shared)	SPI1	P9.30	112
SCLK (shared)	SPI1	P9.31	110
CS0 (Sensor 1)	SPI1_0	P9.20	12
CS1 (Sensor 2)	SPI1_1	P9.19	13

4.2.2 Functional structure and design guidelines

The circuit board is divided into two sections, corresponding to the two tasks: signal processing and transmission (signal electronics) and power supply to the components (power electronics). The signal electronics include two Serial Peripheral Interface (SPI) interfaces for the sensors, one connection each for the gyro motor and the drive wheel, two UART RS485 converters for the two Dynamixel servo motors, and the connection of the BBB. The power electronics offer two connections for a six-cell Lithium Polymer Battery (LiPo) and a three-cell LiPo, two 24 V connections for the gyroscope motor and the drive wheel, two 12 V connections for the Dynamixel servo motors, and a DCDC converter on 5 V for the power supply of the WLAN router, the BBB, and the receiver of the *Jeti Spin 66 pro opto*. An overview of all components and connections is illustrated in the functional structure in Figure 4.9:

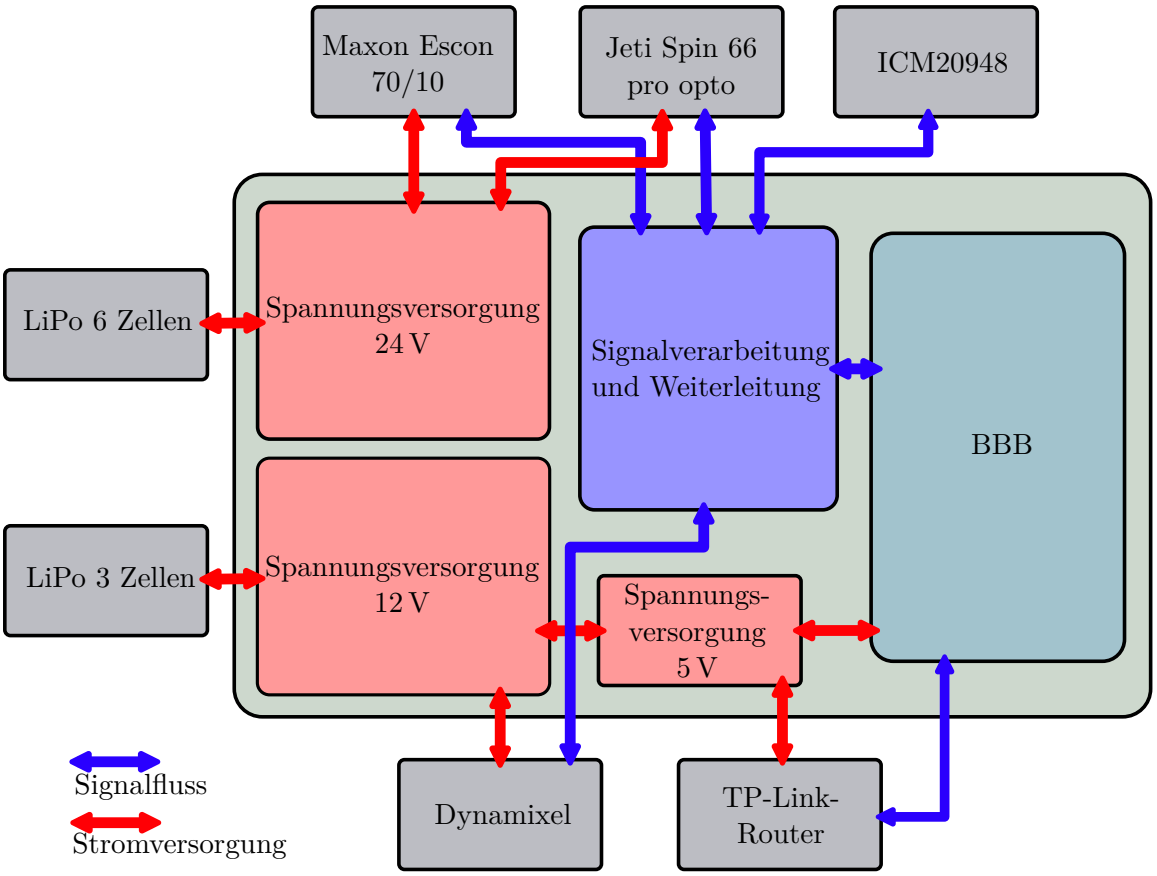


Figure 4.9: Functional Structure Electronics

To ensure clean signal routing and stable power supply, the circuit board is designed in accordance with the following design guidelines [3]:

- Circuit board: The circuit board is designed as a two-layer printed circuit board.
- Layout: The high currents and rapid current changes of the motor drivers generate strong electromagnetic fields that can interfere with sensitive signal lines. The high return currents via the ground can lead to undesirable voltage drops and thus to an undefined or noisy reference potential. To optimize the Electromagnetic Compatibility (EMC) and ensure high signal quality, the power electronics and signal electronics are spatially separated.
- Ground: Two ground planes are used, which are connected at only one defined point. The separation ensures that the return currents of the power electronics do not affect the signal ground. The connection ensures a common reference potential. The use of ground planes also reduces the impedance of the return current path, which minimizes ground noise, signal reflections, and loop formation. The defined connection of the ground planes should also have the lowest possible impedance and be placed so that the return current path for the supply voltage of the signal electronics is as short as possible.

- Cable routing: Wires should not interrupt the ground planes if possible. The width of current-carrying conductor tracks is adjusted according to the expected load current. For very high currents, copper planes should be used instead of simple wires to reduce power loss, thermal resistance, and voltage drops. Right angles are avoided due to possible signal reflections.[3, S. 16].
- Smoothing capacitors: To suppress high-frequency interference, ceramic capacitors with a capacity of 100 nF are used at the outputs of DC-DC converters, at each supply pin of the integrated circuits, and at the power supply connectors. These are positioned as close as possible to the respective source of interference and connected to ground via short return paths to increase their effectiveness.
- support capacitors: To keep the supply voltage stable, even during rapid load changes and fluctuations in the LiPos, large support capacitors (Electrolytic Capacitors (E-Caps)) are used for each voltage rail to serve as energy buffers. A defined discharge path is provided to ensure that the E-Caps can be safely discharged without any connected consumers.
- Thermal relief: All solder pads should have thermal relief spokes to improve solderability.

4.2.3 Power electronics

The following section describes the design of the power electronics based on the design guidelines and requirements described above. The external connection is made with XT60 connectors. In order to be able to disconnect the LiPos from the rest of the board via a switch, a high-side switch circuit (similar to [33, S. 89]) is used (see Figure 4.10):

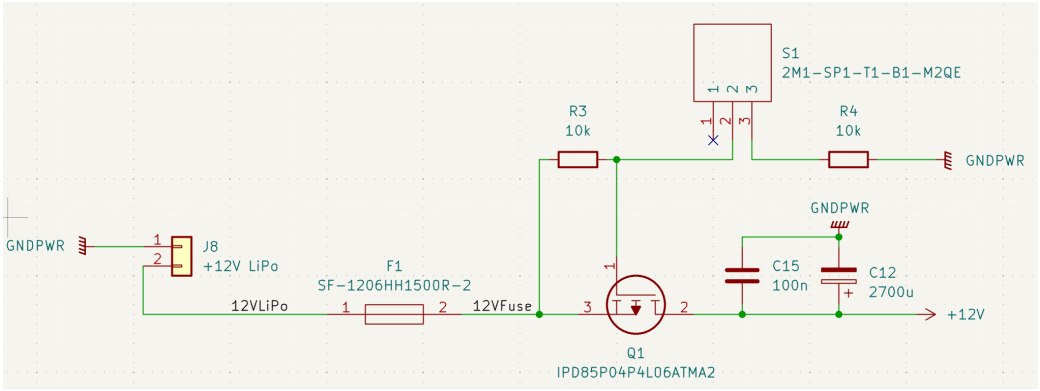


Figure 4.10: High-side switch circuit Power supply 12 V

A fuse is installed directly after the input of the LiPos to protect the circuit board and components from damage. The transistor is a P-channel MOSFET [16], which conducts or blocks between source and drain depending on the gate-source voltage. If the gate-source voltage is greater than approx. −1.7 V, the MOSFET blocks between source and gate. Above this threshold, the conductivity of the transistor increases as the gate-source voltage decreases. From approximately −5 V - −10 V, the MOSFET can be assumed to be a

conductor [16, S. 2, S. 5].

The source is connected to the LiPo. The gate-source voltage can now be changed via a switch with a voltage divider with R_3 and R_4. If the switch is open, resistor R_3 is not connected to ground and no voltage drops. This means that the gate-source voltage is $V_{GS} = 0\,\mathrm{V}$ and the transistor blocks. When the switch is closed, a voltage drops across resistor R_3, so that the gate-source voltage becomes negative and the transistor conducts. The voltage divider is designed so that the gate-source voltage when the switch is closed is between $-5\,\mathrm{V}$ and $-10\,\mathrm{V}$ for both maximum and minimum LiPo charge levels. A smoothing capacitor and a support capacitor are installed at the drain output to ensure the most stable power supply possible.

The 12 V power supply and the 24 V power supply are very similar in design. The voltage dividers for switching on and off are adapted to the respective voltage level, and two MOSFETs are used in parallel in the 24 V power supply to comply with the thermal specifications [33, S. 89f].

The 5 V power supply is implemented with a DC-DC converter that is fed by the 12 V power supply. Smoothing and support capacitors are used at both the input and output. An overview of the entire circuit diagram can be found in Appendix A.

4.2.4 Signal electronics

This section describes the design of the signal electronics based on the design guidelines and requirements described above. The external connection is made with JST-XH connectors. Both the SPI connections for the sensors and the connections for the motor drivers require hardly any additional circuitry. Apart from the smoothing capacitors, the only additional component used is a pull-down resistor for the enable pin of the *Maxon Escon 70/10* to ensure a defined state at all times.
In order to avoid the need for level converters when converting from Transistor-Transistor Logic (TTL) to RS485 protocol, a transceiver that can work with 3.3 V logic is selected. In addition, the transceiver should have an auto-direction feature to simplify the implementation of communication. The *THVD1426* transceiver [38] meets these requirements and has a sufficiently high data rate. The transceiver is installed in accordance with the [38, S. 18] data sheet (see Figure 4.11):

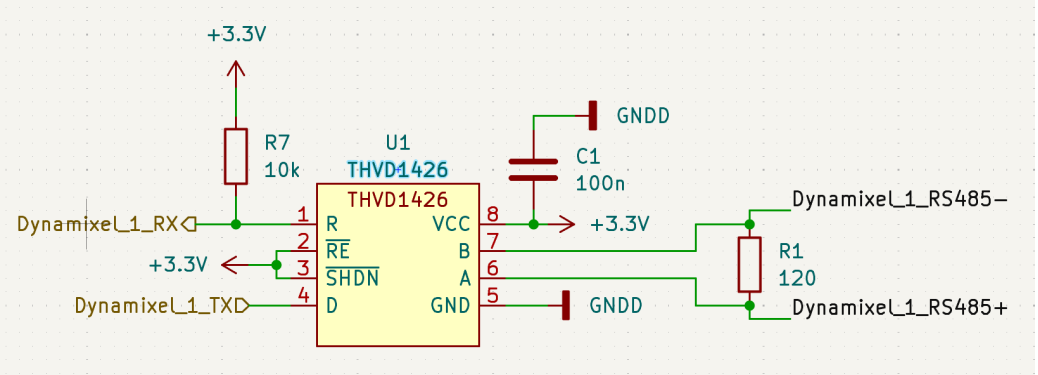


Figure 4.11: Circuit for converting TTL-RS485 protocol

Since the RS485 protocol is a differential signal with a typical line impedance of 120 Ω, terminating resistors must be provided at both ends of the lines [32]. Such a resistor is therefore installed on the circuit board. An overview of the entire circuit diagram can be found in Appendix A.

4.2.5 Layout

The layout of the circuit board can be seen in Figure 4.12.

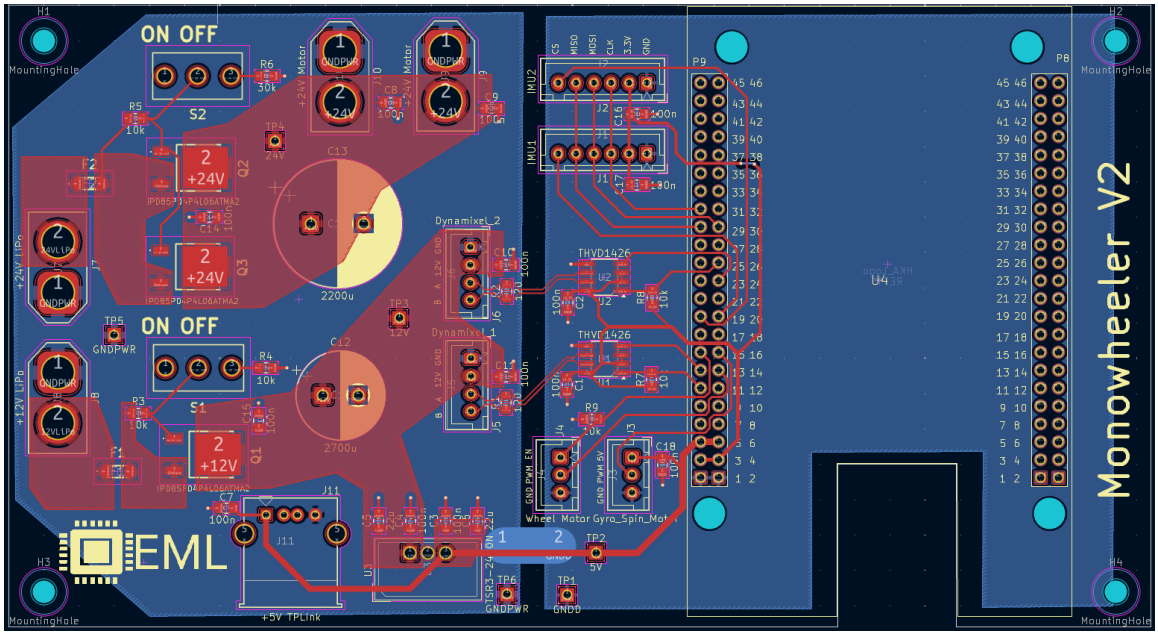


Figure 4.12: Circuit board layout

The power electronics are located on the left side and the signal electronics on the right side. The two ground planes are connected at one point by a thick copper track, directly below the conductor that carries the 5 V power supply for the BBB. In addition, copper pads are

installed that serve as measuring pads or as soldering points should additional cables be needed or corrections be necessary.

4.2.6 Results

All design guidelines and requirements were implemented. The components can be controlled successfully and reliably, there are no thermal problems, and the power supplied is sufficient. No terminating resistor has been installed on the receiver side of the RS485 bus at the Dynamixels, but there are irregular communication problems that may be resolved by using the second terminating resistor at the Dynamixels. The errors occur so rarely that no negative effect on the behavior of the vehicle can be determined. A picture of the finished vehicle can be seen in Figure 4.13:

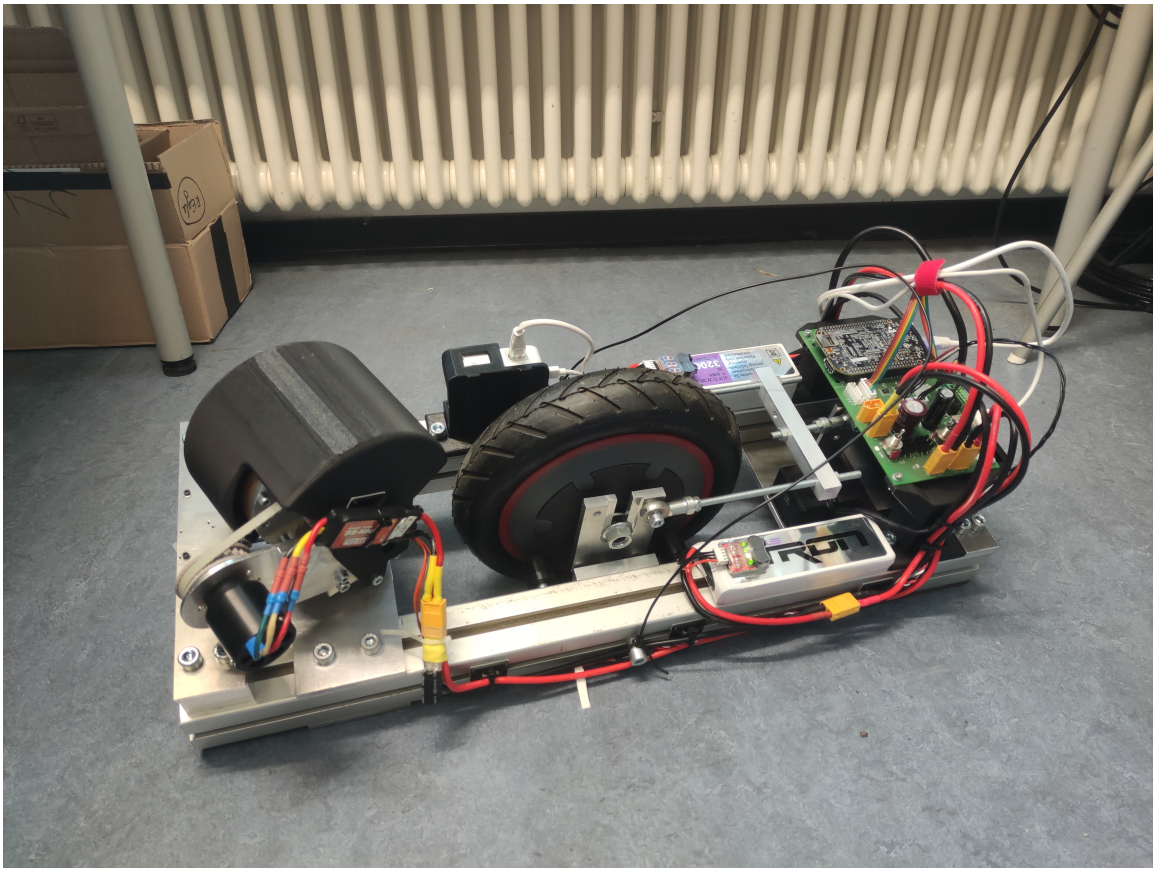


Figure 4.13: Monowheeler construction

5 Sensor concept and signal processing

The quality of sensor data plays a key role in controlling a balancing vehicle. Since this is an unstable system, the quality of the controller depends directly on the accuracy, consistency, and temporal availability of the measured variables. Inaccuracies, delays, or noise in the sensor data can significantly impair stability and even lead to loss of balance. For this reason, a robust sensor concept is required that records the relevant physical variables and processes them using suitable signal processing methods so that they can be reliably used for the control algorithms. Both the selection of suitable sensor types and the implementation of filtering and fusion methods play an important role here.

For balancing vehicles, the current position in space is the relevant physical variable. Orientation is crucial for balancing, and translational movements are also relevant in some cases. There are essentially two types of sensors that are suitable for use on a mobile vehicle to determine these variables:

- Inertial Measurement Units (IMUs): IMUs, also known as inertial sensors, measure accelerations (accelerometer) and angular velocities (gyroscope). The angular velocities can thus be measured directly via the sensors, while the angles can be determined by integration or by a suitable combination of the sensor data. IMUs operate independently of most environmental factors such as light, sound, and ground surface, function independently of their position in space, and can capture all six degrees of freedom. In addition, IMUs are inexpensive, small, can be mounted anywhere on the vehicle, and enable high sampling frequencies of 9 kHz [15, S. 60] and above. The major disadvantage of these sensors is their high susceptibility to vibrations.
- Optical/acoustic sensors: These include, for example, distance sensors such as ultrasonic, infrared, or lidar sensors, as well as camera systems. They enable the absolute position of the vehicle to be determined via geometric measurements, such as distance to the ground or to objects. This allows orientation to be determined without integration or other algorithms. This also makes these sensors very robust against vibrations. The disadvantages of optical sensors are their dependence on environmental conditions such as ground conditions or lighting, possible angle restrictions due to the sensors' field of view, the lack of direct measurement of angular velocity, and the limited ability to detect rotations around the vertical axis (yaw). Camera systems also require special reference objects in the immediate vicinity. Conventional distance sensors such as Time of Flight (ToF) sensors also only achieve low sampling frequencies of approx. 50 Hz, beyond which accuracy begins to decline significantly [1].

Due to the advantages and disadvantages of the two sensor types described above, IMUs should be used.

5.1 sensor fusion

IMUs measure the accelerations (accelerometer) and angular velocities (gyroscope) for each axis. The coordinate system of an IMU is fixed to the body, i.e., it rotates with the sensor and thus with the vehicle, so that the values are available directly in the reference coordinate system of the *Monowheelers*. The angular velocities can thus be measured directly. The yaw angle is not relevant for balancing, but the pitch and roll angles must be determined. The angles can be determined in two ways: geometrically via the direction of gravitational acceleration using the accelerometers, or via the integration of the gyroscopes.

Since gravitational acceleration always has the same orientation in the world coordinate system, it can serve as a reference for determining position using the accelerometers. The orientation can be determined, for example, using the Euler angle convention and a complete 3D rotation matrix in order to be able to capture free 3D rotations for all arbitrary angles [39, S. 6f]. In the present system, the angles are constructively limited to $\pm 5°$ for pitch and $\pm 20°$ for roll, and in normal operation, only angles of $\approx \pm 3°$ occur for both axes. For a complete representation of free rotation, all three accelerometer axes are also included. Section 5.2 also shows that the quality of the sensor values of the accelerometer of the z axis is extremely poor in the relevant range. Therefore, the pitch and roll angles in this work are determined directly from the x and y components of the accelerometers:

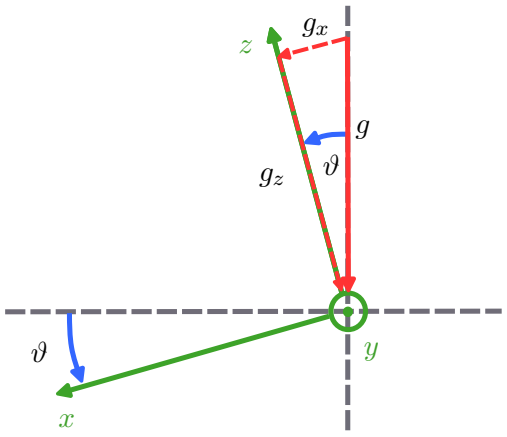


Figure 5.1: Determination of the pitch angle based on gravitational acceleration

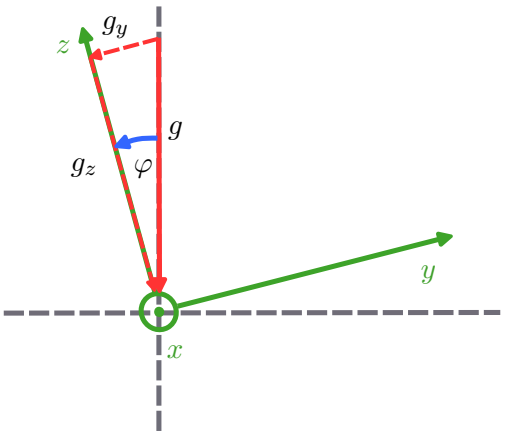


Figure 5.2: Determination of the roll angle based on gravitational acceleration

From Figure 5.1, the relationship for calculating the pitch angle follows:

$$a_x = g_x = g \cdot \sin \vartheta \tag{5.1}$$

$$\vartheta = \arcsin \frac{a_x}{g} \tag{5.2}$$

Similarly, the relationship for calculating the roll angle follows from Figure 5.2:

$$a_y = -g_y = -g \cdot \sin \varphi \tag{5.3}$$

$$\varphi = -\arcsin \frac{a_y}{g} \tag{5.4}$$

The following calculation rule applies for calculating the angles from the discrete sensor data of the gyroscopes:

$$\vartheta = \vartheta_{old} + \dot{\vartheta} \cdot T_a \tag{5.5}$$

$$\varphi = \varphi_{old} + \dot{\varphi} \cdot T_a \tag{5.6}$$

Both sensor types have specific strengths and weaknesses. Accelerometers measure not only gravitational acceleration, but also acceleration caused by rotational and translational movements, which leads to errors in angle estimation. However, they provide stable values over the long term, as no integration is required [39, S. 5]. Gyroscopes, on the other hand, respond very well to fast movements and accurately capture the dynamics of the vehicle. However, they are susceptible to drift, as integration errors gradually accumulate [39, S. 4f].

To estimate orientation accurately and robustly, the two types of sensors can be combined. The goal of this sensor fusion is to leverage the strengths of each sensor to compensate for the weaknesses of the other. Gyroscopes compensate for the short-term, dynamic errors of accelerometers, and accelerometers correct for the long-term drift of gyroscopes [42]. There are numerous methods for fusing accelerometer and gyroscope data, ranging from simple complementary filters and Kalman filters to advanced prediction and optimization algorithms such as the Madgwick filter [12, S. 20ff]. Due to the very sluggish system behavior caused by high inertia and the severely limited angular ranges, a simple complementary filter is used in this system. More complex fusion algorithms such as advanced Kalman filters or optimization methods offer no significant advantage for the present conditions.
The number and positioning of the IMUs allow the accelerometer data to be partially corrected by using geometric relationships to calculate rotational acceleration components. However, this is not necessary in the present system: due to the inertial behavior of the system, the accelerations caused by vehicle rotation are low. The translational accelerations also remain small due to the slow driving maneuvers. Both the rotational and translational accelerations, which distort the angle estimation from the accelerometer data, are filtered by the sensor fusion and can therefore be neglected. A single IMU is sufficient for determining the orientation.

The structure of a complementary filter can be seen in Figure 5.3:

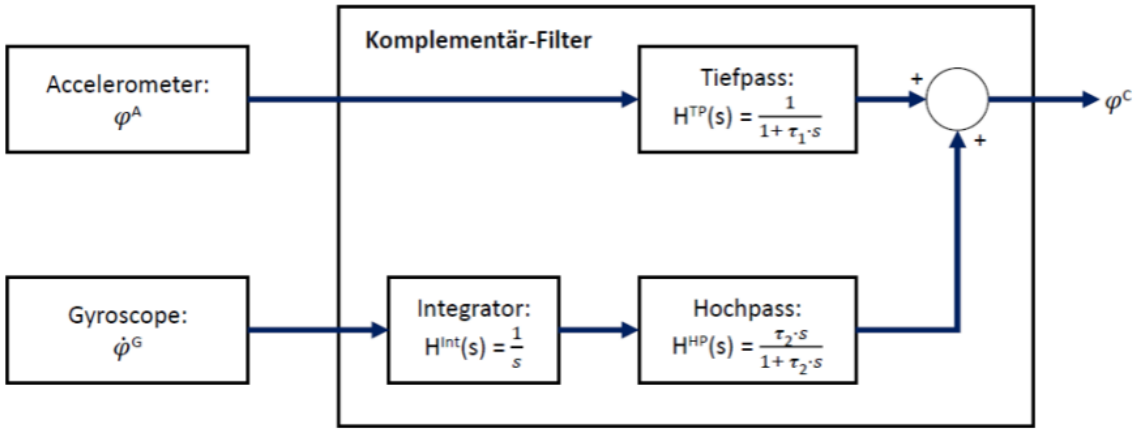


Figure 5.3: Structure of complementary filter [42, S. 84]

The angle is calculated from the accelerometer data using the calculation rule from Equation 5.1 or Equation 5.3. This angle is filtered with a low-pass filter to filter out the dynamic influences of rotational and translational accelerations, so that only the long-term orientation is calculated based on the acceleration due to gravity. The gyroscope data is integrated into an angle and then filtered with a high-pass filter to eliminate drift and map only the dynamic movements. These two variables are then combined to form the actual angle. If the cut-off frequencies for the high-pass and low-pass filters are set to be the same, the overall transfer function of the filter is 1, which means that no phase shift occurs [42, S. 85]. The discrete calculation for the pitch and roll angles is obtained with Equation 5.1 and Equation 5.3 to [42, S. 85]:

$$\vartheta = \alpha \cdot (\vartheta_{old} + \dot{\vartheta} \cdot T_a) + (1 - \alpha) \cdot \arcsin \frac{a_x}{g} \tag{5.7}$$

$$\varphi = \alpha \cdot (\varphi_{old} + \dot{\varphi} \cdot T_a) - (1 - \alpha) \cdot \arcsin \frac{a_y}{g} \tag{5.8}$$

. The coefficient α determines the cutoff frequency for the high-pass and low-pass filters and thus the frequency ranges in which the accelerometer or gyroscope data dominates. $f = 0.16\,\text{Hz}$ is selected as the cutoff frequency because the signal quality of the gyroscopes is significantly better than that of the accelerometers (see Section 5.2), allowing dynamic components of the signal to be captured more effectively. This allows α to be determined:

$$\alpha = \frac{\tau}{\tau + T_a}, \text{ mit } \tau = \frac{1}{2 \cdot \pi \cdot f} \tag{5.9}$$

5.2 Signal characterization and frequency analysis

In order to evaluate the suitability of the IMU sensor data for control purposes, a detailed analysis of its signal quality is required. In particular, the Signal to Noise Ratio (SNR) and the spectral properties of the measured variables should be examined. By examining the frequency range using Fourier transformation, characteristic signal components and dominant noise components can be identified. This analysis enables a well-founded assessment of the

measurement quality and at the same time provides a basis for the design of suitable filters that suppress disturbing influences.

The IMU data consists of three components:

$$\text{data} = \text{Nutzsignal} + \text{Mechanisches Rauschen} + \text{Sensorrauschen} \tag{5.10}$$

The ratio between the power of the useful signal and the power of the noise, also known as SNR, is decisive for the signal quality. Since the power of a signal is proportional to the square of its Root Mean Square (RMS), the SNR can also be calculated using the RMS. The SNR is typically specified on a logarithmic scale and can be determined as follows for a discrete signal x of length N [29, S. 219]:

$$SNR = 10 \cdot \log_{10} \frac{P_{Signal}}{P_{Rauschen}} \text{dB} = 20 \cdot \log_{10} \frac{RMS_{Signal}}{RMS_{Rauschen}} \text{dB, mit:} \tag{5.11}$$

$$RMS = \sqrt{\frac{1}{N} \cdot \sum_{i=0}^{N-1} x_i^2} \tag{5.12}$$

It is not possible to make a general statement about the SNR at which a signal is usable or good, as this varies depending on the application. In principle, a high value is better, and for most applications, $SNR = 10\,\text{dB}$ is the lower limit; for more sensitive systems, values of $SNR = 30\,\text{dB}+$ are sometimes required.

5.2.1 Calibration of the sensors

Before the IMU can be used, the sensors must be calibrated. The actual values can be determined from the raw data as follows:

$$data = scale \cdot (data_{raw} - offset) \tag{5.13}$$

The scaling factors are taken from the manufacturer's data sheet [15] and are accurate enough without additional calibration. The offsets are determined by aligning the IMU with the xy plane parallel to the ground and performing a measurement. For all signals that should measure zero as a measured variable in this position, the offset can simply be determined from the mean value of the measurement series. Only the z-accelerometer must additionally take into account the offset of the gravitational acceleration. The calculation is performed in the Python script *calib.py*. In addition, the orientation of the sensor axes must be adjusted to the orientation of the coordinate system of the *Monowheelers*.

5.2.2 Experimental Signal Analysis

An experiment is performed to analyze the signal quality. To do this, the vehicle is excited manually without active control in the system-relevant frequency range with system-relevant amplitude, and the data from the IMU is recorded.
The primary source of noise is the mechanical vibrations caused by the flywheel mass for

the gyroscopic effect. This has a constant speed of 5000 rpm (see Chapter 6). The frequency of these disturbances can be determined as follows:

$$f_v = \frac{1}{60\,\mathrm{s}} \cdot 5000\,\mathrm{rpm} \approx 83\,\mathrm{Hz} \tag{5.14}$$

Since this disturbance is in a much higher frequency range than the useful signal, which is below 15 Hz [33, S. 105ff], it can be assumed that the frequency of the disturbances represents the highest frequency occurring in the signal. The IMU *ICM20948* selected in [33] samples the internal ADC converters of the accelerometers with 4.5 kHz [15, S. 12] and those of the gyroscopes with 9 kHz [15, S. 11]. If no internal filter is used, new values are also provided at this frequency. The sampling theorem (Nyquist-Shannon) is therefore complied with on the IMU side, as the sampling frequency is many times greater than the highest frequency occurring in the signal. Since sampling must be performed at least twice the frequency of the highest frequency occurring in the signal in order to reconstruct a signal without errors, the IMU is sampled at a sampling frequency of 500 Hz. This allows possible interference at twice (166 Hz) and three times (249 Hz) the fundamental frequency of the original 83 Hz to be detected and practical effects such as non-ideal filters to be compensated for.

The data is stored in an Comma Separated Values (CSV) file and evaluated using the Python script *imu_signal_analyzer.py*. The useful signal and noise must be extracted from the measured signal for the calculation of SNR according to Equation 5.11. For this purpose, the measured signal is filtered with an Fast Fourier Transformation (FFT). After transformation into the frequency domain, all frequency components above the cutoff frequency are eliminated. The back transformation into the time domain then yields the useful signal. Listing 1 shows the implementation of the FFT filter:

```
1  def lowpass_filter_fft(signal, sampling_rate, cutoff_freq_hz):
2      n = len(signal)
3      fft_vals = np.fft. fft(signal)
4
5      A = int(n/sampling_rate*cutoff_freq_hz)
6      B = n-A
7      fft_vals[A:B] = 0
8
9      filtered_signal = np.fft.ifft(fft_vals). real
10     return filtered_signal
```

Listing 1: FFT low-pass filter

The cutoff frequency is based on the relevant frequencies of the signals. A lower cutoff frequency of $f_{cutoff} = 2\,\mathrm{Hz}$ is selected for the accelerometer data, since in principle only the orientation relative to the acceleration due to gravity is relevant (see Section 5.1). A higher cutoff frequency of $f_{cutoff} = 10\,\mathrm{Hz}$ is selected for the gyroscope data, as the gyroscopes are intended to detect dynamic changes (see Section 5.1). The noise can then be calculated by subtracting the useful signal from the measured signal. The RMS of the signal and noise can then be calculated according to Equation 5.11. Before this, the DC component of the signals is eliminated to avoid distortion due to static offsets, as only the temporal changes in the signal are of interest. This allows the SNR to be determined according to Equation 5.11 (see Listing 2):

```python
def calc_RMS(signal):
    signal = signal - np.mean(signal)
    return np.sqrt(np.mean(signal**2))

def calc_SNR(signal_reference, signal_raw):
    rms_signal = calc_RMS(signal_reference)
    noise = signal_raw - signal_reference
    rms_noise = calc_RMS(noise)
    SNR = 20*np.log10(rms_signal/rms_noise)
    return SNR
```

Listing 2: SNR Determination in Python

The Figure 5.4 shows an excerpt from the experiment. Even from this view, it is clear that the signal quality is poor. In the accelerometer data, the movements of the vehicle itself are barely visible in the reference signal, and the noise is sometimes as high as $\pm g$, which corresponds to an angular deviation of $\pm 90°$. The gyroscope data is somewhat better, but noise still predominates here as well. This is also reflected in the SNR values, which can be seen in Table 5.1. Values around $SNR = -20\,\mathrm{dB}$ for the accelerometer data and $SNR = 0\,\mathrm{dB}$ for the gyroscope data show that the signals are unusable without a filter.

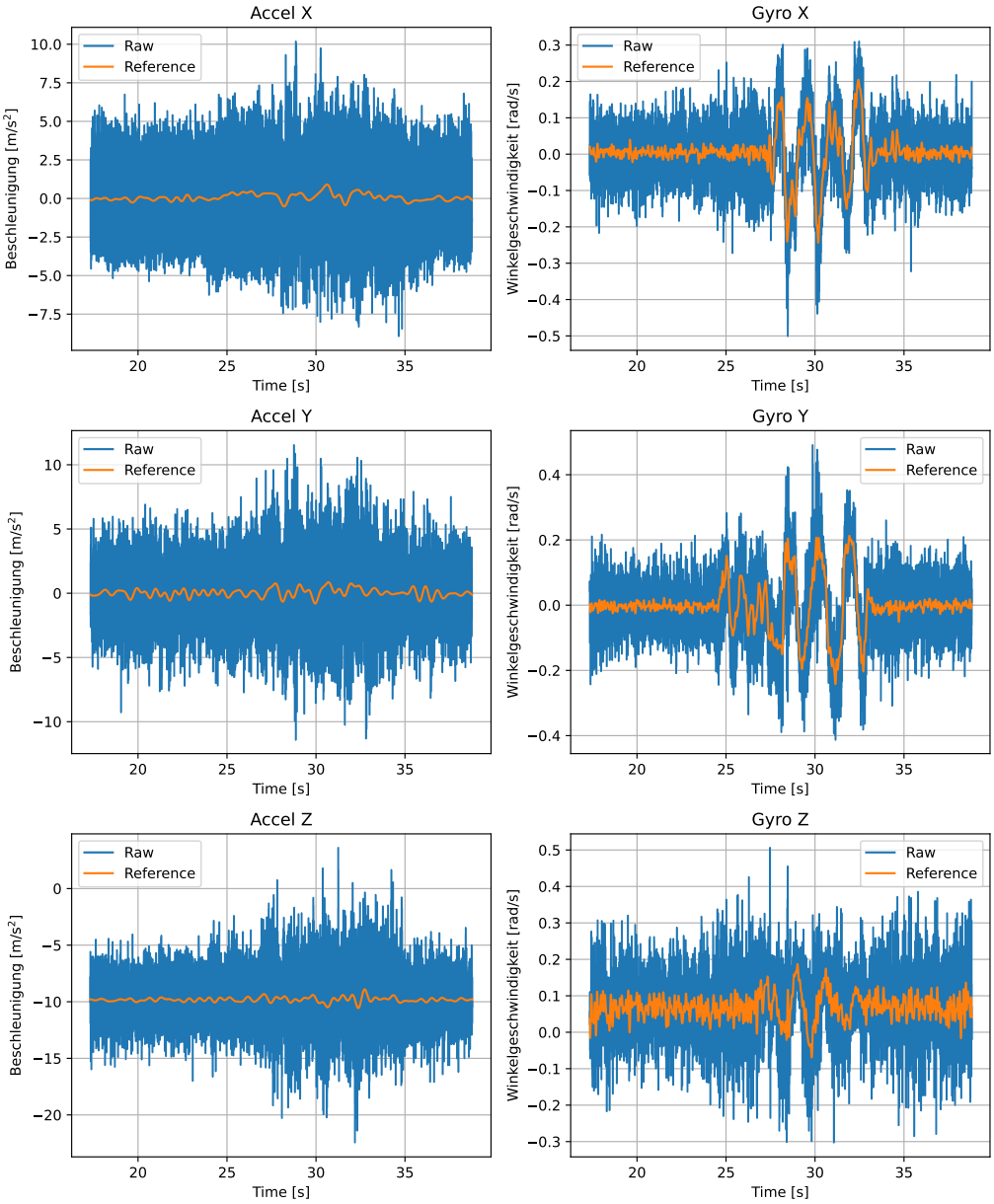


Figure 5.4: IMU Raw data

A frequency analysis is performed to identify the type of interference and to obtain information for the filter design. The calculation is also performed in the Python script *imu_signal_analyzer.py* (see Listing 3):

```python
def calc_fft_norm(signal, sampling_rate):
    n = len(signal)
    signal = signal - np.mean(signal)
    fft_vals = np.fft.fft(signal)
    fft_abs = np.abs(fft_vals)
    fft_abs = np.abs(fft_vals) / n
    fft_abs = fft_abs[:n//2]
    fft_abs[1:-1] *= 2
    frqs = np.linspace (start=0, stop=sampling_rate, num=n+1)
    frqs = frqs[:n//2]
    return frqs, fft_abs
```

Listing 3: FFT analysis in Python

The results of the FFT analysis can be seen in Figure 5.5:

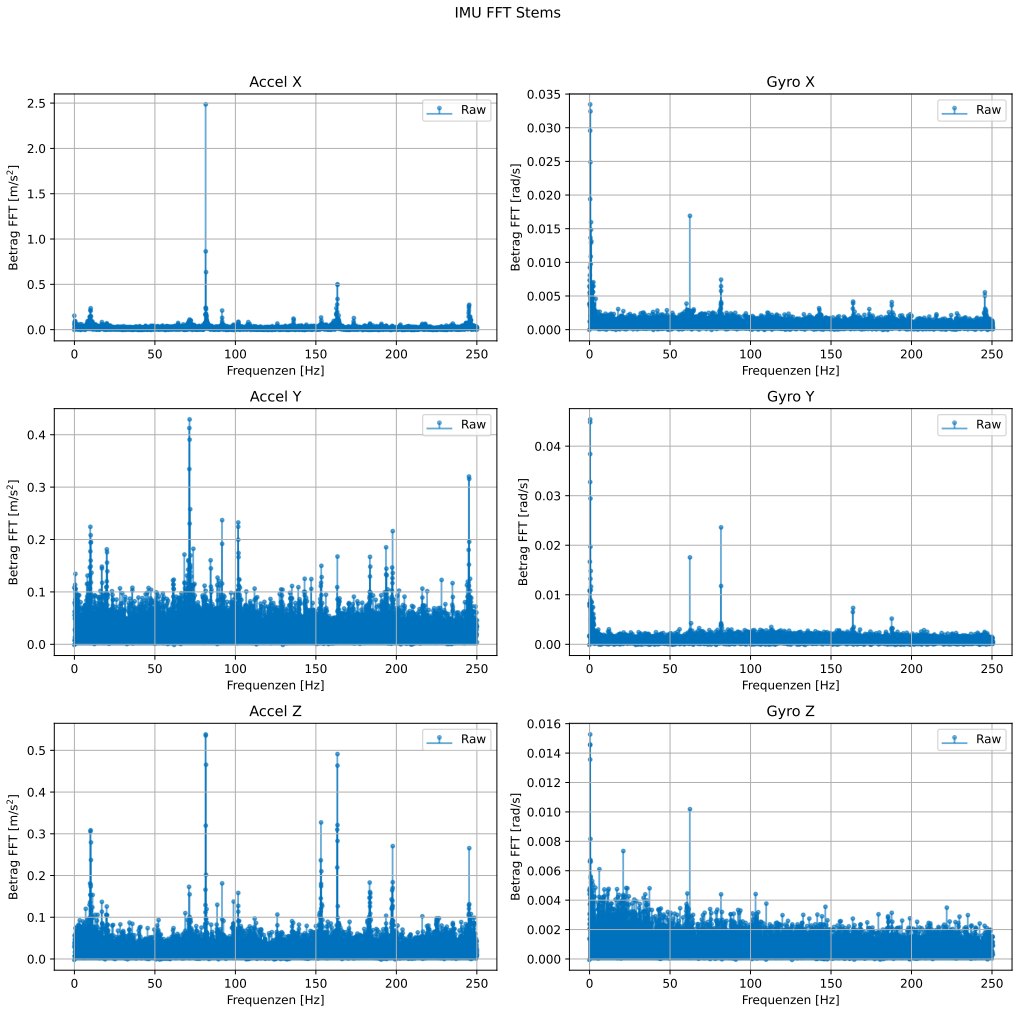


Figure 5.5: IMU Data Raw FFT

This confirms the assumption that the main cause of the disturbances is the gyroscopes. In 83 Hz, 163 Hz, and 249 Hz, large deflections can be seen in all sensor data, which corresponds exactly to multiples of the gyroscope frequency. This phenomenon is particularly evident in the accelerometer data for the x axis. It is also apparent that the gyroscope data is significantly better, as despite the disturbances, there is a considerable deflection at the relevant frequencies (<10 Hz), while the disturbances clearly predominate in the accelerometer data. In the accelerometer data, a further deflection can be seen at 10 Hz. Since this can only be observed in all three accelerometers, it can be assumed that this is a sensor-specific interference.

