



Fakultät für Maschinenbau und Mechatronik
Studiengang Mechatronik

Balancieren eines einrädrigen Fahrzeugs mit Hilfe des Gyroskopischen Effekts

Balancing a One-Wheeled Vehicle Using the Gyroscopic Effect

Bachelorarbeit (B.Eng.)

von
Ruben Egle
geb. am 09.11.2001
Matr.-Nr.: 79733

Betreuer der Hochschule Karlsruhe
Prof. Dr.-Ing. Joachim Wietzke

Korreferent der Hochschule Karlsruhe
Prof. Dr.-Ing. Tobias Baas

Bearbeitungszeitraum
22.05.2025 bis 20.10.2025

Bachelor-Thesis:

Ruben Egle
Mechatronik

Arbeitsplatz:

Hochschule Karlsruhe
Fakultät für Maschinenbau und Mechatronik
76133 Karlsruhe

Betreuer am Arbeitsplatz:

Prof. Dr. Joachim Wietzke

Betreuerender 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 Mikrocontroller
- 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

Erklärung

Ich versichere hiermit wahrheitsgemäß, die Abschlussarbeit selbstständig angefertigt, alle benutzten Hilfsmittel vollständig und genau angegeben und alles einzeln kenntlich gemacht zu haben, was aus Arbeiten anderer unverändert oder mit Abänderungen entnommen wurde.

Karlsruhe, den 16. Oktober 2025

Unterschrift: R.Egle

Danksagung

An dieser Stelle möchte ich mich bei allen bedanken, die mich bei der Erstellung dieser Bachelorarbeit begleitet und unterstützt haben. Als Erstes gebührt der Dank Herrn Prof. Dr. Joachim Wietzke, der diese Arbeit betreut hat. Er begleitete das Projekt von Beginn an mit herausragender fachlicher und organisatorischer Unterstützung.

Ein besonderer Dank gilt Prof. Dr.-Ing. Tobias Baas für die Betreuung der Arbeit als Korreferent, die Bereitstellung der Laborräume und Ressourcen des EML, sowie die fachliche Expertise, die wesentlich zum Erfolg dieser Arbeit beigetragen hat.

Ebenso möchte ich mich bei Herrn Prof. Dr.-Ing. Ansgar Blessing für die wertvollen fachlichen Impulse und Beiträge bedanken.

Abschließend möchte ich meiner Familie und meiner Freundin für ihre mentale Unterstützung, ihre Hilfe beim Korrekturlesen sowie bei der Beschaffung von Literatur danken. Ohne ihre Rückendeckung wäre diese Arbeit in dieser Form nicht möglich gewesen.

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.

Nomenklatur

Formelzeichen

F_G	Gewichtskraft des Fahrzeugs	N	8, 10, 11, 56, 57, 58, 74, 77, 85, 86
F_H	Haftkraft im Aufstandspunkt des Fahrzeugs	N	8, 10, 11
F_N	Normalkraft im Aufstandspunkt des Fahrzeugs	N	8, 10
F_{GP}	Gewichtskraft der Plattform des Fahrzeugs	N	8, 56, 57
F_{ZF}	Zentrifugalkraft durch die Kurvenfahrt des Fahrzeugs	N	10, 11, 12
J_K	Massenträgheitsmoment des Kreisels	kg m^2	10, 11, 74, 77, 85, 86
J_z	Massenträgheitsmoment um die z-Achse	kg m^2	11, 74, 85, 86
$J_{x,\tau}$	Massenträgheitsmoment um die x-Achse bezogen auf den Aufstandspunkt	kg m^2	10, 11, 74, 77, 85, 86
$J_{y,\tau}$	Massenträgheitsmoment um die y-Achse bezogen auf den Aufstandspunkt	kg m^2	8, 56, 57, 58
$M_{K\psi}$	Durch Gyroskop erzeugtes Moment um z-Achse	N m	11, 12
$M_{K\varphi}$	Durch Gyroskop erzeugtes Moment um x-Achse	N m	10, 12
S_G	Schwerpunkt des Fahrzeugs		8, 10, 11
S_P	Schwerpunkt der Plattform des Fahrzeugs		8
T_a	Abtastzeit	s	33, 34
$\dot{\psi}_K$	Winkelgeschwindigkeit des Gyroskops um seine Hochachse	rad	10, 11, 16, 74, 75, 76, 77, 85
ψ	Gieren – Rotation um z-Achse gemäß DIN 70000	rad	10, 11, 12, 74, 85, 86, 93
ψ_K	Winkel des Gyroskops um seine Hochachse	rad	11, 74, 75, 76, 86
φ	Rollen – Rotation um x-Achse gemäß DIN 70000	rad	10, 11, 12, 32, 33, 34, 46, 71, 74, 75, 76, 85, 86, 93
ϑ	Nicken – Rotation um y-Achse gemäß DIN 70000	rad	8, 32, 33, 34, 46, 50, 51, 56, 57, 59, 63
g	Gravitationskonstante	m s^{-2}	32, 33, 34, 37

h_0	Höhe des Schwerpunkts S_G in Ruhelage	m	8, 10, 11, 56, 57, 58, 74, 77, 85, 86
m_{ges}	Gesamtmasse des Fahrzeugs	kg	11, 85, 86
p	Verschiebung des Aufstandspunkts relativ zum Fahrzeug	m	8, 14, 56, 59
r_{Kurve}	Momentaner Kurvenradius	m	11
v	Geschwindigkeit in x-Richtung des Fahrzeugs	m s^{-1}	11, 12, 85, 86, 88, 94
w_K	Konstante Winkelgeschwindigkeit des Gyroskops	rad	10, 11, 20, 74, 77, 85, 86

Abkürzungen

ADC	Analog to Digital Converter	21, 36, 82, 120, 121
BBB	BeagleBone Black	23, 24, 25, 29, 95, 96, 97, 100, 101, 112, 114, 115, 116, 119, 120, 122, 124, 127, 128, 143
BLDC Motor	Brushless DC Motor	13
CEMF	Counter Electromotive Force	20
CMG	Control Moment Gyroscope	3
CSV	Comma Separated Values	36, 95, 128
DLP	Digital Lowpass	40, 41, 42, 43, 44, 45, 46, 139, 140, 141, 7, 8, 9, 10, 11, 12
ELKO	Elektrolytkondensator	26
EML	Embedded Mechatronics Laboratory	2, 95, 100, 115, 119, 120, 122
EMV	Elektromagnetische Verträglichkeit	25, 131
FFT	Fast Fourier Transformation	36, 39, 40, 42, 44, 45, 139, 140, 141, 8, 10, 12
FIFO	First In First Out	120, 121
GPIO	General Purpose Input Output	24, 95, 97, 120

HKA	Hochschule Karlsruhe für Technik und Wirtschaft	2
IDE	Integrated Development Environment	96
IIR	Infinite Impulse Response	105, 106, 145
IMU	Inertial Measurement Unit	24, 31, 32, 33, 34, 35, 36, 38, 39, 40, 41, 42, 43, 44, 45, 46, 50, 115, 117, 118, 128, 129, 131, 139, 140, 141, 143, 7, 8, 9, 10, 11, 12
LiPo	Lithium-Polymer-Akkumulator	25, 26, 27
LQR	Linear Quadratic Regulator	55, 56, 58, 66, 73, 76, 79, 85, 86, 131, 132, 145
mDNS	Multicast DNS	96, 124, 127
MIMO	Multiple Input Multiple Output	66
MMAP	Memory Map	112, 115, 119, 120
MPC	Model Predictive Control	4, 55, 134
PWM	Pulse-Width Modulation	20, 21, 119, 120
RMS	Root Mean Square	35, 36
SCP	Secure Copy	96
SIMO	Single Input Multiple Output	66, 70, 71, 73, 80, 85
SISO	Single Input Single Output	66
SNR	Signal to Noise Ratio	34, 35, 36, 37, 41, 42, 43, 45, 46, 131, 143, 145
SOS	Second-Order-Section	105, 106, 145
SPI	Serial Peripheral Interface	25, 28, 95, 97, 114, 115, 116, 118, 140
SSH	Secure Shell	95, 96, 97
ToF	Time of Flight	31
TTL	Transistor-Transistor-Logik	28, 139

UART	Universal Asynchronous Receiver/Transmitter	23, 25, 95, 97, 114, 122
UDP	User Datagram Protocol	95, 100, 123, 124, 126, 127, 128
ufw	Uncomplicated Firewall	97

Inhaltsverzeichnis

Nomenklatur	xi
1 Einleitung	1
2 Stand der Technik	3
2.1 Gyroskopischer Effekt	3
2.2 Einrädrige Fahrzeuge	4
2.3 Einrädrige Fahrzeuge mit dem gyroskopischen Effekt	4
3 Systembeschreibung	7
3.1 Teilsystem Nicken	7
3.2 Teilsystem Rollen und Gieren	8
3.2.1 Gyroskopischer Effekt	9
3.2.2 Anwendung des gyroskopischen Effekts	10
3.2.3 Interpretation und reales System	11
4 Hardwareintegration	13
4.1 Aktorik	13
4.1.1 Dynamixel Servomotoren	13
4.1.2 Kreiselmotor	20
4.1.3 Antriebsrad	21
4.2 Elektronik	23
4.2.1 Übersicht Komponenten	23
4.2.2 Funktionsstruktur und Designrichtlinien	24
4.2.3 Leistungselektronik	26
4.2.4 Signalelektronik	27
4.2.5 Layout	28
4.2.6 Ergebnisse	29
5 Sensorkonzept und Signalverarbeitung	31
5.1 Sensorfusion	32
5.2 Signalcharakterisierung und Frequenzanalyse	34
5.2.1 Kalibrierung der Sensoren	35
5.2.2 Experimentelle Signalanalyse	35
5.2.3 Signalverbesserung durch Filterung auf Imu-Hardware	40
5.2.4 Signalverbesserung durch externen Tiefpassfilter	43
5.2.5 Übersicht Signalqualität	45
5.2.6 Validierung Sensorkonzept und Signalverarbeitung	46
6 Iterative Reglerentwicklung zum Balancieren	49
6.1 Abtastfrequenz	49

6.2 Teilsystem Nicken	50
6.2.1 PID-Regler	51
6.2.2 Zustandsregler	55
6.3 Teilsystem Rollen und Gieren	65
6.3.1 PID-Regler	66
6.3.2 Zustandsregler	73
6.4 Gesamtsystem Ergebnis	80
7 Regelung und Trajektorien zum Kurvenfahren	85
7.1 Direkte Regelungskonzepte	85
7.2 Trajektorienbasierte Steuerung	86
7.2.1 Passive Trajektorie	87
7.2.2 Aktive Trajektorie	88
7.3 Ergebnisse Kurvenfahren	94
8 Hardwareintegration und Regelkreis auf Mikrocontroller	95
8.1 Entwicklungsumgebung Entwicklungsrechner	96
8.2 Entwicklungsumgebung BeagleBone Black	97
8.3 Targetanwendung	97
8.3.1 Softwarearchitektur	99
8.3.2 Control-Comp	101
8.3.3 Comm-Comp	123
8.4 Python-Gui	126
8.4.1 Hauptprogramm	126
8.4.2 UDP-Klient	127
8.4.3 Datenextraktion	128
8.4.4 Datenaufzeichnung	128
8.4.5 Plot-Manager	129
9 Zusammenfassung	131
10 Ausblick	133
Literaturverzeichnis	135
Abbildungsverzeichnis	139
Tabellenverzeichnis	142
Quelltextverzeichnis	144
A Schaltplan	1
B Signalanalyse	7
C Code	13
C.1 CPitchController	13
C.2 CPIDController	14
C.3 CController	15

1 Einleitung

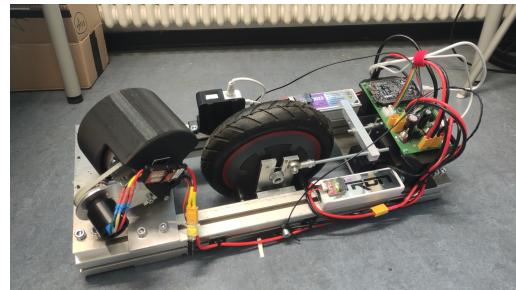
Mit der zunehmenden Autonomisierung und Miniaturisierung mobiler Systeme gewinnt die Regelung instabiler, dynamischer und eng gekoppelter Systeme in der Robotik zunehmend an Bedeutung. Instabile Systeme finden sich in zahlreichen technischen Anwendungen und können durch viele verschiedene Herangehensweise stabilisiert und gesteuert werden. Besonders komplex sind Systeme mit mehreren Freiheitsgraden wie das inverse 3D-Pendel. Die Herausforderung der Regelung solcher Systeme gewinnt eine zusätzliche, praktische Dimension, wenn sie nicht nur stationär, sondern mobil eingesetzt werden. Einrädrige Fahrzeuge müssen zum Balancieren die zwei Freiheitsgrade Rollen und Nicken aktiv, und den Freiheitsgrad Gieren passiv kontrollieren. Zur freien Bewegung in der Ebene muss auch das Gieren aktiv kontrolliert werden können, wobei die Freiheitsgrade oft hochgradig miteinander gekoppelt sind. Meistens handelt es sich bei solchen Systemen um unteraktuierte Systeme, sodass die Kopplung der Freiheitsgrade gezielt genutzt werden muss, um das gewünschte Ziel zu erfüllen. Dies stellt besonders hohe Anforderungen an die Regelung und das Systemdesign solcher Fahrzeuge.

Während viele mobile Robotersysteme von Natur aus statisch stabil sind und komplexe dynamische Manöver ausführen können, benötigen sie keine aktive Stabilisierung. Umgekehrt konzentrieren sich klassische Balanciersysteme meist auf die aktive Stabilisierung eines instabilen Zustands, sind jedoch in ihrer Mobilität stark eingeschränkt. Obwohl es zahlreiche Konzepte und Demonstratoren für selbstbalancierende Fahrzeuge gibt, existiert bislang nur ein elektronisch stabilisiertes Fahrzeug dieser Größenordnung, das eine vollständig freie Bewegung in der Ebene mit nur einem Rad realisieren kann. Dieses Fahrzeug verwendet mit der Kombination aus einer Schwungmasse und einem Antriebsrad verbreitete Konzepte zur Stabilisierung und Steuerung instabiler Systeme. Das Ziel dieser Arbeit ist es, ein einrädriges Fahrzeug, den sog. *Monowheeler*, mit dem gyroskopischen Effekt zu balancieren und dessen Fähigkeiten zum Kurvenfahren zu untersuchen. Damit wird ein neuartiges Konzept zur Stabilisierung und Steuerung eines unteraktuierten, instabilen Systems mit hochgradig gekoppelter Dynamik entwickelt, erprobt und vorgestellt.

Der *Monowheeler* ist ein einrädriges Fahrzeug, dessen Antriebsrad an einer Plattform aufgehängt ist. Das Fahrzeug ist inspiriert von dem *One-Wheeled Vehicle* von Charles Taylor [36], der sein Einrad in der Größenordnung eines Autos von Hand stabilisiert. Abbildung 1.1b und Abbildung 1.1a zeigen den Monowheeler und das historische Vorbild von Charles Taylor.



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



(b) Monowheeler

Zur Kontrolle der instabilen Nickdynamik des *Monowheelers* kann das Rad relativ zur Plattform verschoben werden. Die ebenfalls instabile Rollbewegung wird durch ein Gyroskop (Kreisel) im vorderen Bereich des Fahrzeugs gesteuert. Das Konzept und die Konstruktion des *Monowheelers* wurden in einem vorhergehenden Forschungs- und Entwicklungsprojekt [33] an der Hochschule Karlsruhe für Technik und Wirtschaft (HKA) im Rahmen des Embedded Mechatronics Laboratory (EML) erarbeitet. In der vorliegenden Arbeit soll die bestehende mechanische Plattform um ein geeignetes Sensorsystem erweitert werden und darauf aufbauend eine Regelung zur Stabilisierung und Steuerung des Fahrzeugs entwickelt und implementiert werden.

Die Arbeit ist wie folgt aufgebaut: Kapitel 2 fasst den aktuellen Stand der Technik und des vorhergehenden Projekts zusammen. Kapitel 3 beschreibt und interpretiert das mathematische Modell des *Monowheelers*. Anschließend wird in Kapitel 4 die bestehende Hardware in Betrieb genommen und vermessen, um die dynamischen Eigenschaften der Aktorik in die Modellierung aufzunehmen. Zur Erfassung der Fahrzeugdaten wird in Kapitel 5 ein geeignetes Sensorkonzept erarbeitet. In Kapitel 6 wird die Entwicklung eines geeigneten Konzepts zur Stabilisierung des Fahrzeugs dokumentiert. Die Fähigkeit des Fahrzeugs, dynamische Manöver wie das Kurvenfahren durchzuführen, wird in Kapitel 7 untersucht. Zum Schluss wird in Kapitel 8 die Integration der Hardware und des Regelkreises auf dem verwendeten Mikrocontroller beschrieben.

2 Stand der Technik

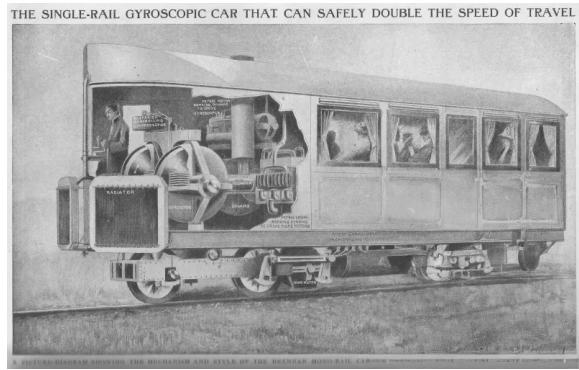
Die Kontrolle einrädriger Fahrzeuge kann auf viele Arten erfolgen. Dabei ist sowohl zwischen der Anzahl der Freiheitsgrade sowie zwischen den Konzepten zur Kontrolle dieser Freiheitsgrade zu differenzieren. Das folgende Kapitel beschreibt den aktuellen Stand der Technik und gibt einen Überblick über theoretische Konzepte und praktische Umsetzungen einrädriger Fahrzeuge. Ein besonderes Augenmerk liegt dabei auf der Verwendung des gyroskopischen Effekts, da dieser einen zentralen Bestandteil des in dieser Arbeit verfolgten Konzepts darstellt.

2.1 Gyrokopischer Effekt

Der gyrokopische Effekt beschreibt das Verhalten eines Kreisels, dessen Achsausrichtung verändert wird. Dabei entsteht ein Drehmoment senkrecht zu der Achse, um die die Veränderung erfolgt. Dieser Effekt wird in verschiedenen technischen Anwendungen eingesetzt, um Systeme aktiv und passiv zu stabilisieren. Ein Beispiel ist das *Monorail* von Louis Brennan aus dem Jahr 1905, bei dem ein Schienenfahrzeug auf einer Schiene mit Hilfe zweier Gyroskope balanciert [5]. Einen ähnlichen Ansatz verfolgt das *Gyrocar* von Pyotr Shilovsky [35], und dessen Weiterentwicklung *Gyro-X*. Keines der Konzepte konnte sich jedoch durchsetzen.



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



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

In der Raumfahrt wird der gyrokopische Effekt in Form von Control Moment Gyroscopes (CMGs) erfolgreich verwendet, um die Lage von Satelliten und Weltraumstationen im Raum zu stabilisieren. Das prominenteste System dieser Art kommt auf der ISS zum Einsatz und hilft dabei, energiesparend die Lage der Raumstation zu kontrollieren [17]. Aber auch kleinere Satelliten nutzen den Effekt, um ihre Lage und Flughöhe zu beeinflussen [21][18].

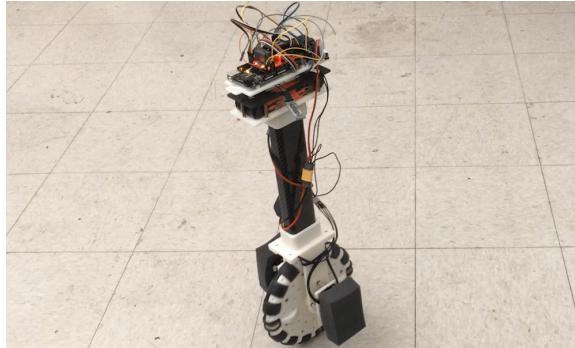
In der Schifffahrt wird der gyrokopische Effekt zur passiven Stabilisierung der Lage von Schiffen gegen Wind und Wellengang genutzt. Dabei wird ein großer Kreisel in einem Gimbal

aufgehängt, sodass die natürliche Neigung eines Kreisels, seine Orientierung beizubehalten, das Schiff stabilisiert und unerwünschte Rollbewegungen minimiert [30].

2.2 Einrädrige Fahrzeuge

In den letzten Jahren wurden viele Konzepte zur Stabilisierung einrädriger Fahrzeuge umgesetzt. Meistens werden hierfür Schwungräder, die aktive Schwerpunktverlagerung durch Antriebs- und Bremskräfte oder sog. Omniwheels verwendet. Omniwheels bestehen aus einem großen Rad, dessen Laufflächen orthogonal zur Ausrichtung des Rads aktuierbar sind. Damit kann solch ein Einrad eigenständig balancieren und sich translatorisch frei bewegen. Allerdings kann die Ausrichtung des Fahrzeugs nicht beeinflusst werden. Ein Beispiel für dieses Konzept wurde mit dem *OmBuro* in [34] vorgestellt.

Das fortgeschrittenste Einrad dieser Art ist der *Mini Wheelbot*, der Anfang 2025 vorgestellt wurde. Der Roboter besteht aus zwei Schwungrädern, die je nach Orientierung als Antriebsrad und als Schwungrad verwendet werden. Mithilfe eines nicht linearen Model Predictive Control (MPC)-Algorithmus ist der *Wheelbot* in der Lage zu balancieren und Kurven zu fahren. Damit kontrolliert ein einrädriger Roboter dieser Art erstmalig alle Freiheitsgrade vollständig [14].



(a) *OmBuro* von Junjie Shen und Dennis Hong [34]



(b) *Mini Wheelbot* [27]

2.3 Einrädrige Fahrzeuge mit dem gyroskopischen Effekt

Die Analyse bestehender Konzepte zeigt, dass der gyroskopische Effekt als primäres Mittel zur Stabilisierung und Steuerung von Einrädern nur selten Anwendung findet. Stattdessen wird dieses Prinzip vorwiegend in anderen Fahrzeugklassen genutzt, oder es werden gänzlich andere Konzepte verwendet.

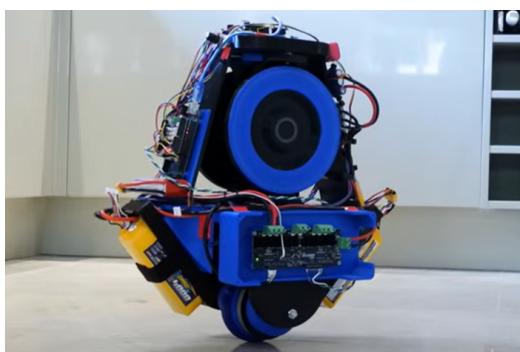


Abbildung 2.3: Einrad mit Gyroskopen von James Bruton [19]

Eine der wenigen Anwendungen findet der gyroskopische Effekt im Einrad von James Bruton, der zwei Kreisel nutzt, um die Rollbewegung des Roboters zu kontrollieren. Dabei wird aber gezielt auf ein Konzept gesetzt, das die Kopplung zwischen Rollen und Gieren durch den Kreisel aufhebt. Die Kreisel drehen sich in entgegengesetzte Richtung, sodass sich ihre Momente durch Bewegungen des Fahrzeugs gegenseitig aufheben. In Fahrtrichtung balanciert das Fahrzeug durch Antriebs- und Bremsmomente. Das Einrad hat somit keine Kontrollautorität über die Fahrtrichtung [19].

Als Grundlage für das in dieser Arbeit entwickelte Fahrzeug dient das *One-Wheeled Vehicle* von Charles F. Taylor.

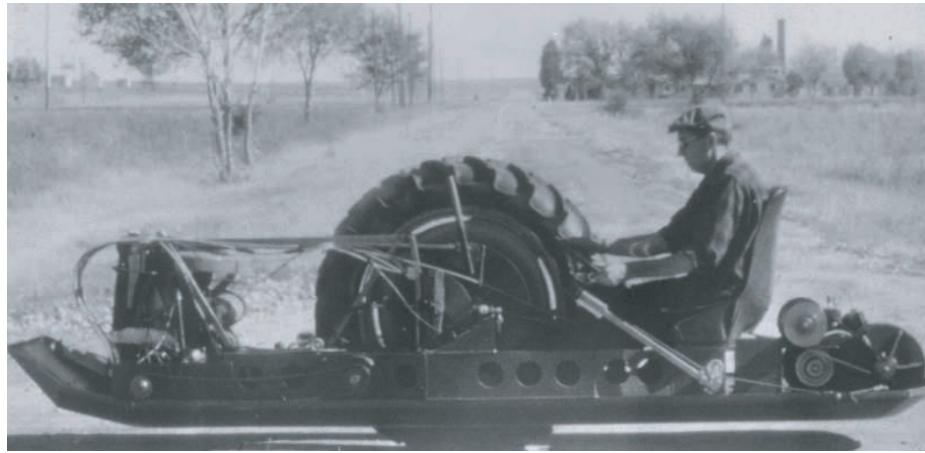


Abbildung 2.4: *One-Wheeled Vehicle* von Charles F. Taylor [22]

Das Fahrzeug besteht aus einer Plattform, in dessen Mitte sich ein großes Antriebsrad befindet. Für die Balance in Fahrtrichtung wird ein Mechanismus eingesetzt, der das Rad gegenüber der Plattform verschiebt. Die Rollbewegungen und das Kurvenfahren werden primär durch ein großes Gyroskop im vorderen Teil der Plattform stabilisiert und gesteuert. Die Steuerung erfolgt dabei rein mechanisch und durch menschliches Eingreifen. Auch dieses Fahrzeug kann alle Freiheitsgrade kontrollieren. Allerdings zeigt Taylors Fahrzeug auch die Einschränkungen des Konzepts auf. Die Stabilisierung durch ein großes Gyroskop funktioniert vor allem bei gleichförmiger Fahrt, verliert jedoch bei niedrigen Geschwindigkeiten an Wirksamkeit, da das Fahrzeug auf die zusätzliche, dynamische Kopplung zwischen der seitlichen Neigung und dem Kurvenfahren angewiesen ist. Enge Kurven und dynamische Manöver sind durch die physikalisch bedingte hohe Trägheit des Systems sowie der begrenzten Kontrollautorität über die Fahrtrichtung nur eingeschränkt möglich.

Die Grundlage für den hier verfolgten Ansatz bildet [33], in dem Taylors Patent [36]

analysiert und ein Prototyp in Modellgröße mit modernen Komponenten entworfen wurde. Während diese Vorarbeit erste Regelungskonzepte für das Balancieren des Fahrzeugs skizzierte, besteht das Hauptziel dieser Arbeit in der vollständigen Inbetriebnahme des Prototyps sowie in der detaillierten Ausarbeitung, Umsetzung und Neuentwicklung der Regelungsstrategien. Zudem soll die Fähigkeit des Fahrzeugs, Kurven zu fahren, untersucht werden.

3 Systembeschreibung

Um ein einrädriges Fahrzeug zu stabilisieren, müssen im Wesentlichen die beiden Freiheitsgrade Nicken und Rollen kontrolliert werden. Für die freie Bewegung in der Ebene ist zusätzlich die Kontrolle über den Freiheitsgrad Gieren erforderlich:

- **Nicken (Pitch):** Längsneigung, beschreibt die Vorwärts-Rückwärts-Balance
- **Rollen (Roll):** Querneigung, beschreibt die Links-Rechts-Balance
- **Gieren (Yaw):** Drehung um die Hochachse, bestimmt die Fahrtrichtung

Da das primäre Ziel dieser Arbeit die Stabilisierung des *Monowheelers* ist, wird für die Analyse und Reglerauslegung bewusst ein vereinfachtes Modell verwendet, das die Dynamik auf die Bewegung eines 3D-inversen Pendels reduziert. Dieses Modell ermöglicht es, die wesentlichen Freiheitsgrade Nicken, Rollen und Gieren getrennt zu betrachten und zu kontrollieren. Mit dieser Vereinfachung kann das Gesamtsystem aufgrund der Orthogonalität der Bewegungsrichtungen sowie der Entkopplung der wirkenden Kräfte und Momente in Längs- und Querebene in zwei voneinander unabhängige Teilsysteme unterteilt werden: Das erste beschreibt das Nicken, das zweite das Rollen und Gieren. Das vereinfachte Pendelmodell eignet sich besonders gut, um das Balancieren zu analysieren und Regelstrategien zu entwickeln, da die Systemdynamik überschaubar bleibt und direkt interpretierbar ist. Gleichzeitig ist zu beachten, dass dieses Modell nicht die vollständige Fahrzeugdynamik abbildet. Insbesondere beim Kurvenfahren treten zusätzliche Kopplungen zwischen den Freiheitsgraden auf, die hier nicht berücksichtigt werden. Die Dynamik des Balancierens wird jedoch gut abgebildet und soll im Verlauf der Arbeit mit dem realen System validiert werden. Die detaillierte Beschreibung und Interpretation der beiden Teilsysteme des *Monowheelers* erfolgt in [33].

3.1 Teilsystem Nicken

Das Teilsystem zur Modellierung des Nickens kann mit den eben beschriebenen Vereinfachungen auf ein 2D-inverses Pendel reduziert werden. Das Fahrzeug wird dabei als Starkörper angenommen. Um das System in der instabilen Ruhelage zu stabilisieren, kann der Aufstandspunkt relativ zur Plattform des Fahrzeugs verschoben werden. Dabei wird angenommen, dass die Dynamik des Verschiebevorgangs (Motormoment, Trägheit etc.) sowie die resultierenden geometrischen Änderungen (Änderung des Massenträgheitsmoments, vertikale Verschiebung des Schwerpunkts durch Mechanik etc.) vernachlässigbar sind [33, vgl. S. 19f]. Die Bewegungssituation kann wie in Abbildung 3.1 zu sehen dargestellt werden:

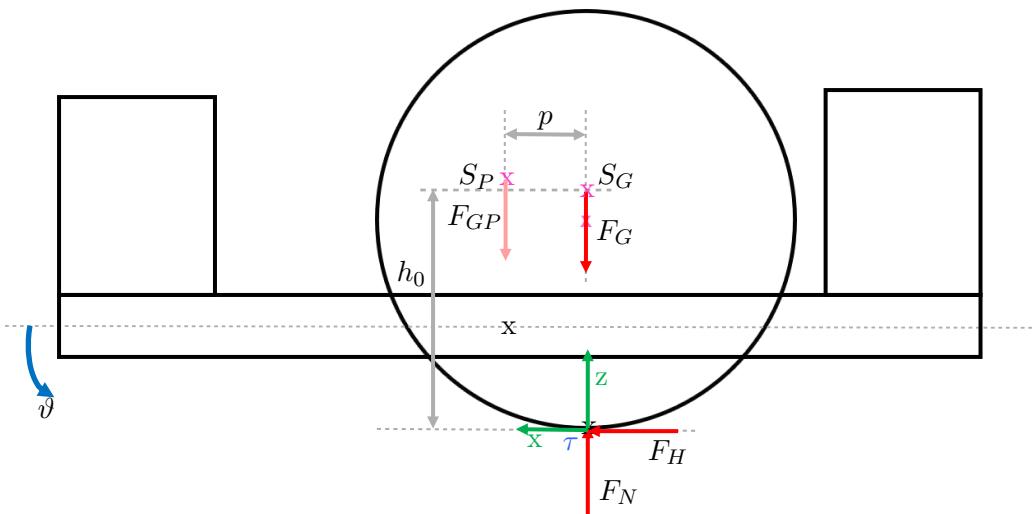


Abbildung 3.1: Freischnitt Nicken [33, S.21]

Durch das Momentengleichgewicht um den Momentanpol τ ergibt sich die vereinfachte Differenzialgleichung [33, S. 21]:

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

Der Nickwinkel ϑ kann also direkt über die lineare Verschiebung des Aufstandspunkts relativ zur Plattform beeinflusst werden. Dadurch muss weder die Plattform geneigt werden, noch das Fahrzeug beschleunigt werden, um Störungen auszugleichen. Außerdem können dauerhafte Störungen problemlos ausgeglichen werden, im Gegensatz zu anderen Konzepten wie beim Balancieren mit einer Schwungscheibe.

3.2 Teilsystem Rollen und Gieren

Der Fokus beim Stabilisieren des Fahrzeugs liegt auf der Kontrolle der Rollbewegung. Ohne aktive Gegensteuerung kippt das Fahrzeug seitlich um. Im Gegensatz dazu lässt sich das Gieren als indifferentes Gleichgewicht beschreiben. Das Gieren bestimmt zwar die Fahrtrichtung, diese ist aber nicht maßgeblich für die Stabilität des Fahrzeugs entscheidend und bleibt ohne äußere Störung konstant. Das Gieren wird daher nur sekundär kontrolliert. Zur Stabilisierung und Steuerung der Roll- und Gierbewegung soll laut Aufgabenstellung der gyroskopische Effekt verwendet werden.

Das Teilsystem zur Modellierung des Rollens und Gierens kann nicht auf ein 2D-Problem reduziert werden, da die beiden Bewegungen aufgrund des gyroskopischen Effekts und der dynamischen Kopplung zwischen Rollen und Gieren beim Kurvenfahren miteinander gekoppelt sind. Es wird weiterhin mit dem vereinfachten Pendelmodell gearbeitet, bei dem der *Monowheeler* als Starrkörper betrachtet wird. Damit kann die Rollbewegung, ähnlich der Nickbewegung, vereinfacht als 2D-inverses Pendel mit instabiler Ruhelage betrachtet werden. Die direkten Einflüsse des Gyroskops, das zur Regelung verwendet wird, sowie die auf einen Starrkörper wirkenden Kräfte beim Kurvenfahren werden durch die Betrachtung des Gierens mit einbezogen. Allerdings handelt es sich weiterhin um ein vereinfachtes Modell,

da weitere Effekte beim Kurvenfahren nicht berücksichtigt werden. So führt beispielsweise das seitliche Kippen eines Rads dazu, dass das Rad aufgrund des Rolling-Constraints einer gekrümmten Bahn folgt [43][4, S. 16ff]. Des Weiteren entstehen dabei zusätzliche Momente durch den gyroskopischen Effekts des Rads. Die zum Balancieren relevante Dynamik wird aber auch hier abgebildet und soll mithilfe des realen Systems validiert werden.

3.2.1 Gyroskopischer Effekt

Die Grundlagen des gyroskopischen Effekts werden in [33, S. 11ff] ausführlich beschrieben. Wird ein rotierender, rotationssymmetrischer Kreisel mit dem Massenträgheitsmoment J_K und der Winkelgeschwindigkeit $\vec{\omega}_K$ zusätzlich mit einer Winkelgeschwindigkeit $\vec{\omega}_R$ um eine weitere Achse gedreht, entsteht ein Drehmoment um die zu beiden Winkelgeschwindigkeiten senkrechte Achse:

$$\vec{M} = J \cdot \vec{\omega}_K \times \vec{\omega}_R \quad (3.2)$$

Durch dieses Prinzip haben rotierende Kreisel eine selbststabilisierende Eigenschaft.

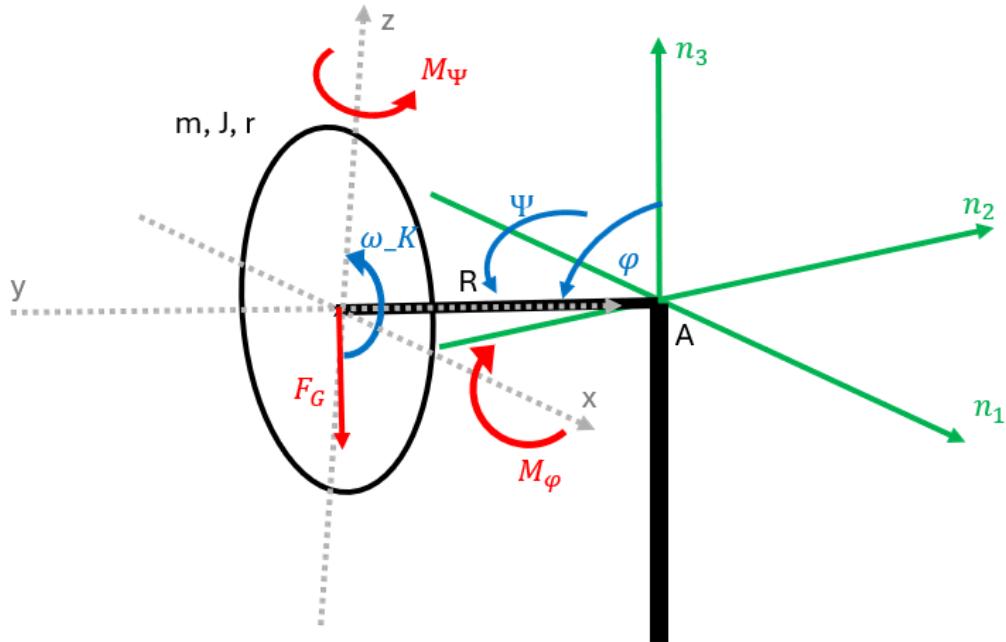


Abbildung 3.2: Freischnitt Kreisel als inverses Pendel [33, S.16]

Wird ein Kreisel auf einem Aufbau montiert, der einem inversen Pendel ähnelt, beginnt er unter dem Einfluss der Schwerkraft seitlich mit der Winkelgeschwindigkeit $\dot{\varphi}$ zu kippen. Das entspricht einer zusätzlichen Drehung des Kreisels, und erzeugt Gleichung 3.2 zufolge ein Moment M_ψ , welches den Kreisel um seine Hochachse beschleunigt. Das führt dann wiederum zu einem weiteren Moment M_φ , welches der ursprünglichen Kippung $\dot{\varphi}$ entgegenwirkt und diese bei geeigneter Parameterwahl sogar umkehrt. In diesem Fall fällt der Kreisel nicht um, sondern führt eine Kombination aus Präzession (Drehung um die Hochachse) und Nutation (periodische Kippbewegung) aus [33, S. 16].

3.2.2 Anwendung des gyroskopischen Effekts

Der gyroskopische Effekt wird auf den *Monowheeler* angewendet, um die Roll- und Gierbewegung zu kontrollieren und stabilisieren. Um die Rollbewegung auch aktiv beeinflussen zu können, wird der Kreisel so aktuiert, dass durch die gezielte Drehung des Kreisels ein Moment erzeugt wird, welches direkt auf die Rollbewegung wirkt. Damit können die gekoppelten Differenzialgleichungen für das vereinfachte Teilsystem Rollen und Nicken aufgestellt werden. Vorerst wird das System als reibungsfrei angenommen. Abbildung 3.3 zeigt den Freischnitt der Rollbewegung mit Einflüssen der Gierbewegung:

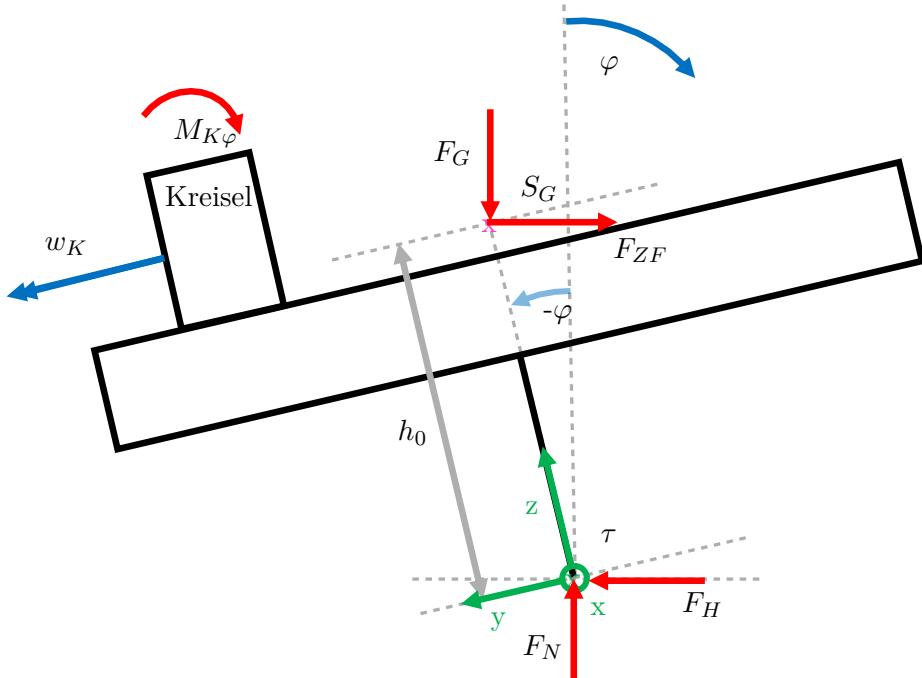


Abbildung 3.3: Freischnitt Rollen [33, S.28]

Die translatorischen Kräfte spielen eine untergeordnete Rolle, da die translatorische Bewegung des Fahrzeugs anhand der Geschwindigkeit und Fahrtrichtung berechnet werden kann. Daher wird das Momentengleichgewicht um den Momentanpol gebildet:

$$J_{x,\tau} \cdot \ddot{\varphi} = M_{K\varphi} + F_G \cdot h_0 \cdot \sin \varphi + F_{ZF} \cdot h_0 \cdot \cos \varphi \quad (3.3)$$

Das Kreiselmoment $M_{K\varphi}$ kann nach Gleichung 3.2 beschreiben werden:

$$M_{K\varphi} = J_K \cdot w_K \cdot (\dot{\psi} + \dot{\psi}_K) \quad (3.4)$$

Abbildung 3.4 zeigt den Freischnitt der Gierbewegung mit Einflüssen der Rollbewegung. Dabei ist auch zu erkennen, dass die Position des Kreisels auf dem Fahrzeug keinen Einfluss auf die Bewegungsgleichungen hat und somit in der Konstruktion frei gewählt werden kann.

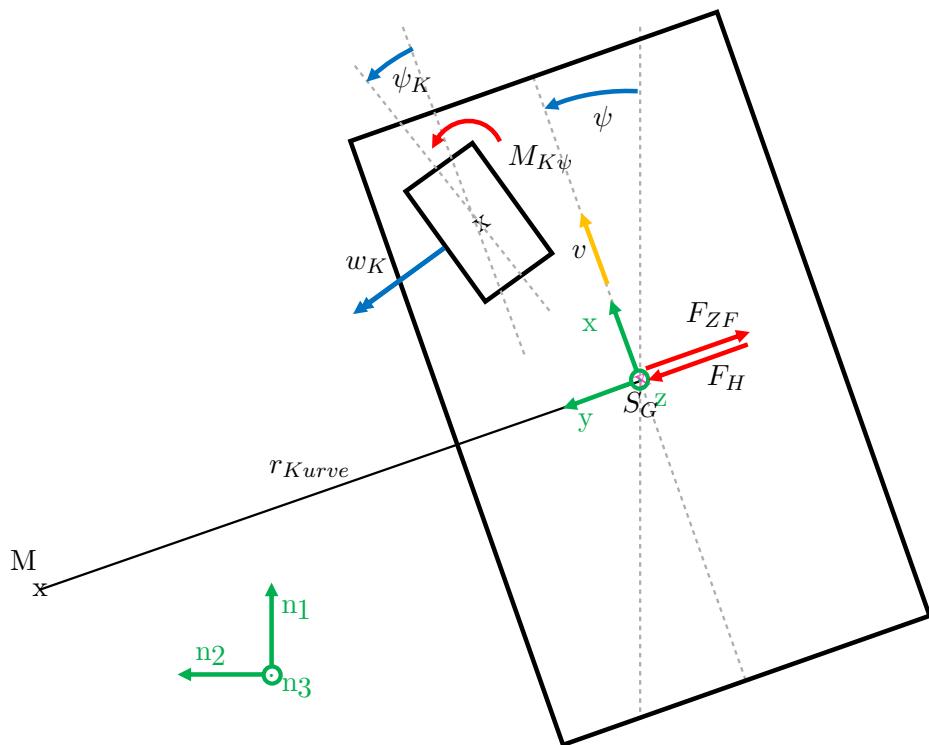


Abbildung 3.4: Freischnitt Gieren [33, S.28]

Das Momentengleichgewicht für die Gierbewegung ergibt folgende Differenzialgleichung:

$$J_z \cdot \ddot{\psi} = M_{K\psi} \quad (3.5)$$

Das Kreiselmoment $M_{K\psi}$ kann nach Gleichung 3.2 beschreiben werden:

$$M_{K\psi} = -J_K \cdot w_K \cdot \dot{\varphi} \quad (3.6)$$

Die Trägheitskraft F_{ZF} , auch Zentrifugalkraft genannt, die als Reaktion auf die Zentripetalbeschleunigung beim Kurvenfahren entsteht, kann wie folgt beschrieben werden:

$$F_{ZF} = m_{ges} \cdot r_{Kurve} \cdot \dot{\psi}^2 = m_{ges} \cdot \frac{v}{\dot{\psi} \cdot \cos \varphi} \cdot \dot{\psi}^2 \quad (3.7)$$

Daraus ergeben sich die folgenden, gekoppelten Differenzialgleichungen [33, S. 28 ff]:

$$J_{x,\tau} \cdot \ddot{\varphi} = J_K \cdot w_K \cdot (\dot{\psi} + \dot{\psi}_K) + F_G \cdot h_0 \cdot \sin \varphi + m_{ges} \cdot v \cdot \dot{\psi} \cdot h_0 \quad (3.8)$$

$$J_z \cdot \ddot{\psi} = -J_K \cdot w_K \cdot \dot{\varphi} \quad (3.9)$$

3.2.3 Interpretation und reales System

Das zuvor entwickelte theoretische Modell beschreibt die Dynamik des Systems unter idealisierten Bedingungen. Besonders das reale Kontaktverhalten des Rades mit dem Untergrund wird dabei nicht berücksichtigt, was in der Praxis dazu führt, dass das Modell die tatsächliche Bewegung nur eingeschränkt abbildet. Das theoretische Modell schwingt nach kleinsten

Anregung unendlich weiter und die Orientierung des Kreisels kann aus dem Weltkoordinatensystem heraus betrachtet nicht nachhaltig verändert werden. Durch die Kopplung der Differenzialgleichungen hat jede Manipulation des Rollwinkels eine direkte Reaktion auf die Gierbewegung, welche dann wiederum das Rollen anregt. Dadurch wird eine gezielte Stabilisierung und das gezielte Kurvenfahren des Fahrzeugs unmöglich. Die Einflüsse der Interaktion des Reifens mit dem Untergrund müssen daher beachtet werden. Drei Effekte sind besonders relevant:

- **Haftreibung beim Gieren:** Tritt auf, wenn das Fahrzeug ein Moment um die Hochachse erfährt, sich aber aufgrund ausreichender Haftung nicht dreht. In diesem Bereich wird die Kopplung zwischen Roll- und Gierbewegung aufgehoben, sodass die Rollbewegung ohne Auswirkung auf die Gierbewegung manipuliert werden kann, solange die Rollgeschwindigkeit so klein ist, dass das daraus resultierende Kreiselmoment $M_{K\psi}$ kleiner als die Haftreibung ist.
- **Gleitreibung beim Gieren:** Entsteht, wenn die aufgebrachten Giermomente die Haftkraft überschreiten und das Rad sich um die Hochachse zu drehen beginnt. Dieser Effekt führt zu Energieverlusten und wirkt dämpfend auf die Gierbewegung, sodass das Fahrzeug nicht schwingt.
- **Gleitreibung beim Rollen:** Seitliche Relativbewegung in der Kontaktzone beim Rollen, verursacht durch Reibung mit dem Untergrund und innere Verformungsarbeit im Reifen. Wirkt als Dämpfung auf die Rollbewegung und reduziert Schwingungen.

Diese Effekte werden teilweise in die Modellierung aufgenommen. Die Parameter werden so gewählt, dass das Verhalten des Modells dem erwarteten Verhalten des Prototyps bestmöglich angenähert wird. Faktoren wie der Untergrund, die Art des Reifens, Temperatur, Luftfeuchtigkeit, Staub und Verschmutzung, Luftdruck des Reifens etc. machen eine theoretische Bestimmung der Reibungsparameter im Rahmen dieser Arbeit nicht praktikabel. Daher wird das Modell verwendet um Konzepte zum Regeln und Steuern zu validieren und auszuarbeiten, diese müssen dann aber auf das reale System angepasst werden bzw. der Regler muss robust genug sein, um die Modellgenauigkeiten auszugleichen.

Werden die Erkenntnisse aus der theoretischen Modellierung mit den Einflüssen der Interaktion zwischen Reifen und Untergrund kombiniert, ergibt sich das folgende Systemverhalten: Ist die Geschwindigkeit v des Fahrzeugs null oder fährt das Fahrzeug mit einer Geschwindigkeit v ungleich null und befindet sich nahe der Ruhelage ($\varphi \approx 0$ und $\dot{\varphi} \approx 0$), ist die Haftreibung um die Hochachse so hoch, dass im Normalbetrieb von $\dot{\psi} = 0$ ausgegangen werden kann. Damit hängt die Rollbewegung nur vom Kippmoment durch das Umfallen sowie dem aktuierten Kreiselmoment ab.

Fährt das Fahrzeug mit einer Geschwindigkeit v und befindet sich nicht in der Ruhelage, wirken der selbststabilisierende Effekt des Gyroskops und die Dynamik der Kurvenfahrt. Das Fahrzeug beginnt zu kippen, wodurch das Kreiselmoment $M_{K\psi}$ in Kombination mit den nicht modellierten Effekten beim Kurvenfahren eine Drehung um die Hochachse auslöst. Das Fahrzeug fährt eine Kurve und sowohl die Zentrifugalkraft F_{ZF} als auch das Kreiselmoment $M_{K\varphi}$, welche beide durch die Gierbewegung entstehen, wirken dem ursprünglichen Kippen entgegen.

4 Hardwareintegration

Die vorhandene Hardware umfasst einen Prototyp mit reduziertem Funktionsumfang sowie sämtliche Bauteile für den Aufbau des vollfunktionsfähigen Fahrzeugs. Die mechanische Konstruktion des *Monowheelers* ist in [33] vollständig dokumentiert. In diesem Kapitel wird die Zusammenführung der einzelnen Komponenten zu einem vollständigen Gesamtsystem beschrieben.

4.1 Aktorik

Die Aktorik des *Monowheelers* umfasst zwei *Dynamixel XH540-W150-T/R* Servomotoren, die zur Regelung der Nick-, Roll- und Gier-Bewegung eingesetzt werden, einen Brushless DC Motor (BLDC Motor), der das Gyroskop mit konstanter Drehzahl antreibt, sowie einen Antriebsmotor, der das Fahrzeug in Fahrtrichtung beschleunigt und abremst.

4.1.1 Dynamixel Servomotoren

Die beiden verwendeten Dynamixel sind jeweils für die Regelung und Steuerung eines der beiden in Kapitel 3 definierten Teilsysteme zuständig. Beide Dynamixel werden per RS485-Protokoll gesteuert. Um im Fehlerfall Schäden an den Motoren und der Hardware zu verhindern, werden vier Mechanismen eingesetzt:

- **Weiche Anschläge:** Für beide Dynamixel werden Anschläge aus TPU als Begrenzung der Winkelauslenkung verwendet, um die Kraftspitzen bei einer Kollision abzufedern und die restliche Hardware zu schützen.
- **Interner Überlastschutz:** die Dynamixel verfügen über einen internen Überlastschutz. Wird der benötigte Strom zu hoch, etwa weil der Dynamixel in den TPU-Anschlag fährt, schaltet sich der Dynamixel automatisch ab.
- **Software Prüfung:** Im Position-Mode lassen sich maximale und minimale Positionen vorgeben, die der Dynamixel nicht überschreitet. Im Velocity-Control-Mode prüft die Software auf dem Mikrocontroller in jeder Iteration, ob die maximale Position überschritten wird.
- **Interner Watchdog:** Erhalten die Dynamixel zu lange (100 ms) keine neuen Befehle, schalten sie sich aus und können nur durch einen Restart wieder aktiviert werden. Dadurch deaktivieren sich die Motoren auch im Falle eines Softwareabsturzes (z. B. durch einen Segmentation Fault).

Um die Dynamik der Stellgröße zu modellieren, werden die Dynamixel im folgenden Abschnitt charakterisiert und als PT1-Glied mit folgender Form angenähert:

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

Alle PT1-Fits werden mit dem Python-Skript *actuator_modelling.py* durchgeführt.

Wheel Dynamixel

Der Dynamixel für die Nick-Bewegung wird im Position-Control-Mode verwendet, da die Stellgröße die Position des Aufstandpunkts im Vergleich zur Plattform ist. Die Verschiebung des Aufstandpunkts wird mittels eines Viergelenk-Mechanismus realisiert. Da der Mechanismus aufgrund konstruktiver Beschränkungen keinen linearen Zusammenhang zwischen Servo-Winkel und Verschiebung hat, wird ein Polynom dritten Grades verwendet, um die Größen ineinander überzuführen. Das Polynom wird mit dem Python-Skript *4barlinkage.py* bzw. *4barlinkageinverse.py* berechnet. Die Berechnungsvorschrift ist in [33, S. 59ff] dokumentiert. Die Vorzeichen sind dabei der Konvention der Drehrichtung des Servomotors und der Definition der Verschiebung p entsprechend gewählt.

Der Dynamixel verfügt über einen internen PID-Regler, der parametrierbar ist. Die besten Ergebnisse für den Proportional- und Differenzial-Anteil lassen sich bei $K_P = 800$, $K_D = 50$ erzielen. Höhere Werte führen zu konstanten Oszillationen und Überschwingern, niedrigere Werte verschlechtern die Dynamik. Für die Vermessung der Stellgröße wird eine Verschiebung p vorgeben und das reale Verhalten aufgezeichnet. Die Sollwertvorgabe orientiert sich an dem realistischen Arbeitsbereich im Reglerbetrieb, abgeschätzt aus der Simulation. Die Ergebnisse einer Messung sind beispielhaft in Abbildung 4.1 zu sehen.