5.2.3 Signal improvement through filtering on IMU hardware

The FFT analysis (see Figure 5.5) shows that most of the noise occurs in a significantly higher frequency range than the useful signal. The signal quality can therefore be significantly improved by using a low-pass filter that suppresses the unwanted frequencies. A disadvantage of low-pass filters is the phase shift of the relevant frequency components. This should be kept to a minimum in order to maintain the phase reserve of the overall system. A cutoff frequency should therefore be selected that is as low as necessary but as high as possible.

The *ICM20948* IMUs offer the option of activating a built-in Digital Lowpass (DLP) with variable cutoff frequency. This has the great advantage that part of the signal processing is already performed on the IMU by a filter that is precisely matched to the hardware. In addition, the filter acts as an anti-aliasing filter on the IMU, and the sampling frequency of the microcontroller can be reduced because the high frequencies are already filtered out of the signal. The disadvantage is that this reduces the frequency at which the IMU provides data to 1125 Hz [15, S. 60, S.64]. If the clocks of the microcontroller and the IMU are not synchronized, this leads to errors in maintaining the sampling time. However, these errors are minor and are accepted for the following reasons: these errors are particularly serious when the data is to be derived. This is not the case, as the data is either used directly or even integrated. In addition, the frequency of the IMU is still significantly higher than the final sampling frequency selected for the microcontroller (see Chapter 8). Furthermore, due to the high inertia caused by the high weight (14 kg) compared to the sampling frequencies, the system frequency of the mechanical structure is so low that the hardware acts as a strong low-pass filter in which such high-frequency changes are lost. This is confirmed by frequency analyses of the vehicle data (see Figure 6.7), in which no frequencies above 7 Hz occur in the system response to various stimuli.
To test the signal quality with DLP of the IMU activated, the three relevant, adjustable cutoff frequencies 5 Hz, 11 Hz, and 23 Hz are tested. For this purpose, two IMUs are connected to obtain a comparison between the unfiltered signal and the filtered signal. The test procedure remains otherwise identical. The reference signal is now obtained from the filtered IMU data using an FFT low-pass filter. This is necessary because even the FFT-filtered signal from the raw values is too poor compared to the DLP-filtered signal to allow an objective comparison. However, it should be noted that the reference signal no longer contains all relevant frequency components due to the DLP. Therefore, the choice of the DLP cutoff frequency must be adapted to the expected frequency range and must not be based solely on the SNR values.
For example, Figure 5.6 and Figure 5.7 show the results of the experiment at an IMU-DLP

cutoff frequency of $f_G = 11\,\text{Hz}$. The results for $f_G = 5\,\text{Hz}$ and $f_G = 23\,\text{Hz}$ can be found in Appendix B.

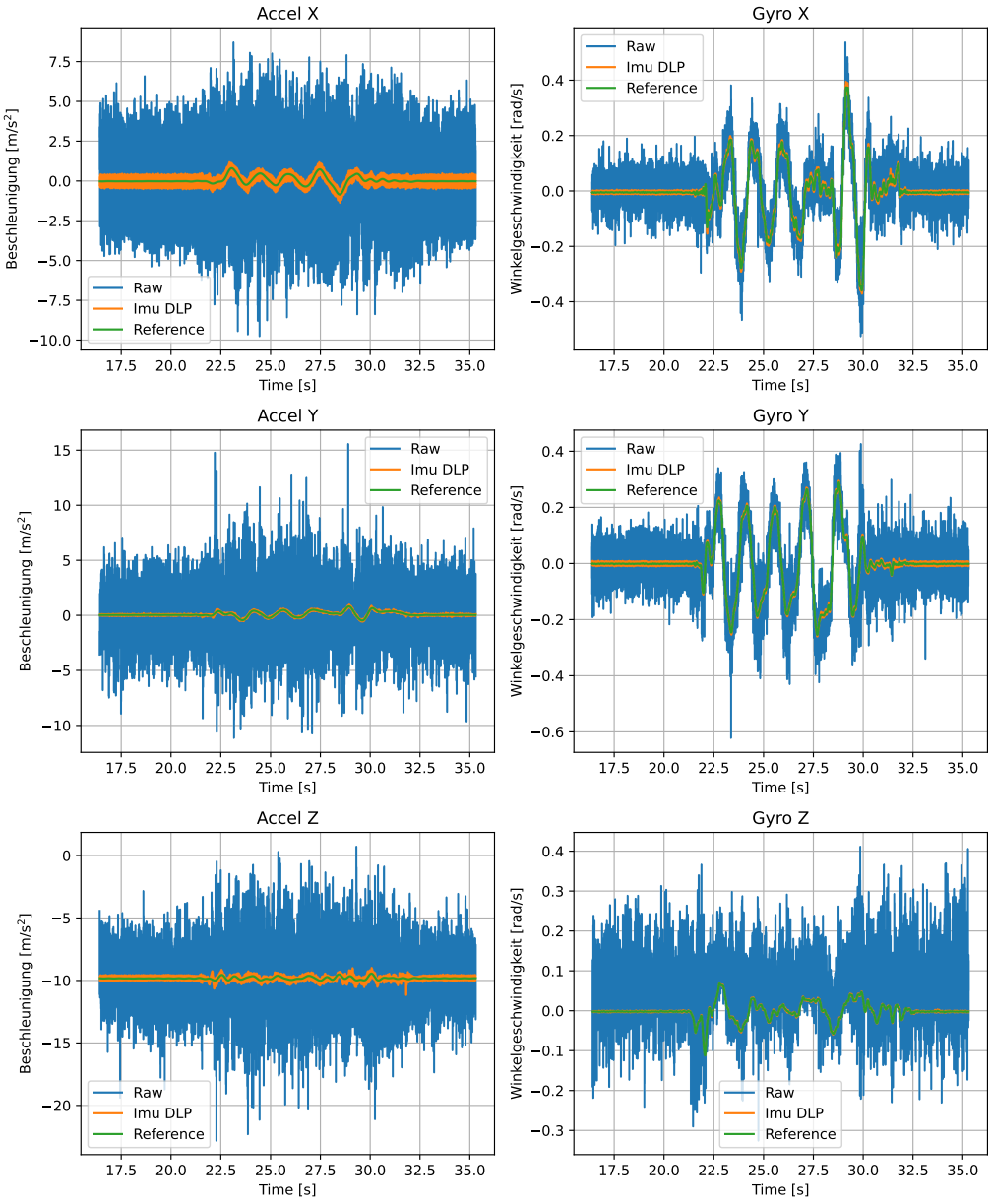


Figure 5.6: IMU Data with IMU-DLP with $f_G = 11\,\text{Hz}$

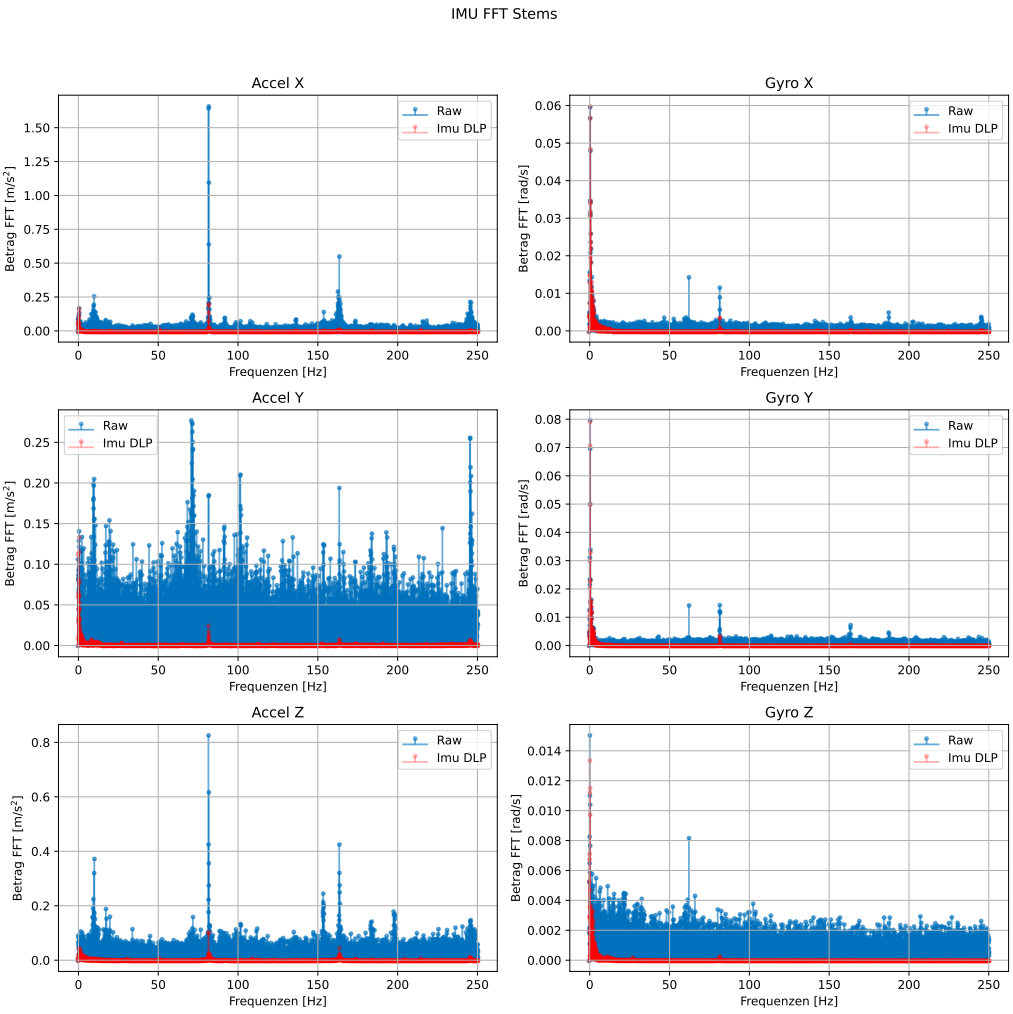


Figure 5.7: IMU data with IMU-DLP with $f_G = 11\,\text{Hz}$ FFT

A comparison of the raw data and the filtered data (see Figure 5.6) shows how much better the signal quality becomes with the filter. The comparison of the frequency spectra in Figure 5.7 confirms this; the high-frequency noise components are attenuated many times over in all signals. This improvement is also reflected in the SNR values (see Table 5.1). The gyroscope data in particular now has excellent SNR values around 25 dB. The signal quality of the accelerometers has also improved significantly, but is still not sufficient. The SNR values are getting better and better at lower cutoff frequencies of the DLPs. Since the accelerometers are only supposed to detect frequencies below 1 Hz, the IMU-DLP with $f_G = 5\,\text{Hz}$ can be selected without any problems in favor of signal quality. For the gyroscopes, the IMU-DLP with $f_G = 11\,\text{Hz}$ FFT offers the best compromise between SNR value, the transmission of relevant frequency components, and the lowest possible phase shift.

5.2.4 Signal improvement through external low-pass filter

Despite the use of the IMU-DLPs, the signal quality of the accelerometers is still not sufficient. This is due to the low SNR values (see Table 5.1). In addition, the frequency analysis of the accelerometers and gyroscopes in Figure B.2 and Figure 5.7 shows that there is still noise at 83 Hz. Therefore, an additional DLP is implemented on the microcontroller.

The DLP should have as monotonous an amplitude characteristic as possible so as not to distort the signal, which can have a negative effect on the control. In addition, the phase shift should be kept to a minimum. Various low-pass filters are generally suitable for filtering:

- Butterworth filter: characterized by a monotonic amplitude characteristic in the passband, with moderate phase shift and good attenuation [26, S. 131] [29, S. 508f].
- Bessel filter: offers an almost linear phase characteristic, which keeps signal distortion to a minimum, but has less steep attenuation in the stopband [26, S. 136].
- Chebyshev filter: enables a steeper slope, but this is accompanied by ripple in the passband, which can distort the signal [29, S. 508f].

As a compromise between attenuation and monotonic amplitude characteristics, a second-order Butterworth filter is chosen. The design is carried out using the Python script *butterworth_filter_designer.py*. The cutoff frequency of $f_G = 40\,\text{Hz}$ is selected so that the interfering frequency 83 Hz is attenuated with a gain of $-20\,\text{dB}$. The Bode diagram of the filter can be seen in Figure 5.8:

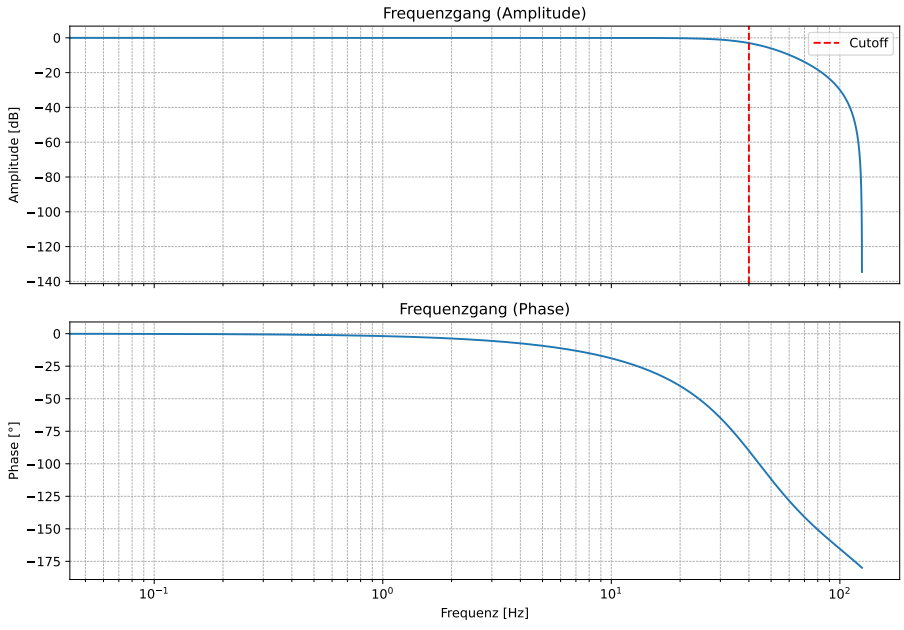


Figure 5.8: Bode diagram Butterworth low-pass 2nd order, $f_G = 40\,\text{Hz}$

The effectiveness of the filter is tested in a further experiment. The reference signal is again the FFT-filtered IMU-DLP signal. In Figure 5.9 and Figure 5.10, the results are shown as

examples for the IMU-DLP with $f_G = 5\,\text{Hz}$. The results for the IMU-DLP with $f_G = 11\,\text{Hz}$ are shown in Figure B.5 and Figure B.6.

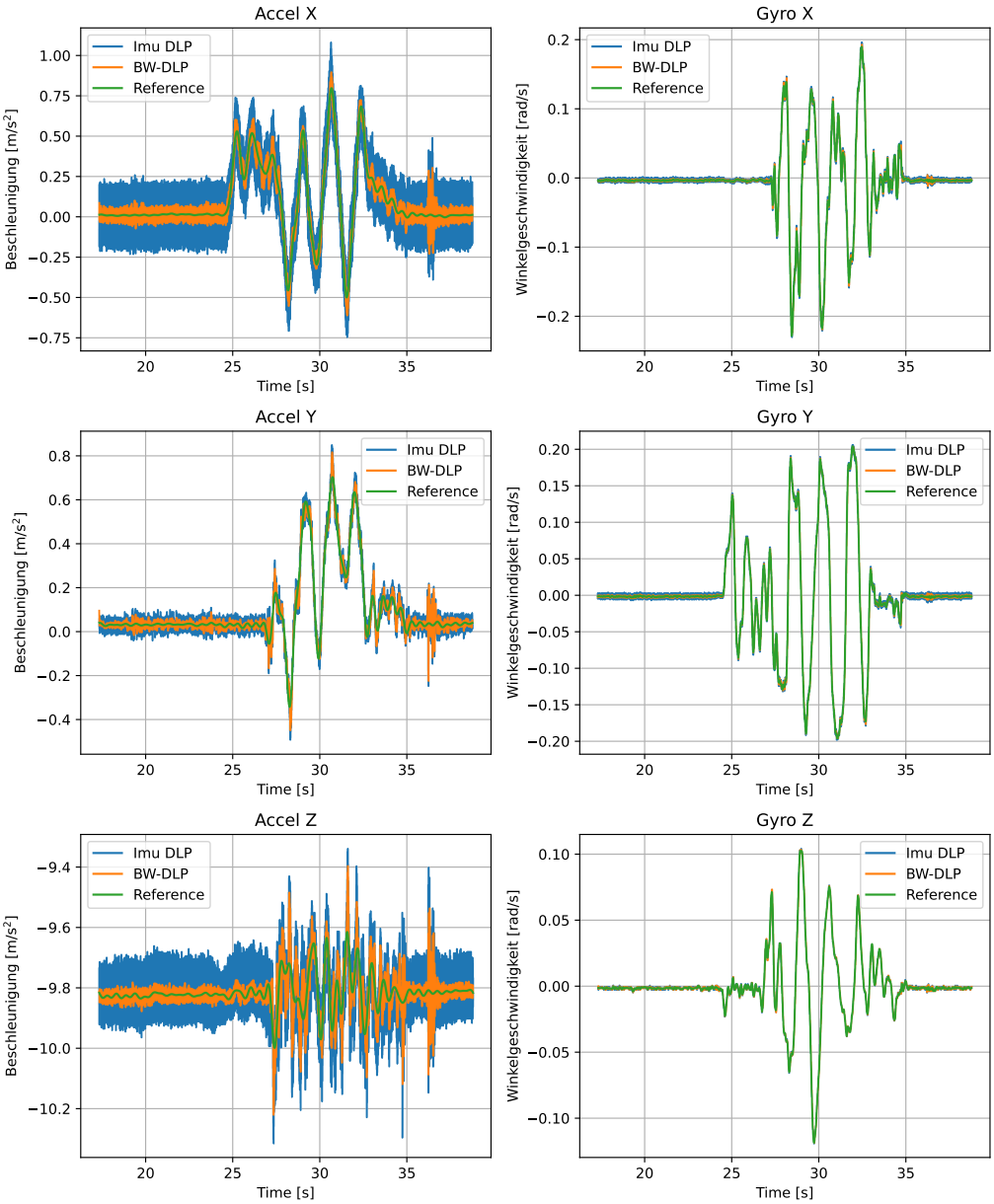


Figure 5.9: IMU data with IMU-DLP with $f_G = 5\,\text{Hz}$ and Butterworth low-pass with $f_G = 40\,\text{Hz}$

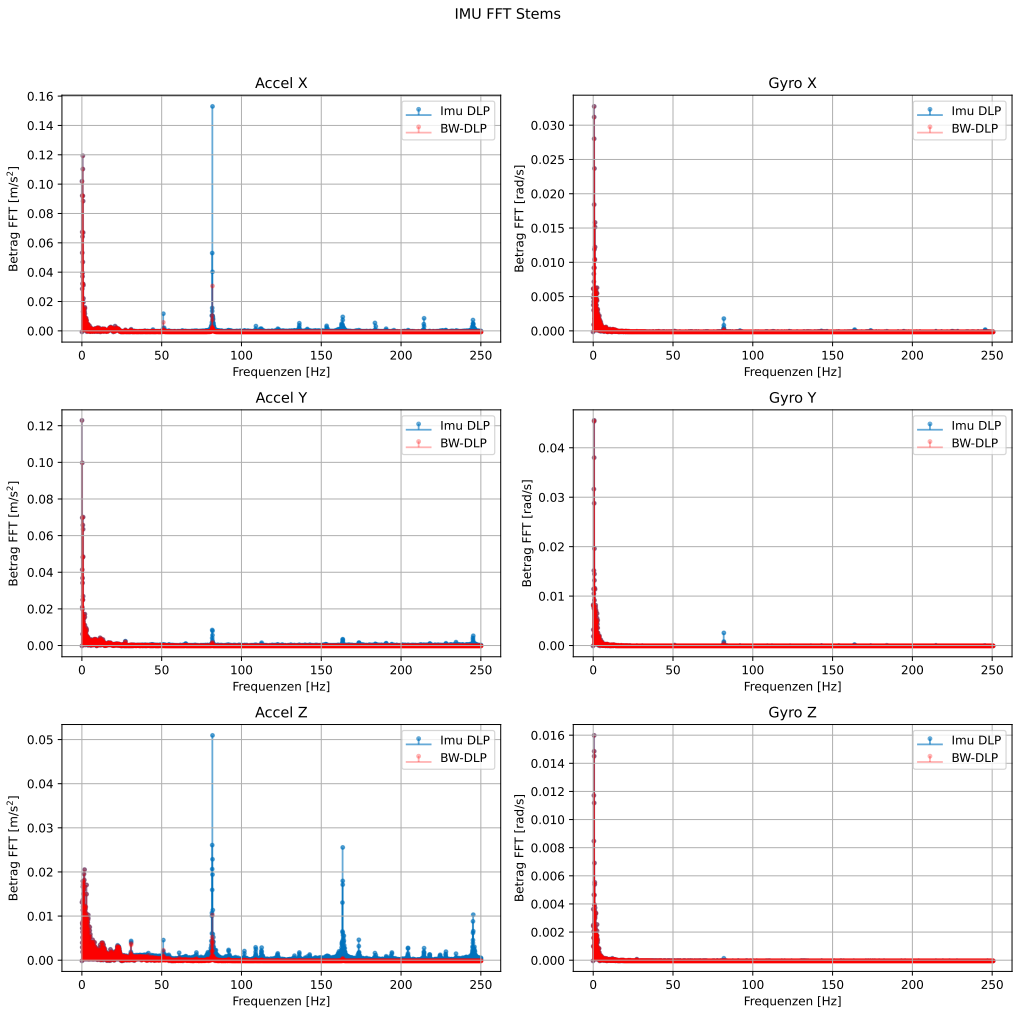


Figure 5.10: IMU data with IMU-DLP with $f_G = 5\,\text{Hz}$ and Butterworth low-pass with $f_G = 40\,\text{Hz}$ FFT

In Figure 5.9, it is clearly evident that the Butterworth low-pass filter significantly attenuates the noise. This can also be seen in the spectra of the signals in Figure 5.10, where the frequency components of the noise are significantly lower. This is also reflected in the greatly improved SNR values (see Table 5.1). In particular, the signal quality of the accelerometers has been significantly improved and is now in a good range with values around $SNR = 14\,\text{dB}$.

5.2.5 Signal quality overview

Table 5.1 shows the results of the SNR values for all tests:

Table 5.1: Übersicht der SNR-Werte der IMU Daten

Datenset	Accelerometer [dB]			Gyroskop [dB]		
	X	Y	Z	X	Y	Z
Raw	-20.9	-18.5	-21.5	0.4	0.9	-6.4
IMU DLP 5 Hz	4.3	13.1	-4.2	26.4	29.4	31.8
IMU DLP 11 Hz	-0.4	11.2	-4.9	24.8	27.0	26.1
IMU DLP 23 Hz	-4.8	9.7	-8.9	20.4	21.9	22.3
IMU DLP 5 Hz + BW-TP 40 Hz	14.6	14.5	0.2	26.2	29.6	28.5
IMU DLP 11 Hz + BW-TP 40 Hz	11.8	14.7	1.1	26.0	29.8	24.9

The insufficient signal quality of the IMU was significantly improved with additional measures, the use of the IMU-integrated DLPs and an additional Butterworth low-pass filter. The SNR values of all sensors could be increased many times over and are now in the good to very good range. However, it is noticeable that the SNR values of the accelerometer of the z axis are poor despite the measures taken. The z component of the gravitational acceleration is calculated by the cos of the angles ϑ and φ, while the x and y components are determined by the sin of the respective angle. For very small angles, the sin has a large slope, and thus the change in the x and y components is large. The cos has virtually no slope for small angles, so small changes do not result in a significant change in the signal. Therefore, the useful signal of the z-axis accelerometer is much smaller and is lost in the noise. To determine the orientation of the vehicle, it is therefore advantageous to use only the data from the x and y accelerometers.

5.2.6 Validation of sensor concept and signal processing

After developing a suitable sensor concept and implementing various measures to ensure signal quality, a test of the entire system is carried out. To do this, the vehicle is again stimulated manually in the relevant frequency and angle ranges. Both the IMU data and the orientation of the vehicle are recorded. In post-processing, the calculation of the orientation of the microcontroller can then be compared with the offline calculation of a Python script (*imu_state_estimation.py*). The results can be seen in Figure 5.11:

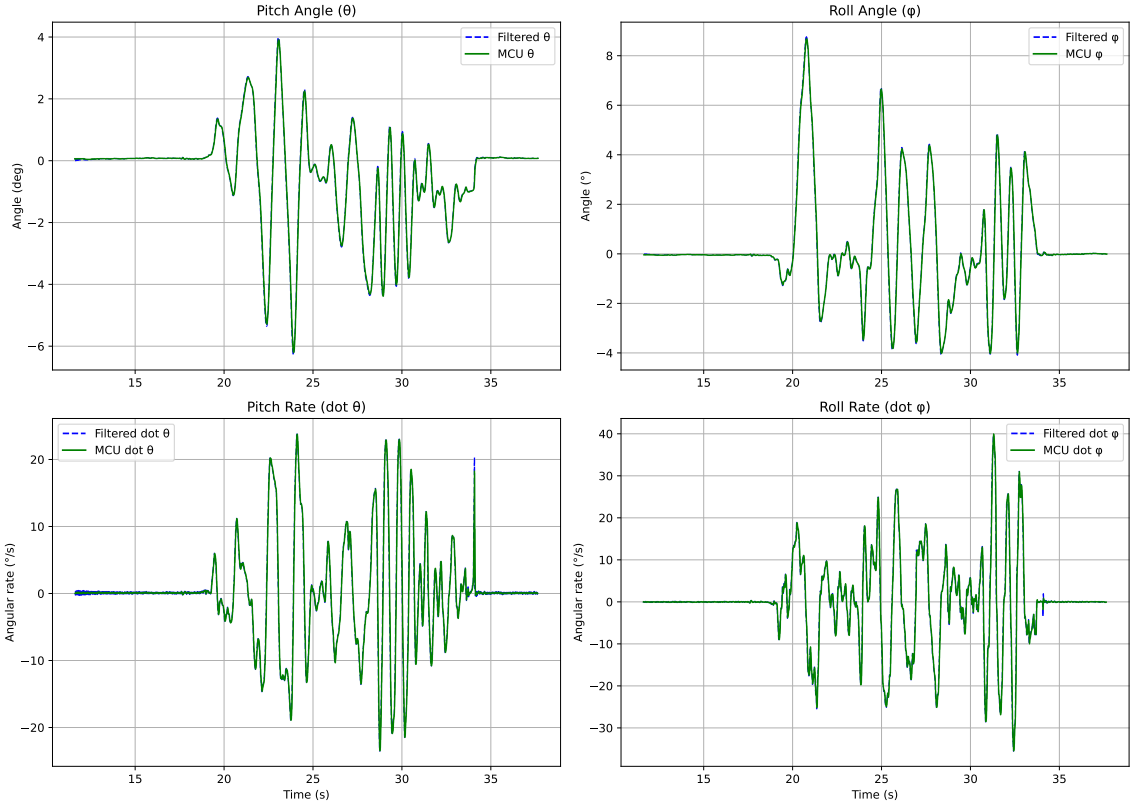


Figure 5.11: Validation of sensor concept and signal processing

The test shows that the movements of the vehicle are recorded precisely. When stationary, there is no drift and all movements are recorded accurately with minimal noise. The orientation calculation on the microcontroller corresponds exactly to the calculation in post-processing.

6 Iterative controller development for balancing

Active control is required to stabilize the unstable subsystems described in Chapter 3 and thus bring the vehicle into balance. In [33], a theoretical and practical design of a PID controller for the pitch subsystem was already carried out based on a simulation and an initial prototype. A theoretical design of a PID controller was also developed for the roll and yaw subsystems.

Based on these results, this chapter develops the control system for the finished vehicle. To do this, controllers are first designed using the model and their properties are checked in a simulation. The simulation results serve as the basis for tests on the real system. The findings from the real-world tests are fed back into the model and simulation to adapt them to the real system or to make specific changes to the controller parameters. This results in an iterative development process that enables the controller parameters to be efficiently tuned to the actual system dynamics.

6.1 sampling frequency

Since the control loop is implemented on a digital system, but the real vehicle is a continuous system, a suitable sampling frequency must be selected. Two factors are relevant here: What is the minimum sampling rate required to discretize the physical signals (including noise and interference), and what sampling rate is required to implement a robust control loop? As shown in Chapter 5, the highest frequency of the IMU signal detected by the microcontroller is 83 Hz, which corresponds to the noise of the gyroscope. According to the sampling theorem, the minimum requirement is therefore (the actual sampling frequency should be set slightly higher, as the sampling theorem assumes ideal filters):

$$f_{a,min} > 2 \cdot 83\,\text{Hz} = 166\,\text{Hz} \tag{6.1}$$

However, in order to guarantee a stable control loop, the sampling frequency should be selected to be approximately five to ten times [9, S. 318], or twenty times [8, Ch. 11], higher than the bandwidth of the system. Although the sampling theorem guarantees error-free reconstruction, the state changes (derivatives) are usually not resolved with sufficient precision and exhibit significant discontinuities. However, these variables are often input variables for actuators that cannot accurately reproduce such jumps. This leads to jerky movements, overshoots, excites vibrations, and reduces the stability of the control loop [8, S. 451f]. In addition, the disturbance behavior of the control loop becomes significantly worse at sampling frequencies lower than ten times the bandwidth of the system [8, S. 451f]. In addition, discretization acts like a zero-order hold element, causing a dead time that reduces the phase reserve of the system. The loss of phase reserve θ can be estimated as follows [8, S. 63]:

$$\theta = -\frac{\omega \cdot T}{2} \tag{6.2}$$

Here, ω describes the frequency at which the amplitude gain of the open system is 0 dB. This frequency can be roughly approximated by the bandwidth of the system, which is approximately 10 Hz for the *Monowheeler*. This results in the following for a sampling frequency of 50 Hz:

$$\theta = -\frac{10 \cdot 2 \cdot \pi \cdot \frac{1}{50\,\mathrm{Hz}}}{2} = -36^\circ \tag{6.3}$$

For a sampling frequency of 200 Hz, the result is already:

$$\theta = -\frac{10 \cdot 2 \cdot \pi \cdot \frac{1}{200\,\mathrm{Hz}}}{2} = -9^\circ \tag{6.4}$$

A high sampling frequency therefore has a positive effect on the control loop. A high sampling frequency of $f_a = 250\,\mathrm{Hz}$ is therefore selected, which can be easily implemented on the microcontroller and corresponds to a sampling time of $T_a = 4\,\mathrm{ms}$.

6.2 Pitch subsystem

Since successful control for a prototype has already been implemented in [33], this is used as the starting point. However, the parameters of the prototype used in [33] differ significantly from those of the finished vehicle. It is also particularly important to note that the prototype did not have a gyroscope, so the IMU data was much less noisy. This meant that additional filters were not required, which has a positive effect on the control system due to better phase characteristics. This is particularly important because [33, S. 106] documents that the system is very sensitive to phase shift.

Special requirements must be taken into account when designing the controller. When modeling in Chapter 3, dynamic effects (e.g., the reaction torque of the actuator) of the displacement process in the pitch subsystem are neglected. In the roll and yaw subsystems, too, the pitch movement leads to disruptive gyroscopic moments depending on the deflection of the gyroscope. In addition, platform accelerations degrade the IMU signals (see Chapter 5). To minimize these negative effects, the system behavior should be fast enough to robustly compensate for all disturbances, but at the same time avoid abrupt accelerations. Classic methods such as Ziegler-Nichols for controller design are designed for more aggressive controller tuning [11] and are therefore of limited use for this application.
The tests performed all follow the same pattern: uprighting at the beginning of the experiment, followed by two short disturbances in the opposite direction and a sustained disturbance at the end. It should be noted that at the beginning of the experiments, the filters for calculating the orientation are not initialized. As a result, the measured angle ϑ slowly approaches the actual angle at the beginning, as the correction is only performed by the inertial accelerometers. This acts as a kind of trajectory and gently guides the controller to $\vartheta = 0^\circ$. In later tests, initialization of the filters before the start of control is introduced.

6.2.1 PID controller

The original PID controller in [33] is implemented in the form

$$u = KPID \cdot (e + TV \cdot \dot{e} + \frac{1}{TN} \cdot e_{sum}) \tag{6.5}$$

. In order to be able to change the individual components independently of each other, the following form is now used:

$$u = KP \cdot e + KD \cdot \dot{e} + KI \cdot e_{sum} \tag{6.6}$$

Improvement of system dynamics

First, the controller parameters of the most robust controller from [33] are tested. This results in the following system behavior with $KP = 0.3;\ KD = 0.045;\ KI = 0.3$ (see Figure 6.1):

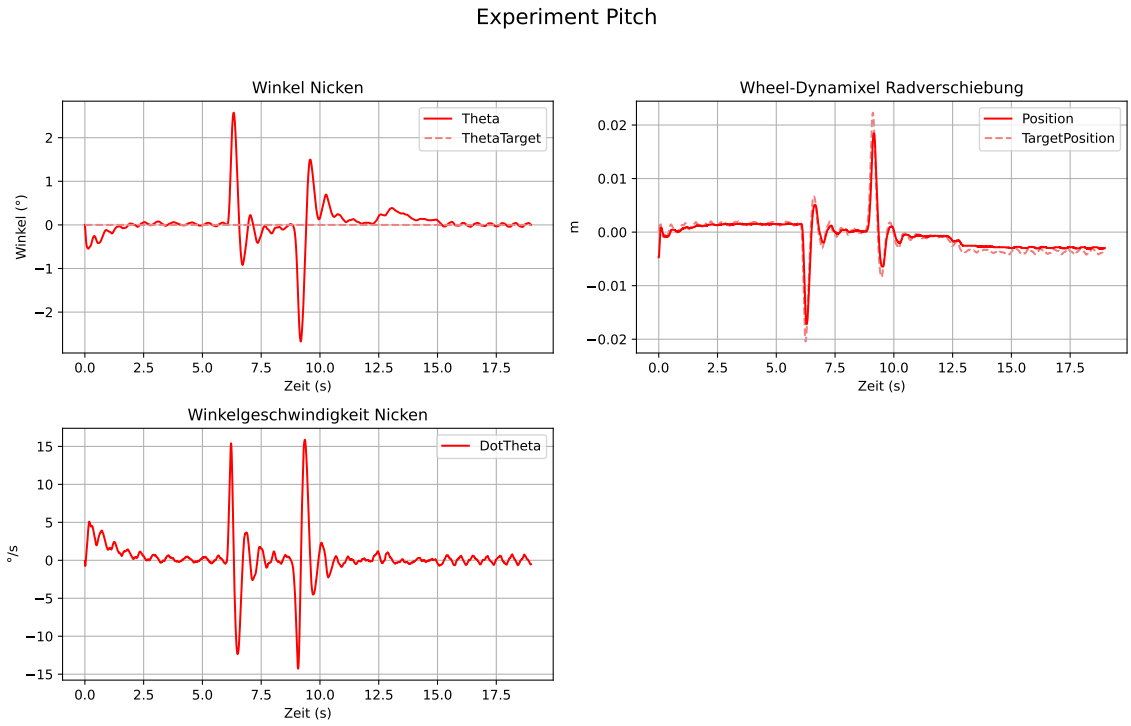


Figure 6.1: Pitch Experiment PID Controller with $KP = 0.3;\ KD = 0.045;\ KI = 0.3$

It can be seen that both dynamic and permanent disturbances can be compensated without any problems and that the system is stable. However, the excitations cause a damped oscillation, which suggests that the damping of the closed system is not large enough. To reduce the oscillation, the damping is increased, resulting in the following controller parameters: $KP = 0.3;\ KD = 0.1;\ KI = 0.3$. The behavior of the system can be seen in Figure 6.2:

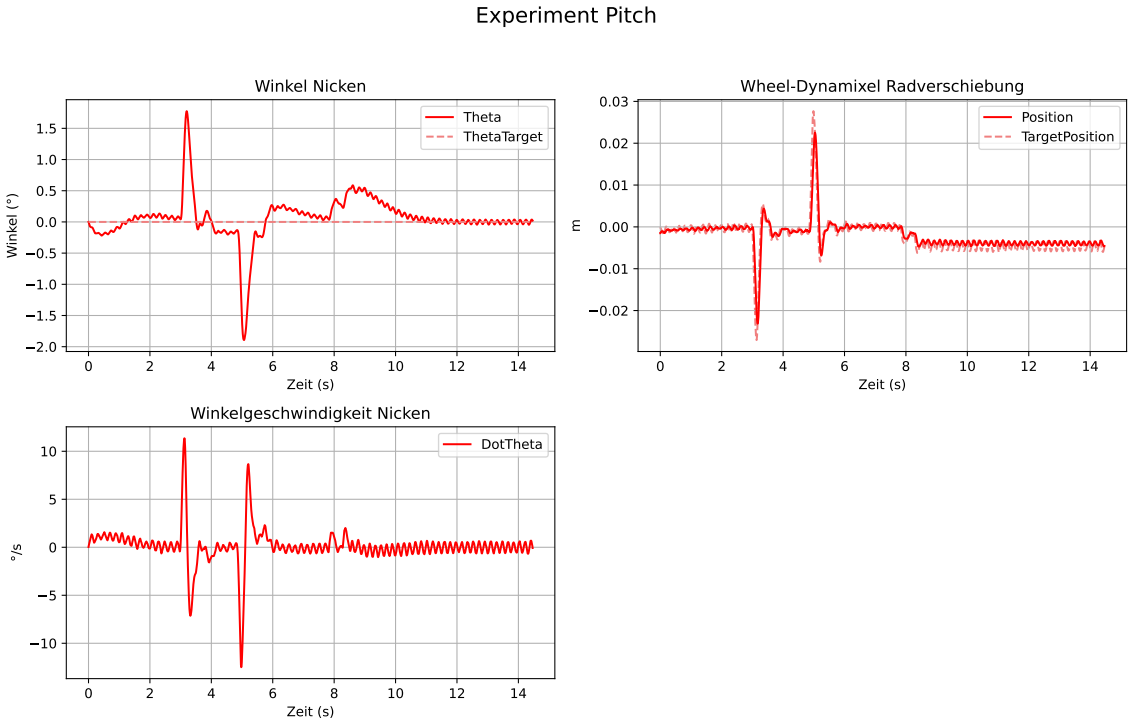


Figure 6.2: Pitch Experiment PID controller with $KP = 0.3;\ KD = 0.1;\ KI = 0.3$

The dynamic behavior has clearly improved. The damped oscillation after excitation has given way to a simple post-oscillation whose amplitude is many times smaller than that of the original oscillation. Further increasing the damping KD or the proportional component KP worsens the dynamic response of the system (see Figure 6.3).

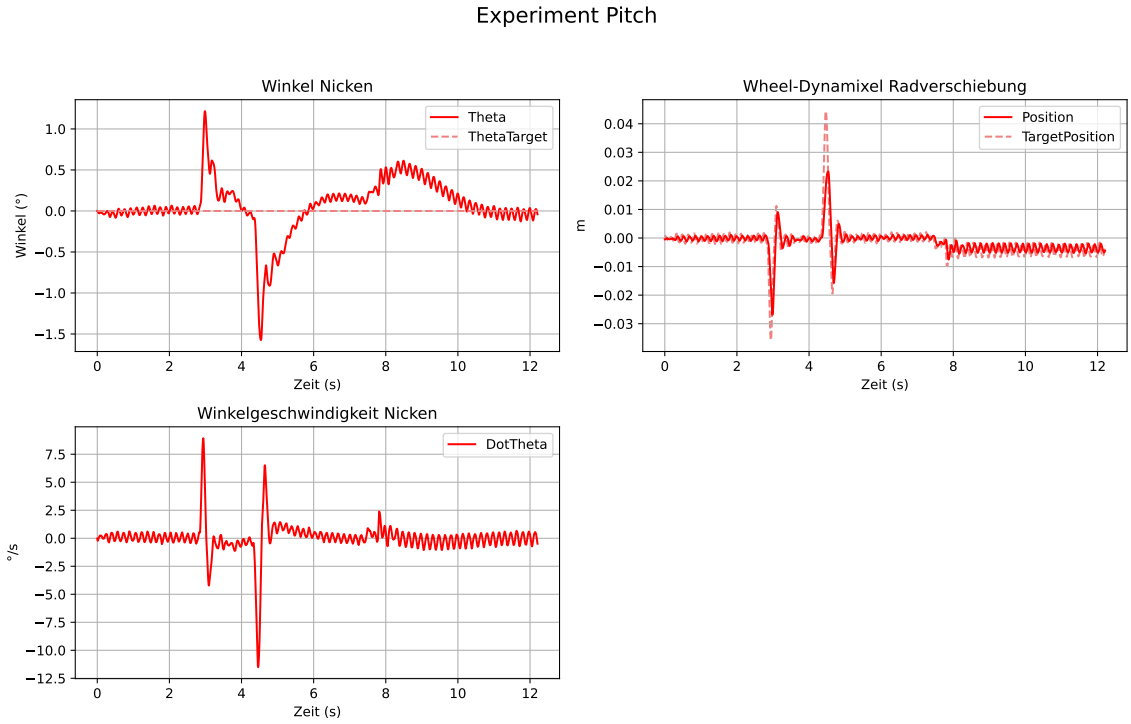


Figure 6.3: Pitch experiment PID controller with $KP = 0.3;\ KD = 0.2;\ KI = 0.3$

The limiting factor here is the actuator, which, as can be seen in Figure 6.3 in plot *Wheel Dynamixel wheel displacement*, can no longer follow the controller output.

Oscillations in steady-state operation

The experiments reveal a second problem: the system oscillates in the rest position, and this oscillation becomes stronger the faster the controller is designed. The oscillation is clearly excited by the controller and disappears when slight mechanical damping is added. This suggests that the combination of an underdamped mechanical system around the rest position and the nonlinearities and dead times of the actuator is the cause. This is consistent with the behavior of the servo. The dynamic characteristics of the Dynamixel servo motor are significantly worse with small setpoint specifications than with larger jumps (see Subsection 4.1.1). Since this is a unicycle, no additional mechanical damping can be built into the system. Adjustments to the internal controller of the Dynamixel servo motor also do not improve the situation. Figure 6.1 shows why the constant oscillation has a smaller amplitude in stationary operation when the controller is slower. The actuator hardly follows the small setpoint changes, so the effect on the vehicle is also greatly reduced. Since the damping KD is largely responsible for the oscillations, gain scheduling is introduced for KD. The goal is to take advantage of the high damping during fast movements and to minimize oscillations in the stationary range by means of very low damping. The damping is scaled using a factor that depends on the current angular velocity:

```
1 double damperD = std::clamp(std::abs(mDotE)/mPID.DScaleThreshold, 0.2, 1.0);
2 mU = mDirection*(mPID.KP*mE + mPID.KD*damperD*mDotE + mPID.KI*mSumE);
```

Listing 4: Nicken Gain-Scheduling

If the angular velocity is below the threshold $DScaleThreshold$, KD is reduced linearly to minimize discontinuities. The controller parameters $KP = 0.3;\ KD = 0.1;\ KI = 0.3$ and gain scheduling result in the following behavior (see Figure 6.4):

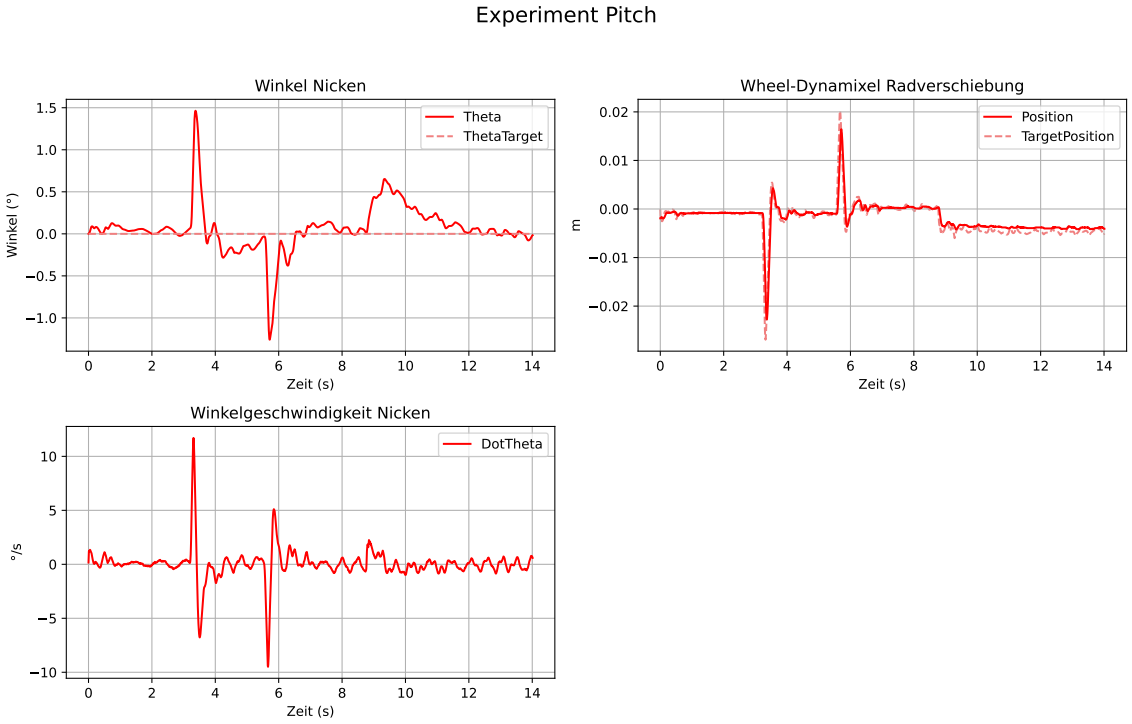


Figure 6.4: Pitch experiment PID controller with $KP = 0.3;\ KD = 0.1;\ KI = 0.3$ with gain scheduling for KD

The dynamic behavior deteriorates only minimally, while the frequency of the oscillation in the steady state is drastically reduced. Due to the low frequency and small amplitude ($< 0.05°$), the oscillation is barely visible to the naked eye and is within a tolerable range.

Targeted and adaptive parameterization of the PID controller effectively stabilizes the pitch subsystem. As a result, the overall system is highly robust and can reliably counteract even major disturbances. However, the dynamics achieved are not optimal, as residual oscillation behavior continues to occur and the oscillations in steady-state operation can be reduced but not completely eliminated.

6.2.2 State controller

The previous section examines the stabilization of pitch using a PID controller. The PID controller ensures sufficient stability and can compensate for major disturbances. Nevertheless, it is apparent that the system dynamics cannot be optimally influenced. In particular, oscillations remain and steady-state oscillations cannot be completely eliminated. Further optimization of the PID controller taking into account the actuator dynamics (PT1 delay) is not possible because the system would then no longer meet the Hurwitz stability criterion. In order to comply with the stability criterion, the time constant of the actuator must be negligibly small compared to the system dynamics [33, S. 104].

Against this background, the transition to state control takes place. A state controller offers several advantages in this situation:

- In contrast to the PID controller, a state controller is based on a mathematical model of the system. This allows it to specifically take internal state variables into account and actively incorporate them into the control [8, Ch. 8].
- Better consideration of actuator dynamics: By explicitly modeling the actuator in the extended state space, delays and inertia can be directly compensated without destabilizing the system.
- Improved damping and reduction of oscillations: By specifically selecting the controller gains (e.g., using LQR or pole placement), the eigenvalue position can be systematically shifted, reducing excess oscillations [8, Ch. 8].

State-based control methods offer various approaches to stabilizing and optimizing dynamic systems, each with different advantages and disadvantages.

- Pole placement: In pole placement, the feedback matrix K is specifically selected so that the poles of the closed system specify the desired properties. This is simple, but does not allow for optimization and does not take into account control effort or robustness [9, Ch. 18].
- Linear Quadratic Regulator (LQR): LQR minimizes a quadratic cost function that weights deviations of the states from zero and the control effort. This method offers systematic and robust optimization between control quality and control effort, but is limited to linear models. The computational effort at runtime is very low and the stability of the closed system is very high. [9, Ch. 22.4].
- H_2/H_∞: H_2 and H_∞ controllers extend the optimization of the LQR approach to the frequency domain: H_2 minimizes the average disturbance energy and is particularly suitable for systems with random disturbances, while H_∞ limits the maximum gain across all frequencies, thus ensuring robustness against model uncertainties. However, both methods are mathematically more complex. [31].
- MPC: MPC predicts the behavior of the system and optimizes the control variables at runtime, taking into account state and control variable constraints, using a defined cost function. This allows very complex systems to be controlled in a targeted and robust manner, but the implementation and computational effort is very high [28].

The LQR method is chosen due to the low computing time at runtime and the good robustness of the closed system. The following describes the implementation of state control for the pitch subsystem, including the selection of state variables and the advantages over the previous PID control.

Modeling in state space

To design the state controller, the pitch subsystem must be converted to state space representation. The system is no longer described by a higher-order differential equation, but as a first-order linear equation system in which the dynamics of the system are described by so-called state variables $x_1, x_2, ...x_n$. The system can be influenced by the controller with the input u. This is a system of the form [9, Ch. 17]:

$$\dot{\vec{x}} = A\vec{x} + Bu \tag{6.7}$$

A is the so-called system matrix and B is the so-called control matrix. The control variable is then calculated using the vector of all states and the controller matrix K:

$$u = -K\vec{x} \tag{6.8}$$

Since the LQR approach only applies to linear systems, the differential equation Equation 3.1 must be linearized. The only nonlinearity is the sine function. Based on the small-angle approximation, the linearized differential equation is:

$$J_{y,\tau} \cdot \ddot{\vartheta} = F_{GP} \cdot p + F_G \cdot h_0 \cdot \vartheta \tag{6.9}$$

From the linearized differential equation for the description of pitching, the following assignment of state variables and their derivatives results:

$$u = p \tag{6.10}$$
$$x_1 = \vartheta \tag{6.11}$$
$$x_2 = \dot{\vartheta} \tag{6.12}$$
$$\dot{x}_1 = x_2 \tag{6.13}$$
$$\dot{x}_2 = \frac{1}{J_{y,\tau}} \cdot (F_{GP} \cdot u + F_G \cdot h_0 \cdot x_1) \tag{6.14}$$

The state variables are the pitch angle and its derivative. The input u of the system corresponds to the wheel displacement. This results in the following system:

$$\begin{bmatrix} \dot{x}_1 \\ \dot{x}_2 \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ \frac{F_G h_0}{J_{y,\tau}} & 0 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + \begin{bmatrix} 0 \\ \frac{F_{GP}}{J_{y,\tau}} \end{bmatrix} u \tag{6.15}$$

A state controller always attempts to pull all state variables to zero. In order to be able to compensate for permanent disturbances and model errors, the state space model must be extended to include the integrated error.

$$u = p \tag{6.16}$$
$$x_1 = \vartheta \tag{6.17}$$

$$x_2 = \dot{\vartheta} \tag{6.18}$$

$$x_3 = \int -\vartheta \; dt \tag{6.19}$$

$$\dot{x}_1 = x_2 \tag{6.20}$$

$$\dot{x}_2 = \frac{1}{J_{y,\tau}} \cdot (F_{GP} \cdot u + F_G \cdot h_0 \cdot x_1) \tag{6.21}$$

$$\dot{x}_3 = -\vartheta \tag{6.22}$$

This results in the following system:

$$\begin{bmatrix} \dot{x}_1 \\ \dot{x}_2 \\ \dot{x}_3 \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 \\ \frac{F_G h_0}{J_{y,\tau}} & 0 & 0 \\ -1 & 0 & 0 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} + \begin{bmatrix} 0 \\ \frac{F_{GP}}{J_{y,\tau}} \\ 0 \end{bmatrix} u \tag{6.23}$$

So far, the dynamics of the actuator have not been taken into account in the model, since the input u directly describes the wheel displacement. In order to represent the delay of the actuator in the model, the state model is expanded again and the wheel displacement becomes a state variable. The input u of the system is now no longer directly the wheel displacement, but the input of the actuator p_{cmd}. The relationship between the input of the actuator and the wheel displacement can be described by a PT1 element:

$$K \cdot p_{cmd} = T \cdot \dot{p} + p \tag{6.24}$$

The parameters K and T are determined in Subsection 4.1.1. The extended system can be described as follows:

$$u = p_{cmd} \tag{6.25}$$

$$x_1 = \vartheta \tag{6.26}$$

$$x_2 = \dot{\vartheta} \tag{6.27}$$

$$x_3 = \int -\vartheta \; dt \tag{6.28}$$

$$x_4 = p \tag{6.29}$$

$$\dot{x}_1 = x_2 \tag{6.30}$$

$$\dot{x}_2 = \frac{1}{J_{y,\tau}} \cdot (F_{GP} \cdot x_4 + F_G \cdot h_0 \cdot x_1) \tag{6.31}$$

$$\dot{x}_3 = -\vartheta \tag{6.32}$$

$$\dot{x}_4 = \frac{1}{T} \cdot (K \cdot u - x_4) \tag{6.33}$$

This results in the following extended system:

$$\begin{bmatrix} \dot{x}_1 \\ \dot{x}_2 \\ \dot{x}_3 \\ \dot{x}_4 \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 & 0 \\ \frac{F_G h_0}{J_{y,\tau}} & 0 & 0 & \frac{F_{GP}}{J_{y,\tau}} \\ -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & -\frac{1}{T} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ 0 \\ \frac{K}{T} \end{bmatrix} u \tag{6.34}$$

This corresponds to the system matrix A, control matrix B, and output matrix C:

$$A = \begin{bmatrix} 0 & 1 & 0 & 0 \\ \frac{F_G h_0}{J_{y,\tau}} & 0 & 0 & \frac{F_{GP}}{J_{yt}} \\ -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & -\frac{1}{T} \end{bmatrix}, \quad B = \begin{bmatrix} 0 \\ 0 \\ 0 \\ \frac{K}{T} \end{bmatrix}, \quad C = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} \tag{6.35}$$

The present state model describes the dynamics of the pitch subsystem, taking into account the influence of the actuator. To assess whether the coupled system can be controlled, controllability is examined. Controllability can be determined using the Cayley-Hamilton theorem [9, S. 491]. For this purpose, the controllability matrix is set up. If the rank of the matrix corresponds to the rank of the system matrix, the system is fully controllable. The calculation is performed in the Python script *pitch_analysis.py* and can be seen in part in Listing 5:

```
# --- Steuerbarkeit ---
n = A.shape[0]
Ctrb = B
for i in range(1, n):
    Ctrb = np.hstack((Ctrb, np.linalg.matrix_power(A, i) @ B))

rank_ctrb = matrix_rank(Ctrb)
print("Rang der Kontrollierbarkeitsmatrix:", rank_ctrb, "von", n)
```

Listing 5: Bestimmung der Steuerbarkeit

The present state model is fully controllable.

Control and simulation in state space

The state controller is designed using LQR in the Python script *pitch_LQR.py* with the Python toolbox *control*. The linear system must be discretized for the discrete controller design. The states and the manipulated variable can then be weighted by the matrix Q and the factor R, and the controller matrix K can be determined using the function *dlqr()*. The corresponding code section can be seen in Listing 6:

```
import control
sys_c = control.ss(A, B, np.eye(3), np.zeros((3,1)))
sys_d = control.c2d(sys_c, ctrl_Ta, method='zoh')
Ad = sys_d.A
Bd = sys_d.B
# === LQR für diskretes System ===
K, Sd, Ed = control.dlqr(Ad, Bd, Q, R)
```

Listing 6: Diskreter LQR-Entwurf in Python

The weighting of the states and manipulated variable influences the controller matrix K. To generate the desired system behavior, the weighting must be adapted to the system. *Bryson's*

Rule can be used as a starting point. Each weighting factor is scaled to the maximum permissible value of the associated state. This is particularly important if the states have different numerical orders of magnitude due to the physical units [13, S. 196]. The values specified in Table 6.1 are used as maximum values:

Table 6.1: Maximal zulässige Werte der Zustände des Teilsystems Nicken

Zustand	Physikalische Größe	Maximaler Wert
x_1	ϑ	$0.02\,\mathrm{rad}$
x_2	$\dot{\vartheta}$	$0.02\,\mathrm{rad\,s^{-1}}$
x_3	$\int -\vartheta$	$0.001\,\mathrm{rad\,s}$
x_4	p	$0.002\,\mathrm{m}$
u	p_{cmd}	$0.002\,\mathrm{m}$

This results in the following weightings:

$$Q = \begin{bmatrix} \frac{1}{0.02} & 0 & 0 & 0 \\ 0 & \frac{1}{0.02} & 0 & 0 \\ 0 & 0 & \frac{1}{0.001} & 0 \\ 0 & 0 & 0 & \frac{1}{0.002} \end{bmatrix}, \quad R = \frac{1}{0.002} \tag{6.36}$$

These weightings can be used as the initial state for further optimizations. To do this, the behavior of the system with state controller is simulated in the same Python script *pitch_LQR.py*. As in the real experiment, righting, a dynamic disturbance, and a permanent disturbance are tested. By gradually adjusting Q and R, the desired system behavior with control matrix K is finally achieved for the following weighting as a compromise between aggressive disturbance behavior, smooth control variable response, and no oscillations:

$$Q = \begin{bmatrix} \frac{1}{0.02} & 0 & 0 & 0 \\ 0 & \frac{1}{0.02} & 0 & 0 \\ 0 & 0 & \frac{1}{0.001} \cdot 15 & 0 \\ 0 & 0 & 0 & \frac{1}{0.002} \cdot 500 \end{bmatrix}, \quad R = \frac{1}{0.002} \cdot 340, \quad K = \begin{bmatrix} 0.666\,381\,56 \\ 0.129\,172\,11 \\ -0.285\,209\,78 \\ 1.028\,201\,38 \end{bmatrix} \tag{6.37}$$

The simulated system behavior can be seen in Figure 6.5:

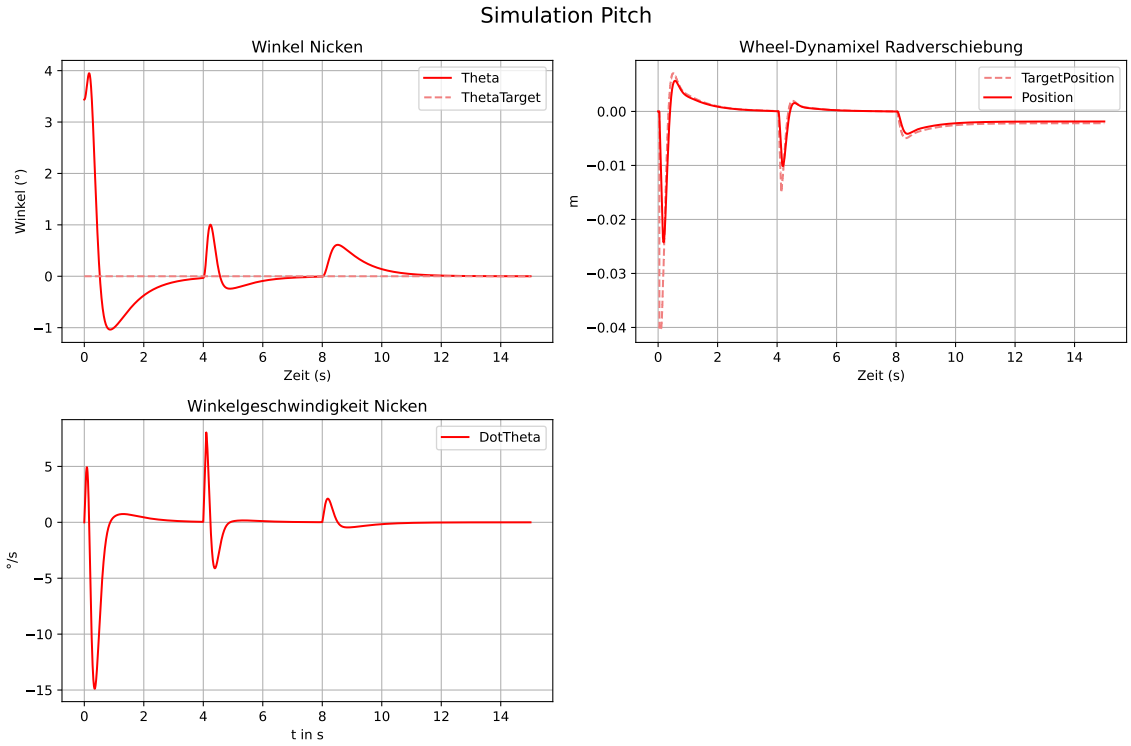


Figure 6.5: Pitch state controller simulation

Real system control

The state controller is implemented on the real system and tested identically to the PID controller. This results in the following system behavior (see Figure 6.6):

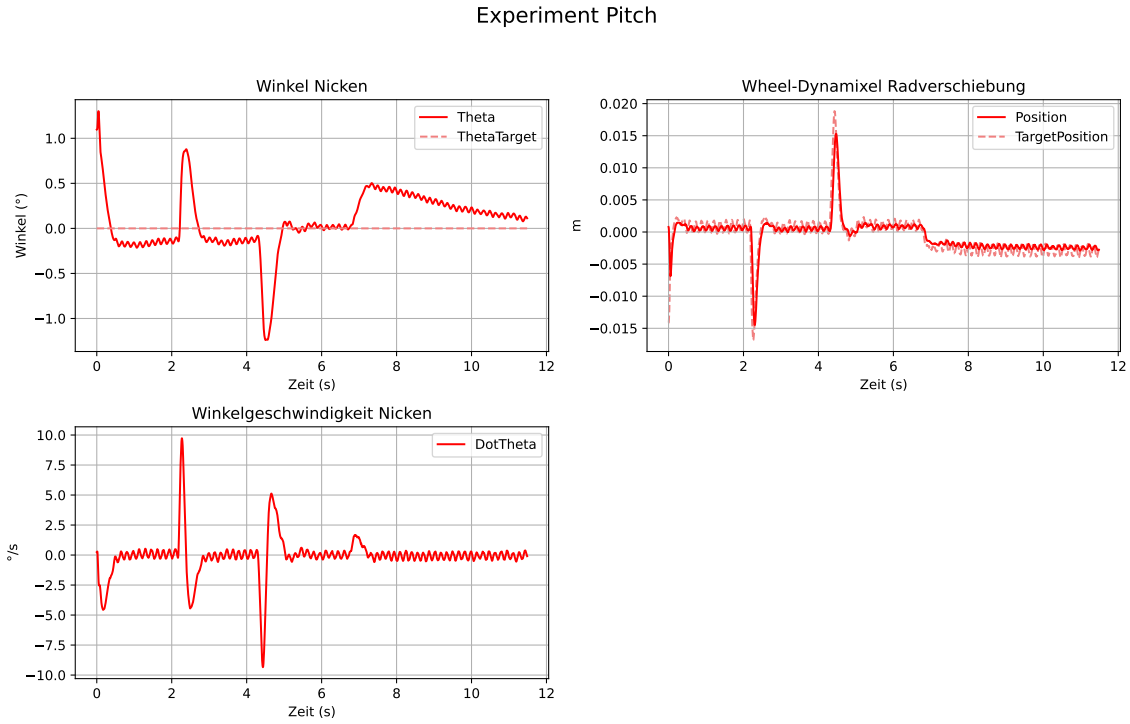


Figure 6.6: Pitch state controller

The behavior of the real system is almost identical to the behavior of the simulation. The dynamics are significantly better compared to the PID controller, and the oscillations after excitation no longer occur. However, the system continues to oscillate in the rest position. Such oscillations can also occur if the controller happens to excite a resonance frequency of the mechanics. To investigate this, a recording is made in which the vehicle is not disturbed, so that only the oscillation in the rest position is recorded. The frequency range of the controller output is then examined (see Figure 6.7):

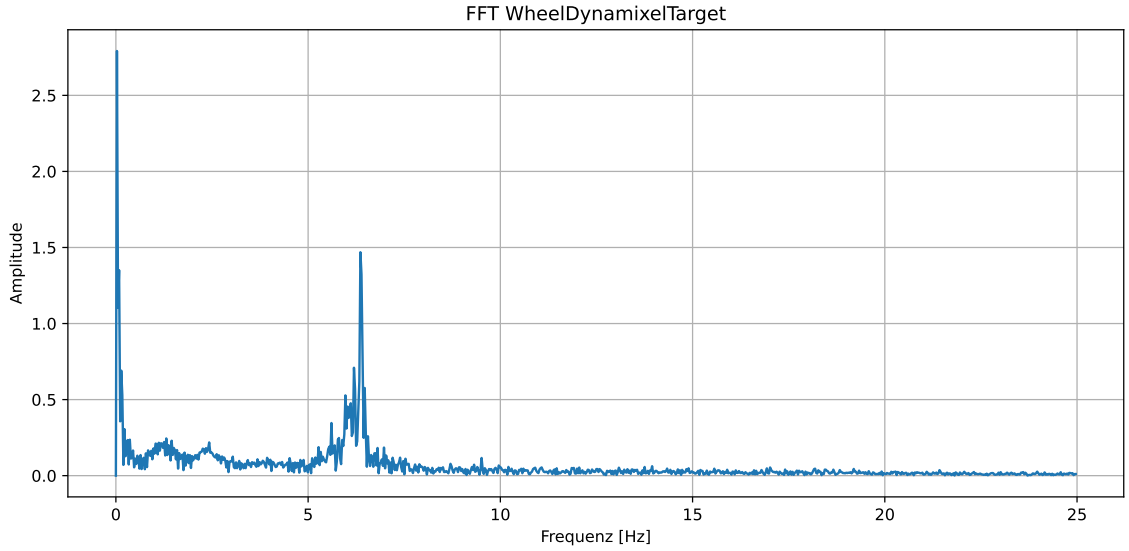


Figure 6.7: FFT controller output

As expected, no frequencies above 8 Hz occur in the signal, since the vehicle's mechanics act as a low-pass filter. However, a clear deflection can be observed at 6.3 Hz. This corresponds to the frequency of the oscillations in the rest position. To test whether this is a resonance frequency of the mechanics, a notch filter with a notch frequency of 6.3 Hz is applied to the controller output and the same test is performed again (see Figure 6.8):

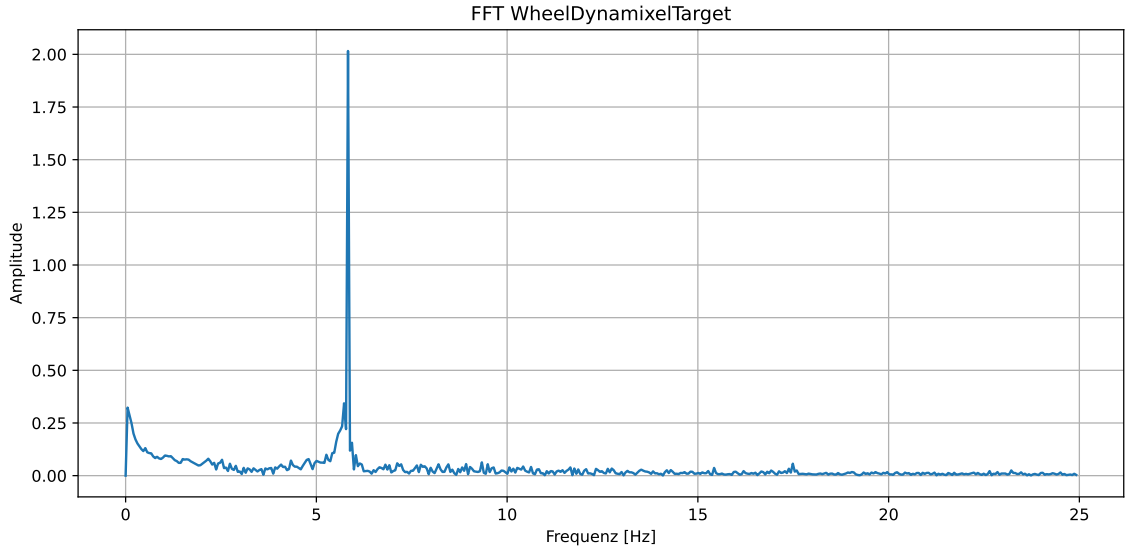


Figure 6.8: FFT controller output with notch filter

The oscillation has shifted to a different frequency, so that the deflection is now at 5.8 Hz. This shows that it is not a resonance frequency of the mechanics. To rule out the possibility that the oscillation is generated by the dynamics of the closed control loop, its poles are

examined. The calculation is also performed in the Python script *pitch_LQR.py*. The poles of the discrete system are as follows:

$$p_1 = 0.880\,342\,93,\ p_2 = 0.977\,292\,11,\ p_3 = 0.985\,676\,23,\ p_4 = 0.994\,653\,85 \tag{6.38}$$

All poles are purely real and lie within the unit circle, so the system is not oscillatory and is stable. This suggests that the oscillations are caused by unmodeled nonlinearities. Although the actuator is modeled as a PT1 element, it actually also has a dead time of approx. 5 ms. Added to this are gear backlash, the static friction of the wheel, and the poor dynamics of the Dynamixel servo at small setpoint values (see Subsection 4.1.1). The fact that the vibration can be suppressed by applying slight mechanical damping to the system (by placing a finger on it) also confirms this interpretation. Such self-sustaining vibrations are also known as limit cycles in control engineering [25, Ch. 4.2].

Experiments with the PID controller have shown that an effective countermeasure is to reduce the amplitude of the oscillation so that the actuator no longer follows the setpoint, thereby dampening the oscillation. Therefore, gain scheduling is used again. If the influence of the angle ϑ and the angular velocity $\dot{\vartheta}$ is reduced to approximately half of the original controller matrix K, the amplitude and frequency of the oscillation in the rest position become so small that the movements are barely noticeable. In order to maintain the good dynamic characteristics of the original controller and at the same time take advantage of the slower parameter set in the rest position, suitable conditions must be found for selecting the active parameter set.

First, the transition from the fast to the slow parameter set is considered. In order for the vehicle to reliably recognize when it is in the rest position after compensating for a disturbance, a range must be defined that is considered to be the rest position. This range should be as small as possible in order to fully utilize the good dynamic characteristics of the fast parameter set, but as large as necessary to reliably recognize the rest position despite the limit cycles. Based on the amplitude and frequency of the oscillation in the rest position, limits can be set for the pitch angle and angular velocity that mark the entry into the rest position. In addition, a minimum duration is defined for which the vehicle must remain within this range to ensure that it is actually in the rest position and not in an overshoot.

Rapid switching back and forth between parameter sets can excite oscillations and negatively affect the dynamic behavior of the vehicle. To prevent minor disturbances from causing unnecessary changes, hysteresis is introduced. The range at which the vehicle recognizes that it is leaving the rest position is set larger than the range for recognizing entry. However, leaving the range once is sufficient to trigger the change to the fast parameter set.

A hard switch between the two parameter sets can lead to discontinuities in the control variable and cause unwanted vibrations and accelerations. Therefore, a switchover time is introduced during which the control variable of both parameter sets is linearly interpolated until the switch to the new parameter set is complete.

It is important to note that the weighting of the error integral must not be changed when switching between parameters, as otherwise the phenomenon shown in Figure 6.9 will occur in the event of permanent disturbances:

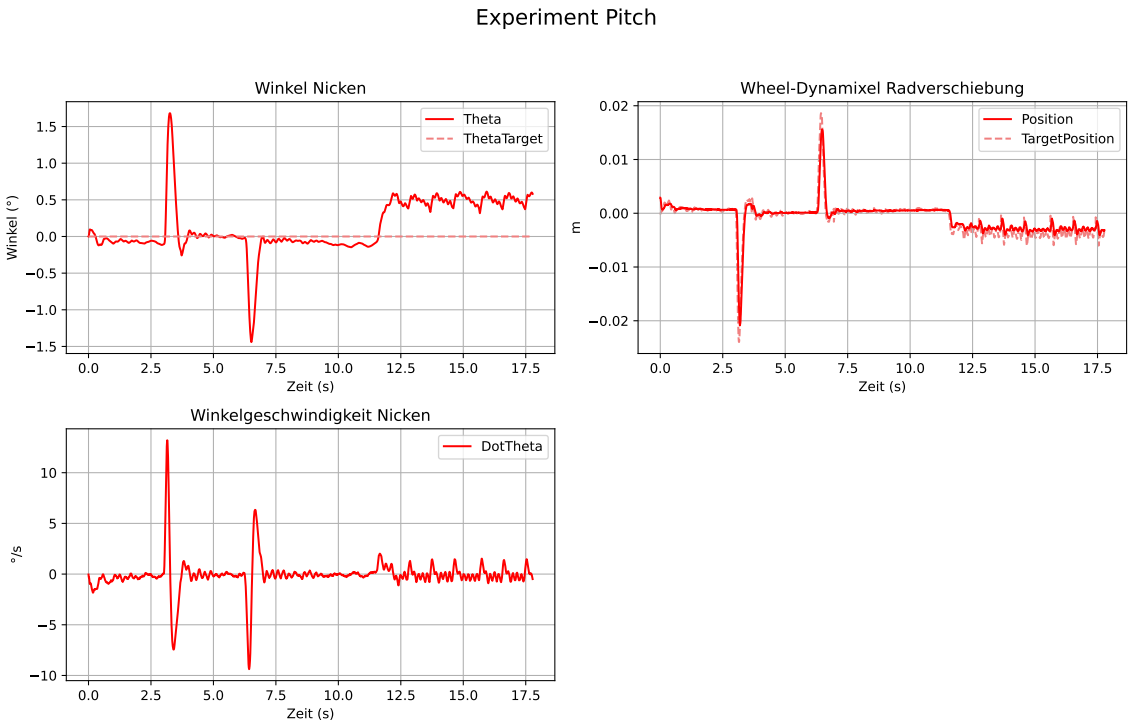


Figure 6.9: Gain scheduling with different weighting of the error integral

To compensate for the permanent disturbance, the controller switches to the fast parameter set due to the large error. As soon as the error is eliminated, the controller switches back to the slower parameter set. The reduced weighting of the error integral now causes the error to increase again because the control variable is reduced. As a result, the controller switches back to the faster parameter set and the process repeats itself, so that the vehicle cannot effectively compensate for permanent disturbances.
The behavior of the system with the state controller with gain scheduling can be seen in Figure 6.10:

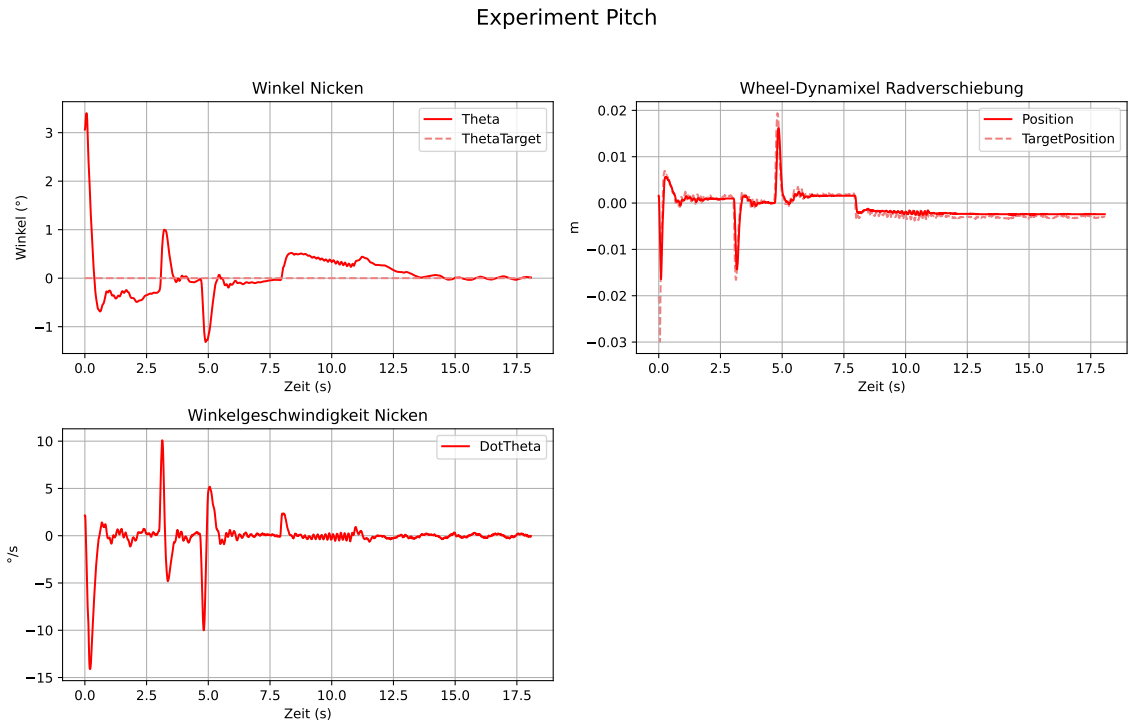


Figure 6.10: Pitch state controller with gain scheduling

It can be seen that the good dynamic behavior of the fast parameter set is largely retained, while the oscillations in the rest position are drastically reduced. The requirements for the controller design for the pitch subsystem are thus fulfilled. Furthermore, it was shown that the simplified model of the pitch movement accurately represents the dynamics of the system and that the behavior of the real vehicle matches the behavior of the model in the simulation.

6.3 Roll and yaw subsystem

As described in Chapter 3, the roll and yaw subsystem is a coupled differential equation system. However, only the stabilization of the roll motion is necessary for balancing, since the coupling between roll and yaw is canceled out at the operating point during balancing by the static friction of the wheel (see Chapter 3).

Although the system has only one input, the rotation of the gyroscope around the vertical axis, it has several outputs. Since the control variable is the angular velocity of the gyroscope around its vertical axis, it only acts in one direction until the mechanically permissible angle of rotation is reached. This means that only a limited reserve of control variable is available: after a deflection of approximately $\pm 15°$, no further control effect can be generated in the same direction. Permanent disturbances cannot therefore be compensated for without further measures. In reality, the vehicle is not ideally balanced, so compensating for this type of permanent disturbance is essential for balancing. The angle of the gyroscope must

therefore also be considered and controlled as an output of the system, in addition to the roll angle. This makes it an Single Input Multiple Output (SIMO) system. In addition to stabilizing the roll angle, the controller must adjust the roll angle based on the angle of the gyroscope so that the gyroscope can be rotated to the center. The active tilt of the vehicle allows gravity to compensate for the permanent disturbance.

There are two tasks that the controller must perform. The main task is to balance the vehicle. To do this, the vehicle must straighten up, suppress short disturbances, and compensate for permanent disturbances. No unwanted yaw movement may occur, as the vehicle should stand still or drive in a straight line. To achieve this, the angular velocity of the roll must not exceed a certain threshold (see Chapter 3). The controller design must therefore not be too aggressive, and robustness is a priority. In addition, the controller should be able to be used for cornering experiments. To do this, the controller must be able to follow a trajectory for the roll angle in order to realize a curve (see Chapter 7).

Classic PID controllers are Single Input Single Output (SISO) systems. In SIMO systems, the feedback of the various outputs when using a PID controller often leads to instabilities. Therefore, the use of an LQR controller is recommended, as these are generally designed for Multiple Input Multiple Output (MIMO) systems. While LQR controllers are primarily designed to stabilize a system around an operating point, PID controllers also enable trajectory tracking without any extensions to the controller structure. Therefore, the following sections will examine the stabilization of the system with both a PID controller and an LQR controller.

6.3.1 PID controller

Since PID controllers are only designed for SISO systems, a PID controller is first designed to control the roll angle. The angle of the gyroscope is initially ignored. This simplification results in an SISO system. An initial theoretical design of a PID controller for the reduced roll subsystem is performed in [33, S. 109] based on theoretical model assumptions. Since the parameters of the finished vehicle differ from the originally assumed values and the model has been expanded (actuator dynamics, better estimation of friction parameters, etc.), the control parameters $KP = 12;\ KD = 1.44;\ KI = 10.62$ determined there can only be used as a starting point for manual tuning in the simulation.

controller design roll angle

The simulation is calculated in the Python script *roll_PID.py*. First, the righting and a permanent disturbance are simulated. The permanent disturbance is essential because the vehicle is not perfectly balanced, and compensating for a permanent disturbance is a prerequisite for balancing. The system behavior for the control parameters $KP = 12;\ KD = 1.44;\ KI = 10.62$ can be seen in Figure 6.11:

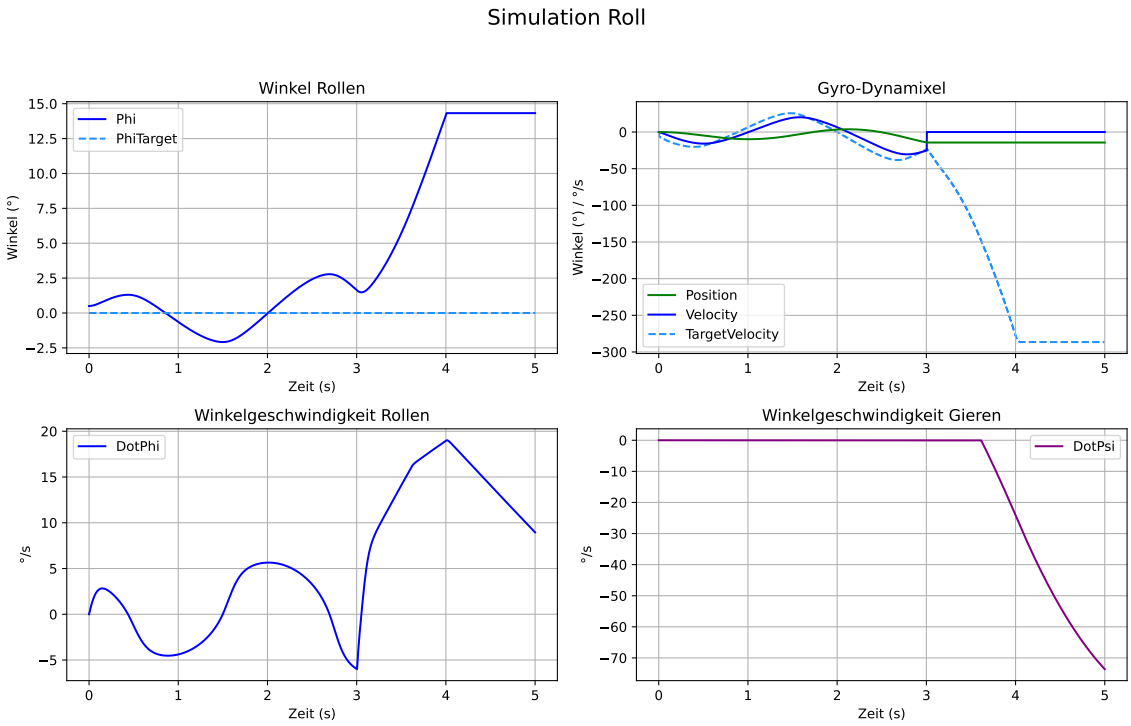


Figure 6.11: Roll simulation with $KP = 12;\ KD = 1.44;\ KI = 10.62$

The system oscillates and the controller is unable to stabilize the vehicle effectively. Therefore, the damping and proportional component are increased and the integral component is drastically reduced to improve dynamics. The following system behavior results for the control parameters $KP = 14;\ KD = 2.8;\ KI = 1.4$ (see Figure 6.12):

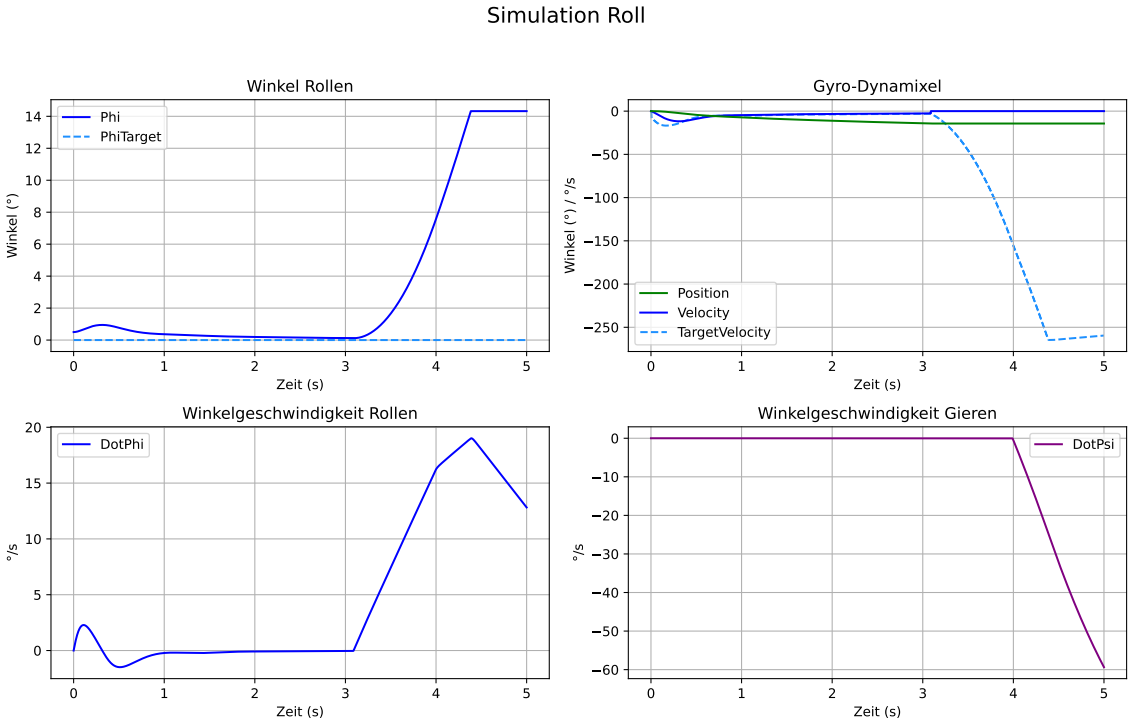


Figure 6.12: Roll simulation with $KP = 14;\ KD = 2.8;\ KI = 1.4$

The system no longer oscillates and the controller can effectively stabilize the system for a certain period of time. However, due to the permanent disturbance, a constant control variable is required. As a result, the gyroscope reaches its maximum deflection and the vehicle falls over. The system behavior can be further improved by slightly modifying the parameters to $KP = 15;\ KD = 3;\ KI = 3$:

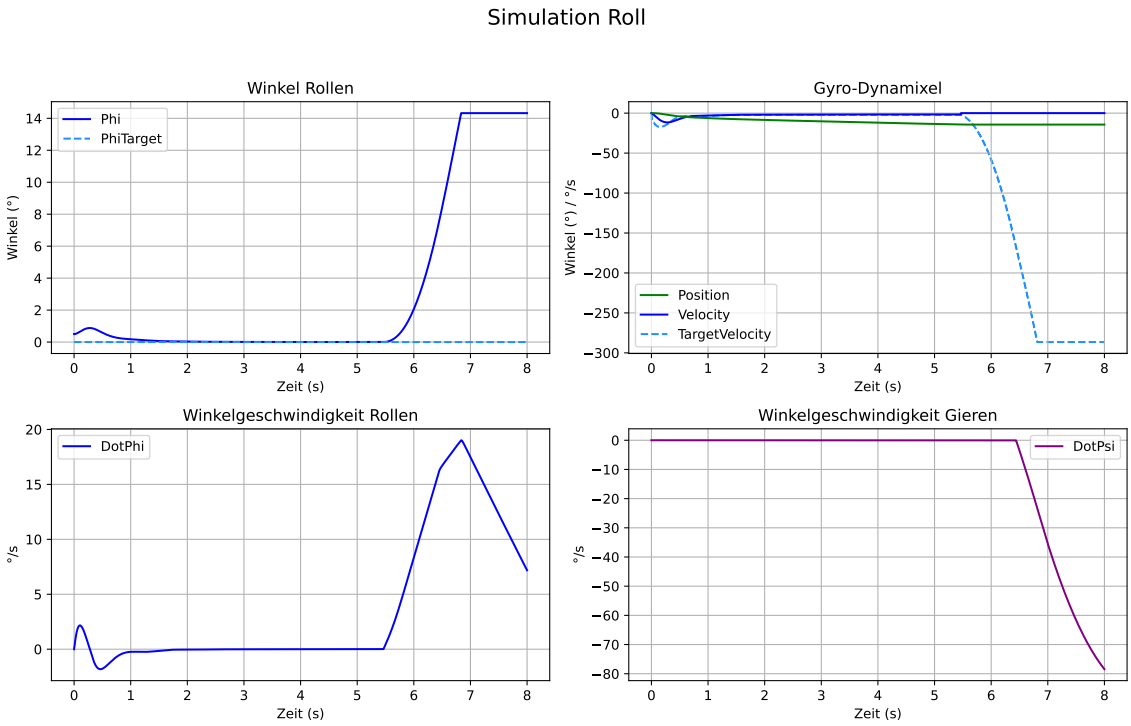


Figure 6.13: Roll simulation with $KP = 15$; $KD = 3$; $KI = 3$

Due to the more effective use of the control variable, the controller can bring the system to the setpoint more quickly and stabilize it above 5 s, while the previous parameter set already exhausts the maximum of the gyroscope after 3 s. In addition, the decoupling of the roll and yaw movements can be seen. As long as the vehicle is close to its rest position, roll movements do not lead to yaw movements. Only when the vehicle tips over and overcomes the static friction of the wheel is the system coupled again.