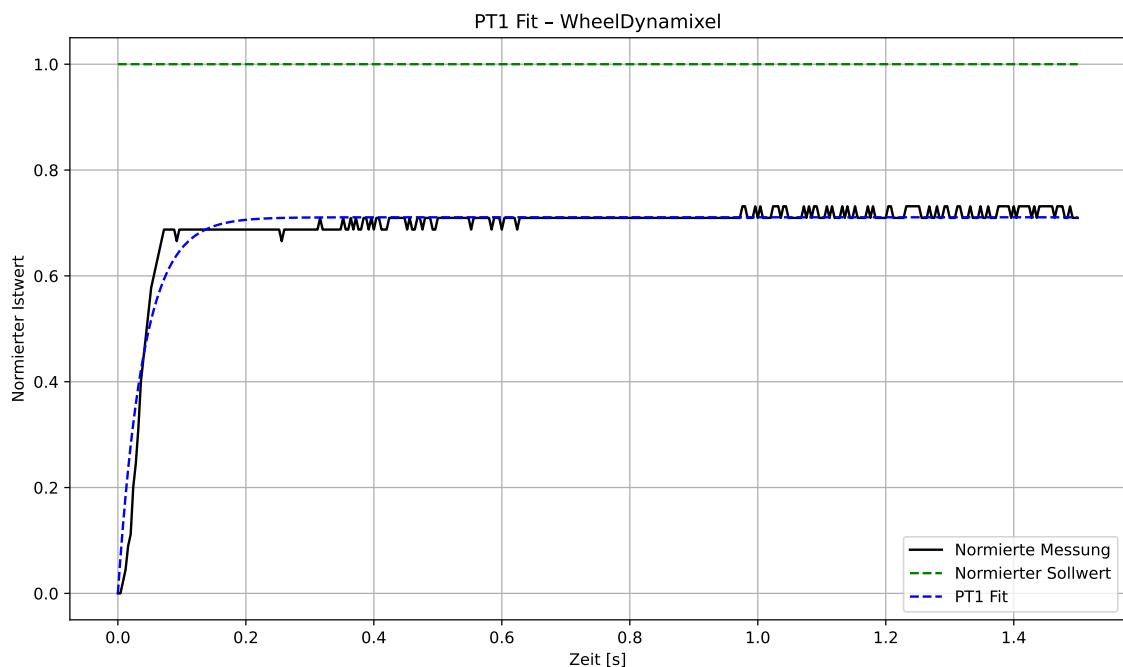


Abbildung 4.1: Wheel Dynamixel PT1 Fit 10% Sollwertsprung, $K_P = 800$, $K_D = 50$, $K_I = 0$

Damit ergeben sich folgende Werte für den PT1-Fit (Tabelle 4.1):

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

Sollwertsprung	K	T
2 %	0,429	0,048
10 %	0,711	0,040
30 %	0,889	0,044

Das in Abbildung 4.1 gezeigte Verhalten entspricht eher einem PT2-Glied mit Totzeit in der Größenordnung 5 ms. Aus Tabelle 4.1 geht zudem hervor, dass die Parameter des PT1-Fits sich je nach Sollwertvorgabe stark unterscheiden. Ein PT1-Fit mit gemittelten Werten für K und T bildet die wesentliche Dynamik aber ausreichend ab und ermöglicht eine simple und robuste Beschreibung der Dynamik der Stellgröße. Ebenfalls fällt die niedrige Verstärkung auf, besonders bei kleinen Sollwertsprüngen. Um dies zu beheben, wird ein Integral-Anteil eingeführt (siehe Abbildung 4.2):

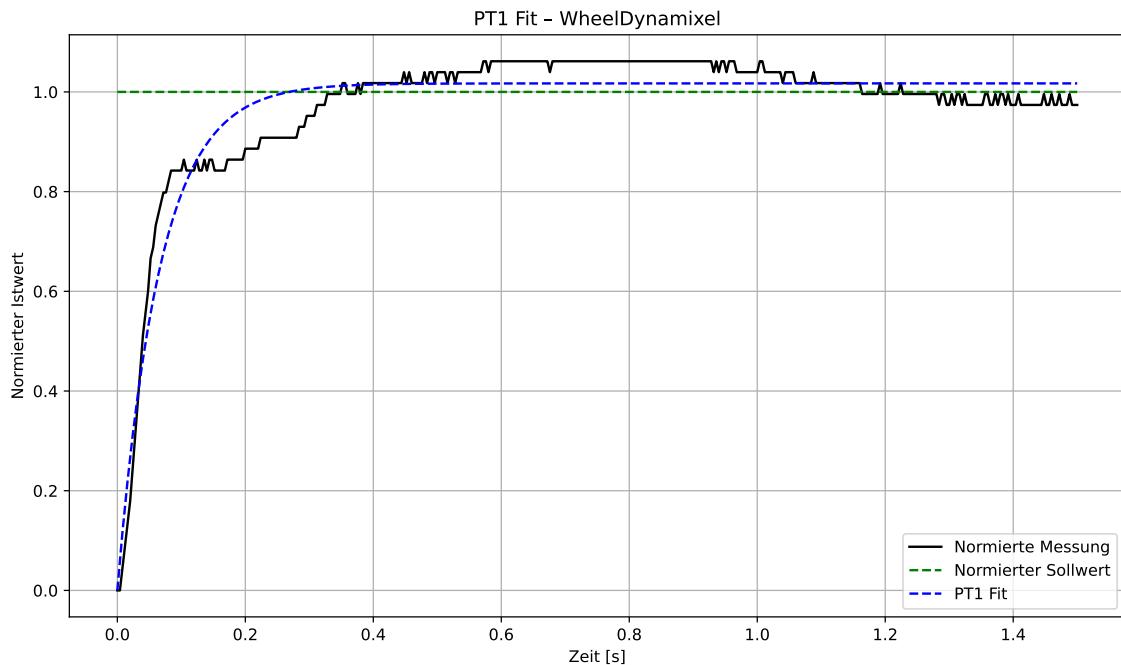


Abbildung 4.2: Wheel Dynamixel PT1 Fit 10% Sollwertsprung, $K_P = 800$, $K_D = 50$, $K_I = 1000$

Damit ergeben sich folgende Werte für den PT1-Fit (Tabelle 4.2):

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

Sollwertsprung	K	T
2 %	1,127	0,243
10 %	1,017	0,066
30 %	0,994	0,042

Grundsätzlich hat sich die Verstärkung bei kleinen Sollwertsprüngen von weniger als 0,5 auf ca. 1 erhöht. In Abbildung 4.2 ist aber zu erkennen, dass das reale System erst auf denselben Endwert wie ohne Integral-Anteil springt, dann dort stehen bleibt bis der Integral-Anteil zu wirken beginnt, bevor sich der Istwert nach einem Überschwinger auf die endgültige Position begibt. In der Realität verändert sich der Sollwert aber ständig, sodass der Integral-Anteil kaum eine Verbesserung der Genauigkeit bringt, da die Zeitkonstante zu groß ist. Zugunsten der Dynamik wird daher auf den Integral-Anteil verzichtet und es werden die Parameter $K_P = 800$, $K_D = 50$, $K_I = 0$ verwendet. Als PT1-Fit werden die Parameter $K = 0,85$ und $T = 0,045$ gewählt.

Gyro Dynamixel

Der Dynamixel für die Roll- und Gier-Bewegung wird im Velcoity-Control-Mode verwendet, da die Stellgröße die Winkelgeschwindigkeit des Gyroskops $\dot{\psi}_K$ ist. Auch hier kommt ein Viergelenk-Mechanismus zum Einsatz, die Stellgröße entspricht aber genau der negativen Winkelgeschwindigkeit des Dynamixels. Das Vorzeichen wird in der Software beachtet. Intern steuert der Dynamixel die Geschwindigkeit mit einem PI-Regler. Zunächst wird die Regelstrecke mit den voreingestellten PI-Parametern des Dynamixels vermessen. Dazu werden mehrere Sprünge im Bereich der erwarteten Stellgröße auf die Strecke gegeben. Beispielhaft ist eine Sprungantwort in Abbildung 4.3 zu sehen.

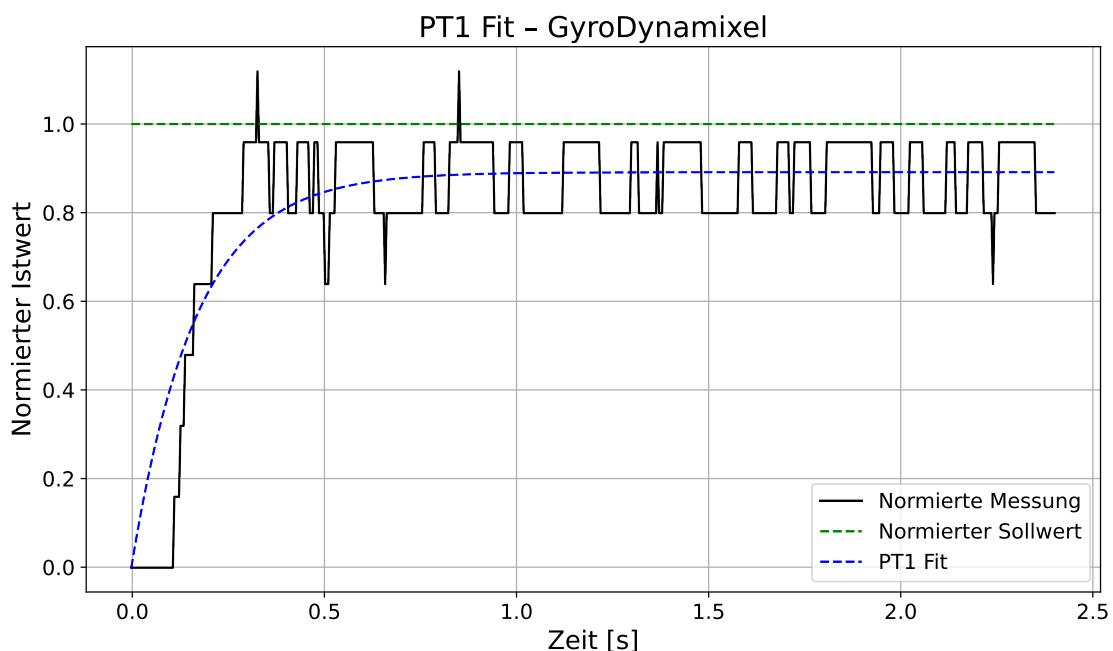


Abbildung 4.3: Kreisel Dynamixel PT1 Fit 3% Sollwertsprung, $K_P = 100$, $K_I = 1980$

In Tabelle 4.3 sind die PT1-Fit Ergebnisse des Gyro Dynamixels für $K_P = 100$, $K_I = 1980$ zu sehen:

Tabelle 4.3: PT1-Fit Gyro Dynamixel mit $K_P = 100$, $K_I = 1980$

Sollwertsprung	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

Aus Tabelle 4.3 ist erkennbar, dass der Dynamixel bei größeren Sprüngen deutlich akkutater (K näher an 1) und schneller (T kleiner) wird. Da die Regelung aber größtenteils im unteren Bereich des Geschwindigkeitsspektrums stattfindet, werden die Reglerparameter auf $K_P = 500$, $K_I = 5000$ erhöht (beispielhafte Sprungantwort in Abbildung 4.4):

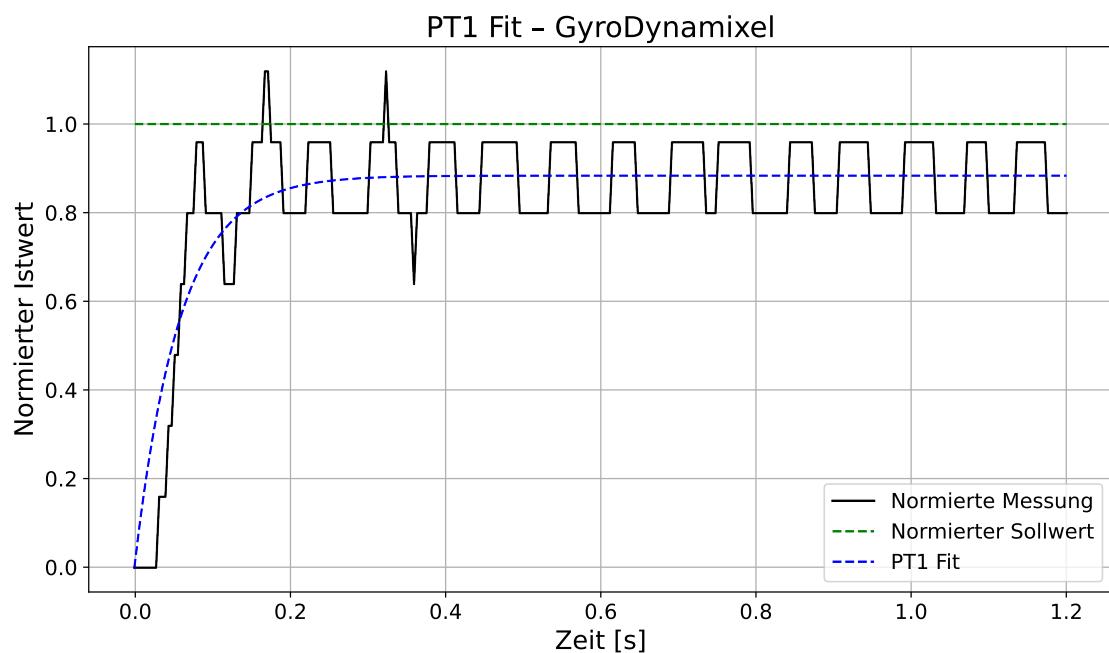


Abbildung 4.4: Kreisel Dynamixel PT1 Fit 3% Sollwertsprung, $K_P = 500$, $K_I = 5000$

In Tabelle 4.4 sind die PT1-Fit Ergebnisse des Gyro Dynamixels für $K_P = 500$, $K_I = 5000$ zu sehen:

Tabelle 4.4: PT1-Fit Gyro Dynamixel mit $K_P = 500$, $K_I = 5000$

Sollwertsprung	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

Die Ergebnisse in Tabelle 4.4 zeigen eine große Verbesserung der Zeitkonstanten, besonders bei kleineren Sprüngen. Allerdings verschlechtert sich die Dynamik bei großen Sprüngen. In Abbildung 4.5 ist zu erkennen, dass der Dynamixel bei größeren Sollwertsprüngen anfängt zu schwingen:

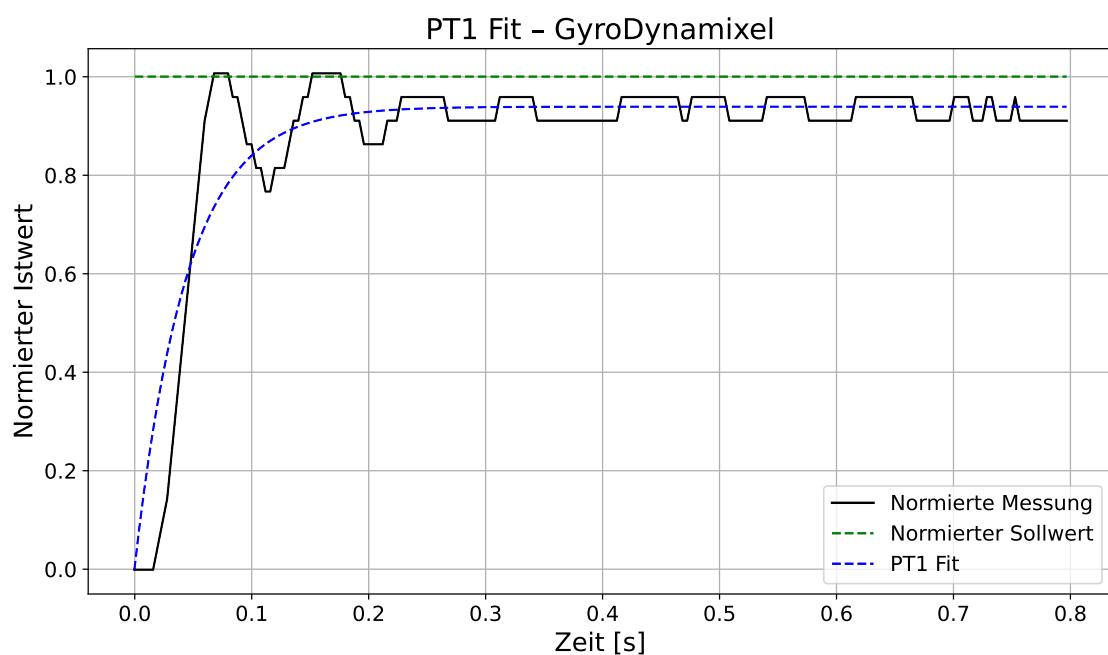


Abbildung 4.5: Kreisel Dynamixel PT1 Fit 10% Sollwertsprung, $K_P = 500$, $K_I = 5000$

Daher werden die Regelparameter verringert, bis sich bei $K_P = 300$, $K_I = 3000$ ein Kompromiss aus schneller Dynamik und einem stabilen System ergibt. Wie in Abbildung 4.6 schwingt das System auch bei dem größten getesteten Sprung nicht:

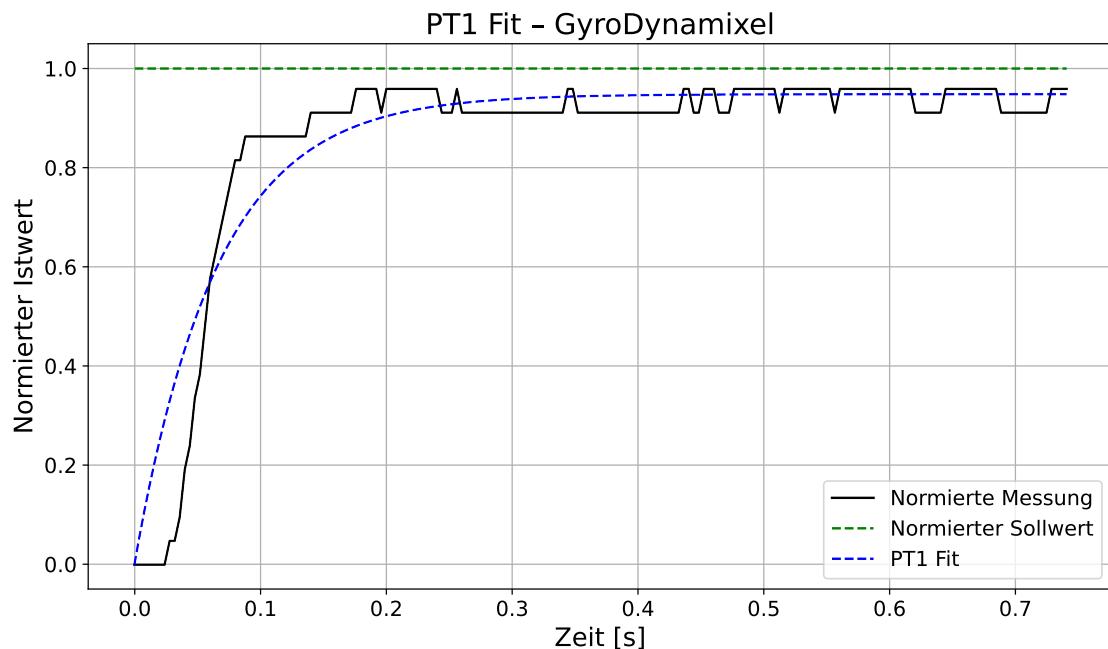


Abbildung 4.6: Kreisel Dynamixel PT1 Fit 10% Sollwertsprung, $K_P = 300$, $K_I = 3000$

In Tabelle 4.5 sind die PT1-Fit Ergebnisse des Gyro Dynamixels für $K_P = 300$, $K_I = 3000$ zu sehen:

Tabelle 4.5: PT1-Fit Gyro Dynamixel mit $K_P = 300$, $K_I = 3000$

Sollwertsprung	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

Mit den angepassten Regelparametern ist die Strecke nun deutlich schneller als mit den Standardeinstellungen. Ähnlich dem Wheel Dynamixel geht aus den Messungen hervor, dass das Verhalten eher einem PT2-Glied mit Totzeit entspricht und sich die Parameter je nach Sollwertvorgabe stark unterscheiden. Ein PT1-Fit mit gemittelten Werten für K und T bildet die wesentliche Dynamik aber ausreichend ab und ermöglicht eine simple und robuste Beschreibung der Dynamik der Stellgröße. Für die Simulation und Modellierung wird eine Verstärkung von $K = 0,85$ und eine Zeitkonstante von $T = 0,130$ gewählt.

4.1.2 Kreiselmotor

Um genügend Moment zur Regelung und Steuerung der Roll- und Gier-Bewegung bereitzustellen, muss sich das Gyroskop mindestens mit einer konstanten Winkelgeschwindigkeit $w_K = 4000 \text{ rpm}$ drehen [33, S. 6]. Dafür wird ein *MEGA Motor ACn 22/20/3* [24] mit einem Riemenantrieb verwendet [33, S. 73]. Als Motortreiber wird ein *Jeti Spin 66 pro opto* [20] verwendet [33, S. 73], der mit der passenden *Jeti Box* konfiguriert wird.

Der Treiber trennt das Steuersignal mit einem Optokoppler von den Motorströmen und braucht neben der Versorgung für den Motor eine eigene 5 V Spannungsversorgung. Dabei wird der Treiber als sensorloser Drehzahlregler verwendet, der die Geschwindigkeit des Motors anhand der Counter Electromotive Force (CEMF) bestimmt, was nur bei angelegter Last und ab einer Mindestdrehzahl zuverlässig funktioniert.

Die Ansteuerung erfolgt mit einem Pulse-Width Modulation (PWM)-Signal mit 50 Hz. Vor dem Betrieb muss der Treiber aktiviert werden, wofür ein PWM-Duty-Cycle leicht unterhalb (ca. 0,1 ms – 0,2 ms) des Wertes für die minimale Drehzahl eingestellt wird. Die Beschleunigung erfolgt mittels einer Rampe. Beim ersten Hochlaufen nach dem Aktivieren wird diese Rampe auf den gesamten Beschleunigungsvorgang angewendet. Bei erneutem Start aus dem Stillstand greift die Rampe erst ab der eingestellten Mindestdrehzahl, was zu Überlastung des Riemenantriebs und der Stromversorgung führen kann. Daher muss die Software sicherstellen, dass der Treiber nach jedem Durchlauf vollständig ausgeschaltet wird.

Alle Experimente in dieser Arbeit werden bei einer Kreiseldrehzahl von $w_K = 5000 \text{ rpm}$ durchgeführt. Diese Drehzahl ergibt sich als Kompromiss zwischen Energieeffizienz und Kreiselmoment, abgeschätzt mit der Simulationsstudie in [33, S. 37ff].

4.1.3 Antriebsrad

Bei dem Antriebsrad handelt es sich um einen E-Scooter Radmotor *Xiaomi M365 Essential 1S Pro*. Der Motor verfügt über 15 Polpaare, ca. eine Drehzahlkonstante $K_V = 22 \text{ min}^{-1} \text{ V}^{-1}$ und eine Drehmomentkonstante $K_M = 0,52 \text{ N m A}^{-1}$ [33, S. 71].

Gesteuert wird das Antriebsrad mit einem *Maxon Escon 70/10*, welcher die Position des Rotors mit den verbauten Hall-Sensoren bestimmt. Der Treiber wird so konfiguriert, dass der Sollwert per PWM-Signal zwischen 10 Hz und 5 kHz vorgegeben werden kann [23, S. 21]. Außerdem verfügt der Motor über ein Freigabe-Signal, um den Motor an- und auszuschalten und über einen analogen Ausgang, der die aktuelle Drehzahl ausgibt. Dabei ist zu beachten, die Konfiguration der analogen Spannung so an die maximale Drehzahl anzupassen, dass das Signal nie den gültigen Spannungsbereich des Analog to Digital Converter (ADC)-Wandlers des Mikrocontrollers über- oder unterschreitet.

Wie bereits erwähnt, verwendet der Treiber die eingebauten Hall-Sensoren des Motors zur Bestimmung der Rotorposition. Allerdings ist das Verhalten des Motors inkonsistent und der Motor ruckelt im Betrieb. Die Sensor-Signale sind bei bestromten Motor so verrauscht, dass eine akkurate Positionsbestimmung des Rotors unmöglich ist (siehe Abbildung 4.7):



Abbildung 4.7: Hall-Sensor Signale Antriebsrad unter Last, unmodifiziert

Die Kabel der Motorphasen und der Hall-Sensoren werden von Werk aus ungeschirmt miteinander verdrillt in einem Kabel geführt. Um die elektromagnetischen Störungen durch die induktive Kopplung zu reduzieren, werden die Sensor-Leitungen möglichst getrennt von den Motorphasen geführt. Durch das sich zeitlich ändernde Magnetfeld in der Motorphase wird in den Signalleitungen ein Strom induziert, und die daraus resultierende, störende Spannung ist proportional zu der Impedanz des Leiters:

$$\vec{u} = \vec{Z} \cdot \vec{i} \quad (4.2)$$

Um die Impedanz und damit auch die störende Spannung zu senken, werden parallel zu den internen 2,7 kΩ Pull-Up-Widerständen im Treiber [23, S. 16] externe 1 kΩ Pull-Up-Widerstände verbaut. Die Impedanz der parallelen Widerstände ist rein real und kann wie folgt bestimmt werden:

$$R_{ges} = \frac{R_1 \cdot R_2}{R_1 + R_2} = \frac{2,7 \text{ k}\Omega \cdot 1 \text{ k}\Omega}{2,7 \text{ k}\Omega + 1 \text{ k}\Omega} \approx 730 \Omega \quad (4.3)$$

Dabei muss beachtet werden, dass die Spannungsquelle der Hall-Sensoren nur begrenzt Strom liefern kann. Pro Sensor ist folgender Strom benötigt:

$$I = \frac{U}{R_{ges}} = \frac{5 \text{ V}}{730 \Omega} \approx 7 \text{ mA} \quad (4.4)$$

Die Spannungsquelle des Treibers liefert maximal 30 mA und es können maximal zwei Hall-Sensoren gleichzeitig schalten. Damit ist der maximale benötigte Strom kleiner als der maximal zur Verfügung stehende Strom. Mit diesen Maßnahmen wird das Rauschen so stark reduziert, dass die Ansteuerung des Motors zuverlässig funktioniert (siehe Abbildung 4.8):

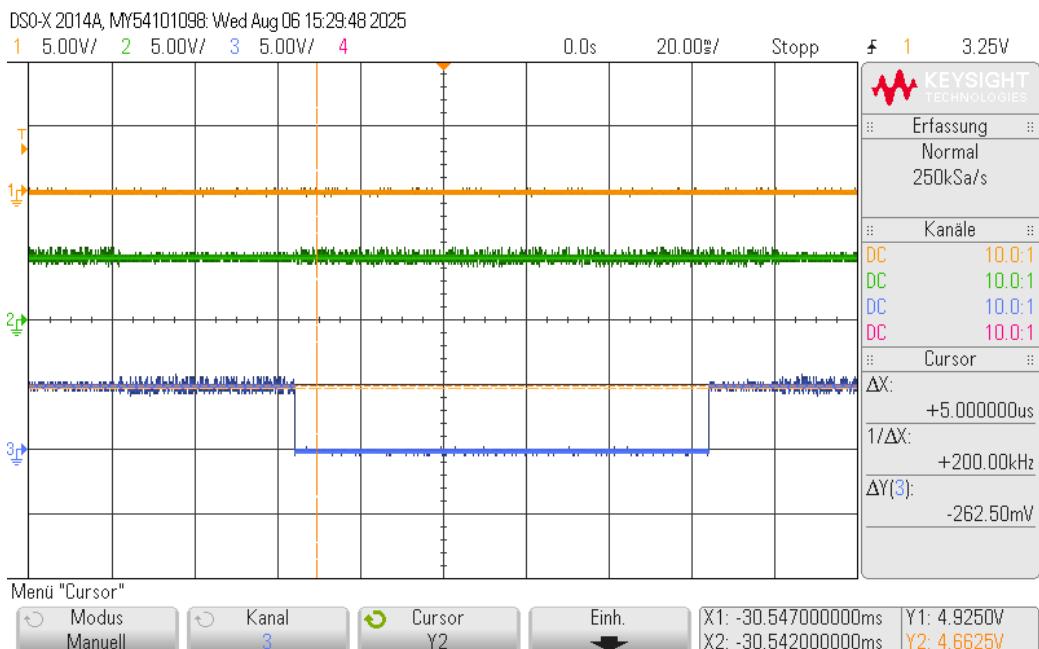


Abbildung 4.8: Hall-Sensor Signale Antriebsrad unter Last, getrennte Kabelführung und Impedanzanpassung

Da die Hall-Sensoren nur sechs Zustände pro Umdrehung liefern, funktioniert die interne Geschwindigkeitsregelung des Motortreibers erst ab ca. 30 rpm (3 Pulse/s) zuverlässig. Ohne Beschleunigungsrampe ist das Drehmoment beim Beschleunigen auf die Mindestdrehzahl zu hoch für das Fahrzeug. Wird jedoch eine Rampe verwendet, ruckelt der Motor beim Anfahren aufgrund der niedrigen Auflösung der Sensoren. Das Drehmoment kann aber sehr präzise vorgegeben werden, auch bei sehr langsamem Geschwindigkeiten. Der Motortreiber bietet zudem die Möglichkeit, die aktuelle Geschwindigkeit als analoge Spannung auszugeben. Dieser Spannungswert ist auch bei sehr niedrigen Geschwindigkeiten akkurat und kontinuierlich. Daher wird der Motor im Current-Control-Mode mit einem externen PID-Geschwindigkeitsregler betrieben.

4.2 Elektronik

Die Elektronik übernimmt zwei zentrale Aufgaben: die Weiterleitung und Verarbeitung von Mess- und Steuersignalen zwischen Sensorik, Aktorik und Mikrocontroller und die Energieversorgung aller Komponenten. Auf Basis der in [33, S. 88ff] entwickelten Platine wird eine überarbeitete Version entworfen. Dabei werden folgende Fehler aus der ersten Iteration der Platine behoben:

- Anschluss Steuersignal Kreiselmotor: Anstatt eines Enable-, PWM- und Direction-Pins braucht der *Jeti Spin 66 pro opto* Treiber eine 5 V Spannungsquelle, einen Masse-Anschluss und einen PWM-Pin.
- Anschluss Steuersignal Antriebsrad: Anstatt eines Enable-, PWM- und Direction-Pins braucht der *Maxon Escon 70/10* Treiber einen Masse-Anschluss, einen PWM-Pin und einen Enable-Pin.
- Spannungsversorgung BeagleBone Black (BBB): Anschluss der 5 V Versorgungsspannung an die Pins *VDD_5V* (P9.5 und P9.6) anstatt an die Pins *SYS_5V* (P9.7 und P9.8).
- Hinzufügen eines USB-Anschlusses für die Versorgung des WLAN-Moduls und Anpassung des Gleichspannungswandlers.
- Fehlerhafte Universal Asynchronous Receiver/Transmitter (UART)-RS485-Protokoll Schaltung: Der gewählte IC zur Konvertierung der UART- und RS485-Protokoll-Signale arbeitet mit 5 V-Spannungsspeigel, während das BBB mit 3,3 V arbeitet. Auf der vorhandenen Platine ist eine mangelhafte Schaltung zur Konvertierung der Spannungsniveaus verbaut, wodurch unzulässige Spannungen an Pin P9.3, P9.4, P9.11, P9.13, P9.24 und P9.26 entstehen können, die zur Zerstörung eines BBB führen.

Während die eingesetzten Komponenten – wie Mikrocontroller, WLAN-Modul und Aktoren – unverändert bleiben, wird die interne Anbindung angepasst, um die zuvor identifizierten Fehler der vorhandenen Platine zu beheben und die elektrischen Eigenschaften zu verbessern.

4.2.1 Übersicht Komponenten

In Tabelle 4.6 sind alle Komponenten und die zugehörigen Schnittstellen aufgelistet, die durch die Platine verbunden werden müssen. Die Auswahl der Komponenten ist in [33] beschrieben.

Tabelle 4.6: Übersicht elektrischer Komponenten und Anschlüsse

Komponente	Stromversorgung	Steuersignale
<i>Maxon Escon 70/10</i>	24 V, max. 15 A	Masse, PWM (3,3 V), Enable (3,3 V)
<i>Jeti Spin 66 pro opto</i>	24 V, max. 70 A	Analog Masse u. Spannung (1,8 V)
<i>TP-Link AC750-WLAN-Router</i>	5 V, max. 2 A	Masse, PWM (3,3 V), 5 V-Pin
BBB	5 V, max. 2 A	-
<i>ICM20948 (IMU)</i>	-	siehe Pin-Belegung in Tabelle 4.6
2x <i>Dynamixel XH540-W150-T/R</i>	12 V, max. 5 A	Masse, 3,3 V-Pin, SPI (3,3 V) RS485-Protokoll

In Tabelle 4.7 sind die Anschlüsse der General Purpose Input Output (GPIO) des BBB dokumentiert:

Tabelle 4.7: Pinbelegung BBB

Funktion	Hardware-Modul	Pin - Nr.	GPIO - Nr.
Spannungsversorgung Input			
GND	-	P9.1 und P9.2	-
5 V	-	P9.5 und P9.6	-
Spannungsversorgung 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 Funktionsstruktur und Designrichtlinien

Die Platine wird in zwei Sektionen aufgeteilt, entsprechend der beiden Aufgaben: der Signalverarbeitung und Weiterleitung (Signalelektronik) sowie der Stromversorgung der

Komponenten (Leistungselektronik). Die Signalelektronik umfasst dabei zwei Serial Peripheral Interface (SPI)-Schnittstellen für die Sensoren, jeweils einen Anschluss für den Kreiselmotor und das Antriebsrad, zwei UART-RS485-Konverter für die beiden Dynamixel Servomotoren und die Anbindung des BBB. Die Leistungselektronik bietet zwei Anschlüsse für einen sechszelligen Lithium-Polymer-Akkumulator (LiPo) und einen dreizelligen LiPo, zwei 24 V-Anschlüsse für den Kreiselmotor und das Antriebsrad, zwei 12 V-Anschlüsse für die Dynamixel Servomotoren und einen DCDC-Wandler auf 5 V, für die Spannungsversorgung des WLAN-Routers, des BBB und des Empfängers des *Jeti Spin 66 pro opto*. Eine Übersicht aller Komponenten und Verbindungen ist in der Funktionsstruktur in Abbildung 4.9 veranschaulicht:

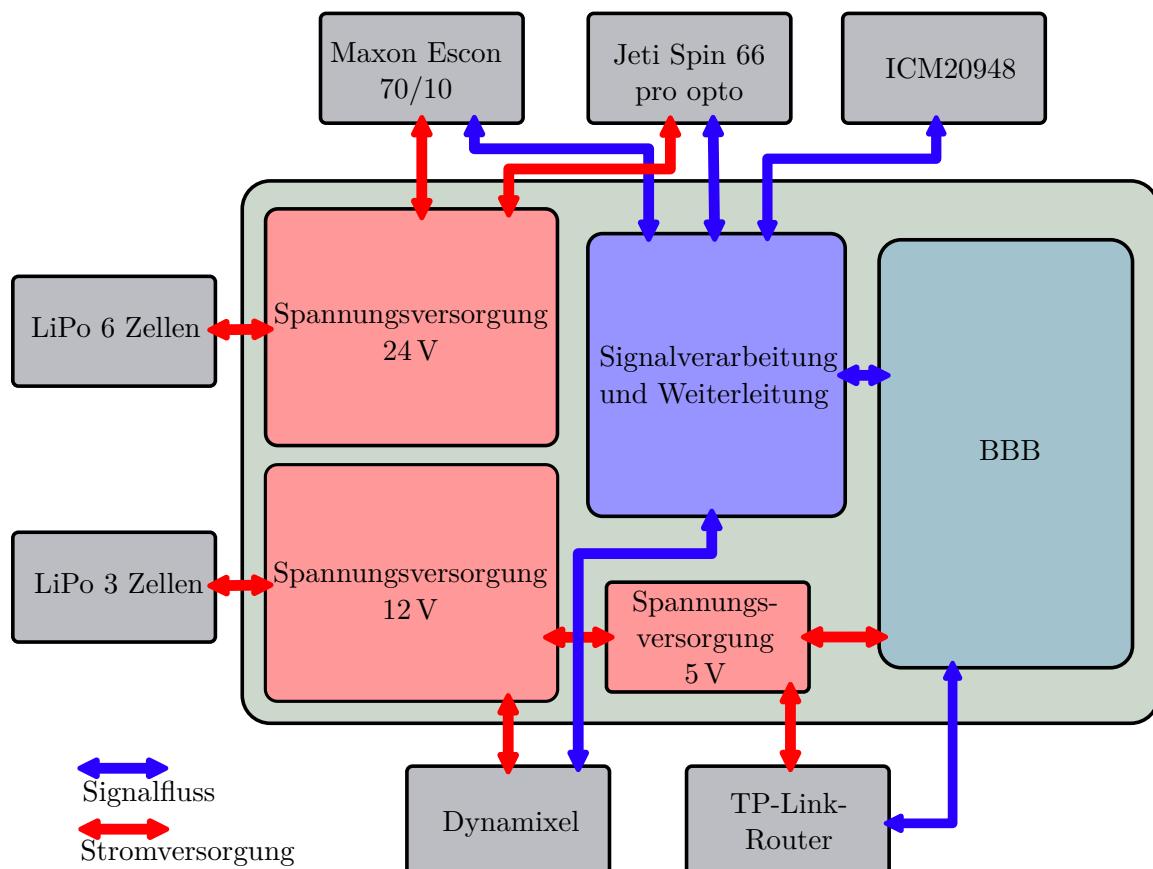


Abbildung 4.9: Funktionsstruktur Elektronik

Um eine saubere Signalführung und stabile Energieversorgung zu gewährleisten, wird die Platine unter Berücksichtigung folgender Designrichtlinien ausgelegt [3]:

- Platine: Die Platine wird als zweilagige Leiterplatte ausgeführt.
- Layout: Die hohen Ströme und schnellen Stromänderungen der Motortreiber erzeugen starke elektromagnetische Felder, die empfindliche Signalleitungen stören können. Die hohen Rückströme über die Masse können zu unerwünschten Spannungsabfällen und damit einem undefinierten oder rauschenden Bezugspotenzial führen. Um die Elektromagnetische Verträglichkeit (EMV) zu optimieren und eine hohe Signalqualität sicherzustellen, werden Leistungselektronik und Signalelektronik räumlich getrennt.

- Masse: Es werden zwei Masseflächen verwendet, die nur an einem einzigen, definierten Punkt verbunden sind. Die Trennung sorgt dafür, dass die Rückströme der Leistungselektronik die Signalmasse nicht beeinflussen. Die Verbindung gewährleistet ein gemeinsames Bezugspotenzial. Durch Verwendung von Masseflächen wird zusätzlich die Impedanz des Rückstrompfads verringert, was Rauschen der Masse, Signalreflexionen und Schleifenbildung minimiert. Die definierte Verbindung der Masseflächen soll ebenfalls eine möglichst niedrige Impedanz aufweisen und so platziert werden, dass der Rückstrompfad für die Versorgungsspannung der Signalelektronik möglichst kurz ist.
- Leitungsführung: Leitungen sollen die Masseflächen möglichst nicht unterbrechen. Stromführende Leiterbahnen werden in ihrer Breite entsprechend dem zu erwartenden Laststrom angepasst. Bei sehr hohen Strömen sollen Kupferflächen anstatt einfachen Leitungen zum Einsatz kommen, um Verlustleistung, thermische Beständigkeit und Spannungsabfälle zu reduzieren. Rechte Winkel werden aufgrund möglicher Signalreflexionen vermieden[3, S. 16].
- Glättungskondensatoren: Zur Unterdrückung hochfrequenter Störungen werden an den Ausgängen von DCDC-Wandlern, an jedem Versorgungspin der integrierten Schaltungen sowie an den Stromversorgungssteckern Keramikkondensatoren mit einer Kapazität von 100 nF eingesetzt. Diese werden so nah wie möglich an der jeweiligen Störquelle positioniert und über kurze Rückstromfade mit der Masse verbunden, um ihre Wirksamkeit zu steigern.
- Stützkondensatoren: Um die Versorgungsspannung stabil zu halten, auch bei schnellen Lastwechseln und Schwankungen der LiPos, werden pro Spannungsschiene große Stützkondensatoren (Elektrolytkondensatoren (ELKOs)) eingesetzt, die als Energiepuffer dienen. Dabei wird ein definierter Entladepfad vorgesehen, um ohne angeschlossene Verbraucher sicherstellen zu können, dass die ELKOs sicher entladen werden.
- Thermische Entlastung: Alle Lötpads sollen thermische Entlastungsspeichen haben, um die Lötbarkeit zu verbessern.

4.2.3 Leistungselektronik

Der folgende Abschnitt beschreibt den Entwurf der Leistungselektronik anhand der beschriebenen Designrichtlinien und Anforderungen. Die Anbindung nach außen erfolgt mit XT60-Steckern. Um die LiPos per Schalter vom Rest der Platine trennen zu können, kommt eine High-Side Switch Schaltung (ähnlich zu [33, S. 89]) zum Einsatz (siehe Abbildung 4.10):

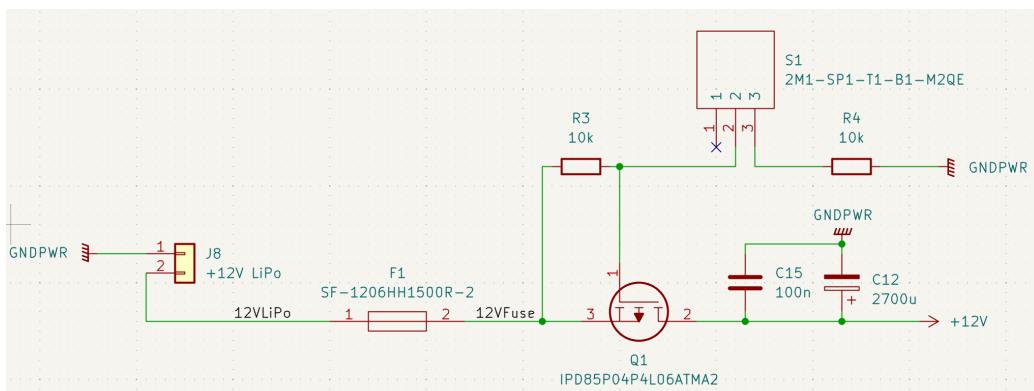


Abbildung 4.10: High-Side Switch Schaltung Spannungsversorgung 12 V

Direkt nach dem Eingang des LiPos wird eine Schmelzsicherung verbaut, um die Platine und Bauteile vor Beschädigungen zu schützen. Bei dem Transistor handelt es sich um einen P-Channel Mosfet [16], der je nach Gate-Source-Spannung zwischen Source und Drain leitet oder sperrt. Ist die Gate-Source-Spannung größer als ca. $-1,7\text{ V}$, sperrt der Mosfet zwischen Source und Gate. Ab diesem Threshold nimmt die Leitfähigkeit des Transistors mit sinkender Gate-Source-Spannung zu. Ab ungefähr $-5\text{ V} - -10\text{ V}$ kann der Mosfet als Leiter angenommen werden [16, S. 2, S. 5].

Source ist mit dem LiPo verbunden. Die Gate-Source Spannung kann nun über einen Schalter mit Spannungsteiler mit R_3 und R_4 verändert werden. Ist der Schalter offen, ist Widerstand R_3 nicht mit Masse verbunden, und es fällt keine Spannung ab. Damit beträgt die Gate-Source Spannung $V_{GS} = 0\text{ V}$ und der Transistor sperrt. Wird der Schalter geschlossen, fällt über dem Widerstand R_3 eine Spannung ab, sodass die Gate-Source-Spannung negativ wird und der Transistor leitet. Der Spannungsteiler wird dabei so ausgelegt, dass die Gate-Source Spannung bei geschlossenem Schalter sowohl bei maximalen als auch minimalen LiPo-Ladestand zwischen $-5\text{ V} - -10\text{ V}$ liegt.

Am Ausgang Drain sind ein Glättungskondensator und ein Stützkondensator verbaut, um eine möglichst stabile Spannungsversorgung zu gewährleisten.

Die 12 V Spannungsversorgung und die 24 V Spannungsversorgung sind sehr ähnlich aufgebaut. Die Spannungsteiler zum Ein- und Ausschalten sind an das jeweilige Spannungsniveau angepasst und bei der 24 V Spannungsversorgung werden zwei Mosfets parallel verwendet, um die thermischen Vorgaben einzuhalten [33, S. 89f].

Die 5 V Spannungsversorgung wird mit einem DCDC-Wandler realisiert, der von der 12 V Spannungsversorgung gespeist wird. Sowohl am Ein- als auch Ausgang werden Glättungs- und Stützkondensatoren verwendet. Eine Übersicht des gesamten Schaltplans ist in Anhang A zu finden.

4.2.4 Signalelektronik

Dieser Abschnitt beschreibt den Entwurf der Signalelektronik anhand der beschriebenen Designrichtlinien und Anforderungen. Die Anbindung nach außen erfolgt mit JST-XH-Steckern.

Sowohl die SPI-Anschlüsse für die Sensoren als auch die Anschlüsse für die Motortreiber erfordern kaum zusätzlichen Schaltungsaufwand. Bis auf die Glättungskondensatoren wird als einziges zusätzliches Bauteil ein Pull-Down-Widerstand für den Enable-Pin des *Maxon Escon 70/10* verwendet, um immer einen definierten Zustand zu garantieren.

Um bei der Konvertierung von Transistor-Transistor-Logik (TTL)-RS485-Protokoll auf Pegelwandler verzichten zu können, wird ein Transceiver gewählt, der mit 3,3 V-Logik arbeiten kann. Außerdem soll der Transceiver ein Auto-Direction Feature haben, um die Implementierung der Kommunikation zu vereinfachen. Der *THVD1426*-Transceiver [38] erfüllt diese Anforderungen und verfügt über eine ausreichend hohe Datenrate. Der Transceiver wird dem Datenblatt [38, S. 18] entsprechend verbaut (siehe Abbildung 4.11):

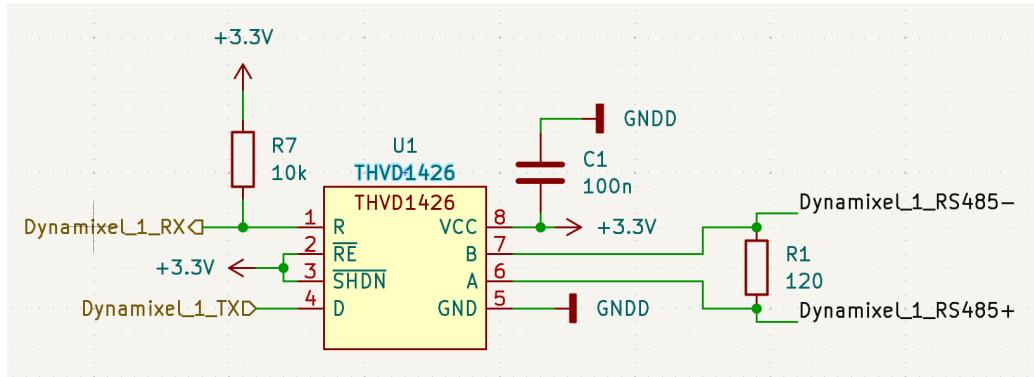


Abbildung 4.11: Schaltung zur Konvertierung von TTL-RS485-Protokoll

Da es sich bei dem RS485-Protokoll um ein differenzielles Signal mit einer typischen Leitungsimpedanz von 120Ω handelt, sind Terminierungswiderstände an beiden Enden der Leitungen vorzusehen [32]. Auf der Platine wird daher solch ein Widerstand verbaut. Eine Übersicht des gesamten Schaltplans ist in Anhang A zu finden.

4.2.5 Layout

In Abbildung 4.12 ist das Layout der Platine zu sehen.

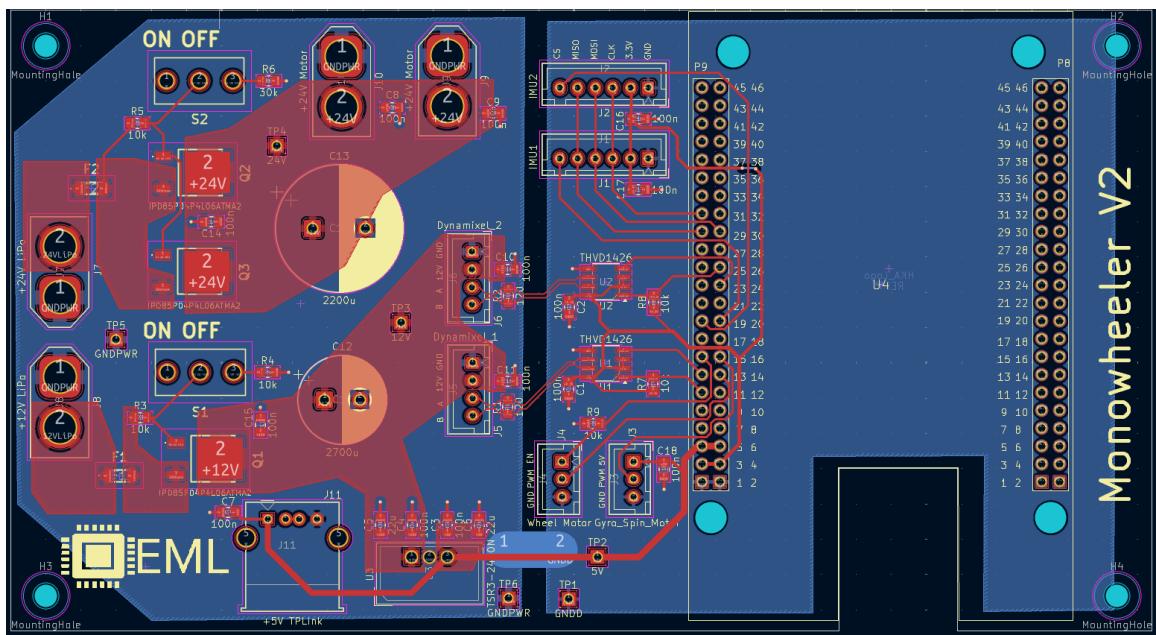


Abbildung 4.12: Platine Layout

Auf der linken Seite befindet sich die Leistungselektronik, auf der rechten Seite die Signalelektronik. Die beiden Masseflächen sind an einer Stelle durch eine dicke Kupferbahn verbunden, direkt unter dem Leiter, der die 5 V-Spannungsversorgung für das BBB führt. Zusätzlich sind Kupferpads verbaut, die als Messpads dienen oder als Lötstelle, sollten zusätzliche Kabel benötigt werden oder Korrekturen erforderlich werden.

4.2.6 Ergebnisse

Alle Designrichtlinien und Anforderungen konnten umgesetzt werden. Die Komponenten können erfolgreich und zuverlässig angesteuert werden und es treten keine thermischen Probleme auf, und auch die bereitgestellte Leistung ist ausreichend. Auf der Empfängerseite des RS485-Busses an den Dynamixeln ist bisher kein Terminierungswiderstand verbaut, es gibt aber unregelmäßige Kommunikationsprobleme, die eventuell durch den Einsatz des zweiten Terminierungswiderstandes an den Dynamixeln behoben werden können. Die Fehler treten so selten auf, dass kein negativer Effekt auf das Verhalten des Fahrzeugs festgestellt werden kann. In Abbildung 4.13 ist ein Bild des fertigen Fahrzeugs zu sehen:

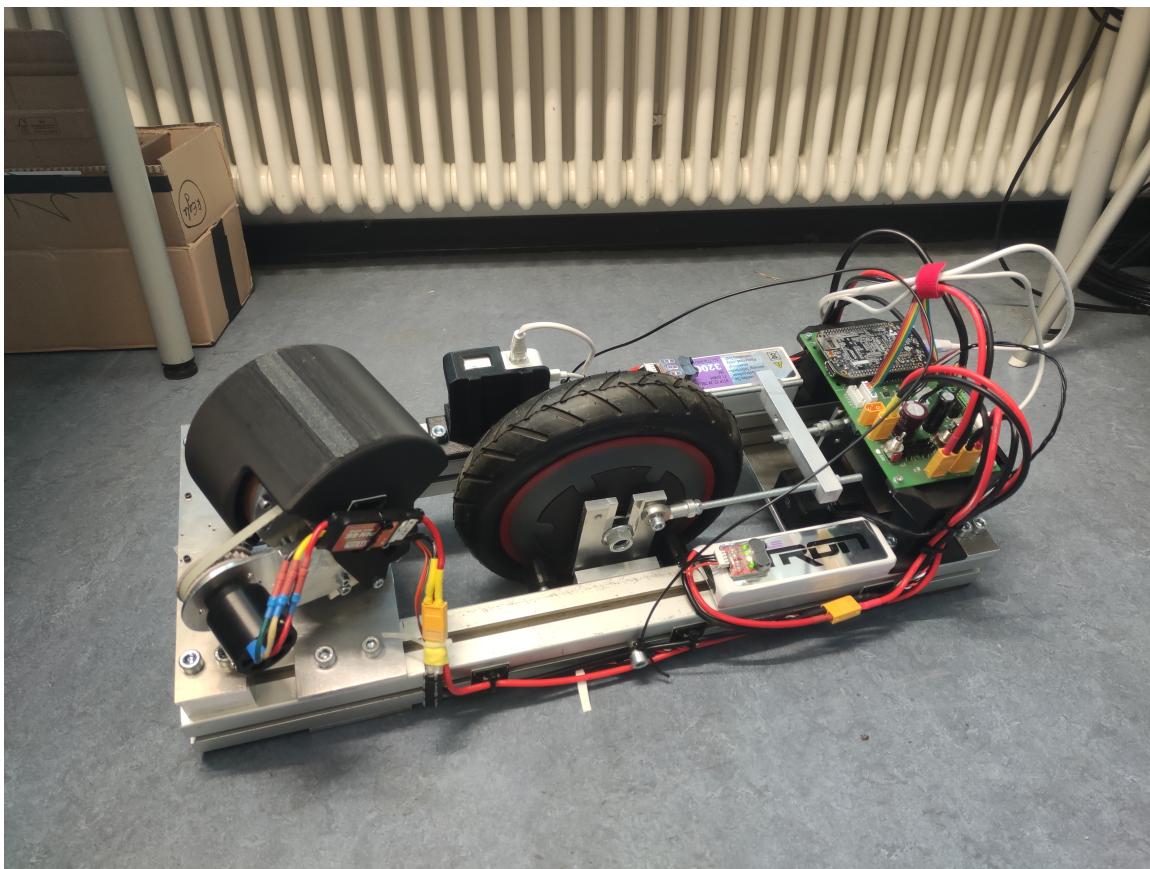


Abbildung 4.13: Monowheeler Aufbau

5 Sensorkonzept und Signalverarbeitung

Für die Regelung eines balancierenden Fahrzeugs spielt die Qualität der Sensordaten eine zentrale Rolle. Da es sich um ein instabiles System handelt, hängt die Reglergüte unmittelbar von der Genauigkeit, Konsistenz und zeitlichen Verfügbarkeit der Messgrößen ab. Ungenauigkeiten, Verzögerungen oder Rauschen in den Sensordaten können die Stabilität erheblich beeinträchtigen und sogar zum Verlust des Gleichgewichts führen. Aus diesem Grund ist ein robustes Sensorkonzept erforderlich, das die relevanten physikalischen Größen erfasst und durch geeignete Signalverarbeitung so aufbereitet, dass sie für die Regelalgorithmen zuverlässig nutzbar sind. Hierbei spielen sowohl die Auswahl geeigneter Sensortypen als auch die Implementierung von Filter- und Fusionsverfahren eine wesentliche Rolle.

Für balancierende Fahrzeuge ist die aktuelle Lage im Raum die relevante physikalische Größe. Ausschlaggebend für das Balancieren ist die Orientierung, teilweise sind auch die translatotrischen Bewegungen relevant. Für die Bestimmung dieser Größen gibt es im Wesentlichen zwei Sensorarten, die für den Einsatz auf einem mobilen Fahrzeug infrage kommen:

- Inertial Measurement Units (IMUs): IMUs, auch Trägheitssensoren genannt, messen Beschleunigungen (Accelerometer) und Winkelgeschwindigkeiten (Gyroskop). Die Winkelgeschwindigkeiten sind damit direkt über die Sensoren messbar, während die Winkel durch Integration bzw. durch eine geeignete Kombination der Sensordaten bestimmt werden können. IMUs operieren unabhängig von den meisten Umweltfaktoren wie Licht, Schall und Untergrund, funktionieren unabhängig von ihrer Lage im Raum und können alle sechs Freiheitsgrade erfassen. Außerdem sind IMUs günstig, klein, können beliebig am Fahrzeug montiert werden und ermöglichen hohe Abtastfrequenzen von 9 kHz [15, S. 60] und mehr. Der große Nachteil dieser Sensoren ist die starke Anfälligkeit gegenüber Vibrationen.
- Optische/Akustische Sensoren: Hierzu zählen beispielsweise Abstandssensoren wie Ultraschall-, Infrarot- oder Lidar-Sensoren sowie Kamerasysteme. Sie ermöglichen die Bestimmung der absoluten Lage des Fahrzeugs über geometrische Messungen, etwa durch Abstand zum Boden oder zu Objekten. Dadurch kann die Orientierung ohne Integration oder anderweitige Algorithmen bestimmt werden. Das macht diese Sensoren auch sehr robust gegenüber Vibrationen. Nachteile optischer Sensoren sind ihre Abhängigkeit von Umgebungsbedingungen wie Untergrundbeschaffenheit oder Beleuchtung, mögliche Winkelbeschränkungen durch das Sichtfeld der Sensoren, die fehlende direkte Messung der Winkelgeschwindigkeit sowie die eingeschränkte Möglichkeit, Drehungen um die Hochachse (Gieren) zu erfassen. Kamerasysteme erfordern zusätzlich spezielle Referenzobjekte in der unmittelbaren Umgebung. Herkömmliche Abstandssensoren wie Time of Flight (ToF)-Sensoren schaffen außerdem nur niedrige Abtastfrequenzen von ca. 50 Hz, darüber hinaus beginnt die Genauigkeit stark nachzulassen [1].