In order to compensate for permanent disturbances, the vehicle must adjust its roll angle so that the control variable causes the gyroscope to move toward the center. A simple version of this idea is implemented in [2]. Depending on the angle of the gyroscope, a fixed setpoint is specified that causes the vehicle to tip over in the other direction, so that the gyroscope must be turned in the opposite direction, back to the center. The setpoint can be specified based on the current angle of the gyroscope. The fixed setpoints between which the system switches must be adjusted to the disturbance. If the setpoints are too large, the vehicle cannot recover when switching and the gyroscope does not have enough reserves to right the vehicle again. If the setpoints are too small, the tilt is not sufficient to cause the vehicle to fall over to the other side, and the direction of rotation of the gyroscope is not reversed. In this case, too, the vehicle will fall over. Figure 6.14 shows how switching between fixed setpoints can stabilize the vehicle permanently:

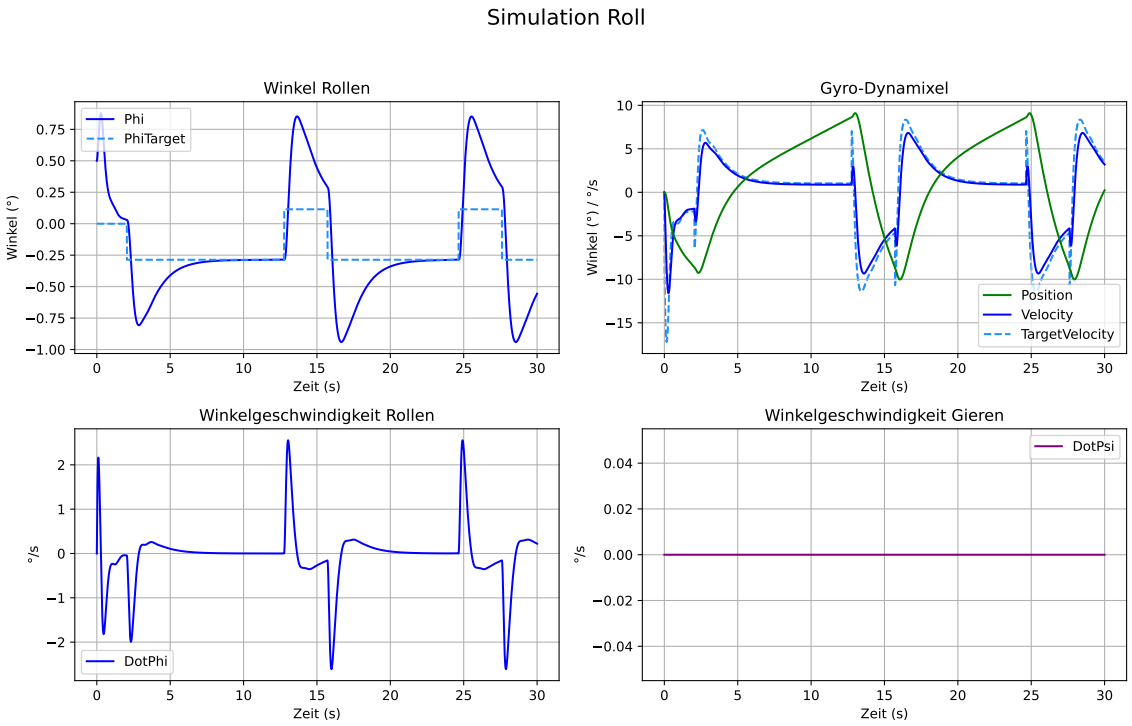


Figure 6.14: Roll simulation with $KP = 15;\ KD = 3;\ KI = 3$ with fixed setpoint

It can be seen that the disturbance actually causes the vehicle to fall in the positive roll direction. If the vehicle is deflected in the negative roll direction, the tilt compensates for part of the disturbance, and only a small control variable is required. However, if the vehicle is deflected in the positive direction, the tilt amplifies the disturbance, and a large control variable is required. As a result, the vehicle remains in the negative tilt position for much longer. This method only works for a very limited range of disturbances. In addition, an additional, short-term disturbance at the wrong moment is enough to cause the vehicle to fall over. Therefore, the system should be expanded back to an SIMO system and the angle of the gyroscope should be included in the control. This is to be implemented with a dynamic setpoint for the roll angle. A second PID controller is used to control the second output, the angle of the gyroscope, of the SIMO system. The second PID controller has the setpoint for the roll angle as its control variable and is thus cascaded with the PID controller for controlling the roll angle. The controller structure can be seen in Figure 6.15.

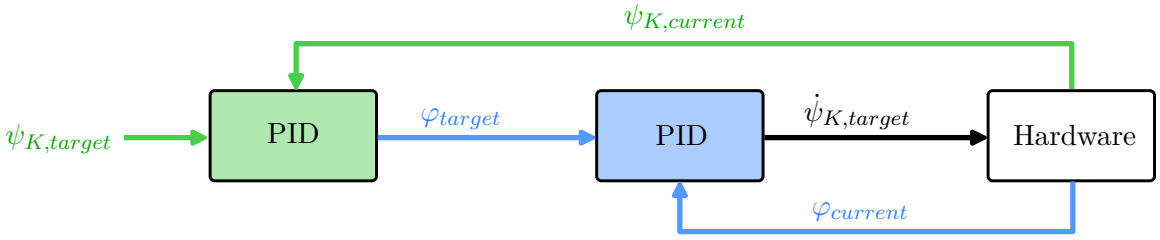


Figure 6.15: Structure of cascaded PID controllers for balancing

The design is determined experimentally using simulation. The parameters $KP = 0.02$; $KD = 0.003$; $KI = 0.01$ offer a compromise between robustness and dynamics. To reduce the phase shift, the input of the PID controller that controls the roll angle is modified. Normally, the control deviation and its derivative are calculated as follows:

$$e = \varphi_{ref} - \varphi \tag{6.39}$$

$$\dot{e} = \frac{e - e_{old}}{Ta} \tag{6.40}$$

Instead, the control deviations and derivative are now calculated as follows:

$$e = \varphi_{ref} - \varphi \tag{6.41}$$

$$\dot{e} = \dot{\varphi}_{ref} - \dot{\varphi} \tag{6.42}$$

This improves the control behavior of the PID controller. To determine the setpoint of the roll speed, the controller output of the second PID controller is derived. In addition to the permanent disturbance, a short-term disturbance is tested in the simulation. The result can be seen in Figure 6.16:

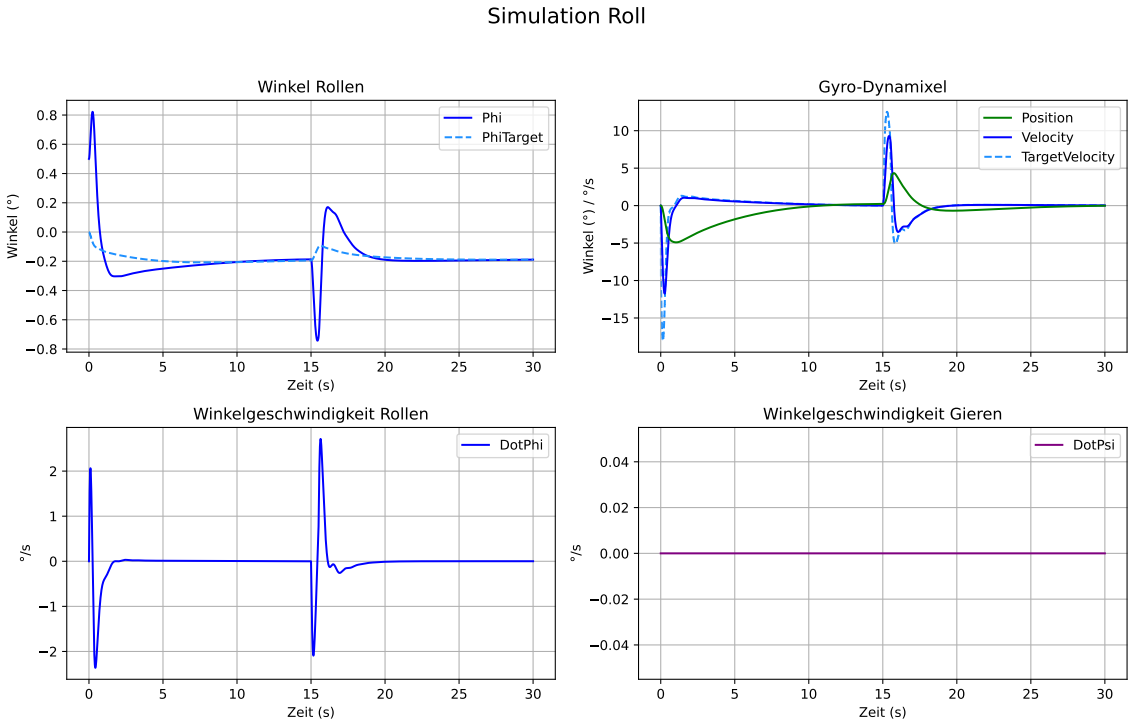


Figure 6.16: Roll simulation with $KP = 15$; $KD = 3$; $KI = 3$ with dynamic setpoint

This fulfills all requirements. The SIMO system with two outputs can be stabilized by using two cascaded PID controllers and can straighten up with realistic control variables, suppress short-term disturbances, and compensate for permanent disturbances. In addition, the system follows a specified trajectory with sufficient accuracy. The controller can be tested on the real system.

Real system control

The PID controller for the roll motion determined in the simulation is implemented on the vehicle. In order to compensate for permanent disturbances, two fixed setpoints are first specified, between which switching takes place depending on the angle of the gyroscope (as in the simulation in Figure 6.14). At the beginning of the experiment, the vehicle straightens up and then compensates for the permanent disturbance caused by the offset of the vehicle's center of gravity. The fixed setpoints are determined experimentally. This results in the following system behavior (see Figure 6.17):

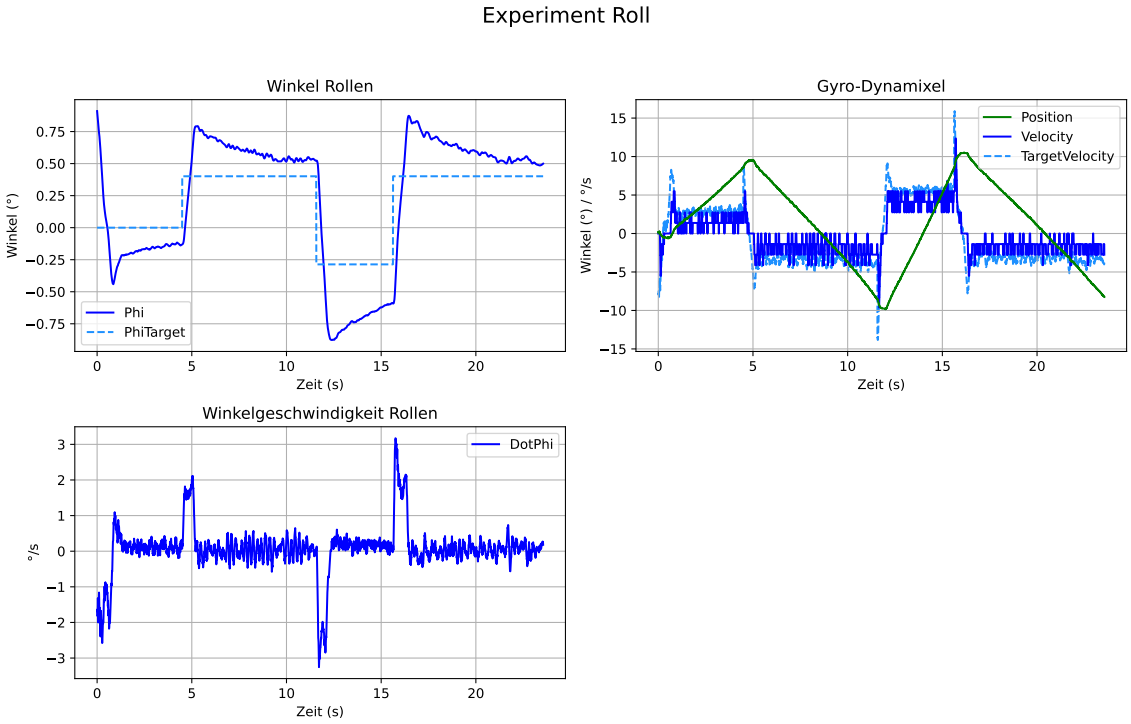


Figure 6.17: Rolling experiment with $KP = 15;\ KD = 3;\ KI = 3$ with fixed setpoint

The behavior of the real system matches the results of the simulation. The vehicle can be stabilized indefinitely and the dynamics are very similar to the simulation. In order to be able to react flexibly to any type of disturbance, the second PID controller is used to generate the setpoint (as in Figure 6.16). The result of the experiment can be seen in Figure 6.18.

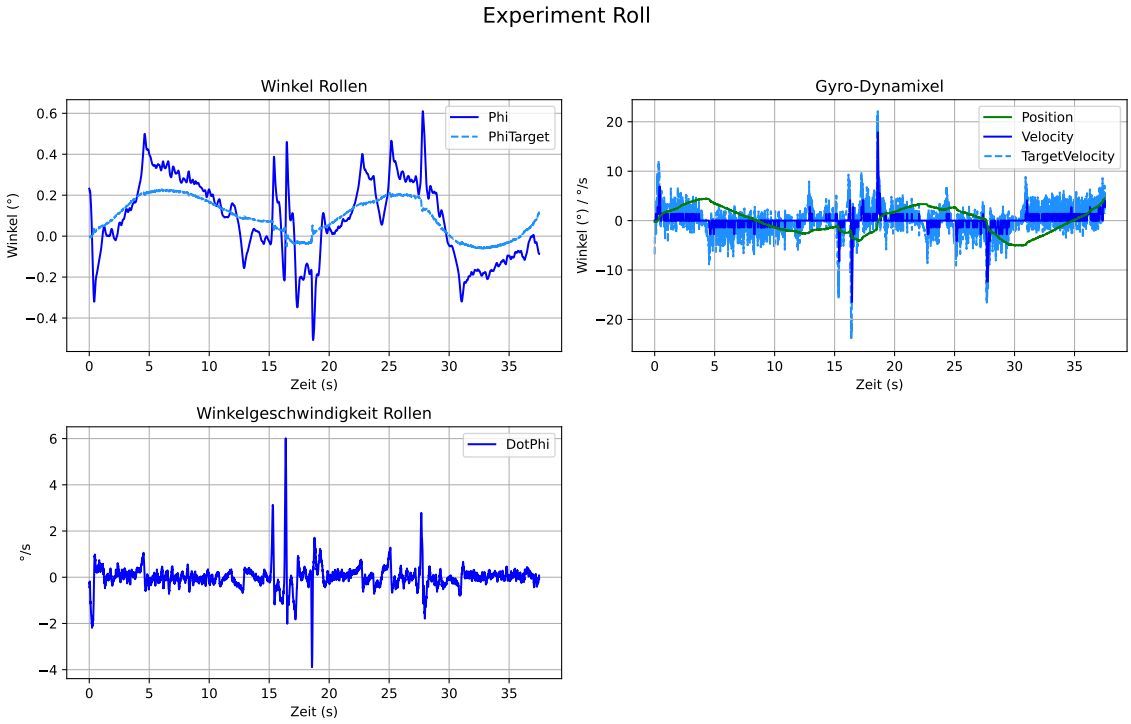


Figure 6.18: Rolling experiment with $KP = 15;\ KD = 3;\ KI = 3$ with dynamic setpoint

In addition to the permanent disturbance, several short-term disturbances affect the system, which are compensated without any problems. The correction and compensation of the permanent disturbance also takes place in accordance with the requirements. However, it is noticeable that the system oscillates at a very low frequency. Similar to the oscillation when controlling the pitch movement, this oscillation can be attributed to nonlinearities and inaccuracies caused by friction, play, and dead times. Due to the small amplitude $A \approx 0.1°$ and frequency $f \approx 0.05\,\text{Hz}$, these oscillations are barely visible to the naked eye and can be ignored in the context of the overall system. It has also been shown that the simplified model of the vehicle adequately represents the dynamics of the real system for the stabilization of roll movements and that the behavior of the real vehicle corresponds to the behavior of the model in the simulation.

6.3.2 State controller

Due to the advantages of a state controller in controlling SIMO systems mentioned in Section 6.3, it should be investigated whether a state controller can achieve better system behavior compared to a PID controller. The basic structure and the different types of state controllers are already described in Subsection 6.2.2. An LQR controller is also to be used for the roll and yaw subsystem.

Modeling in state space

The differential equation system Equation 3.8 must be linearized and is simplified by decoupling roll and yaw at the operating point for balancing. Using the small-angle approximation, the following linear system results:

$$J_{x,\tau} \cdot \ddot{\varphi} = J_G \cdot w_G \cdot \dot{\psi}_G + F_G \cdot h_0 \cdot \varphi \tag{6.43}$$

$$J_z \cdot \ddot{\psi} = 0 \tag{6.44}$$

The input of the system u is the angular velocity of the gyroscope. As already described, the angle of the gyroscope is also to be recorded as a state, as well as the error integral of the angle, in order to guarantee that the gyroscope remains in the center even in the event of permanent disturbances. The remaining states and their derivatives are selected as follows:

$$u = \dot{\psi}_G \tag{6.45}$$

$$x_1 = \varphi \tag{6.46}$$

$$x_2 = \dot{\varphi} \tag{6.47}$$

$$x_3 = \psi_G \tag{6.48}$$

$$x_4 = \int \psi_G \, dt \tag{6.49}$$

$$\dot{x}_1 = x_2 \tag{6.50}$$

$$\dot{x}_2 = \frac{1}{J_{x,\tau}} \cdot (J_G \cdot w_G \cdot u + F_G \cdot h_0 \cdot x_1) \tag{6.51}$$

$$\dot{x}_3 = u \tag{6.52}$$

$$\dot{x}_4 = x_3 \tag{6.53}$$

This results in the following system:

$$\begin{bmatrix} \dot{x}_1 \\ \dot{x}_2 \\ \dot{x}_3 \\ \dot{x}_4 \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 & 0 \\ \frac{F_G \cdot h_0}{J_{x,\tau}} & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} + \begin{bmatrix} 0 \\ \frac{J_G \cdot w_G}{J_{x,\tau}} \\ 1 \\ 0 \end{bmatrix} u \tag{6.54}$$

The state model is fully controllable (determined with the Python script *roll_analysis.py*).

Control and simulation in state space

The selection of weighting factors and the discrete controller design are analogous to Subsection 6.2.2. The values specified in Table 6.2 are used as maximum values:

Table 6.2: Maximal zulässige Werte der Zustände des Teilsystems Rollen

Zustand	Physikalische Größe	Maximaler Wert
x_1	φ	$0.02\,\mathrm{rad}$
x_2	$\dot{\varphi}$	$0.02\,\mathrm{rad\,s^{-1}}$
x_3	ψ_G	$0.17\,\mathrm{rad}$
x_4	$\int \psi_G$	$1\,\mathrm{rad\,s}$
u	$\dot{\psi}_G$	$5\,\mathrm{rad\,s^{-1}}$

This results in the following weightings:

$$Q = \begin{bmatrix} \frac{1}{0.02} & 0 & 0 & 0 \\ 0 & \frac{1}{0.02} & 0 & 0 \\ 0 & 0 & \frac{1}{0.17} & 0 \\ 0 & 0 & 0 & \frac{1}{1} \end{bmatrix}, \quad R = \frac{1}{5} \tag{6.55}$$

These weightings can be used as the initial state for further optimizations. For this purpose, the behavior of the system with state controller is simulated in the same Python script *roll_LQR.py*. As before, a righting process, a permanent disturbance, and a short-term disturbance are simulated. By gradually adjusting Q and R, the following weighting results in a controller with a realistic control variable response and good dynamics:

$$Q = \begin{bmatrix} \frac{1}{0.02} & 0 & 0 & 0 \\ 0 & \frac{1}{0.02} & 0 & 0 \\ 0 & 0 & \frac{1}{0.17} \cdot 10 & 0 \\ 0 & 0 & 0 & \frac{1}{1} \end{bmatrix}, \quad R = \frac{1}{5} \cdot 100, \quad K = \begin{bmatrix} 21.876\,329\,95 \\ 3.161\,821\,22 \\ -1.831\,431\,59 \\ -0.214\,242\,86 \end{bmatrix} \tag{6.56}$$

The simulated system behavior can be seen in Figure 6.19:

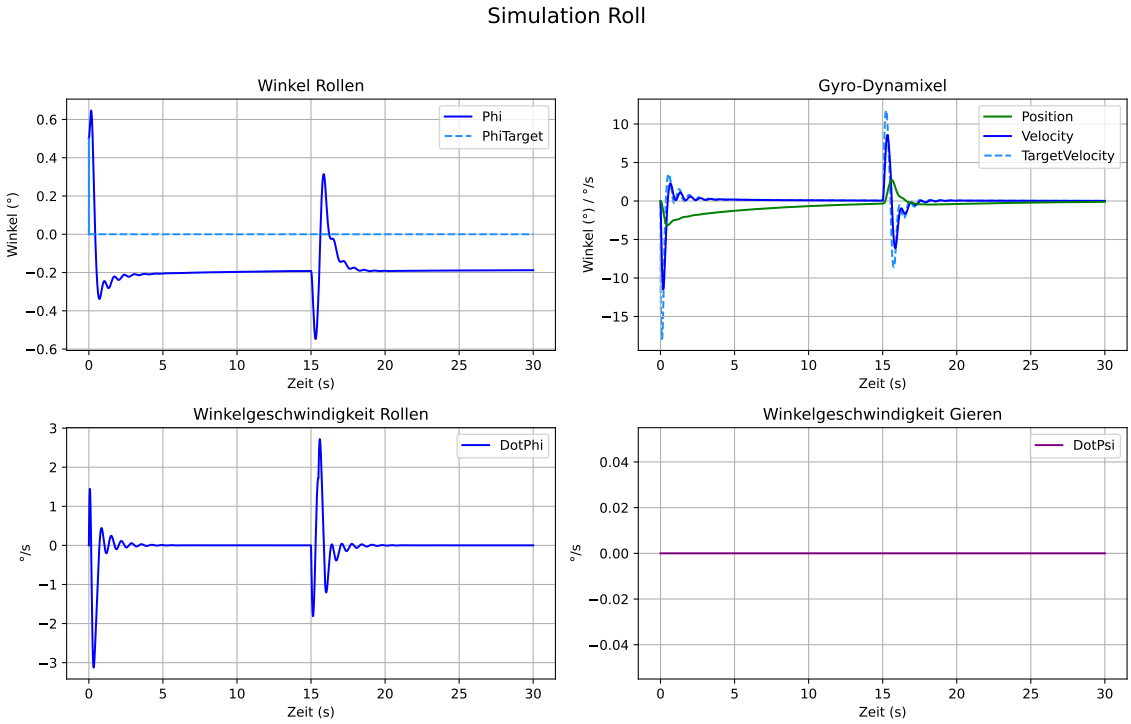


Figure 6.19: Roll state controller without modeling of the actuator

The simulation shows the decoupling due to static friction, as the vehicle does not rotate around the vertical axis despite the rolling motion. However, the system oscillates. This is due to the additional phase shift caused by the actuator, as the actuator cannot follow the specified trajectory quickly enough (see gyro-dynamixel plot in Figure 6.19). This problem can be solved in two ways: Either the controller is slowed down so much that

the delay caused by the actuator becomes irrelevant, or the dynamics of the actuator are included in the state model. However, the LQR approach does not allow the controller to be slowed down without drastically worsening the dynamic properties. Therefore, the actuator dynamics determined in Subsection 4.1.1 are included in the state model as a PT1 element. The angular velocity $\dot{\psi}_G$ becomes the new state. The input of the system u is now the control variable $\dot{\psi}_{Gcmd}$ for the actuator. The relationship between the angular velocity of the gyroscope and the control variable is described by a PT1 element:

$$K \cdot \dot{\psi}_{Gcmd} = T \cdot \dot{\psi}_G + \psi_G \tag{6.57}$$

This results in the extended system:

$$u = \dot{\psi}_{Gcmd} \tag{6.58}$$
$$x_1 = \varphi \tag{6.59}$$
$$x_2 = \dot{\varphi} \tag{6.60}$$
$$x_3 = \psi_G \tag{6.61}$$
$$x_4 = \int \psi_G \, dt \tag{6.62}$$
$$x_5 = \dot{\psi}_G \tag{6.63}$$
$$\dot{x}_1 = x_2 \tag{6.64}$$
$$\dot{x}_2 = \frac{1}{J_{x,\tau}} \cdot (J_G \cdot w_G \cdot x_5 + F_G \cdot h_0 \cdot x_1) \tag{6.65}$$
$$\dot{x}_3 = x_5 \tag{6.66}$$
$$\dot{x}_4 = x_3 \tag{6.67}$$
$$\dot{x}_5 = \frac{1}{T} \cdot (K \cdot u - x_5) \tag{6.68}$$

This results in the following system:

$$\begin{bmatrix} \dot{x}_1 \\ \dot{x}_2 \\ \dot{x}_3 \\ \dot{x}_4 \\ \dot{x}_5 \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ \frac{F_G \cdot h_0}{J_{x,\tau}} & 0 & 0 & 0 & \frac{J_G \cdot w_G}{J_{x,\tau}} \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & -\frac{1}{T} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ \frac{K}{T} \end{bmatrix} u \tag{6.69}$$

The best dynamic behavior in the simulation was achieved with the following weighting factors:

$$Q = \begin{bmatrix} \frac{1}{0.02} & 0 & 0 & 0 & 0 \\ 0 & \frac{1}{0.02} & 0 & 0 & 0 \\ 0 & 0 & \frac{1}{0.17} \cdot 10 & 0 & 0 \\ 0 & 0 & 0 & \frac{1}{1} \cdot 100 & 0 \\ 0 & 0 & 0 & 0 & \frac{1}{5} \end{bmatrix}, \quad R = \frac{1}{5} \cdot 1000, \quad K = \begin{bmatrix} 41.570\,314\,79 \\ 5.308\,060\,25 \\ -1.596\,561\,93 \\ -0.683\,466\,2 \\ 1.966\,801\,53 \end{bmatrix} \tag{6.70}$$

The simulated system behavior can be seen in Figure 6.20:

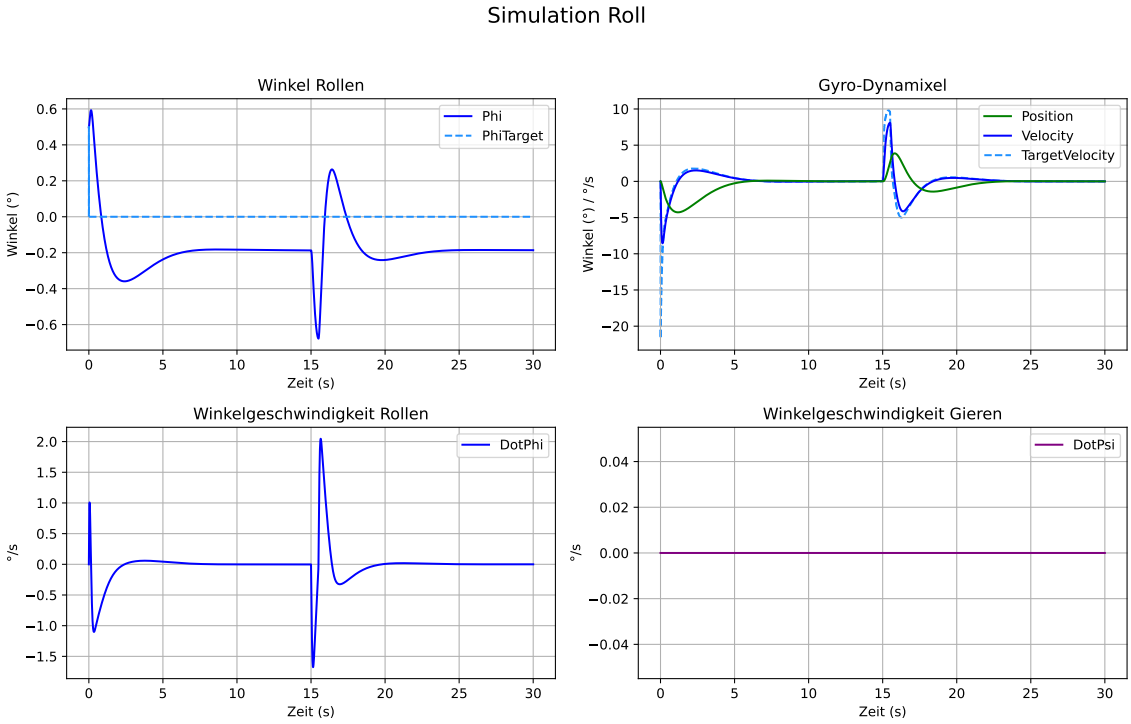


Figure 6.20: Roll state controller with modeling of the actuator

The system behavior has improved significantly and the oscillations no longer occur. However, it can be seen that the controller sends a large setpoint jump to the actuator during uprighting. Subsection 4.1.1 shows that the actuator can oscillate when there are large setpoint jumps, making the system unstable. In order to maintain good disturbance behavior but prevent the large setpoint jump during righting, a trajectory is specified. State controllers are actually designed to stabilize a system in a rest position and cannot follow a setpoint without further measures. However, a similar behavior can be achieved by using a trick. The state vector is manipulated so that the roll angle is no longer used directly as a state, but rather the difference between the current roll angle and the specified roll angle. To avoid large jumps, the setpoint is filtered with a first-order low-pass filter. The implementation in the simulation is shown in excerpts in Listing 7:

```
ctrl_phi = 0
ctrl_phi_target = 0
ctrl_w = 0
ctrl_GF = 0.98
...
# Initial condition
phi[0] = np.deg2rad(0.5)
ctrl_phi_target = phi[0]
ctrl_w = 0
for n in range(1, steps):
    ...
    ctrl_phi_target = (1-ctrl_GF)*ctrl_w + ctrl_GF*ctrl_phi_target
    ctrl_dot_psi_K_target = -np.clip(Kd[0, 0]*(ctrl_phi-ctrl_phi_target)
                                     + Kd[0, 1]*ctrl_dot_phi
                                     + Kd[0, 2]*ctrl_psi_K
                                     + Kd[0, 3]*ctrl_psi_K_sum
                                     + Kd[0, 4]*ctrl_dot_psi_K,
                                     -dot_psi_K_max, dot_psi_K_max)
    ...
```

Listing 7: Rollen Trajektorie zum Aufrichten

The system behavior can be seen in Figure 6.21:

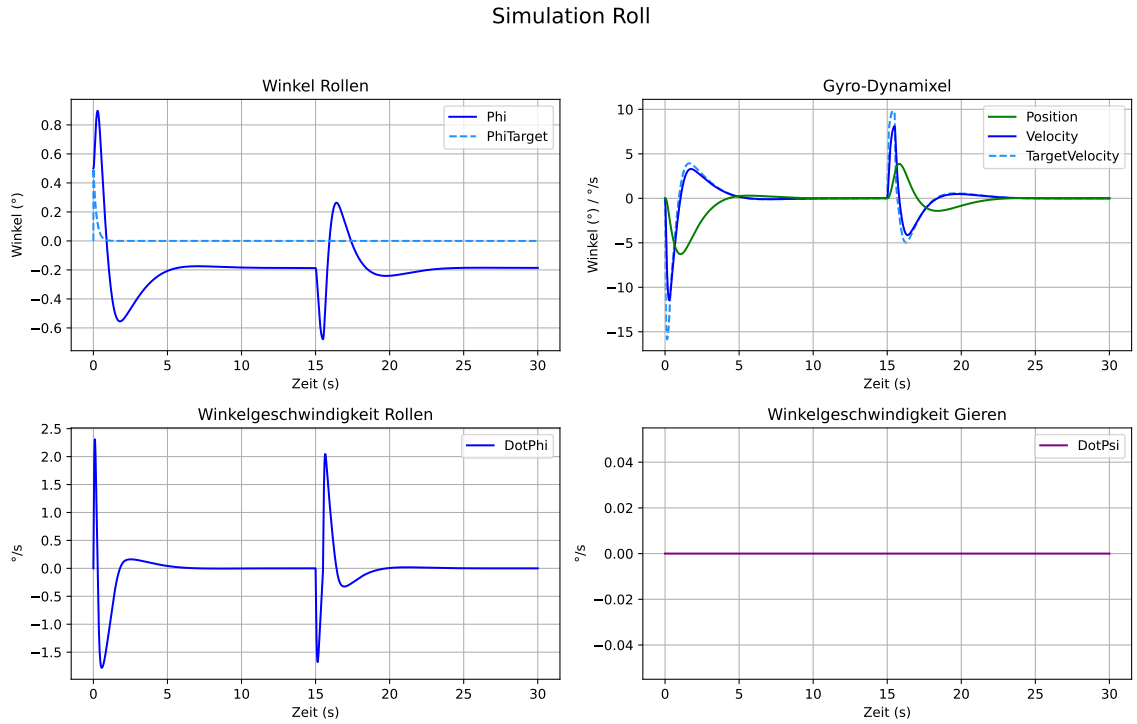


Figure 6.21: Roll state controller with modeling of the actuator and trajectory

The LQR controller basically meets all requirements for balancing. Uprighting is achieved

with a realistic control variable effort, short-term disturbances are effectively suppressed, and permanent disturbances can be compensated for by a targeted deviation of the roll angle. The controller automatically balances the vehicle by tilting it in response to the disturbance. This means that no permanent control variable is required and the gyroscope can be kept in the center.

A direct comparison of the simulation of the state controller (Figure 6.21) and the PID controller (Figure 6.16) shows that the dynamic behavior of the state controller is only slightly better than that of the PID controller. The PID controller also has the advantage of being able to follow trajectories or setpoint jumps without further measures—a requirement for performing curve driving experiments. The state controller, on the other hand, attempts to control all states to zero. Even with the trick of manipulating the state, permanent control deviations remain due to the lack of an integrator for the roll angle. However, an integrator must not be introduced, as it is precisely this permanent deviation that is necessary to permanently balance the vehicle against disturbances by means of a slight tilt.

In order to still enable different setpoints to be achieved with a state controller, a second, specially adapted controller would therefore be necessary, which does not take the gyroscopic angle and its integrator into account, but instead contains an integrator for the roll angle. The PID controller implicitly has this division due to its separate structure for setpoint and roll angle control. For this reason, the PID controller is used in the real vehicle.

6.4 Overall system result

Both the pitch subsystem and the roll and yaw subsystem were successfully stabilized.
In the pitch subsystem, the state controller offers an advantage over the PID controller due to its superior dynamics. Gain scheduling is also used to reduce the oscillations of the system in the rest position, known as limit cycles. The system can effectively suppress short disturbances, compensate for long-term disturbances, and right the vehicle. This avoids strong accelerations and angular velocities in order to reduce disturbances to the sensors and the roll and yaw subsystem.

The roll and yaw subsystem can be considered decoupled for balancing purposes due to the static friction of the tire. Two PID controllers are used to stabilize the vehicle as an SIMO system with two outputs. One PID controller regulates the roll movement, while the other PID controller specifies a trajectory that ensures that the angle of the gyroscope remains within the permissible range. This allows the system to right itself, suppress short-term disturbances, and compensate for long-term disturbances. By splitting the two controllers, a desired roll angle can also be achieved without changing the controller structure. This capability is required for cornering. In addition, a state controller was designed to stabilize the vehicle. Since the state controller did not achieve any significant advantages in terms of system dynamics and is unable to follow setpoint specifications due to the controller structure, the vehicle is stabilized with the PID controller.

Both subsystems can be stabilized with the designed controllers. Therefore, the vehi-

cle should now be considered as a complete system. Figure 6.22 shows all system variables when balancing in one place. The vehicle must right itself, compensate for several short disturbances, and compensate for the permanent disturbances caused by the inaccuracies of the vehicle.

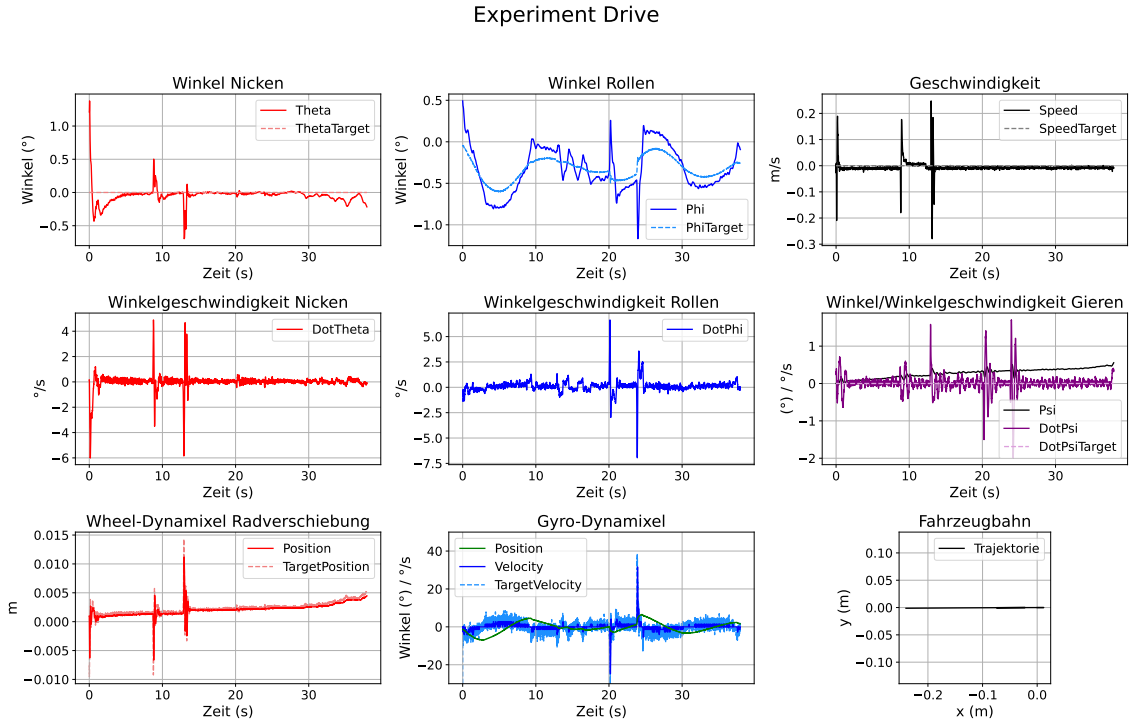


Figure 6.22: System test Vehicle balancing

The behavior of the individual subsystems in the overall system corresponds to their behavior when considered in isolation. It can also be seen that the subsystems have little influence on each other, which is also part of the simplifications made. The yaw movement also behaves as expected. When compensating for disturbances in the roll movement, there are minimal deviations in the yaw movement, but these do not significantly change the orientation of the vehicle. This shows that the decoupling of the roll and yaw subsystems at the operating point for balancing is as expected.
The fluctuations in the vehicle's speed are noticeable. Although the vehicle is actually stationary, the wheel moves several times. This is due to the pitch control, which shifts the wheel relative to the platform. Since the inertia of the wheel is lower than that of the platform, it is mainly the wheel that moves.

To illustrate the movements of the vehicle, the yaw angle is calculated in post-processing from the angular velocity of the yaw movement. Since the vehicle is always in the small-angle approximation range, it can be assumed that the yaw movement in the vehicle coordinate system corresponds to the yaw movement in the world coordinate system. Due to the bias of the gyroscope data, the yaw angle drifts over time, giving the impression that the vehicle is slowly turning, even though it is actually stationary. The same applies to the trajectory.

It is calculated in post-processing from the velocity vector. The magnitude is known from the current velocity, and the direction is determined from the current yaw angle. The ADC converter for determining the current velocity is also subject to bias, so that the trajectory also gives the impression that the vehicle is moving slowly, even when it is stationary. When the vehicle is in motion, these errors are negligible.

When the vehicle is in motion, the static friction of the wheel around the vertical axis decreases significantly. In addition, the effects of the rolling wheel were negligible in the modeling. Therefore, tests should be carried out to determine whether the vehicle can also be stabilized while driving. To do this, the vehicle is straightened up, accelerated to a specified speed, and then braked to a standstill. This results in the following behavior of the entire system (see Figure 6.23):

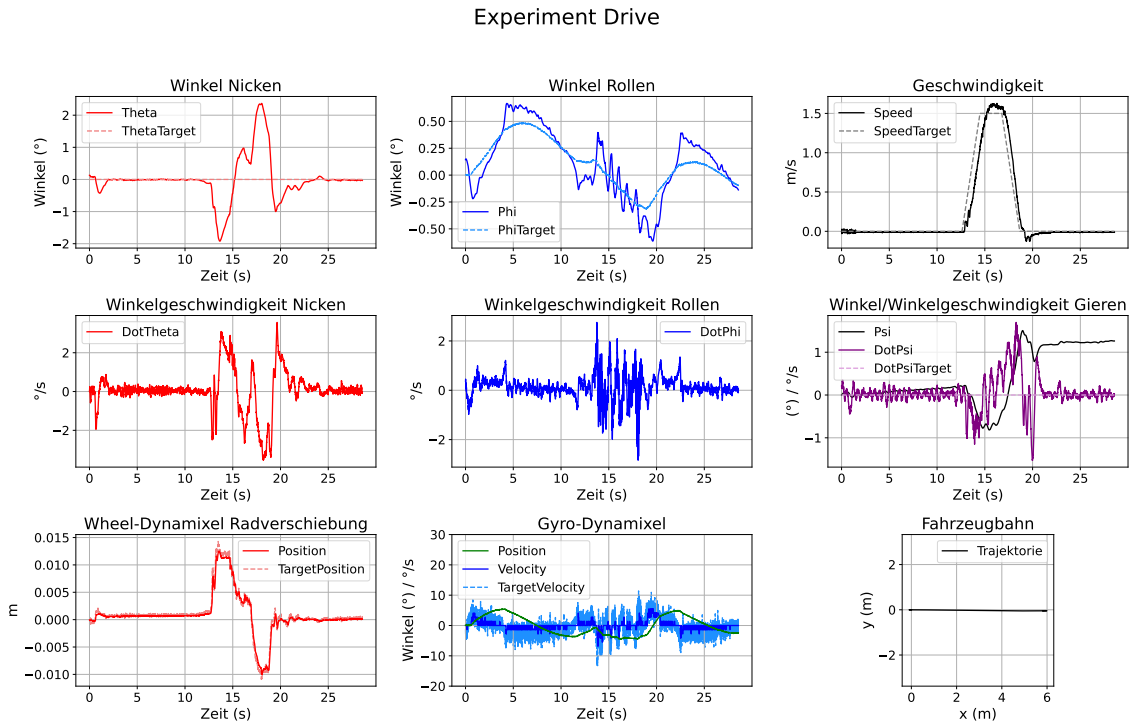


Figure 6.23: System test: Vehicle driving in a line

The vehicle can also be successfully stabilized while driving. As expected, greater yaw movements occur during driving than when stationary. However, the friction of the wheel is sufficient to cause the yaw movement to converge to zero. However, the system is significantly more sensitive when driving than when stationary. If the vehicle is not properly balanced, it can balance without any problems when stationary by tilting. While driving, however, the rolling constraint forces the wheel onto a circular path [43][4, S, 16ff], causing the vehicle to make an unwanted turn. This removes the decoupling between rolling and yawing, and the vehicle can no longer be effectively stabilized. However, if the vehicle is sufficiently well balanced, the tilt is so small that this effect does not occur. The vehicle can successfully drive straight ahead.