Aufgrund der beschriebenen Vor- und Nachteile der beiden Sensorarten sollen IMUs verwendet werden.

5.1 Sensorfusion

IMUs messen für jede Achse die Beschleunigungen (Accelerometer) und Winkelgeschwindigkeiten (Gyroskop). Das Koordinatensystem einer IMU ist körperfest, das heißt es dreht sich mit dem Sensor und damit mit dem Fahrzeug mit, sodass die Werte direkt im Referenzkoordinatensystem des *Monowheelers* vorliegen. Die Winkelgeschwindigkeiten sind damit direkt messbar. Der Gierwinkel ist zum Balancieren nicht relevant, aber der Nick- und Rollwinkel müssen bestimmt werden. Die Winkel können auf zwei Arten bestimmt werden: geometrisch über die Richtung der Erdbeschleunigung mit den Accelerometern oder über die Integration der Gyroskope.

Da die Erdbeschleunigung im Weltkoordinatensystem immer die gleiche Orientierung hat, kann sie als Referenz für die Lagebestimmung mit den Accelerometern dienen. Die Bestimmung der Orientierung kann beispielsweise mithilfe der Euler-Winkel-Konvention und einer vollständigen 3D-Rotationsmatrix erfolgen, um freie 3D-Rotationen für alle beliebigen Winkel erfassen zu können [39, S. 6f]. Im vorliegenden System sind die Winkel konstruktiv auf $\pm 5^\circ$ für das Nicken und auf $\pm 20^\circ$ für das Rollen begrenzt, und im normalen Betrieb treten für beide Achsen lediglich Winkel von $\approx \pm 3^\circ$ auf. Für die vollständige Darstellung einer freien Rotation werden außerdem alle drei Accelerometer-Achsen miteinbezogen. In Abschnitt 5.2 wird zudem gezeigt, dass die Qualität der Sensorwerte des Accelerometers der z -Achse im relevanten Bereich äußerst schlecht ist. Daher werden die Nick- und Rollwinkel in dieser Arbeit direkt aus den x - und y -Komponenten der Accelerometer bestimmt:

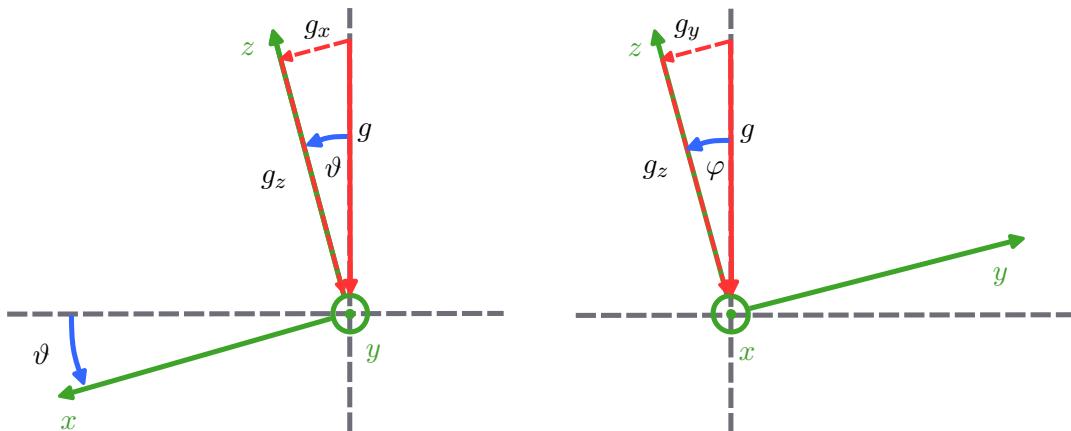


Abbildung 5.1: Bestimmung des Nickwinkels anhand der Erdbeschleunigung

Abbildung 5.2: Bestimmung des Rollwinkels anhand der Erdbeschleunigung

Aus Abbildung 5.1 folgt der Zusammenhang für die Berechnung des Nickwinkels:

$$a_x = g_x = g \cdot \sin \vartheta \quad (5.1)$$

$$\vartheta = \arcsin \frac{a_x}{g} \quad (5.2)$$

Analog dazu folgt aus Abbildung 5.2 der Zusammenhang für die Berechnung des Rollwinkels:

kels:

$$a_y = -g_y = -g \cdot \sin \varphi \quad (5.3)$$

$$\varphi = -\arcsin \frac{a_y}{g} \quad (5.4)$$

Für die Berechnung der Winkel aus den diskreten Sensordaten der Gyroskope gilt die Rechenvorschrift:

$$\vartheta = \vartheta_{old} + \dot{\vartheta} \cdot T_a \quad (5.5)$$

$$\varphi = \varphi_{old} + \dot{\varphi} \cdot T_a \quad (5.6)$$

Beide Sensortypen weisen dabei spezifische Stärken und Schwächen auf. Die Accelerometer messen nicht nur die Erdbeschleunigung, sondern auch die Beschleunigungen durch rotatorische und translatorische Bewegungen, wodurch es zu Fehlern in der Winkelschätzung kommt. Dafür liefern sie langfristig stabile Werte, da keine Integration erforderlich ist [39, S. 5]. Die Gyroskope hingegen reagieren sehr gut auf schnelle Bewegungen und erfassen die Dynamik des Fahrzeugs präzise. Dafür sind sie anfällig für Drift, da sich die Integrationsfehler nach und nach aufsummieren [39, S. 4f].

Um akkurat und robust die Orientierung zu schätzen, können die beiden Sensorarten kombiniert werden. Das Ziel dieser Sensorfusion ist es, die Stärken des jeweiligen Sensors zu nutzen, um die Schwächen des anderen auszugleichen. Die Gyroskope kompensieren die kurzfristigen, dynamischen Fehler der Accelerometer und die Accelerometer korrigieren den langfristigen Drift der Gyroskope [42]. Zur Fusion von Accelerometer- und Gyroskopdaten existieren zahlreiche Verfahren, von einfachen Komplementärfiltern über Kalman-Filter bis hin zu erweiterten Vorhersage- und Optimierungsalgorithmen wie dem Madgwick-Filter [12, S. 20ff]. Aufgrund des sehr trügen Systemverhaltens durch die hohen Trägheiten und der stark begrenzten Winkelbereiche kommt in diesem System ein einfacher Komplementärfilter zum Einsatz. Komplexere Fusionsalgorithmen wie erweiterte Kalman-Filter oder Optimierungsverfahren bieten für die vorliegenden Bedingungen keinen wesentlichen Vorteil.

Durch die Anzahl und Positionierung der IMUs können die Accelerometerdaten teilweise bereinigt werden, indem geometrische Zusammenhänge genutzt werden, um rotatorische Beschleunigungsanteile herauszurechnen. Im vorliegenden System ist dies jedoch nicht erforderlich: Aufgrund des trügen Systemverhaltens sind die durch die Fahrzeugrotation entstehenden Beschleunigungen gering. Auch die translatorischen Beschleunigungen bleiben aufgrund der langsamen Fahrmanöver klein. Sowohl die rotatorischen als auch die translatorischen Beschleunigungen, die die Winkelschätzung aus den Accelerometerdaten verfälschen, werden durch die Sensorfusion gefiltert und können daher vernachlässigt werden. Eine einzelne IMU reicht für die Bestimmung der Orientierung aus.

In Abbildung 5.3 ist die Struktur eines Komplementärfilters zu sehen:

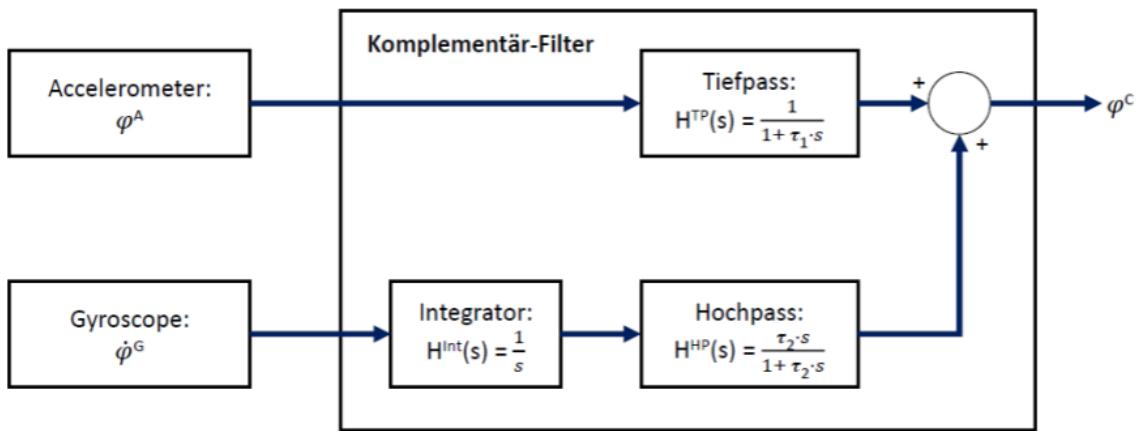


Abbildung 5.3: Struktur Komplementärfilter [42, S. 84]

Mit der Rechenvorschrift aus Gleichung 5.1 bzw. Gleichung 5.3 wird der Winkel aus den Accelerometerdaten berechnet. Dieser Winkel wird mit einem Tiefpass gefiltert, um die dynamischen Einflüsse durch rotatorische und translatorische Beschleunigungen zu filtern, sodass nur die langfristige Orientierung anhand der Erdbeschleunigung berechnet wird. Die Gyroskopdaten werden zu einem Winkel integriert und dann mit einem Hochpass gefiltert, um den Drift zu eliminieren und nur die dynamischen Bewegungen abzubilden. Diese beiden Größen werden dann zum tatsächlichen Winkel zusammengeführt. Wenn die Grenzfrequenzen für den Hochpass und Tiefpass gleich gewählt werden, ist die Gesamtübertragungsfunktion des Filters 1, weswegen keine Phasenverschiebung entsteht [42, S. 85]. Die diskrete Berechnung für den Nick- und Rollwinkel ergibt sich mit Gleichung 5.1 und Gleichung 5.3 zu [42, S. 85]:

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

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

Der Koeffizient α bestimmt dabei die Grenzfrequenz für den Hoch- und Tiefpass, und somit die Frequenzbereiche, in denen die Accelerometer- bzw. Gyroskopdaten dominieren. Als Grenzfrequenz wird $f = 0,16\text{ Hz}$ gewählt, da die Signalqualität der Gyroskope deutlich besser ist als die der Accelerometer (siehe Abschnitt 5.2) und sich damit dynamische Anteile des Signals besser erfassen lassen. Damit kann α bestimmt werden:

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

5.2 Signalcharakterisierung und Frequenzanalyse

Um die Eignung der IMU-Sensordaten für die Regelung zu bewerten, ist eine detaillierte Analyse ihrer Signalqualität erforderlich. Dabei sollen insbesondere die Signal to Noise Ratio (SNR) und die spektralen Eigenschaften der Messgrößen untersucht werden. Durch die Betrachtung im Frequenzbereich mittels Fourier-Transformation lassen sich charakteristische

Signalanteile sowie dominierende Rauschkomponenten identifizieren. Diese Analyse ermöglicht eine fundierte Einschätzung der Messqualität und liefert gleichzeitig eine Grundlage für die Auslegung geeigneter Filter, die störende Einflüsse unterdrücken.

Die IMU-Daten setzen sich aus drei Anteilen zusammen:

$$\text{data} = \text{Nutzsignal} + \text{Mechanisches Rauschen} + \text{Sensorrauschen} \quad (5.10)$$

Entscheidend für die Signalqualität ist das Verhältnis zwischen der Leistung des Nutzsignals und der Leistung des Rauschens, auch genannt SNR. Da die Leistung eines Signals proportional zu dessen Root Mean Square (RMS) im Quadrat ist, kann die SNR auch über den RMS berechnet werden. Die SNR wird typischerweise im logarithmischen Maßstab angegeben und lässt sich wie folgt für ein diskretes Signal x der Länge N bestimmen [29, S. 219]:

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

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

Es lässt sich keine grundlegende Aussage darüber treffen, ab welcher SNR ein Signal brauchbar oder gut ist, da dies je nach Anwendung variiert. Grundsätzlich ist ein hoher Wert besser und für die meisten Anwendungen ist $SNR = 10 \text{ dB}$ die untere Grenze, für sensiblere Systeme sind mitunter Werte von $SNR = 30 \text{ dB+}$ erforderlich.

5.2.1 Kalibrierung der Sensoren

Bevor die IMU verwendet werden kann, müssen die Sensoren kalibriert werden. Die tatsächlichen Werte lassen sich dabei folgendermaßen aus den Rohdaten bestimmen:

$$\text{data} = scale \cdot (\text{data}_{\text{raw}} - offset) \quad (5.13)$$

Die Skalierungsfaktoren werden dem Datenblatt des Herstellers [15] entnommen und sind ohne zusätzliche Kalibrierung genau genug. Die Offsets werden bestimmt, indem die IMU mit der xy -Ebene parallel zum Boden ausgerichtet wird und eine Messung durchgeführt wird. Für alle Signale, die als Messgröße in dieser Position null messen sollten, kann das Offset einfach aus dem Mittelwert der Messreihe bestimmt werden. Einzig das z-Accelerometer muss zusätzlich den Offset der Erdbeschleunigung berücksichtigen. Die Berechnung erfolgt im Python-Skript *calib.py*. Zusätzlich muss die Orientierung der Achsen der Sensoren an die Orientierung des Koordinatensystems des *Monowheelers* angepasst werden.

5.2.2 Experimentelle Signalanalyse

Um die Signalqualität zu analysieren, wird ein Experiment durchgeführt. Dazu wird das Fahrzeug ohne aktive Regelung von Hand in dem systemrelevanten Frequenzbereich mit systemrelevanter Amplitude angeregt und die Daten der IMU aufgezeichnet.

Die primäre Quelle für Rauschen sind die mechanischen Vibrationen durch die Schwungmasse

für den gyroskopischen Effekt. Diese hat eine konstante Drehzahl von 5000 rpm (siehe Kapitel 6). Die Frequenz dieser Störungen kann wie folgt ermittelt werden:

$$f_v = \frac{1}{60 \text{ s}} \cdot 5000 \text{ rpm} \approx 83 \text{ Hz} \quad (5.14)$$

Da diese Störung in einem viel höheren Frequenzbereich liegt als das Nutzsignal, welches unter 15 Hz liegt [33, S. 105ff], kann angenommen werden, dass die Frequenz der Störungen die höchste im Signal vorkommende Frequenz darstellt. Die in [33] gewählte IMU *ICM20948* tastet die internen ADC-Wandler der Accelerometer mit 4,5 kHz [15, S. 12] ab, die der Gyroskope mit 9 kHz [15, S. 11]. Wird kein interner Filter genutzt, werden neue Werte auch mit dieser Frequenz bereitgestellt. Das Abtasttheorem (Nyquist-Shannon) wird auf Seite der IMU also eingehalten, da die Abtastfrequenz um ein Vielfaches größer ist als die höchste im Signal vorkommende Frequenz. Da mindestens mit der doppelten Frequenz im Vergleich zur höchsten im Signal vorkommenden Frequenz abgetastet werden muss, um ein Signal fehlerfrei rekonstruieren zu können, wird die IMU mit einer Abtastfrequenz von 500 Hz abgetastet. Damit können mögliche Interferenzen bei doppelter (166 Hz) und dreifacher (249 Hz) Grundfrequenz der ursprünglichen 83 Hz erfassen werden und praktische Effekte wie nicht ideale Filter ausgeglichen werden.

Die Daten werden in einer Comma Separated Values (CSV)-Datei gespeichert und mit dem Python-Skript *imu_signal_analyzer.py* ausgewertet. Aus dem gemessenen Signal muss für die Berechnung der SNR gemäß Gleichung 5.11 das Nutzsignal und das Rauschen extrahiert werden. Dazu wird das gemessene Signal mit einer Fast Fourier Transformation (FFT) gefiltert. Nach der Transformation in den Frequenzbereich werden alle Frequenzanteile oberhalb der Cutoff-Frequenz eliminiert. Die Rücktransformation in den Zeitbereich liefert dann das Nutzsignal. Listing 1 zeigt die Implementierung des FFT-Filters:

```

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 Tiefpassfilter

Die Cutoff-Frequenz orientiert sich an den relevanten Frequenzen der Signale. Für die Accelerometerdaten wird eine niedrigere Cutoff-Frequenz von $f_{cutoff} = 2 \text{ Hz}$ gewählt, da im Prinzip nur die Ausrichtung zur Erdbeschleunigung relevant ist (siehe Abschnitt 5.1). Für die Gyroskopdaten wird eine höhere Cutoff-Frequenz von $f_{cutoff} = 10 \text{ Hz}$ gewählt, da die Gyroskope die dynamischen Änderungen erfassen sollen (siehe Abschnitt 5.1). Damit lässt sich dann das Rauschen berechnen, indem das Nutzsignal vom gemessenen Signal abgezogen wird. Anschließend kann der RMS des Signals und des Rauschens nach Gleichung 5.11 berechnet werden. Davor wird noch der Gleichanteil der Signale eliminiert, um eine Verfälschung durch

statische Offsets zu vermeiden, da nur die zeitlichen Änderungen des Signals interessant sind. Damit lässt sich die SNR nach Gleichung 5.11 bestimmen (siehe Listing 2):

```

1 def calc_RMS(signal):
2     signal = signal - np.mean(signal)
3     return np.sqrt(np.mean(signal**2))
4
5 def calc_SNR(signal_reference, signal_raw):
6     rms_signal = calc_RMS(signal_reference)
7     noise = signal_raw - signal_reference
8     rms_noise = calc_RMS(noise)
9     SNR = 20*np.log10(rms_signal/rms_noise)
10    return SNR

```

Listing 2: SNR Bestimmung in Python

Im Abbildung 5.4 ist ein Ausschnitt aus dem Experiment zu sehen. Bereits aus dieser Ansicht wird deutlich, dass die Signalqualität schlecht ist. Bei den Accelerometerdaten sind die Bewegungen des Fahrzeugs selbst im Referenz-Signal kaum zu sehen und das Rauschen beträgt teilweise bis zu $\pm g$, was einer Winkelabweichung von $\pm 90^\circ$ entspricht. Die Gyroskopdaten sind etwas besser, aber auch hier überwiegt das Rauschen. Dies spiegelt sich auch in den SNR-Werten wider, die in Tabelle 5.1 zu sehen sind. Werte um $SNR = -20$ dB für die Accelerometerdaten und $SNR = 0$ dB für die Gyroskopdaten zeigen, dass die Signale ohne Filter unbrauchbar sind.

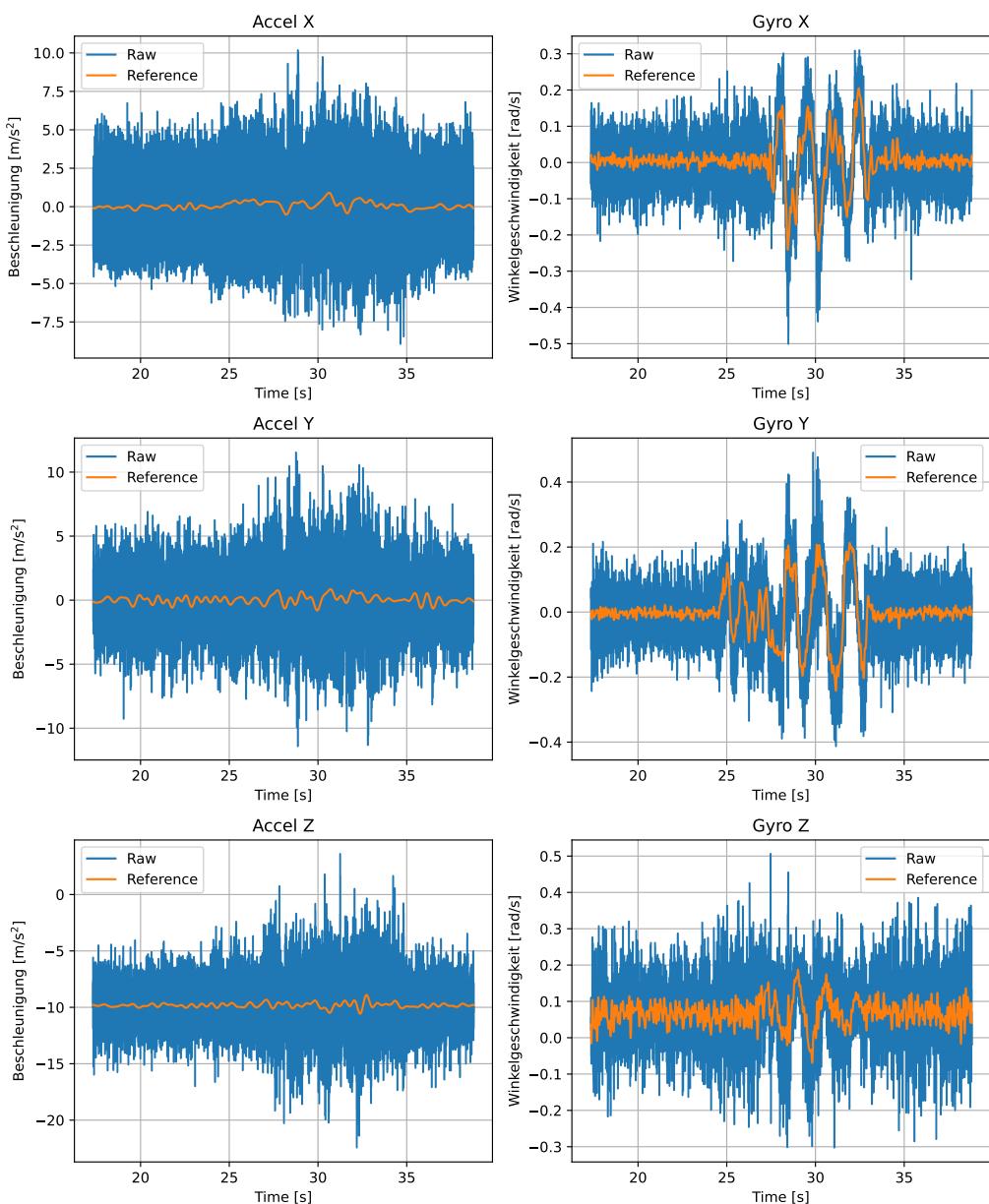


Abbildung 5.4: IMU Daten Raw

Um die Art der Störung zu identifizieren und um Informationen für das Filterdesign zu gewinnen, wird eine Frequenzanalyse durchgeführt. Die Berechnung erfolgt ebenfalls im Python-Skript *imu_signal_analyzer.py* (siehe Listing 3):

```

1 def calc_fft_norm(signal, sampling_rate):
2     n = len(signal)
3     signal = signal - np.mean(signal)
4     fft_vals = np.fft.fft(signal)
5     fft_abs = np.abs(fft_vals)
6     fft_abs = np.abs(fft_vals) / n
7     fft_abs = fft_abs[:n//2]
8     fft_abs[1:-1] *= 2
9     frqs = np.linspace(start=0, stop=sampling_rate, num=n+1)
10    frqs = frqs[:n//2]
11    return frqs, fft_abs

```

Listing 3: FFT Analyse in Python

Die Ergebnisse der FFT-Analyse sind in Abbildung 5.5 zu sehen:

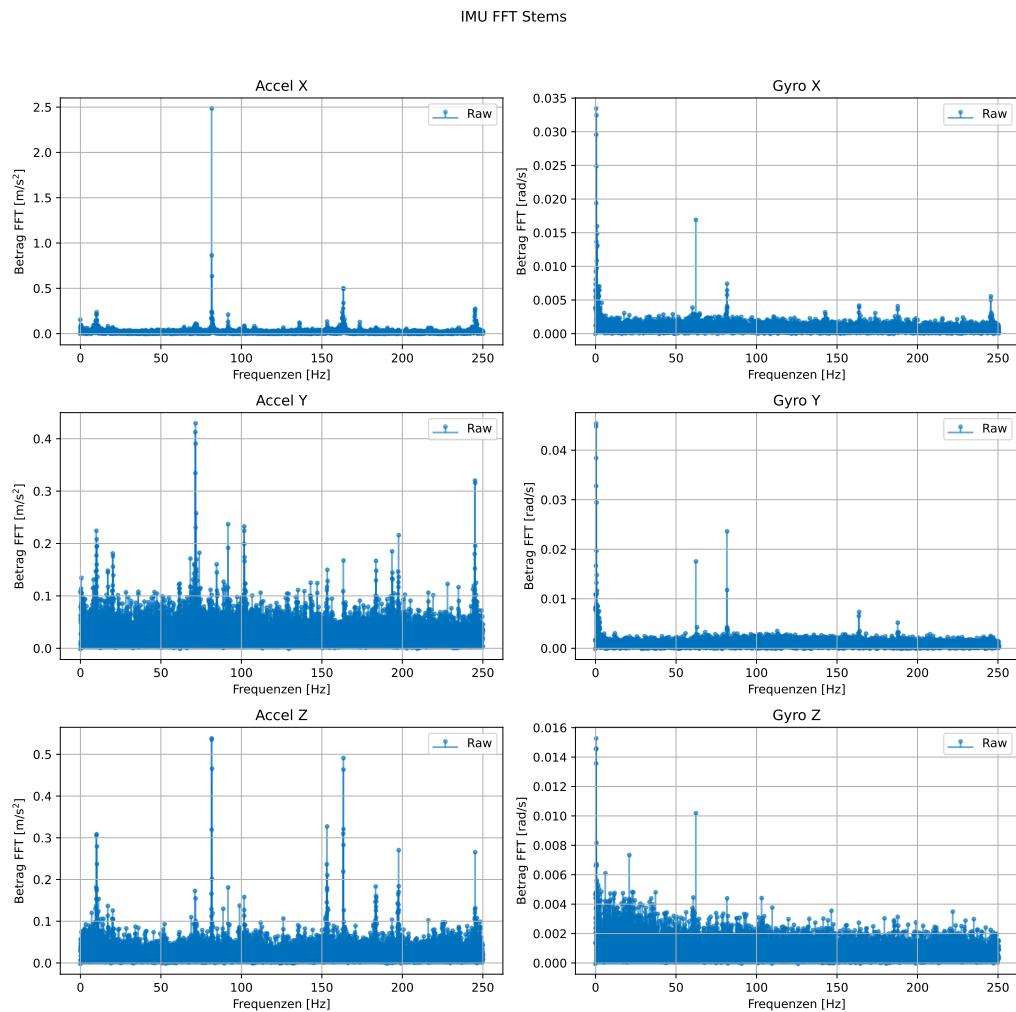


Abbildung 5.5: IMU Daten Raw FFT

Damit wird die Vermutung, dass die Hauptursache der Störungen der Kreisel ist, bestätigt. Sowohl bei 83 Hz, 163 Hz und 249 Hz sind in allen Sensordaten große Ausschläge zu erkennen, was genau den Vielfachen der Frequenz des Kreisels entspricht. Besonders stark ist dieses Phänomen in den Accelerometerdaten der *x*-Achse zu sehen. Es wird auch ersichtlich, dass die Gyroskopdaten erheblich besser sind, da trotz der Störungen ein erheblicher Ausschlag bei den relevanten Frequenzen (<10 Hz) zu verzeichnen ist, während bei den Accelerometerdaten die Störungen deutlich überwiegen. Bei den Accelerometerdaten ist jeweils ein weiterer Ausschlag bei 10 Hz zu sehen. Da dies nur in allen drei Accelerometern beobachtbar ist, ist davon auszugehen, dass es sich um eine sensorspezifische Störung handelt.

5.2.3 Signalverbesserung durch Filterung auf Imu-Hardware

Aus der FFT-Analyse (siehe Abbildung 5.5) geht hervor, dass sich der Großteil des Rauschens in einem deutlich höheren Frequenzbereich abspielt als das Nutzsignal. Die Signalqualität kann daher erheblich durch den Einsatz eines Tiefpassfilters gesteigert werden, das die unerwünschten Frequenzen unterdrückt. Ein Nachteil von Tiefpassfiltern ist die Phasenverschiebung der relevanten Frequenzanteile. Diese sollte minimal gehalten werden, um die Phasenreserve des Gesamtsystems zu erhalten. Es sollte also eine Grenzfrequenz gewählt werden, die so niedrig wie nötig aber so hoch wie möglich ist.

Die *ICM20948* IMUs bieten die Möglichkeit, einen eingebauten Digital Lowpass (DLP) mit variabler Grenzfrequenz zu aktivieren. Das hat den großen Vorteil, dass ein Teil der Signalverarbeitung schon auf der IMU vorgenommen wird, von einem Filter, der genau auf die Hardware abgestimmt ist. Außerdem wirkt der Filter auf der IMU wie ein Anti-Aliasing-Filter, und die Abtastfrequenz des Mikrocontrollers kann verringert werden, da die hohen Frequenzen bereits aus dem Signal gefiltert werden. Der Nachteil ist, dass dadurch die Frequenz, mit der die IMU Daten bereitstellt, auf 1125 Hz sinkt [15, S. 60, S.64]. Werden die Clocks des Mikrocontrollers und der IMU nicht synchronisiert, kommt es damit zu Fehlern beim Einhalten der Abtastzeit. Diese Fehler sind aber aus den folgenden Gründen klein und werden in Kauf genommen: diese Fehler wiegen besonders schwer, wenn die Daten abgeleitet werden sollen. Dies ist nicht der Fall, da die Daten entweder direkt verwendet oder sogar integriert werden. Zusätzlich ist die Frequenz der IMU immer noch deutlich höher als die schlussendlich gewählte Abtastfrequenz des Mikrocontrollers (siehe Kapitel 8). Die Systemfrequenz des mechanischen Aufbaus ist zudem aufgrund der hohen Trägheiten durch das hohe Gewicht (14 kg) im Vergleich zu den Abtastfrequenzen so gering, dass die Hardware wie ein starker Tiefpassfilter wirkt, in dem solche hochfrequenten Änderungen untergehen. Dies bestätigt sich bei Frequenzanalysen der Fahrzeugdaten (siehe Abbildung 6.7), bei denen keine Frequenzen über 7 Hz in der Systemantwort auf verschiedene Anregungen auftreten.

Um die Signalqualität mit aktivierten DLP der IMU zu testen, werden die drei relevanten, einstellbaren Grenzfrequenzen 5 Hz, 11 Hz und 23 Hz getestet. Dazu werden zwei IMUs angeschlossen, um einen Vergleich zwischen dem ungefilterten Signal und dem gefilterten Signal zu erhalten. Der Versuchsablauf bleibt ansonsten identisch. Das Referenz-Signal wird nun mit einem FFT-Tiefpass aus den gefilterten IMU Daten gewonnen. Dies ist notwendig, weil selbst das FFT-gefilterte Signal aus den Rohwerten im Vergleich zu den DLP-gefilterten zu schlecht ist, um einen objektiven Vergleich zu ermöglichen. Allerdings muss beachtet werden, dass das Referenz-Signal durch den DLP nicht mehr alle relevanten Frequenzanteile

enthält. Daher muss die Wahl der DLP-Grenzfrequenz an den erwarteten Frequenzbereich angepasst werden und darf nicht nur aufgrund der SNR-Werte erfolgen.

Beispielhaft sind in Abbildung 5.6 und Abbildung 5.7 die Ergebnisse des Experiments bei einer IMU-DLP Grenzfrequenz von $f_G = 11$ Hz zu sehen. Die Ergebnisse für $f_G = 5$ Hz und $f_G = 23$ Hz sind in Anhang B zu finden.

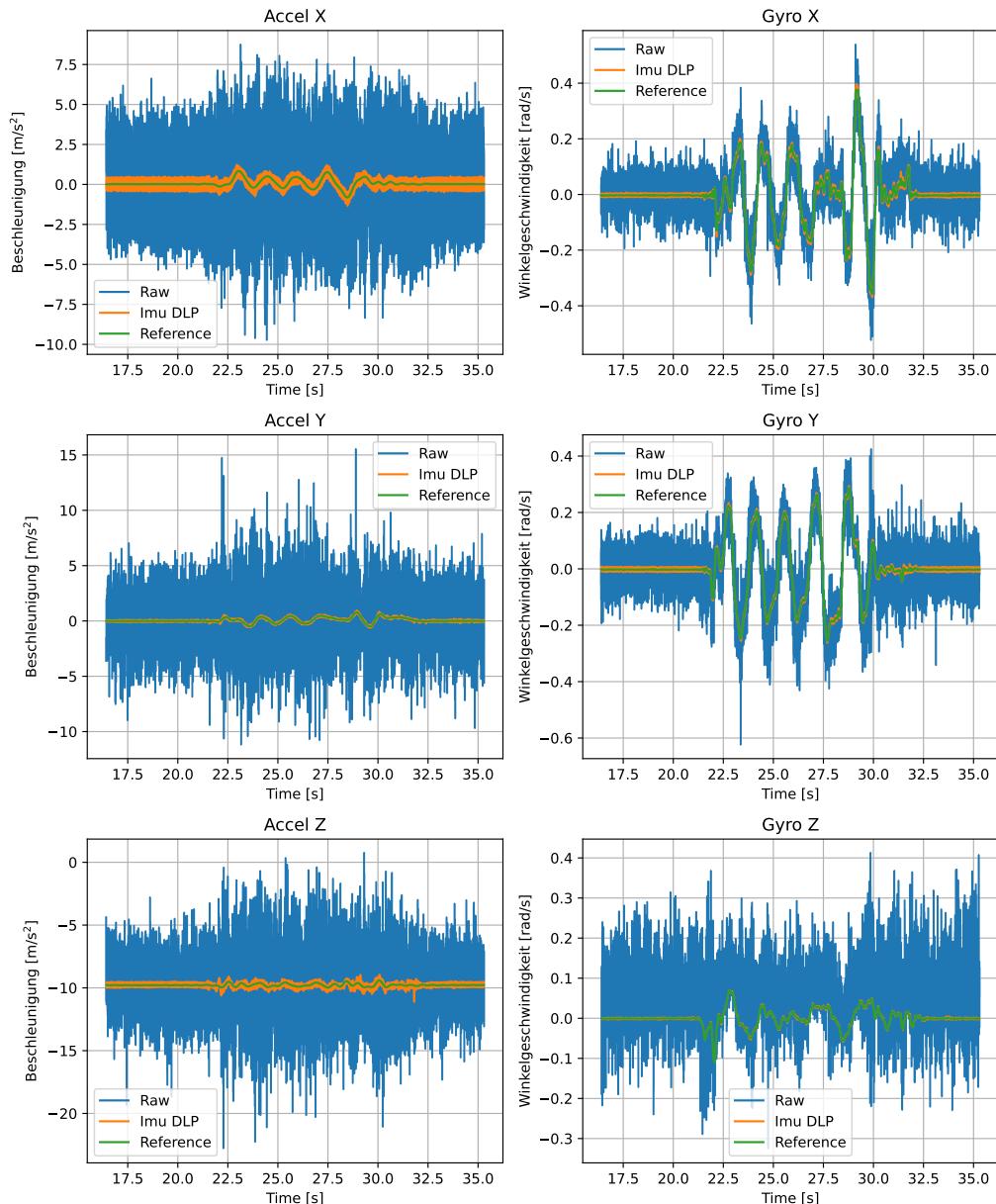


Abbildung 5.6: IMU Daten mit IMU-DLP mit $f_G = 11$ Hz

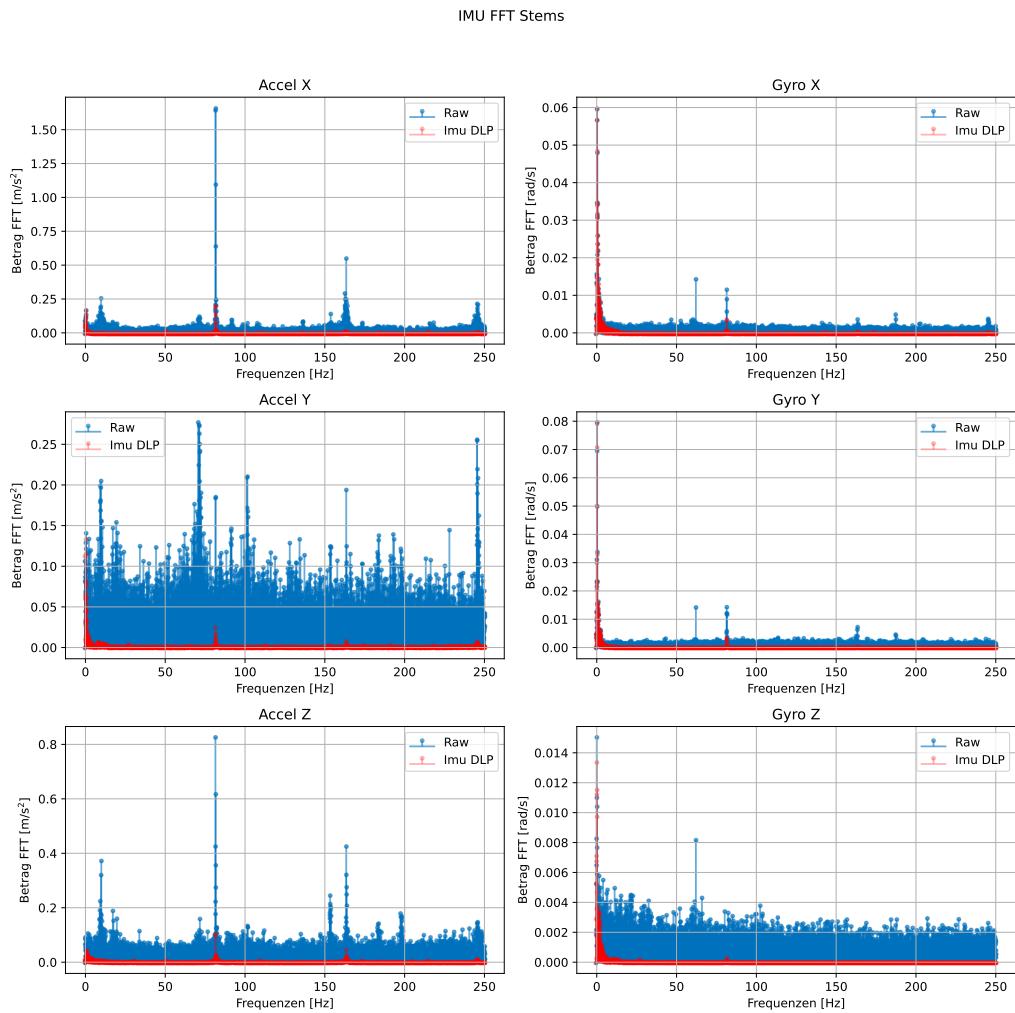


Abbildung 5.7: IMU Daten mit IMU-DLP mit $f_G = 11 \text{ Hz}$ FFT

Der Vergleich der Rohdaten und der gefilterten Daten (siehe Abbildung 5.6) zeigt, wie viel besser die Signalqualität durch das Filter wird. Der Vergleich der Frequenzspektren in Abbildung 5.7 bestätigt dies, die hochfrequenten Rauschanteile sind in allen Signalen um ein Vielfaches gedämpft. Diese Verbesserung spiegelt sich auch in den SNR-Werten wieder (siehe Tabelle 5.1). Besonders die Gyroskopdaten haben jetzt ausgezeichnete SNR-Werte um die 25 dB. Auch die Signalqualität der Accelerometer hat sich stark verbessert, ist aber trotzdem noch nicht ausreichend. Die SNR-Werte werden bei niedrigeren Grenzfrequenzen des DLPs immer besser. Da die Accelerometer nur Frequenzen unter 1 Hz erfassen sollen, kann ohne Probleme der IMU-DLP mit $f_G = 5 \text{ Hz}$ zugunsten der Signalqualität gewählt werden. Für die Gyroskope ergibt sich für den IMU-DLP mit $f_G = 11 \text{ Hz}$ FFT der beste Kompromiss aus SNR-Wert, dem Durchlassen relevanter Frequenzanteile und möglichst geringer Phasenverschiebung.

5.2.4 Signalverbesserung durch externen Tiefpassfilter

Trotz des Einsatzes des IMU-DLPs ist die Signalqualität der Accelerometer noch nicht ausreichend. Das ergibt sich aus den niedrigen SNR-Werten (siehe Tabelle 5.1). Außerdem ist in der Frequenzanalyse der Accelerometer und Gyroskope in Abbildung B.2 und Abbildung 5.7 zu sehen, dass bei 83 Hz immer noch Rauschen vorliegt. Daher wird ein zusätzlicher DLP auf dem Mikrocontroller implementiert.

Der DLP soll eine möglichst monotone Amplitudencharakteristik aufweisen, um das Signal nicht zu verzerren, was sich negativ auf die Regelung auswirken kann. Zudem soll die Phasenverschiebung minimal gehalten werden. Für die Filterung kommen grundsätzlich verschiedene Tiefpassfilter infrage:

- Butterworth-Filter: zeichnet sich durch eine monotone Amplitudencharakteristik im Durchlassbereich aus, bei moderater Phasenverschiebung und guter Dämpfung [26, S. 131] [29, S. 508f].
- Bessel-Filter: bietet eine nahezu lineare Phasencharakteristik, wodurch Signalverzerrungen minimal bleiben, hat aber eine weniger steile Dämpfung im Sperrbereich [26, S. 136].
- Chebyshev-Filter: ermöglicht eine steilere Flankensteilheit, geht dafür aber mit Welligkeiten im Durchlassbereich einher, die das Signal verzerren können [29, S. 508f].

Als Kompromiss zwischen Dämpfung und monotoner Amplitudencharakteristik fällt die Wahl auf einen Butterworth-Filter zweiter Ordnung. Die Auslegung erfolgt mit dem Python-Skript *butterworth_filter_designer.py*. Die Grenzfrequenz von $f_G = 40$ Hz wird dabei so gewählt, dass die störende Frequenz 83 Hz mit einer Verstärkung von -20 dB abgeschwächt wird. Das Bode-Diagramm des Filters ist in Abbildung 5.8 zu sehen:

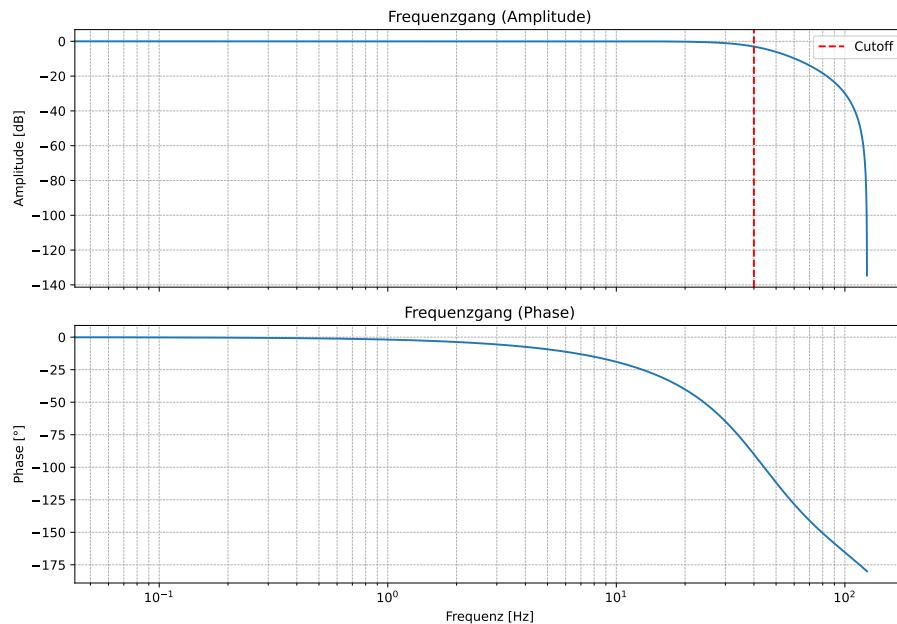


Abbildung 5.8: Bode-Diagramm Butterworth-Tiefpass 2. Ordnung, $f_G = 40$ Hz

In einem weiteren Experiment wird die Effektivität des Filters getestet. Das Referenz-Signal ist dabei wieder das FFT-gefilterte IMU-DLP Signal. In Abbildung 5.9 und Abbildung 5.10 sind die Ergebnisse beispielhaft für den IMU-DLP mit $f_G = 5$ Hz gezeigt. Die Ergebnisse für den IMU-DLP mit $f_G = 11$ Hz sind in Abbildung B.5 und Abbildung B.6 gezeigt.

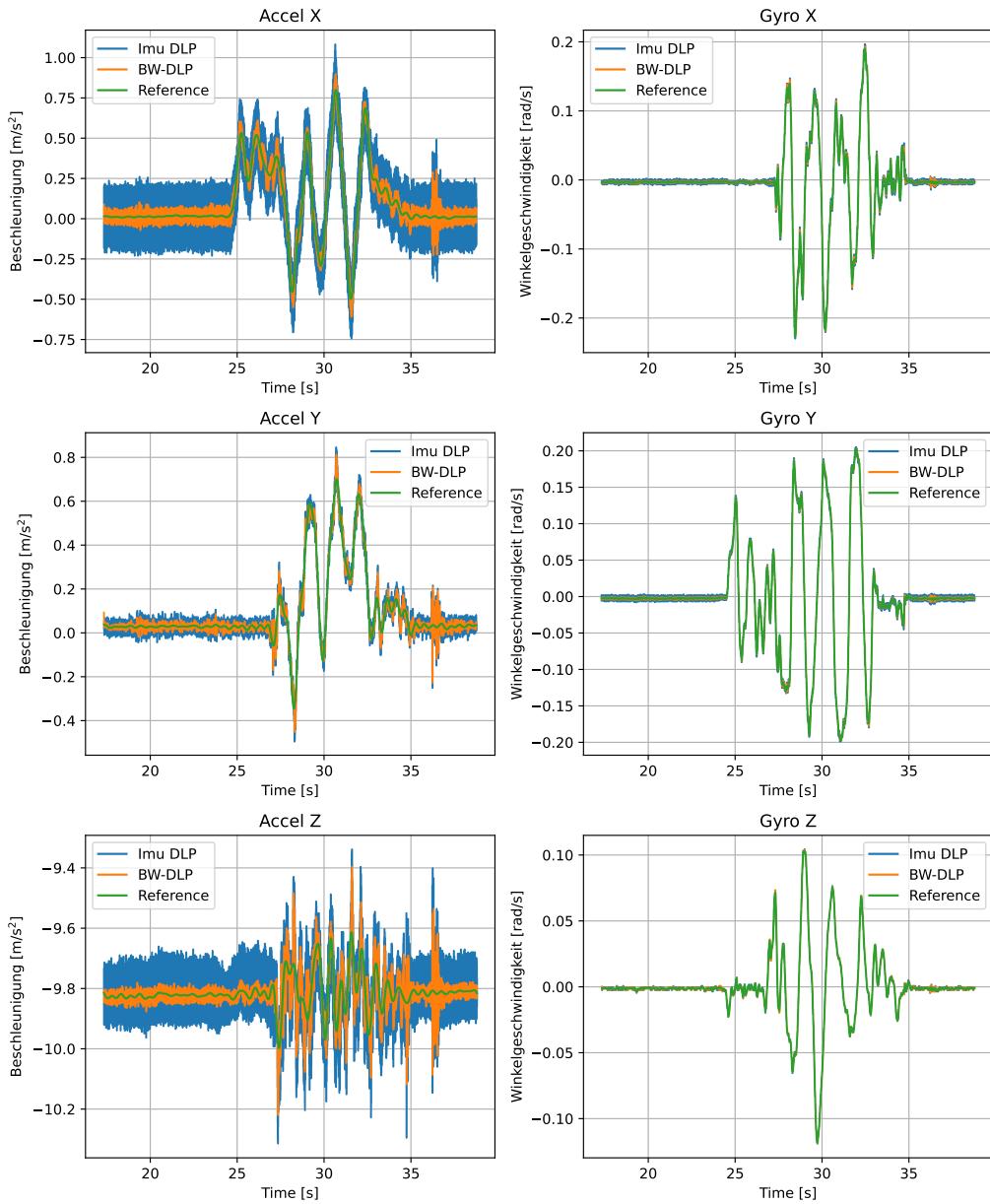


Abbildung 5.9: IMU Daten mit IMU-DLP mit $f_G = 5$ Hz und Butterworth-Tiefpass mit $f_G = 40$ Hz

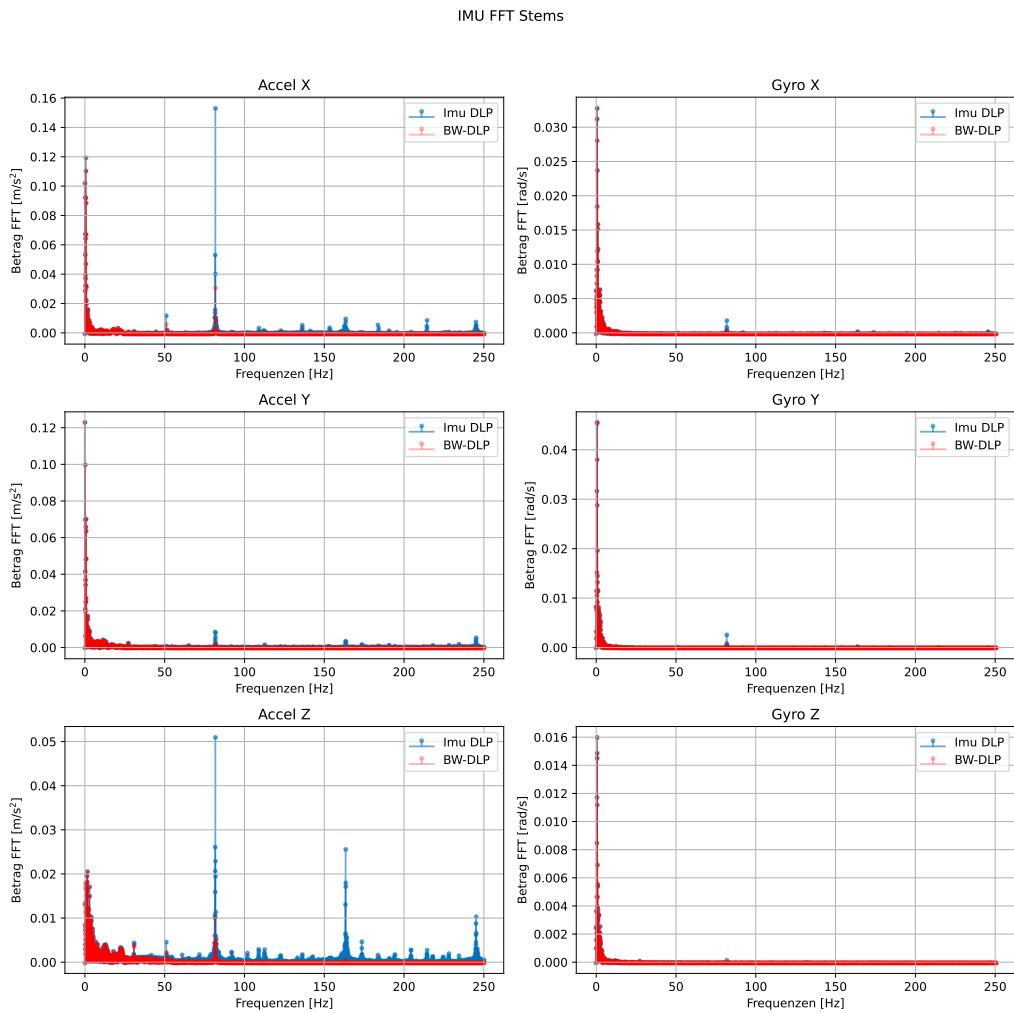


Abbildung 5.10: IMU Daten mit IMU-DLP mit $f_G = 5$ Hz und Butterworth-Tiefpass mit $f_G = 40$ Hz FFT

In Abbildung 5.9 ist eindeutig erkennbar, dass der Butterworth-Tiefpassfilter das Rauschen stark dämpft. Das ist auch in den Spektren der Signale in Abbildung 5.10 zu sehen, die Frequenzanteile des Rauschens sind deutlich geringer. Das spiegelt sich auch in stark verbesserten SNR-Werten wieder (siehe Tabelle 5.1). Besonders die Signalqualität der Accelerometer konnte erheblich verbessert werden und befindet sich mit Werten um $SNR = 14$ dB jetzt in einem guten Bereich.

5.2.5 Übersicht Signalqualität

Die Tabelle 5.1 zeigt die Ergebnisse der SNR-Werte aller Tests:

Tabelle 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

Die unzureichende Signalqualität der IMU konnte mit zusätzlich Maßnahmen, dem Einsatz des IMU-integrierten DLPs und einem zusätzlichen Butterworth-Tiefpassfilter, maßgeblich verbessert werden. Die SNR-Werte aller Sensoren konnten um ein Vielfaches gesteigert werden und befinden sich im guten und sehr guten Bereich. Auffällig ist aber, dass die SNR-Werte des Accelerometers der *z*-Achse trotz der Maßnahmen schlecht sind. Der *z*-Anteil der Erdbeschleunigung berechnet sich durch den \cos der Winkel ϑ und φ , während der *x*- und *y*-Anteil jeweils über den \sin des jeweiligen Winkels bestimmt wird. Für sehr kleine Winkel hat der \sin eine große Steigung, und somit ist die Änderung der *x*- und *y*-Komponente groß. Der \cos hat für kleine Winkel im Prinzip keine Steigung, sodass kleine Änderungen keine ausschlaggebende Änderung des Signals mit sich führen. Daher ist das Nutzsignal des Accelerometer der *z*-Achse sehr viel kleiner und geht im Rauschen unter. Für die Bestimmung der Orientierung des Fahrzeugs ist es daher vorteilhaft, nur die Daten der *x*- und *y*-Accelerometer zu verwenden.

5.2.6 Validierung Sensorkonzept und Signalverarbeitung

Nach der Erarbeitung eines geeigneten Sensorkonzepts und der Implementierung verschiedener Maßnahmen, um die Signalqualität sicherzustellen, wird ein Test des Gesamtsystems durchgeführt. Dazu wird das Fahrzeug wieder von Hand in den relevanten Frequenz- und Winkelbereichen angeregt. Dabei werden sowohl die IMU-Daten als auch die Orientierung des Fahrzeugs aufgezeichnet. Im Post-Processing kann dann die Berechnung der Orientierung des Mikrocontrollers mit der Offline-Berechnung eines Python-Skripts (*imu_state_estimation.py*) verglichen werden. Die Ergebnisse sind in Abbildung 5.11 zu sehen:

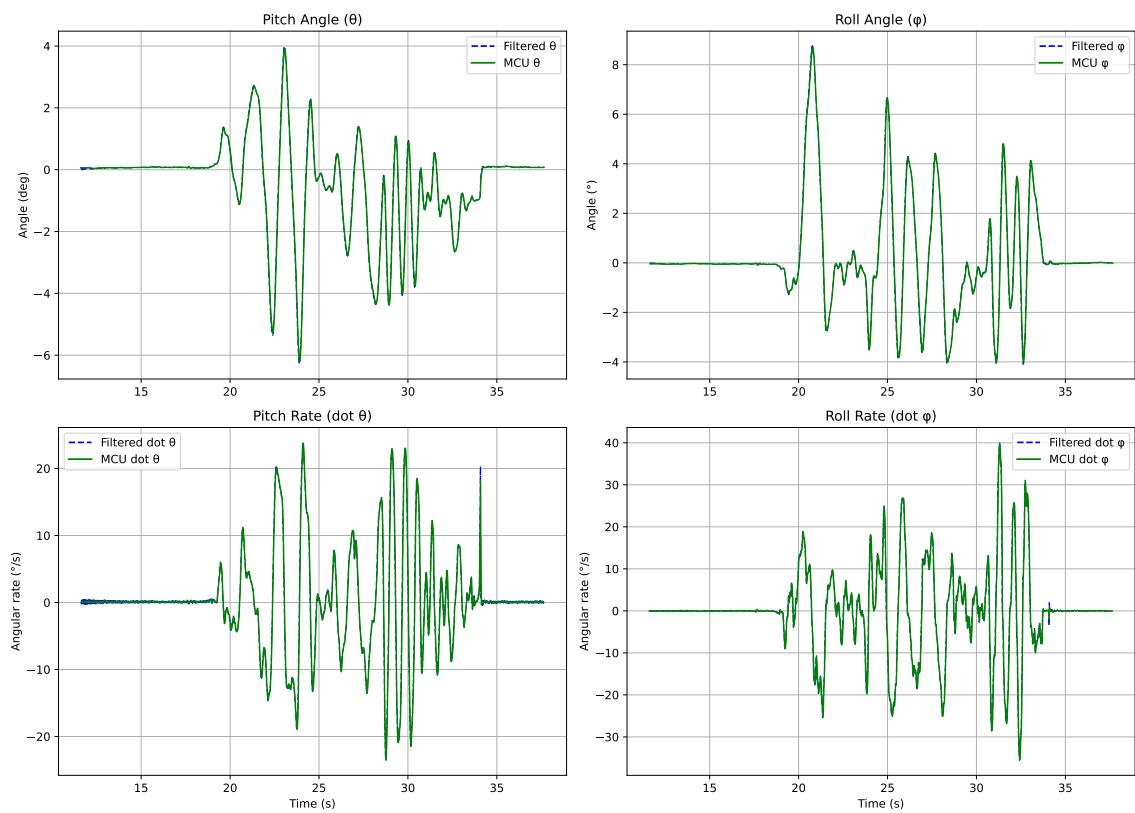


Abbildung 5.11: Validierung Sensorkonzept und Signalverarbeitung

Der Test zeigt, dass die Bewegungen des Fahrzeugs präzise erfasst werden. Im Stillstand gibt es keinen Drift und alle Bewegungen werden akkurat aufgezeichnet bei minimalem Rauschen. Die Berechnung der Orientierung auf dem Mikrocontroller deckt sich genau mit der Berechnung im Post-Processing.

6 Iterative Reglerentwicklung zum Balancieren

Um die in Kapitel 3 beschriebenen instabilen Teilsysteme zu stabilisieren und damit das Fahrzeug zum Balancieren zu bringen, ist eine aktive Regelung erforderlich. In [33] erfolgte bereits eine theoretische und praktische Auslegung eines PID-Reglers für das Teilsystem Nicken auf Basis einer Simulation und eines ersten Prototyps. Für das Teilsystem Rollen und Gieren wurde ergänzend eine theoretische Auslegung eines PID-Reglers entwickelt.

Aufbauend auf diesen Ergebnissen wird in diesem Kapitel die Regelung für das fertige Fahrzeug entwickelt. Dazu werden zunächst Regler anhand des Modells entworfen und deren Eigenschaften in einer Simulation überprüft. Die Simulationsergebnisse dienen als Grundlage für Tests am realen System. Die Erkenntnisse aus den realen Versuchen fließen zurück, um das Modell und die Simulation an das reale System anzupassen oder die Reglerparameter gezielt zu verändern. Auf diese Weise entsteht ein iterativer Entwicklungsprozess, der eine effiziente Abstimmung von Reglerparametern auf die tatsächliche Systemdynamik ermöglicht.

6.1 Abtastfrequenz

Da der Regelkreis auf einem digitalen System umgesetzt wird, es sich aber bei dem realen Fahrzeug um ein kontinuierliches System handelt, muss eine geeignete Abtastfrequenz gewählt werden. Dabei sind zwei Faktoren relevant: Welche Abtastung ist mindestens notwendig, um die physikalischen Signale zu diskretisieren (inklusive Rauschen und Störungen) und welche Abtastung ist notwendig, um einen robusten Regelkreis implementieren zu können. Wie in Kapitel 5 gezeigt, beträgt die höchste Frequenz des vom Mikrocontroller erfassten IMU-Signals 83 Hz, was dem Rauschen des Kreisels entspricht. Dem Abtasttheorem zufolge ergibt sich also die Mindestanforderung (die reale Abtastfrequenz sollte dabei etwas höher gewählt werden, da das Abtasttheorem von idealen Filtern ausgeht):

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

Um einen stabilen Regelkreis zu garantieren, ist die Abtastfrequenz aber ca. fünf- bis zehnmal [9, S. 318], bzw. zwanzigmal [8, Ch. 11] so hoch wie die Bandbreite des Systems zu wählen. Zwar garantiert das Abtasttheorem die fehlerfreie Rekonstruktion, allerdings werden die Zustandsänderungen (Ableitungen) meist nicht ausreichend fein aufgelöst und weisen starke Unstetigkeiten auf. Diese Größen sind aber oft Eingangsgrößen für Aktoren, die solche Sprünge nicht exakt abbilden können. Das führt zu ruckartigen Bewegungen, Überschwingern, regt Schwingungen an und mindert die Stabilität des Regelkreises [8, S. 451f]. Außerdem wird das Störverhalten des Regelkreises bei Abtastfrequenzen niedriger als der zehnfachen Bandbreite des Systems signifikant schlechter [8, S. 451f]. Zusätzlich wirkt die Diskretisierung wie ein Zero-Order-Hold Glied und verursacht damit eine Totzeit, die

die Phasenreserve des Systems verringert. Der Verlust der Phasenreserve θ lässt sich wie folgt abschätzen [8, S. 63]:

$$\theta = -\frac{\omega \cdot T}{2} \quad (6.2)$$

Dabei beschreibt ω die Frequenz, bei der die Amplitudenverstärkung des offenen Systems 0 dB beträgt. Diese Frequenz lässt sich grob mit der Bandbreite des Systems annähern, die beim *Monowheeler* ungefähr bei 10 Hz liegt. Damit ergibt sich bei einer Abtastfrequenz von 50 Hz folgendes:

$$\theta = -\frac{10 \cdot 2 \cdot \pi \cdot \frac{1}{50 \text{Hz}}}{2} = -36^\circ \quad (6.3)$$

Bei 200 Hz-Abtastfrequenz ergibt sich bereits:

$$\theta = -\frac{10 \cdot 2 \cdot \pi \cdot \frac{1}{200 \text{Hz}}}{2} = -9^\circ \quad (6.4)$$

Eine hohe Abtastfrequenz wirkt sich also positiv auf den Regelkreis aus. Es wird daher eine hohe, aber gut auf dem Mikrocontroller realisierbare Abtastfrequenz von $f_a = 250$ Hz gewählt, was einer Abtastzeit von $T_a = 4$ ms entspricht.

6.2 Teilsystem Nicken

Da bereits in [33] eine erfolgreiche Regelung für einen Prototyp implementiert wurde, wird dies als Ausgangspunkt genutzt. Die Parameter des in [33] genutzten Prototyps unterscheiden sich jedoch maßgeblich von dem fertigen Fahrzeug. Besonders wichtig ist zudem, dass der Prototyp nicht über den Kreisel verfügte, sodass die IMU-Daten sehr viel weniger verrauscht waren. Dadurch konnte auf zusätzliche Filter verzichtet werden, was sich aufgrund besserer Phaseneigenschaften positiv auf die Regelung auswirkt. Das ist besonders von Bedeutung, da in [33, S. 106] dokumentiert ist, dass das System sehr empfindlich auf Phasenverschiebung reagiert.

Bei der Auslegung des Reglers sind besondere Anforderungen zu beachten. Bei der Modellierung in Kapitel 3 werden dynamische Effekte (z.B. das Reaktionsmoment des Aktors) des Verschiebevorgangs im Teilsystem Nicken vernachlässigt. Auch im Teilsystem Rollen und Gieren führt die Nickbewegung je nach Auslenkung des Kreisels zu störenden Kreiselmomenten. Hinzu kommt, dass Beschleunigungen der Plattform die IMU-Signale verschlechtern (siehe Kapitel 5). Um diese negativen Effekte zu minimieren, soll das Systemverhalten zwar ausreichend schnell sein, um robust alle Störungen ausgleichen zu können, aber gleichzeitig abrupte Beschleunigungen vermeiden. Klassische Verfahren wie Ziegler-Nichols zur Reglerauslegung sind auf aggressivere Reglerabstimmungen ausgelegt [11] und für diese Anwendung daher nur begrenzt nutzbar.

Die durchgeführten Tests folgen alle dem gleichen Schema: dem Aufrichten zu Beginn des Experiments, anschließend zwei kurzen Störungen in entgegengesetzte Richtung und einer dauerhaften Störung am Ende. Dabei ist zu beachten, dass zu Beginn der Experimente die Filter für die Berechnung der Orientierung nicht initialisiert sind. Dadurch nähert sich der gemessene Winkel ϑ am Anfang langsam dem tatsächlichen Winkel an, da die Korrektur nur über die trügen Accelerometer erfolgt. Das wirkt wie eine Art Trajektorie und führt den

Regler sanft zu $\vartheta = 0^\circ$ hin. In späteren Tests wird eine Initialisierung der Filter vor Start der Regelung eingeführt.

6.2.1 PID-Regler

Der ursprüngliche PID-Regler in [33] ist in der Form

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

implementiert. Um die einzelnen Anteile unabhängig voneinander verändern zu können, wird nun auf folgende Form gewechselt:

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

Verbesserung der System-Dynamik

Als erstes werden die Reglerparameter des robustesten Reglers aus [33] getestet. Damit ergibt sich mit $KP = 0,3$; $KD = 0,045$; $KI = 0,3$ folgendes Systemverhalten (siehe Abbildung 6.1):

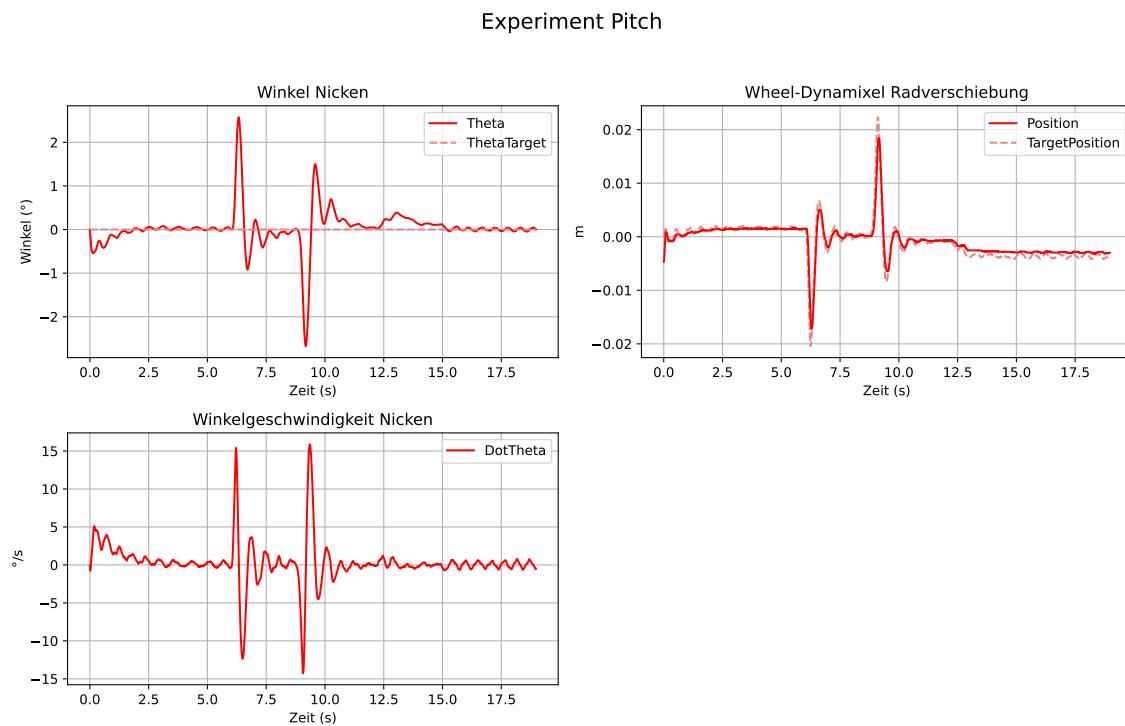


Abbildung 6.1: Pitch Experiment PID-Regler mit $KP = 0,3$; $KD = 0,045$; $KI = 0,3$

Dabei ist zu erkennen, dass sowohl die dynamischen als auch dauerhaften Störungen ohne Probleme ausgeglichen werden können und das System stabil ist. Allerdings verursachen

die Anregungen eine gedämpfte Schwingung, was darauf schließen lässt, dass die Dämpfung des geschlossenen Systems nicht groß genug ist. Um die Schwingung zu reduzieren, wird die Dämpfung erhöht, sodass sich folgende Reglerparameter ergeben: $KP = 0,3$; $KD = 0,1$; $KI = 0,3$. Das Verhalten des Systems ist in Abbildung 6.2 zu sehen:

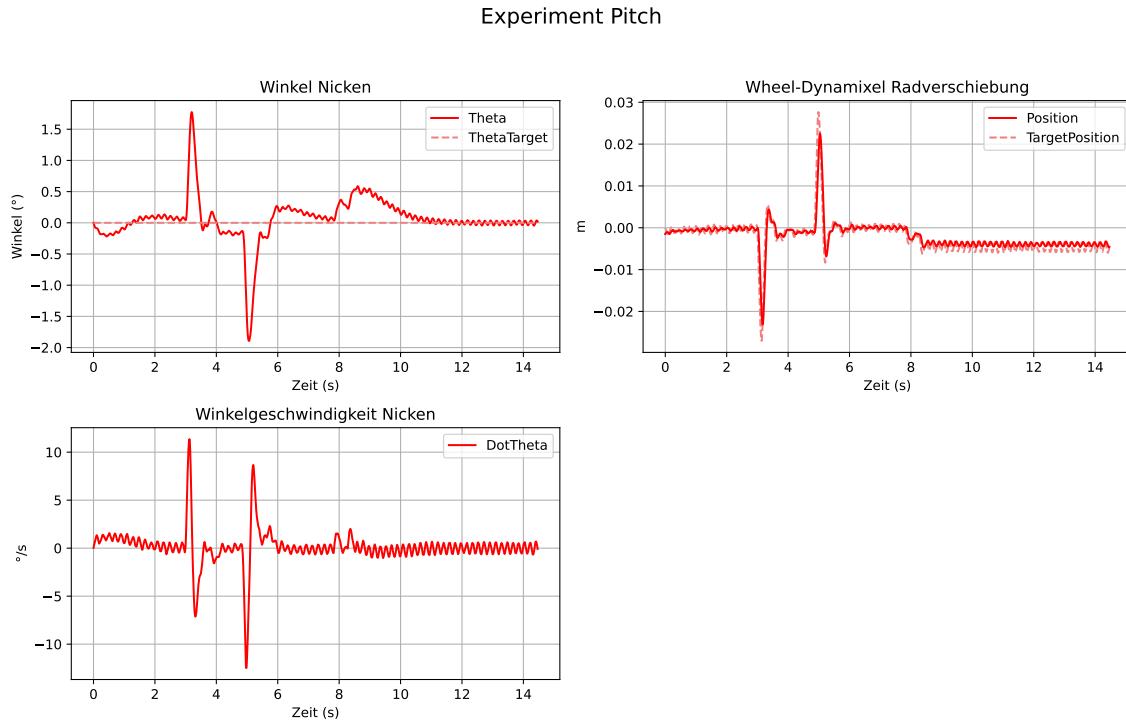


Abbildung 6.2: Pitch Experiment PID-Regler mit $KP = 0,3$; $KD = 0,1$; $KI = 0,3$

Das dynamische Verhalten hat sich eindeutig verbessert. Die gedämpfte Schwingung nach der Anregung ist einem einfachen Nachschwinger gewichen, dessen Amplitude um ein Vielfaches kleiner ist als die der ursprünglichen Schwingung. Ein weiteres Erhöhen der Dämpfung KD oder des Proportional-Anteils KP verschlechtert die dynamische Reaktion des Systems (siehe Abbildung 6.3).

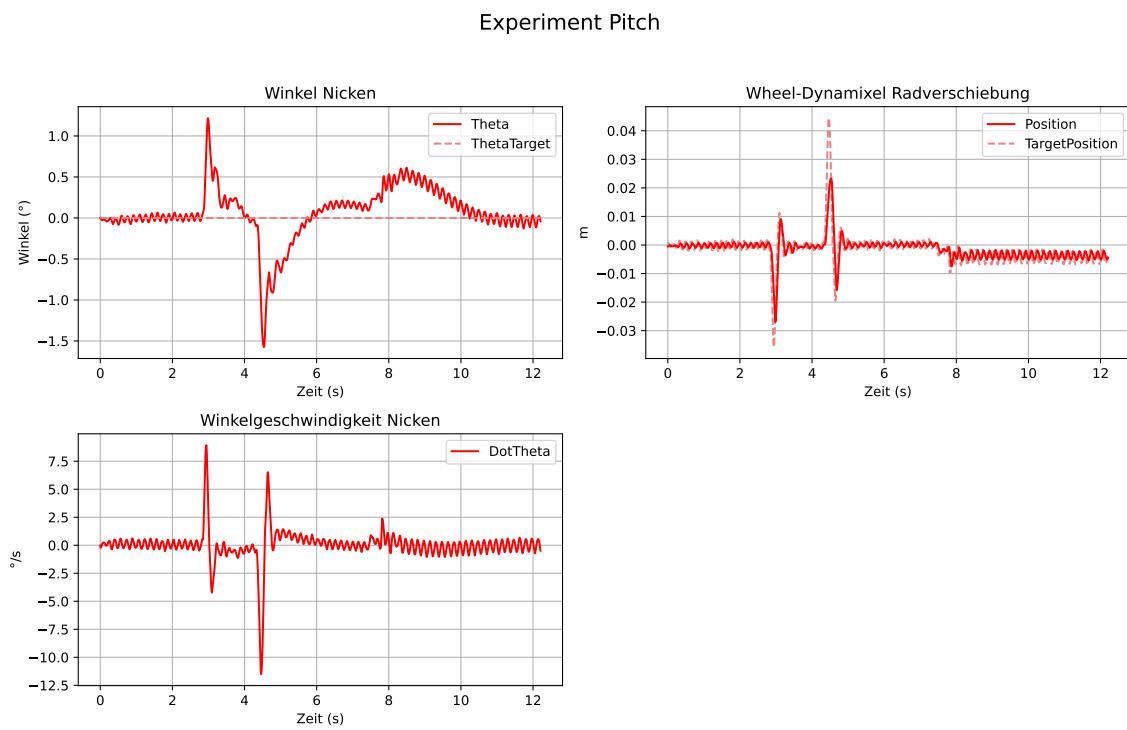


Abbildung 6.3: Pitch Experiment PID-Regler mit $KP = 0,3$; $KD = 0,2$; $KI = 0,3$

Der begrenzende Faktor ist dabei der Aktor, der, wie in Abbildung 6.3 im Plot *Wheel-Dynamixel Radverschiebung* zu sehen, dem Reglerausgang nicht mehr folgen kann.

Schwingungen im stationären Betrieb

Die Experimente zeigen ein zweites Problem: das System oszilliert in der Ruhelage, und diese Schwingung wird stärker, je schneller der Regler ausgelegt wird. Die Schwingung wird eindeutig vom Regler angeregt und verschwindet durch das Hinzufügen einer leichten mechanischen Dämpfung. Das spricht dafür, dass die Kombination aus einem unterdämpften mechanischen System um die Ruhelage sowie den Nichtlinearitäten und Totzeiten des Stellglieds die Ursache ist. Das deckt sich mit dem Verhalten des Servos. Die dynamischen Eigenschaften des Dynamixel-Servomotors sind bei kleinen Sollwertvorgaben deutlich schlechter als bei größeren Sprüngen (siehe Unterabschnitt 4.1.1). Da es sich um ein Einrad handelt, kann keine zusätzliche mechanische Dämpfung in das System eingebaut werden. Auch Anpassungen am internen Regler des Dynamixel-Servomotors bringen keine Verbesserung. Aus Abbildung 6.1 wird ersichtlich, wieso die konstante Schwingung im stationären Betrieb eine kleinere Amplitude hat, wenn der Regler langsamer ist. Das Stellglied folgt den kleinen Sollwertänderungen kaum noch, sodass auch der Effekt auf das Fahrzeug stark reduziert wird. Da die Dämpfung KD maßgeblich für die Schwingungen verantwortlich ist, wird Gain-Scheduling für KD eingeführt. Das Ziel ist es, bei schnellen Bewegungen die Vorteile der hohen Dämpfung auszunutzen, und im stationären Bereich die Oszillationen durch eine sehr geringe Dämpfung zu minimieren. Dabei wird die Dämpfung mit einem von der momentanen Winkelgeschwindigkeit abhängigen Faktor skaliert:

```

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

Liegt die Winkelgeschwindigkeit unter dem Threshold $DScaleThreshold$, wird KD linear verringert, um Unstetigkeiten zu minimieren. Mit den Reglerparametern $KP = 0,3$; $KD = 0,1$; $KI = 0,3$ und dem Gain-Scheduling ergibt sich folgendes Verhalten (siehe Abbildung 6.4):

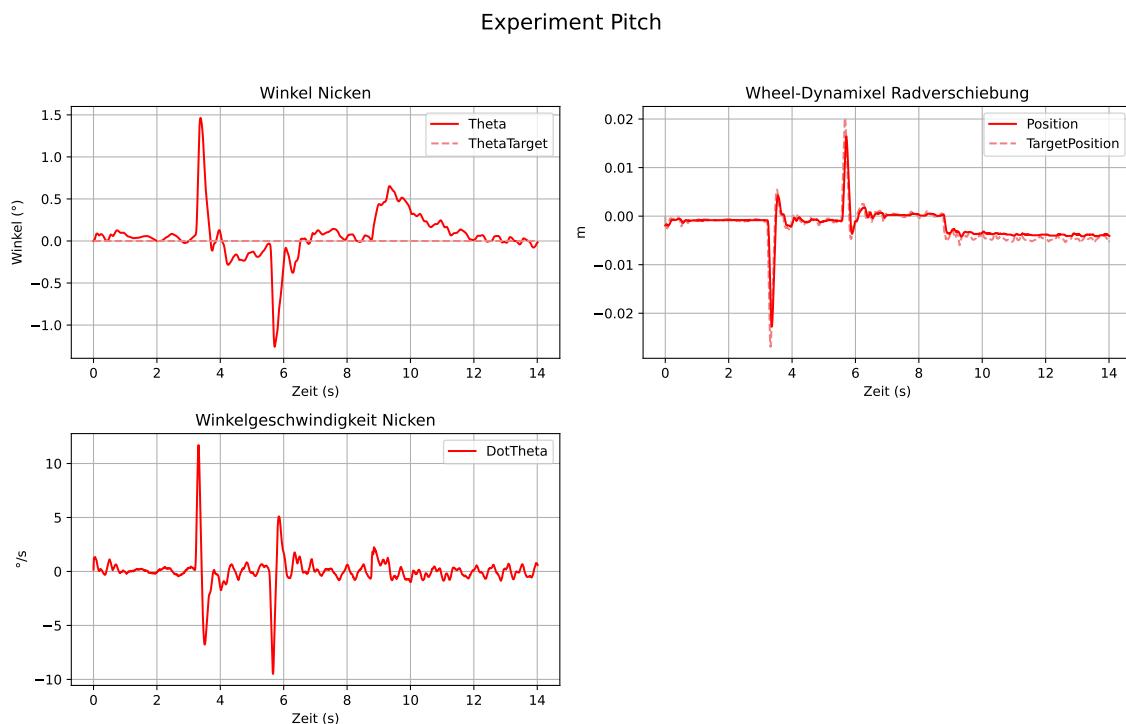


Abbildung 6.4: Pitch Experiment PID-Regler mit $KP = 0,3$; $KD = 0,1$; $KI = 0,3$ mit Gain-Scheduling für KD

Das dynamische Verhalten verschlechtert sich nur minimal, während die Frequenz der Schwingung im stationären Bereich drastisch reduziert wird. Mit dem bloßen Auge ist die Schwingung aufgrund der niedrigen Frequenz und der kleinen Amplitude ($< 0,05^\circ$) kaum mehr erkennbar und befinden sich in einem tolerierbaren Bereich.

Durch eine gezielte und adaptive Parametrierung des PID-Reglers lässt sich das Teilsystem Nicken wirksam stabilisieren. Das Gesamtsystem weist dadurch eine hohe Robustheit auf und kann auch größeren Störungen zuverlässig entgegenwirken. Die erzielte Dynamik ist jedoch nicht optimal, da weiterhin ein Restschwingverhalten auftritt und die Oszillationen im stationären Betrieb zwar verringert, jedoch nicht vollständig eliminiert werden können.

6.2.2 Zustandsregler

Im vorangegangenen Abschnitt wird die Stabilisierung des Nickens durch einen PID-Regler untersucht. Der PID-Regler gewährleistet eine ausreichende Stabilität und kann größere Störungen kompensieren. Dennoch zeigt sich, dass die Systemdynamik nicht optimal beeinflussbar ist. Insbesondere verbleiben Nachschwinger, und stationäre Oszillationen lassen sich nicht vollständig eliminieren. Eine weitergehende Optimierung des PID-Reglers unter Berücksichtigung der Stellglieddynamik (PT1-Verzögerung) ist nicht möglich, da das System dadurch das Hurwitz-Stabilitätskriterium nicht mehr erfüllt. Damit das Stabilitätskriterium eingehalten wird, muss die Zeitkonstante des Stellglieds im Vergleich zur Systemdynamik vernachlässigbar klein sein [33, S. 104].

Vor diesem Hintergrund erfolgt der Übergang zu einer Zustandsregelung. Ein Zustandsregler verspricht in dieser Situation mehrere Vorteile:

- Ein Zustandsregler basiert im Gegensatz zum PID-Regler auf einem mathematischen Modell des Systems. Dadurch kann er die internen Zustandsgrößen gezielt berücksichtigen und aktiv in die Regelung einbeziehen [8, Ch. 8].
- Bessere Berücksichtigung der Stellglieddynamik: Durch die explizite Modellierung des Stellglieds im erweiterten Zustandsraum lassen sich Verzögerungen und Trägheiten direkt kompensieren, ohne dass das System instabil wird.
- Verbesserte Dämpfung und Reduktion von Nachschwingern: Über die gezielte Wahl der Reglerverstärkungen (z.B. mittels LQR oder Pole-Placement) lässt sich die Eigenwertlage systematisch verschieben, wodurch überschüssige Schwingungen reduziert werden [8, Ch. 8].

Zustandsbasierte Regelungsverfahren bieten verschiedene Ansätze zur Stabilisierung und Optimierung dynamischer Systeme, die unterschiedliche Vor- und Nachteile aufweisen.

- Polplatzierung: Bei der Polplatzierung wird die Rückführmatrix K gezielt so gewählt, dass die Pole des geschlossenen Systems gewünschte Eigenschaften vorgeben. Das ist simpel, erlaubt jedoch keine Optimierung und berücksichtigt weder Stellaufwand noch Robustheit [9, Ch. 18].
- Linear Quadratic Regulator (LQR): LQR minimiert eine quadratische Kostenfunktion, die Abweichungen der Zustände von null und den Stellaufwand gewichtet. Dieses Verfahren bietet eine systematische und robuste Optimierung zwischen Regelgüte und Stellaufwand, ist jedoch auf lineare Modelle begrenzt. Der Rechenaufwand zur Laufzeit ist sehr gering und die Stabilität des geschlossenen Systems ist sehr hoch [9, Ch. 22.4].
- H_2/H_∞ : H_2 - und H_∞ -Regler erweitern die Optimierung des LQR-Ansatzes in den Frequenzbereich: H_2 minimiert die mittlere Störenergie und eignet sich besonders für Systeme mit zufälligen Störungen, während H_∞ die maximale Verstärkung über alle Frequenzen begrenzt und damit Robustheit gegenüber Modellunsicherheiten gewährleistet. Beide Methoden sind jedoch mathematisch komplexer [31].
- MPC: MPC sagt das Verhalten des Systems vorher und optimiert die Stellgrößen unter Berücksichtigung von Zustands- und Stellgrößenbeschränkungen zur Laufzeit anhand einer definierten Kostenfunktion. Dadurch lassen sich sehr komplexe Systeme gezielt

und robust regeln, allerdings ist der Implementierungs- und Rechenaufwand sehr hoch [28].

Aufgrund der geringen Rechenzeit zur Laufzeit bei gleichzeitig guter Robustheit des geschlossenen Systems wird das LQR-Verfahren gewählt. Im Folgenden wird die Implementierung der Zustandsregelung für das Nicken-Teilsystem beschrieben, einschließlich der Auswahl der Zustandsgrößen sowie der Vorteile gegenüber der bisherigen PID-Regelung.

Modellierung im Zustandsraum

Für den Entwurf des Zustandsreglers muss das Teilsystem Nicken in die Zustandsraumdarstellung überführt werden. Dabei wird das System nicht mehr mit einer Differenzialgleichung höherer Ordnung beschrieben, sondern als lineares Gleichungssystem erster Ordnung, indem die Dynamik des Systems mit sogenannten Zustandsgrößen x_1, x_2, \dots, x_n beschrieben wird. Das System kann mit den Eingang u durch den Regler beeinflusst werden. Dabei handelt es sich um ein System der Form [9, Ch. 17]:

$$\dot{\vec{x}} = A\vec{x} + Bu \quad (6.7)$$

Bei A handelt es sich um die sog. Systemmatrix und B ist die sog. Steuermatrix. Die Stellgröße wird dann mit dem Vektor aller Zustände und der Reglermatrix K berechnet:

$$u = -K\vec{x} \quad (6.8)$$

Da der LQR-Ansatz nur für lineare Systeme gilt, muss die Differenzialgleichung Gleichung 3.1 linearisiert werden. Die einzige Nichtlinearität ist die Sinusfunktion. Aufgrund der Kleinwinkelnäherung ergibt sich die linearisierte Differenzialgleichung:

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

Aus der linearisierten Differenzialgleichung für die Beschreibung des Nickens folgt folgende Zuordnung der Zustandsgrößen und deren Ableitungen:

$$u = p \quad (6.10)$$

$$x_1 = \vartheta \quad (6.11)$$

$$x_2 = \dot{\vartheta} \quad (6.12)$$

$$\dot{x}_1 = x_2 \quad (6.13)$$

$$\dot{x}_2 = \frac{1}{J_{y,\tau}} \cdot (F_{GP} \cdot u + F_G \cdot h_0 \cdot x_1) \quad (6.14)$$

Die Zustandsgrößen sind der Nickwinkel und dessen Ableitung. Der Eingang u des Systems entspricht der Radverschiebung. Damit ergibt sich folgendes System:

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

Ein Zustandsregler versucht immer, alle Zustandsgrößen auf null zu ziehen. Um auch dauerhafte Störungen und Modellfehler ausgleichen zu können, muss das Zustandsraummodell um den integrierten Fehler erweitert werden.

$$u = p \quad (6.16)$$

$$x_1 = \vartheta \quad (6.17)$$

$$x_2 = \dot{\vartheta} \quad (6.18)$$

$$x_3 = \int -\vartheta \, dt \quad (6.19)$$

$$\dot{x}_1 = x_2 \quad (6.20)$$

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

$$\dot{x}_3 = -\vartheta \quad (6.22)$$

Damit ergibt sich folgendes System:

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

Bisher wird die Dynamik des Stellglieds nicht im Modell berücksichtigt, da der Eingang u direkt die Radverschiebung beschreibt. Um die Verzögerung des Aktors im Modell abzubilden, wird das Zustandsmodell erneut erweitert und die Radverschiebung wird zur Zustandsgröße. Der Eingang u des Systems ist jetzt nicht mehr direkt die Radverschiebung, sondern der Eingang des Aktors p_{cmd} . Der Zusammenhang zwischen dem Eingang des Aktors und der Radverschiebung kann durch ein PT1-Glied beschrieben werden:

$$K \cdot p_{cmd} = T \cdot \dot{p} + p \quad (6.24)$$

Die Ermittlung der Parameter K und T erfolgt in Unterabschnitt 4.1.1. Das erweiterte System lässt sich wie folgt beschreiben:

$$u = p_{cmd} \quad (6.25)$$

$$x_1 = \vartheta \quad (6.26)$$

$$x_2 = \dot{\vartheta} \quad (6.27)$$

$$x_3 = \int -\vartheta \, dt \quad (6.28)$$

$$x_4 = p \quad (6.29)$$

$$\dot{x}_1 = x_2 \quad (6.30)$$

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

$$\dot{x}_3 = -\vartheta \quad (6.32)$$

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

Damit ergibt sich das folgende erweiterte System:

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

Das entspricht der Systemmatrix A , Steuermatrix B und Ausgangsmatrix C :

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

Das vorliegende Zustandsmodell beschreibt die Dynamik des Teilsystems Nicken unter Berücksichtigung des Einflusses des Stellglieds. Um zu bewerten, ob das gekoppelte System geregelt werden kann, wird die Steuerbarkeit untersucht. Die Steuerbarkeit kann mit dem Cayley-Hamilton-Theorem bestimmt werden [9, S. 491]. Dazu wird die Kontrollierbarkeitsmatrix aufgestellt. Entspricht der Rang der Matrix dem Rang der Systemmatrix, ist das System vollständig steuerbar. Die Berechnung erfolgt in dem Python-Skript *pitch_analysis.py* und ist in Listing 5 teilweise zu sehen:

```

1 # --- Steuerbarkeit ---
2 n = A.shape[0]
3 Ctrb = B
4 for i in range(1, n):
5     Ctrb = np.hstack((Ctrb, np.linalg.matrix_power(A, i) @ B))
6
7 rank_ctrb = matrix_rank(Ctrb)
8 print("Rang der Kontrollierbarkeitsmatrix:", rank_ctrb, "von", n)

```

Listing 5: Bestimmung der Steuerbarkeit

Das vorliegende Zustandsmodell ist vollständig steuerbar.

Regelung und Simulation im Zustandsraum

Die Auslegung des Zustandsreglers mittels LQR erfolgt in dem Python-Skript *pitch_LQR.py* mit der Pyhton-Toolbox *control*. Das lineare System muss für den diskreten Reglerentwurf diskretisiert werden. Anschließend können die Zustände und die Stellgröße durch die Matrix Q und den Faktor R gewichtet werden und mit der Funktion *dlqr()* die Reglermatrix K bestimmt werden. Der entsprechende Code-Abschnitt ist in Listing 6 zu sehen:

```

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

```

Listing 6: Diskreter LQR-Entwurf in Python

Die Gewichtung der Zustände und Stellgröße beeinflusst die Reglermatrix K . Um das gewünschte Systemverhalten zu erzeugen, muss die Gewichtung auf das System angepasst

werden. Als Ausgangspunkt kann *Bryson's Rule* verwendet werden. Dabei wird jeder Gewichtungsfaktor auf den maximal zulässigen Wert des zugehörigen Zustands skaliert. Das ist besonders wichtig, wenn die Zustände aufgrund der physikalischen Einheiten unterschiedliche, numerische Größenordnungen haben [13, S. 196]. Als maximale Werte werden die in Tabelle 6.1 angegebenen Größen verwendet:

Tabelle 6.1: Maximal zulässige Werte der Zustände des Teilsystems Nicken

Zustand	Physikalische Größe	Maximaler Wert
x_1	ϑ	0,02 rad
x_2	$\dot{\vartheta}$	0,02 rad s $^{-1}$
x_3	$\int -\vartheta$	0,001 rad s
x_4	p	0,002 m
u	p_{cmd}	0,002 m

Damit ergeben sich folgende Gewichtungen:

$$Q = \begin{bmatrix} \frac{1}{0,02} & 0 & 0 & 0 \\ 0 & \frac{1}{0,02} & 0 & 0 \\ 0 & 0 & \frac{1}{0,001} & 0 \\ 0 & 0 & 0 & \frac{1}{0,002} \end{bmatrix}, \quad R = \frac{1}{0,002} \quad (6.36)$$

Diese Gewichtungen können als Ausgangszustand für weitere Optimierungen verwendet werden. Dazu wird im selben Python-Skript *pitch_LQR.py* das Verhalten des Systems mit Zustandsregler simuliert. Wie beim realen Experiment wird das Aufrichten, eine dynamische Störung und eine dauerhafte Störung getestet. Durch die schrittweise Anpassung von Q und R ergibt sich schließlich für die folgende Gewichtung das gewünschte Systemverhalten mit Reglermatrix K als Kompromiss zwischen aggressivem Störverhalten, sanftem Stellgrößenverlauf und keinen Schwingungen:

$$Q = \begin{bmatrix} \frac{1}{0,02} & 0 & 0 & 0 \\ 0 & \frac{1}{0,02} & 0 & 0 \\ 0 & 0 & \frac{1}{0,001} \cdot 15 & 0 \\ 0 & 0 & 0 & \frac{1}{0,002} \cdot 500 \end{bmatrix}, \quad R = \frac{1}{0,002} \cdot 340, \quad K = \begin{bmatrix} 0,666\,381\,56 \\ 0,129\,172\,11 \\ -0,285\,209\,78 \\ 1,028\,201\,38 \end{bmatrix} \quad (6.37)$$

Das simulierte Systemverhalten ist in Abbildung 6.5 zu sehen:

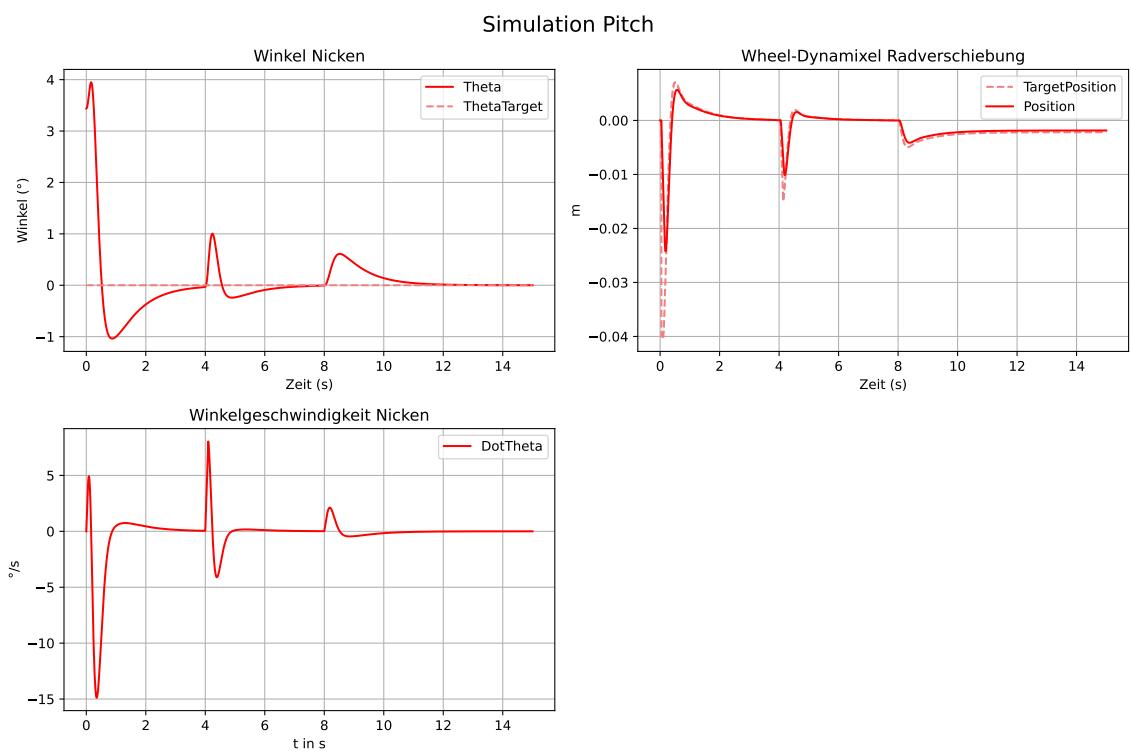


Abbildung 6.5: Nicken Zustandsregler Simulation

Regelung reales System

Der Zustandsregler wird auf dem realen System implementiert und identisch zum PID-Regler getestet. Dabei ergibt sich folgendes Systemverhalten (siehe Abbildung 6.6):

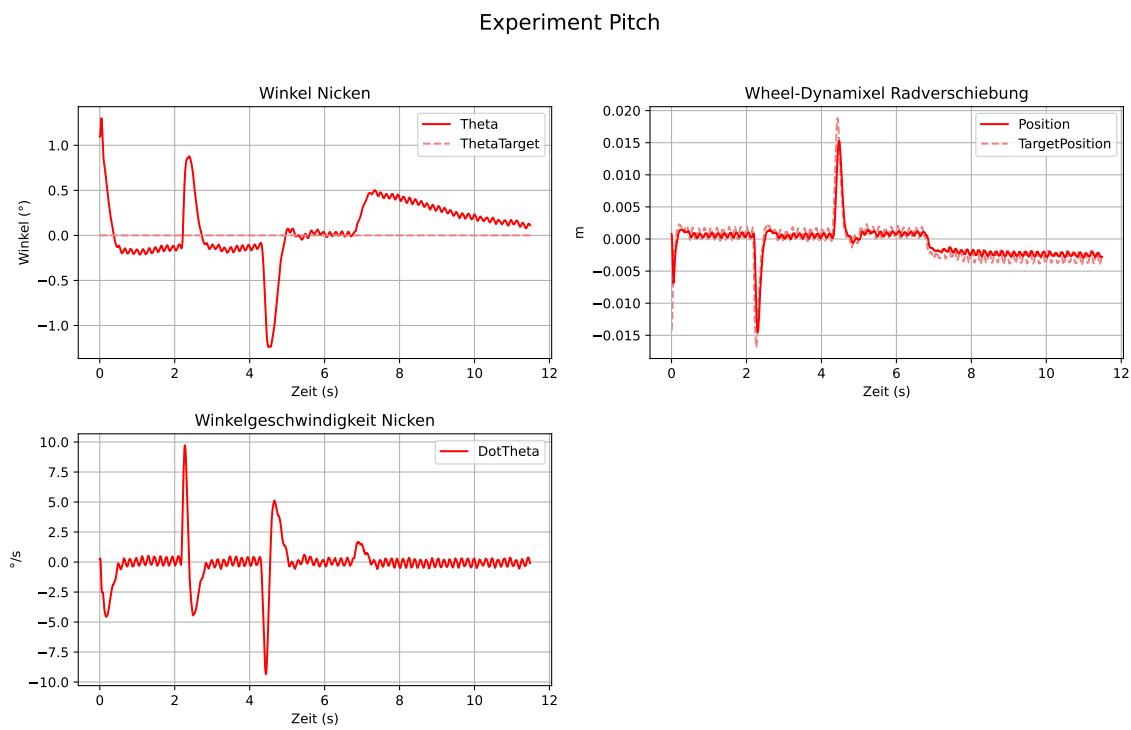


Abbildung 6.6: Nicken Zustandsregler

Das Verhalten des realen Systems ist fast identisch mit dem Verhalten der Simulation. Die Dynamik ist im Vergleich zum PID-Regler deutlich besser und die Schwingungen nach Anregung treten nicht mehr auf. Allerdings oszilliert das System weiterhin in der Ruhelage. Solche Schwingungen können auch auftreten, wenn der Regler zufällig eine Resonanzfrequenz der Mechanik anregt. Um dies zu untersuchen, wird eine Aufnahme gemacht, bei der das Fahrzeug nicht gestört wird, sodass nur die Schwingung in der Ruhelage aufgezeichnet wird. Anschließend wird der Frequenzbereich des Reglerausgangs untersucht (siehe Abbildung 6.7):

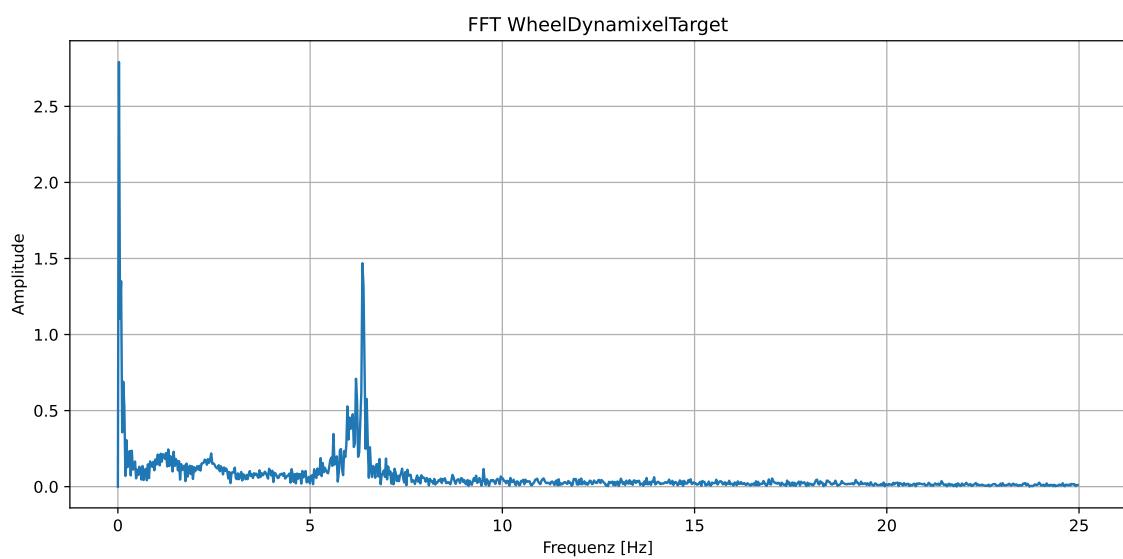


Abbildung 6.7: FFT Reglerausgang

Wie zu erwarten treten in dem Signal keine Frequenzen über 8 Hz auf, da die Mechanik des Fahrzeugs wie ein Tiefpassfilter wirkt. Allerdings ist ein eindeutiger Ausschlag bei 6,3 Hz zu beobachten. Das entspricht der Frequenz der Oszillationen in der Ruhelage. Um zu testen, ob es sich um eine Resonanzfrequenz der Mechanik handelt, wird ein Notchfilter mit einer Notch-Frequenz von 6,3 Hz auf den Reglerausgang angewendet und derselbe Test erneut durchgeführt (siehe Abbildung 6.8):

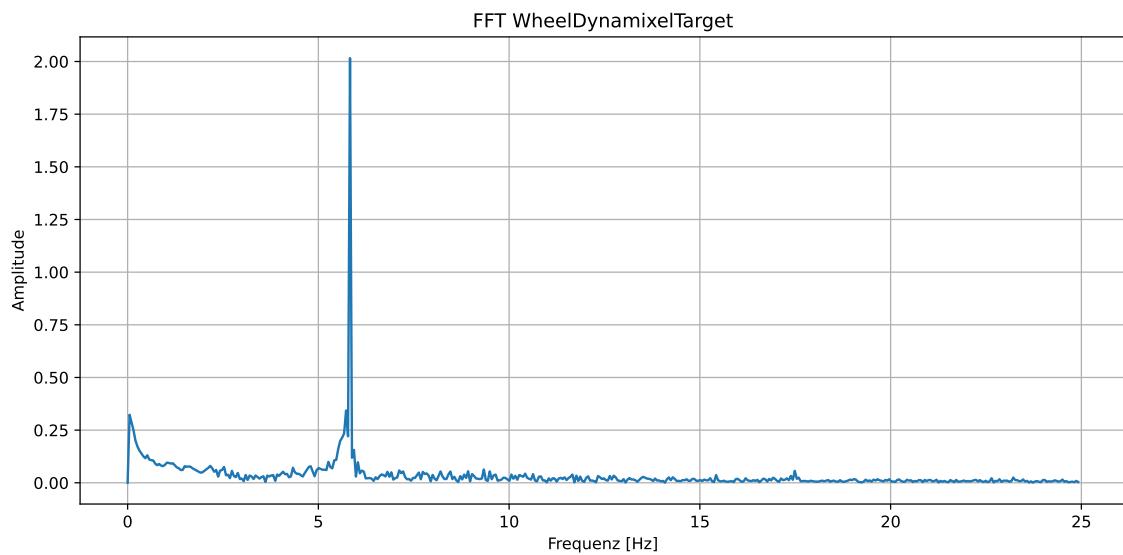


Abbildung 6.8: FFT Reglerausgang mit Notchfilter

Die Schwingung hat sich auf eine andere Frequenz verlagert, sodass der Ausschlag nun bei 5,8 Hz ist. Das zeigt, dass es sich nicht um eine Resonanzfrequenz der Mechanik handelt.

Um auszuschließen, dass die Schwingung durch die Dynamik des geschlossenen Regelkreises erzeugt werden, werden dessen Pole untersucht. Die Berechnung erfolgt auch in dem Python-Skript *pitch_LQR.py*. Für die Pole des diskreten Systems ergibt sich:

$$p_1 = 0,880\,342\,93, \quad p_2 = 0,977\,292\,11, \quad p_3 = 0,985\,676\,23, \quad p_4 = 0,994\,653\,85 \quad (6.38)$$

Alle Pole sind rein reell und liegen innerhalb des Einheitskreises, daher ist das System nicht schwingungsfähig und stabil. Das lässt darauf schließen, dass die Schwingungen aufgrund von nicht modellierten Nichtlinearitäten entstehen. Der Aktor ist zwar als PT1-Glied modelliert, hat eigentlich aber auch eine Totzeit von ca. 5 ms. Hinzu kommt Getriebespiel, die Haftreibung des Rads und die schlechte Dynamik des Dynamixel-Servos bei kleinen Sollwertvorgaben (siehe Unterabschnitt 4.1.1). Auch der Fakt, dass die Schwingung durch leichtes Einbringen von mechanischer Dämpfung ins System (Auflegen eines Fingers) unterdrückt wird, bestätigt diese Interpretation. Solche selbsterhaltenden Schwingungen werden in der Regelungstechnik auch Limit Cycles genannt [25, Ch. 4.2].

Die Experimente mit dem PID-Regler haben gezeigt, dass eine effektive Gegenmaßnahme darin besteht, die Amplitude der Schwingung so zu senken, dass der Aktor der Vorgabe nicht mehr folgt und somit die Schwingung dämpft. Daher wird erneut Gain-Scheduling eingesetzt. Wird der Einfluss des Winkels ϑ und der Winkelgeschwindigkeit $\dot{\vartheta}$ auf ca. die Hälfte der ursprünglichen Reglermatrix K reduziert, wird die Amplitude und Frequenz der Schwingung in der Ruhelage so gering, dass die Bewegungen kaum erkennbar sind. Um die guten Dynamikeigenschaften des ursprünglichen Reglers beizubehalten und gleichzeitig die Vorteile des langsameren Parametersets in der Ruhelage nutzen zu können, müssen geeignete Bedingungen für die Wahl des aktiven Parametersets gefunden werden.

Zuerst wird der Übergang vom schnellen in das langsame Parameterset betrachtet. Damit das Fahrzeug nach dem Ausgleich einer Störung zuverlässig erkennt, wann es sich in der Ruhelage befindet, muss ein Bereich definiert werden, der als Ruhelage gilt. Dieser Bereich soll so klein wie möglich sein, um die guten Dynamikeigenschaften des schnellen Parametersets voll zu nutzen, aber so groß wie nötig, um trotz der Limit Cycles die Ruhelage zuverlässig zu erkennen.

Anhand der Amplitude und Frequenz der Schwingung in der Ruhelage können Grenzen für den Nickwinkel und die Winkelgeschwindigkeit festgelegt werden, die den Eintritt in die Ruhelage markieren. Zusätzlich wird eine Mindestdauer definiert, für die sich das Fahrzeug innerhalb dieses Bereichs befinden muss, um sicherzustellen, dass es sich tatsächlich in der Ruhelage befindet und sich nicht in einem Überschwinger befindet.

Schnelles Hin- und Herwechseln zwischen den Parametersets kann Schwingungen anregen und das dynamische Verhalten des Fahrzeugs negativ beeinflussen. Um zu verhindern, dass kleinste Störungen unnötige Wechsel provozieren, wird eine Hysterese eingeführt. Der Bereich, ab dem das Fahrzeug erkennt, dass es die Ruhelage verlässt, wird größer gewählt, als der Bereich zum Erkennen des Eintritts. Allerdings reicht ein einmaliges Verlassen des Bereichs aus, um den Wechsel zum schnellen Parameterset zu veranlassen.

Ein hartes Umschalten zwischen den zwei Parametersets kann zu Unstetigkeiten in der Stellgröße führen und unerwünschte Schwingungen und Beschleunigungen hervorrufen. Daher wird eine Umschaltzeit eingeführt, in der die Stellgröße beider Parametersets linear interpoliert wird, bis die Umstellung auf das neue Parameterset vollständig erfolgt ist.

Es muss beachtet werden, die Gewichtung des Fehlerintegrals beim Umschalten zwischen den Parametern nicht zu verändern, da sonst das in Abbildung 6.9 gezeigte Phänomen bei dauerhaften Störungen auftritt:

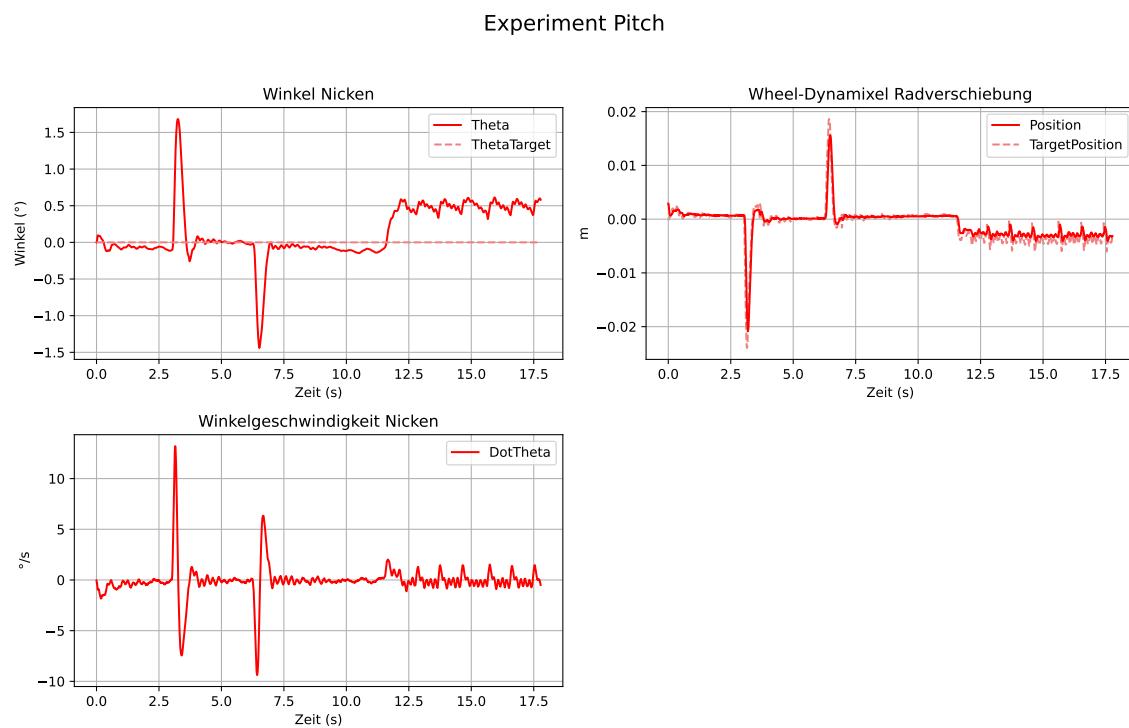


Abbildung 6.9: Gain-Scheduling mit unterschiedlicher Gewichtung des Fehlerintegrals

Um die dauerhafte Störung auszugleichen wechselt der Regler aufgrund des großen Fehlers zum schnellen Parameterset. Sobald der Fehler abgebaut wird, schaltet der Regler auf das langsamere Parameterset um. Die reduzierte Gewichtung des Fehlerintegrals führt nun dazu, dass der Fehler wieder größer wird, weil die Stellgröße verringert wird. Dadurch schaltet der Regler wieder zum schnelleren Parameterset und der Vorgang wiederholt sich, sodass das Fahrzeug dauerhafte Störungen nicht effektiv ausgleichen kann.