A comparison of the simulated system behavior with the behavior of the real system has shown that the simplifications made in the modeling in Chapter 3 sufficiently represent the dynamics relevant for balancing. Using the model and simulation, robust controllers were designed for both subsystems, whose dynamic behavior was validated on the real vehicle. Thus, the *Monowheeler* can be successfully stabilized.

7 Control and trajectories for cornering

While the previous chapters focused on stabilizing the vehicle, this chapter looks at cornering. The aim is to investigate whether basic maneuvers for changing direction are possible with the available hardware.

It should be noted that the work is deliberately based on a simplified model of the vehicle (see Chapter 3) and is limited to classic control strategies such as PID and LQR controllers. PID controllers are not actually designed for SIMO systems, while LQR controllers are only intended for stabilization and not for following trajectories. Modern methods suitable for this problem area, such as nonlinear or linear model predictive controllers for trajectory tracking (NMPC or LTV-MPC) or advanced trajectory planners, are not considered here as they go beyond the scope of this work. Instead, a proof of concept will be provided to show that the system is fundamentally capable of performing complex maneuvers such as changes of direction.

This chapter is therefore intended as an exploratory approach, primarily to provide practical evidence that the developed vehicle is fundamentally capable of cornering.

7.1 Direct control concepts

First, we will examine whether a classic controller can be designed that can control the coupled subsystem of rolling and yawing. This would allow the vehicle to actively navigate curves. Since this is an SIMO system, because both rolling and yawing must be controlled, an LQR controller should be used. To do this, the differential equation system Equation 3.8 must be linearized. Using the small-angle approximation, the following linear system results:

$$J_{x,\tau} \cdot \ddot{\varphi} = J_G \cdot w_G \cdot (\dot{\psi} + \dot{\psi}_G) + F_G \cdot h_0 \cdot \varphi + m_{total} \cdot v \cdot \dot{\psi} \cdot h_0 \tag{7.1}$$

$$J_z \cdot \ddot{\psi} = -J_G \cdot w_G \cdot \dot{\varphi} \tag{7.2}$$

The model is transferred to the state space in the same way as subsubsection 6.3.2, only with the extension to the yaw movement. The input of the u system is the angular velocity of the gyroscope. As already described, the angle of the gyroscope is also to be recorded as a state, as well as the error integral of the angle, in order to guarantee that the gyroscope remains in the center even in the event of permanent disturbances. The remaining states and their derivatives are selected as follows:

$$u = \dot{\psi}_G \tag{7.3}$$

$$x_1 = \varphi \tag{7.4}$$

$$x_2 = \dot{\varphi} \tag{7.5}$$

$$x_3 = \dot{\psi} \tag{7.6}$$
$$x_4 = \psi_G \tag{7.7}$$
$$x_5 = \int \psi_G \, dt \tag{7.8}$$
$$\dot{x}_1 = x_2 \tag{7.9}$$
$$\dot{x}_2 = \frac{1}{J_{x,\tau}} \cdot (J_G \cdot w_G \cdot (x_3 + u) + F_G \cdot h_0 \cdot x_1 + m_{total} \cdot v \cdot x_3 \cdot h_0) \tag{7.10}$$
$$\dot{x}_3 = -\frac{1}{J_z} \cdot (J_G \cdot w_G \cdot x_2) \tag{7.11}$$
$$\dot{x}_4 = u \tag{7.12}$$
$$\dot{x}_5 = x_4 \tag{7.13}$$

This results in the following system:

$$\begin{bmatrix} \dot{x}_1 \\ \dot{x}_2 \\ \dot{x}_3 \\ \dot{x}_4 \\ \dot{x}_5 \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ \frac{F_G \cdot h_0}{J_{x,\tau}} & 0 & \frac{J_G \cdot w_G + m_{total} \cdot v \cdot h_0}{J_{x,\tau}} & 0 & 0 \\ 0 & -\frac{J_G \cdot w_G}{J_z} & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{bmatrix} + \begin{bmatrix} 0 \\ \frac{J_G \cdot w_G}{J_{x,\tau}} \\ 0 \\ 1 \\ 0 \end{bmatrix} u \tag{7.14}$$

As described in Subsection 6.2.2, the controllability of this state model is checked in the Python script *roll_yaw_analysis.py*. This shows that only four of the five states are controllable. The uncontrollable state is the yaw movement. The system is therefore not fully controllable, and a direct state controller such as LQR can stabilize the system but cannot directly influence the yaw movement. Even extending the model to fully capture the dynamics of cornering does not change the controllability (see [14, S. 4]).

7.2 Trajectory-based control

Since direct control of the yaw movement in the overall system is not practicable within the scope of this work, the coupling between roll and yaw must be used to specifically influence the yaw movement. As described in Chapter 3, a roll movement generates a moment around the vehicle's vertical axis due to the gyroscopic effect. A targeted roll movement must briefly overcome the static friction of the wheel so that a yaw movement is caused. The roll speed is then adjusted so that the vehicle returns to the operating point for balancing ($\varphi \approx 0,\ \dot{\psi} \approx 0$). During this maneuver, the vehicle must not leave the permissible range of the roll angle and the gyroscopic angle, as otherwise the vehicle can no longer be stabilized. This goal can theoretically be achieved by two types of trajectory specification, which are presented below.

The trajectories of the rolling motion when cornering and balancing continue to be implemented by the PID controller designed in Section 6.3 for controlling the rolling motion.

7.2.1 Passive Trajectory

Chapter 3 describes the self-stabilizing effect of a gyroscope in space. Theoretically, this effect can also be used for cornering. To do this, the vehicle is deliberately made to tip over. The gyroscope is then fixed by the actuator so that the gyroscopic moments are transferred directly to the vehicle. The gyroscope generates a moment around its vertical axis due to the tilting movement. This causes the vehicle to start cornering. The rotation around the vertical axis in turn generates a gyroscopic moment that counteracts the original rolling movement. If the gyroscopic moments are large enough, the tilting movement of the vehicle is slowed down until a balance is achieved in which the vehicle maintains its roll angle as far as possible while cornering. The vehicle can then be righted again by specifically controlling the gyroscope. Since there is no active control of the system after the rollover has been initiated, this procedure is referred to as passive. In the simulation, the system behaves as expected (see Figure 7.1):

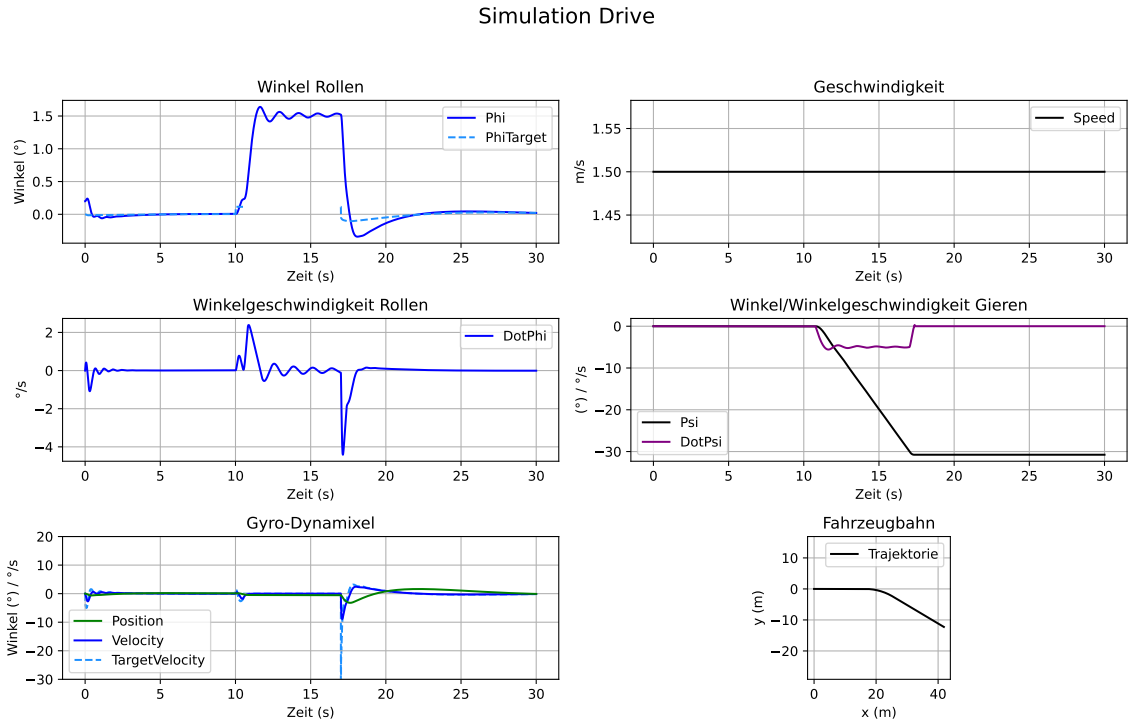


Figure 7.1: Simulation of vehicle cornering through passive trajectory

At $t = 10\,\mathrm{s}$, the vehicle is tilted by a setpoint value for the roll angle. A short time later, the controller is switched off and the actuator fixes the gyroscope. The vehicle then tilts until it reaches the equilibrium described above. The vehicle can then be righted. In theory, this allows for almost unlimited changes in direction, as no control variable is required when cornering, meaning that the gyroscope does not leave its permissible angle range.

However, this principle cannot be directly applied to the *Monowheeler*. In order to stabilize the vehicle within an angle range from which it can right itself again, a significantly higher

gyroscope speed is required. Simulation and observations of the real system show that the gyroscope speed would have to be increased from the current value of 5000 rpm to at least 7000 rpm. This is currently not possible without further measures due to the high power requirements of the drive. In addition, the roll and yaw movements are uncontrolled, meaning that the rotation rate around the vertical axis, and thus the curve radius, can hardly be influenced. Furthermore, the system is unable to respond to disturbances or environmental factors in this way, making it less robust.

7.2.2 Active trajectory

Since curve driving is not practical with the current setup due to passive specification of the roll movement, and direct controllability of the yaw movement is also desired, a trajectory for the roll movement should be actively specified.

Fixed trajectory

The simplest trajectory for the roll movement with which a yaw movement can be generated is the specification of a fixed roll speed. This is to be tested on the real system to see whether the concept of active control of the roll movement for cornering works in principle. First, the vehicle is accelerated to a speed of $v = 1.5\,\mathrm{m\,s^{-1}}$. Then, the setpoint for the roll movement is no longer generated by the PID controller for balancing, but a fixed roll speed is specified. At the end, the vehicle is righted again by setting the setpoint for the roll angle back to the PID controller for balancing. The results of the experiment can be seen in Figure 7.2.

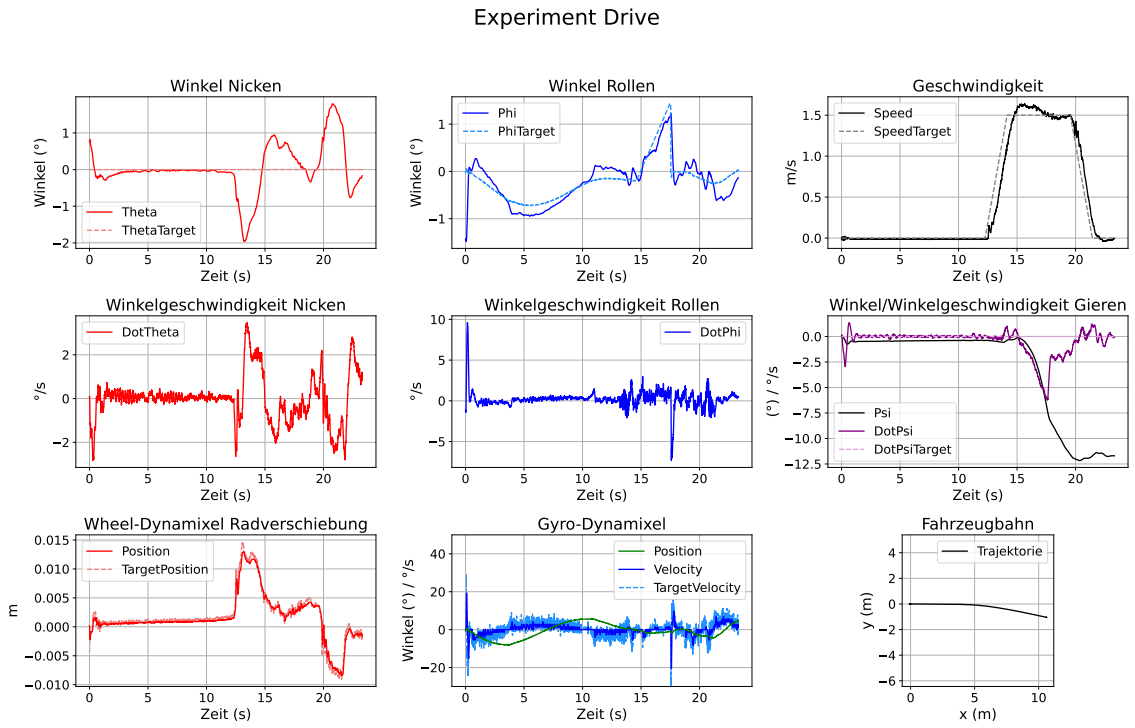


Figure 7.2: System test vehicle cornering with fixed roll trajectory

With the concept presented, the *Monowheeler* can be used to actively corner. The specified roll speed briefly overcomes the static friction of the wheel and causes a targeted rotation around the vertical axis. However, cornering with this method is not very robust. Small differences in the balance of the vehicle and the initial conditions have a significant impact on cornering, sometimes even destabilizing the system.

Controller-based trajectory

In order to make cornering behavior more robust and easier to control, a controller-based trajectory is used instead of a fixed trajectory. For this purpose, a PID controller is designed to control the yaw rate. The manipulated variable is the roll rate, from which a trajectory for the roll movement can then be generated. This results in a cascaded controller structure (see Figure 7.3) that is similar to the control for balancing.

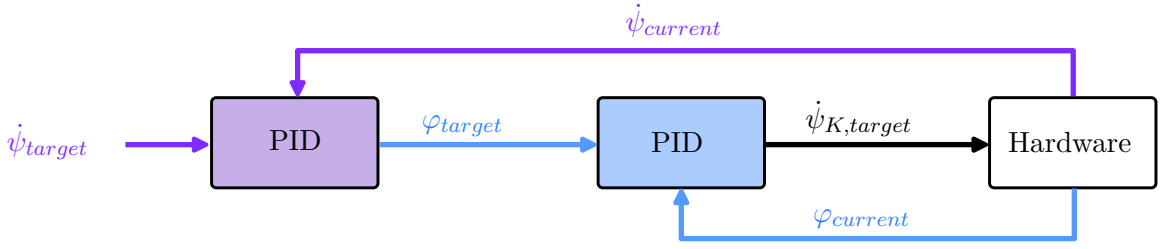


Figure 7.3: Structure of cascaded PID controller for cornering

The design is determined experimentally in the simulation. Figure 7.4 shows the behavior of the system in the simulation with the controller parameters $KP = 0.8$; $KD = 0.15$; $KI = 0$.

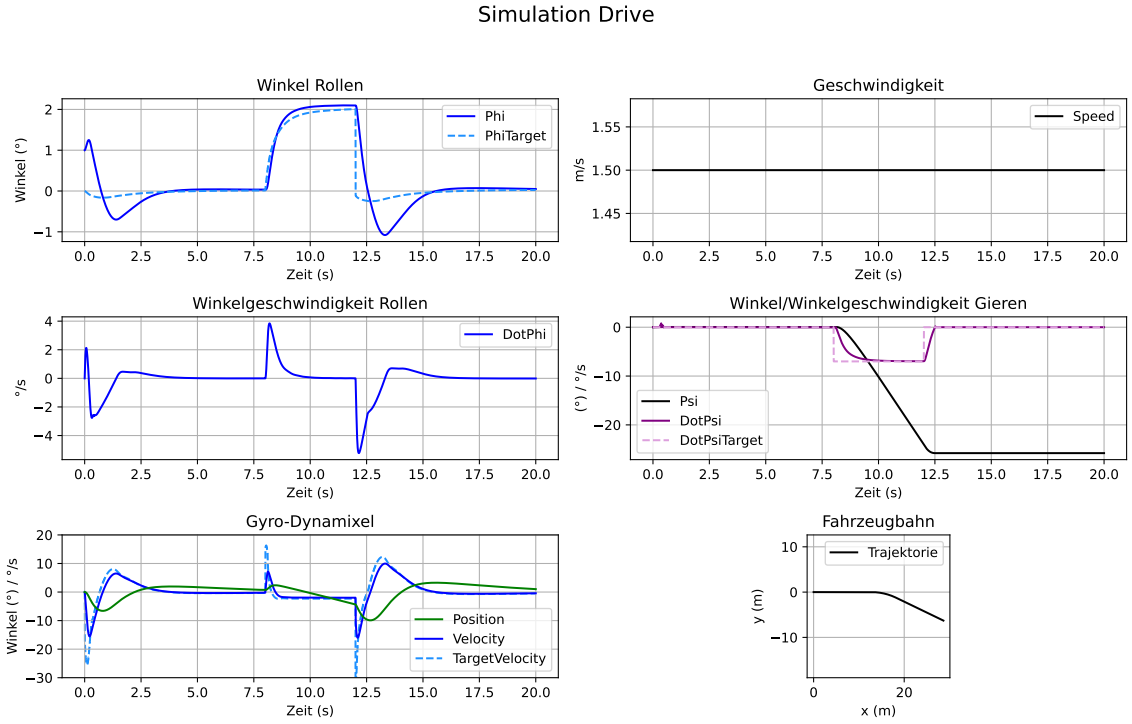
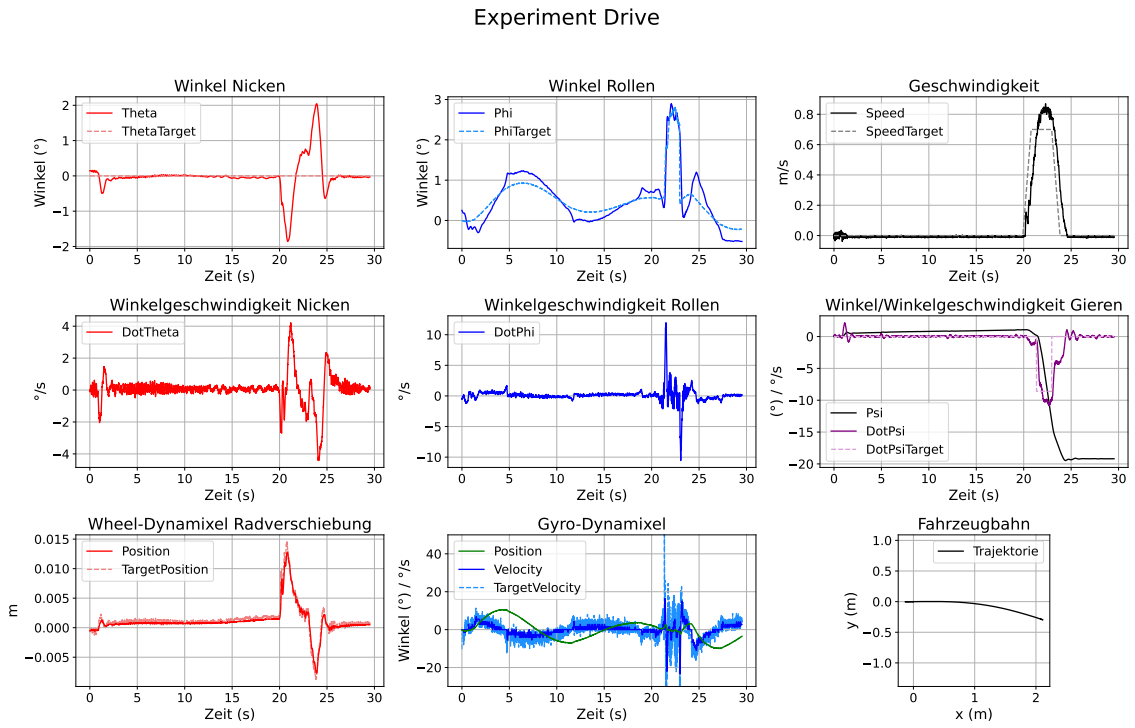
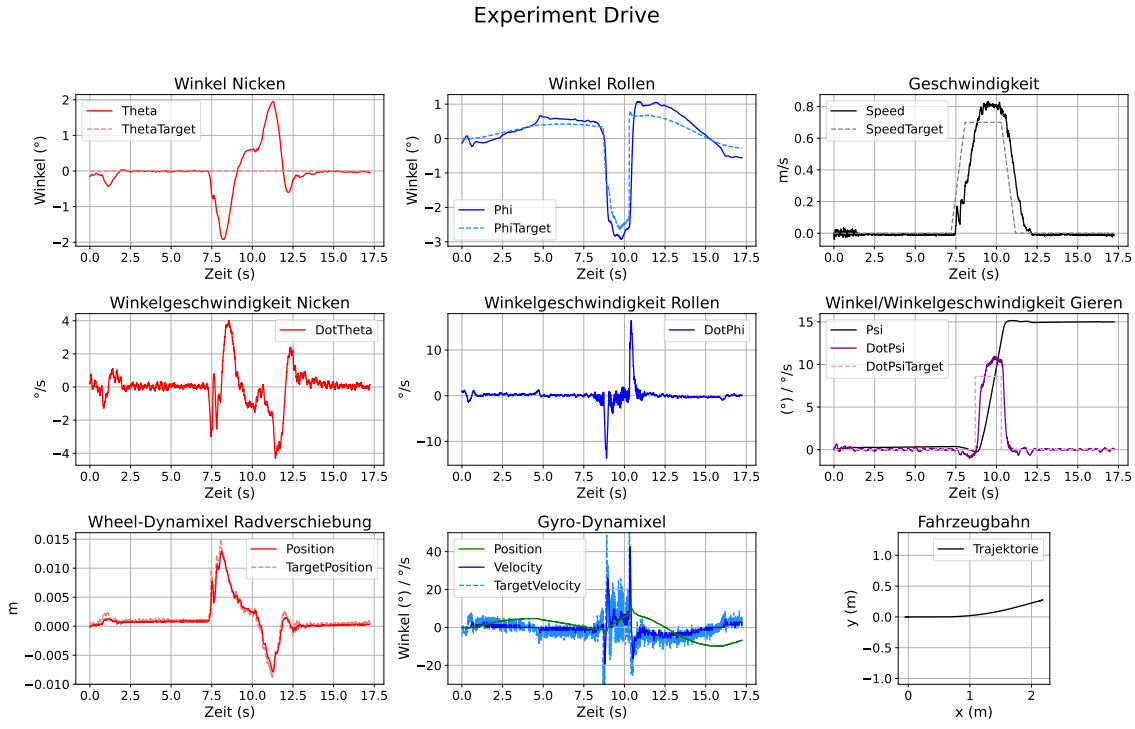


Figure 7.4: Simulation of vehicle cornering through active trajectory

At $t = 8\,\text{s}$, the system switches from the PID controller for balancing to the PID controller for cornering. The vehicle drives through a specific curve using the generated trajectory. It can then be raised. The system can thus successfully drive through a specific curve in the simulation, and the concept is also tested on the real system (see Figure 7.5a and Figure 7.5b).



(a) Right curve



(b) Left curve

Figure 7.5: System test vehicle cornering with controller-based roll trajectory

The *Monowheeler* can drive corners precisely using the concept presented. The controller parameters from the simulation are tested on the real system and improved experimentally until the results shown can be achieved with the parameters $KP = 1.2;\ KD = 0.2;\ KI = 1.5$. Both left and right turns are possible, and how tight or wide the turn is can be determined directly by specifying the yaw rate. A tighter curve can be seen in Figure 7.6.

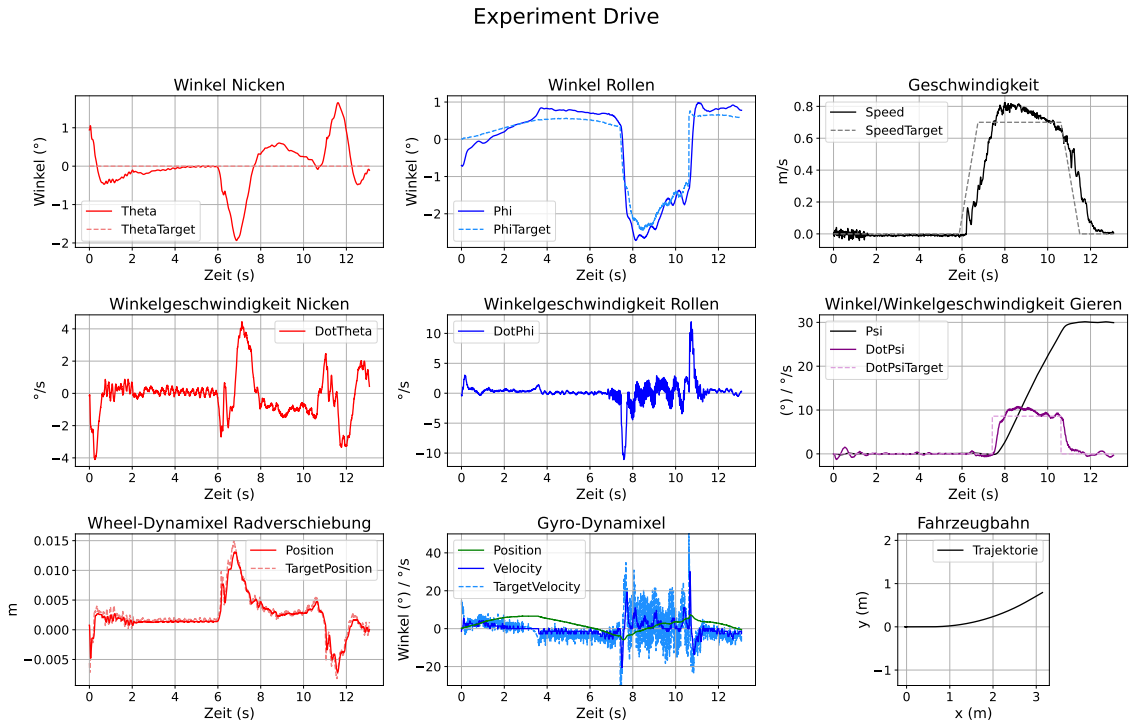


Figure 7.6: System test vehicle tight left curve with controller-based roll trajectory

Cornering is now so robust that even more complex maneuvers, such as an S-curve, can be performed (see Figure 7.7). This shows that cornering with the developed controller is targeted and reproducible, and that the system is robust against disturbances.

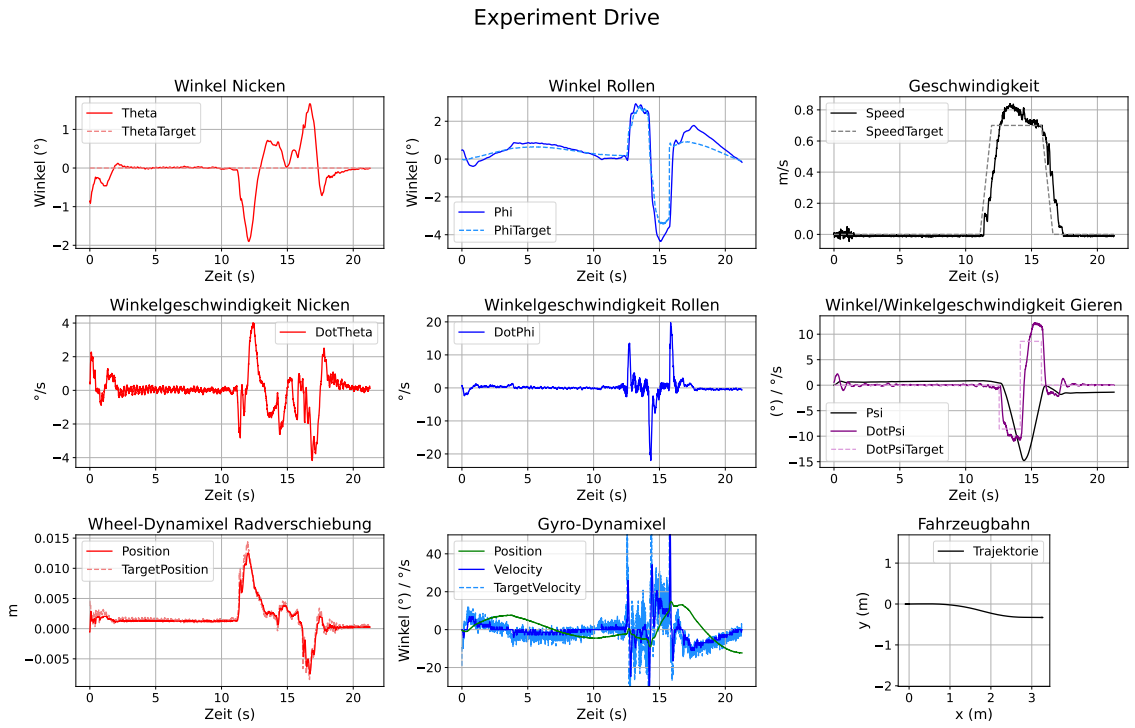


Figure 7.7: System test vehicle slalom with controller-based roll trajectory

Despite this successful implementation, there are also limitations. Although the PID controller can control cornering using the trajectory trick, it is not a true multi-variable controller and cannot include the angle of the gyroscope in the control. For the duration of the maneuver, it must therefore be assumed that the gyroscope has sufficient angle reserve. The trajectory is also not optimized with regard to the gyroscope angle, as would be possible with a more complex algorithm.

A more significant challenge also arises from the structural assumption of the control concept. Balancing requires that the roll and yaw movements be decoupled. This assumption applies at the operating point ($\varphi \approx 0,\ \dot{\psi} \approx 0$) and enables stable control there (see Section 6.3).

When stationary, this decoupling is very strong and the system is very robust. However, when driving, this decoupling is increasingly eliminated: The static friction of the wheel decreases significantly with increasing speed, and the unmodeled effects of the wheel (see Chapter 3) when cornering gain influence, causing roll and yaw dynamics to influence each other. At the same time, the vehicle leaves its operating point during a cornering maneuver. This makes it difficult for the system to stabilize again in the long term after major maneuvers. In some cases, a yaw movement remains even after the vehicle has been straightened up, so that the wheel's static friction has already been overcome. In addition, the gyroscope is strongly deflected when cornering. In order to bring the gyroscope back to the center, the PID controller specifies a roll movement for balancing purposes. However, this pronounced roll movement leads to further yaw movements while driving and further destabilizes the vehicle.

In practice, the system operates more robustly at lower speeds because the static friction of the wheel is greater and the dynamic coupling between rolling and yawing is lower. Therefore, the turns shown are performed at a speed of $v = 0.7\,\mathrm{m\,s^{-1}}$ instead of the $v = 1.5\,\mathrm{m\,s^{-1}}$ from the simulation. In faster or tighter curves, the vehicle may initially remain stable, but often fails to return to the original operating point for balancing. As long as the gyroscope still has sufficient angular reserves, the vehicle remains stable for a short time, but sooner or later it will fall over. A pragmatic solution to this problem is to brake or stop the vehicle after a maneuver. At low speeds or when stationary, the decoupling of rolling and yawing is strong enough to compensate for the larger deviations from the operating point due to the lack of cornering dynamics and significantly higher static friction of the wheel, and the vehicle can stabilize reliably and permanently. This behavior is not a random weakness, but a structural limitation of classic control concepts.

7.3 Results of cornering

In summary, it can be said that the developed control concept enables targeted and robust cornering. The coupling of the degrees of freedom of roll and yaw can be used in a targeted manner to reliably implement a desired yaw movement by means of a controller-based trajectory specification for the roll movement. The limitations of the approach result from the decoupling of roll and yaw dynamics assumed for balancing, which is no longer completely given after highly dynamic maneuvers. Within these assumptions, however, the system exhibits stable and reproducible driving behavior. This enables the *Monowheeler* to drive targeted curves. It should also be noted that, despite the simplifications made, the simulation fundamentally reflects the behavior of the system. This is because the dynamics of the gyroscope significantly determine cornering, and the unmodeled components are less important, especially at slower speeds.

8 Hardware integration and control loop on microcontroller

To implement the developed control structures on the actual vehicle, the control loop is implemented on a microcontroller. This handles the processing of sensor data, the implementation of the controllers, and the control of the actuators. An BeagleBone Black is used for this purpose, which is suitable for controlling many components (actuators and sensors) due to its extensive peripherals (SPI, UART, GPIOs, etc.). It is also the EML laboratory standard, providing a framework for controlling many hardware components.
The software for controlling the *Monowheelers* is based on the principle shown in Figure 8.1. The development computer is connected via WLAN to a TP-Link router, which is connected to the BBB. On the Linux development computer, the software is started on the BBB via an Secure Shell (SSH) connection. The application consists of three threads, which are described in detail in this chapter. After starting the communication thread, a Python GUI can be started on the development computer, which receives the current vehicle data via User Datagram Protocol (UDP)/IP, displays it in live plots, and saves it in an CSV file.

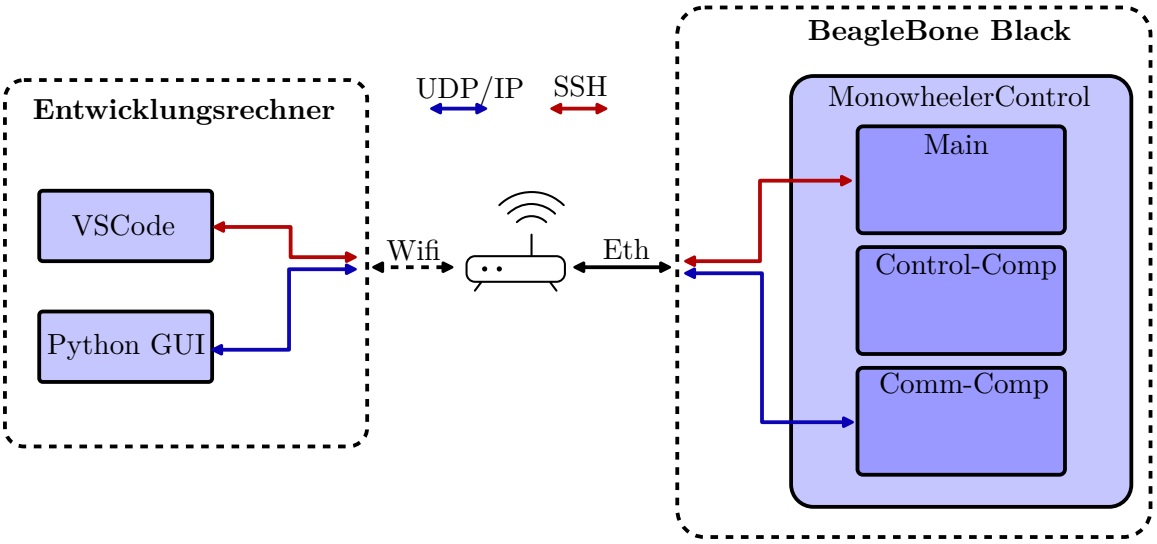


Figure 8.1: Software overview

The following sections describe the individual components of the system.

8.1 Development environment Development computer

The project is designed for a Linux operating system with VS Code as Integrated Development Environment (IDE), but can also be operated from the terminal. The build process is

managed with a make file. A cross-compiler for Linux Arm is used as the compiler. Care must be taken to ensure that the compiler does not perform any optimization (compiler flag -O0), as memory mapping and global variables can be incorrectly optimized away as containers for communication between threads. Asynchronous processes and timing can also be disrupted. In addition, the application must be statically linked, as the BBB does not have a complete and up-to-date dynamic runtime environment for libraries, meaning that all dependencies must be integrated directly into the binary.

To start the application on the BBB, the current binary is transferred via Secure Copy (SCP). To establish the connection, the development computer must be on the network of the TP-Link router connected to the BBB. There, the BBB with Multicast DNS (mDNS) is accessible under the host name „BeagleBone. local“Ṫhe application can then be started and managed in the terminal via SSH. The Remote GDB Debugger is used as the debugger. To do this, the GDB server is started on the BBB so that the GDB debugger of the development computer can connect to the BBB.

All of the tasks described are automated by the Bash script *manage_monowheeler.sh*. The functions are documented in the script's help text:

```
Usage: ./manage_monowheeler.sh [OPTION]

Options:
-h,  --help                    Show this help message
-g,  --gui                     Start the GUI on the host machine
-bd, --build-debug             Build the application in Debug mode for BeagleBone
  ↪  Black (BBB)
-br, --build-release           Build the application in Release mode for
  ↪  BeagleBone Black (BBB)
-r,  --run                     Build, copy, and run the Release version on BBB
-c,  --clean                   Clean the project (remove compiled files)
-gdb,--start-gdb               Start gdbserver on BBB with Debug build
-po, --poweroff                Power off the BeagleBone Black
-rb, --reboot                  Reboot the BeagleBone Black

```

Listing 8: Automatisierung der Entwicklungsumgebung

In addition, the most important functions (build, clean, run, debug, reboot, poweroff) are created as VS Code tasks. This means that these tasks can be executed at the touch of a button. The Python GUI is also automated as a VS Code task.

8.2 BeagleBone Black development environment

An official Debian IoT image is used on the BBB, which is originally based on the low-latency kernel *5.10.168-ti-r80*. In the course of the project, the real-time kernel *5.10.212-bone-rt-r78* will be used to enable more accurate timing.

In order to use the Internet without security risks (e.g., to install the new kernel), root logins via SSH are prohibited. To still be able to execute important commands (e.g., the Target application) with root privileges, the file */etc/sudoers. d/noPw* contains a list of commands that can be executed with root privileges without a password prompt. In addition, Uncomplicated Firewall (ufw) is enabled and blocks all connections. The ports used (SSH and debugger) must be enabled accordingly.

To manage the hardware peripherals, the kernel uses a so-called device tree. The device tree is a structured data description to define the hardware available on a platform and its configuration. Instead of storing hardware information permanently in the kernel code, the device tree allows a flexible and modular description so that the kernel can correctly initialize and use the available peripheral modules such as GPIO, UART, I^2C, or SPI. *U-Boot overlays* are used for this purpose on the BBB. These device tree overlays make it possible to dynamically extend or change hardware configurations without having to recompile the kernel. Integration is done via the configuration file */boot/uEnv.txt*, in which the corresponding overlays are loaded. This allows the required interfaces to be activated and configured. It is also possible to create your own overlays, e.g., to change the pin multiplexing configuration. This allows pins to be flexibly assigned to different functions, although conflicts with already assigned interfaces (such as HDMI or audio) must be taken into account.

8.3 Target application

The architecture and flow of the target application is shown in Figure 8.2. It consists of three components, each running in its own thread. The main component manages the other two components and processes user input. In addition, the other two components are created and started. The control component handles the implementation of the control loops and the control of the hardware, while the communication component sends vehicle data to the Python GUI.

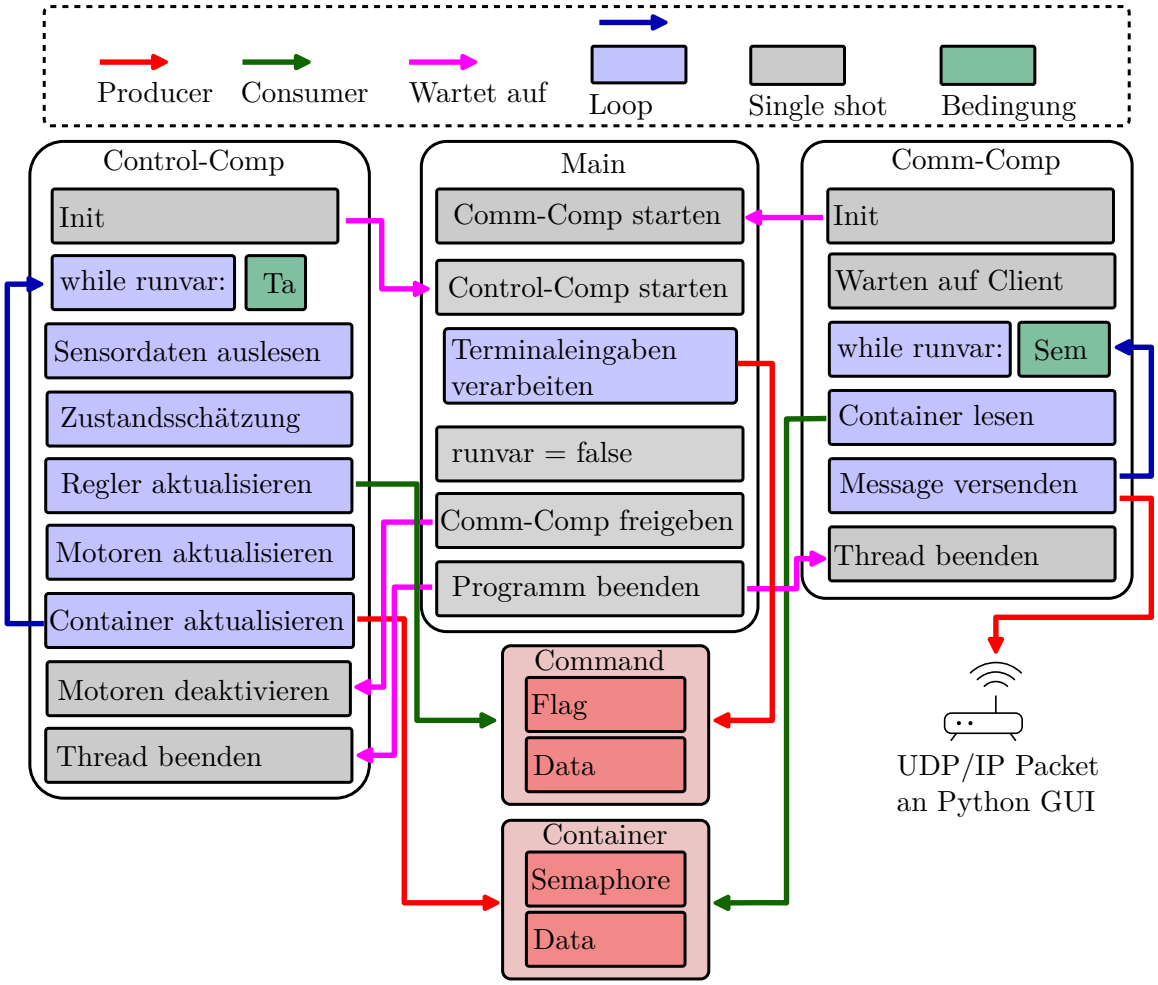


Figure 8.2: Architecture of the target application

After the two components have been started, the Main processes inputs in the terminal. The vehicle can receive various commands via a simple command line input, which are interpreted by the class *CCommand*. The text entered is first broken down into individual tokens and the first token is evaluated as a command. The following commands are supported:

- `drive <value>` – accelerates the vehicle to the target speed `<value>`
- `left` – initiates a left turn
- `right` – initiates a right turn
- `slalom` – starts a slalom maneuver

The command is stored in a global object. The control computer then accesses the command and initiates its implementation. Synchronization between the producer (main) and consumer (control computer) is achieved using an atomic flag that signals whether new data is available in the global object. The producer only writes new commands to it if the container is empty. Otherwise, the input is rejected and a corresponding error message is issued. In each cycle, the consumer checks whether new data is available, processes it, and then resets the flag. A complex handshake protocol is not necessary in this scenario, as

commands are only entered by the user at relatively long intervals and multiple entries are handled by the error message. This ensures that commands are neither processed multiple times nor simultaneously, without compromising the real-time capability of the consumer. The command to terminate the program immediately is handled separately and takes effect immediately.