Das Verhalten des Systems mit dem Zustandsregler mit Gain-Scheduling ist in Abbildung 6.10 zu sehen:

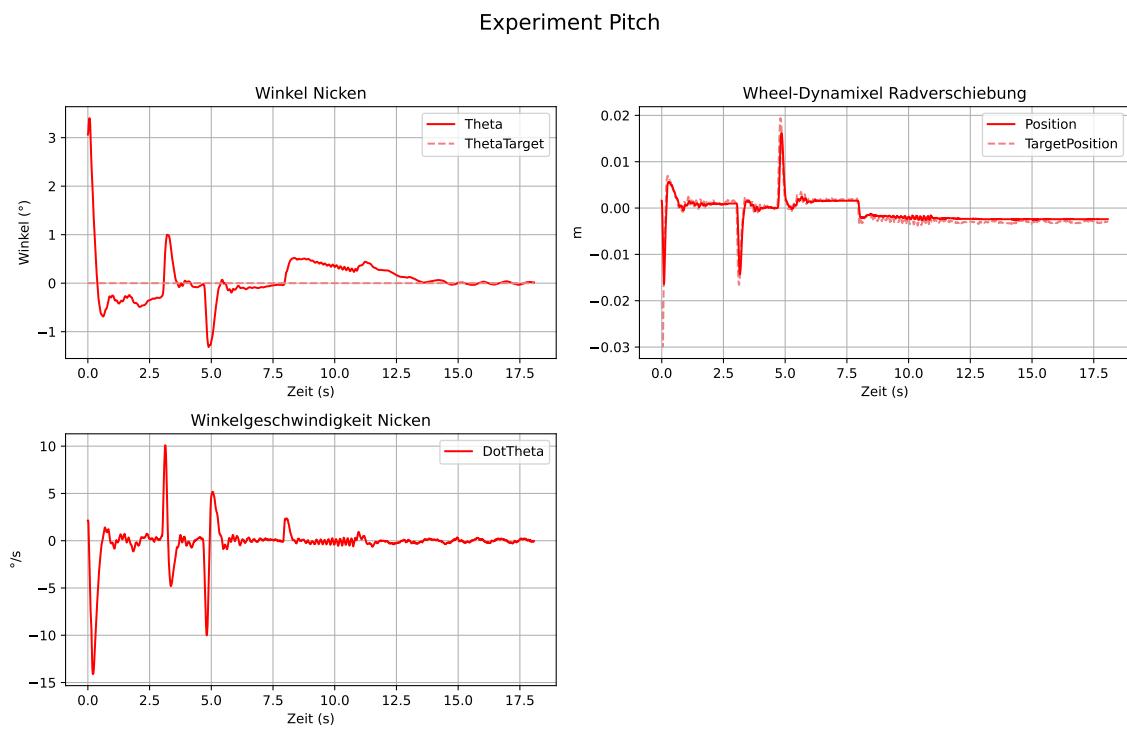


Abbildung 6.10: Nicken Zustandsregler mit Gain-Scheduling

Es ist zu erkennen, dass das gute dynamische Verhalten des schnellen Parametersets weitestgehend erhalten bleibt, während die Oszillationen in der Ruhelage drastisch reduziert werden. Die Anforderungen an den Reglerentwurf für das Teilsystem Nicken sind damit erfüllt. Des Weiteren konnte gezeigt werden, dass das vereinfachte Modell der Nickbewegung die Dynamik des Systems ausreichend genau abbildet und dass sich das Verhalten des realen Fahrzeugs mit dem Verhalten des Modells in der Simulation deckt.

6.3 Teilsystem Rollen und Gieren

Bei dem Teilsystem Rollen und Gieren handelt es sich wie in Kapitel 3 beschrieben um ein gekoppeltes Differenzialgleichungssystem. Zum Balancieren ist aber nur die Stabilisierung der Rollbewegung nötig, da die Kopplung zwischen Rollen und Gieren im Arbeitspunkt beim Balancieren durch die Haftreibung des Rads aufgehoben wird (siehe Kapitel 3).

Das System hat zwar nur einen Input, die Rotation des Kreisels um die Hochachse, aber mehrere Outputs. Da die Stellgröße die Winkelgeschwindigkeit des Kreisels um seine Hochachse ist, wirkt diese nur so lange in eine Richtung, bis der mechanisch zulässige Drehwinkel erreicht ist. Damit steht nur eine begrenzte Reserve an Stellgröße zur Verfügung: Nach einer Auslenkung von etwa $\pm 15^\circ$ kann keine weitere Stellwirkung in derselben Richtung mehr erzeugt werden. Dauerhafte Störungen können damit ohne weitere Maßnahmen nicht ausgeglichen werden. In der Realität ist das Fahrzeug nicht ideal ausbalanciert, sodass das Ausgleichen dieser Art der dauerhaften Störung essenziell für das Balancieren ist. Der

Winkel des Kreisels muss daher, zusätzlich zum Rollwinkel, ebenfalls als Ausgang des Systems betrachtet und geregelt werden. Damit handelt es sich um ein Single Input Multiple Output (SIMO)-System. Der Regler muss zusätzlich zur Stabilisierung des Rollwinkels den Rollwinkel anhand des Winkels des Kreisels so anpassen, dass der Kreisel in die Mitte gedreht werden kann. Durch die aktive Schrägstellung des Fahrzeugs kompensiert die Schwerkraft die dauerhafte Störung.

Es gibt zwei Aufgaben, die der Regler erfüllen muss. Die Hauptaufgabe ist das Balancieren des Fahrzeugs. Dazu muss sich das Fahrzeug aufrichten, kurze Störungen unterdrücken und dauerhafte Störungen ausgleichen. Dabei darf keine unerwünschte Gierbewegung auftreten, da das Fahrzeug still stehen soll bzw. in einer Linie fahren soll. Dafür darf die Winkelgeschwindigkeit des Rollens eine gewisse Schwelle nicht überschreiten (siehe Kapitel 3). Der Reglerentwurf darf daher nicht zu aggressiv erfolgen und Robustheit steht im Vordergrund. Zusätzlich soll der Regler für Experimente zum Kurvenfahren genutzt werden können. Dafür muss der Regler in der Lage sein, einer Trajektorie für den Rollwinkel zu folgen, um damit eine Kurve zu realisieren (siehe Kapitel 7).

Klassische PID-Regler sind Single Input Single Output (SISO)-Systeme. In SIMO-Systemen führt die Rückkopplung der verschiedenen Ausgänge bei der Verwendung eines PID-Reglers oft zu Instabilitäten. Daher bietet sich die Verwendung eines LQR-Reglers an, da diese allgemein für Multiple Input Multiple Output (MIMO)-Systeme ausgelegt sind. Während LQR-Regler in erster Linie zur Stabilisierung eines Systems um einen Arbeitspunkt ausgelegt sind, ermöglichen PID-Regler aber auch ohne Erweiterungen der Reglerstruktur das Nachfahren von Trajektorien. Daher sollen die folgenden Abschnitte die Stabilisierung des Systems sowohl mit einem PID-Regler als auch mit einem LQR-Regler untersuchen.

6.3.1 PID-Regler

Da PID-Regler nur für SISO-Systeme ausgelegt sind, wird zunächst ein PID-Regler zur Kontrolle des Rollwinkels ausgelegt. Der Winkel des Kreisels wird zunächst ignoriert. Durch diese Vereinfachung handelt es sich um ein SISO-System. Eine erste, theoretische Auslegung eines PID-Reglers für das reduzierte Teilsystem Rollen erfolgt in [33, S. 109] auf Grundlage theoretischer Modellannahmen. Da sich die Parameter des fertigen Fahrzeugs von den ursprünglich angenommenen Werten unterscheiden, und das Modell erweitert wurde (Dynamik des Stellglieds, bessere Abschätzung von Reibungsparametern etc.), können die dort bestimmten Regelparameter $KP = 12$; $KD = 1,44$; $KI = 10,62$ nur als Ausgangspunkt für das manuelle Tuning in der Simulation genutzt werden.

Reglerentwurf Rollwinkel

Die Simulation wird im Python-Skript *roll_PID.py* berechnet. Dabei wird zunächst das Aufrichten und eine dauerhafte Störung simuliert. Die dauerhafte Störung ist essenziell, da das Fahrzeug nicht perfekt ausbalanciert ist, und das Ausgleichen einer dauerhaften Störung die Voraussetzung zum Balancieren ist. Das Systemverhalten für die Regelparameter $KP = 12$; $KD = 1,44$; $KI = 10,62$ ist in Abbildung 6.11 zu sehen:

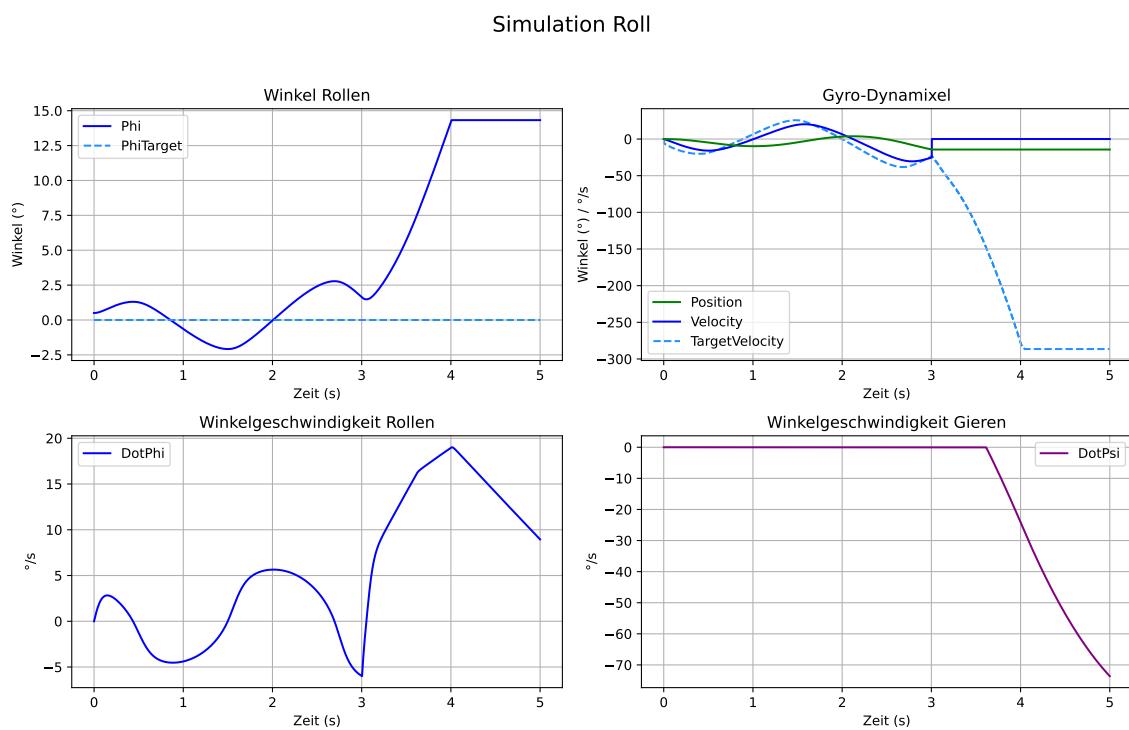


Abbildung 6.11: Rollen Simulation mit $KP = 12$; $KD = 1,44$; $KI = 10,62$

Das System schwingt sich auf und der Regler ist nicht in der Lage, das Fahrzeug effektiv zu stabilisieren. Daher wird die Dämpfung und der Proportionalanteil erhöht und der Integralanteil drastisch reduziert, um die Dynamik zu verbessern. Für die Regelparameter $KP = 14$; $KD = 2,8$; $KI = 1,4$ ergibt sich folgendes Systemverhalten (siehe Abbildung 6.12):

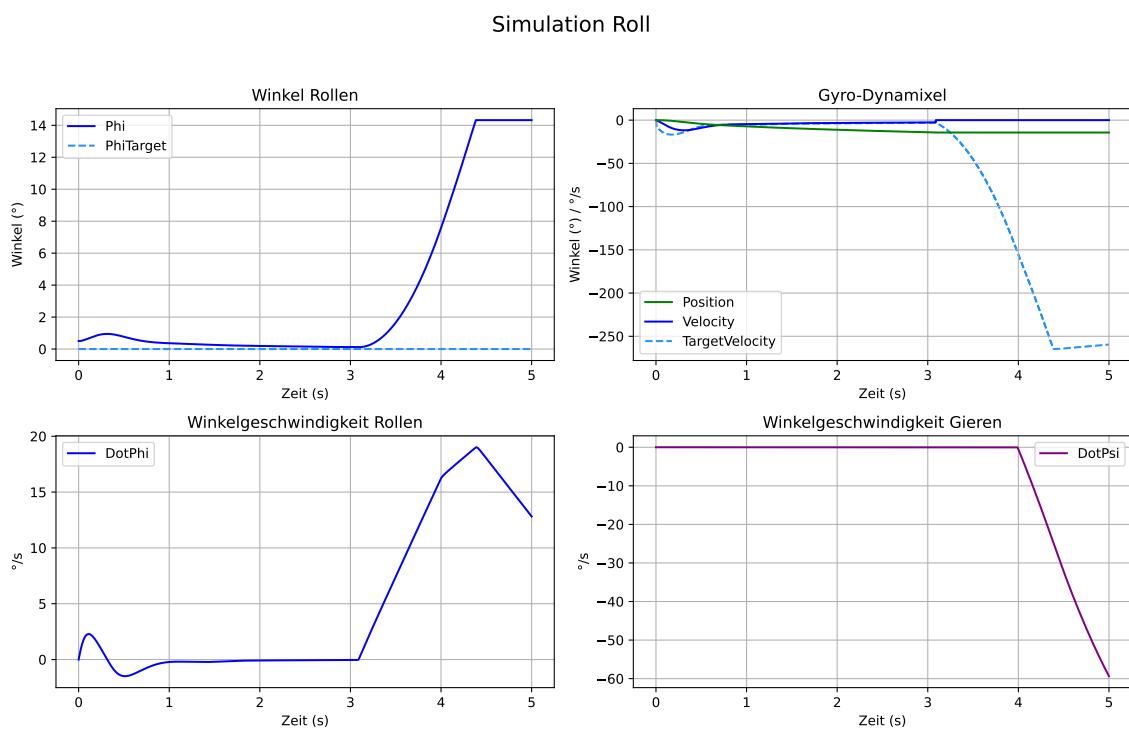


Abbildung 6.12: Rollen Simulation mit $KP = 14$; $KD = 2,8$; $KI = 1,4$

Das System schwingt nicht mehr und der Regler kann das System für eine gewisse Zeit effektiv stabilisieren. Aufgrund der dauerhaften Störung ist aber eine konstante Stellgröße erforderlich. Dadurch erreicht der Kreisel seine maximale Auslenkung und das Fahrzeug fällt um. Das Systemverhalten kann durch eine leichte Modifikation der Parameter zu $KP = 15$; $KD = 3$; $KI = 3$ weiter verbessert werden:

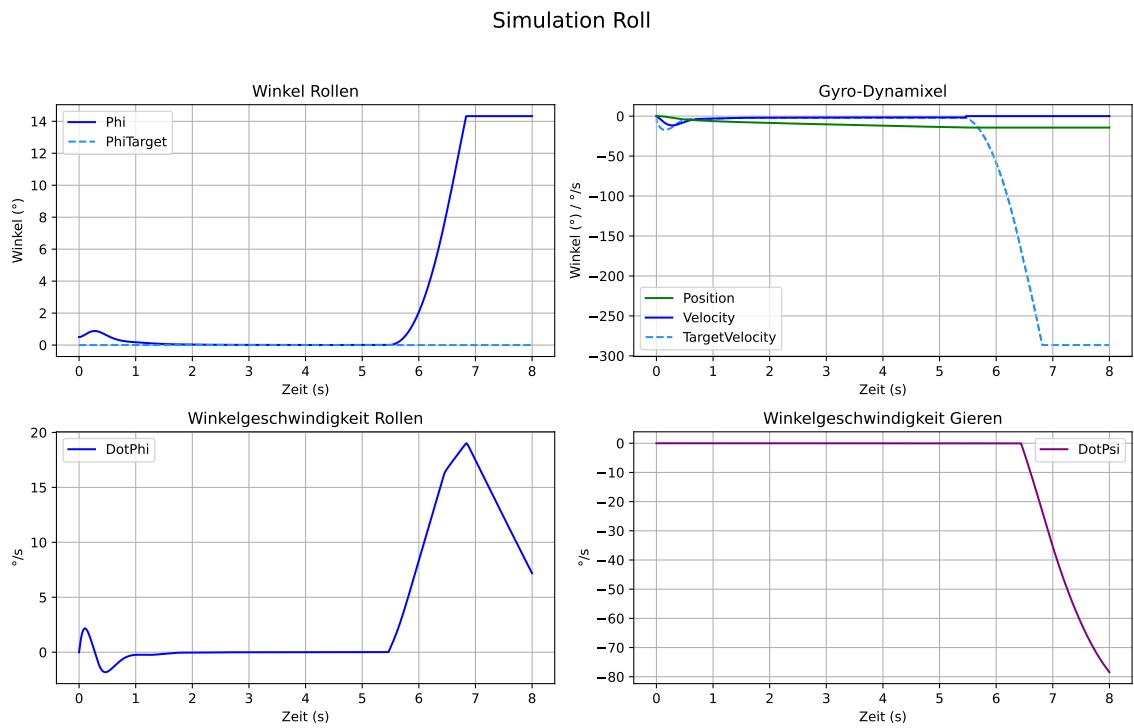


Abbildung 6.13: Rollen Simulation mit $KP = 15$; $KD = 3$; $KI = 3$

Der Regler kann aufgrund der effektiveren Nutzung der Stellgröße das System schneller zum Sollwert führen und für über 5 s stabilisieren, während das vorherige Parameterset bereits nach 3 s das Maximum des Kreisels ausschöpft. Außerdem ist die Entkopplung der Roll- und Gierbewegung zu sehen. Solange sich das Fahrzeug nahe der Ruhelage befindet, führen Rollbewegungen nicht zu Gierbewegungen. Erst wenn das Fahrzeug umfällt und die Haftreibung des Rads überwindet, ist das System wieder gekoppelt.

Um dauerhafte Störungen ausgleichen zu können, muss das Fahrzeug seinen Rollwinkel so anpassen, dass die Stellgröße dafür sorgt, dass sich der Kreisel zur Mitte hin bewegt. Eine simple Version dieser Idee wird in [2] umgesetzt. Abhängig vom Winkel des Kreisels wird ein fester Sollwert vorgegeben, der das Fahrzeug dazu bringt, in die andere Richtung umzufallen, sodass der Kreisel in die entgegengesetzte Richtung, zurück zur Mitte, gedreht werden muss. Die Vorgabe des Sollwerts kann anhand des aktuellen Winkels des Kreisels getätigt werden. Die festen Sollwerte, zwischen welchen gewechselt wird, müssen an die Störung angepasst werden. Sind die Sollwerte zu groß, kann sich das Fahrzeug beim Wechsel nicht mehr fangen und der Kreisel hat nicht genug Reserven, um das Fahrzeug wieder aufzurichten. Sind die Sollwerte zu klein, reicht das Schrägstellen nicht aus, um das Fahrzeug zum Umfallen auf die andere Seite zu bringen, und die Drehrichtung des Kreisels wird nicht umgedreht. Auch in diesem Fall fällt das Fahrzeug um. Abbildung 6.14 zeigt, wie das Umschalten zwischen festen Sollwerten das Fahrzeug dauerhaft stabilisieren kann:

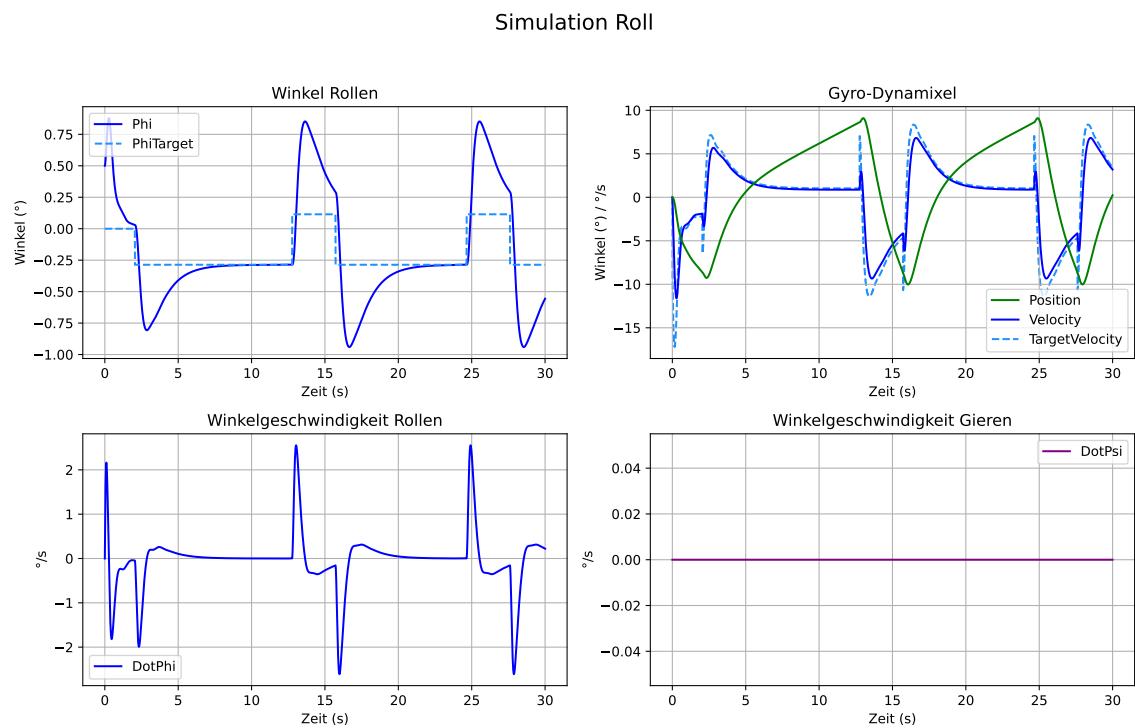


Abbildung 6.14: Rollen Simulation mit $KP = 15$; $KD = 3$; $KI = 3$ mit fester Sollwertvorgabe

Dabei ist zu erkennen, dass das Fahrzeug durch die Störung eigentlich in positive Rollrichtung fällt. Wird das Fahrzeug in negative Rollrichtung ausgelenkt, kompensiert die Schräglage einen Teil der Störung, und es ist nur eine kleine Stellgröße erforderlich. Ist das Fahrzeug jedoch in positive Richtung ausgelenkt, verstärkt die Schräglage die Störung, und es ist eine große Stellgröße erforderlich. Daher ist das Fahrzeug viel länger in der negativen Schräglage. Diese Methode funktioniert nur für einen sehr begrenzten Bereich von Störungen. Außerdem reicht eine zusätzliche, kurzzeitige Störung im falschen Moment, und das Fahrzeug fällt um. Daher soll das System wieder auf ein SIMO-System erweitert werden und der Winkel des Kreisels in die Regelung einbezogen werden. Dies soll mit einer dynamischen Sollwertvorgabe des Rollwinkels umgesetzt werden. Um den zweiten Output, den Winkel des Kreisels, des SIMO-Systems zu regeln, wird ein zweiter PID-Regler verwendet. Der zweite PID-Regler hat den Sollwert für den Rollwinkel als Stellgröße und ist damit mit dem PID-Regler zur Steuerung des Rollwinkels kaskadiert. Die Reglerstruktur ist in Abbildung 6.15 zu sehen.

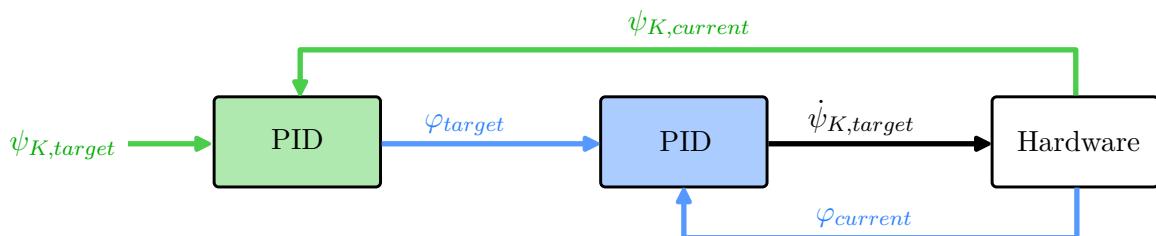


Abbildung 6.15: Struktur kaskadierter PID-Regler zum Balancieren

Die Auslegung erfolgt experimentell anhand der Simulation. Dabei bieten die Parameter $KP = 0,02$; $KD = 0,003$; $KI = 0,01$ einen Kompromiss aus Robustheit und Dynamik. Um die Phasenverschiebung zu reduzieren, wird der Eingang des PID-Reglers, der den Rollwinkel regelt, modifiziert. Normalerweise wird die Regelabweichung und deren Ableitung folgendermaßen berechnet:

$$e = \varphi_{ref} - \varphi \quad (6.39)$$

$$\dot{e} = \frac{e - e_{old}}{Ta} \quad (6.40)$$

Stattdessen wird die Regelabweichungen und Ableitung nun wie folgt berechnet:

$$e = \varphi_{ref} - \varphi \quad (6.41)$$

$$\dot{e} = \dot{\varphi}_{ref} - \dot{\varphi} \quad (6.42)$$

Dadurch wird das Führungsverhalten des PID-Reglers verbessert. Um den Sollwert der Rollgeschwindigkeit zu bestimmen, wird der Reglerausgang des zweiten PID-Reglers abgeleitet. Zusätzlich zu der dauerhaften Störung wird eine kurzzeitige Störung in der Simulation getestet. Das Ergebnis ist in Abbildung 6.16 zu sehen:

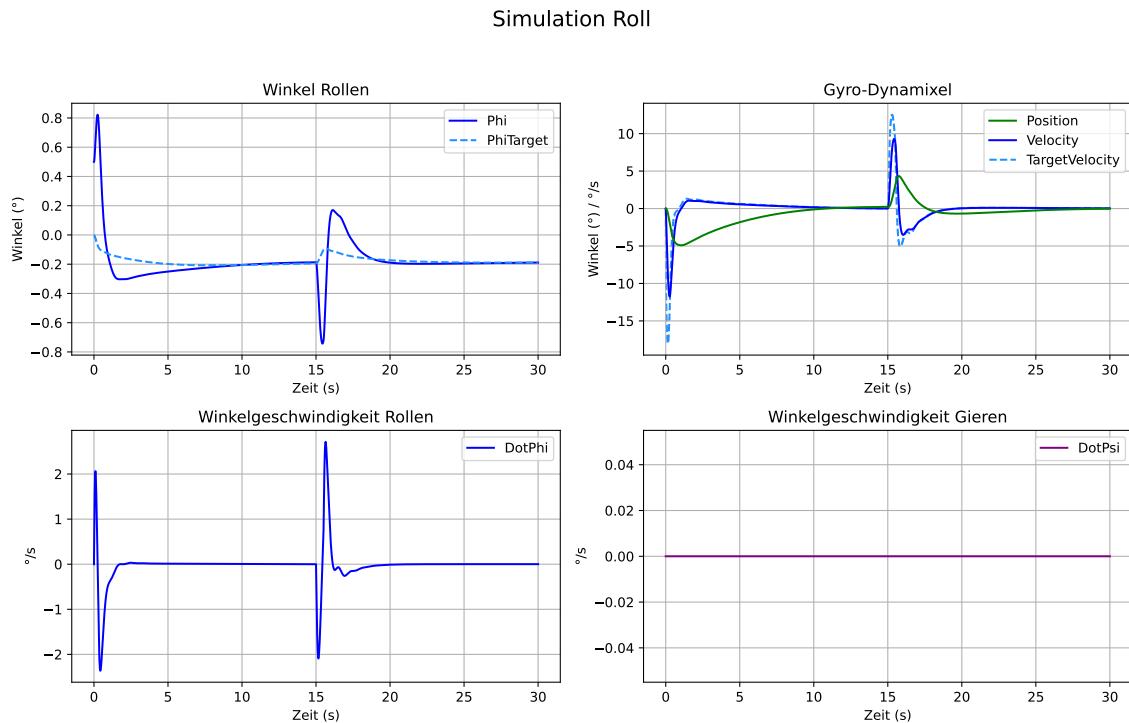


Abbildung 6.16: Rollen Simulation mit $KP = 15$; $KD = 3$; $KI = 3$ mit dynamischer Sollwertvorgabe

Damit sind alle Anforderungen erfüllt. Das SIMO-System mit zwei Outputs kann durch den Einsatz von zwei kaskadierten PID-Reglern stabilisiert werden und kann sich bei realistischem Stellgrößenaufwand aufrichten, kurzzeitige Störungen unterdrücken und dauerhafte Störungen ausgleichen. Zusätzlich folgt das System einer vorgegebenen Trajektorie mit ausreichender Genauigkeit. Der Regler kann am realen System getestet werden.

Regelung reales System

Der in der Simulation ermittelte PID-Regler für die Rollbewegung wird auf dem Fahrzeug implementiert. Um dauerhafte Störungen ausgleichen zu können, werden zunächst zwei feste Sollwerte vorgeben, zwischen denen abhängig vom Winkel des Kreisels umgeschaltet wird (wie in der Simulation in Abbildung 6.14). Das Fahrzeug richtet sich zu Beginn des Experiments auf und gleicht dann die dauerhafte Störung aus, die durch den Offset des Schwerpunkts des Fahrzeugs entsteht. Die festen Sollwerte werden experimentell ermittelt. Damit ergibt sich folgendes Systemverhalten (siehe Abbildung 6.17):

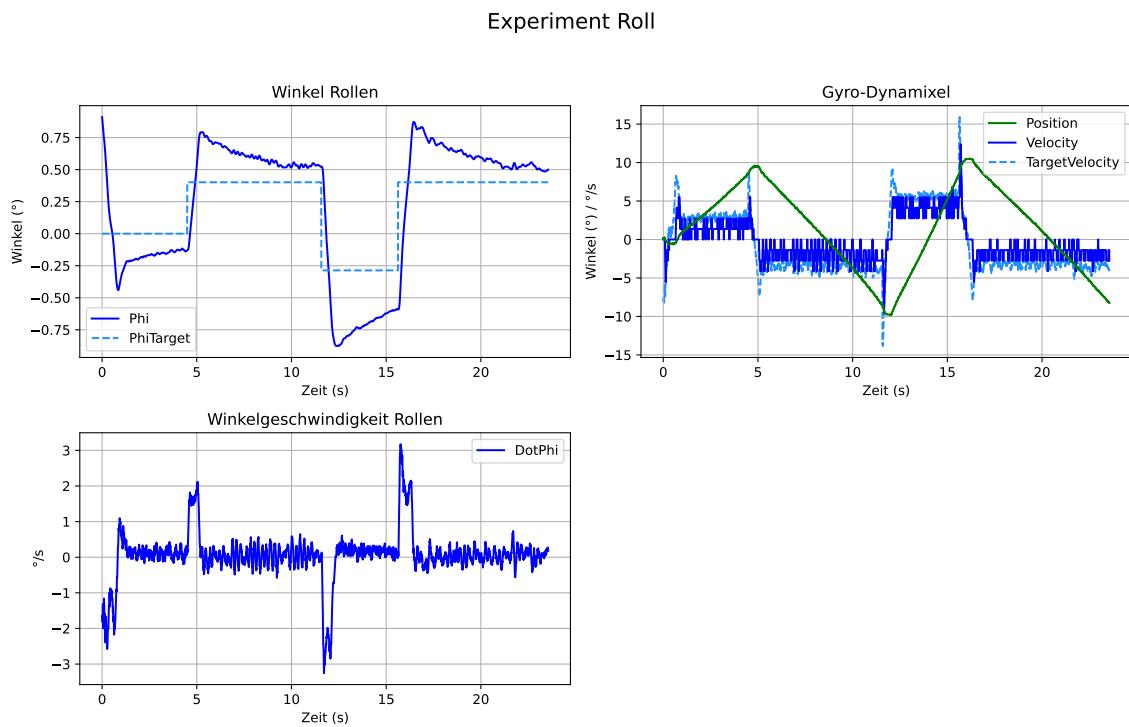


Abbildung 6.17: Rollen Experiment mit $KP = 15$; $KD = 3$; $KI = 3$ mit fester Sollwertvorgabe

Das Verhalten des realen Systems deckt sich mit den Ergebnissen der Simulation. Das Fahrzeug kann auf unbegrenzte Zeit stabilisiert werden und die Dynamik ist der Simulation sehr ähnlich. Um flexibel auf jede Art der Störung reagieren zu können, wird der zweite PID-Regler zur Generierung des Sollwerts genutzt (wie in Abbildung 6.16). Das Ergebnis des Experiments ist in Abbildung 6.18 zu sehen.

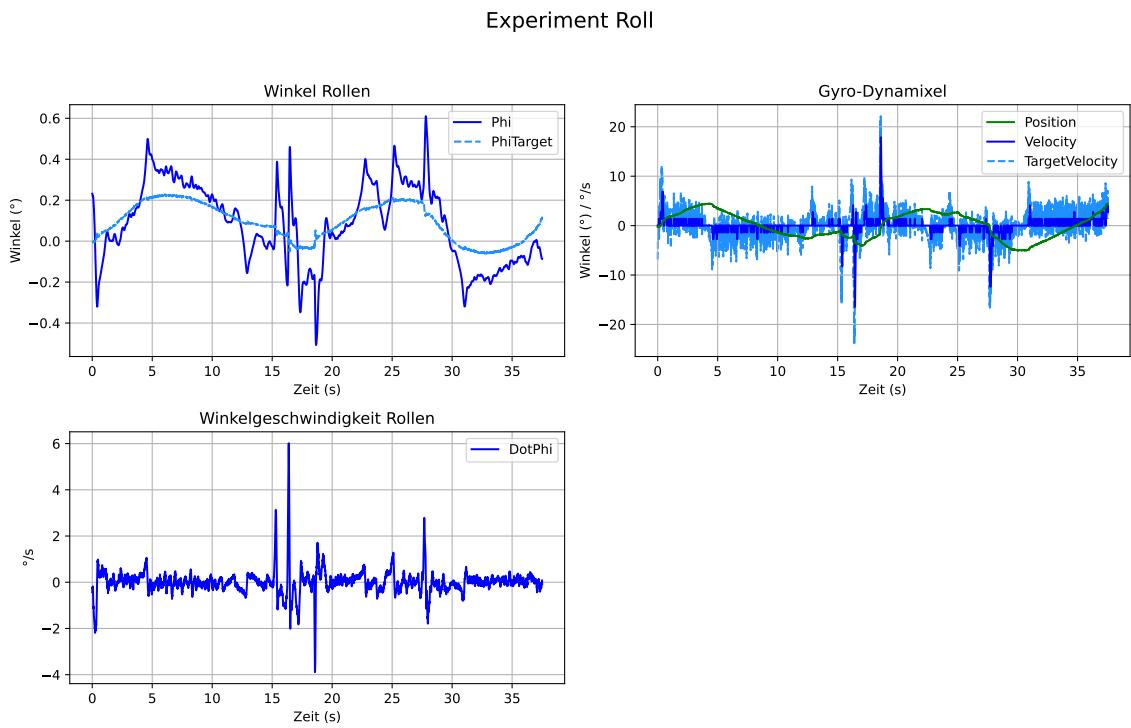


Abbildung 6.18: Rollen Experiment mit $KP = 15$; $KD = 3$; $KI = 3$ mit dynamischer Sollwertvorgabe

Zusätzlich zu der dauerhaften Störung wirken einige kurzzeitige Störungen auf das System, die ohne Probleme ausgeglichen werden. Auch das Aufrichten und der Ausgleich der dauerhaften Störung erfolgt gemäß den Anforderungen. Auffällig ist aber, dass das System mit einer sehr kleinen Frequenz schwingt. Dieses Schwingen lässt sich ähnlich wie das Schwingen beim Regeln der Nickbewegung auf Nichtlinearitäten und Ungenauigkeiten durch Reibung, Spiel und Totzeiten zurückführen. Aufgrund der kleinen Amplitude $A \approx 0,1^\circ$ und Frequenz $f \approx 0,05 \text{ Hz}$ sind diese Schwingungen mit dem bloßen Auge kaum zu erkennen und können im Kontext des Gesamtsystems ignoriert werden. Auch für die Stabilisierung der Rollbewegungen konnte gezeigt werden, dass das vereinfachte Modell des Fahrzeugs die Dynamik des realen Systems ausreichend genau abbildet, und dass sich das Verhalten des realen Fahrzeugs mit dem Verhalten des Modells in der Simulation deckt.

6.3.2 Zustandsregler

Aufgrund der in Abschnitt 6.3 genannten Vorteile eines Zustandsreglers beim Regeln von SIMO-Systemen soll untersucht werden, ob ein Zustandsregler im Vergleich zu einem PID-Regler ein besseres Systemverhalten erzielen kann. Der grundlegende Aufbau und die verschiedenen Arten von Zustandsreglern sind bereits im Unterabschnitt 6.2.2 beschrieben. Auch für das Teilsystem Rollen und Gieren soll ein LQR-Regler zum Einsatz kommen.

Modellierung im Zustandsraum

Das Differenzialgleichungssystem Gleichung 3.8 muss linearisiert werden und vereinfacht sich durch die Entkopplung zwischen Rollen und Gieren im Arbeitspunkt zum Balancieren. Mit der Kleinwinkelnäherung ergibt sich das folgende lineare System:

$$J_{x,\tau} \cdot \ddot{\varphi} = J_K \cdot w_K \cdot \dot{\psi}_K + F_G \cdot h_0 \cdot \varphi \quad (6.43)$$

$$J_z \cdot \ddot{\psi} = 0 \quad (6.44)$$

Der Eingang des Systems u ist die Winkelgeschwindigkeit des Kreisels. Wie bereits beschrieben, soll der Winkel des Kreisels auch als Zustand aufgenommen werden, sowie das Fehlerintegral des Winkels, um auch bei dauerhaften Störungen zu garantieren, dass der Kreisel in der Mitte bleibt. Die restlichen Zustände sowie deren Ableitungen werden wie folgt gewählt:

$$u = \dot{\psi}_K \quad (6.45)$$

$$x_1 = \varphi \quad (6.46)$$

$$x_2 = \dot{\varphi} \quad (6.47)$$

$$x_3 = \psi_K \quad (6.48)$$

$$x_4 = \int \psi_K \, dt \quad (6.49)$$

$$\dot{x}_1 = x_2 \quad (6.50)$$

$$\dot{x}_2 = \frac{1}{J_{x,\tau}} \cdot (J_K \cdot w_K \cdot u + F_G \cdot h_0 \cdot x_1) \quad (6.51)$$

$$\dot{x}_3 = u \quad (6.52)$$

$$\dot{x}_4 = x_3 \quad (6.53)$$

Damit ergibt sich das folgende System:

$$\begin{bmatrix} \dot{x}_1 \\ \dot{x}_2 \\ \dot{x}_3 \\ \dot{x}_4 \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 & 0 \\ \frac{F_G \cdot h_0}{J_{x,\tau}} & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} + \begin{bmatrix} 0 \\ \frac{J_K \cdot w_K}{J_{x,\tau}} \\ 1 \\ 0 \end{bmatrix} u \quad (6.54)$$

Das Zustandsmodell ist vollständig steuerbar (ermittelt mit dem Python-Skript *roll_analysis.py*).

Regelung und Simulation im Zustandsraum

Die Wahl der Gewichtungsfaktoren und der diskrete Reglerentwurf erfolgt analog zu Unterabschnitt 6.2.2. Als maximale Werte werden die in Tabelle 6.2 angegebenen Größen verwendet:

Tabelle 6.2: Maximal zulässige Werte der Zustände des Teilsystems Rollen

Zustand	Physikalische Größe	Maximaler Wert
x_1	φ	0,02 rad
x_2	$\dot{\varphi}$	0,02 rad s ⁻¹
x_3	ψ_K	0,17 rad
x_4	$\int \psi_K$	1 rad s
u	$\dot{\psi}_K$	5 rad s ⁻¹

Damit ergeben sich folgende Gewichtungen:

$$Q = \begin{bmatrix} \frac{1}{0,02} & 0 & 0 & 0 \\ 0 & \frac{1}{0,02} & 0 & 0 \\ 0 & 0 & \frac{1}{0,17} & 0 \\ 0 & 0 & 0 & \frac{1}{1} \end{bmatrix}, \quad R = \frac{1}{5} \quad (6.55)$$

Diese Gewichtungen können als Ausgangszustand für weitere Optimierungen verwendet werden. Dazu wird im selben Python-Skript *roll_LQR.py* das Verhalten des Systems mit Zustandsregler simuliert. Es wird wie bisher ein Aufrichtvorgang, eine dauerhafte und eine kurzzeitige Störung simuliert. Durch die schrittweise Anpassung von Q und R ergibt sich für die folgende Gewichtung ein Regler mit realistischem Stellgrößenverlauf und guter Dynamik:

$$Q = \begin{bmatrix} \frac{1}{0,02} & 0 & 0 & 0 \\ 0 & \frac{1}{0,02} & 0 & 0 \\ 0 & 0 & \frac{1}{0,17} \cdot 10 & 0 \\ 0 & 0 & 0 & \frac{1}{1} \end{bmatrix}, \quad R = \frac{1}{5} \cdot 100, \quad K = \begin{bmatrix} 21,876\,329\,95 \\ 3,161\,821\,22 \\ -1,831\,431\,59 \\ -0,214\,242\,86 \end{bmatrix} \quad (6.56)$$

Das simulierte Systemverhalten ist in Abbildung 6.19 zu sehen:

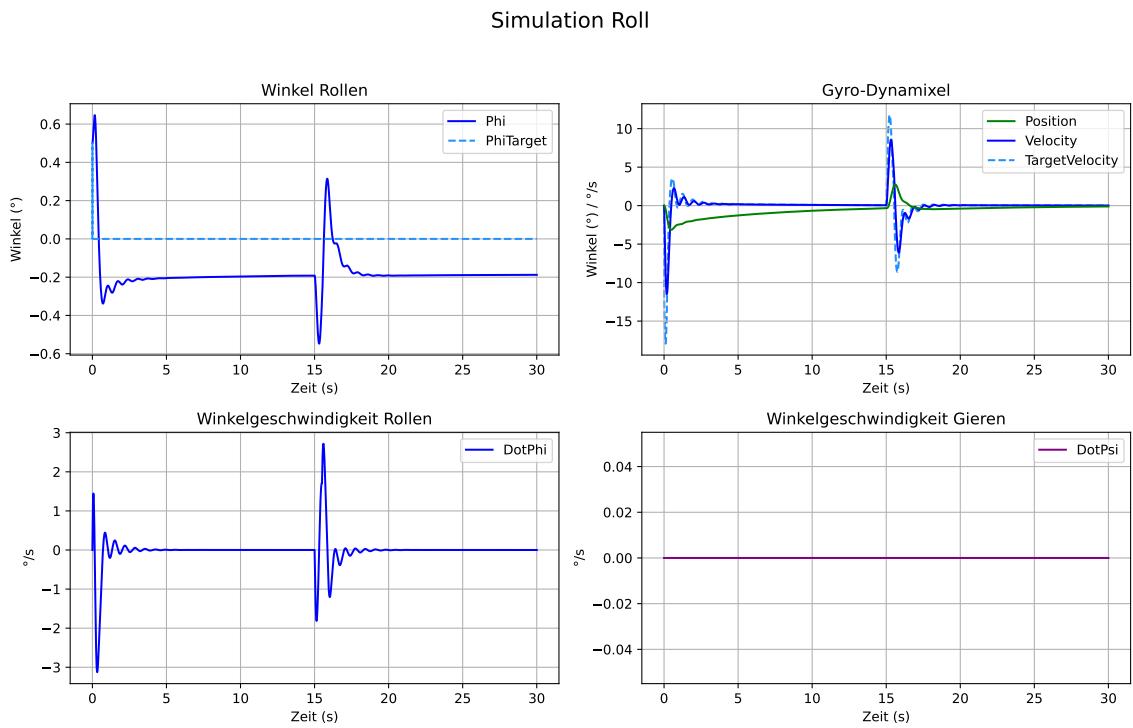


Abbildung 6.19: Rollen Zustandsregler ohne Modellierung des Aktors

Die Simulation zeigt die Entkopplung durch die Haftriebung, da sich das Fahrzeug trotz Rollbewegung nicht um die Hochachse dreht. Allerdings schwingt das System. Das liegt an der zusätzlichen Phasenverschiebung durch das Stellglied, da der Aktor der vorgegebenen Trajektorie nicht ausreichend schnell folgen kann (siehe Gyro-Dynamixel-Plot in Abbildung 6.19). Dieses Problem kann auf zwei Arten gelöst werden: Der Regler wird so langsam gemacht, dass die Verzögerung durch das Stellglied irrelevant wird, oder die Dynamik des Stellglieds wird mit in das Zustandsmodell aufgenommen. Der Regler lässt sich mit dem LQR-Ansatz aber nicht langsamer machen, ohne die Dynamikeigenschaften drastisch zu verschlechtern. Daher wird die in Unterabschnitt 4.1.1 ermittelte Stellglieddynamik als PT1-Glied mit in das Zustandsmodell aufgenommen. Die Winkelgeschwindigkeit $\dot{\psi}_K$ wird zum neuen Zustand. Der Eingang des Systems u ist nun die Steuergröße $\dot{\psi}_{Kcmd}$ für das Stellglied. Der Zusammenhang zwischen der Winkelgeschwindigkeit des Kreisels und der Steuergröße wird durch ein PT1-Glied beschrieben:

$$K \cdot \dot{\psi}_{Kcmd} = T \cdot \dot{\psi}_K + \psi_K \quad (6.57)$$

Damit ergibt sich das erweiterte System:

$$u = \dot{\psi}_{Kcmd} \quad (6.58)$$

$$x_1 = \varphi \quad (6.59)$$

$$x_2 = \dot{\varphi} \quad (6.60)$$

$$x_3 = \psi_K \quad (6.61)$$

$$x_4 = \int \psi_K dt \quad (6.62)$$

$$\dot{x}_5 = \dot{\psi}_K \quad (6.63)$$

$$\dot{x}_1 = x_2 \quad (6.64)$$

$$\dot{x}_2 = \frac{1}{J_{x,\tau}} \cdot (J_K \cdot w_K \cdot x_5 + F_G \cdot h_0 \cdot x_1) \quad (6.65)$$

$$\dot{x}_3 = x_5 \quad (6.66)$$

$$\dot{x}_4 = x_3 \quad (6.67)$$

$$\dot{x}_5 = \frac{1}{T} \cdot (K \cdot u - x_5) \quad (6.68)$$

Damit ergibt sich das folgende System:

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

Das beste dynamische Verhalten ergab sich in der Simulation für die folgenden Gewichtungsfaktoren:

$$Q = \begin{bmatrix} \frac{1}{0,02} & 0 & 0 & 0 & 0 \\ 0 & \frac{1}{0,02} & 0 & 0 & 0 \\ 0 & 0 & \frac{1}{0,17} \cdot 10 & 0 & 0 \\ 0 & 0 & 0 & \frac{1}{1} \cdot 100 & 0 \\ 0 & 0 & 0 & 0 & \frac{1}{5} \end{bmatrix}, \quad R = \frac{1}{5} \cdot 1000, \quad K = \begin{bmatrix} 41,570\,314\,79 \\ 5,308\,060\,25 \\ -1,596\,561\,93 \\ -0,683\,466\,2 \\ 1,966\,801\,53 \end{bmatrix} \quad (6.70)$$

Das damit simulierte Systemverhalten ist in Abbildung 6.20 zu sehen:

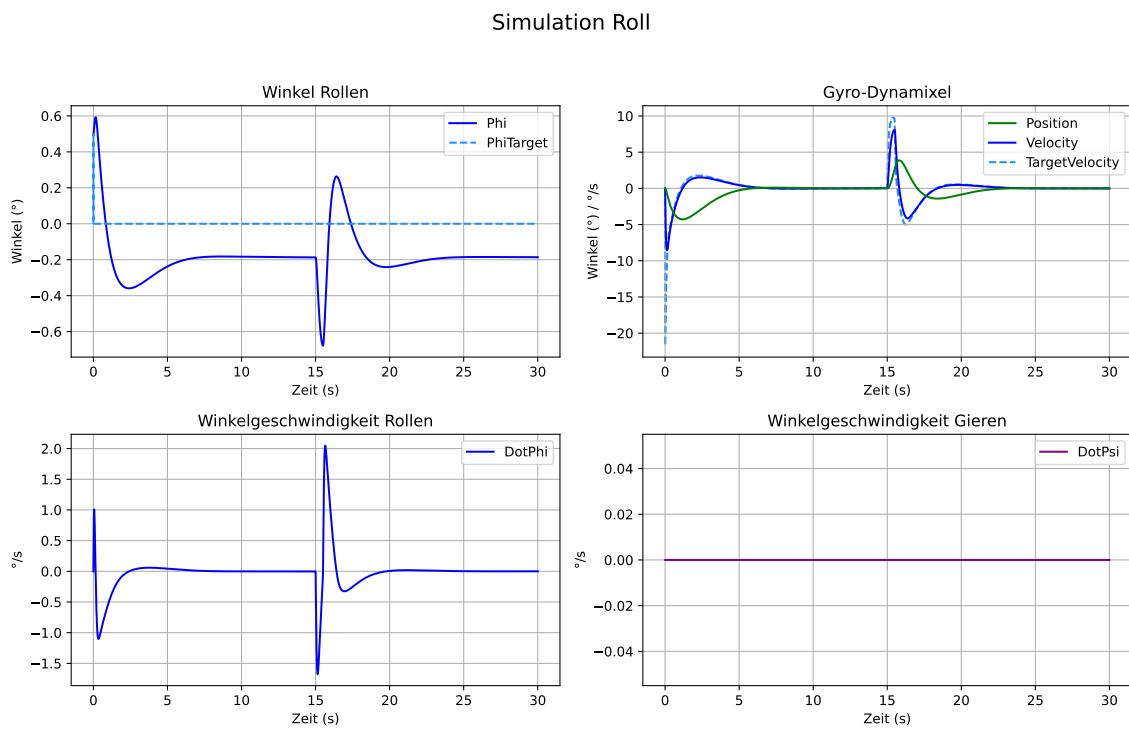


Abbildung 6.20: Rollen Zustandsregler mit Modellierung des Aktors

Das Systemverhalten hat sich deutlich verbessert und die Schwingungen treten nicht mehr auf. Allerdings ist zu sehen, dass der Regler beim Aufrichten einen großen Sollwertsprung auf den Aktor gibt. In Unterabschnitt 4.1.1 wird gezeigt, dass der Aktor bei großen Sollwertsprüngen schwingen kann, und damit das System instabil macht. Um das gute Störverhalten beizubehalten, aber den großen Sollwertsprung beim Aufrichten zu verhindern, wird eine Trajektorie vorgegeben. Zustandsregler sind eigentlich dafür ausgelegt, ein System in einer Ruhelage zu stabilisieren und können ohne weitere Maßnahmen keinem Sollwert folgen. Durch einen Trick kann aber ein ähnliches Verhalten erreicht werden. Dabei wird der Zustandsvektor so manipuliert, dass der Rollwinkel nicht mehr direkt als Zustand genutzt wird, sondern die Differenz zwischen dem aktuellen Rollwinkel und dem vorgegebenen Rollwinkel. Um große Sprünge zu vermeiden, wird der Sollwert mit einem Tiefpass erster Ordnung gefiltert. Die Umsetzung in der Simulation ist in Listing 7 ausschnittsweise gezeigt:

```

1 ctrl_phi = 0
2 ctrl_phi_target = 0
3 ctrl_w = 0
4 ctrl_GF = 0.98
5 ...
6 # Initial condition
7 phi[0] = np.deg2rad(0.5)
8 ctrl_phi_target = phi[0]
9 ctrl_w = 0
10 for n in range(1, steps):
11     ...
12     ctrl_phi_target = (1-ctrl_GF)*ctrl_w + ctrl_GF*ctrl_phi_target
13     ctrl_dot_psi_K_target = -np.clip(Kd[0, 0]*(ctrl_phi-ctrl_phi_target)
14                                         + Kd[0, 1]*ctrl_dot_phi
15                                         + Kd[0, 2]*ctrl_psi_K
16                                         + Kd[0, 3]*ctrl_psi_K_sum
17                                         + Kd[0, 4]*ctrl_dot_psi_K,
18                                         -dot_psi_K_max, dot_psi_K_max)
19     ...

```

Listing 7: Rollen Trajektorie zum Aufrichten

Das Systemverhalten ist in Abbildung 6.21 zu sehen:

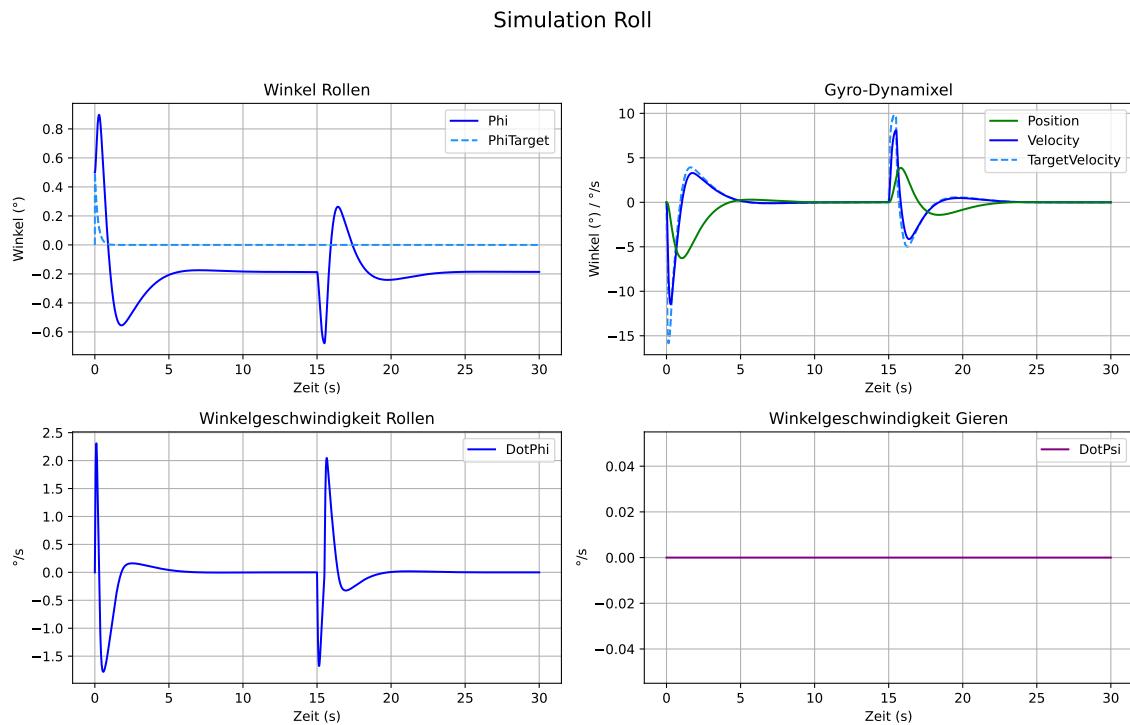


Abbildung 6.21: Rollen Zustandsregler mit Modellierung des Aktors und Trajektorie

Der LQR-Regler erfüllt grundsätzlich alle Anforderungen zum Balancieren. Das Aufrichten

wird mit einem realistischem Stellgrößenaufwand erreicht, kurzzeitige Störungen werden effektiv unterdrückt, und dauerhafte Störungen können durch eine gezielte Abweichung des Rollwinkels kompensiert werden. Der Regler balanciert das Fahrzeug also automatisch aus, in dem das Fahrzeug passend zur Störung schräg gestellt wird. Auf diese Weise ist keine dauerhafte Stellgröße erforderlich, und der Kreisel kann in der Mitte gehalten werden.

Im direkten Vergleich der Simulation des Zustandsreglers (Abbildung 6.21) und des PID-Reglers (Abbildung 6.16) zeigt sich, dass das dynamische Verhalten des Zustandsreglers nur geringfügig besser ist als das des PID-Reglers. Der PID-Regler hat zudem den Vorteil, ohne weitere Maßnahmen Trajektorien oder Sollwertsprünge folgen zu können - eine Anforderung, um Experiment zum Kurvenfahren durchführen zu können. Der Zustandsregler hingegen versucht, alle Zustände auf null zur regeln. Selbst mit dem Trick des manipulierten Zustands bleiben aufgrund des fehlenden Integrators für den Rollwinkel dauerhafte Regelabweichungen. Ein Integrator darf jedoch nicht eingeführt werden, da gerade diese bleibende Abweichung notwendig ist, um das Fahrzeug durch eine leichte Schrägstellung dauerhaft gegen Störungen auszubalancieren.

Um das Erreichen verschiedener Sollwerte dennoch mit einem Zustandsregler zu ermöglichen, wäre daher ein zweiter, speziell angepasster Regler notwendig, der den Kreiselwinkel und dessen Integrator nicht berücksichtigt, dafür aber einen Integrator für den Rollwinkel enthält. Der PID-Regler besitzt diese Aufteilung implizit durch seine getrennte Struktur für Sollwert- und Rollwinkelregelung. Aus diesem Grund wird der PID-Regler im realen Fahrzeug eingesetzt.

6.4 Gesamtsystem Ergebnis

Sowohl das Teilsystem Nicken als auch das Teilsystem Rollen und Gieren konnte erfolgreich stabilisiert werden.

Beim Teilsystem Nicken bietet der Zustandsregler aufgrund der überlegenen Dynamik einen Vorteil gegenüber dem PID-Regler. Um die Schwingungen des Systems in der Ruhelage, sog. Limit Cycles, zu reduzieren, wird außerdem Gain-Scheduling eingesetzt. Das System kann effektiv kurze Störungen unterdrücken, langfristige Störungen ausgleichen und das Fahrzeug aufrichten. Dabei werden starke Beschleunigungen und Winkelgeschwindigkeiten vermieden, um Störungen der Sensoren und des Teilsystems Rollen und Gieren zu reduzieren.

Das Teilsystem Rollen und Gieren kann zum Balancieren durch die Haftreibung des Reifens entkoppelt betrachtet werden. Zur Stabilisierung des Fahrzeugs als SIMO-System mit zwei Ausgängen werden zwei PID-Regler verwendet. Der eine PID-Regler regelt die Rollbewegung, während der andere PID-Regler eine Trajektorie vorgibt, die dafür sorgt, dass der Winkel des Gyroskops im zulässigen Bereich bleibt. Dadurch kann sich das System aufrichten, kurzzeitige Störungen unterdrücken und langfristige Störungen ausgleichen. Durch die Aufteilung der zwei Regler kann außerdem ein gewünschter Rollwinkel erreicht werden, ohne die Reglerstruktur zu verändern. Diese Fähigkeit wird zum Kurvenfahren benötigt. Zusätzlich wurde ein Zustandsregler zur Stabilisierung des Fahrzeugs entworfen. Da der Zustandsregler keine nennenswerten Vorteile hinsichtlich der Dynamik des Systems erzielte, und aufgrund der Reglerstruktur nicht in der Lage ist, Sollwertvorgaben zu folgen, wird das Fahrzeug mit

dem PID-Regler stabilisiert.

Beide Teilsysteme können mit den entworfenen Reglern stabilisiert werden. Daher soll nun die Betrachtung des Fahrzeugs als Gesamtsystem erfolgen. Abbildung 6.22 zeigt alle Systemgrößen beim Balancieren auf einer Stelle. Das Fahrzeug muss sich aufrichten, mehrere kurze Störungen ausgleichen sowie die dauerhaften Störungen durch die Ungenauigkeiten des Fahrzeugs kompensieren.

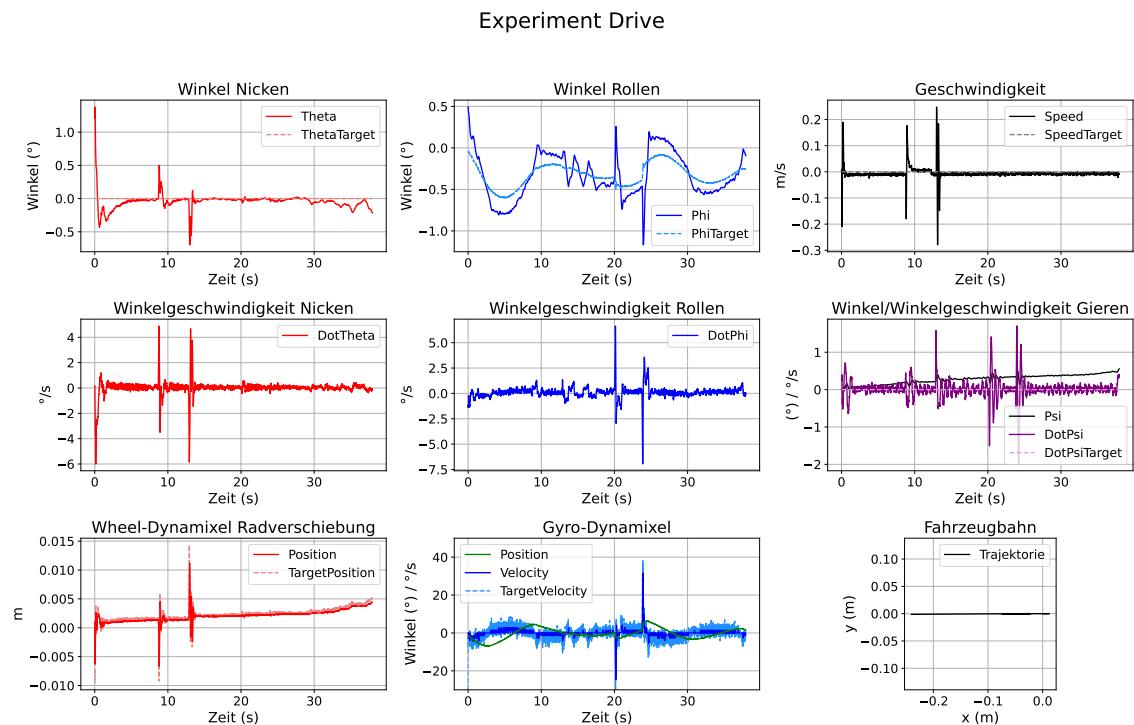


Abbildung 6.22: Systemtest Fahrzeug Balancieren

Das Verhalten der einzelnen Teilsysteme entspricht auch im Gesamtsystem dem Verhalten bei der isolierten Betrachtung. Es ist auch zu erkennen, dass die Teilsysteme sich gegenseitig kaum beeinflussen, was ebenfalls Teil der getroffenen Vereinfachungen ist. Die Gierbewegung verhält sich ebenfalls wie erwartet. Beim Ausgleichen der Störungen in der Rollbewegung gibt es minimale Ausschläge der Gierbewegung, die die Orientierung des Fahrzeugs aber nicht maßgeblich verändern. Das zeigt, dass die Entkopplung der Teilsysteme Rollen und Gieren im Arbeitspunkt für das Balancieren wie erwartet vorliegt.

Auffällig sind die Ausschläge in der Geschwindigkeit des Fahrzeugs. Obwohl das Fahrzeug eigentlich still steht, bewegt sich das Rad mehrfach. Das liegt an der Regelung des Nickens, welche das Rad relativ zur Plattform verschiebt. Da die Trägheit des Rads geringer ist, als die der Plattform, bewegt sich hauptsächlich das Rad.

Um die Bewegungen des Fahrzeugs zu veranschaulichen wird im Post-Processing aus der Winkelgeschwindigkeit der Gierbewegung der Gierwinkel berechnet. Da sich das Fahrzeug immer

im Bereich der Kleinwinkelnäherung befindet, kann davon ausgegangen werden, dass die Gierbewegung im Fahrzeug-Koordinatensystem der Gierbewegung im Welt-Koordinatensystem entspricht. Durch den Bias der Gyroskopdaten driftet der Gierwinkel mit der Zeit, und erweckt den Eindruck, als ob sich das Fahrzeug langsam drehen würde, auch wenn es in der Realität still steht. Gleiches gilt für die Bahn. Sie wird im Post-Processing aus dem Geschwindigkeitsvektor berechnet. Der Betrag ist durch die momentane Geschwindigkeit bekannt, die Richtung ergibt sich aus dem aktuellen Gierwinkel. Der ADC-Wandler zur Bestimmung der aktuellen Geschwindigkeit unterliegt ebenfalls einem Bias, sodass auch die Bahn den Anschein erweckt, als würde sich das Fahrzeug langsam bewegen, auch wenn es still steht. Befindet sich das Fahrzeug in Bewegung, fallen diese Fehler kaum ins Gewicht.

Ist das Fahrzeug in Bewegung, sinkt die Haftreibung des Rads um die Hochachse erheblich. Zusätzlich wurden die Effekte des rollenden Rads bei der Modellierung vernachlässigbar. Daher soll getestet werden, ob das Fahrzeug auch während dem Fahren stabilisiert werden kann. Dazu richtet sich das Fahrzeug auf und wird auf eine vorgegebene Geschwindigkeit beschleunigt und wieder bis zum Stillstand abgebremst. Dabei ergibt sich das folgende Verhalten des gesamten Systems (siehe Abbildung 6.23):

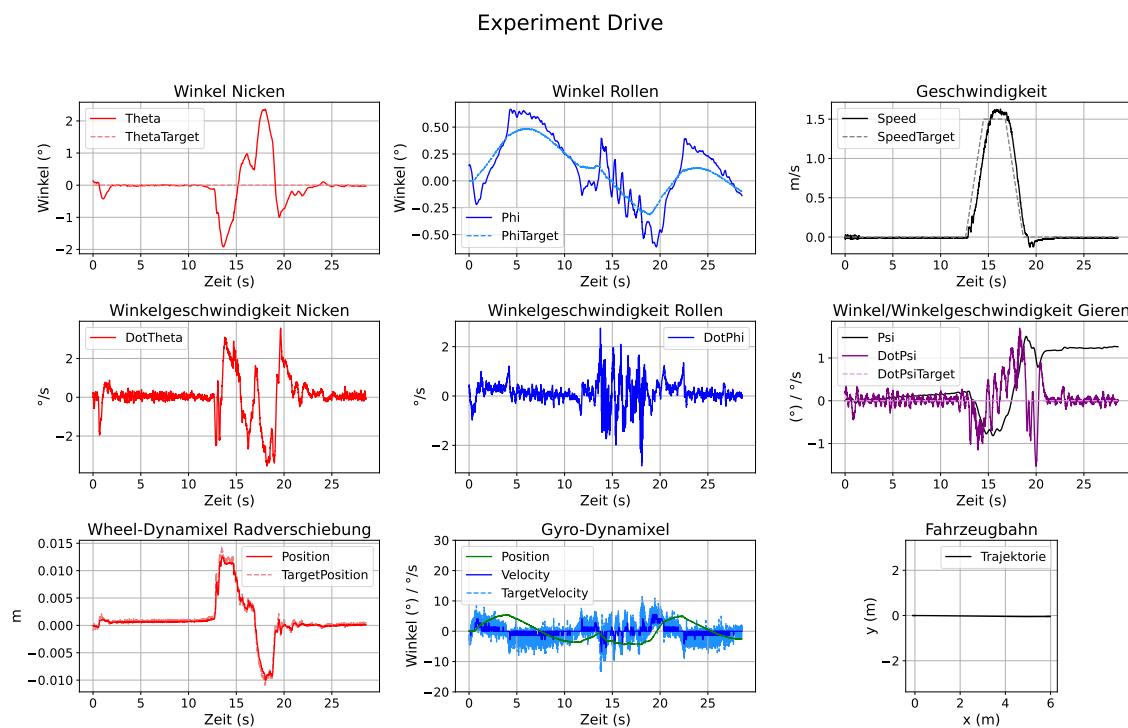


Abbildung 6.23: Systemtest Fahrzeug Linie fahren

Das Fahrzeug kann auch beim Fahren erfolgreich stabilisiert werden. Wie erwartet treten während der Fahrt größere Gierbewegung auf, als im Stand. Allerdings reicht die Reibung des Rads aus, um die Gierbewegung gegen null konvergieren zu lassen. Das System ist beim Fahren aber deutlich empfindlicher als im Stand. Ist das Fahrzeug nicht richtig ausbalanciert, kann es im Stand ohne Probleme balancieren, indem es sich schräg stellt. Während

des Fahrens zwingt die Schräglage das Rad durch das Rolling-Constraint aber auf eine Kreisbahn [43][4, S, 16ff], sodass das Fahrzeug eine unerwünschte Kurve fährt. Dadurch wird die Entkopplung zwischen Rollen und Gieren aufgehoben, und das Fahrzeug kann nicht mehr effektiv stabilisiert werden. Ist das Fahrzeug aber ausreichend gut ausbalanciert, ist die Schräglage so klein, dass dieser Effekt nicht auftritt. Das Fahrzeug kann erfolgreich geradeaus fahren.

Der Vergleich des simulierten Systemverhaltens mit dem Verhalten des realen Systems hat gezeigt, dass die getroffenen Vereinfachungen bei der Modellierung in Kapitel 3 die für das Balancieren maßgebliche Dynamik ausreichend abbilden. Anhand des Modells und der Simulation konnten robuste Regler für beide Teilsysteme entworfen werden, deren dynamisches Verhalten am realen Fahrzeug validiert werden konnte. Somit kann der *Monowheeler* erfolgreich stabilisiert werden.

7 Regelung und Trajektorien zum Kurvenfahren

Während in den vorherigen Kapiteln der Schwerpunkt auf der Stabilisierung des Fahrzeugs liegt, wird in diesem Kapitel das Kurvenfahren betrachtet. Ziel ist zu untersuchen, ob mit der vorhandenen Hardware grundlegende Manöver zur Richtungsänderung möglich sind.

Dabei ist zu beachten, dass die Arbeit bewusst auf einem vereinfachten Modell des Fahrzeugs aufbaut (siehe Kapitel 3) und sich auf klassische Reglerstrategien wie PID- und LQR-Regler beschränkt. PID-Regler sind eigentlich nicht für SIMO-Systeme ausgelegt, während LQR-Regler nur zur Stabilisierung und nicht zum Folgen von Trajektorien gedacht sind. Moderne, für dieses Problemfeld geeignete Verfahren wie nichtlineare oder lineare, modellprädiktive Regler zum Folgen von Trajektorien (NMPC bzw. LTV-MPC) oder fortgeschrittene Trajektorienplaner werden hier nicht berücksichtigt, da dies über den Rahmen dieser Arbeit hinaus geht. Stattdessen soll ein Proof-of-Concept geliefert werden, der zeigen soll, dass das System grundsätzlich in der Lage ist, komplexe Manöver wie Richtungsänderungen durchzuführen.

Das Kapitel versteht sich daher als explorativer Ansatz, der vor allem den praktischen Nachweis liefern soll, dass das entwickelte Fahrzeug prinzipiell in der Lage ist, Kurvenfahrten durchzuführen.

7.1 Direkte Regelungskonzepte

Zunächst soll überprüft werden, ob ein klassischer Regler ausgelegt werden kann, der das gekoppelte Teilsystem aus Rollen und Gieren steuern kann. Damit könnte das Fahrzeug aktiv Kurven fahren. Da es sich um ein SIMO-System handelt, weil sowohl das Rollen als auch das Gieren gesteuert werden muss, soll ein LQR-Regler zum Einsatz kommen. Dazu muss das Differenzialgleichungssystem Gleichung 3.8 linearisiert werden. Mit der Kleinwinkelnäherung ergibt sich das folgende lineare System:

$$J_{x,\tau} \cdot \ddot{\varphi} = J_K \cdot w_K \cdot (\dot{\psi} + \dot{\psi}_K) + F_G \cdot h_0 \cdot \varphi + m_{ges} \cdot v \cdot \dot{\psi} \cdot h_0 \quad (7.1)$$

$$J_z \cdot \ddot{\psi} = -J_K \cdot w_K \cdot \dot{\varphi} \quad (7.2)$$

Die Überführung des Modells in den Zustandsraum erfolgt analog zu Unterunterabschnitt 6.3.2, nur mit der Erweiterung auf die Gierbewegung. Der Eingang des Systems u ist die Winkelgeschwindigkeit des Kreisels. Wie bereits beschrieben, soll der Winkel des Kreisels auch als Zustand aufgenommen werden, sowie das Fehlerintegral des Winkels, um auch bei dauerhaften Störungen zu garantieren, dass der Kreisel in der Mitte bleibt. Die restlichen Zustände sowie deren Ableitungen werden wie folgt gewählt:

$$u = \dot{\psi}_K \quad (7.3)$$

$$x_1 = \varphi \quad (7.4)$$

$$x_2 = \dot{\varphi} \quad (7.5)$$

$$x_3 = \dot{\psi} \quad (7.6)$$

$$x_4 = \psi_K \quad (7.7)$$

$$x_5 = \int \psi_K \, dt \quad (7.8)$$

$$\dot{x}_1 = x_2 \quad (7.9)$$

$$\dot{x}_2 = \frac{1}{J_{x,\tau}} \cdot (J_K \cdot w_K \cdot (x_3 + u) + F_G \cdot h_0 \cdot x_1 + m_{ges} \cdot v \cdot x_3 \cdot h_0) \quad (7.10)$$

$$\dot{x}_3 = -\frac{1}{J_z} \cdot (J_K \cdot w_K \cdot x_2) \quad (7.11)$$

$$\dot{x}_4 = u \quad (7.12)$$

$$\dot{x}_5 = x_4 \quad (7.13)$$

Damit ergibt sich das folgende System:

$$\begin{bmatrix} \dot{x}_1 \\ \dot{x}_2 \\ \dot{x}_3 \\ \dot{x}_4 \\ \dot{x}_5 \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ \frac{F_G \cdot h_0}{J_{x,\tau}} & 0 & \frac{J_K \cdot w_K + m_{ges} \cdot v \cdot h_0}{J_{x,\tau}} & 0 & 0 \\ 0 & -\frac{J_K \cdot w_K}{J_z} & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{bmatrix} + \begin{bmatrix} 0 \\ \frac{J_K \cdot w_K}{J_{x,\tau}} \\ 0 \\ 1 \\ 0 \end{bmatrix} u \quad (7.14)$$

Wie in Unterabschnitt 6.2.2 beschrieben wird für dieses Zustandsmodell die Steuerbarkeit im Python-Skript *roll_yaw_analysis.py* geprüft. Dabei ergibt sich, dass nur vier der fünf Zustände steuerbar sind. Der nicht steuerbare Zustand ist dabei die Gierbewegung. Das System ist daher nicht vollständig steuerbar, und ein direkter Zustandsregler wie LQR kann das System zwar stabilisieren, aber die Gierbewegung nicht direkt beeinflussen. Auch eine Erweiterung des Modells zur vollständigen Erfassung der Dynamik beim Kurvenfahren verändert die Steuerbarkeit nicht (vgl. [14, S. 4]).

7.2 Trajektorienbasierte Steuerung

Da eine direkte Regelung der Gierbewegung im Gesamtsystem im Rahmen dieser Arbeit nicht praktikabel ist, muss die Kopplung zwischen Rollen und Gieren genutzt werden, um die Gierbewegung gezielt zu beeinflussen. Wie in Kapitel 3 beschrieben, erzeugt eine Rollbewegung durch den gyroskopischen Effekt ein Moment um die Hochachse des Fahrzeugs. Durch eine gezielte Rollbewegung muss die Haftreibung des Rads kurzzeitig überwunden werden, sodass eine Gierbewegung hervorgerufen wird. Anschließend wird die Rollgeschwindigkeit so angepasst, dass das Fahrzeug in den Arbeitspunkt zum Balancieren ($\varphi \approx 0$, $\dot{\psi} \approx 0$) zurückkehrt. Bei diesem Manöver darf das Fahrzeug den zulässigen Bereich des Rollwinkels und des Kreiselwinkels nicht verlassen, da das Fahrzeug sonst nicht mehr stabilisiert werden kann. Dieses Ziel kann theoretisch durch zwei Arten der Trajektorienvorgabe erreicht werden, die im Folgenden vorgestellt werden.

Die Trajektorien der Rollbewegung beim Kurvenfahren und Balancieren werden weiterhin von dem in Abschnitt 6.3 ausgelegten PID-Regler für die Steuerung der Rollbewegung umgesetzt.

7.2.1 Passive Trajektorie

In Kapitel 3 wird der selbststabilisierende Effekt eines Gyroskops im Raum beschrieben. Theoretisch kann dieser Effekt auch für das Kurvenfahren verwendet werden. Dazu wird das Fahrzeug absichtlich zum Umkippen gebracht. Dann wird der Kreisel vom Aktor fixiert, sodass die Kreiselmomente direkt auf das Fahrzeug übertragen werden. Der Kreisel erzeugt durch die Kippbewegung ein Moment um seine Hochachse. Dadurch beginnt das Fahrzeug eine Kurve zu fahren. Die Drehung um die Hochachse erzeugt wiederum ein Kreiselmoment, dass der ursprünglichen Rollbewegung entgegenwirkt. Sind die Kreiselmomente groß genug, wird die Kippbewegung des Fahrzeugs abgebremst, bis sich ein Gleichgewicht einstellt, bei dem das Fahrzeug seinen Rollwinkel weitestgehend beibehält, während es eine Kurve fährt. Anschließend kann das Fahrzeug durch das gezielte Ansteuern des Kreisels wieder aufgerichtet werden. Da nach der Einleitung des Umfallens keine aktive Ansteuerung des Systems erfolgt, wird dieses Vorgehen als passiv bezeichnet. In der Simulation verhält sich das System wie erwartet (siehe Abbildung 7.1):

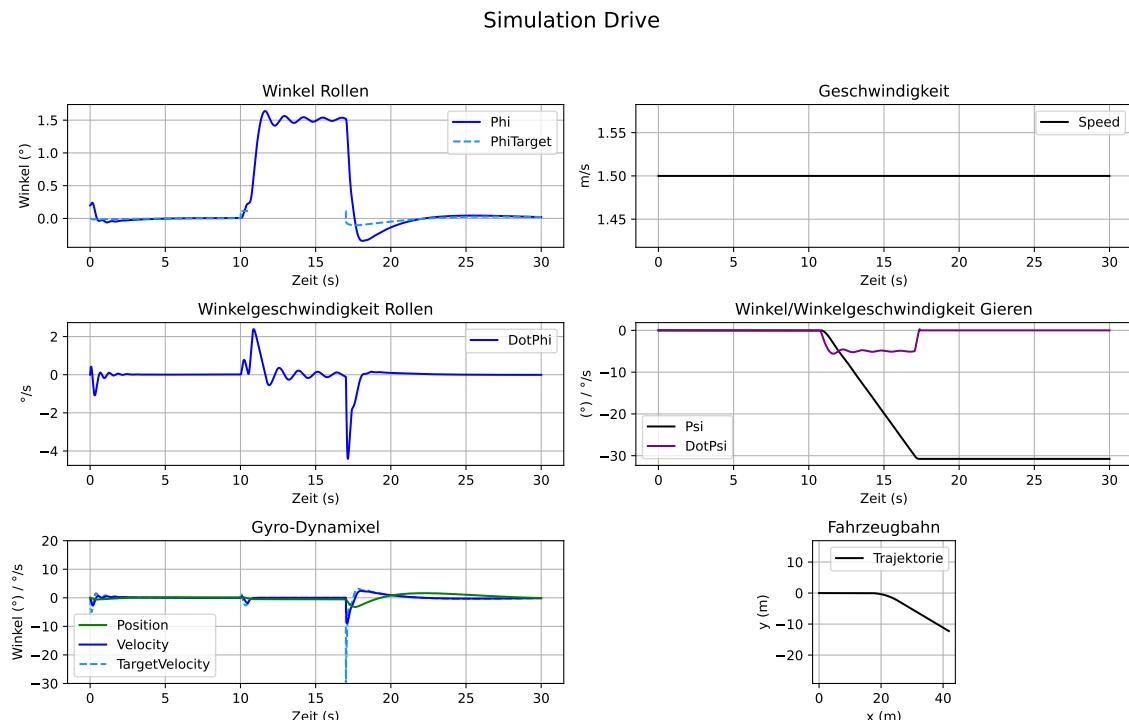


Abbildung 7.1: Simulation Fahrzeug Kurvenfahren durch passive Trajektorie

Bei $t = 10\text{ s}$ wird das Fahrzeug durch eine Sollwertvorgabe des Rollwinkels zum Kippen gebracht. Kurze Zeit später wird der Regler ausgeschaltet und der Aktor fixiert den Kreisel. Das Fahrzeug kippt dann bis zum Erreichen des eben beschriebenen Gleichgewichts. Anschließend kann das Fahrzeug aufgerichtet werden. In der Theorie können damit fast unbegrenzte Richtungsänderungen vorgenommen werden, da beim Kurvenfahren keine Stellgröße erforderlich ist, sodass der Kreisel seinen zulässigen Winkelbereich nicht verlässt.

Allerdings lässt sich dieses Prinzip nicht direkt auf dem *Monowheeler* umsetzen. Um das Fahrzeug in einem Winkelbereich stabilisieren zu können, aus dem es sich wieder aufrichten kann, ist eine deutlich höhere Drehzahl des Kreisels vonnöten. Aus der Simulation und Beobachtungen des realen Systems ergibt sich, dass die Drehzahl des Kreisels von dem aktuellen Wert von 5000 rpm auf mind. 7000 rpm gesteigert werden müsste. Dies ist ohne weitere Maßnahmen aufgrund des hohen Leistungsbedarfs des Antriebs momentan nicht möglich. Außerdem ist die Roll- und Gierbewegung unkontrolliert, sodass die Drehrate um die Hochachse, und somit der Kurvenradius, kaum beeinflusst werden können. Hinzu kommt, dass das System auf diese Weise nicht auf Störungen oder Umweltfaktoren eingehen kann und dadurch weniger robust ist.

7.2.2 Aktive Trajektorie

Da das Kurvenfahren durch eine passive Vorgabe der Rollbewegung mit dem aktuellen Aufbau nicht praktikabel ist, und zusätzlich eine direkte Steuerbarkeit der Gierbewegung gewünscht ist, soll aktiv eine Trajektorie für die Rollbewegung vorgegeben werden.

Feste Trajektorie

Die einfachste Trajektorie für die Rollbewegung, mit der eine Gierbewegung erzeugt werden kann, ist die Vorgabe einer festen Rollgeschwindigkeit. Damit soll am realen System getestet werden, ob das Konzept der aktiven Steuerung der Rollbewegung zum Kurvenfahren prinzipiell funktioniert. Zuerst wird das Fahrzeug auf eine Geschwindigkeit von $v = 1,5 \text{ m s}^{-1}$ beschleunigt. Anschließend wird die Sollwertvorgabe für die Rollbewegung nicht mehr durch den PID-Regler zum Balancieren generiert, sondern eine feste Rollgeschwindigkeit vorgegeben. Am Ende wird das Fahrzeug wieder aufgerichtet, indem die Sollwertvorgabe für den Rollwinkel wieder durch den PID-Regler zum Balancieren vorgenommen wird. Die Ergebnisse des Experiments sind in Abbildung 7.2 zu sehen.

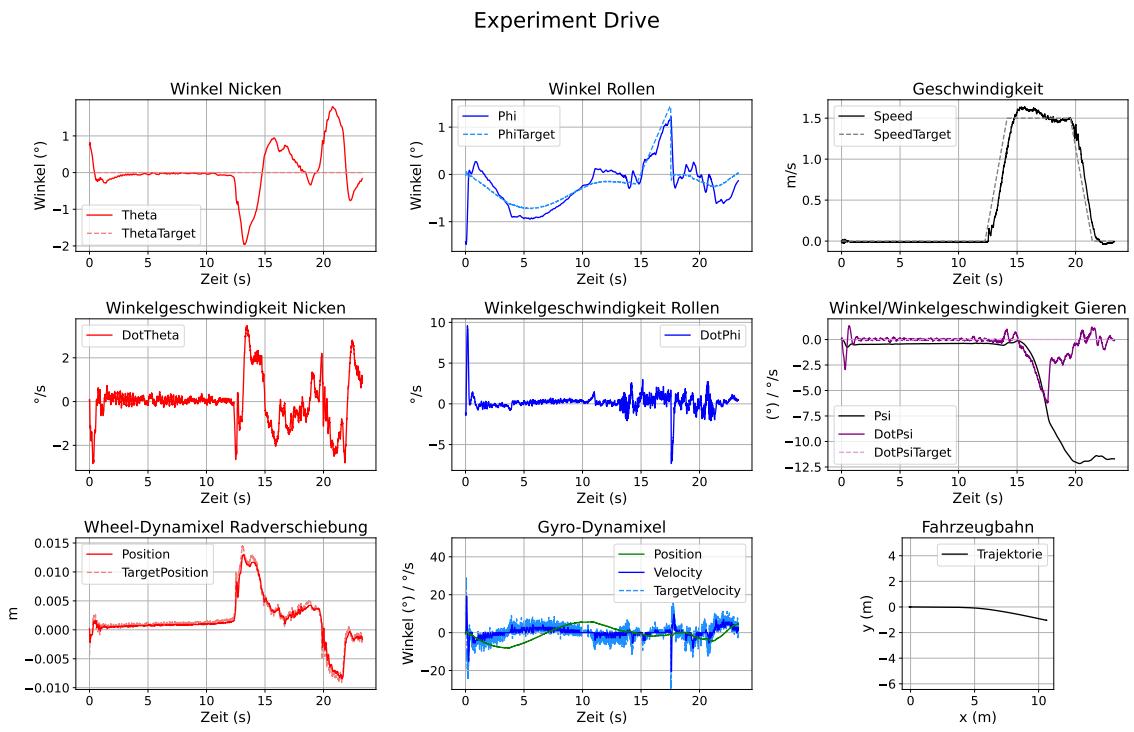


Abbildung 7.2: Systemtest Fahrzeug Kurvenfahren mit fester Roll-Trajektorie

Mit dem vorgestellten Konzept lassen sich mit dem *Monowheeler* aktiv Kurven fahren. Durch die vorgegebene Rollgeschwindigkeit wird die Haftreibung des Rads kurzzeitig überwunden und gezielt eine Drehung um die Hochachse verursacht. Allerdings ist das Kurvenfahren mit dieser Methode nicht sehr robust. Geringe Unterschiede in der Balance des Fahrzeugs und der Anfangsbedingungen beeinflussen die Kurvenfahrt stark, teilweise bis zur Destabilisierung des Systems.

Reglerbasierte Trajektorie

Um das Verhalten beim Kurvenfahren robuster zu machen und besser steuern zu können, wird anstelle der festen Trajektorie eine reglerbasierte Trajektorie eingesetzt. Dazu wird ein PID-Regler entworfen, der die Giergeschwindigkeit regelt. Die Stellgröße ist dabei die Rollgeschwindigkeit, aus der dann eine Trajektorie für die Rollbewegung generiert werden kann. So entsteht eine kaskadierte Reglerstruktur (siehe Abbildung 7.3), die der Regelung zum Balancieren ähnelt.

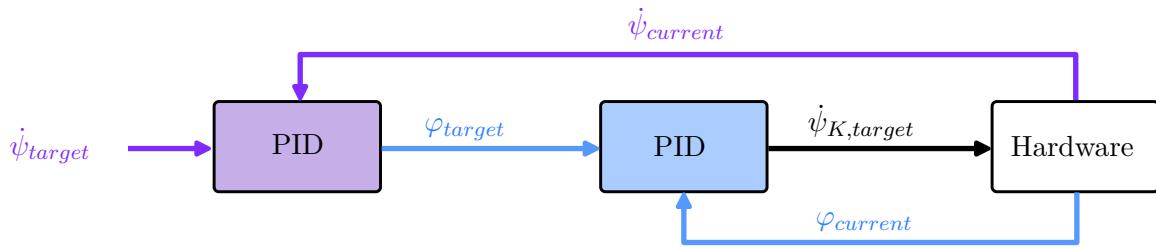


Abbildung 7.3: Struktur kaskadierter PID-Regler zum Kurvenfahren

Die Auslegung erfolgt experimentell in der Simulation. Abbildung 7.4 zeigt das Verhalten des Systems in der Simulation mit den Reglerparametern $KP = 0,8$; $KD = 0,15$; $KI = 0$.

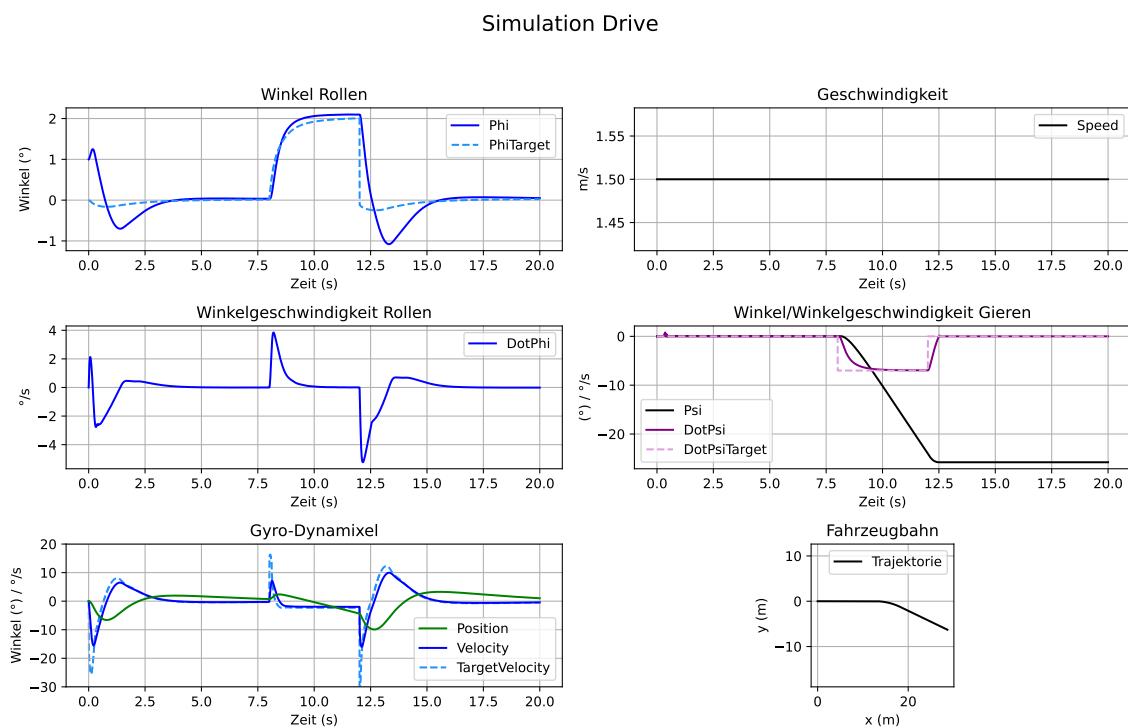
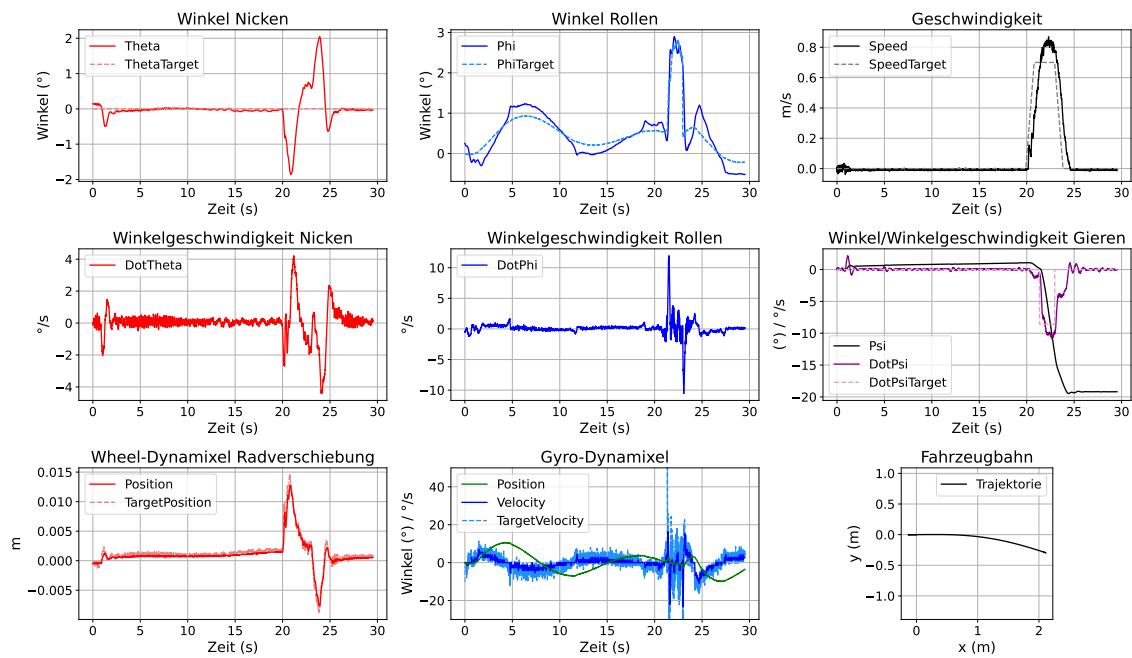


Abbildung 7.4: Simulation Fahrzeug Kurvenfahren durch aktive Trajektorie

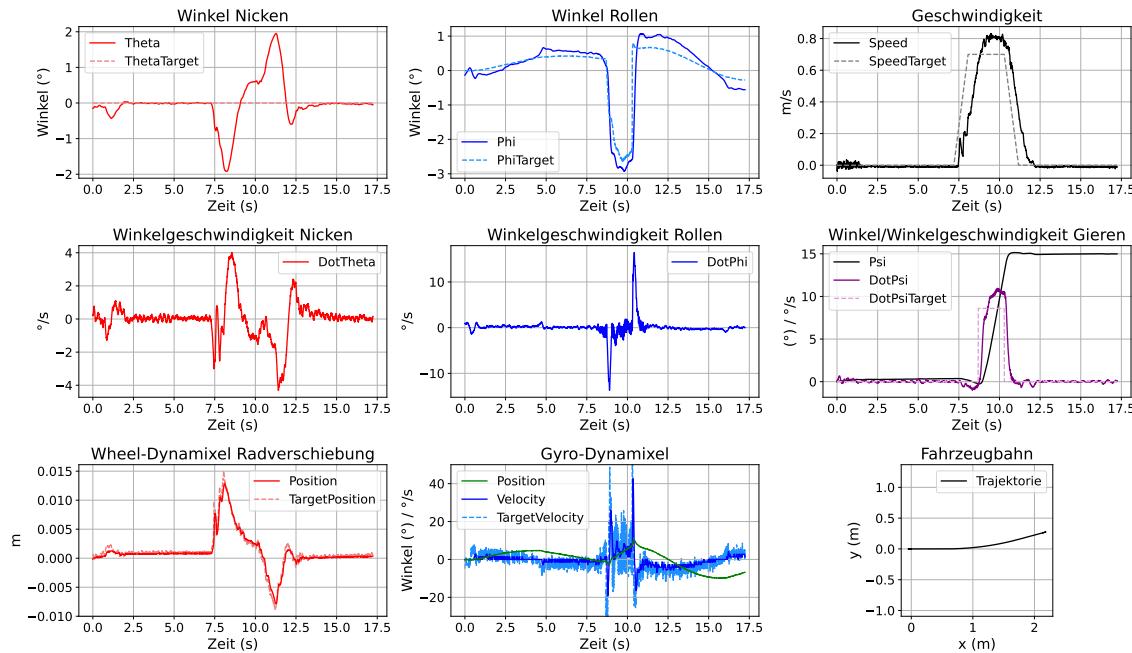
Bei $t = 8\text{ s}$ wird vom PID-Regler zum Balancieren auf den PID-Regler zum Kurvenfahren umgeschaltet. Durch die generierte Trajektorie fährt das Fahrzeug gezielt eine Kurve. Anschließend kann das Aufrichten erfolgen. Das System kann damit in der Simulation erfolgreich eine gezielte Kurve fahren, und das Konzept wird ebenfalls am realen System getestet (siehe Abbildung 7.5a und Abbildung 7.5b).

Experiment Drive



(a) Rechtskurve

Experiment Drive



(b) Linkskurve

Abbildung 7.5: Systemtest Fahrzeug Kurvenfahrt mit reglerbasierter Roll-Trajektorie

Der *Monowheeler* kann mit dem vorgestellten Konzept gezielt Kurven fahren. Die Reglerparameter aus der Simulation werden am realen System erprobt und experimentell verbessert, bis sich mit den Parametern $KP = 1,2$; $KD = 0,2$; $KI = 1,5$ die gezeigten Ergebnisse erzielen lassen. Sowohl Links- und Rechtskurven sind möglich und wie eng oder weit die Kurve ist, kann direkt durch die Vorgabe der Giergeschwindigkeit bestimmt werden. In Abbildung 7.6 ist eine engere Kurve zu sehen.

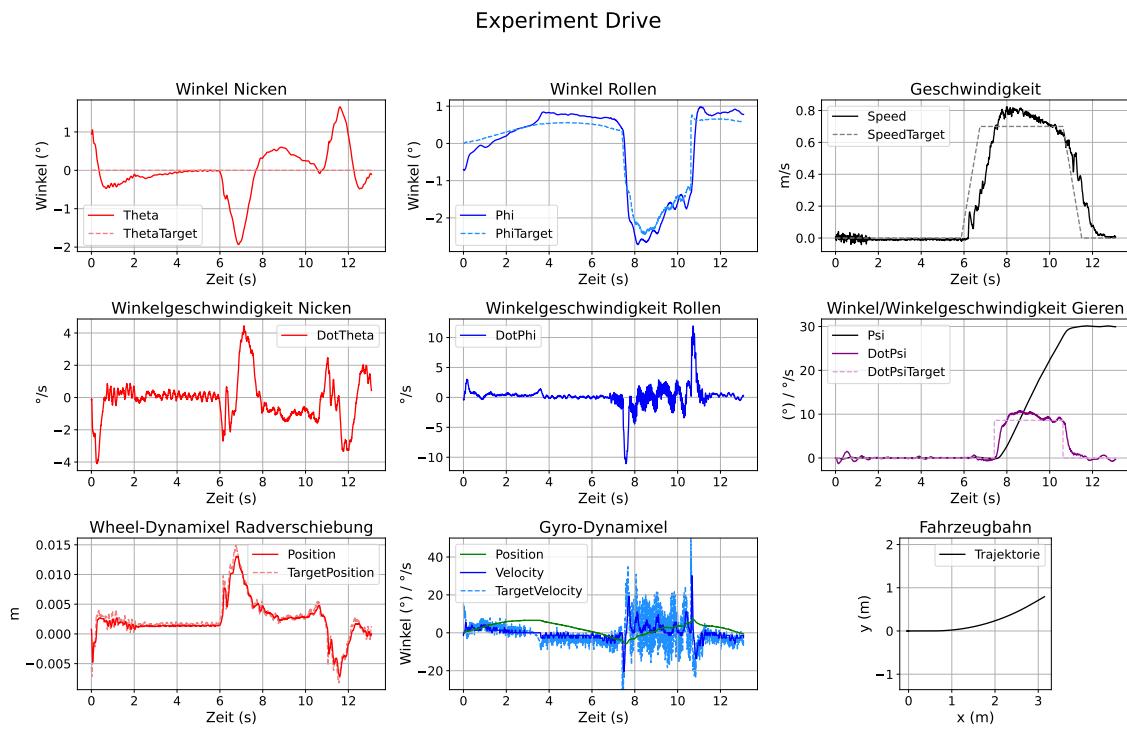


Abbildung 7.6: Systemtest Fahrzeug enge Linkskurve mit reglerbasierter Roll-Trajektorie

Das Kurvenfahren ist nun so robust, dass sich auch komplexere Manöver, wie eine S-Kurve, fahren lassen (siehe Abbildung 7.7). Das zeigt, dass das Kurvenfahren mit dem entwickelten Regler zielgerichtet und reproduzierbar erfolgt, und das System robust gegenüber Störungen ist.

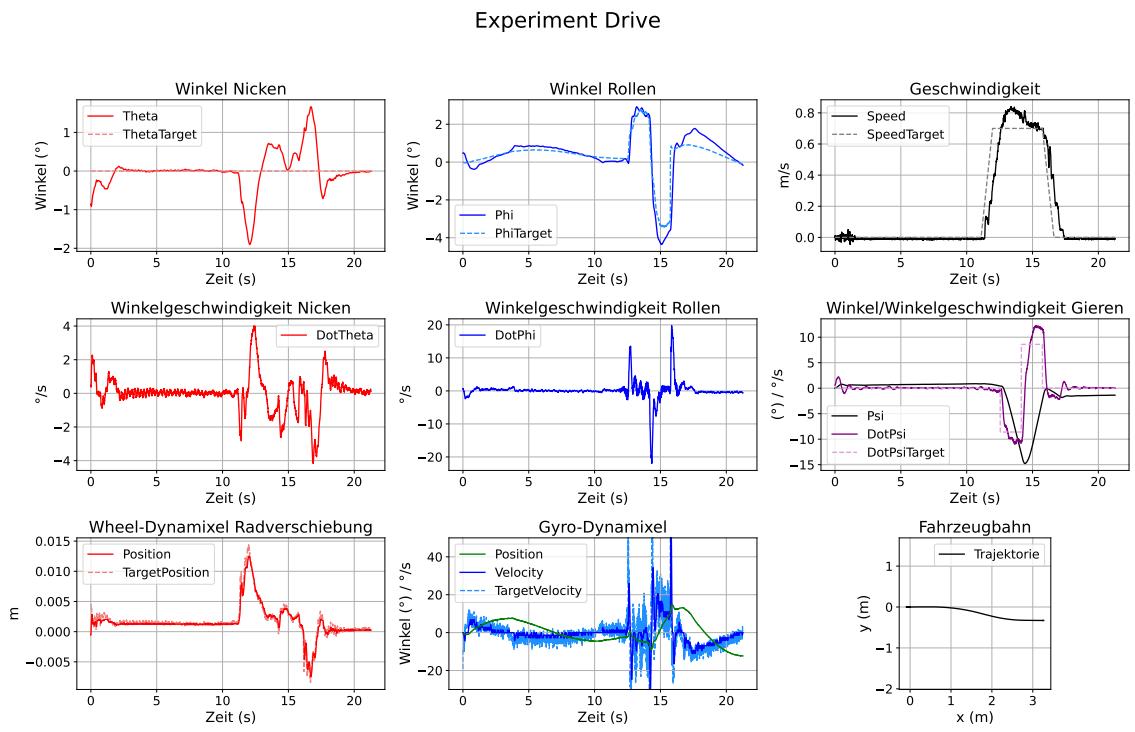


Abbildung 7.7: Systemtest Fahrzeug Slalom mit reglerbasierter Roll-Trajektorie

Trotz dieser erfolgreichen Umsetzung gibt es auch Einschränkungen. Der PID-Regler kann durch den Trick mit der Trajektorie zwar das Kurvenfahren regeln, ist aber kein echter Mehrgrößenregler und kann den Winkel des Kreisels nicht in die Regelung miteinbeziehen. Für die Dauer des Manövers muss also davon ausgegangen werden, dass der Kreisel genügend Winkelreserve hat. Die Trajektorie wird so auch nicht hinsichtlich des Kreiselwinkels optimiert, wie es mit einem komplexeren Algorithmus möglich wäre.

Eine wesentlichere Herausforderung ergibt sich zudem aus der strukturellen Annahme des Regelungskonzepts. Für das Balancieren wird vorausgesetzt, dass die Roll- und Gierbewegung entkoppelt sind. Diese Annahme gilt im Arbeitspunkt ($\varphi \approx 0$, $\dot{\psi} \approx 0$) und ermöglicht dort eine stabile Regelung (siehe Abschnitt 6.3).

Im Stand ist diese Entkopplung sehr stark und das System sehr robust. Beim Fahren wird diese Entkopplung jedoch zunehmend aufgehoben: Die Haftreibung des Rads sinkt signifikant mit wachsender Geschwindigkeit und die nicht modellierten Effekte des Rads (siehe Kapitel 3) beim Kurvenfahren gewinnen an Einfluss, wodurch sich Roll- und Gierdynamik gegenseitig beeinflussen. Gleichzeitig verlässt das Fahrzeug während eines Manövers zum Kurvenfahren seinen Arbeitspunkt. Dadurch fällt es dem System schwer, sich nach größeren Manövern langfristig wieder zu stabilisieren. Teilweise bleibt auch nach dem Aufrichten eine Gierbewegung erhalten, sodass die Haftreibung des Rads bereits überwunden ist. Zusätzlich wird der Kreisel beim Kurvenfahren stark ausgelenkt. Um den Kreisel wieder in die Mitte zu führen, gibt der PID-Regler zum Balancieren eine Rollbewegung vor. Diese ausgeprägte Rollbewegung führt beim Fahren aber zu weiteren Gierbewegungen und destabilisiert das

Fahrzeug zusätzlich.

In der Praxis zeigt sich, dass das System bei geringeren Geschwindigkeiten robuster arbeitet, da die Haftreibung des Rads größer und die dynamische Kopplung zwischen Rollen und Gieren geringer ist. Daher werden die gezeigten Kurvenfahrten mit einer Geschwindigkeit von $v = 0,7 \text{ m s}^{-1}$ durchgeführt, anstelle der $v = 1,5 \text{ m s}^{-1}$ aus der Simulation. Bei schnelleren oder engeren Kurven kann das Fahrzeug zwar zunächst stabil bleiben, schafft es aber häufig nicht, den ursprünglichen Arbeitspunkt zum Balancieren wieder zu erreichen. Solange der Kreisel noch ausreichend Winkelreserven hat, bleibt das Fahrzeug kurzzeitig stabil, fällt jedoch früher oder später um. Eine pragmatische Lösung für dieses Problem besteht darin, das Fahrzeug nach einem Manöver abzubremsen bzw. anzuhalten. Bei niedrigen Geschwindigkeiten oder im Stand ist die Entkopplung von Rollen und Gieren aufgrund der fehlenden Kurvendynamik und deutlich höheren Haftreibung des Rads stark genug, um die größeren Abweichungen vom Arbeitspunkt auszugleichen, und das Fahrzeug kann sich zuverlässig und dauerhaft stabilisieren. Dieses Verhalten ist keine zufällige Schwäche, sondern eine strukturelle Einschränkung klassischer Regelungskonzepte.

7.3 Ergebnisse Kurvenfahren

Zusammenfassend lässt sich festhalten, dass das entwickelte Reglerkonzept ein gezieltes und robustes Kurvenfahren ermöglicht. Die Kopplung der Freiheitsgrade Rollen und Gieren lässt sich gezielt nutzen, um durch eine reglerbasierte Trajektorienvorgabe für die Rollbewegung eine gewünschte Gierbewegung zuverlässig umzusetzen. Die Grenzen des Ansatzes ergeben sich aus der zum Balancieren angenommenen Entkopplung von Roll- und Gierdynamik, die nach stark dynamischen Manövern nicht mehr vollständig gegeben ist. Innerhalb dieser Annahmen zeigt das System jedoch ein stabiles und reproduzierbares Fahrverhalten. Damit ist der *Monowheeler* in der Lage, gezielte Kurven zu fahren. Es ist außerdem festzuhalten, dass die Simulation das Verhalten des Systems trotz der getroffenen Vereinfachungen grundsätzlich abbildet. Das liegt daran, dass die Dynamik des Kreisels das Kurvenfahren maßgeblich bestimmt, und vor allem bei langsameren Fahrten die nicht modellierten Anteile weniger stark ins Gewicht fallen.

8 Hardwareintegration und Regelkreis auf Mikrocontroller

Zur Umsetzung der entwickelten Reglerstrukturen auf dem realen Fahrzeug wird der Regelkreis auf einem Mikrocontroller realisiert. Dieser übernimmt die Verarbeitung der Sensordaten, die Umsetzung der Regler, sowie die Ansteuerung der Aktoren. Dafür kommt ein BeagleBone Black zum Einsatz, welches sich aufgrund der umfangreichen Peripherie (SPI, UART, GPIOs etc.) für die Ansteuerung vieler Komponenten (Aktoren und Sensoren) eignet. Außerdem ist es im EML Laborstandard, sodass ein Framework zur Ansteuerung vieler Hardwarekomponenten vorliegt.

Die Software zur Steuerung des *Monowheelers* ist nach dem in Abbildung 8.1 gezeigten Prinzip aufgebaut. Der Entwicklungsrechner ist per WLAN mit einem TP-Link Router verbunden, welcher mit dem BBB verbunden ist. Auf dem Linux-Entwicklungsrechner wird die Software per Secure Shell (SSH)-Verbindung auf dem BBB gestartet. Die Anwendung besteht aus drei Threads, die im Verlauf dieses Kapitels detailliert beschrieben werden. Nach dem Start des Kommunikation-Thread kann auf dem Entwicklungsrechner eine Python-Gui gestartet werden, die die aktuellen Fahrzeugdaten per User Datagram Protocol (UDP)/IP empfängt, in Live-Plots darstellt und in einer CSV-Datei abspeichert.

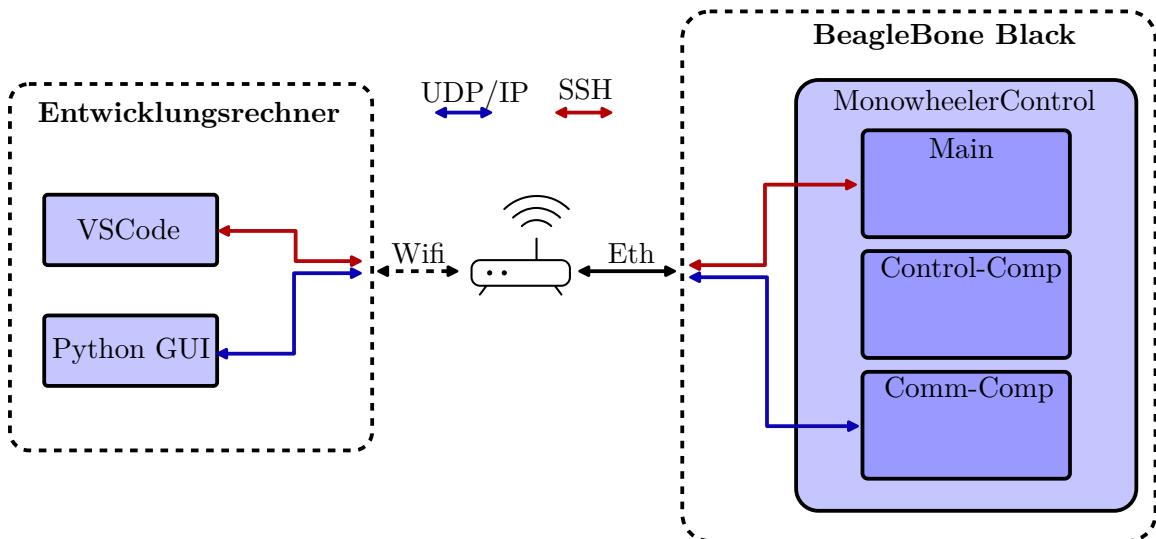


Abbildung 8.1: Übersicht Software

In den folgenden Abschnitten werden die einzelnen Komponenten des Systems dargestellt.