The vehicle data is transferred from the control computer, which acts as the producer, to the communication computer, which acts as the consumer, via a global container. To make the data transfer thread-safe and avoid race conditions, the container contains a synchronization mechanism in addition to the data. Semaphores are used because the data may only be read once. Normally, a cross-handshake is necessary in such a context. The control computer signals the presence of new data via semaphore, and the communication computer reports when processing is complete. This ensures that data is not processed multiple times, simultaneously, or partially.

However, in this case, the consumer is guaranteed to be faster than the producer. In addition, the control component must not be blocked under any circumstances, as this would jeopardize the stability of the vehicle. For these reasons, a simple semaphore that signals to the communication component when new data is available is sufficient.

If the input to stop the program is made, the other components are signaled to terminate with the variable *runvar*. The control component terminates after the next sampling step. To ensure that the communication component is not blocked on the semaphore, the main releases the semaphore again. Once both the control component and the communication component have terminated, the program is terminated.

8.3.1 Software architecture

The control of the *Monowheelers* and the data exchange with the Python GUI are divided between two components. A simplified structure of the classes is shown in Figure 8.3.

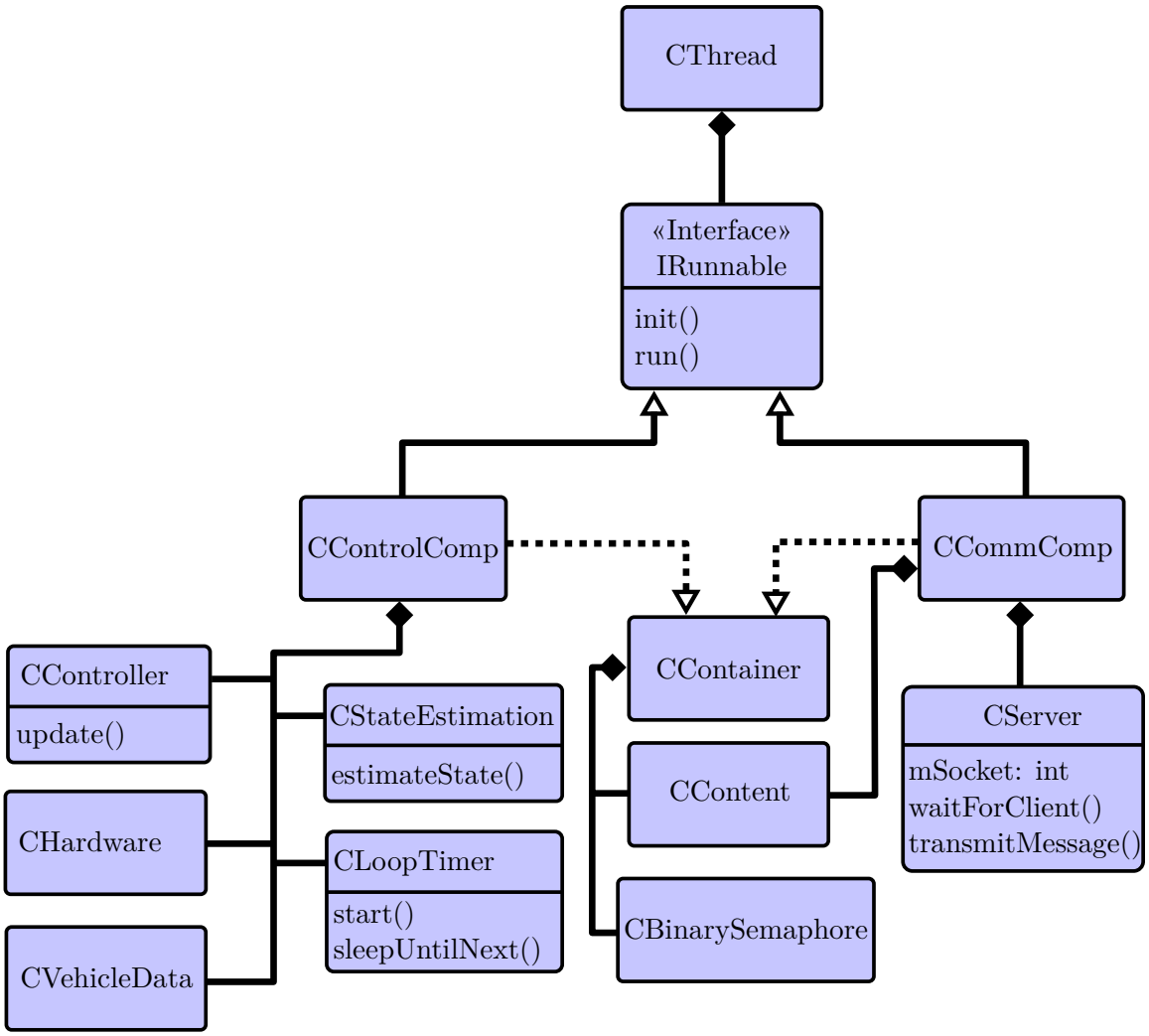


Figure 8.3: Class diagram of the target application (simplified)

Both components inherit from the virtual class *IRunnable* and thus offer a uniform interface with an *init()* and an *run()* function. The thread of the respective component is managed by the class *CThread*, so that, for example, different priorities or the scheduling method can be specified. The two classes *IRunnable* and *CThread* for using and managing Posix threads on the BBB are part of the framework from EML and are structured according to the principles described in [41] and [40].
The Control-Comp handles the timing, control loop, and hardware control, while the Comm-Comp sends the vehicle data via UDP. This division is chosen in order to meet the real-time requirements of the system as accurately as possible. Control loops place strict real-time demands on the microcontroller, as data acquisition, calculation, and control of the actuators must take place at defined time intervals. If delays or variable latencies (jitter) occur, the effective sampling interval changes, which jeopardizes the stability of the controller, amplifies vibrations, or can lead to loss of control of the vehicle. However, Linux is not real-time capable because the scheduling and processing of interrupts does not follow deterministic timing. Therefore, every context switch introduces additional, non-deterministic latency. Context switches should therefore be avoided as much as possible. For this reason, all

tasks that are part of the control loop are executed sequentially in a thread (in the control computer). This means that a context switch only occurs when the control computer enters sleep mode until the next sampling interval.

The real-time capability of the Linux system can be improved by choosing the right kernel. By default, the Debian IOT image for the BBB comes with a low-latency kernel that is already optimized for low latency. Even better are real-time kernels, such as the *5.10.212-bone-rt-r78*, which is used for this project. The differences in timing between the two kernel variants are examined in more detail in Subsection 8.3.2.
There are numerous parameters throughout the software that can be adjusted, from controller parameters and hardware parameterization to filter coefficients. To create a uniform, clear interface, a namespace *Monowheeler* is created in the file *MonowheelerConstants.hpp*. All parameters that are not fixed (e.g., memory addresses of certain registers) are stored in this namespace and can be integrated at the appropriate locations.

Presenting all classes and functions in detail would go beyond the scope of this work. Therefore, the most important modules are described in simplified form in the following sections. A complete and detailed description can be found directly in the source code and in the accompanying *README* files of the repository.

8.3.2 Control-Comp

The Control-Comp is responsible for timing, reading the sensors, signal processing, control, and actuation. All of these tasks are outsourced to dedicated classes (see Figure 8.3) and merged in the Control-Comp. The following sections describe the implementation of the individual modules.

Timer

The Timer class is designed to offer three functions: to start the timer, to read the current time, and to wait until the next sampling step. On classic microcontrollers without an operating system, hardware timers can be used for time measurement. This enables hard real-time operation. On a Linux system such as the BBB, however, hardware interrupts are managed by the Linux kernel. Therefore, Posix functions can be used in user space to access the hardware timers.

The function *clock_gettime()* returns the current time accurate to the microsecond. The timer can be selected independently. The *CLOCK_MONOTONIC* clock is suitable for this application, as it is a monotonically increasing timer that counts the time since the system started without being influenced by time jumps. This allows the current system time to be recorded and stored when the timer starts. The function for reading the current time then only has to subtract the time at the start of the timer from the current system time.

The function *clock_nanaosleep()* is used for the function to wait in sleep mode until the next sampling step. This function can either sleep for a relative time or until a specific absolute time. For the relative time, the time at the beginning and end of the loop cycle is

stored. The difference gives the runtime, so that the remaining time until the next sampling step can be calculated and slept. For absolute time, the time at the start of the timer is stored. The absolute time of the first sampling step can be calculated by adding the sampling time to the start time. The function *clock_nanosleep()* then sleeps until this time is reached. All other wake-up times can be calculated by adding the sampling time to the previous timestamp. Both of these options should be tested.

For the jitter test, a sampling time of 2 ms is selected, which corresponds to a frequency of 500 Hz. This frequency is higher than the sampling frequency expected to be required in the later application scenario. This subjects the system to a more stringent requirement, and it can be assumed that the system will achieve comparable or better results at lower utilization. The tests thus serve as a worst-case scenario. Over a period of approximately 10 s, the absolute timestamps of each sampling step are recorded and then analyzed using the Python script *jitter.py*. First, the relative time measurement is tested with the low-latency kernel (see Figure 8.4):

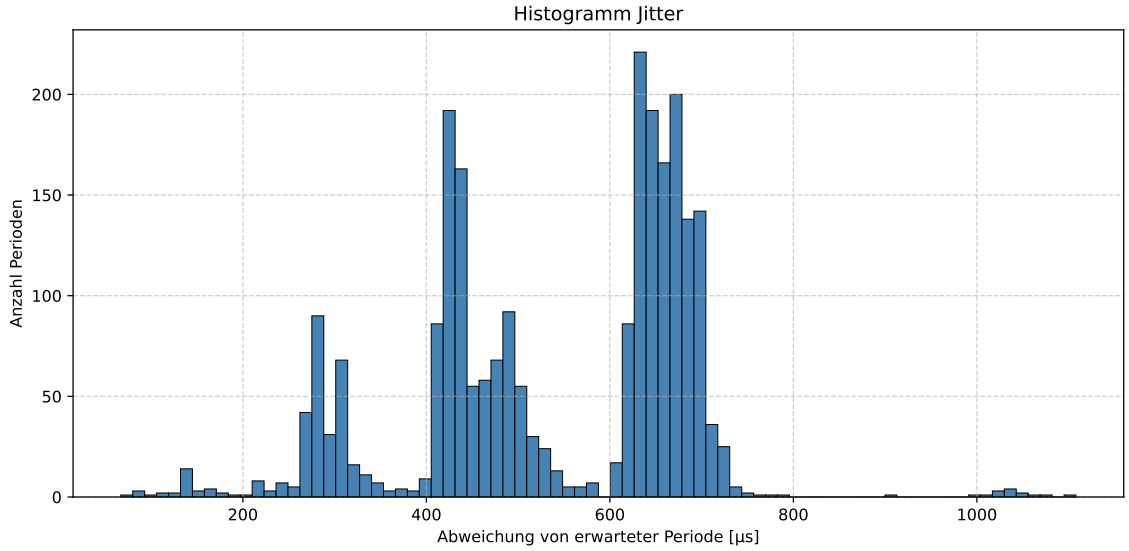


Figure 8.4: Histogram jitter test low-latency kernel with relative time measurement

The average jitter is 540 µs, and the maximum jitter is 1108 µs. This corresponds to an average error of 25 % and a maximum error of over 50 %. This jitter is not acceptable for a control loop. Therefore, we switch to the real-time kernel already described and perform the same test with the same software (see Figure 8.5):

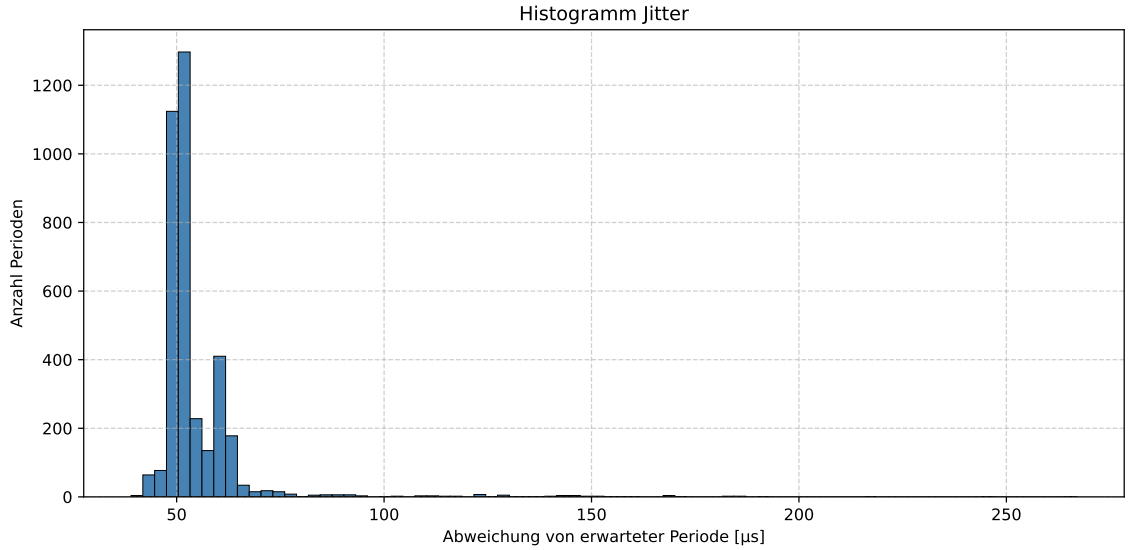


Figure 8.5: Histogram Jitter Test Real-time kernel with relative time measurement

The results have improved significantly and the jitter has decreased considerably. The average jitter has decreased by a factor of 10 to 55 µs, and the maximum jitter is now 267 µs. However, the histogram shows that the frequency of outliers has decreased significantly. It can also be seen, however, that the jitter is not just a random error, but also contains an offset, as all values are around 50 µs. This can be explained by the fact that when measuring the loop cycle time, the time that arises during scheduling when the thread is put into sleep mode and then woken up again is not taken into account. Therefore, absolute time measurement is tested (see Figure 8.6). The time is recorded once per loop cycle, thus taking into account any delay in scheduling.

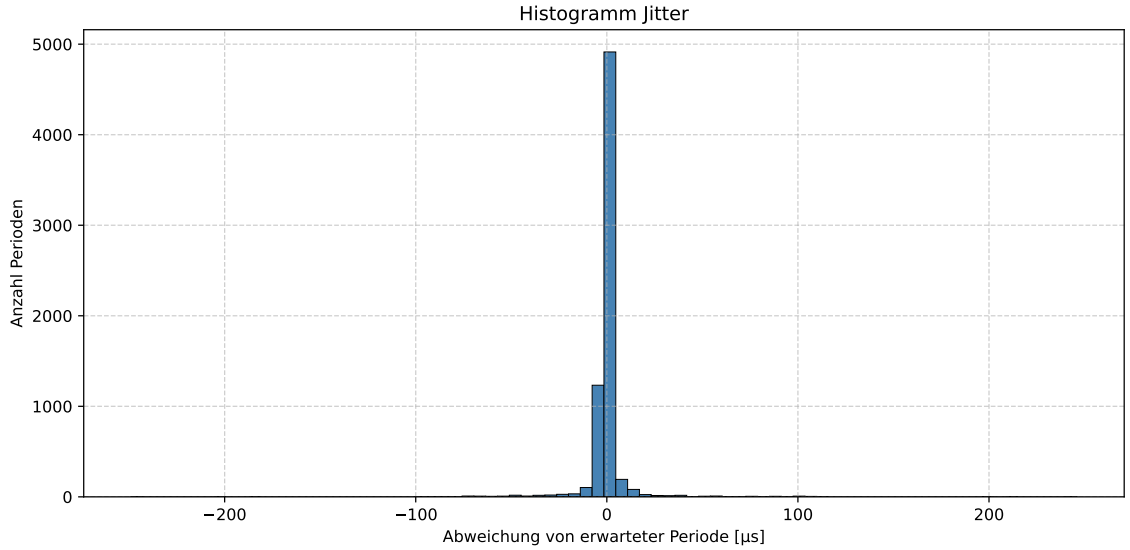


Figure 8.6: Histogram Jitter Test Realtime kernel with absolute time measurement

As expected, the jitter now approximates a normal distribution around zero. The mean jitter has been reduced by a factor of 10 and is now 4 µs. Outliers with a maximum value of 249 µs occur so rarely that they are negligible. A jitter in the single-digit microsecond range is completely unproblematic for a control loop of such an inertial system.

The implementation of the function for starting the timer and waiting in sleep mode until the next sampling step can be seen in Listing 9:

```
1  void CLoopTimer::start()
2  {
3      clock_gettime(CLOCK_MONOTONIC, &mStartTime);
4      mWakeTime = mStartTime;
5  }
6  void CLoopTimer::sleepUntilNext()
7  {
8      mWakeTime.tv_nsec += mTaNS;
9      if (mWakeTime.tv_nsec >= 1'000'000'000L) {
10         mWakeTime.tv_sec += mWakeTime.tv_nsec / 1'000'000'000L;
11         mWakeTime.tv_nsec %= 1'000'000'000L;
12     }
13     struct timespec now;
14     clock_gettime(CLOCK_MONOTONIC, &now);
15     int64_t overrunUs = (now.tv_sec - mWakeTime.tv_sec) * 1'000'000L +
16                         (now.tv_nsec - mWakeTime.tv_nsec) / 1'000L;
17     if (overrunUs > 0) {
18         REPORT_ERROR("Overrun: ", overrunUs, " µs late");
19     }
20     clock_nanosleep(CLOCK_MONOTONIC, TIMER_ABSTIME, &mWakeTime, nullptr);
21 }
```

Listing 9: Ausschnitt Implementierung Timer

Signal processing

The task of the signal processing module is to implement the filters developed in Chapter 5 on the microcontroller. This task is performed by the class *CStateEstimation*. An overview of the structure of the class can be found in Figure 8.7.

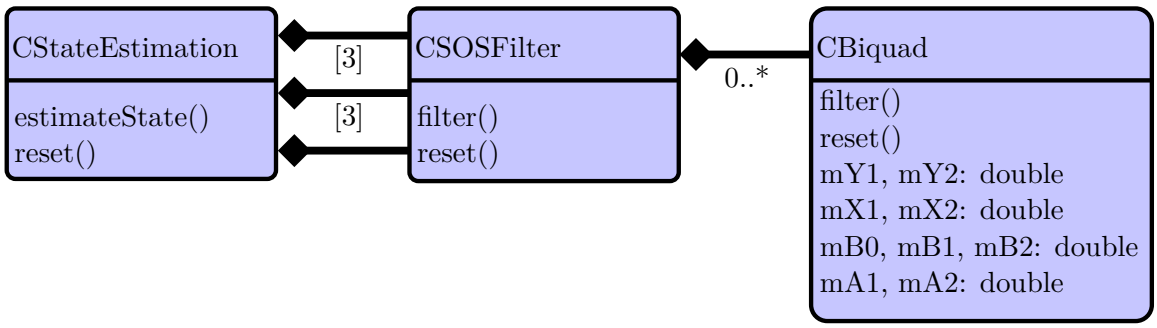


Figure 8.7: Class diagram of *CStateEstimation* (simplified)

At each sampling step, the function *estimateState()* is called and the current sensor data is transferred. The sensor data is first filtered. For this purpose, the class *CStateEstimation* has two arrays, each with three objects, and another single object. The two arrays each contain the low-pass filters for the accelerometers and gyroscopes for all axes. The additional object represents a low-pass filter for the speed of the vehicle. The sensor data is then merged according to Chapter 5 to enable the most accurate state estimation possible. The implementation of the function can be seen in Listing 10:

```
void CStateEstimation::estimateState(const CImuData& pImuData, CStateData
↪ &pStateData)
{
    pStateData.mDotTheta = mDLPGyro[1].filter(pImuData.mGyroY);
    pStateData.mDotPhi = mDLPGyro[0].filter(pImuData.mGyroX);
    pStateData.mDotPsi = mDLPGyro[2].filter(pImuData.mGyroZ);
    pStateData.mSpeed = mDLPSpeed.filter(pStateData.mSpeed);

    double xAccelFil = mDLPAccel[0].filter(pImuData.mAccelX);
    double yAccelFil = mDLPAccel[1].filter(pImuData.mAccelY);

    pStateData.mTheta = mAlpha*(pStateData.mTheta + pStateData.mDotTheta*mTa) +
    ↪ (1-mAlpha)*(std::asin(xAccelFil / g));
    pStateData.mPhi = mAlpha*(pStateData.mPhi + pStateData.mDotPhi*mTa) +
    ↪ (1-mAlpha)*(-std::asin(yAccelFil / g));
}
```

Listing 10: Ausschnitt Implementierung Zustandsschätzung

The filters are implemented in the CSOSFilter class. An Second-Order-Section (SOS) filter consists of a sequence of second-order filters to ensure numerical stability for higher-order Infinite Impulse Response (IIR) filters. The individual second-order filter blocks are implemented in the *CBiquad* class. For this purpose, a discrete second-order IIR filter is implemented as shown in Listing 11, whose parameters are specified in the constructor.

```
double CBiquad::filter(double pX)
{
    double y = mB0*pX + mB1*mX1 + mB2*mX2 - (mA1*mY1 + mA2*mY2);
    mY2 = mY1;
    mY1 = y;
    mX2 = mX1;
    mX1 = pX;
    return y;
}
```

Listing 11: Ausschnitt Implementierung IIR-Filter zweiter Ordnung

Any number of such filter blocks are stored in a dynamic array in the *CSOSFilter* class, so that any type of filter (notch, low-pass, high-pass, etc.) can be implemented in any order. To do this, only the filter coefficients need to be selected accordingly. The filter modules are then called up one after the other as shown in Listing 12, resulting in any IIR filter.

```cpp
CSOSFilter::CSOSFilter(std::vector<std::array<double, 6>> pCoeffs)
{
    for (const auto coeff : pCoeffs) {
        mBiquads.emplace_back(coeff);
    }
}
double CSOSFilter::filter(double pX)
{
    double y = pX;
    for (CBiquad& biquad : mBiquads) {
        y = biquad.filter(y);
    }
    return y;
}
```

Listing 12: Ausschnitt Implementierung IIR-Filter als SOS-Filter

Control

The control concept for the *Monowheeler* involves direct control of the vehicle's pitch, roll, and speed and is to be implemented in class *CController* class on the microcontroller. In addition, the movement of the entire vehicle can be specified by the higher-level specification of trajectories, including the control of the yaw movement. The vehicle can stand still, drive in a straight line, and drive around curves. The controllers are coordinated centrally in class *CController*, while the individual controllers are outsourced. Class *CController* provides an *update()* function that contains the entire control loop. The class structure of the implementation can be seen in Figure 8.8, and the complete implementation of the *update()* function can be found in Section C.3.

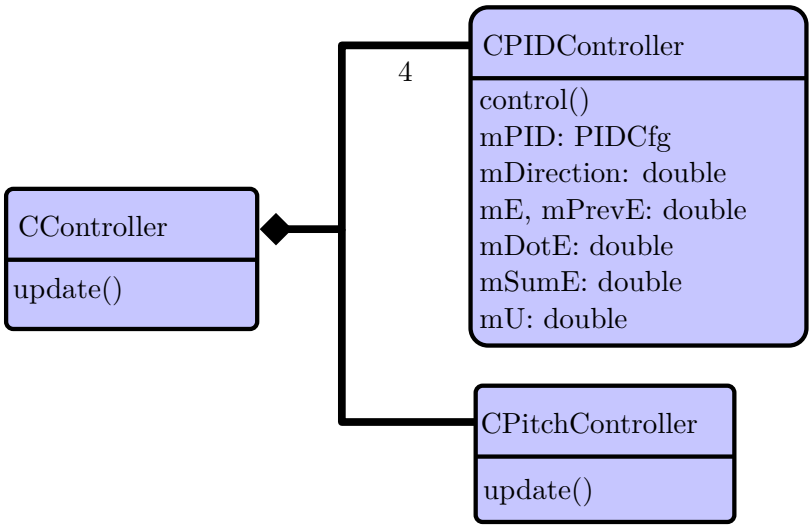


Figure 8.8: Class diagram of *CController* (simplified)

The concept for controlling speed, roll, and yaw is based on PID controllers. The PID

controllers are implemented in the class *CPIDController*. The class contains internal variables for storing the parameters and various control deviations and provides an overloaded function *control()*. This function updates the internal variables and returns the control variable. The function takes either the classic setpoint and actual value as arguments, or, in addition, a setpoint and actual value for the derivation of the variable to be controlled. In the case of the *Monowheelers*, the angular velocity is available directly as a sensor value. It therefore makes sense to use this value directly in the PID controller instead of first determining the angle, then calculating and deriving the control deviation (for more information, see Equation 6.39 ff). The implementation of the function *control()* with additional setpoint and actual value of the derivative can be seen in Listing 13. In addition, anti-windup is introduced for the integrator and the control variable is limited.

```
1  double CPIDController::control(const double pW, const double pDotW, const double
   ↪  pX, const double pDotX)
2  {
3      mE = pW - pX;
4      mDotE = pDotW - pDotX;
5      if (std::abs(mU) < mPID.UMax) {
6          mSumE += mE*mPID.Ta;
7      }
8      mU = std::clamp(mDirection*(mPID.KP*mE + mPID.KD*mDotE + mPID.KI*mSumE),
       ↪  -mPID.UMax, mPID.UMax);
9      mEPrev = mE;
10     return mU;
11 }
```

Listing 13: Ausschnitt Implementierung PID-Regler

The entire section for the implementation of the PID controller with both versions of the *control()* function can be seen in Section C.2.

The *update()* function of the *CController* class can be divided into two sections. On the one hand, there are the controllers that have the motor control variables as their output and implement specifications for the physical system. These controllers are called at the end of each run. These include the state controller for the pitch movement, the PID controller for the roll movement, and the PID controller for the speed of the vehicle. In addition, there are the specifications for trajectories and setpoints, which are responsible for the higher-level motion planning. These include the trajectories of the speed and the roll movement. Depending on the current mode, different concepts are used for setpoint generation. First, the controllers for regulating the physical system are presented.

The controller for pitch movement designed in Section 6.2 is to be implemented in the class *CPitchController*. The class should provide a function for updating the internal variables (controller output, etc.) that returns the current manipulated variable. A normal state controller can be implemented by simply multiplying the controller matrix by the states (see Listing 14):

```
mUCMD = -std::clamp((mLQRFast[0]*pTheta
                    + mLQRFast[1]*pDotTheta
                    + mLQRFast[2]*mThetaSum
                    + mLQRFast[3]*pDynamixel), -mUCMDMax, mUCMDMax);
```

Listing 14: Ausschnitt Implementierung des Zustandsreglers Nicken

However, this is a controller with state-based gain scheduling. Therefore, a simplified state machine is also implemented. There are four modes:

- Mode Fast: The fast parameter set is used to calculate the manipulated variable. Used when there are certain deviations in angle or angular velocity.
- Mode Slow: The slow parameter set is used to calculate the control variable. Used when the system is stable for a sufficiently long time.
- Mode TransitionSlow2Fast: Transition between mode Slow and mode Fast. The outputs of the two parameter sets are linearly interpolated to avoid discontinuities in the control variable. Active for a specified period of time after the change.
- Mode TransitionFast2Slow: Transition between Mode Fast and Mode Slow, identical function to Mode TransitionSlow2Fast.

At the start of the *update ()* function, various conditions are checked depending on the mode and internal states are adjusted if necessary. An example can be seen in Listing 15.

```
case Mode::Fast:
if (std::abs(pTheta) < mThetaLowerThreshold && std::abs(pDotTheta) <
  ↪  mDotThetaLowerThreshold) {
    mTSlow += mTa;
    if (mTSlow > mTSlowThreshold) {
        mMode = Mode::TransitionFast2Slow;
        mTTransition = 0;
        mTSlow = 0;
    }
}
break;
```

Listing 15: Ausschnitt Implementierung der Zustandsübergänge Nicken

The control variable can then be calculated based on the mode. In Fast and Slow mode, the control variable is determined as in Listing 14; in the transition modes, the calculation is performed as shown in Listing 16:

```
case Mode::TransitionSlow2Fast:
    uCMDFast = -std::clamp((mLQRFast[0]*pTheta
                    + mLQRFast[1]*pDotTheta
                    + mLQRFast[2]*mThetaSum
                    + mLQRFast[3]*pDynamixel), -mUCMDMax, mUCMDMax);
    uCMDSlow = -std::clamp((mLQRSlow[0]*pTheta
                    + mLQRSlow[1]*pDotTheta
                    + mLQRSlow[2]*mThetaSum
                    + mLQRSlow[3]*pDynamixel), -mUCMDMax, mUCMDMax);
    blend = std::clamp(mTTransition/mTTransitionMax, 0.0, 1.0);
    mUCMD = blend*uCMDFast + (1-blend)*uCMDSlow;
    mTTransition += mTa;
    break;
```

Listing 16: Ausschnitt Implementierung der Interpolation Nicken

The complete function *update()* can be found in Section C.1.

The speed and roll movement are controlled using the *CPIDController* class already presented. The speed controller is stopped at a standstill to prevent the integrator from building up slowly due to small offsets in the measurement data. The controller call can be seen in Listing 17.

```
pMotorData.mWheelDynamixelTarget = mPitchController.control(pStateData.mTheta,
↪  pStateData.mDotTheta, pMotorData.mWheelDynamixelPosition);

pMotorData.mGyroDynamixelTarget = mRollController.control(pStateData.mPhiTarget,
↪  dotPhiTarget, pStateData.mPhi, pStateData.mDotPhi);
if (pStateData.mSpeedTarget == 0 && abs(pStateData.mSpeed) < 0.05) {
    pMotorData.mWheelTorqueTarget = 0;
}
else {
    pMotorData.mWheelTorqueTarget =
    ↪  mSpeedController.control(pStateData.mSpeedTarget, pStateData.mSpeed);
}
```

Listing 17: Ausschnitt Implementierung der Regler der physikalischen Größen in der Funktion *update()* aus *CController*

Unlike the trajectory for the rolling motion, the trajectory for the speed is not controller-based. To prevent large jumps in the motor torque of the drive wheel and to ensure smooth starting, the target speed is specified with a ramp. The setpoint is slowly brought to the target value using a maximum rate of change. The calculation is performed in the generic function *ramp()* (see Listing 18) . During balancing, the speed can be entered by the user; when cornering, a fixed speed is specified.

```
double CController::ramp(double pTarget, double pCurrentTarget, double pMaxRate)
{
    double deltaMax = pMaxRate * mTa;
    double diff = pTarget - pCurrentTarget;

    if (diff > deltaMax)
    {
        pCurrentTarget += deltaMax;
    }
    else if (diff < -deltaMax)
    {
        pCurrentTarget -= deltaMax;
    }
    else
    {
        pCurrentTarget = pTarget;
    }

    return pCurrentTarget;
}
```

Listing 18: Ausschnitt Implementierung der Rampe für sanftes Anfahren

The concept for controlling rolling for balancing and cornering involves two PID controllers (see Section 6.3 and Chapter 7). The PID controller for balancing specifies a setpoint for the roll angle depending on the position of the gyroscope. Together with the old setpoint, the desired roll speed can be determined. During cornering, the setpoint for the roll speed is specified by the PID controller for controlling the yaw speed. Together with the previous value for the roll angle, the new setpoint for the roll angle can be calculated by integrating the roll speed. Switching between the different modes is done based on internal states (reaching certain setpoints, elapsed time, etc.) and user input. The implementation of trajectory generation is shown in Listing 19.

```
switch (mMode)
{
case EMode::Balance:
    phiTargetNew = mRollTargetBalanceController.control(0, 0,
    ↪  pMotorData.mGyroDynamixelPosition, pMotorData.mGyroDynamixelVelocity);
    dotPhiTarget = (phiTargetNew - pStateData.mPhiTarget)/mTa;

    pStateData.mPhiTarget = phiTargetNew;
    pStateData.mSpeedTarget = ramp(mTargetVelocity, pStateData.mSpeedTarget,
    ↪  Monowheeler::MAX_ACCEL);
    pStateData.mDotPsiTarget = mDotPsiTarget;
    pStateData.mThetaTarget = 0;
    ...
case EMode::Corner:
    pStateData.mDotPsiTarget = mDotPsiTarget;
    dotPhiTarget = mRollTargeYawController.control(pStateData.mDotPsiTarget,
    ↪  pStateData.mDotPsi);
    pStateData.mPhiTarget += dotPhiTarget*mTa;
    pStateData.mSpeedTarget = ramp(mTargetVelocity, pStateData.mSpeedTarget,
    ↪  Monowheeler::MAX_ACCEL);
    pStateData.mThetaTarget = 0;
    ...
default:
    break;
}
```

Listing 19: Ausschnitt Implementierung der Trajektoriengenerierung in der Funktion *update()* aus *CController*

Hardware

The task of the hardware module is to provide a uniform interface for controlling all sensors and actuators. This is implemented at various levels of abstraction:

- Level of abstraction 1: The class *CHardware* represents the highest level of abstraction. It serves as the central interface for managing the hardware and acts as an adapter by performing project-specific conversions, restrictions, and calibrations and translating the desired physical values into the form expected by the generic motor and sensor classes. The rest of the software interacts exclusively with this level.
- Abstraction level 2: This level consists of classes such as *CImu* or *CSpin66Motor*, which provide uniform interfaces for individual device types. The devices are configured there on a project-specific basis and the functions required in the specific project are made available.
- Abstraction level 3: This level contains the generic driver classes such as *CICM20948* or *CDynamixelUARTMMAP*. They implement the basic logic for specific device types such as special sensors or motor controllers. These classes are reusable independently of the project and work with device-specific units and raw data.

- Abstraction level 4: The classes provide direct access to the peripherals of the BBB via Memory Map (MMAP). They implement the physical level of communication. The advantage of using MMAP accesses is the speed of access, as the Linux drivers and the associated slow context switches can be bypassed.

The structure of the classes is shown in Figure 8.9.

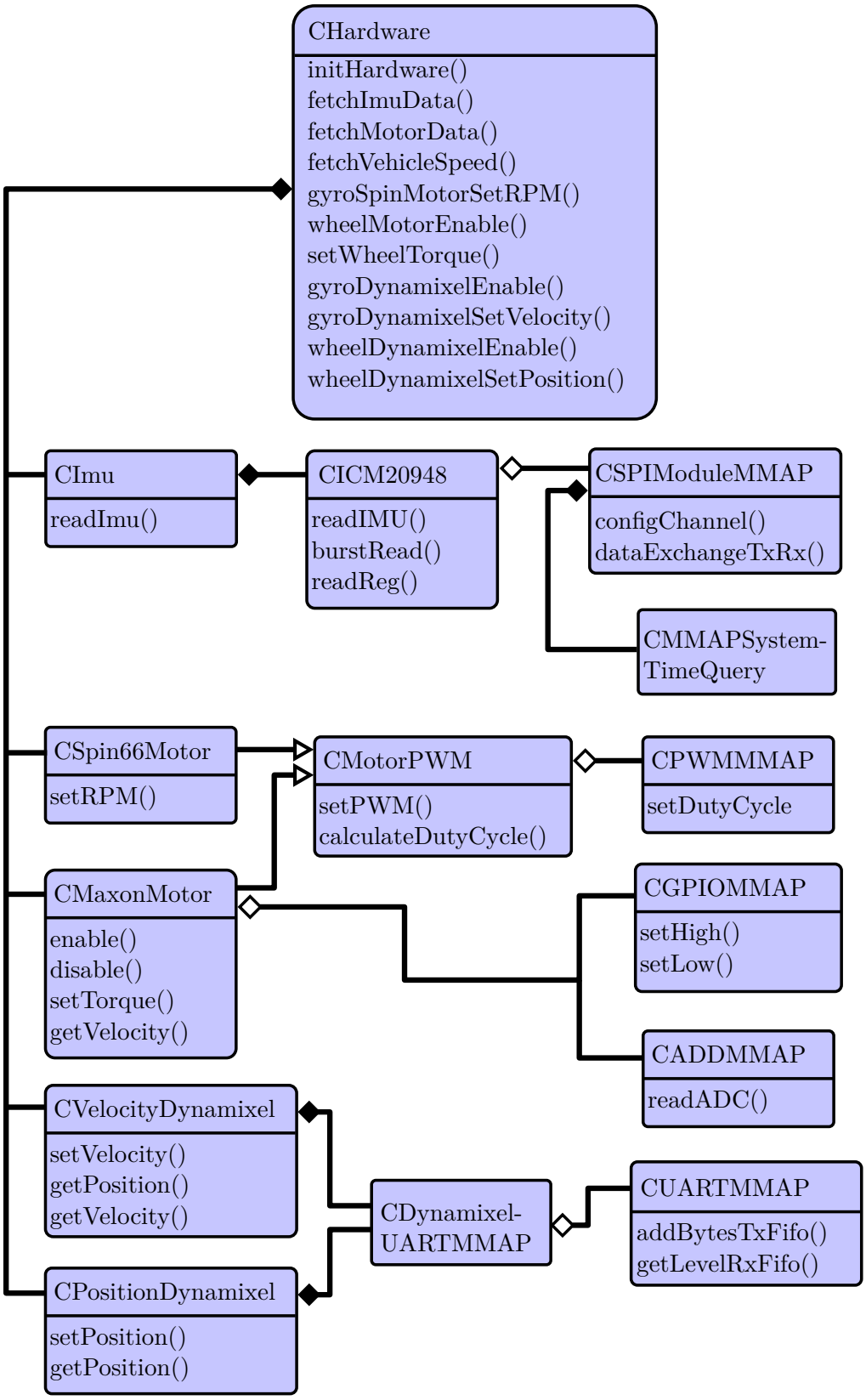


Figure 8.9: Class diagram of *CHardware* (simplified)

The classes of the lowest abstraction level access the hardware modules directly (e.g., UART, SPI, etc.) of the BBB. These are so-called shared resources, as all instances in the software access the same hardware modules. If several instances access the same hardware, undefined behavior may occur, as register states are overwritten in an uncoordinated manner. This leads to inconsistent peripheral states, data loss, and data corruption. Damage to the hardware can occur, especially when controlling the actuators. Therefore, a mechanism should be implemented to prevent multiple instances from accessing the same hardware modules in an uncoordinated manner.
Design patterns, i.e., proven solution patterns for recurring software problems, should be used for this purpose. A simple singleton is not sufficient here, as there are often several hardware modules of the same type (e.g., UART1... 4). Therefore, a multiton behavior must be implemented so that one instance can be created per key. The factory pattern is also used to encapsulate the creation and management of these instances and to minimize code repetition. A factory creates and manages all objects in a central location. For this purpose, the class *CInterfaceManager* is implemented, which combines the factory pattern with the multiton pattern.
All classes at the lowest level of abstraction have only private constructors, but declare the class *CInterfaceManager* as *friend class*. This ensures that instances of these classes can only be created by the function *getInstance()* from *CInterfaceManager*. The interface manager is implemented as a template consisting of a key and a hardware module type. For each registered data type, a map is created that stores pointers to all created instances and the associated key. In addition, the map stores information on whether an instance can be accessed multiple times. For example, it is possible to control multiple Dynamixel motors with one UART, and many hardware modules have multiple channels. However, it is the responsibility of the interface classes to check whether the accesses are compatible (same configuration of the hardware module) and to manage multiple channels themselves.

If the *getInstance()* method is now called, the map checks whether an instance has already been created for this key. If this is not the case, a new instance is created on the heap, added to the map, and passed to the caller. The instance is stored in an *std::shared_ptr*, so that memory management is automatic and no manual release is necessary. If an instance already exists for the requested key, there are two possibilities. If both the new request and the existing instance allow multiple accesses to an instance, the existing instance is returned to the caller. If multiple access is not allowed, the return of *std::nullopt* signals that the hardware module is not available. In addition, the class *CInterfaceManager* class is made thread-safe by a mutex. However, this does not guarantee the thread safety of the interface instances; this is the responsibility of the individual classes. Since all accesses in this project are made from a single thread, the Control Comp, thread safety is guaranteed. The implementation of the *CInterfaceManager* class can be seen in Listing 20.

```cpp
template <typename InterfaceIdentifier, typename InterfaceType>
class CInterfaceManager{
public:
    CInterfaceManager(){}
    static std::optional<std::shared_ptr<InterfaceType>>
    ↪ getInstance(InterfaceIdentifier pId, bool pAllowMultipleInstances = false)
    {
        std::lock_guard<std::mutex> lock(mMtx);
        auto it = mInstances.find(pId);
        if (it != mInstances.end()) {
            if (pAllowMultipleInstances && it->second.second) {
                return it->second.first;
            }
            else {
                return std::nullopt;
            }
        }
        auto instance = std::shared_ptr<InterfaceType>(new InterfaceType(pId));
        mInstances[pId] = {instance, pAllowMultipleInstances};
    return instance;
    }
private:
    static inline std::map<InterfaceIdentifier,
    ↪ std::pair<std::shared_ptr<InterfaceType>, bool>> mInstances;
    static inline std::mutex mMtx;
};
```

Listing 20: Ausschnitt Implementierung des Interface-Managers

IMU

The IMU used is the *ICM20948* sensor, which is to be read out via SPI. The EMLs framework already contains an MMAP class for the SPI peripheral on the BBB. The class *CSPIModuleMMAP* provides a function for configuring communication (word length, frequency, start bit, etc.) and a function for exchanging data. The *CICM20948* class is intended to implement the logic for configuring the SPI interface and communication with the sensor.
The BBB acts as the SPI master, and the IMU acts as the SPI slave. The SPI protocol is full duplex, i.e., both the master and slave continuously send data during transmission. In this context, a read operation refers to the slave sending the contents of specified registers to the master. The master does not transmit any meaningful data during this time. In a write operation, the master sends data to the slave, which writes this data to a specified register. The slave sends dummy data during this process. The SPI interface is configured in accordance with the timing requirements in [15, S. 17] and the general description of SPI communication in [15, S. 31] of the *ICM20948* sensors. The word length is one byte, and each SPI operation consists of at least two bytes. During the first byte, the BBB transmits the address byte; the sensor does not send any valid data. The structure of the address byte is shown in Figure 8.10. The MSB indicates whether it is a read or write operation, and the remaining bits contain the register address. The address space of the sensor is divided into

four register banks. This means that the 7-bit register addresses can be assigned multiple times; which function is addressed depends on the currently selected bank.