8.1 Entwicklungsumgebung Entwicklungsrechner

Das Projekt ist für ein Linux-Betriebssystem mit VS-Code als Integrated Development Environment (IDE) ausgelegt, kann aber auch aus dem Terminal heraus bedient werden. Der Build-Vorgang wird mit einem Make-File verwaltet. Als Compiler wird ein Cross-Compiler für Linux-Arm verwendet. Dabei muss darauf geachtet werden, dass der Compiler keine Optimierung vornimmt (Compiler Flag -O0), da Memory-Mapping und globale Variablen als Container für die Kommunikation zwischen Threads fälschlicherweise wegoptimiert werden können. Auch können asynchrone Abläufe und Timing durcheinander gebracht werden. Außerdem muss die Anwendung statisch gelinkt werden, da auf dem BBB keine vollständige und aktuelle dynamische Laufzeitumgebung für Bibliotheken zur Verfügung steht und somit sämtliche Abhängigkeiten direkt in das Binary integriert werden müssen.

Um die Anwendung auf dem BBB zu starten, wird die aktuelle Binary per Secure Copy (SCP) übertragen. Um die Verbindung aufzubauen, muss der Entwicklungsrechner im Netzwerk des TP-Link Routers sein, der mit dem BBB verbunden ist. Dort ist das BBB mit Multicast DNS (mDNS) unter dem Hostnamen „BeagleBone.local“ erreichbar. Anschließend kann die Anwendung per SSH im Terminal gestartet und verwaltet werden. Als Debugger wird der Remote GDB Debugger verwendet. Dazu wird der GDB-Server auf dem BBB gestartet, sodass sich der GDB-Debugger des Entwicklungsrechners mit dem BBB verbinden kann.

Alle beschriebenen Aufgaben werden durch das Bash-Skript *manage_monowheeler.sh* automatisiert. Die Funktionen sind im Help-Text des Skripts dokumentiert:

```

1 Usage: ./manage_monowheeler.sh [OPTION]
2
3 Options:
4 -h, --help                                Show this help message
5 -g, --gui                                    Start the GUI on the host machine
6 -bd, --build-debug                           Build the application in Debug mode for BeagleBone
   ↳ Black (BBB)
7 -br, --build-release                        Build the application in Release mode for
   ↳ BeagleBone Black (BBB)
8 -r, --run                                     Build, copy, and run the Release version on BBB
9 -c, --clean                                   Clean the project (remove compiled files)
10 -gdb,--start-gdb                            Start gdbserver on BBB with Debug build
11 -po, --poweroff                             Power off the BeagleBone Black
12 -rb, --reboot                               Reboot the BeagleBone Black
13

```

Listing 8: Automatisierung der Entwicklungsumgebung

Zusätzlich werden die wichtigsten Funktionen (build, clean, run, debug, reboot, poweroff) als VS-Code Tasks angelegt. Somit können diese Aufgaben per Knopfdruck ausgeführt werden. Die Python-Gui wird ebenfalls als VS-Code Task automatisiert.

8.2 Entwicklungsumgebung BeagleBone Black

Auf dem BBB wird ein offizielles Debian-IoT-Image verwendet, welches ursprünglich auf dem Low-Latency-Kernel *5.10.168-ti-r80* basiert. Im Verlauf des Projekts wird auf den Realtime-Kernel *5.10.212-bone-rt-r78* gewechselt, um genaueres Timing zu ermöglichen.

Um ohne Sicherheitsrisiken das Internet nutzen zu können (z.B. zur Installation des neuen Kernels), werden Root-Logins per SSH untersagt. Um trotzdem wichtige Befehle (z.B. die Targetanwendung) mit Root Rechten ausführen zu können, gibt es in der Datei */etc/sudoers.d/noPw* eine Liste mit Befehlen, die ohne Passwortabfrage mit Root Rechten ausgeführt werden können. Zusätzlich ist die Uncomplicated Firewall (ufw) aktiviert und blockiert alle Verbindungen. Die verwendeten Ports (SSH und Debugger) müssen dementsprechend freigegeben werden.

Um die Hardware-Peripherie zu verwalten, verwendet der Kernel einen sog. Device Tree. Der Device Tree ist eine strukturierte Datenbeschreibung um die auf einer Plattform vorhandene Hardware sowie deren Konfiguration zu definieren. Anstatt Hardwareinformationen fest im Kernelcode zu hinterlegen, erlaubt der Device Tree eine flexible und modulare Beschreibung, sodass der Kernel beim Booten die vorhandenen Peripheriemodule wie GPIO, UART, I²C oder SPI korrekt initialisieren und nutzen kann. Auf dem BBB kommen dafür *U-Boot Overlays* zum Einsatz. Diese Device-Tree-Overlays ermöglichen es, Hardwarekonfigurationen dynamisch zu erweitern oder zu ändern, ohne den Kernel neu kompilieren zu müssen. Die Einbindung erfolgt über die Konfigurationsdatei */boot/uEnv.txt*, in der die entsprechenden Overlays geladen werden. Damit können die erforderlichen Schnittstellen aktiviert und konfiguriert werden.

Darüber hinaus besteht die Möglichkeit, eigene Overlays zu erstellen, z.B. um die Pin-Multiplexing-Konfiguration zu verändern. Auf diese Weise lassen sich Pins flexibel unterschiedlichen Funktionen zuordnen, wobei Konflikte mit bereits belegten Schnittstellen (etwa HDMI oder Audio) berücksichtigt werden müssen.

8.3 Targetanwendung

Die Architektur und der Ablauf der Targetanwendung ist in Abbildung 8.2 dargestellt. Diese besteht aus drei Komponenten, die jeweils in einem eigenen Thread laufen. Die Main-Komponente verwaltet die anderen beiden Komponenten und verarbeitet Nutzereingaben. Zusätzlich werden die beiden anderen Komponenten erstellt und gestartet. Die Control-Comp kümmert sich um die Umsetzung der Regelkreise und die Ansteuerung der Hardware, während die Comm-Comp Fahrzeugdaten an die Python-Gui verschickt.

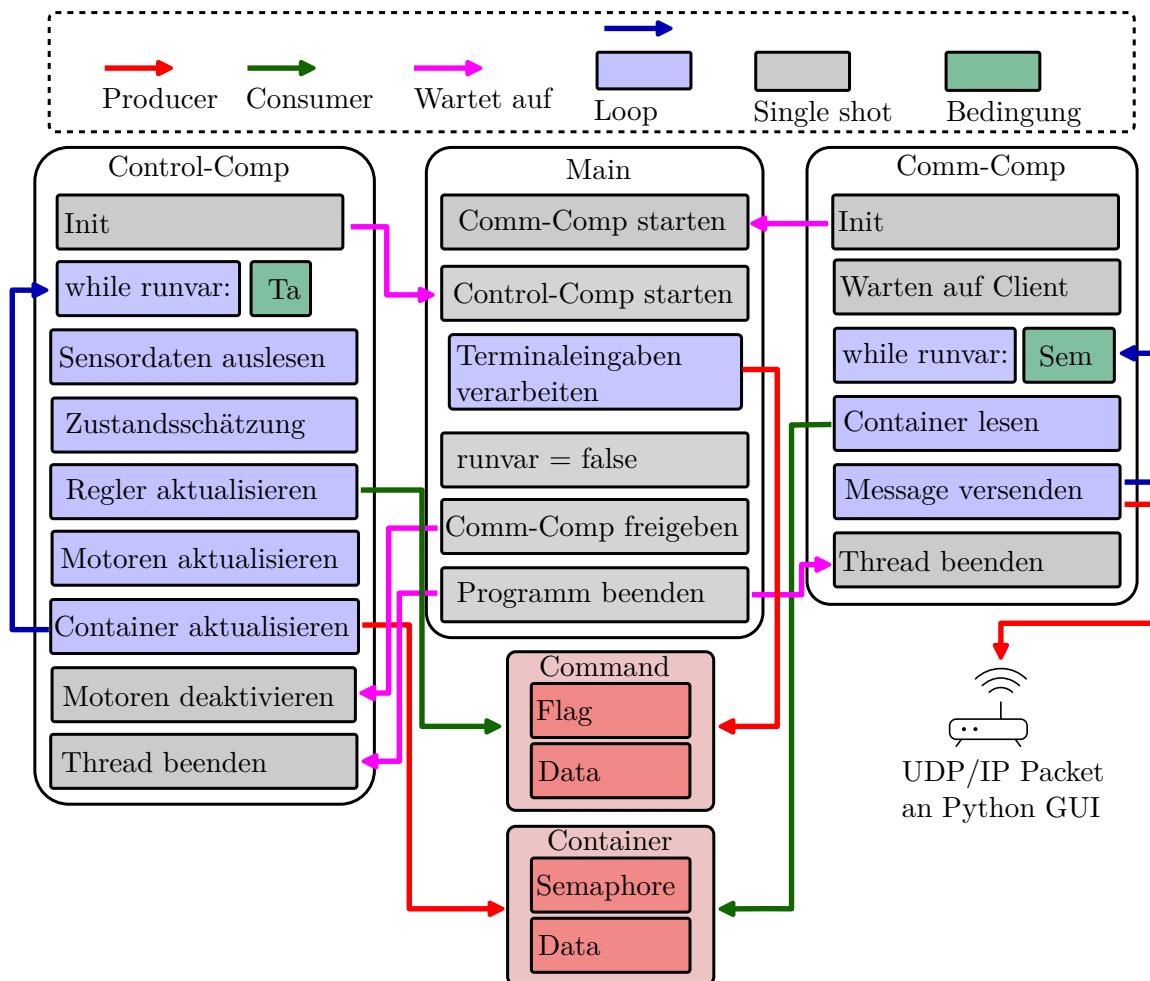


Abbildung 8.2: Architektur der Targetanwendung

Nach dem Start der beiden Komponenten verarbeitet die Main Eingaben im Terminal. Das Fahrzeug kann über eine einfache Befehlszeileneingabe verschiedene Kommandos empfangen, die durch die Klasse *CCommand* interpretiert werden. Dabei wird der eingegebene Text zunächst in einzelne Tokens zerlegt und das erste Token als Befehl ausgewertet. Unterstützt werden folgende Kommandos:

- **drive <value>** – beschleunigt das Fahrzeug auf die Sollgeschwindigkeit <value>
- **left** – leitet eine Linkskurve ein
- **right** – leitet eine Rechtskurve ein
- **slalom** – startet ein Slalommanöver

Der Befehl wird in einem globalen Objekt gespeichert. Die Control-Comp greift dann auf den Befehl zu und leitet dessen Umsetzung ein. Die Synchronisierung zwischen Producer (Main) und Consumer (Control-Comp) erfolgt mit einer atomic Flag, die signalisiert, ob neue Daten im globalen Objekt vorliegen. Der Producer schreibt nur dann neue Kommandos hinein, wenn der Container leer ist. Andernfalls wird die Eingabe abgewiesen und eine entsprechende Fehlermeldung ausgegeben. Der Consumer prüft in jedem Zyklus, ob neue Daten

verfügbar sind, verarbeitet diese und setzt das Flag anschließend zurück. Ein komplexes Handshake-Protokoll ist in diesem Szenario nicht erforderlich, da Befehle vom Nutzer nur in vergleichsweise großen zeitlichen Abständen eingegeben werden und Mehrfacheingaben durch die Fehlermeldung behandelt werden. Auf diese Weise wird sichergestellt, dass Befehle weder mehrfach noch gleichzeitig verarbeitet werden, ohne dass die Echtzeitfähigkeit des Consumers beeinträchtigt wird. Der Befehl zur sofortigen Beendigung des Programms wird separat behandelt und ist unmittelbar wirksam.

Die Übertragung der Fahrzeugdaten von der Control-Comp, die als Producer agiert, zur Comm-Comp, die die Rolle des Consumers übernimmt, erfolgt über einen globalen Container. Um die Datenübertragung thread-sicher zu gestalten und Race-Conditions zu vermeiden, enthält der Container neben den Daten einen Mechanismus zur Synchronisierung. Da die Daten nur einmal gelesen werden dürfen, werden Semaphoren eingesetzt. Normalerweise ist in so einem Kontext ein Cross-Handshake notwendig. Dabei signalisiert die Control-Comp das Vorhandensein neuer Daten per Semaphore, und die Comm-Comp meldet, wenn die Verarbeitung fertig ist. So wird garantiert, dass Daten weder mehrfach, gleichzeitig noch teilweise verarbeitet werden. Allerdings ist der Consumer in diesem Fall garantiert schneller als der Producer. Außerdem darf die Control-Comp unter keinen Umständen blockiert werden, da dies die Stabilität des Fahrzeugs gefährdet. Aus diesen Gründen ist eine einfache Semaphore, die der Comm-Comp signalisiert, wenn neue Daten vorhanden sind, ausreichend.

Wird die Eingabe zum Stoppen des Programms getätigkt, wird den anderen Komponenten mit der Variable *runvar* signalisiert, sich zu beenden. Die Control-Komponente beendet sich nach dem nächsten Abtastschritt. Um sicherzustellen, dass die Kommunikations-Komponente nicht auf der Semaphore blockiert ist, gibt die Main die Semaphore erneut frei. Haben sich sowohl die Control-Comp als auch die Comm-Comp beendet, wird das Programm beendet.

8.3.1 Softwarearchitektur

Die Regelung des *Monowheelers* und der Datenaustausch mit der Python Gui sind auf zwei Komponenten aufgeteilt. Eine vereinfachte Struktur der Klassen ist in Abbildung 8.3 abgebildet.

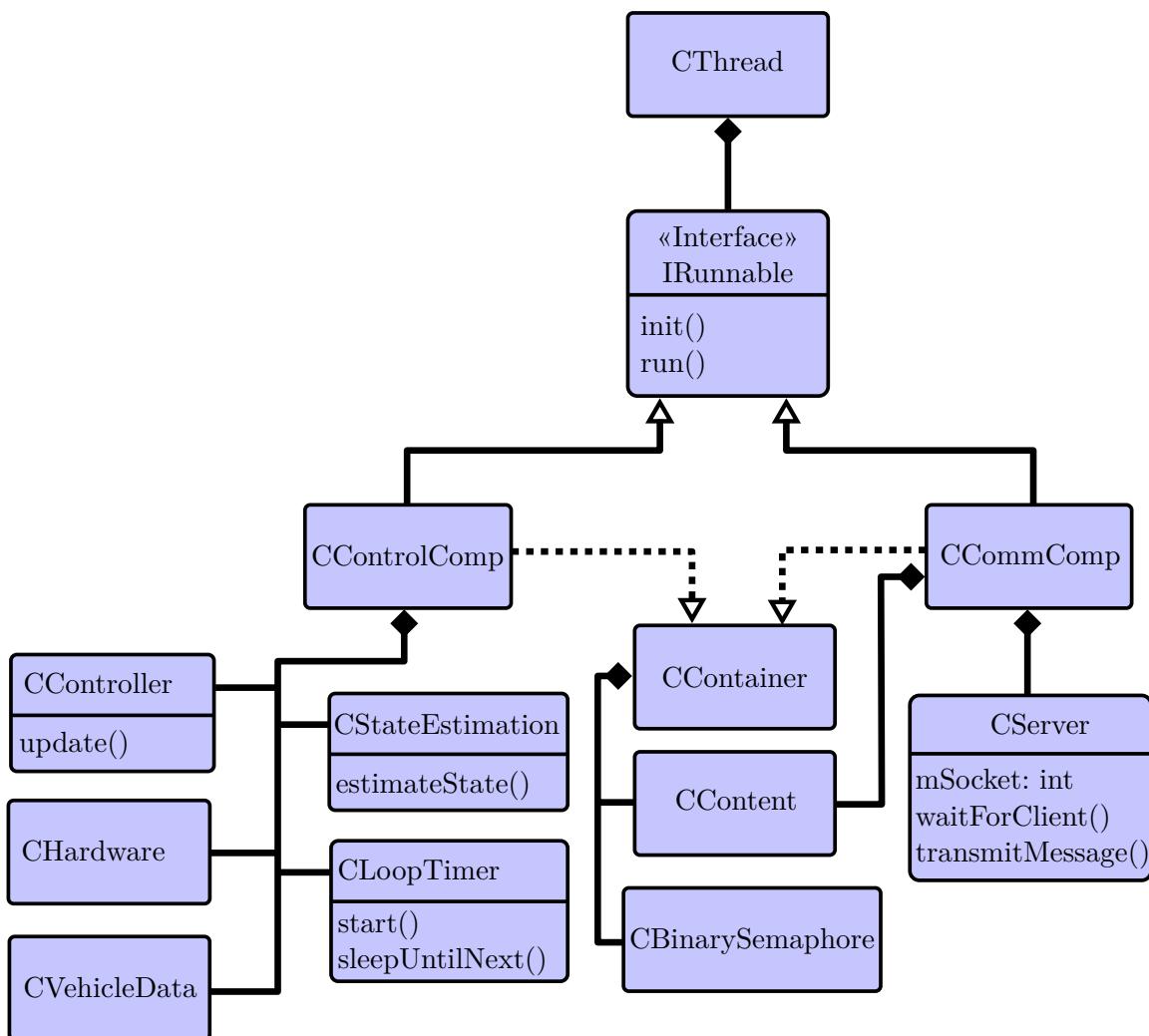


Abbildung 8.3: Klassendiagramm der Targetanwendung (vereinfacht)

Beide Komponenten erben von der virtuellen Klasse `IRunnable` und bieten so ein einheitliches Interface mit einer `init()` und einer `run()`-Funktion. Die Verwaltung des Threads der jeweiligen Komponente wird von der Klasse `CThread` übernommen, sodass beispielsweise verschiedene Prioritäten oder die Scheduling-Methode festgelegt werden können. Die beiden Klassen `IRunnable` und `CThread` zur Verwendung und Verwaltung von Posix-Threads auf dem BBB sind Teil des Frameworks aus dem EML und nach den in [41] und [40] beschriebenen Prinzipien aufgebaut.

Die Control-Comp kümmert sich um das Timing, den Regelkreis sowie die Ansteuerung der Hardware, während die Comm-Comp die Fahrzeugdaten per UDP versendet. Diese Aufteilung wird gewählt, um die Echtzeitanforderungen des Systems möglichst genau einhalten zu können. Regelkreise stellen strenge Echtzeitanforderungen an den Mikrocontroller, da die Datenerfassung, Berechnung und Ansteuerung der Aktoren in definierten Zeitintervallen erfolgen müssen. Kommt es zu Verzögerungen oder variablen Latenzen (Jitter), verändert sich das effektive Abtastintervall, was die Stabilität des Reglers gefährdet, Schwingungen verstärken oder zum Kontrollverlust des Fahrzeugs führen kann. Linux ist aber nicht echt-

zeitfähig, weil das Scheduling und die Verarbeitung von Interrupts keinem deterministischen Timing folgt. Daher bringt jeder Kontextwechsel zusätzliche, nicht deterministische Latenz mit sich. Darum sollten Kontextwechsel so gut es geht vermieden werden. Aus diesem Grund werden alle Aufgaben, die Teil des Regelkreises sind, in einem Thread (in der Control-Comp) hintereinander ausgeführt. Somit gibt es einen Kontextwechsel nur, wenn die Control-Comp bis zum nächsten Abtastintervall in den Sleep-Zustand übergeht.

Die Echtzeitfähigkeit des Linux-Systems kann durch die Wahl des Kernels verbessert werden. Standardmäßig kommt das Debian-IOT-Image für das BBB mit einem Low-Latency-kernel, der bereits für geringe Latenz optimiert ist. Noch besser sind Real-Time-Kernel, wie der *5.10.212-bone-rt-r78*, der für dieses Projekt verwendet wird. Die Unterschiede hinsichtlich des Timings der beiden Kernel-Varianten wird in Unterabschnitt 8.3.2 genauer untersucht.

In der gesamten Software gibt es zahlreiche Parameter, von den Reglerparametern über die Parametrierung der Hardware bis hin zu Filterkoeffizienten, die angepasst werden können. Um eine einheitliche, übersichtliche Schnittstelle zu schaffen, wird in der Datei *MonowheelerConstants.hpp* ein Namespace *Monowheeler* angelegt. Alle Parameter, die nicht fest vorgegeben sind (z.B. Speicheradressen bestimmter Register), werden in diesem Namespace gespeichert und können an den entsprechenden Stellen eingebunden werden.

Alle Klassen und Funktion im Detail vorzustellen, würde den Rahmen dieser Arbeit sprengen. Daher werden in den folgenden Abschnitten die wichtigsten Module in vereinfachter Form exemplarisch beschrieben. Eine vollständige und detaillierte Darstellung findet sich direkt im Quellcode sowie in den begleitenden *README*-Dateien des Repositories.

8.3.2 Control-Comp

Die Control-Comp ist für das Timing, das Auslesen der Sensoren, die Signalverarbeitung, die Regelung und die Ansteuerung der Aktoren zuständig. Alle diese Aufgaben werden in dedizierte Klassen ausgelagert (siehe Abbildung 8.3) und in der Control-Comp zusammengeführt. In den folgenden Abschnitten wird die Implementierung der einzelnen Module vorgestellt.

Timer

Die Timer-Klasse soll drei Funktionen bieten: zum Starten des Timers, zum Auslesen der aktuellen Zeit und zum Warten bis zum nächsten Abtastschritt. Auf klassischen Mikrocontrollern ohne Betriebssystem können Hardware-Timer zur Zeitmessung verwendet werden. Dadurch ist harte Echtzeit möglich. Auf einem Linux-System wie dem BBB werden die Hardware-Interrupts allerdings vom Linux-Kernel verwaltet. Daher können Posix-Funktionen im Userspace verwendet werden, um auf die Hardware-Timer zuzugreifen.

Die Funktion *clock_gettime()* gibt die aktuelle Zeit auf Mikrosekunden genau zurück. Der Zeitgeber kann dabei selbst gewählt werden. Für diese Anwendung eignet sich die Uhr *CLOCK_MONOTONIC*, da es sich um einen monoton steigenden Timer handelt, der die Zeit seit dem Systemstart ohne Beeinflussung durch Zeitsprünge zählt. Damit kann die aktuelle Systemzeit beim Start des Timers erfasst und gespeichert werden. Die Funktion

zum Auslesen der aktuellen Zeit muss dann nur die Zeit beim Start des Timers von der aktuellen Systemzeit abziehen.

Für die Funktion zum Warten im Sleep-Zustand bis zum nächsten Abtastschritt wird die Funktion `clock_nanosleep()` verwendet. Diese Funktion kann entweder eine relative Zeit schlafen, oder bis zu einem bestimmten, absoluten Zeitpunkt. Für die relative Zeit wird die Zeit zu Beginn und am Ende des Schleifendurchlaufs gespeichert. Aus der Differenz ergibt sich die Laufzeit, sodass die verbleibende Zeit bis zum nächsten Abtastschritt berechnet und geschlafen werden kann. Bei der absoluten Zeit wird die Zeit beim Start des Timers gespeichert. Der absolute Zeitpunkt des ersten Abtastschritts kann berechnet werden, in dem die Abtastzeit auf die Startzeit addiert wird. Die Funktion `clock_nanosleep()` schläft dann bis zum Erreichen dieses Zeitpunkts. Alle weiteren Zeitpunkte zum Aufwachen können berechnet werden, in dem die Abtastzeit auf den vorherigen Zeitstempel addiert wird. Beide dieser Optionen sollen getestet werden.

Für den Jitter-Test wird eine Abtastzeit von 2 ms gewählt, was einer Frequenz von 500 Hz entspricht. Diese Frequenz liegt über der im späteren Anwendungsszenario voraussichtlich benötigten Abtastfrequenz. Dadurch wird das System einer strengeren Anforderung ausgesetzt, und es ist davon auszugehen, dass das System bei geringerer Auslastung vergleichbare oder bessere Ergebnisse erzielt. Die Tests dienen somit als Worst-Case-Betrachtung. Über einen Zeitraum von ca. 10 s werden die absoluten Zeitstempel jedes Abtastschritts aufgezeichnet und anschließend mit dem Python-Skript `jitter.py` analysiert. Zuerst wird die relative Zeitmessung mit dem Low-Latency-Kernel getestet (siehe Abbildung 8.4):

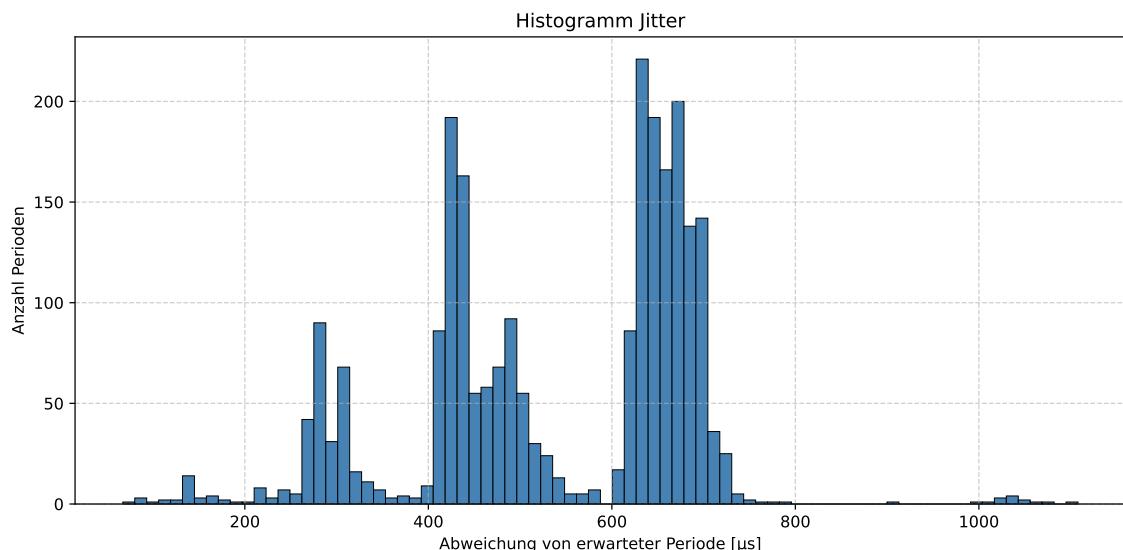


Abbildung 8.4: Histogramm Jitter Test Low-Latency-Kernel mit relativer Zeitmessung

Der mittlere Jitter beträgt 540 μs, und der maximale Jitter 1108 μs. Das entspricht einem durchschnittlichen Fehler von 25 % und einem maximalen Fehler von über 50 %. Dieser Jitter ist für einen Regelkreis nicht akzeptabel. Daher wird auf den bereits beschriebenen Realtime-Kernel umgestellt und derselbe Test mit der gleichen Software durchgeführt (siehe Abbildung 8.5):

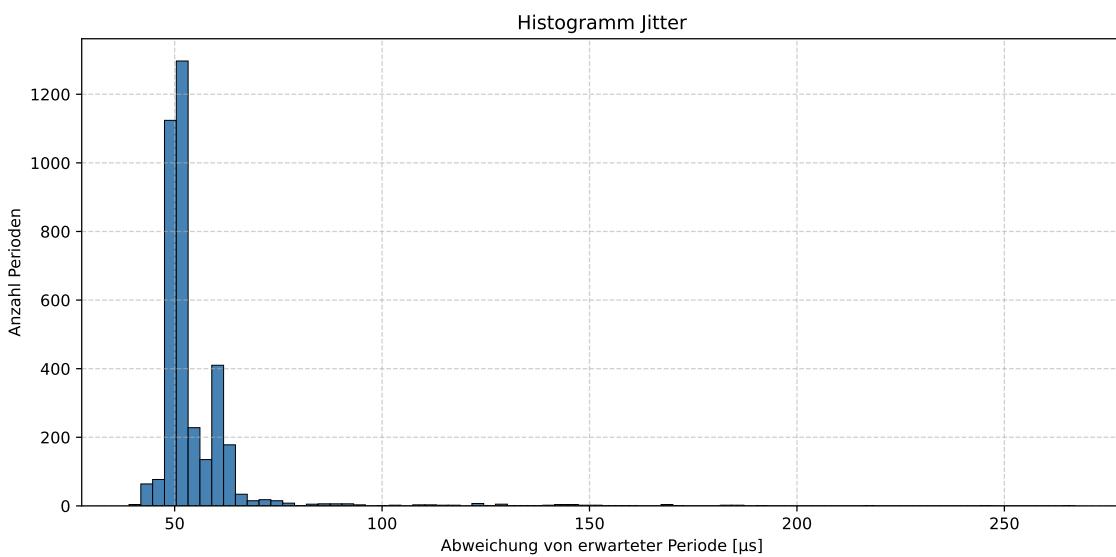


Abbildung 8.5: Histogramm Jitter Test Realtime-Kernel mit relativer Zeitmessung

Die Ergebnisse haben sich deutlich verbessert und der Jitter hat stark abgenommen. Der durchschnittliche Jitter ist mit 55 μs um den Faktor 10 kleiner geworden, der maximale Jitter beträgt nun 267 μs. In dem Histogramm ist aber zu erkennen, dass die Häufigkeit der Ausreißer stark nachgelassen hat. Es ist allerdings auch zu erkennen, dass der Jitter nicht nur aus einem zufälligen Fehler besteht, sondern auch ein Offset enthält, da alle Werte um 50 μs liegen. Das ist damit zu erklären, dass bei der Messung des Schleifendurchlaufs die Zeit nicht berücksichtigt wird, die beim Scheduling entsteht, wenn der Thread in den Schlafen-Zustand versetzt und wieder aufgeweckt wird. Daher wird die absolute Zeitmessung getestet (siehe Abbildung 8.6). Dabei wird die Zeit einmal pro Schleifendurchlauf erfasst, und somit jede Verzögerung beim Scheduling berücksichtigt.

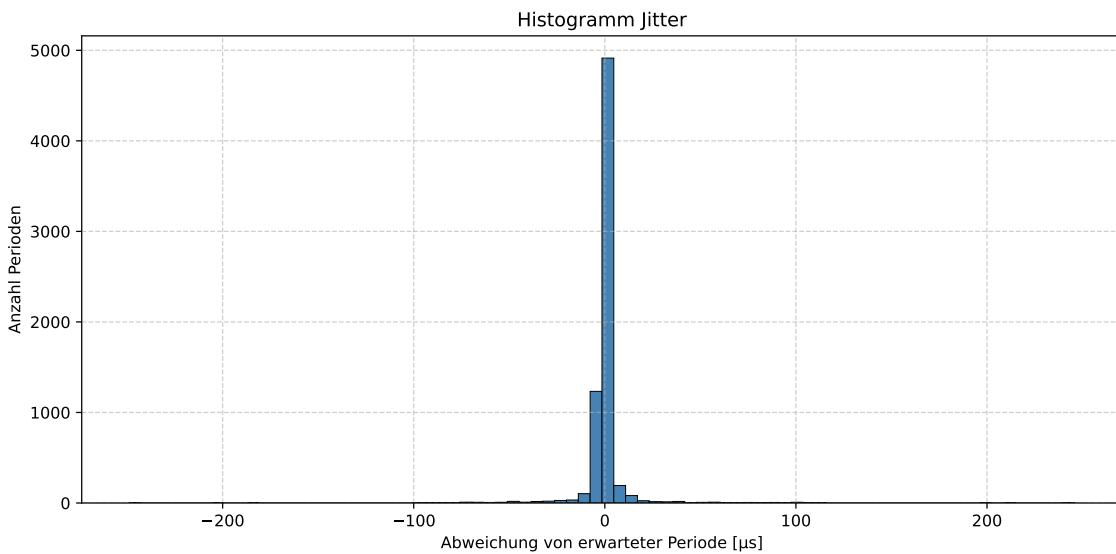


Abbildung 8.6: Histogramm Jitter Test Realtime-Kernel mit absoluter Zeitmessung

Der Jitter entspricht jetzt wie erwartet annähernd einer Normalverteilung um null herum. Der mittlere Jitter konnte erneut um den Faktor 10 verringert werden und beträgt nun 4 µs. Die Ausreißer mit einem Maximalwert von 249 µs treten so selten auf, dass sie vernachlässigbar sind. Ein Jitter im einstelligen Mikrosekundenbereich ist für einen Regelkreis eines derart trügen Systems völlig unproblematisch.

Die Implementierung der Funktion zum Starten des Timers und zum Warten im Sleep-Zustand bis zum nächsten Abtastschritt ist in Listing 9 zu sehen:

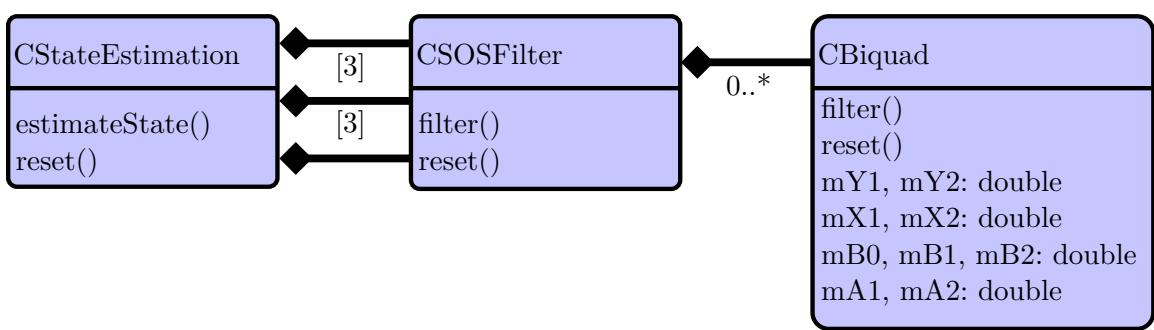
```

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

Signalverarbeitung

Die Aufgabe des Moduls für die Signalverarbeitung ist es, die in Kapitel 5 erarbeiteten Filter auf dem Mikrocontroller umzusetzen. Diese Aufgabe wird von der Klasse *CStateEstimation* übernommen. Eine Übersicht über die Struktur der Klasse ist in Abbildung 8.7 zu finden.

Abbildung 8.7: Klassendiagramm von *CStateEstimation* (vereinfacht)

Bei jedem Abtastschritt wird die Funktion *estimateState()* aufgerufen und die aktuellen Sensordaten übergeben. Die Sensordaten werden zunächst gefiltert. Dafür verfügt die Klasse *CStateEstimation* über zwei Arrays mit jeweils drei Objekten und ein weiteres Einzelobjekt. Die beiden Arrays beinhalten jeweils die Tiefpassfilter für die Accelerometer und Gyroskope für alle Achsen. Das zusätzliche Objekt repräsentiert einen Tiefpassfilter für die Geschwindigkeit des Fahrzeugs. Anschließend werden die Sensordaten gemäß Kapitel 5 fusioniert, um eine möglichst akkurate Zustandsschätzung zu ermöglichen. Die Implementierung der Funktion ist in Listing 10 zu sehen:

```

1 void CStateEstimation::estimateState(const CImuData& pImuData, CStateData
2   &pStateData)
3 {
4     pStateData.mDotTheta = mDLPGyro[1].filter(pImuData.mGyroY);
5     pStateData.mDotPhi = mDLPGyro[0].filter(pImuData.mGyroX);
6     pStateData.mDotPsi = mDLPGyro[2].filter(pImuData.mGyroZ);
7     pStateData.mSpeed = mDLPSpeed.filter(pStateData.mSpeed);
8
9     double xAccelFil = mDLPAccel[0].filter(pImuData.mAccelX);
10    double yAccelFil = mDLPAccel[1].filter(pImuData.mAccelY);
11
12    pStateData.mTheta = mAlpha*(pStateData.mTheta + pStateData.mDotTheta*mTa) +
13      (1-mAlpha)*(std::asin(xAccelFil / g));
14    pStateData.mPhi = mAlpha*(pStateData.mPhi + pStateData.mDotPhi*mTa) +
15      (1-mAlpha)*(-std::asin(yAccelFil / g));
16 }
  
```

Listing 10: Ausschnitt Implementierung Zustandsschätzung

Die Filter werden in der Klasse *CSOSFilter* umgesetzt. Ein Second-Order-Section (SOS)-Filter besteht aus einer Abfolge von Filtern zweiter Ordnung, um numerische Stabilität bei Infinite Impulse Response (IIR)-Filtern höheren Grades zu gewährleisten. Die einzelnen Filterbausteine zweiter Ordnung werden in der Klasse *CBiquad* implementiert. Dazu wird ein diskreter IIR-Filter zweiter Ordnung wie in Listing 11 zu sehen umgesetzt, dessen Parameter im Konstruktor festgelegt werden.

```

1 double CBiquad::filter(double pX)
2 {
3     double y = mB0*pX + mB1*mX1 + mB2*mX2 - (mA1*mY1 + mA2*mY2);
4     mY2 = mY1;
5     mY1 = y;
6     mX2 = mX1;
7     mX1 = pX;
8     return y;
9 }
```

Listing 11: Ausschnitt Implementierung IIR-Filter zweiter Ordnung

Eine beliebige Anzahl solcher Filterbausteine wird in der Klasse *CSOSFilter* in einem dynamischen Array gespeichert, sodass jede Art von Filter (Notch, Tiefpass, Hochpass usw.) in beliebiger Ordnung umgesetzt werden kann. Dazu müssen nur die Filterkoeffizienten entsprechend gewählt werden. Die Filterbausteine werden dann wie in Listing 12 gezeigt nacheinander aufgerufen und ergeben so einen beliebigen IIR-Filter.

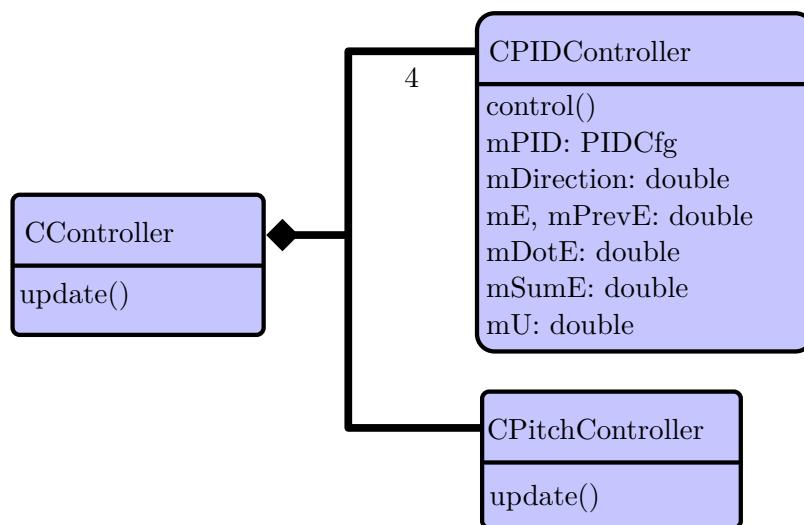
```

1 CSOSFilter::CSOSFilter(std::vector<std::array<double, 6>> pCoeffs)
2 {
3     for (const auto coeff : pCoeffs) {
4         mBiquads.emplace_back(coeff);
5     }
6 }
7 double CSOSFilter::filter(double pX)
8 {
9     double y = pX;
10    for (CBiquad& biquad : mBiquads) {
11        y = biquad.filter(y);
12    }
13    return y;
14 }
```

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

Regelung

Das Konzept für die Regelung des *Monowheelers* beinhaltet die direkte Regelung der Nickbewegung, der Rollbewegung und der Geschwindigkeit des Fahrzeugs und soll in der Klasse *CController* auf dem Mikrocontroller umgesetzt werden. Zusätzlich kann durch die übergeordnete Vorgabe von Trajektorien die Bewegung des gesamten Fahrzeugs vorgegeben werden, inklusive der Steuerung der Gierbewegung. Das Fahrzeug kann still stehen, in einer Linie fahren und Kurven fahren. Die Koordinierung der Regler erfolgt zentral in der Klasse *CController*, während die einzelnen Regler ausgelagert werden. Die Klasse *CController* bietet eine *update()*-Funktion, die die gesamte Regelschleife enthält. Die Klassenstruktur der Implementierung ist in Abbildung 8.8 zu sehen, die vollständige Implementierung der Funktion *update()* ist in Abschnitt C.3 zu finden.

Abbildung 8.8: Klassendiagramm von *CController* (vereinfacht)

Das Konzept zur Regelung der Geschwindigkeit, des Rollens und des Gierens basiert auf PID-Reglern. Die PID-Regler werden in der Klasse *CPIDController* realisiert. Die Klasse enthält interne Variablen zum Speichern der Parameter und verschiedener Regelabweichungen und stellt eine überladene Funktion *control()* bereit. Diese Funktion aktualisiert die internen Größen und gibt die Stellgröße zurück. Die Funktion nimmt als Argumente entweder klassisch den Sollwert und Istwert, oder zusätzlich dazu einen Soll- und Istwert für die Ableitung der zu regelnden Größe. Im Fall des *Monowheelers* liegt die Winkelgeschwindigkeit direkt als Sensorwert vor. Es macht daher Sinn, diesen Wert direkt im PID-Regler zu verwenden, anstatt erst den Winkel zu bestimmen, dann die Regelabweichung zu berechnen und abzuleiten (für mehr Informationen siehe Gleichung 6.39 ff). Die Implementierung der Funktion *control()* mit zusätzlichem Soll- und Istwert der Ableitung ist in Listing 13 zu sehen. Zusätzlich wird Anti-Windup für den Integrator eingeführt und die Stellgröße begrenzt.

```

1 double CPIDController::control(const double pW, const double pDotW, const double
2   ↪  pX, const double pDotX)
3 {
4   mE = pW - pX;
5   mDotE = pDotW - pDotX;
6   if (std::abs(mU) < mPID.UMax) {
7     mSumE += mE*mPID.Ta;
8   }
9   mU = std::clamp(mDirection*(mPID.KP*mE + mPID.KD*mDotE + mPID.KI*mSumE),
10    ↪ -mPID.UMax, mPID.UMax);
11  mEPrev = mE;
12  return mU;
13 }
  
```

Listing 13: Ausschnitt Implementierung PID-Regler

Der gesamte Ausschnitt für die Implementierung des PID-Reglers mit beiden Versionen der *control()*-Funktion ist in Abschnitt C.2 zu sehen.

Die *update()*-Funktion der Klasse *CController* lässt sich in zwei Abschnitte gliedern. Auf der einen Seite gibt es die Regler, die als Ausgang direkt die Motorstellgrößen haben und Vorgaben für das physikalische System umsetzen. Diese Regler werden am Ende jedes Durchlaufs aufgerufen. Dazu zählt der Zustandsregler für die Nickbewegung, der PID-Regler für die Rollbewegung und der PID-Regler für die Geschwindigkeit des Fahrzeugs. Zusätzlich gibt es die Vorgabe von Trajektorien und Sollwerten, die für die übergeordnete Bewegungsplanung verantwortlich sind. Dazu gehören die Trajektorien der Geschwindigkeit und der Rollbewegung. Abhängig vom aktuellen Modus werden unterschiedliche Konzepte für die Sollwertgenerierung verwendet. Zunächst werden die Regler zur Regelung des physikalischen Systems vorgestellt.

Der in Abschnitt 6.2 ausgelegte Regler für die Nickbewegung soll in der Klasse *CPitchController* implementiert werden. Die Klasse soll eine Funktion zum Update der internen Größen (Reglerausgang etc.) bieten, die die aktuelle Stellgröße zurückgibt. Ein normaler Zustandsregler kann durch eine einfache Multiplikation der Reglermatrix mit den Zuständen implementiert werden (siehe Listing 14):

```

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

```

Listing 14: Ausschnitt Implementierung des Zustandsreglers Nicken

Allerdings handelt es sich um einen Regler mit zustandsbasiertem Gain-Scheduling. Daher wird zusätzlich eine vereinfachte Zustandsmaschine implementiert. Dabei gibt es die folgenden vier Modi:

- Mode Fast: Das schnelle Parameterset wird eingesetzt, um die Stellgröße zu berechnen. Wird verwendet, wenn gewisse Abweichungen in Winkel oder Winkelgeschwindigkeit vorliegen.
- Mode Slow: Das langsame Parameterset wird eingesetzt, um die Stellgröße zu berechnen. Wird verwendet, wenn das System lange genug stabil ist.
- Mode TransitionSlow2Fast: Übergang zwischen Modus Slow und Modus Fast. Die Ausgänge der beiden Parametersets werden linear interpoliert, sodass Unstetigkeiten in der Stellgröße vermieden werden. Ist für eine vorgegebene Zeitdauer nach dem Wechsel aktiv.
- Mode TransitionFast2Slow: Übergang zwischen Modus Fast und Modus Slow, identische Funktion zu Mode TransitionSlow2Fast.

Zu Beginn der *update()*-Funktion werden je nach Modus verschiedene Bedingungen geprüft und gegebenenfalls interne Zustände angepasst. Ein Beispiel ist in Listing 15 zu sehen.

```

1 case Mode::Fast:
2 if (std::abs(pTheta) < mThetaLowerThreshold && std::abs(pDotTheta) <
3     ↵ mDotThetaLowerThreshold) {
4     mTSlow += mTa;
5     if (mTSlow > mTSlowThreshold) {
6         mMode = Mode::TransitionFast2Slow;
7         mTTransition = 0;
8         mTSlow = 0;
9     }
10 break;

```

Listing 15: Ausschnitt Implementierung der Zustandsübergänge Nicken

Anschließend kann anhand vom Modus die Stellgröße berechnet werden. Im Modus Fast und Slow wird die Stellgröße wie in Listing 14 bestimmt, in den Übergangsmodi erfolgt die Berechnung wie in Listing 16 beispielhaft gezeigt:

```

1 case Mode::TransitionSlow2Fast:
2     uCMDFast = -std::clamp((mLQRFast[0]*pTheta
3                             + mLQRFast[1]*pDotTheta
4                             + mLQRFast[2]*mThetaSum
5                             + mLQRFast[3]*pDynamixel), -mUCMDMax, mUCMDMax);
6     uCMDSlow = -std::clamp((mLQRSlow[0]*pTheta
7                             + mLQRSlow[1]*pDotTheta
8                             + mLQRSlow[2]*mThetaSum
9                             + mLQRSlow[3]*pDynamixel), -mUCMDMax, mUCMDMax);
10    blend = std::clamp(mTTransition/mTTransitionMax, 0.0, 1.0);
11    mUCMD = blend*uCMDFast + (1-blend)*uCMDSlow;
12    mTTransition += mTa;
13    break;

```

Listing 16: Ausschnitt Implementierung der Interpolation Nicken

Die vollständige Funktion *update()* ist in Abschnitt C.1 zu finden.

Die Regelung der Geschwindigkeit und der Rollbewegung erfolgt mit der bereits vorgestellten Klasse *CPIDController*. Der Regler für die Geschwindigkeit wird im Stillstand angehalten, um ein langsames Aufbauen des Integrators durch kleine Offsets der Messdaten zu verhindern. Der Aufruf der Regler ist in Listing 17 zu sehen.

```

1 pMotorData.mWheelDynamixelTarget = mPitchController.control(pStateData.mTheta,
    ↳ pStateData.mDotTheta, pMotorData.mWheelDynamixelPosition);
2
3 pMotorData.mGyroDynamixelTarget = mRollController.control(pStateData.mPhiTarget,
    ↳ dotPhiTarget, pStateData.mPhi, pStateData.mDotPhi);
4 if (pStateData.mSpeedTarget == 0 && abs(pStateData.mSpeed) < 0.05) {
5     pMotorData.mWheelTorqueTarget = 0;
6 }
7 else {
8     pMotorData.mWheelTorqueTarget =
    ↳ mSpeedController.control(pStateData.mSpeedTarget, pStateData.mSpeed);
9 }
```

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

Die Vorgabe der Trajektorie für die Geschwindigkeit erfolgt im Gegensatz zu der Trajektorie für die Rollbewegung nicht reglerbasiert. Um große Sprünge im Motormoment des Antriebsrads zu verhindern und sanftes Anfahren zu garantieren, wird die Sollgeschwindigkeit mit einer Rampe vorgegeben. Dabei wird anhand einer maximalen Änderungsrate der Sollwert langsam an den Zielwert herangeführt. Die Berechnung erfolgt in der generischen Funktion *ramp()* (siehe Listing 18). Beim Balancieren kann die Geschwindigkeit vom Nutzer eingegeben werden, beim Kurvenfahren wird eine feste Geschwindigkeit vorgeschrieben.

```

1 double CController::ramp(double pTarget, double pCurrentTarget, double pMaxRate)
2 {
3     double deltaMax = pMaxRate * mTa;
4     double diff = pTarget - pCurrentTarget;
5
6     if (diff > deltaMax)
7     {
8         pCurrentTarget += deltaMax;
9     }
10    else if (diff < -deltaMax)
11    {
12        pCurrentTarget -= deltaMax;
13    }
14    else
15    {
16        pCurrentTarget = pTarget;
17    }
18
19    return pCurrentTarget;
20 }
```

Listing 18: Ausschnitt Implementierung der Rampe für sanftes Anfahren

Das Konzept für die Regelung des Rollens zum Balancieren und Kurvenfahren beinhaltet zwei PID-Regler (siehe Abschnitt 6.3 und Kapitel 7). Der PID-Regler zum Balancieren gibt

abhängig von der Position des Kreisels einen Sollwert für den Rollwinkel vor. Zusammen mit dem alten Sollwert kann die gewünschte Rollgeschwindigkeit bestimmt werden. Während einer Kurvenfahrt wird der Sollwert für die Rollgeschwindigkeit durch den PID-Regler zur Steuerung der Giergeschwindigkeit vorgegeben. Zusammen mit dem vorherigen Wert für den Rollwinkel kann durch die Integration der Rollgeschwindigkeit der neue Sollwert für den Rollwinkel berechnet werden. Die Umschaltung zwischen den verschiedenen Modi erfolgt anhand innerer Zustände (das Erreichen bestimmter Sollwerte, Ablauf einer Zeit etc.) und durch die Nutzereingaben. Die Implementierung der Trajektoriengenerierung ist in Listing 19 gezeigt.