In the case of a read operation, the BBB sends invalid data in all subsequent bytes of the SPI transmission, and the *ICM20948* sensor responds with the contents of the requested register. Burst reads are possible. If an SPI transmission consists of more than two bytes, the *ICM20948* sensor increments the register address and thus sends a memory area starting at the address specified in the address byte.
SPI communication works in the same way for write operations as it does for read operations, except that the *ICM20948* sensor also sends invalid data during the data bytes, while the BBB sends the register contents. The form of the data is shown in Figure 8.10.

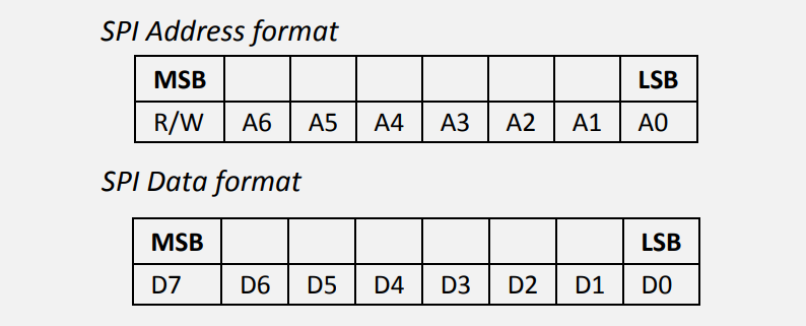


Figure 8.10: SPI Communication *ICM20948* [15, S. 31]

Two functions for read and write operations are implemented to implement the SPI communication described in the previous paragraph. The *burstRead()* function reads a variable-length memory area. The implementation of the function can be found in Listing 21:

```
1  CICM20948::Status CICM20948::burstRead(uint8_t pStartAddr, uint8_t pNumBytes,
   ↪  uint8_t *pData)
2  {
3      uint8_t transLen = pNumBytes + 1;
4      uint32_t rx[transLen] = {0};
5      uint32_t tx[transLen] = {0};
6      tx[0] = READ | pStartAddr;
7      auto status = mSPI->dataExchangeTxRx(mSPIChannel, tx, rx, transLen);
8      if (status != CSPIModuleMMAP::Status::OKAY) {
9          REPORT_ERROR("Imu burst read failed");
10         return Status::FAILED_TO_READ_SENSOR_DATA;
11     }
12     for (uint8_t i = 1; i < transLen; ++i) {
13         pData[i-1] = static_cast<uint8_t>(rx[i]);
14     }
15     return Status::OKAY;
16 }
```

Listing 21: Ausschnitt Implementierung der Funktion *burstRead()* aus *CICM20948*

Although IMU supports burst writes, writing individual registers is sufficient because write

operations are only used for configuring the sensor. The write operation is implemented in the function *writeReg ()*. An additional option for checking the register contents is implemented to ensure that the write operation was successful. The implementation of the function can be found in Listing 22.

```
1  CICM20948::Status CICM20948::writeReg(uint8_t pAddr, uint8_t pData, bool check)
2  {
3      uint8_t transLen = 2;
4      uint32_t rx[transLen] = {0};
5      uint32_t tx[transLen] = {0};
6      tx[0] = WRITE | pAddr;
7      tx[1] = pData;
8
9      auto status = mSPI->dataExchangeTxRx(mSPIChannel, tx, rx, transLen);
10     if (status != CSPIModuleMMAP::Status::OKAY) {
11         REPORT_ERROR("Imu write register failed");
12         return Status::FAILED_TO_READ_SENSOR_DATA;
13     }
14
15     if (check) {
16         uint8_t byte;
17         if (burstRead(pAddr, 1, &byte) != Status::OKAY || byte != pData) {
18             REPORT_ERROR("Imu writeReg failed, expected 0x" , std::hex,
               ↪  static_cast<int>(pData), " received: ", std::hex,
               ↪  static_cast<int>(byte));
19             return Status::FAILED_TO_READ_SENSOR_DATA;
20         }
21     }
22     return Status::OKAY;
23 }
```

Listing 22: Ausschnitt Implementierung der Funktion *writeReg()* aus *CICM20948*

The sensor is configured with the function *initImu()*. By default, all accelerometers and gyroscopes are put into operation at maximum sampling frequency. In addition, the scaling of the sensors and the cut-off frequency of the internal low-pass filter can be set. To do this, a software reset is first performed to bring the sensor into a defined state. This ensures that all registers are filled with the default values. Then, the *WHO_AM_I* register is used to check whether communication is working correctly. If the register contains the expected value, the accelerometers and gyroscopes are temporarily deactivated to ensure error-free configuration. The scaling and low-pass filters of the accelerometers and gyroscopes are then set to the desired values via the corresponding registers. Finally, the accelerometers and gyroscopes are reactivated.

In addition to the functions already described, a function *readImu()* is to be implemented that reads out all gyroscopes and accelerometers. For this purpose, it is particularly important to read all sensors as a burst read, since an SPI transfer is atomic. This means that the registers are not changed during an SPI read operation. If the individual bytes are read in several SPI transfers, the registers may be overwritten by the IMU, resulting in corrupt data being read. The implementation of the functions can be seen in Listing 23:

```
CICM20948::Status CICM20948::readImu(rawData &pData)
{
    uint8_t numBytes = 12;
    uint8_t rawData[numBytes] = {0};

    Status ret = burstRead(ACCEL_XOUT_H, numBytes, rawData);
    if (ret != Status::OKAY) {
        return Status::FAILED_TO_READ_SENSOR_DATA;
    }

    pData.xAccel = (rawData[0] << 8) | rawData[1];
    pData.yAccel = (rawData[2] << 8) | rawData[3];
    pData.zAccel = (rawData[4] << 8) | rawData[5];

    pData.xGyro  = (rawData[6] << 8) | rawData[7];
    pData.yGyro  = (rawData[8] << 8) | rawData[9];
    pData.zGyro  = (rawData[10] << 8) | rawData[11];

    return Status::OKAY;
}
```

Listing 23: Ausschnitt Implementierung der Funktion *readImu()* aus *CICM20948*

The class *CImu* is used as a wrapper for the *CICM20948* class. It simplifies the configuration of the sensors and calibrates the data during readout, so that the *readImu()* function of the *CImu* class directly contains the physical variables.

PWM motors

Both the drive wheel and the motor for the constant gyro speed are PWM-controlled motors. However, there are some differences in the details of the control. Therefore, the general PWM logic should be encapsulated in the *CMotorPWM* class, from which the device-specific classes for the two motors can inherit. The MMAP class *CPWMMMAP* for controlling the PWM hardware of the BBB is part of the framework from the EML.

The class *CMotorPWM* configures and manages access to the hardware with an instance of the *CPWMMMAP* class. It also provides a function that converts a physical setpoint into the corresponding duty cycle. This is done using a linear mapping that scales the setpoint from the physical range to the corresponding duty cycle. The implementation can be seen in Listing 24:

```
double CMotorPWM::calculateDutyCyclePercent(double pTarget)
{
    return (mPWMCfg.dutyCyclePercentMax - mPWMCfg.dutyCyclePercentMin) /
    ↪  (mPWMCfg.maxTarget - mPWMCfg.minTarget) * (pTarget - mPWMCfg.minTarget) +
    ↪  mPWMCfg.dutyCyclePercentMin;
}

CMotorPWM::Status CMotorPWM::setPWM(double pDutyCyclePercent)
{
    if (mPWM == nullptr) {
        return Status::PWM_MODULE_NOT_AVAILABLE;
    }
    if (mPWM->setDutyCycle(mPWMPin, pDutyCyclePercent) != CPWMMMAP::Status::OKAY)
    ↪  {
        return Status::PWM_MODULE_ERROR;
    }
    return Status::OKAY;
}
```

Listing 24: Ausschnitt Implementierung der Klasse *CMotorPWM*

In addition to controlling the enable output, the *CMaxonMotor* class also implements the logic for determining the speed of the drive wheel. The enable output, which can be used to activate and deactivate the motor, is controlled by an GPIO. The class *CGPIOMMAP* is also part of the EML framework. To determine the speed of the vehicle, the motor driver of the drive wheel is configured so that the current speed of the motor is output as an analog voltage. This voltage can be measured using the class *CADCMMAP*. The class *CSpin66Motor* implements the activation and deactivation of the motor by means of a specific PWM duty cycle (see Subsection 4.1.2).

ADC-Memory-Map

The ADC converter of the BBB is used to measure the analog voltage of the motor driver of the drive wheel, which contains information about the current speed. To ensure sufficiently fast access to the ADC hardware, control should be performed directly via the hardware registers (MMAP).

The ADC converter is designed as a sequence-controlled converter. The central components are 16 so-called steps. Each step contains the complete configuration for a measurement. This includes the input to be measured, the sample time, the operating mode, and the averaging of several measurements. The ADC converter processes all active steps cyclically and writes the results of the respective measurement to one of two First In First Out (FIFO) buffers. The measurement is tagged with the day of the corresponding step. There are two relevant operating modes: one-shot and continuous measurement. In one-shot measurements, the measurement configured in the step is performed once after the trigger; in continuous measurement, the measurements of the active steps are performed cyclically.
The hardware is to be controlled in class *CADCMMAP*. The individual steps are the resources that must be protected against multiple access. Therefore, class *CInterfaceManager*

ensures that one instance can be created per step. However, since multiple instances can access the same ADC converter in parallel, there is a risk of data loss. If one instance reads the FIFO buffer, measured values that are actually assigned to another step could be removed. For this reason, a static array *mValueQueues* is introduced, which contains a queue for each step. In addition, the function *readFifo()* (see Listing 25) is implemented, which reads all data from the associated FIFO buffer and appends it to the correct queue. To read the data, each instance can call this function so that all FIFO entries can be assigned to the correct step. Each step instance can then access the data in its own queue.

```
1  void CADCMMAP::readFifo()
2  {
3      ...
4      uint32_t regValue;
5      uint32_t stepIdx;
6      uint16_t data;
7      while (readRegister(fifoCountOffset) > 0) {
8          regValue = readRegister(fifoDataOffset);
9          stepIdx = (regValue & MASK_STEP_IDX) >> 16;
10         data = static_cast<uint16_t>(regValue & MASK_DATA);
11         mValueQueues[stepIdx].push(data);
12     }
13 }
```

Listing 25: Ausschnitt Implementierung der Funktion *readFifo()* aus der Klasse *CADCMMAP*

To read the current ADC data, the *readADC()* function is implemented. If the step is operated in one-shot mode, the measurement is triggered first. The *readFifo()* function is then called until until a measured value appears in the step's queue. If the step is operated in continuous mode, the function *readADC()* is called once to read all data from the FIFO. In both cases, the last value in the queue is then returned and all other data is discarded because it is no longer current. The implementation of the *readADC()* function can be seen in Listing 26.

```
CADCMMAP::Status CADCMMAP::readADC(uint16_t& pValue)
{
    switch (mADCConfig.mode)
    {
    case CADCConfig::Mode::ONESHOT:
        setBits(OFFS_STEPENABLE, 0x01 << mStepIdx);
        while (mValueQueues[mStepIdx-1].empty()) {
            readFifo();
        }
        break;
    case CADCConfig::Mode::CONTINUOUS:
        readFifo();
        break;
    default:
        return Status::CONFIG_ERROR;
    }
    bool newVal = false;
    // Only returns latest value, all other values are discarted
    while (!mValueQueues[mStepIdx-1].empty()) {
        pValue = mValueQueues[mStepIdx-1].front();
        mValueQueues[mStepIdx-1].pop();
        newVal = true;
    }
    if (!newVal) {
        REPORT_ERROR("No new value available");
        return Status::NO_VALUE;
    }
    return Status::OKAY;
}
```

Listing 26: Ausschnitt Implementierung der Funktion *readADC()* aus der Klasse *CADCMMAP*

Dynamixel servo motors

The Dynamixel servo motors are controlled by the UARTs of the BBB, which are controlled by the *CUARTMMAP* class, which is part of the EML framework. Communication follows the guidelines of the Dynamixel Protocol 2.0 (see [6] for specifications). The EML framework also includes a class *CDynamixelUARTMMAP*, which already implements basic communication with the Dynamixel Protocol 2.0. Based on this, a general function for reading and writing registers of a Dynamixel servo motor can be implemented. In addition, functions for writing and reading the registers relevant to this project (torque enable, bus watchdog, goal position, etc.) are created. The list of all registers and their functionality can be found in [7].

The classes *CPositionDynamixel* and *CVelocityDynamixel* configure a Dynamixel servo motor according to the operating mode and provide the necessary functions for controlling and activating the motors (*setPosition(), setVelocity(), setTorqueEnable()* etc.).

8.3.3 Comm-Comp

The Comm-Comp is responsible for sending vehicle data to the Python GUI via UDP. As with the Control-Comp, there are dedicated classes for the respective tasks. UDP communication is handled by the *CServer* class, which sends data packets in the form of *CMessage*. After communication has been initialized, the current vehicle data is sent at each sampling step. Synchronization is performed as described at the beginning of the chapter using a semaphore. The implementation of the loop responsible for sending the data can be seen in Listing 27:

```
1  void CCommComp::run()
2  {
3      if (!mClientConnected){
4          cout << "Comm end" << endl;
5          return;
6      }
7
8      cout << " CCommThread running " << endl;
9      while(runvar)
10     {
11         myContainer.getContent(true,mData);
12         if (!runvar) break;
13         if (!mServer.transmitMessage(mData)){
14             cout << "CCommThread: Client disconncted or error while transmitting
                   ↪  message" << endl;
15             break;
16         }
17     }
18     cout<<"Comm End"<<endl;
19 };
```

Listing 27: Ausschnitt Implementierung der Funktion *run()* aus *CCommComp*

data classes

The class *CMessage* class stores a uint8_t array. The length is determined by the size of the *CContent* class, which contains the vehicle data to be sent. The *setContent()* function stores an *CContent* object in the uint8_t array. This is necessary because only untyped byte sequences can be transmitted when sending UDP packets.

Server

The class *CServer* implements an UDP server and consists of three functions: The *init()* function creates and initializes an UDP socket and binds it to a fixed port. The *init()* function can be seen in Listing 28:

```
1  bool CServer::init()
2  {
3      mSocket = socket(AF_INET, SOCK_DGRAM, 0);
4      if (mSocket < 0) {
5          REPORT_ERROR_ERRNO("Socket failed to open");
6          return false;
7      }
8
9      sockaddr_in localAddr{};
10     localAddr.sin_family = AF_INET;
11     localAddr.sin_port = htons(Monowheeler::UDP_PORT);
12     localAddr.sin_addr.s_addr = INADDR_ANY;
13     if (bind(mSocket, (sockaddr*)&localAddr, sizeof(localAddr)) < 0) {
14         REPORT_ERROR_ERRNO("Failed to bind Socket");
15         return false;
16     }
17
18     return true;
19 }
```

Listing 28: Ausschnitt Implementierung der Funktion *init()* aus *CServer*

Since the BBB is always accessible under the same mDNS name and the port is not changed, the Python GUI can send messages to the Comm-Comp in a targeted manner. However, since the IP address of the development computer changes frequently, or different computers are used that are accessible in different ways, the Comm-Comp must wait for a message from the development computer. The *waitForClient ()* is responsible for this. The thread waits for a UDP message for a certain amount of time. When the expected message is received, the sender's address is stored and the connection is marked as „active“ (see Listing 29) .

```cpp
bool CServer::waitForClient(size_t pTimeout)
{
    fd_set readfds;
    FD_ZERO(&readfds);
    FD_SET(mSocket, &readfds);

    timeval timeout{};
    timeout.tv_sec = pTimeout;
    timeout.tv_usec = 0;

    char buffer;
    socklen_t addrLen = sizeof(mClientAddr);

    int result = select(mSocket + 1, &readfds, nullptr, nullptr, &timeout);
    if (result > 0 && FD_ISSET(mSocket, &readfds)) {
        ssize_t recvLen = recvfrom(mSocket, &buffer, sizeof(buffer), 0,
        ↪  (sockaddr*)&mClientAddr, &addrLen);
        if (recvLen > 0) {
            mConnected = true;
            return true;
        }
    }
    return false;
}
```

Listing 29: Ausschnitt Implementierung der Funktion *waitForClient()* aus *CServer*

Subsequently, *CContent* objects can be sent to the stored address using the *transmitMessage()* function. The implementation is shown in Listing 30.

```cpp
bool CServer::transmitMessage(CContent &pContent)
{
    CMessage msg;
    msg.setContent(pContent);
    if (!mConnected) {
        REPORT_ERROR("No client connected");
        return false;
    }

    ssize_t sent = sendto(mSocket, msg.mData, sizeof(CMessage), 0,
    ↪  (sockaddr*)&mClientAddr, sizeof(mClientAddr));
    return sent == (ssize_t)sizeof(CMessage);
}
```

Listing 30: Ausschnitt Implementierung der Funktion *transmitMessage()* aus *CServer*

8.4 Python GUI

A Python GUI is implemented to visualize and record the data from the experiments performed on the development computer. Since the GUI has to process and display the data in real time, the *PyQt5* framework is used. The logic of the framework is implemented in C++ and is specially optimized for continuously updated applications .

8.4.1 Main Program

The application basically consists of three modules: an UDP client *UDPClient*, the data plots *PlotManager*, and the data logger *DataRecorder*. The modules are managed by the main program *main.py*. The main program initializes the graphical user interface and delegates the data between the modules.
After the individual modules have been initialized, a background thread (see Listing 31) is started in which the UDP client continuously receives the vehicle data. The data is then passed on to the logger and stored in a queue. Meanwhile, the GUI is initialized in the main thread and a timer is started. In the timer callback (see Listing 32), the queue with the vehicle data is processed periodically and the data is transferred to the plot manager to update the GUI display elements.

```
def receive_loop():
try:
    while True:
        data = client.recv_message(ctypes.sizeof(CContent))
        msg = CContent.from_bytes(data)
        recorder.record(msg)
        data_queue.put(msg)
except Exception as e:
    print("Error while receiving data:", e)
except KeyboardInterrupt:
    print("End program")

thread = threading.Thread(target=receive_loop, daemon=True)
thread.start()
```

Listing 31: Ausschnitt Implementierung des Hintergrundthreads der Python-Gui

```
def timer_callback():
    while not data_queue.empty():
        msg = data_queue.get_nowait()
        plotter.update(msg)

timer = QTimer()
timer.timeout.connect(timer_callback)
timer.start(10)
```

Listing 32: Ausschnitt Implementierung des Timer-Callbacks der Python-Gui

8.4.2 UDP client

As described in subsubsection 8.3.3, the Python GUI sends an UDP message to the BBB to start communication. The BBB then begins sending the vehicle data to the address extracted from the UDP packet. The *UDPClient* class implements the logic for communication. To do this, an UDP socket is created and the message for initializing communication is sent to the BBB via mDNS (see Listing 33):

```
def send_initial_message(self):
    if self.sock is None:
        raise RuntimeError("Socket not initialized")
    initial_byte = b'\x01'
    try:
        self.ip = socket.gethostbyname(self.host)
        self.sock.sendto(initial_byte, (self.ip, self.port))
        print(f"Sent initial byte to {self.ip}:{self.port}")
        return True
    except Exception as e:
        print(f"Failed to send initial byte: {e}")
        return False
```

Listing 33: Ausschnitt Implementierung des *send_inital_message()*-Funktion der Python-Gui

In addition, a function for receiving the UDP packets of the BBB is implemented. To do this, the system waits for a certain amount of time to see if an UDP packet arrives at the socket. The implementation can be seen in Listing 34.

```
def recv_message(self, bufsize) -> bytes:
    if self.sock is None:
        raise RuntimeError("Socket not initialized")
    try:
        data, addr = self.sock.recvfrom(bufsize)
        return data
    except socket.timeout:
        print("Receive timed out")
        return b''
    except Exception as e:
        print(f"Receive error: {e}")
        return b''
```

Listing 34: Ausschnitt Implementierung des *recv_message()*-Funktion der Python-Gui

8.4.3 Data extraction

The data from the UDP packets must be interpreted correctly, as they are transmitted and received as a simple byte sequence. The *ctypes* module is used for this purpose, which makes it possible to implement C-like data structures in Python. This creates a data class

CContent is created, whose memory layout corresponds to that of the C++ class *CContent*. The incoming raw data can then be converted into an instance of the *CContent* class using the function *from_bytes()* (see Listing 35).

```
class CContent(ctypes.Structure):
    _fields_ = [
        ("mTimeUs", ctypes.c_int64),
        ("mStateData", CStateData),
        ("mMotorData", CMotorData),
    ]

    @classmethod
    def from_bytes(cls, data: bytes) -> "CContent":
        if len(data) != ctypes.sizeof(cls):
            raise ValueError(f"Expected {ctypes.sizeof(cls)} bytes, got
            ↪ {len(data)}")
        return cls.from_buffer_copy(data)
```

Listing 35: Ausschnitt Implementierung der *CContent*-Klasse der Python-Gui

8.4.4 data recording

The class *DataRecorder* records the received data in an CSV file. Depending on the selected mode, different subsets of the data (IMU, status, or motor information) are extracted and written to the file. The modes are configured according to a generic pattern, allowing the recorder to be adapted to new data formats very quickly. For each subset of data, a function is created that extracts this data from the data packet from the BBB. These functions are stored in an array so that this array can be iterated at runtime to record all subsets. The implementation of the *record()* function can be seen in Listing 36.

```
def record(self, msg: CContent):
    row = [msg.mTimeUs]
    for extractor in self.extractors:
        row += extractor(msg)
    self.writer.writerow(row)
```

Listing 36: Ausschnitt Implementierung des *record()*-Funktion der Python-Gui

8.4.5 Plot Manager

The central widget is provided by *PlotManager*. Depending on the selected plots (e.g., IMU values, state variables, motor data, etc.), the corresponding plot widgets are generated and arranged in a grid layout. Each plot is responsible for updating its own data and implements a uniform interface for this purpose. Similar to the recorder, all active plots are stored in an array, so that only this array needs to be iterated at runtime.
To standardize the implementation, there is an abstract base class `BasePlot`, which specifies the methods *update()* and *get_widget()* as abstract interfaces. All concrete plots (e.g.,

IMUPlot, *StatePlot*, *MotorPlot*) are derived from this class. This ensures a modular and extensible structure in which new plots can be added without changing the parent classes.

9 Summary

In this work, a novel concept for a mobile, actively stabilized, single-wheeled vehicle was developed and tested using the *Monowheeler*. To control the unstable pitch dynamics, the wheel can be shifted relative to the platform. The equally unstable roll motion is controlled by a gyroscope. By utilizing the coupled dynamics of the roll and yaw movements, yaw can be controlled in a targeted manner, enabling maneuvers to change direction. With the implementation of suitable control algorithms, the vehicle can be effectively stabilized and perform complex maneuvers such as changes of direction.

To achieve this goal, the existing hardware was first put into operation. Particular attention was paid to the implementation of suitable safety mechanisms for switching off the actuators in order to prevent damage to the hardware. A circuit board was designed and manufactured to connect the control signals and power electronics, the microcontroller, the actuators, and the sensors. In order to ensure the data integrity of the control signals despite the interfering power electronics, the layout was designed with special consideration given to the EMC. To this end, the signal and power electronics were spatially separated, separate ground planes with defined connection points were used, and the cable routing was designed so that return current paths remained as short and interference-free as possible. The targeted use of smoothing and support capacitors also ensured a stable supply voltage for all components. Overall, these measures resulted in clean signal routing and robust electrical behavior of the system.

To determine the current orientation of the vehicle under real-time requirements for the control loop, a suitable sensor system was selected and a corresponding algorithm for filtering and data fusion was implemented. An IMU is used, whose measured values for acceleration and angular velocity are filtered internally by the sensor. In addition, digital Butterworth low-pass filters are used to optimize the SNR ratio. For a robust estimation of the vehicle states, the measured data is then fused with a complementary filter.

To stabilize the inherently unstable vehicle, two largely independent subsystems were identified: the pitch subsystem and the roll and yaw subsystem. Suitable concepts for controlling the respective degrees of freedom were designed for both systems. An iterative process was used for this purpose, which uses analytically designed controllers as a starting point. The parameters were first optimized step by step using a simulation of the system behavior. The results were then validated and refined on the real vehicle. It was shown that the simplified model of the *Monowheelers* accurately represents the dynamics of the real vehicle when balancing. Classic and model-based controller concepts were designed and compared for both subsystems. A gain-scheduling LQR controller is used for the pitch subsystem, which offers better dynamics than a classic PID controller. The roll and yaw subsystems are considered separately for balancing purposes so that the degrees of freedom at the operating point do not influence each other. The roll movement can be effectively controlled with a PID controller. Since the control variable of the gyroscope is limited due

to the limited angular deflection, the controller must manipulate the orientation of the vehicle so that the gyroscope can always be rotated to the center. To do this, a second PID controller generates a trajectory for the roll movement. A model-based LQR controller could also be designed, but due to its structure, it does not offer the possibility of following other trajectories for the roll movement. Therefore, the system consisting of two PID controllers is used.

In addition to stabilizing the vehicle, this work aimed to investigate a concept for cornering. To this end, the coupling of the degrees of freedom of roll and yaw is exploited to achieve a targeted yaw movement through a roll movement. The roll movement is specified by a controller-based trajectory and implemented by the controller to control the roll movement. Several experiments have shown that this concept enables the vehicle to reliably navigate curves with variable radii.

A microcontroller with a Linux real-time kernel is used to control the hardware and implement the control loop for controlling the vehicle. The software was implemented in the *C++* programming language and provides a uniform interface for managing the microcontroller's hardware peripherals. In addition, the real-time suitability of the system was investigated. The vehicle data during an experiment can be visualized and recorded live with a Python GUI.

This enabled the goal of this work, as formulated in the task description, to be achieved: The *Monowheeler* was successfully stabilized using the gyroscopic effect, and the vehicle is capable of driving curves independently. The presented concept shows that the underactuated, unstable system with strongly coupled dynamics can be effectively controlled. Thus, a novel approach to the active stabilization and control of a single-wheeled vehicle was successfully implemented and experimentally validated.

10 Outlook

The experiments conducted also reveal the limitations of the current system. The mathematical model of the vehicle is primarily designed for balancing and ignores the dynamic effects of the drive wheel when cornering. Classic control approaches cannot control the underactuated multi-variable system of rolling and yawing as a complete system. However, by cascading several PID controllers, the system can be controlled with limitations. To do this, the desired effect is achieved by specifying a suitable trajectory for the roll movement. When balancing, the gyroscopic angle is minimized by tilting the vehicle. This means that the yaw movement cannot be controlled and balancing must be based on decoupling the roll and yaw movements. When cornering, the yaw movement should be controlled, but this means that the gyroscopic angle can no longer be taken into account.

This means that the vehicle can be effectively stabilized at the operating point for an unlimited period of time. When cornering, the vehicle can control the yaw movement as long as the gyroscope has sufficient reserve. However, cornering causes the vehicle to leave the operating point permitted for balancing, so that after dynamic maneuvers, the vehicle is sometimes unable to return to the stable operating point for balancing. This effect can be reduced by stopping the vehicle after a maneuver, as the operating point is larger when stationary due to greater static friction and the lack of dynamic coupling between rolling and yawing caused by the drive wheel.

In order to control the nonlinear, underactuated system with actuator limitations and tightly coupled degrees of freedom as a complete system, various approaches can be pursued in future work. All approaches require precise modeling of the vehicle's dynamics, including accurate knowledge of the friction parameters. An accurate model of the vehicle can be determined, for example, using the Lagrange formalism. The friction parameters can be determined experimentally through targeted measurements and subsequent model adjustment. Possible approaches to effectively control the overall system can be divided into three categories.

- **Advanced trajectory planners:** Advanced trajectory planners such as multiple shooting or sampling-based methods (e.g., RRT*) allow the calculation of complex movements for highly nonlinear and underactuated systems. They directly take into account system dynamics, couplings between degrees of freedom, and control variable limitations. Typically, special trajectories are optimized offline and can then serve as a reference for a feedback system.
- **Learning-based approaches:** Learning-based methods, such as reinforcement learning or model-based optimal control learning, make it possible to learn complex control strategies directly from simulations or experimental data. They are particularly suitable when the dynamics are highly nonlinear or cannot be modeled exactly, as they can adaptively take into account couplings and friction effects.

- **Model predictive approaches:** Model predictive control (MPC/NMPC) uses a mathematical model of the system to calculate optimal control variables in real time, taking into account system limits and constraints. They combine the advantages of trajectory planning and constraint handling and are particularly effective in enabling underactuated systems to follow precise trajectories, even with highly coupled degrees of freedom. Unlike trajectory planners, they are not limited to implementing a single, predetermined movement, but can control the system in general. However, they are very computationally intensive due to continuous optimization.

Bibliography

[1] *A guide to using the VL53L4CD ultra lite driver (ULD).* URL: https://www.st.com/resource/en/user_manual/um2931-a-guide-to-using-the-vl53l4cd-ultra-lite-driver-uld-stmicroelectronics.pdf (visited on 08/21/2025) (cit. on p. 31).

[2] *Balancing on Inline Wheels with Gyroscopes.* URL: https://www.youtube.com/watch?v=drxzs0Dnz14 (visited on 09/09/2025) (cit. on p. 69).

[3] *Best Practices for Board Layout of Motor Drivers.* URL: https://www.ti.com/lit/an/slva959b/slva959b.pdf?ts=1749690141678 (visited on 08/19/2025) (cit. on pp. 25, 26).

[4] Bloch, A. M. *Nonholonomic Mechanics and Control.* 2nd. Vol. 24. Interdisciplinary Applied Mathematics. With the collaboration of J. Baillieul, P. Crouch, J.E. Marsden, and D. Zenkov. © Springer-Verlag New York 2003, 2015. New York, Heidelberg, Dordrecht, London: Springer, 2015. ISBN: 978-1-4939-3016-6. DOI: 10.1007/978-1-4939-3017-3. URL: https://doi.org/10.1007/978-1-4939-3017-3 (cit. on pp. 8, 81).

[5] Brennan, L. "Means for imparting stability to unstable bodies". 796893. US Patent Office. 1905 (cit. on p. 3).

[6] *Dynamixel Protocol 2.0.* URL: https://emanual.robotis.com/docs/en/dxl/protocol2/ (visited on 09/18/2025) (cit. on p. 119).

[7] *Dynamixel XH540-W150-T/R.* URL: https://emanual.robotis.com/docs/en/dxl/x/xh540-w150/ (visited on 09/18/2025) (cit. on p. 119).

[8] Franklin, G. F., Powell, J. D., and Workman, M. L. *Digital Control of Dynamic Systems.* 3rd. Half Moon Bay, CA, USA: Ellis-Kagle Press, 1998, corrections and minor modifications 2020. URL: https://www.researchgate.net/profile/Jd-Powell-2/publication/243772327_Digital_Control_of_Dynamic_Systems/links/65789e08fc4b416622bb4deb/Digital-Control-of-Dynamic-Systems.pdf (cit. on pp. 49, 55).

[9] Goodwin, G. C., Graebe, S. F., and Salgado, M. E. *Control System Design.* Prentice Hall, 2000. ISBN: 9780139586538 (cit. on pp. 49, 55, 56, 58).

[10] *Gyro-X-1967.* URL: https://www.lanemotormuseum.org/wp-content/uploads/2014/12/gyrox_1967web1a-cdd.jpg (visited on 09/20/2025) (cit. on p. 3).

[11] Hang, C., Åström, K., and Ho, W. "Refinements of the Ziegler–Nichols tuning formula". In: *IEE Proceedings D (Control Theory and Applications)* 138 (2 1991), pp. 111–118. DOI: 10.1049/ip-d.1991.0015. eprint: https://digital-library.theiet.org/doi/pdf/10.1049/ip-d.1991.0015. URL: https://digital-library.theiet.org/doi/abs/10.1049/ip-d.1991.0015 (cit. on p. 50).

[12] Henrich, L. A. "Evaluation of IMU Orientation Estimation Algorithms Using a Three-Axis Gimbal". Friedrich-Alexander-Universitat Erlangen-Nürnberg (FAU), 2019. URL: https://www.mad.tf.fau.de/files/2020/02/BT_LeaHenrich_Final.pdf (visited on 08/22/2025) (cit. on p. 33).

[13] *Hespanha-Linear Systems-Part IV*. URL: https://metr4202.uqcloud.net/2014/lectures/Hespanha-Linear%20Systems-Part%20IV.pdf (visited on 09/05/2025) (cit. on p. 59).

[14] Hose, H., Weisgerber, J., and Trimpe, S. *The Mini Wheelbot: A Testbed for Learning-based Balancing, Flips, and Articulated Driving*. 2025. arXiv: 2502.04582 [cs.RO]. URL: https://arxiv.org/abs/2502.04582 (cit. on pp. 4, 84).

[15] *ICM-20948 World's Lowest Power 9-Axis MEMS MotionTracking™ Device*. URL: https://invensense.tdk.com/wp-content/uploads/2016/06/DS-000189-ICM-20948-v1.3.pdf (visited on 08/21/2025) (cit. on pp. 31, 35, 36, 40, 113, 114).

[16] *IPD85P04P4L-06 datasheet*. URL: https://www.mouser.de/datasheet/2/196/Infineon_IPD85P04P4L_06_DataSheet_v01_01_EN-3362745.pdf (visited on 08/19/2025) (cit. on pp. 26, 27).

[17] *ISS Motion Control System*. URL: https://ntrs.nasa.gov/api/citations/20240003654/downloads/ISS_MCS_Ops_POIWG_April24_Final_PDF.pdf (visited on 09/20/2025) (cit. on p. 3).

[18] Jacot, A. D. and Liska, D. J. "Control Moment Gyros in Attitude Control". In: *Journal of Spacecraft and Rockets* 3.9 (1966), pp. 1313–1320 (cit. on p. 3).

[19] *James Bruton - One-Wheel Balancing Robot*. URL: https://www.youtube.com/watch?v=fNQkZ7MmGio (visited on 09/23/2025) (cit. on p. 5).

[20] *Jeti Spin 66 opto pro*. URL: https://www.hacker-motor-shop.com/SPIN-66-Pro-OPTO.htm?a=article&ProdNr=51007011&p=10675 (visited on 08/15/2025) (cit. on p. 20).

[21] Lappas, V. J., Steyn, W. H., and Underwood, C. I. "Attitude Control for Small Satellites Using Control Moment Gyros". In: *Acta Astronautica* 51.1-9 (2002), pp. 101–111. DOI: 10.1016/S0094-5765(02)00089-9 (cit. on p. 3).

[22] Lew, E., Orazov, B., and O'Reilly, O. "The dynamics of Charles Taylor's remarkable one-wheeled vehicle". In: *Regular and Chaotic Dynamics* 13 (Aug. 2008), pp. 257–266. DOI: 10.1134/S1560354708040035 (cit. on pp. 1, 5).

[23] *Maxon Escon 70/10 Gerätereferenz*. URL: https://www.maxongroup.de/medias/sys_master/root/9350562643998/422969-ESCON-70-10-Geraete-Referenz-De.pdf (visited on 08/15/2025) (cit. on pp. 20, 21).

[24] *MEGA MOTOR ACn 22/20/3*. URL: https://www.megamotor.cz/v4/scriptx/default.php?&sid=6c18bedb3cbe944b86bd8fcaa81a31b6&page_id=lang_ger (visited on 08/15/2025) (cit. on p. 20).

[25] Mellodge, P. "Chapter 4 - Characteristics of Nonlinear Systems". In: *A Practical Approach to Dynamical Systems for Engineers*. Ed. by Mellodge, P. Woodhead Publishing, 2016, pp. 215–250. ISBN: 978-0-08-100202-5. DOI: https://doi.org/10.1016/B978-0-08-100202-5.00004-8. URL: https://www.sciencedirect.com/science/article/pii/B9780081002025000048 (cit. on p. 63).

[26] Meyer, M. *Signalverarbeitung: Analoge und digitale Signale, Systeme und Filter*. 9., korrigierte Auflage. Springer Vieweg, 2021. ISBN: 978-3-658-32800-9. DOI: 10.1007/978-3-658-32801-6. URL: https://link.springer.com/book/10.1007/978-3-658-32801-6 (cit. on p. 43).

[27] *Mini Wheelbot*. URL: https://www.linkedin.com/posts/sebastian-trimpe-2472a0a3_icra2025-ieee-robotics-activity-7328901914195103746-zr5j (visited on 09/20/2025) (cit. on p. 4).

[28] *Model Predictive Control*. URL: https://www.ist.uni-stuttgart.de/research/group-of-frank-allgoewer/model-predictive-control/?utm_source=chatgpt.com (visited on 09/06/2025) (cit. on p. 55).

[29] Oppenheim, A. V. and Schafer, R. W. *Discrete-Time Signal Processing*. 3rd. Pearson, 2010. ISBN: 978-0-13-198842-2 (cit. on pp. 35, 43).

[30] Rahiman, W., Ghazali, M. H. M., and Mahmood, I. A.-T. "Revolutionizing Marine Stability: A Review of Cutting-Edge Gyro Stabilizer Designs and Techniques for Vessels". In: *IEEE Access* 13 (2025), pp. 89542–89555. DOI: 10.1109/ACCESS.2025.3571710 (cit. on p. 3).

[31] *Robust control (H infinity and H2)*. URL: https://dycsyt.com/en/robust-control-h-infinity-and-h2/ (visited on 09/05/2025) (cit. on p. 55).

[32] *RS-485 Basics: When Termination Is Necessary, and How to Do It Properly*. URL: https://www.ti.com/lit/ta/ssztb23/ssztb23.pdf?utm_source=chatgpt.com&ts=1755544148510&ref_url=https%253A%252F%252Fchatgpt.com%252F (visited on 08/19/2025) (cit. on p. 28).

[33] Ruben Egle, S. W. "Entwurf und Konzeption eines Monowheelers". Hochschule Karlsruhe, 2024 (cit. on pp. 2, 5, 7–11, 13, 14, 19, 20, 22, 23, 26, 27, 36, 49–51, 55, 66).

[34] Shen, J. and Hong, D. "OmBURo: A Novel Unicycle Robot with Active Omnidirectional Wheel". In: *2020 IEEE International Conference on Robotics and Automation (ICRA)*. 2020, pp. 8237–8243. DOI: 10.1109/ICRA40945.2020.9196927 (cit. on p. 4).

[35] Shilovsky, P. *The Gyroscope: Its Practical Construction and Application*. E. & F.N. Spon, Limited, 1924. URL: https://books.google.com.my/books?id=8kIkAAAAMAAJ (cit. on p. 3).

[36] Taylor, C. F. "One-Wheeled Vehicle". 3145797. US Patent Office. 1964 (cit. on pp. 1, 5).

[37] *The Gyro Monorail: How To Make Trains Better With A Gyroscope*. URL: https://hackaday.com/wp-content/uploads/2024/01/Brennan_monorail-copy.jpg?w=800 (visited on 09/20/2025) (cit. on p. 3).

[38] *THVD1406, THVD1426 3.3-V to 5-V RS-485 Transceivers with Auto-direction Control and ±12-kV IEC ESD Protection*. URL: https://www.ti.com/lit/ds/symlink/thvd1426.pdf?ts=1755621647998&ref_url=https%253A%252F%252Fwww.ti.com%252Fproduct%252Fde-de%252FTHVD1426 (visited on 08/19/2025) (cit. on p. 27).

[39] Wetzstein, G. *EE 267 Virtual Reality Course Notes: 3-DOF Orientation Tracking with IMUs*. URL: https://stanford.edu/class/ee267/notes/ee267_notes_imu.pdf (visited on 08/22/2025) (cit. on pp. 32, 33).

[40] Wietzke, J. *Embedded Technologies*. Springer Berlin, Heidelberg, 2012 (cit. on p. 98).

[41] Wietzke, J. and Tran, M. T. *Automotive Embedded Systeme*. Springer Berlin, Heidelberg, 2005 (cit. on p. 98).

[42] Wietzke, P. D.-I. J. and Meindl, M. M. *Mechatronische Systeme Vorlesungsskript*. Hochschule Karlsruhe (cit. on pp. 33, 34).

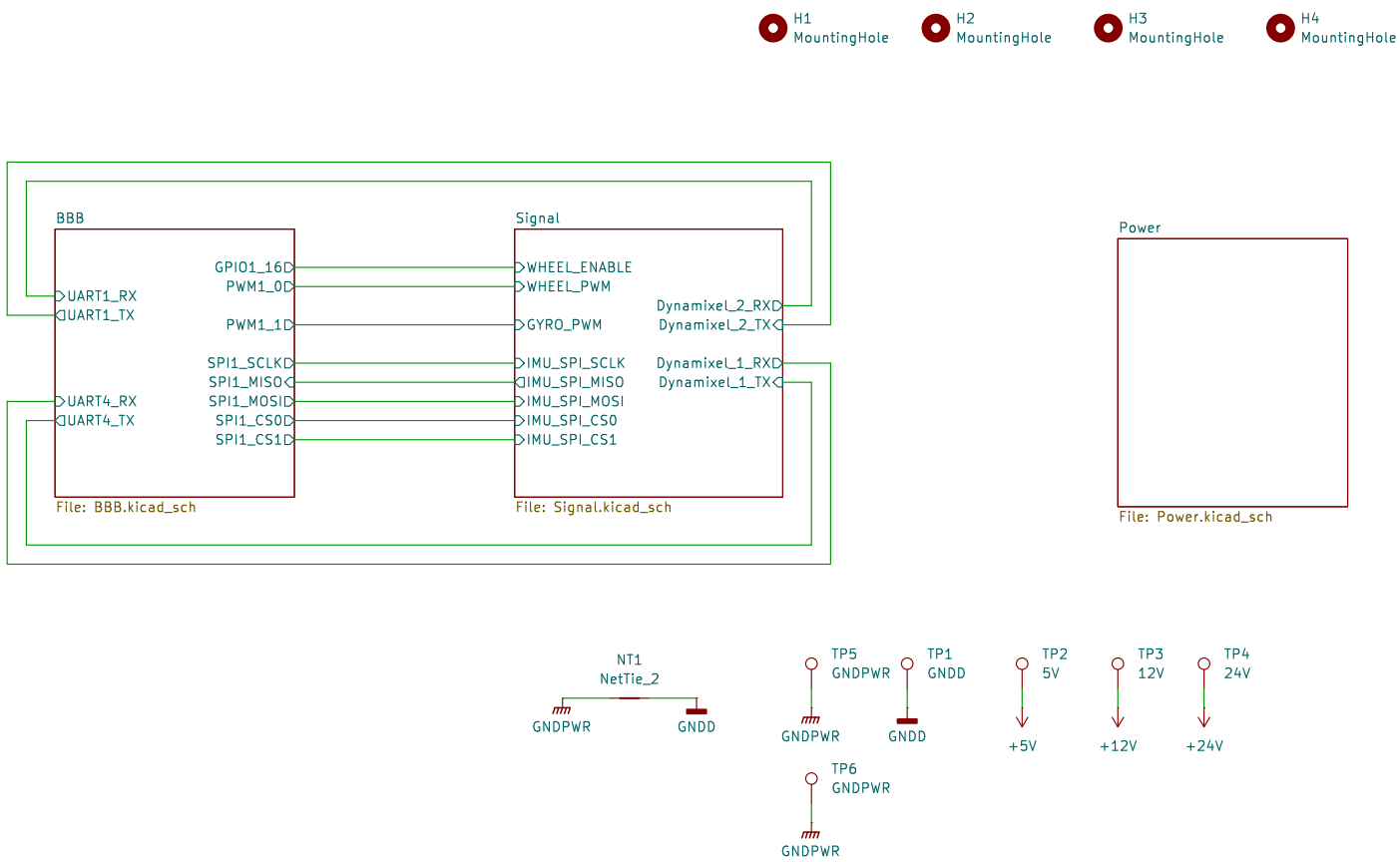
[43] Xu, Y. and Sun, L. W. "Dynamics of a Rolling Disk and a Single Wheel Robot on an Inclined Plane". In: *Proceedings of the 2000 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2000)*. *. IEEE. Takamatsu, Japan, 2000 (cit. on pp. 8, 81).

List of Figures

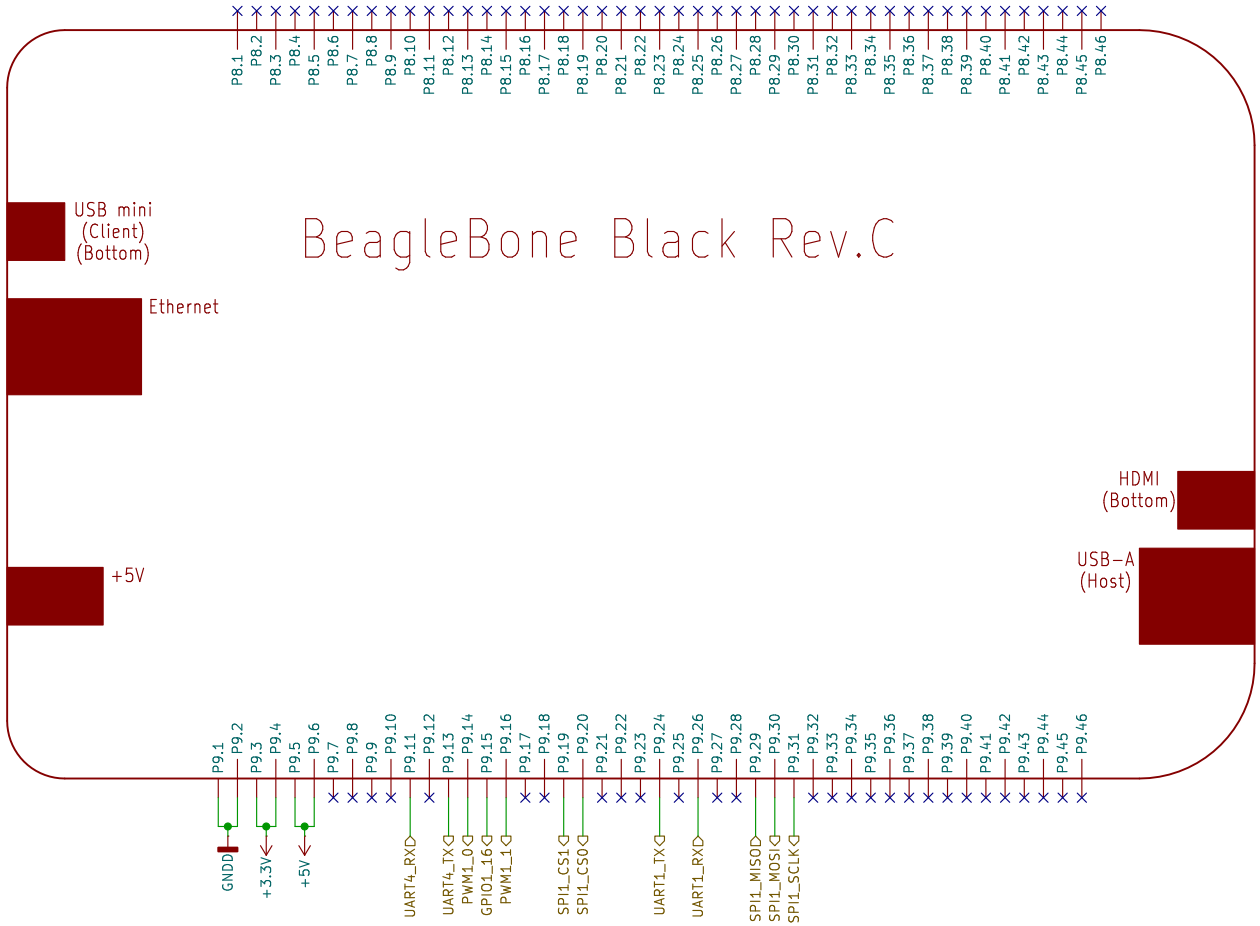
List of Tables

List of Listings

A Circuit Diagram



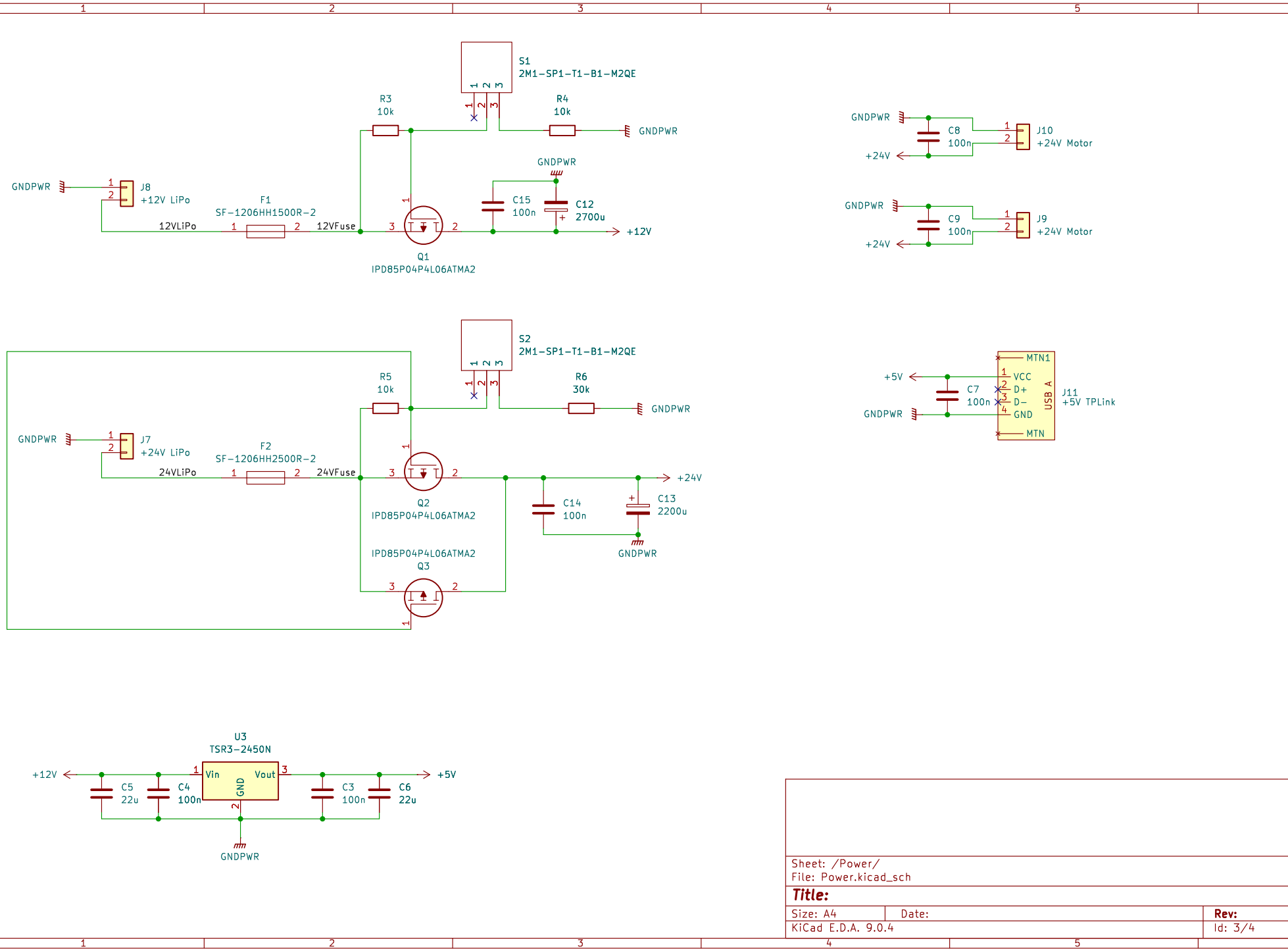
Sheet: / File: MonowheelerPCBV2.kicad_sch		
Title:		
Size: A4	Date:	**Rev:**
KiCad E.D.A. 9.0.4		Id: 1/4

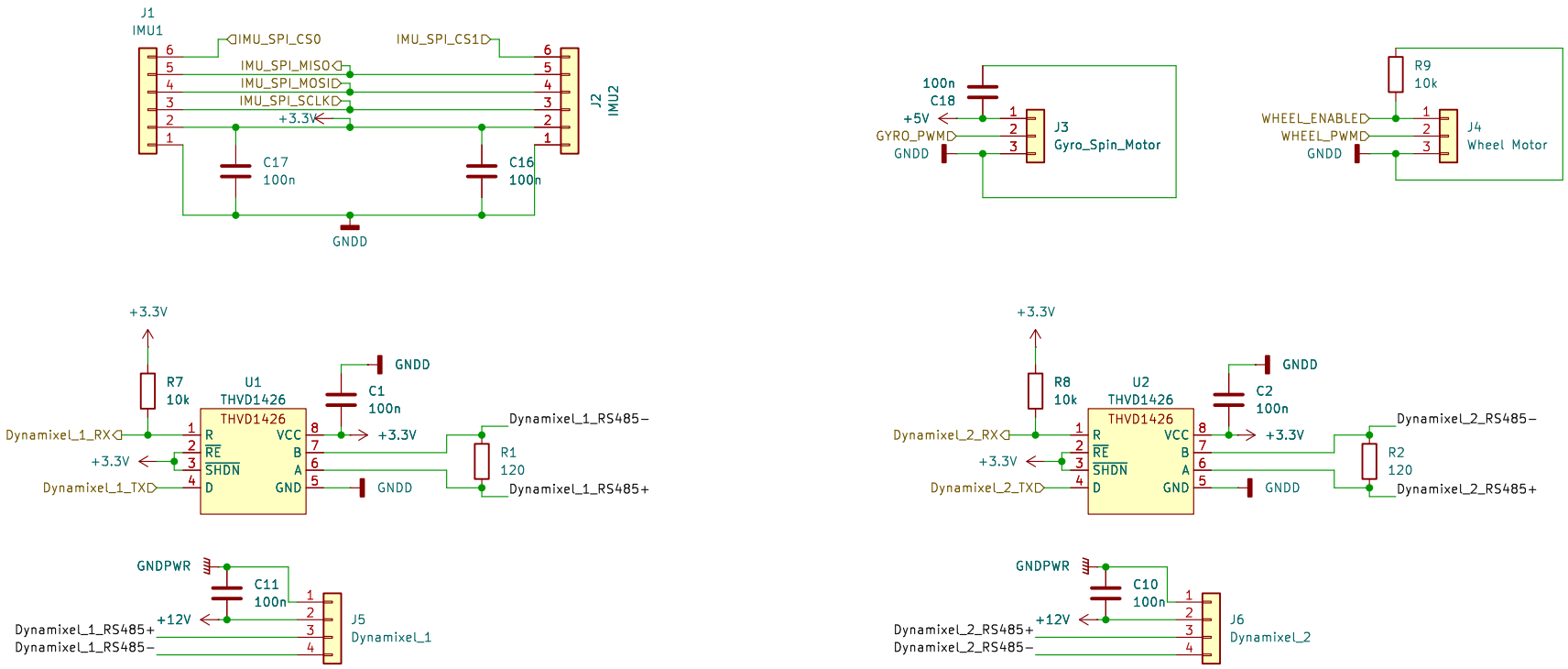


Sheet: /BBB/
File: BBB.kicad_sch

Title:

Size: A4	Date:	**Rev:**
KiCad E.D.A. 9.0.4		Id: 2/4


S1
2M1-SP1-T1-B1-M2QE
R3
10k
R4
10k
GNDPWR
J8
+12V LiPo
F1
SF-1206HH1500R-2
12VLiPo
12VFuse
C15
100n
C12
2700u
+12V
Q1
IPD85P04P4L06ATMA2
C8
100n
J10
+24V Motor
+24V
C9
100n
J9
+24V Motor
S2
2M1-SP1-T1-B1-M2QE
R5
10k
R6
30k
J7
+24V LiPo
F2
SF-1206HH2500R-2
24VLiPo
24VFuse
Q2
IPD85P04P4L06ATMA2
Q3
IPD85P04P4L06ATMA2
C14
100n
C13
2200u
+24V
+5V
C7
100n
MTN1
VCC
D+
D−
GND
MTN
USB A
J11
+5V TPLink
U3
TSR3-2450N
Vin
GND
Vout
C5
22u
C4
100n
C3
100n
C6
22u
+12V
+5V
Sheet: /Power/
File: Power.kicad_sch
Title:
Size: A4
Date:
Rev:
KiCad E.D.A. 9.0.4
Id: 3/4



Sheet: /Signal/
File: Signal.kicad_sch

Title:

Size: A4	Date:	**Rev:**
KiCad E.D.A. 9.0.4		Id: 4/4

B Signal Analysis

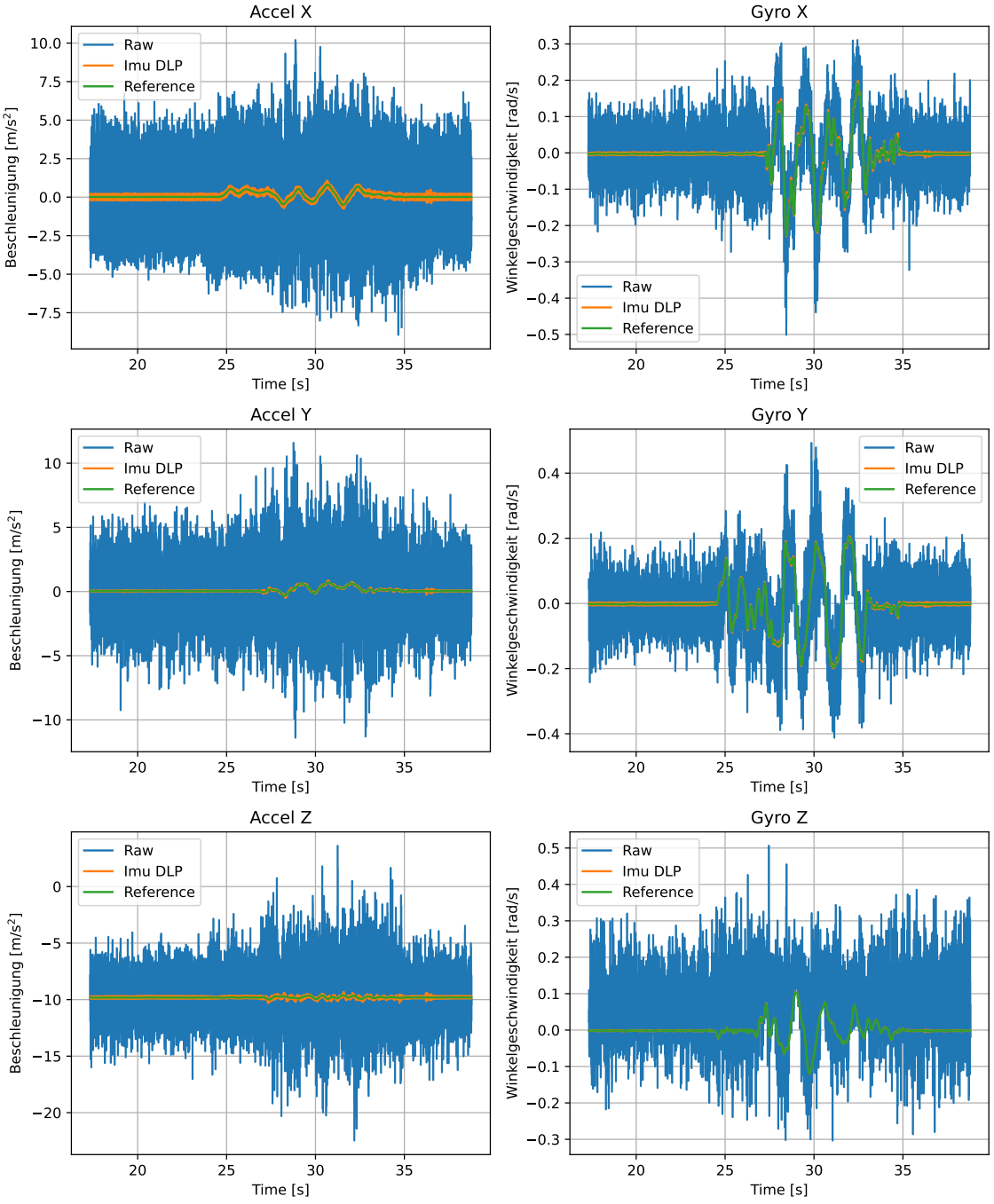


Figure B.1: IMU Daten mit IMU-DLP mit $f_G = 5\,\mathrm{Hz}$

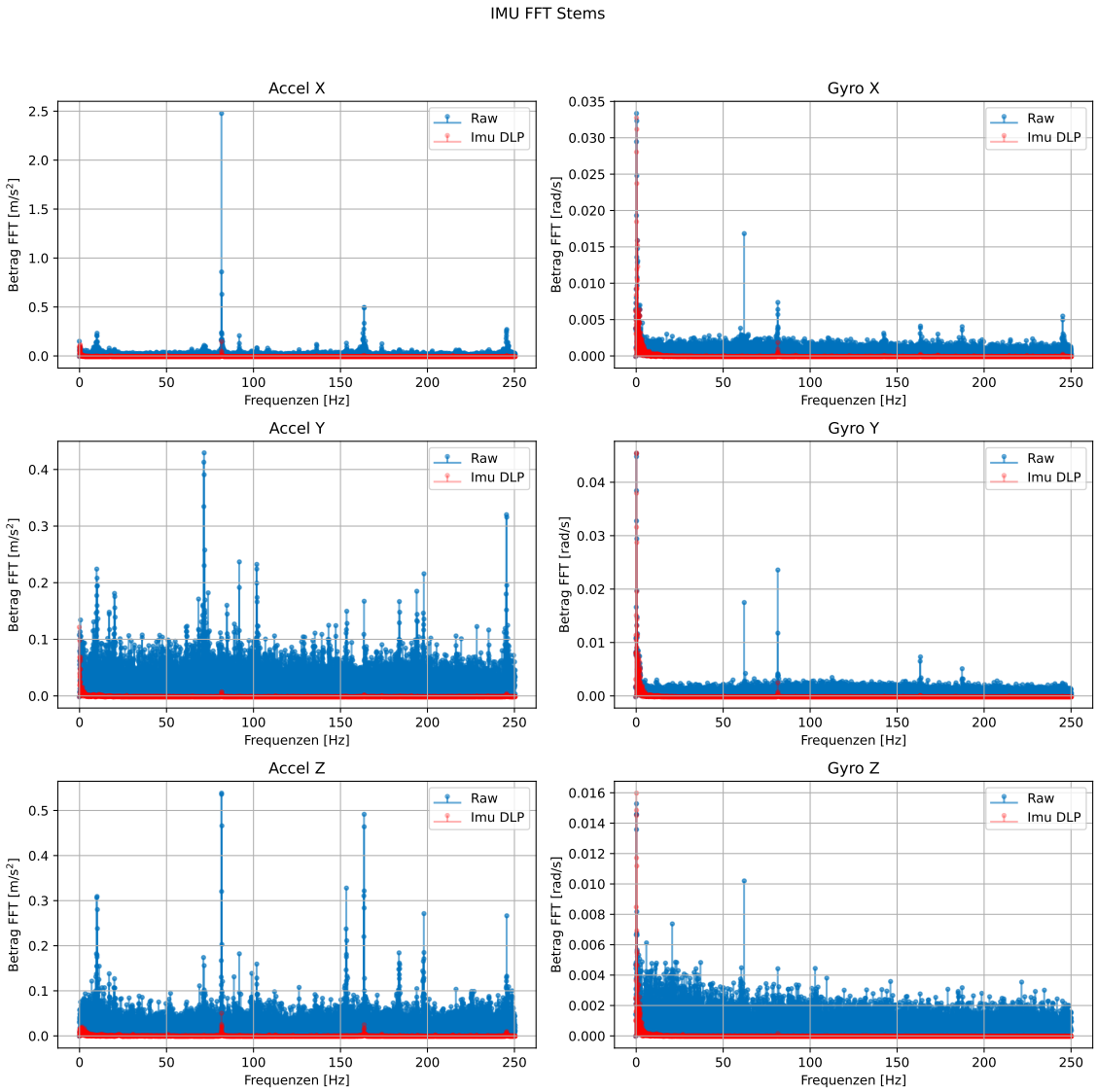


Figure B.2: IMU Daten mit IMU-DLP mit $f_G = 5\,\text{Hz}$ FFT

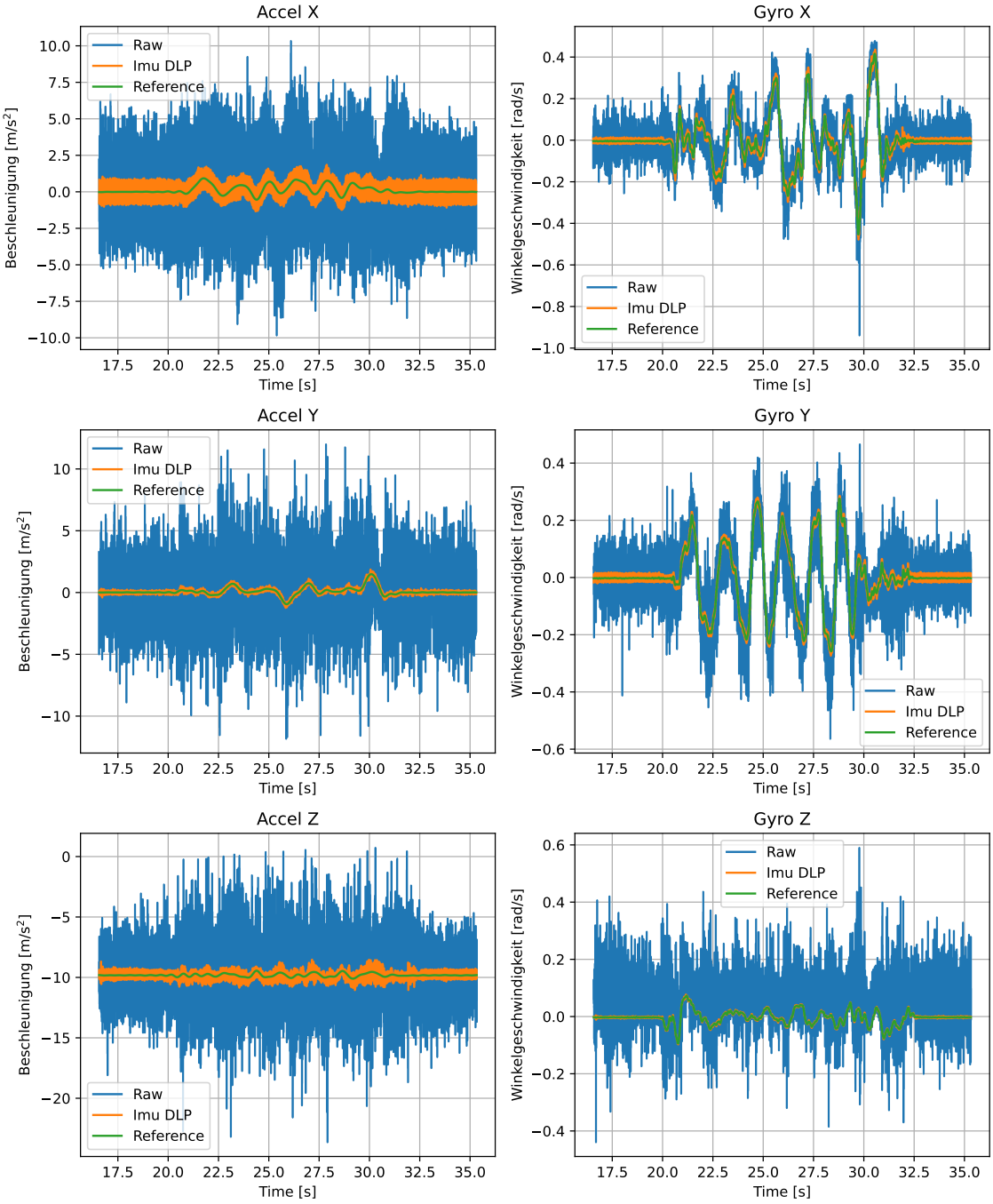


Figure B.3: IMU Daten mit IMU-DLP mit $f_G = 23\,\mathrm{Hz}$

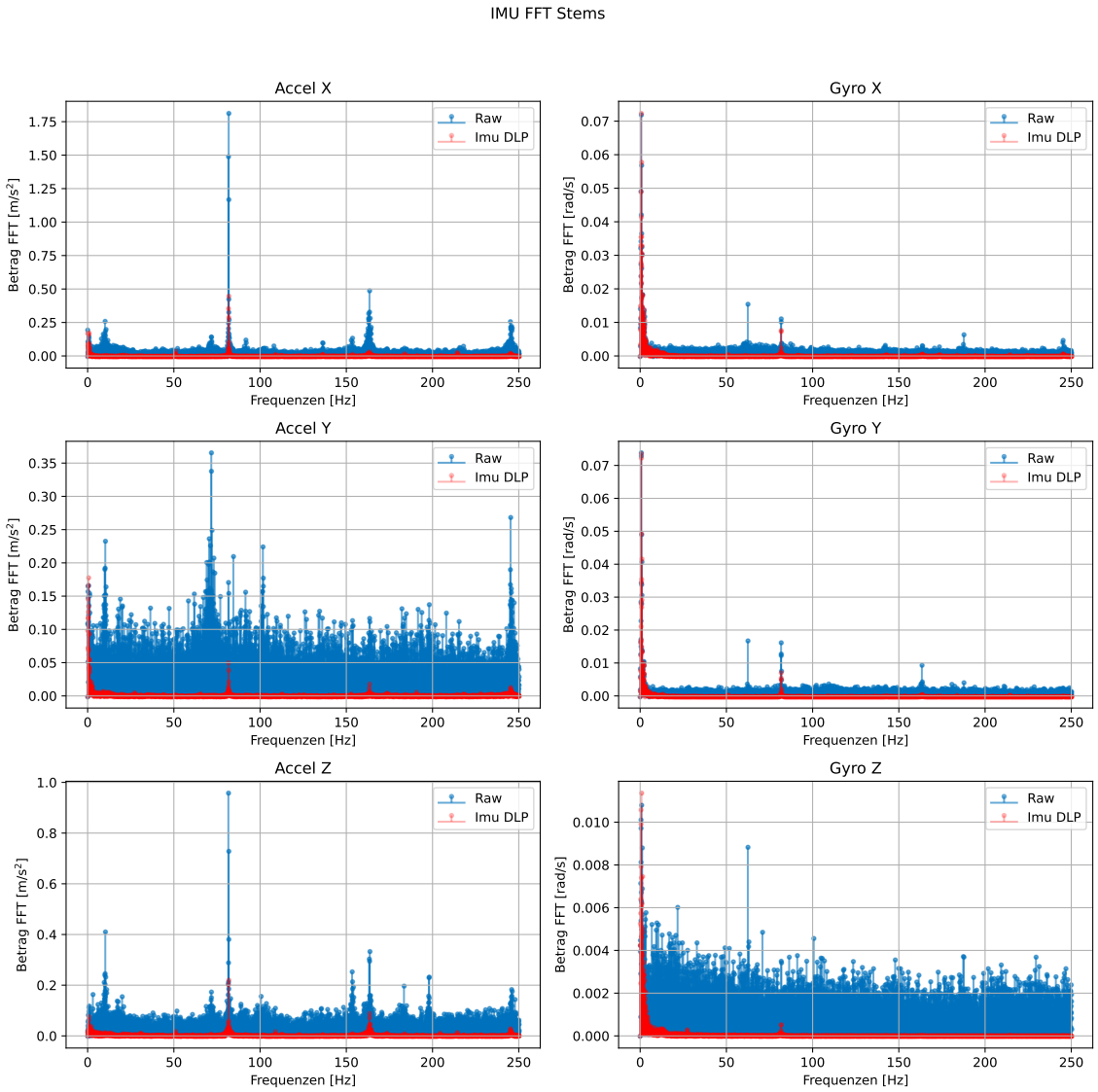


Figure B.4: IMU Daten mit IMU-DLP mit $f_G = 23\,\text{Hz}$ FFT

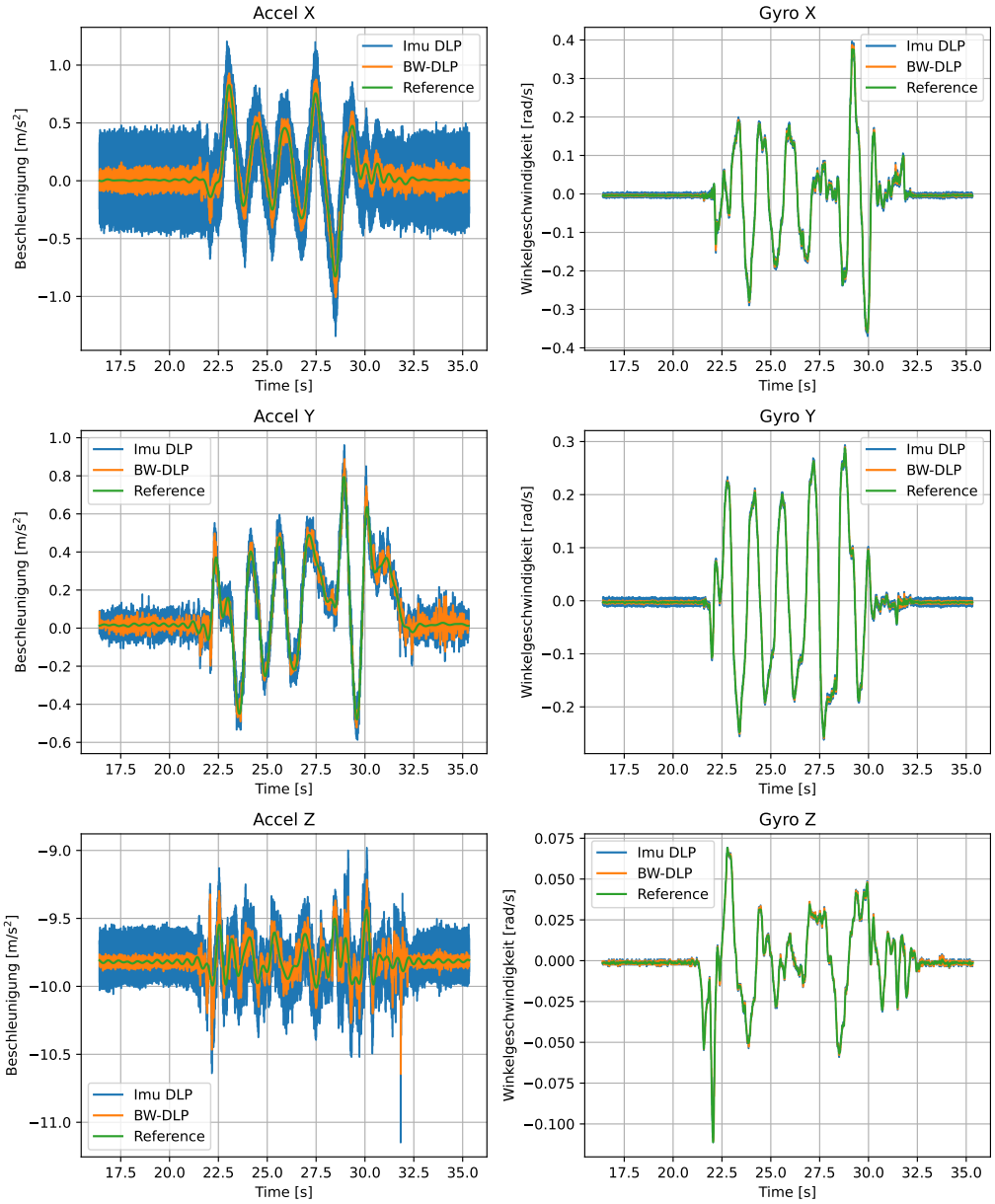


Figure B.5: IMU Daten mit IMU-DLP mit $f_G = 11\,\text{Hz}$ und Butterworth-Tiefpass mit $f_G = 40\,\text{Hz}$

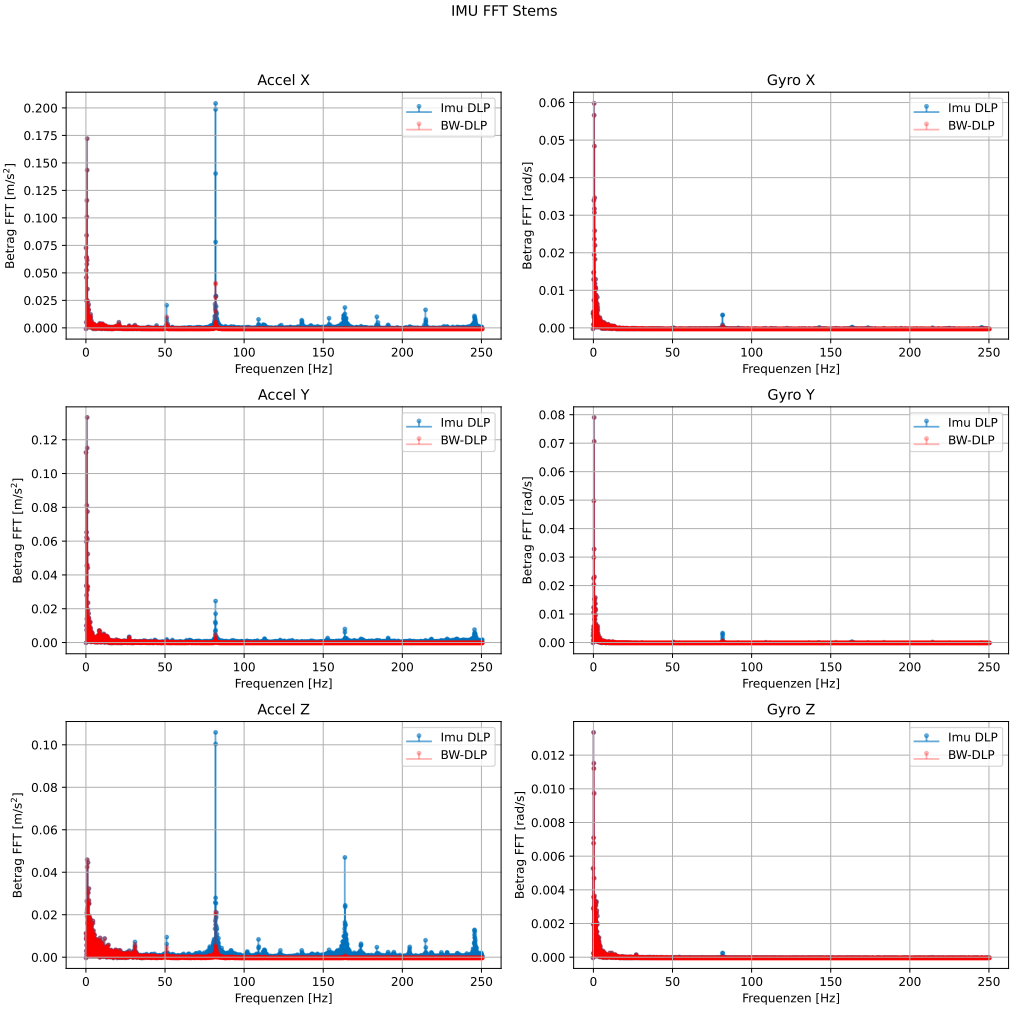


Figure B.6: IMU Daten mit IMU-DLP mit $f_G = 11\,\text{Hz}$ und Butterworth-Tiefpass mit $f_G = 40\,\text{Hz}$ FFT

C Code

C.1 CPitchController

```
double CPitchController::control(double pTheta, double pDotTheta, double
↪  pDynamixel)
{
    switch (mMode)
    {
    case Mode::Slow:
        if (std::abs(pTheta) > mThetaUpperThreshold || std::abs(pDotTheta) >
        ↪  mDotThetaUpperThreshold) {
            mMode = Mode::TransitionSlow2Fast;
            mTTransition = 0;
        }
        break;
    case Mode::Fast:
        if (std::abs(pTheta) < mThetaLowerThreshold && std::abs(pDotTheta) <
        ↪  mDotThetaLowerThreshold) {
            mTSlow += mTa;
            if (mTSlow > mTSlowThreshold) {
                mMode = Mode::TransitionFast2Slow;
                mTTransition = 0;
                mTSlow = 0;
            }
        }
        break;
    case Mode::TransitionSlow2Fast:
        if (mTTransition > mTTransitionMax) {
            mMode = Mode::Fast;
        }
        break;
    case Mode::TransitionFast2Slow:
        if (mTTransition > mTTransitionMax) {
            mMode = Mode::Slow;
        }
        break;
    default:
        break;
    }

    mThetaSum += - pTheta*mTa;

    double uCMDFast;
    double uCMDSlow;
    double blend;

```

```
41      switch (mMode)
42      {
43      case Mode::Fast:
44          mUCMD = -std::clamp((mLQRFast[0]*pTheta
45                          + mLQRFast[1]*pDotTheta
46                          + mLQRFast[2]*mThetaSum
47                          + mLQRFast[3]*pDynamixel), -mUCMDMax, mUCMDMax);
48          break;
49      case Mode::Slow:
50          mUCMD = -std::clamp((mLQRSlow[0]*pTheta
51                          + mLQRSlow[1]*pDotTheta
52                          + mLQRSlow[2]*mThetaSum
53                          + mLQRSlow[3]*pDynamixel), -mUCMDMax, mUCMDMax);
54          break;
55      case Mode::TransitionSlow2Fast:
56          uCMDFast = -std::clamp((mLQRFast[0]*pTheta
57                          + mLQRFast[1]*pDotTheta
58                          + mLQRFast[2]*mThetaSum
59                          + mLQRFast[3]*pDynamixel), -mUCMDMax, mUCMDMax);
60          uCMDSlow = -std::clamp((mLQRSlow[0]*pTheta
61                          + mLQRSlow[1]*pDotTheta
62                          + mLQRSlow[2]*mThetaSum
63                          + mLQRSlow[3]*pDynamixel), -mUCMDMax, mUCMDMax);
64          blend = std::clamp(mTTransition/mTTransitionMax, 0.0, 1.0);
65          mUCMD = blend*uCMDFast + (1-blend)*uCMDSlow;
66          mTTransition += mTa;
67          break;
68      case Mode::TransitionFast2Slow:
69          uCMDFast = -std::clamp((mLQRFast[0]*pTheta
70                          + mLQRFast[1]*pDotTheta
71                          + mLQRFast[2]*mThetaSum
72                          + mLQRFast[3]*pDynamixel), -mUCMDMax, mUCMDMax);
73          uCMDSlow = -std::clamp((mLQRSlow[0]*pTheta
74                          + mLQRSlow[1]*pDotTheta
75                          + mLQRSlow[2]*mThetaSum
76                          + mLQRSlow[3]*pDynamixel), -mUCMDMax, mUCMDMax);
77          blend = std::clamp(mTTransition/mTTransitionMax, 0.0, 1.0);
78          mUCMD = blend*uCMDSlow + (1-blend)*uCMDFast;
79          mTTransition += mTa;
80          break;
81      default:
82          mUCMD = 0;
83          break;
84      }
85      return mUCMD;
86  }
```

C.2 CPIDController

```
1  double CPIDController::control(const double pW, const double pX)
2  {
3      mE = pW - pX;
```

```
    mDotE = (mE -mEPrev) / mPID.Ta;
    if (std::abs(mU) < mPID.UMax) {
        mSumE += mE*mPID.Ta;
    }
    mU = std::clamp(mDirection*(mPID.KP*mE + mPID.KD*mDotE + mPID.KI*mSumE),
    ↪ -mPID.UMax, mPID.UMax);
    mEPrev = mE;
    return mU;
}

double CPIDController::control(const double pW, const double pDotW, const double
↪ pX, const double pDotX)
{
    mE = pW - pX;
    mDotE = pDotW - pDotX;
    if (std::abs(mU) < mPID.UMax) {
        mSumE += mE*mPID.Ta;
    }
    mU = std::clamp(mDirection*(mPID.KP*mE + mPID.KD*mDotE + mPID.KI*mSumE),
    ↪ -mPID.UMax, mPID.UMax);
    mEPrev = mE;
    return mU;
}
```

C.3 CController

```
void CController::update(CCommand& pCmd, CStateData& pStateData, CMotorData&
↪ pMotorData)
{
    double phiTargetNew, dotPhiTarget;
    switch (mMode)
    {
    case EMode::Balance:
        phiTargetNew = mRollTargetBalanceController.control(0, 0,
        ↪ pMotorData.mGyroDynamixelPosition, pMotorData.mGyroDynamixelVelocity);
        dotPhiTarget = (phiTargetNew - pStateData.mPhiTarget)/mTa;

        pStateData.mPhiTarget = phiTargetNew;
        pStateData.mSpeedTarget = ramp(mTargetVelocity, pStateData.mSpeedTarget,
        ↪ Monowheeler::MAX_ACCEL);
        pStateData.mDotPsiTarget = mDotPsiTarget;
        pStateData.mThetaTarget = 0;

        CCommand::ECommands cmd;
        double cmdData;
        if (pCmd.getData(cmd, cmdData)) {
            switch (cmd)
            {
            case CCommand::ECommands::Drive:
                mTargetVelocity = std::clamp(cmdData, -Monowheeler::MAX_SPEED,
                ↪ Monowheeler::MAX_SPEED);
                mDotPsiTarget = 0.0;
```

```
23                  break;
24              case CCommand::ECommands::CornerLeft:
25                  mTargetVelocity = 0.7;
26                  mCornerAcceleration = true;
27                  mCornerRight = false;
28                  mSlalom = false;
29                  break;
30              case CCommand::ECommands::CornerRight:
31                  mTargetVelocity = 0.7;
32                  mCornerAcceleration = true;
33                  mCornerRight = true;
34                  mSlalom = false;
35                  break;
36              case CCommand::ECommands::Slalom:
37                  mTargetVelocity = 0.7;
38                  mCornerAcceleration = true;
39                  mCornerRight = true;
40                  mSlalom = true;
41              default:
42                  break;
43              }
44          }
45
46          if (mCornerAcceleration && abs(mTargetVelocity - pStateData.mSpeed) <
            ↪  0.05) {
47              mCornerAcceleration = false;
48              mMode = EMode::Corner;
49              if (mCornerRight) {
50                  mDotPsiTarget = -0.15;
51              }
52              else{
53                  mDotPsiTarget = 0.15;
54              }
55          }
56          break;
57
58          case EMode::Corner:
59              pStateData.mDotPsiTarget = mDotPsiTarget;
60              dotPhiTarget =
                ↪  mRollTargeYawController.control(pStateData.mDotPsiTarget,
                ↪  pStateData.mDotPsi);
61              pStateData.mPhiTarget += dotPhiTarget*mTa;
62              pStateData.mSpeedTarget = ramp(mTargetVelocity,
                ↪  pStateData.mSpeedTarget, Monowheeler::MAX_ACCEL);
63              pStateData.mThetaTarget = 0;
64
65              mCounter++;
66              if (mCounter > 500) {
67                  if (mSlalom) {
68                      mSlalom = false;
69                      mDotPsiTarget = 0.15;
70                  }
71                  else {
72                      mMode = EMode::Balance;
```

```
73                      mDotPsiTarget = 0;
74                      mTargetVelocity = 0;
75                  }
76                  mCounter = 0;
77              }
78          break;
79
80      default:
81          break;
82      }
83
84      pMotorData.mWheelDynamixelTarget = mPitchController.control(pStateData.mTheta,
        ↪  pStateData.mDotTheta, pMotorData.mWheelDynamixelPosition);
85
86      pMotorData.mGyroDynamixelTarget =
        ↪  mRollController.control(pStateData.mPhiTarget, dotPhiTarget,
        ↪  pStateData.mPhi, pStateData.mDotPhi);
87
88      pMotorData.mWheelTorqueTarget =
        ↪  mSpeedController.control(pStateData.mSpeedTarget, pStateData.mSpeed);
89  }
```