```

1 switch (mMode)
2 {
3     case EMode::Balance:
4         phiTargetNew = mRollTargetBalanceController.control(0, 0,
5             ↪ pMotorData.mGyroDynamixelPosition, pMotorData.mGyroDynamixelVelocity);
6         dotPhiTarget = (phiTargetNew - pStateData.mPhiTarget)/mTa;
7
8         pStateData.mPhiTarget = phiTargetNew;
9         pStateData.mSpeedTarget = ramp(mTargetVelocity, pStateData.mSpeedTarget,
10             ↪ Monowheeler::MAX_ACCEL);
11        pStateData.mDotPsiTarget = mDotPsiTarget;
12        pStateData.mThetaTarget = 0;
13        ...
14    case EMode::Corner:
15        pStateData.mDotPsiTarget = mDotPsiTarget;
16        dotPhiTarget = mRollTargeYawController.control(pStateData.mDotPsiTarget,
17             ↪ pStateData.mDotPsi);
18        pStateData.mPhiTarget += dotPhiTarget*mTa;
19        pStateData.mSpeedTarget = ramp(mTargetVelocity, pStateData.mSpeedTarget,
20             ↪ Monowheeler::MAX_ACCEL);
21        pStateData.mThetaTarget = 0;
22        ...
23    default:
24        break;
25 }

```

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

Hardware

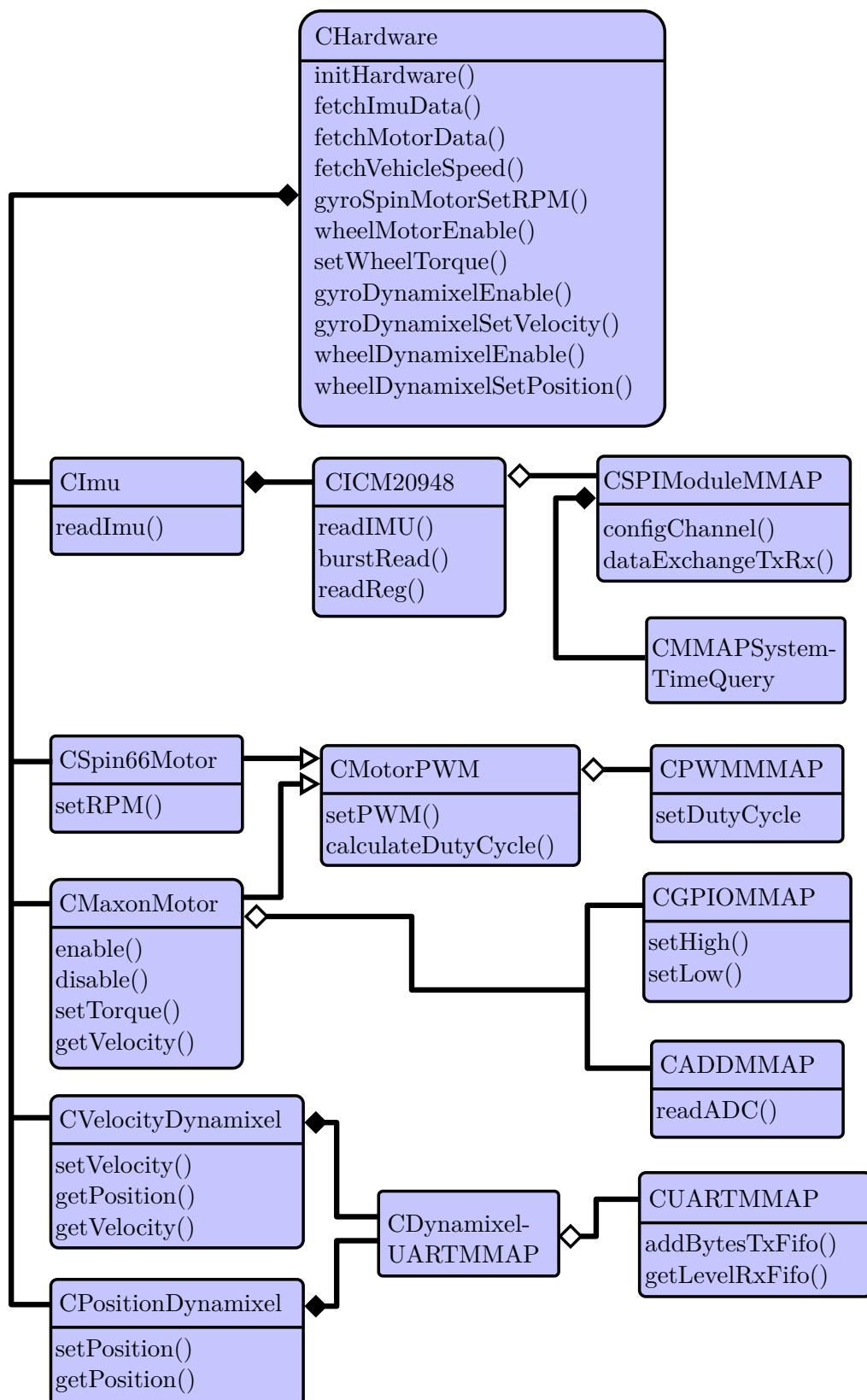
Die Aufgabe des Hardware-Moduls ist es, ein einheitliches Interface zur Ansteuerung aller Sensoren und Aktoren zu bieten. Dies wird auf verschiedenen Abstraktionsebenen implementiert:

- Abstraktionsebene 1: Die Klasse *CHardware* stellt die höchste Abstraktionsebene dar. Sie dient als zentrale Schnittstelle zur Verwaltung der Hardware und fungiert als Adapter, indem sie projektspezifische Umrechnungen, Einschränkungen und Kalibrierungen durchführt und die gewünschten physikalischen Werte in die Form übersetzt, die

von den generischen Motor- und Sensorklassen erwartet wird. Der Rest der Software interagiert ausschließlich mit dieser Ebene.

- Abstraktionsebene 2: Diese Ebene besteht aus Klassen wie *CImu* oder *CSpin66Motor*, die einheitliche Schnittstellen für einzelne Gerätetypen bereitstellen. Die Geräte werden dort projektspezifisch konfiguriert und die Funktionen zur Verfügung gestellt, die im konkreten Projekt benötigt werden.
- Abstraktionsebene 3: Auf dieser Ebene befinden sich die generischen Treiberklassen wie *CICM20948* oder *CDynamixelUARTMMAP*. Sie implementieren die grundlegende Logik für bestimmte Gerätetypen wie spezielle Sensoren oder Motorcontroller. Diese Klassen sind projektunabhängig wiederverwendbar und arbeiten mit gerätespezifischen Einheiten und Rohdaten.
- Abstraktionsebene 4: Die Klassen bieten einen direkten Zugriff auf die Peripherie des BBB per Memory Map (MMAP). Sie setzen die physikalische Ebene der Kommunikation um. Der Vorteil bei der Verwendung von MMAP-Zugriffen ist die Geschwindigkeit der Zugriffe, da die Linux-Treiber und die damit verbundenen, langsamen Kontextwechsel umgangen werden können.

Die Struktur der Klassen ist in Abbildung 8.9 dargestellt.

Abbildung 8.9: Klassendiagramm von `CHardware` (vereinfacht)

Die Klassen der untersten Abstraktionsebene greifen direkt auf die Hardware-Module (z.B. UART, SPI etc.) des BBB zu. Dabei handelt es sich um sog. Shared-Ressourcen, da alle Instanzen in der Software auf dieselben Hardware-Module zugreifen. Greifen mehrere Instanzen auf dieselbe Hardware zu, kann es zu undefiniertem Verhalten kommen, da Registerzustände unkoordiniert überschrieben werden. Dies führt zu inkonsistenten Zuständen der Peripherie, Datenverlust und Datenkorruption. Besonders beim Ansteuern der Aktoren kann es zu Schäden an der Hardware kommen. Daher soll ein Mechanismus implementiert werden, der verhindert, dass mehrere Instanzen unkoordiniert auf dieselben Hardware-Module zugreifen können.

Dazu sollen Design-Patterns verwendet werden, also bewährte Lösungsmuster für wiederkehrende Softwareprobleme. Ein einfaches Singleton reicht hier nicht aus, da oft mehrere Hardware-Module desselben Typs vorhanden sind (z.B. UART1...4). Daher muss ein Multiton-Verhalten implementiert werden, sodass pro Key (Schlüssel) eine Instanz erzeugt werden kann. Um die Erzeugung und Verwaltung dieser Instanzen zu kapseln und um Code-Wiederholungen zu minimieren, kommt außerdem das Factory-Muster zum Einsatz. Eine Factory erzeugt und verwaltet alle Objekte an einer zentralen Stelle. Dafür wird die Klasse *CInterfaceManager* implementiert, die das Factory-Muster mit dem Multiton-Muster kombiniert.

Alle Klassen der untersten Abstraktionsebene haben nur private Konstruktoren, deklarieren die Klasse *CInterfaceManager* aber als *friend class*. So wird sichergestellt, dass Instanzen dieser Klassen nur durch die Funktion *getInstance()* aus *CInterfaceManager* erstellt werden können. Der Interface-Manager wird als Template implementiert, das aus einem Key und einem Typ Hardware-Modul besteht. Für jeden registrierten Datentyp wird eine Map angelegt, die Pointer auf alle erstellten Instanzen und den zugehörigen Key speichert. Zusätzlich wird in der Map hinterlegt, ob mehrfach auf eine Instanz zugegriffen werden darf. Es ist beispielsweise möglich, mehrere Dynamixel-Motoren mit einer UART zu steuern und viele Hardware-Module haben mehrere Channels. Es liegt aber in der Verantwortung der Interface-Klassen, zu prüfen, ob die Zugriffe kompatibel sind (gleiche Konfiguration des Hardware-Moduls) und mehrere Channels selbst zu verwalten.

Wird nun die *getInstance()*-Methode aufgerufen, wird in der Map geprüft, ob bereits eine Instanz zu diesem Key erstellt wurde. Ist dies nicht der Fall, wird eine neue Instanz auf dem Heap erstellt, in die Map eingefügt und dem Aufrufer übergeben. Die Instanz wird dabei in einem *std::shared_ptr* gespeichert, sodass die Speicherverwaltung automatisch erfolgt und keine manuelle Freigabe notwendig ist. Gibt es bereits eine Instanz zu dem angefragten Key, gibt es zwei Möglichkeiten. Erlauben es sowohl die neue Anfrage, als auch die vorhandene Instanz, dass mehrfache Zugriffe auf eine Instanz möglich sind, wird die bestehende Instanz an den Aufrufer zurückgegeben. Wird der mehrfache Zugriff nicht zugelassen, wird durch die Rückgabe von *std::nullopt* signalisiert, dass das Hardware-Modul nicht verfügbar ist. Zusätzlich wird die Klasse *CInterfaceManager* durch einen Mutex threadsafe gestaltet. Allerdings garantiert dies nicht die Threadsafety der Interface-Instanzen, dies liegt in der Verantwortung der einzelnen Klassen. Da alle Zugriffe in diesem Projekt aus einem Thread, der Control-Comp, erfolgen, ist die Threadsafety garantiert. Die Implementierung der Klasse *CInterfaceManager* ist in Listing 20 zu sehen.

```

1 template <typename InterfaceIdentifier, typename InterfaceType>
2 class CInterfaceManager{
3 public:
4     CInterfaceManager(){}
5     static std::optional<std::shared_ptr<InterfaceType>>
6         → getInstance(InterfaceIdentifier pId, bool pAllowMultipleInstances = false)
7     {
8         std::lock_guard<std::mutex> lock(mMtx);
9         auto it = mInstances.find(pId);
10        if (it != mInstances.end()) {
11            if (pAllowMultipleInstances && it->second.second) {
12                return it->second.first;
13            }
14            else {
15                return std::nullopt;
16            }
17        }
18        auto instance = std::shared_ptr<InterfaceType>(new InterfaceType(pId));
19        mInstances[pId] = {instance, pAllowMultipleInstances};
20        return instance;
21    }
22 private:
23     static inline std::map<InterfaceIdentifier,
24         → std::pair<std::shared_ptr<InterfaceType>, bool>> mInstances;
25     static inline std::mutex mMtx;
26 };

```

Listing 20: Ausschnitt Implementierung des Interface-Managers

IMU

Bei der verwendeten IMU handelt es sich um den Sensor *ICM20948*, der per SPI ausgeleren werden soll. Es gibt im Framework des EMLs bereits eine MMAP-Klasse für die SPI-Peripherie auf dem BBB. Die Klasse *CSPIModuleMMAP* bietet eine Funktion zur Konfiguration der Kommunikation (Wortlänge, Frequenz, Startbit usw.) und eine Funktion für den Austausch von Daten. Die Klasse *CICM20948* soll die Logik zur Konfiguration der SPI-Schnittstelle und Kommunikation mit dem Sensor implementieren.

Das BBB fungiert als SPI-Master, die IMU als SPI-Slave. Das SPI-Protokoll ist vollduplex, d.h. während der Übertragung schicken sowohl Master als auch Slave kontinuierlich Daten. In diesem Kontext wird von einer Leseoperation gesprochen, wenn der Slave den Inhalt spezifizierter Register an den Master schickt. Der Master überträgt währenddessen keine sinnvollen Daten. Bei einer Schreiboperation sendet der Master Daten an den Slave, der diese Daten in ein spezifiziertes Register schreibt. Der Slave sendet dabei Dummy-Daten. Die Konfiguration der SPI-Schnittstelle erfolgt gemäß den Timing-Anforderungen in [15, S. 17] und der allgemeinen Beschreibung der SPI-Kommunikation in [15, S. 31] der *ICM20948*-Sensoren. Die Wortlänge beträgt ein Byte und jede SPI-Operation besteht aus mindestens zwei Bytes. Während des ersten Bytes überträgt das BBB das Adress-Byte, der Sensor schickt keine gültigen Daten. Der Aufbau des Adress-Bytes ist in Abbildung 8.10 gezeigt.

Das MSB gibt an, ob es sich um eine Lese- oder Schreiboperation handelt und die restlichen Bits enthalten die Registeradresse. Der Adressraum des Sensors ist in vier Register-Bänke unterteilt. Dadurch können die 7-Bit-Registeradressen mehrfach belegt sein, welche Funktion angesprochen wird, hängt von der aktuell ausgewählten Bank ab.

Im Falle einer Leseoperation schickt das BBB in allen weiteren Bytes der SPI-Übertragung ungültige Daten und der *ICM20948*-Sensor antwortet mit dem Inhalt des angefragten Registers. Dabei sind Burst-Reads möglich. Besteht eine SPI-Übertragung aus mehr als zwei Bytes, inkrementiert der *ICM20948*-Sensor die Registeradresse und schickt so einen Speicherbereich, beginnend bei der Adresse, die im Adress-Byte spezifiziert wird.

Die SPI-Kommunikation funktioniert bei einer Schreiboperation analog zu der Leseoperation, nur dass der *ICM20948*-Sensor auch während den Daten-Bytes ungültige Daten schickt, während das BBB den Registerinhalt versendet. Die Form der Daten ist in Abbildung 8.10 dargestellt.

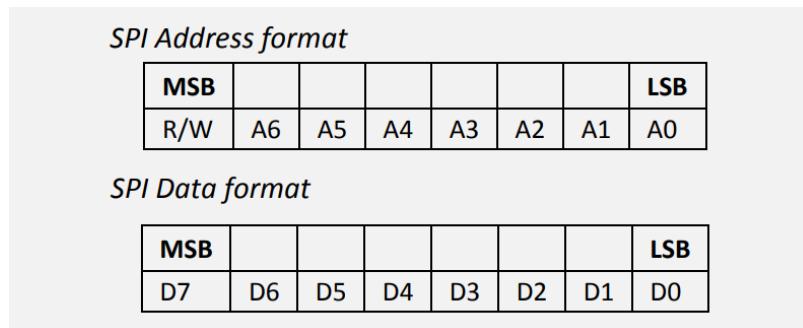


Abbildung 8.10: SPI Kommunikation *ICM20948* [15, S. 31]

Es werden zwei Funktionen für Lese- und Schreiboperation implementiert, um die im vorherigen Absatz beschriebene SPI-Kommunikation umzusetzen. Die *burstRead()* Funktion liest dabei einen Speicherbereich variabler Länge aus. Die Implementierung der Funktion ist in Listing 21 zu finden:

```

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

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

Die IMU unterstützt zwar Burst-Writes, da Schreiboperationen aber nur für die Konfiguration des Sensors genutzt werden, reicht das Schreiben einzelner Register. Die Schreiboperation wird in der Funktion *writeReg()* umgesetzt. Dabei wird zusätzlich eine Möglichkeit zur Überprüfung des Registerinhalts implementiert, um sicherzustellen, dass die Schreiboperation erfolgreich war. Die Implementierung der Funktion ist in Listing 22 zu sehen.

```

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,
19                         " received: ", std::hex,
20                         " static_cast<int>(pData), " static_cast<int>(byte));
21            return Status::FAILED_TO_READ_SENSOR_DATA;
22        }
23    }
24
25    return Status::OKAY;
26 }
```

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

Der Sensor wird mit der Funktion *initImu()* konfiguriert. Standardmäßig werden alle Accelerometer und alle Gyroskope bei maximaler Abtastfrequenz in Betrieb genommen. Zusätzlich kann die Skalierung der Sensoren und die Grenzfrequenz des internen Tiefpassfilters eingestellt werden. Dafür wird zuerst ein Software-Reset durchgeführt, um den Sensor in einen definierten Zustand zu bringen. So kann davon ausgegangen werden, dass alle Register mit den Standardwerten gefüllt sind. Dann wird mit dem Register *WHO_AM_I* geprüft, ob die Kommunikation fehlerlos funktioniert. Enthält das Register den erwarteten Wert, werden die Accelerometer und Gyroskope vorübergehend deaktiviert, um eine fehlerfreie Konfiguration zu garantieren. Danach werden die Skalierung und die Tiefpassfilter der Accelerometer und Gyroskope über die entsprechenden Register auf die gewünschten Werte eingestellt. Zum Abschluss werden die Accelerometer und Gyroskope wieder aktiviert.

Zusätzlich zu den bereits beschriebenen Funktionen soll eine Funktion *readImu()* implementiert werden, die alle Gyroskope und Accelerometer ausliest. Dafür ist es besonders wichtig, alle Sensoren als Burst-Read zu lesen, da eine SPI-Übertragung atomic ist. Das bedeutet, dass die Register während einer SPI-Leseoperation nicht verändert werden. Liest man die einzelnen Bytes in mehreren SPI-Übertragungen, werden die Register von der IMU eventuell überschrieben, sodass korrupte Daten gelesen werden. Die Implementierung der Funktionen ist in Listing 23 zu sehen:

```

1 CICM20948::Status CICM20948::readImu(rawData &pData)
2 {
3     uint8_t numBytes = 12;
4     rawData[numBytes] = {0};
5
6     Status ret = burstRead(ACCEL_XOUT_H, numBytes, rawData);
7     if (ret != Status::OKAY) {
8         return Status::FAILED_TO_READ_SENSOR_DATA;
9     }
10
11    pData.xAccel = (rawData[0] << 8) | rawData[1];
12    pData.yAccel = (rawData[2] << 8) | rawData[3];
13    pData.zAccel = (rawData[4] << 8) | rawData[5];
14
15    pData.xGyro = (rawData[6] << 8) | rawData[7];
16    pData.yGyro = (rawData[8] << 8) | rawData[9];
17    pData.zGyro = (rawData[10] << 8) | rawData[11];
18
19    return Status::OKAY;
20 }
```

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

Die Klasse *CImu* wird als Wrapper für die *CICM20948*-Klasse verwendet. Sie erleichtert die Konfiguration der Sensoren und kalibriert die Daten beim Auslesen, sodass die *readImu()*-Funktion der *CImu*-Klasse direkt die physikalischen Größen enthält.

PWM-Motoren

Es handelt sich sowohl beim Antriebsrad als auch beim Motor für die konstante Kreiseldrehzahl um PWM-gesteuerte Motoren. Allerdings gibt es einige Unterschiede im Detail der Ansteuerung. Daher soll die allgemeine PWM-Logik in der Klasse *CMotorPWM* gekapselt werden, von der die gerätespezifischen Klassen für die beiden Motoren erben können. Die MMAPI-Klasse *CPWMMMAP* für die Steuerung der PWM-Hardware des BBB ist Teil des Frameworks aus dem EML.

Die Klasse *CMotorPWM* konfiguriert und verwaltet den Zugriff auf die Hardware mit einer Instanz der *CPWMMMAP*-Klasse. Außerdem wird eine Funktion bereitgestellt, die einen physikalischen Sollwert in den zugehörigen Duty-Cycle umrechnet. Dafür wird eine lineare Abbildung verwendet, die den Sollwert aus dem physikalischen Bereich auf den entsprechenden Duty-Cycle skaliert. Die Implementierung ist in Listing 24 zu sehen:

```

1 double CMotorPWM::calculateDutyCyclePercent(double pTarget)
2 {
3     return (mPWMCfg.dutyCyclePercentMax - mPWMCfg.dutyCyclePercentMin) /
4         (mPWMCfg.maxTarget - mPWMCfg.minTarget) * (pTarget - mPWMCfg.minTarget) +
5         mPWMCfg.dutyCyclePercentMin;
6 }
7
8 CMotorPWM::Status CMotorPWM::setPWM(double pDutyCyclePercent)
9 {
10    if (mPWM == nullptr) {
11        return Status::PWM_MODULE_NOT_AVAILABLE;
12    }
13    if (mPWM->setDutyCycle(mPWMPin, pDutyCyclePercent) != CPWMMMAP::Status::OKAY)
14    {
15        return Status::PWM_MODULE_ERROR;
16    }
17    return Status::OKAY;
18 }
```

Listing 24: Ausschnitt Implementierung der Klasse *CMotorPWM*

Die Klasse *CMaxonMotor* implementiert neben der Ansteuerung des Enable-Ausgangs auch die Logik zur Bestimmung der Geschwindigkeit des Antriebsrads. Der Enable-Ausgang, mit dem der Motor aktiviert und deaktiviert werden kann, wird durch einen GPIO angesteuert. Die Klasse *CGPIOMMAP* ist ebenfalls Teil des EML-Frameworks. Um die Geschwindigkeit des Fahrzeugs zu bestimmen, wird der Motortreiber des Antriebsrads so konfiguriert, dass die aktuelle Drehzahl des Motors mit einer analogen Spannung ausgegeben wird. Diese Spannung kann mithilfe der Klasse *CADCMMAP* gemessen werden.

Die Klasse *CSpin66Motor* implementiert die Aktivierung und Deaktivierung des Motors durch einen bestimmten PWM-Duty-Cycle (siehe Unterabschnitt 4.1.2).

ADC-Memory-Map

Mit dem ADC-Wandler des BBB soll die analoge Spannung des Motortreibers des Antriebsrads erfasst werden, die Informationen über die aktuelle Geschwindigkeit enthält. Um einen ausreichend schnellen Zugriff auf die ADC-Hardware sicherzustellen, soll die Ansteuerung direkt über die Hardware-Register (MMAP) erfolgen.

Der ADC-Wandler ist als sequenzgesteuerter Wandler aufgebaut. Die zentralen Bausteine sind 16 sog. Steps. Jeder Step enthält die vollständige Konfiguration für eine Messung. Dazu zählt der zu messende Eingang, die Sample-Zeit, der Betriebsmodus und die Mittlung mehrerer Messungen. Der ADC-Wandler arbeitet alle aktiven Steps zyklisch ab und schreibt die Ergebnisse der jeweiligen Messung in einen von zwei First In First Out (FIFO)-Puffern. Dabei wird die Messung mit dem Tag des zugehörigen Steps versehen. Es gibt zwei relevante Betriebsmodi: One-Shot und kontinuierliche Messung. Bei One-Shot Messungen wird die im Step konfigurierte Messung einmalig nach dem Trigger ausgeführt, bei der kontinuierlichen Messung werden die Messungen der aktiven Steps zyklisch durchgeführt.

Die Ansteuerung der Hardware soll in der Klasse *CADCMMAP* erfolgen. Die einzelnen Steps sind dabei die Ressourcen, die gegen mehrfachen Zugriff geschützt werden müssen. Daher wird mit der Klasse *CInterfaceManager* sichergestellt, dass pro Step eine Instanz erstellt werden kann. Da jedoch mehrere Instanzen parallel auf denselben ADC-Wandler zugreifen können, besteht die Gefahr von Datenverlust. Liest eine Instanz den FIFO-Puffer, könnten dabei Messwerte entfernt werden, die eigentlich einem anderen Step zugeordnet sind. Daher wird ein statisches Array *mValueQueues* eingeführt, das eine Queue für jeden Step enthält. Zusätzlich wird die Funktion *readFifo()* (siehe Listing 25) implementiert, die alle Daten aus dem zugehörigen FIFO-Puffer liest und an die richtige Queue anhängt. Zum Auslesen der Daten kann jede Instanz diese Funktion aufrufen, sodass alle FIFO-Einträge dem richtigen Step zugeordnet werden können. Anschließend kann jede Step-Instanz auf die Daten in der eigenen Queue zugreifen.

```

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*

Um die aktuellen ADC-Daten zu lesen, wird die Funktion *readADC()* implementiert. Wird der Step im One-Shot Modus betrieben, wird zuerst die Durchführung der Messung getriggert. Anschließend wird die *readFifo()*-Funktion so lange aufgerufen, bis ein Messwert in der Queue des Steps auftaucht. Wird der Step im kontinuierlichen Modus betrieben, wird die Funktion *readADC()* einmal aufgerufen, um alle Daten aus der FIFO zu lesen. In beiden Fällen wird anschließend der letzte Wert in der Queue zurückgegeben, und alle anderen Daten verworfen, da sie nicht mehr aktuell sind. Die Implementierung der Funktion *readADC()* ist in Listing 26 zu sehen.

```

1 CADCMMAP::Status CADCMMAP::readADC(uint16_t& pValue)
2 {
3     switch (mADCConfig.mode)
4     {
5         case CADCConfig::Mode::ONESHOT:
6             setBits(OFFS_STEPEENABLE, 0x01 << mStepIdx);
7             while (mValueQueues[mStepIdx-1].empty()) {
8                 readFifo();
9             }
10            break;
11        case CADCConfig::Mode::CONTINUOUS:
12            readFifo();
13            break;
14        default:
15            return Status::CONFIG_ERROR;
16        }
17        bool newVal = false;
18        // Only returns latest value, all other values are discarded
19        while (!mValueQueues[mStepIdx-1].empty()) {
20            pValue = mValueQueues[mStepIdx-1].front();
21            mValueQueues[mStepIdx-1].pop();
22            newVal = true;
23        }
24        if (!newVal) {
25            REPORT_ERROR("No new value available");
26            return Status::NO_VALUE;
27        }
28        return Status::OKAY;
29    }

```

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

Dynamixel-Servomotoren

Die Ansteuerung der Dynamixel-Servomotoren erfolgt mit den UARTs des BBB, welche durch die Klasse *CUARTMMAP* angesteuert werden, die Teil des EML-Frameworks ist. Die Kommunikation folgt den Richtlinien des Dynamixel-Protokolls 2.0 (Spezifikationen siehe [6]). Im EML-Framework gibt es auch eine Klasse *CDynamixelUARTMMAP*, welche die grundlegende Kommunikation mit dem Dynamixel Protokoll 2.0 bereits implementiert. Darauf aufbauend kann eine allgemeine Funktion zum Lesen und Schreiben von Registern eines Dynamixel-Servomotors umgesetzt werden. Zusätzlich werden Funktionen zum Schreiben und Lesen der für dieses Projekt relevanten Register (Torque Enable, Bus Watchdog, Goal Position uvm.) erstellt. Die Liste aller Register und deren Funktionalität findet sich in [7].

Die Klassen *CPositionDynamixel* und *CVelocityDynamixel* konfigurieren einen Dynamixel-Servomotor entsprechend des Betriebsmodus und stellen die nötigen Funktionen zur Ansteuerung und Aktivierung der Motoren bereit (*setPosition()*, *setVelocity()*, *setTorqueEnable()* etc.).

8.3.3 Comm-Comp

Die Comm-Comp ist für das Verschicken der Fahrzeugdaten an die Python-Gui per UDP verantwortlich. Wie bei der Control-Comp gibt es dedizierte Klassen für die jeweiligen Aufgaben. Die UDP-Kommunikation wird von der Klasse *CServer* gehandhabt, die Datenpakete der Form *CMessage* versendet. Nach der Initialisierung der Kommunikation werden jeden Abtastschritt die aktuellen Fahrzeugdaten verschickt. Die Synchronisierung erfolgt dabei wie zu Beginn des Kapitels beschrieben mit einer Semaphore. Die Implementierung der Schleife, die für das Verschicken der Daten zuständig ist, ist in Listing 27 zu sehen:

```

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 disconnected or error while transmitting
15                message" << endl;
16            break;
17        }
18        cout<<"Comm End"<<endl;
19    };

```

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

Datenklassen

Die Klasse *CMessage* speichert einen *uint8_t*-Array. Die Länge wird dabei durch die Größe der Klasse *CContent* bestimmt, welche die zu sendenden Fahrzeugdaten enthält. Mit der Funktion *setContent()* wird ein *CContent*-Objekt in dem *uint8_t*-Array gespeichert. Dies ist nötig, da beim Verschicken von UDP-Paketen nur untypisierte Bytefolgen übertragen werden können.

Server

Die Klasse *CServer* implementiert einen UDP-Server und besteht aus drei Funktionen: Die *init()*-Funktion erstellt und initialisiert ein UDP-Socket und bindet diesen an einen festen Port. Die *init()*-Funktion ist in Listing 28 zu sehen:

```

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*

Da das BBB immer unter demselben mDNS-Namen erreichbar ist und der Port nicht verändert wird, kann die Python-Gui gezielt Nachrichten an die Comm-Comp senden. Da sich die IP-Adresse des Entwicklungsrechners aber häufig ändert, bzw. verschiedene Rechner verwendet werden, die unterschiedlich erreichbar sind, muss die Comm-Comp auf eine Nachricht des Entwicklungsrechners warten. Dafür ist die Funktion *waitForClient()* zuständig. Dabei wartet der Thread blockierend für eine bestimmte Zeit auf eine UDP-Nachricht. Geht die erwartete Nachricht ein, wird die Adresse des Absenders gespeichert und die Verbindung als „aktiv“ markiert (siehe Listing 29).

```

1 bool CServer::waitForClient(size_t pTimeout)
2 {
3     fd_set readfds;
4     FD_ZERO(&readfds);
5     FD_SET(mSocket, &readfds);
6
7     timeval timeout{};
8     timeout.tv_sec = pTimeout;
9     timeout.tv_usec = 0;
10
11    char buffer;
12    socklen_t addrLen = sizeof(mClientAddr);
13
14    int result = select(mSocket + 1, &readfds, nullptr, nullptr, &timeout);
15    if (result > 0 && FD_ISSET(mSocket, &readfds)) {
16        ssize_t recvLen = recvfrom(mSocket, &buffer, sizeof(buffer), 0,
17            (sockaddr*)&mClientAddr, &addrLen);
18        if (recvLen > 0) {
19            mConnected = true;
20            return true;
21        }
22    }
23    return false;
}

```

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

Anschließend können mit der Funktion *transmitMessage()* *CContent*-Objekte an die gespeicherte Adresse versendet werden. Die Implementierung ist in Listing 30 gezeigt.

```

1 bool CServer::transmitMessage(CContent &pContent)
2 {
3     CMessage msg;
4     msg.setContent(pContent);
5     if (!mConnected) {
6         REPORT_ERROR("No client connected");
7         return false;
8     }
9
10    ssize_t sent = sendto(mSocket, msg.mData, sizeof(CMessage), 0,
11        (sockaddr*)&mClientAddr, sizeof(mClientAddr));
12    return sent == (ssize_t)sizeof(CMessage);
}

```

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

8.4 Python-Gui

Um die Daten der durchgeföhrten Experimente auf dem Entwicklungsrechner zu visualisieren und aufzuzeichnen, wird eine Python-Gui implementiert. Da die Gui die Datenmenge in Echtzeit verarbeiten und darstellen muss, kommt das *PyQt5*-Framework zum Einsatz. Die Implementierung der Logik des Frameworks erfolgt in C++ und ist besonders für kontinuierlich aktualisierte Anwendungen optimiert.

8.4.1 Hauptprogramm

Grundsätzlich besteht die Anwendung aus drei Modulen: einem UDP-Klient *UDPCClient*, den Plots der Daten *PlotManager* und dem Logger der Daten *DataRecorder*. Die Module werden durch das Hauptprogramm *main.py* verwaltet. Das Hauptprogramm übernimmt die Initialisierung der grafischen Oberfläche und delegiert die Daten zwischen den Modulen.

Nach der Initialisierung der einzelnen Module wird ein Hintergrundthread (siehe Listing 31) gestartet, in dem der UDP-Klient kontinuierlich die Fahrzeugdaten empfängt. Anschließend werden die Daten an den Logger weitergegeben und in einer Queue gespeichert. Währenddessen wird im Main-Thread die Gui initialisiert und ein Timer gestartet. Im Timer-Callback (siehe Listing 32) wird die Queue mit den Fahrzeugdaten periodisch verarbeitet, und die Daten werden an den Plot-Manager übergeben, um die Anzeigeelemente der Gui zu aktualisieren.

```

1 def receive_loop():
2     try:
3         while True:
4             data = client.recv_message(ctypes.sizeof(CContent))
5             msg = CContent.from_bytes(data)
6             recorder.record(msg)
7             data_queue.put(msg)
8     except Exception as e:
9         print("Error while receiving data:", e)
10    except KeyboardInterrupt:
11        print("End program")
12
13 thread = threading.Thread(target=receive_loop, daemon=True)
14 thread.start()
```

Listing 31: Ausschnitt Implementierung des Hintergrundthreads der Python-Gui

```

1 def timer_callback():
2     while not data_queue.empty():
3         msg = data_queue.get_nowait()
4         plotter.update(msg)
5
6 timer = QTimer()
7 timer.timeout.connect(timer_callback)
8 timer.start(10)

```

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

8.4.2 UDP-Klient

Wie in Unterunterabschnitt 8.3.3 beschrieben, schickt die Python-Gui eine UDP-Nachricht an das BBB, um die Kommunikation zu starten. Das BBB beginnt dann, die Fahrzeugdaten an die aus dem UDP-Paket extrahierte Adresse zu schicken. Die Klasse *UDPClient* implementiert die Logik für die Kommunikation. Dazu wird ein UDP-Socket erstellt, und die Nachricht zur Initialisierung der Kommunikation per mDNS an das BBB geschickt (siehe Listing 33):

```

1 def send_initial_message(self):
2     if self.sock is None:
3         raise RuntimeError("Socket not initialized")
4     initial_byte = b'\x01'
5     try:
6         self.ip = socket.gethostbyname(self.host)
7         self.sock.sendto(initial_byte, (self.ip, self.port))
8         print(f"Sent initial byte to {self.ip}:{self.port}")
9         return True
10    except Exception as e:
11        print(f"Failed to send initial byte: {e}")
12        return False

```

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

Zusätzlich wird eine Funktion zum Empfangen der UDP-Pakete des BBB implementiert. Dafür wird für eine bestimmte Zeit gewartet, ob ein UDP-Paket beim Socket eingeht. Die Implementierung ist in Listing 34 zu sehen.

```

1 def recv_message(self, bufsize) -> bytes:
2     if self.sock is None:
3         raise RuntimeError("Socket not initialized")
4     try:
5         data, addr = self.sock.recvfrom(bufsize)
6         return data
7     except socket.timeout:
8         print("Receive timed out")
9         return b''
10    except Exception as e:
11        print(f"Receive error: {e}")
12        return b''

```

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

8.4.3 Datenextraktion

Die Daten aus den UDP-Paketen müssen korrekt interpretiert werden, da sie als einfache Byte-Folge übertragen und empfangen werden. Dafür wird das Modul *ctypes* verwendet, was es ermöglicht, C-ähnliche Datenstrukturen in Python zu implementieren. Damit wird eine Datenklasse *CContent* erstellt, deren Speicherlayout dem der C++-Klasse *CContent* entspricht. Die eingehenden Rohdaten können dann mit der Funktion *from_bytes()* in eine Instanz der *CContent*-Klasse umgewandelt werden (siehe Listing 35).

```

1 class CContent(ctypes.Structure):
2     _fields_ = [
3         ("mTimeUs", ctypes.c_int64),
4         ("mStateData", CStateData),
5         ("mMotorData", CMotorData),
6     ]
7
8     @classmethod
9     def from_bytes(cls, data: bytes) -> "CContent":
10        if len(data) != ctypes.sizeof(cls):
11            raise ValueError(f"Expected {ctypes.sizeof(cls)} bytes, got"
12                           f" {len(data)}")
12        return cls.from_buffer_copy(data)

```

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

8.4.4 Datenaufzeichnung

Die Klasse *DataRecorder* übernimmt die Aufzeichnung der empfangenen Daten in eine CSV-Datei. Abhängig vom gewählten Modus werden verschiedene Teilmengen der Daten (IMU-, Zustands- oder Motorinformationen) extrahiert und in die Datei geschrieben. Die Konfiguration der Modi erfolgt nach einem generischen Muster, sodass der Recorder sehr schnell an neue Datenformate angepasst werden kann. Für jede Teilmenge an Daten wird eine Funktion erstellt, die diese Daten aus dem Datenpaket vom BBB extrahiert. Diese

Funktionen werden in einem Array gespeichert, sodass zur Laufzeit über dieses Array iteriert werden kann, um alle Teilmengen aufzuzeichnen. Die Implementierung der *record()*-Funktion ist in Listing 36 zu sehen.

```

1 def record(self, msg: CContent):
2     row = [msg.mTimeUs]
3     for extractor in self.extractors:
4         row += extractor(msg)
5     self.writer.writerow(row)

```

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

8.4.5 Plot-Manager

Das zentrale Widget wird durch den *PlotManager* bereitgestellt. Abhängig von den ausgewählten Plots (z.B. IMU-Werte, Zustandsgrößen, Motordaten etc.) werden die zugehörigen Plot-Widgets erzeugt und in einem Rasterlayout angeordnet. Jeder Plot ist für die Aktualisierung seiner Daten selbst verantwortlich und implementiert hierzu eine einheitliche Schnittstelle. Ähnlich wie beim Recorder werden alle aktiven Plots in einem Array gespeichert, sodass zur Laufzeit nur über dieses Array iteriert werden muss.

Zur Vereinheitlichung der Implementierung existiert eine abstrakte Basisklasse *BasePlot*, welche die Methoden *update()* und *get_widget()* als abstrakte Schnittstellen vorgibt. Sämtliche konkreten Plots (z. B. *IMUPlot*, *StatePlot*, *MotorPlot*) leiten sich von dieser Klasse ab. Dadurch wird eine modulare und erweiterbare Struktur gewährleistet, bei der neue Plots ohne Änderungen an den übergeordneten Klassen ergänzt werden können.

9 Zusammenfassung

Mit dem *Monowheeler* wurde in dieser Arbeit ein neuartiges Konzept für ein mobiles, aktiv stabilisiertes, einrädriges Fahrzeug entwickelt und erprobt. Zur Kontrolle der instabilen Nickdynamik kann das Rad relativ zur Plattform verschoben werden. Die ebenfalls instabile Rollbewegung wird durch ein Gyroskop gesteuert. Durch die Nutzung der gekoppelten Dynamik der Roll- und Gierbewegung kann das Gieren gezielt gesteuert werden, sodass Manöver zur Richtungsänderung möglich sind. Mit der Implementierung geeigneter Kontrollalgorithmen kann das Fahrzeug effektiv stabilisiert werden und komplexe Manöver wie Richtungsänderungen durchführen.

Um dieses Ziel zu erreichen, wurde zunächst die vorliegende Hardware in Betrieb genommen. Ein besonderes Augenmerk lag hierbei auf der Implementierung geeigneter Sicherheitsmechanismen zur Abschaltung der Aktoren, damit Beschädigungen der Hardware ausgeschlossen werden können. Für die Anbindung der Steuersignale und Leistungselektronik, des Mikrocontrollers sowie der Aktorik und der Sensorik wurde eine Platine entworfen und gefertigt. Um die Datenintegrität der Steuersignale trotz der störenden Leistungselektronik sicherzustellen, wurde das Layout unter besonderer Berücksichtigung der EMV ausgelegt. Dazu wurden Signal- und Leistungselektronik räumlich getrennt, getrennte Masseflächen mit definiertem Verbindungspunkt verwendet und Leitungsführungen so gestaltet, dass Rückstrompfade möglichst kurz und störungsarm bleiben. Durch den gezielten Einsatz von Glättungs- und Stützkondensatoren konnte zudem eine stabile Versorgungsspannung aller Komponenten gewährleistet werden. Diese Maßnahmen führten insgesamt zu einer sauberen Signalführung und einem robusten elektrischen Verhalten des Systems.

Zur Bestimmung der aktuellen Orientierung des Fahrzeugs unter Echtzeitanforderungen für den Regelkreis wurde ein geeignetes Sensorsystem ausgewählt und entsprechender Algorithmus zur Filterung und Datenfusion implementiert. Zum Einsatz kommt eine IMU, deren Messwerte für die Beschleunigungen und Winkelgeschwindigkeiten sensorintern gefiltert werden. Zusätzlich kommen digitale Butterworth-Tiefpassfilter zum Einsatz, um das SNR-Verhältnis zu optimieren. Für eine robuste Schätzung der Fahrzeugzustände werden die Messdaten anschließend mit einem Komplementärfilter fusioniert.

Zur Stabilisierung des inhärent instabilen Fahrzeugs konnten zwei größtenteils unabhängige Teilsysteme identifiziert werden: das Teilsystem Nicken und das Teilsystem Rollen und Gieren. Für beide Systeme konnten geeignete Konzepte zur Regelung der jeweiligen Freiheitsgrade entworfen werden. Dafür wurde ein iterativer Prozess genutzt, der als Ausgangslage analytisch ausgelegte Regler verwendet. Die Parameter wurden zunächst schrittweise mithilfe einer Simulation des Systemverhaltens optimiert. Anschließend wurden die Ergebnisse am realen Fahrzeug validiert und verfeinert. Dabei konnte gezeigt werden, dass das vereinfachte Modell des *Monowheelers* die Dynamik des realen Fahrzeugs beim Balancieren zutreffend abbildet. Für beide Teilsysteme wurden klassische und modellbasierte Reglerkonzepte ausgelegt und verglichen. Für das Teilsystem Nicken kommt ein Gain-Scheduling LQR-Regler

zum Einsatz, der gegenüber einem klassischen PID-Regler eine bessere Dynamik bietet. Das Teilsystem Rollen und Gieren wird zum Balancieren entkoppelt betrachtet, sodass sich die Freiheitsgrade im Arbeitspunkt gegenseitig nicht beeinflussen. Die Rollbewegung kann mit einem PID-Regler effektiv kontrolliert werden. Da die Stellgröße des Kreisels aufgrund der begrenzten Winkelauslenkung limitiert ist, muss der Regler die Orientierung des Fahrzeugs so manipulieren, dass der Kreisel immer in die Mitte gedreht werden kann. Dazu generiert ein zweiter PID-Regler eine Trajektorie für die Rollbewegung. Ein modellbasierter LQR-Regler konnte ebenfalls entworfen werden, bietet aufgrund seiner Struktur aber nicht die Möglichkeit, anderen Trajektorien für die Rollbewegung zu folgen. Daher kommt das System aus zwei PID-Reglern zum Einsatz.

Neben der Stabilisierung des Fahrzeugs sollte in dieser Arbeit ein Konzept zum Kurvenfahren untersucht werden. Dafür wird die Kopplung der Freiheitsgrade Rollen und Gieren ausgenutzt, um durch eine Rollbewegung eine gezielte Gierbewegung zu erzielen. Die Rollbewegung wird durch eine reglerbasierte Trajektorie vorgegeben und von dem Regler zur Steuerung der Rollbewegung umgesetzt. In mehreren Experimenten konnte gezeigt werden, dass das Fahrzeug mit diesem Konzept Kurven mit variablem Radius zuverlässig fahren kann.

Um die Hardware anzusteuern und den Regelkreis zur Steuerung des Fahrzeugs zu implementieren, wird ein Mikrocontroller mit Linux-RealTime Kernel genutzt. Die Software wurde in der Programmiersprache *C++* implementiert und bietet ein einheitliches Interface für die Verwaltung der Hardware-Peripherie des Mikrocontrollers. Zusätzlich wurde die Echtzeiteignung des Systems untersucht. Die Fahrzeugdaten während eines Experiments können live mit einer Python-Gui visualisiert und aufgezeichnet werden.

Damit konnte das in der Aufgabenstellung formulierte Ziel dieser Arbeit erreicht werden: Der *Monowheeler* wurde erfolgreich mithilfe des gyroskopischen Effekts stabilisiert, und das Fahrzeug ist in der Lage, eigenständig Kurven zu fahren. Das vorgestellte Konzept zeigt, dass das unteraktivierte, instabile System mit stark gekoppelter Dynamik effektiv kontrolliert werden kann. Damit wurde ein neuartiger Ansatz zur aktiven Stabilisierung und Steuerung eines einrädrigen Fahrzeugs erfolgreich umgesetzt und experimentell validiert.

10 Ausblick

Bei den durchgeführten Experimenten zeigen sich auch die Grenzen des aktuellen Systems. Das mathematische Modell des Fahrzeugs ist hauptsächlich für das Balancieren ausgelegt und ignoriert dynamische Effekte des Antriebsrads beim Kurvenfahren. Klassische Regelungsansätze können das unteraktuierte Mehrgrößensystem Rollen und Gieren nicht als Gesamtsystem steuern. Durch die Kaskadierung mehrerer PID-Regler kann das System jedoch mit Einschränkungen kontrolliert werden. Dazu wird durch Vorgabe einer geeigneten Trajektorie für die Rollbewegung der gewünschte Effekt herbeigeführt. Beim Balancieren wird durch das Schrägstellen des Fahrzeugs der Kreiselwinkel minimiert. Dadurch kann die Gierbewegung nicht kontrolliert werden und es muss zum Balancieren von einer Entkopplung der Roll- und Gierbewegung ausgegangen werden. Beim Kurvenfahren soll die Gierbewegung gesteuert werden, dadurch kann aber der Kreiselwinkel nicht mehr berücksichtigt werden.

Das führt dazu, dass das Fahrzeug im Arbeitspunkt effektiv auf unbegrenzte Zeit stabilisiert werden kann. Beim Kurvenfahren kann das Fahrzeug die Gierbewegung so lange gezielt kontrollieren, wie der Kreisel genügend Reserve bietet. Durch das Kurvenfahren verlässt das Fahrzeug aber den für das Balancieren zulässigen Arbeitspunkt, sodass es das Fahrzeug nach dynamischen Manövern teilweise nicht schafft, in den stabilen Arbeitspunkt zum Balancieren zurückzukehren. Dieser Effekt kann verringert werden, indem das Fahrzeug nach einem Manöver angehalten wird, da der Arbeitspunkt im Stand aufgrund der größeren Haftreibung und fehlende dynamischen Kopplung zwischen Rollen und Gieren durch das Antriebsrad größer ist.

Um das nichtlineare, unteraktuierte System mit Stellgliedbegrenzungen und eng gekoppelten Freiheitsgraden als Gesamtsystem steuern zu können, können in zukünftigen Arbeiten verschiedene Ansätze verfolgt werden. Für alle Ansätze ist eine präzise Modellierung der Dynamik des Fahrzeugs nötig, inklusive genauer Kenntnisse der Reibungsparameter. Ein genaues Modell des Fahrzeugs kann beispielsweise durch den Lagrange-Formalismus bestimmt werden. Die Reibungsparameter können experimentell durch gezielte Messungen und anschließender Modellanpassung ermittelt werden. Mögliche Ansätze, um das Gesamtsystem effektiv zu steuern, lassen sich in drei Kategorien einteilen.

- **Fortgeschrittene Trajektorienplaner:** Fortgeschrittene Trajektorienplaner wie Multiple Shooting oder Sampling-basierte Methoden (z.B. RRT*) erlauben die Berechnung komplexer Bewegungen für stark nichtlineare und unteraktuierte Systeme. Sie berücksichtigen direkt die Systemdynamik, Kopplungen zwischen Freiheitsgraden und die Stellgrößenbegrenzungen. Typischerweise werden spezielle Trajektorien offline optimiert und können anschließend als Referenz für ein Feedback-System dienen.
- **Lernbasierte Ansätze:** Lernbasierte Methoden, wie Reinforcement Learning oder modellbasiertes Optimal Control Learning, ermöglichen es, komplexe Steuerstrategien direkt aus Simulationen oder experimentellen Daten zu erlernen. Sie sind besonders

geeignet, wenn die Dynamik stark nichtlinear ist oder nicht exakt modelliert werden kann, da sie Kopplungen und Reibungseffekte adaptiv berücksichtigen können.

- **Modellprädiktive Ansätze:** Modellprädiktive Regelungen (MPC/NMPC) nutzen ein mathematisches Modell des Systems, um in Echtzeit optimale Stellgrößen unter Berücksichtigung von Systemgrenzen und Constraints zu berechnen. Sie kombinieren die Vorteile der Trajektorienplanung und der Constraints-Behandlung und sind besonders effektiv, um interaktivierte Systemen präzise Trajektorien folgen zu lassen, selbst bei stark gekoppelten Freiheitsgraden. Im Gegensatz zu Trajektorienplanern sind sie nicht auf die Umsetzung einer einzelnen, vorgegebenen Bewegung beschränkt, sondern können das System allgemein steuern. Allerdings sind sie aufgrund der kontinuierlichen Optimierung sehr rechenintensiv.

Literaturverzeichnis

- [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 (online - zuletzt aufgerufen am 21.08.2025) (siehe S. 31).
- [2] *Balancing on Inline Wheels with Gyroscopes*. URL: <https://www.youtube.com/watch?v=drxzsoDnz14> (online - zuletzt aufgerufen am 09.09.2025) (siehe S. 69).
- [3] *Best Practices for Board Layout of Motor Drivers*. URL: <https://www.ti.com/lit/an/slva959b/slva959b.pdf?ts=1749690141678> (online - zuletzt aufgerufen am 19.08.2025) (siehe S. 25, 26).
- [4] Bloch, A. M. *Nonholonomic Mechanics and Control*. 2nd. Bd. 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> (siehe S. 9, 83).
- [5] Brennan, L. „Means for imparting stability to unstable bodies“. 796893. US Patent Office. 1905 (siehe S. 3).
- [6] *Dynamixel Protocol 2.0*. URL: <https://emanual.robotis.com/docs/en/dxl/protocol2/> (online - zuletzt aufgerufen am 18.09.2025) (siehe S. 122).
- [7] *Dynamixel XH540-W150-T/R*. URL: <https://emanual.robotis.com/docs/en/dxl/xh540-w150/> (online - zuletzt aufgerufen am 18.09.2025) (siehe S. 122).
- [8] Franklin, G. F., Powell, J. D. ; 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 (siehe S. 49, 50, 55).
- [9] Goodwin, G. C., Graebe, S. F. ; Salgado, M. E. *Control System Design*. Prentice Hall, 2000. ISBN: 9780139586538 (siehe S. 49, 55, 56, 58).
- [10] *Gyro-X-1967*. URL: https://www.lanemotormuseum.org/wp-content/uploads/2014/12/gyrox_1967web1a-cdd.jpg (online - zuletzt aufgerufen am 20.09.2025) (siehe S. 3).
- [11] Hang, C., Åström, K. ; Ho, W. „Refinements of the Ziegler–Nichols tuning formula“. In: *IEE Proceedings D (Control Theory and Applications)* 138 (2 1991), S. 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> (siehe S. 50).

- [12] Henrich, L. A. „Evaluation of IMU Orientation Estimation Algorithms Using a Three-Axis Gimbal“. Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU), 2019. URL: https://www.mad.tf.fau.de/files/2020/02/BT_LeaHenrich_Final.pdf (online - zuletzt aufgerufen am 22.08.2025) (siehe S. 33).
- [13] *Hespanha-Linear Systems-Part IV*. URL: <https://metr4202.uqcloud.net/2014/lectures/Hespanha-Linear%20Systems-Part%20IV.pdf> (online - zuletzt aufgerufen am 05.09.2025) (siehe S. 59).
- [14] Hose, H., Weisgerber, J. ; 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> (siehe S. 4, 86).
- [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> (online - zuletzt aufgerufen am 21.08.2025) (siehe S. 31, 35, 36, 40, 115, 116).
- [16] *IPD85P04P4L-06 datasheet*. URL: https://www.mouser.de/datasheet/2/196/Infineon_IPD85P04P4L_06_DataSheet_v01_01_EN-3362745.pdf (online - zuletzt aufgerufen am 19.08.2025) (siehe S. 27).
- [17] *ISS Motion Control System*. URL: https://ntrs.nasa.gov/api/citations/20240003654/downloads/ISS_MCS_Ops_POIWG_April24_Final_PDF.pdf (online - zuletzt aufgerufen am 20.09.2025) (siehe S. 3).
- [18] Jacot, A. D. ; Liska, D. J. „Control Moment Gyros in Attitude Control“. In: *Journal of Spacecraft and Rockets* 3.9 (1966), S. 1313–1320 (siehe S. 3).
- [19] *James Bruton - One-Wheel Balancing Robot*. URL: <https://www.youtube.com/watch?v=fNQkZ7MmGio> (online - zuletzt aufgerufen am 23.09.2025) (siehe S. 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> (online - zuletzt aufgerufen am 15.08.2025) (siehe S. 20).
- [21] Lappas, V. J., Steyn, W. H. ; Underwood, C. I. „Attitude Control for Small Satellites Using Control Moment Gyros“. In: *Acta Astronautica* 51.1-9 (2002), S. 101–111. DOI: [10.1016/S0094-5765\(02\)00089-9](https://doi.org/10.1016/S0094-5765(02)00089-9) (siehe S. 3).
- [22] Lew, E., Orazov, B. ; O'Reilly, O. „The dynamics of Charles Taylor's remarkable one-wheeled vehicle“. In: *Regular and Chaotic Dynamics* 13 (Aug. 2008), S. 257–266. DOI: [10.1134/S1560354708040035](https://doi.org/10.1134/S1560354708040035) (siehe S. 2, 5).
- [23] *Maxon Escon 70/10 Gerätreferenz*. URL: https://www.maxongroup.de/medias/sys_master/root/9350562643998/422969-ESCON-70-10-Geraete-Referenz-De.pdf (online - zuletzt aufgerufen am 15.08.2025) (siehe S. 21, 22).
- [24] *MEGA MOTOR ACn 22/20/3*. URL: https://www.megamotor.cz/v4/scriptx/default.php?&sid=6c18bedb3cbe944b86bd8fcaa81a31b6&page_id=lang_ger (online - zuletzt aufgerufen am 15.08.2025) (siehe S. 20).
- [25] Mellodge, P. „Chapter 4 - Characteristics of Nonlinear Systems“. In: *A Practical Approach to Dynamical Systems for Engineers*. Hrsg. von Mellodge, P. Woodhead Publishing, 2016, S. 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> (siehe S. 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> (siehe S. 43).
- [27] *Mini Wheelbot*. URL: https://www.linkedin.com/posts/sebastian-trimpe-2472a0a3_icra2025-ieee-robotics-activity-7328901914195103746-zr5j (online - zuletzt aufgerufen am 20.09.2025) (siehe S. 4).
- [28] *Model Predictive Control*. URL: https://www.ist.uni-stuttgart.de/research/group-of-frank-allgoewer/model-predictive-control/?utm_source=chatgpt.com (online - zuletzt aufgerufen am 06.09.2025) (siehe S. 56).
- [29] Oppenheim, A. V. ; Schafer, R. W. *Discrete-Time Signal Processing*. 3rd. Pearson, 2010. ISBN: 978-0-13-198842-2 (siehe S. 35, 43).
- [30] Rahiman, W., Ghazali, M. H. M. ; Mahmood, I. A.-T. „Revolutionizing Marine Stability: A Review of Cutting-Edge Gyro Stabilizer Designs and Techniques for Vessels“. In: *IEEE Access* 13 (2025), S. 89542–89555. DOI: 10.1109/ACCESS.2025.3571710 (siehe S. 4).
- [31] *Robust control (H infinity and H2)*. URL: <https://dycsyt.com/en/robust-control-h-infinity-and-h2/> (online - zuletzt aufgerufen am 05.09.2025) (siehe S. 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 (online - zuletzt aufgerufen am 19.08.2025) (siehe S. 28).
- [33] Ruben Egle, S. W. „Entwurf und Konzeption eines Monowheelers“. Hochschule Karlsruhe, 2024 (siehe S. 2, 5, 7–11, 13, 14, 20, 21, 23, 26, 27, 36, 49–51, 55, 66).
- [34] Shen, J. ; Hong, D. „OmBURo: A Novel Unicycle Robot with Active Omnidirectional Wheel“. In: *2020 IEEE International Conference on Robotics and Automation (ICRA)*. 2020, S. 8237–8243. DOI: 10.1109/ICRA40945.2020.9196927 (siehe S. 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> (siehe S. 3).
- [36] Taylor, C. F. „One-Wheeled Vehicle“. 3145797. US Patent Office. 1964 (siehe S. 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 (online - zuletzt aufgerufen am 20.09.2025) (siehe S. 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 (online - zuletzt aufgerufen am 19.08.2025) (siehe S. 28).
- [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 (online - zuletzt aufgerufen am 22.08.2025) (siehe S. 32, 33).
- [40] Wietzke, J. *Embedded Technologies*. Springer Berlin, Heidelberg, 2012 (siehe S. 100).

- [41] Wietzke, J. ; Tran, M. T. *Automotive Embedded Systeme*. Springer Berlin, Heidelberg, 2005 (siehe S. 100).
- [42] Wietzke, P. D.-I. J. ; Meindl, M. M. *Mechatronische Systeme Vorlesungsskript*. Hochschule Karlsruhe (siehe S. 33, 34).
- [43] Xu, Y. ; 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 (siehe S. 9, 83).

Abbildungsverzeichnis

2.3 Einrad mit Gyroskopen von James Bruton [19]	5
2.4 <i>One-Wheeled Vehicle</i> von Charles F. Taylor [22]	5
3.1 Freischnitt Nicken [33, S.21]	8
3.2 Freischnitt Kreisel als inverses Pendel [33, S.16]	9
3.3 Freischnitt Rollen [33, S.28]	10
3.4 Freischnitt Gieren [33, S.28]	11
4.1 Wheel Dynamixel PT1 Fit 10% Sollwertsprung, $K_P = 800, K_D = 50, K_I = 0$	14
4.2 Wheel Dynamixel PT1 Fit 10% Sollwertsprung, $K_P = 800, K_D = 50, K_I = 1000$	15
4.3 Kreisel Dynamixel PT1 Fit 3% Sollwertsprung, $K_P = 100, K_I = 1980$	17
4.4 Kreisel Dynamixel PT1 Fit 3% Sollwertsprung, $K_P = 500, K_I = 5000$	18
4.5 Kreisel Dynamixel PT1 Fit 10% Sollwertsprung, $K_P = 500, K_I = 5000$	19
4.6 Kreisel Dynamixel PT1 Fit 10% Sollwertsprung, $K_P = 300, K_I = 3000$	19
4.7 Hall-Sensor Signale Antriebsrad unter Last, unmodifiziert	21
4.8 Hall-Sensor Signale Antriebsrad unter Last, getrennte Kabelführung und Impedanzanpassung	22
4.9 Funktionsstruktur Elektronik	25
4.10 High-Side Switch Schaltung Spannungsversorgung 12 V	27
4.11 Schaltung zur Konvertierung von TTL-RS485-Protokoll	28
4.12 Platine Layout	29
4.13 Monowheeler Aufbau	30
5.1 Bestimmung des Nickwinkels anhand der Erdbeschleunigung	32
5.2 Bestimmung des Rollwinkels anhand der Erdbeschleunigung	32
5.3 Struktur Komplementärfilter [42, S. 84]	34
5.4 IMU Daten Raw	38
5.5 IMU Daten Raw FFT	39
5.6 IMU Daten mit IMU-DLP mit $f_G = 11 \text{ Hz}$	41
5.7 IMU Daten mit IMU-DLP mit $f_G = 11 \text{ Hz}$ FFT	42
5.8 Bode-Diagramm Butterworth-Tiefpass 2. Ordnung, $f_G = 40 \text{ Hz}$	43
5.9 IMU Daten mit IMU-DLP mit $f_G = 5 \text{ Hz}$ und Butterworth-Tiefpass mit $f_G = 40 \text{ Hz}$	44
5.10 IMU Daten mit IMU-DLP mit $f_G = 5 \text{ Hz}$ und Butterworth-Tiefpass mit $f_G = 40 \text{ Hz}$ FFT	45
5.11 Validierung Sensorkonzept und Signalverarbeitung	47
6.1 Pitch Experiment PID-Regler mit $K_P = 0,3; K_D = 0,045; K_I = 0,3$	51
6.2 Pitch Experiment PID-Regler mit $K_P = 0,3; K_D = 0,1; K_I = 0,3$	52
6.3 Pitch Experiment PID-Regler mit $K_P = 0,3; K_D = 0,2; K_I = 0,3$	53
6.4 Pitch Experiment PID-Regler mit $K_P = 0,3; K_D = 0,1; K_I = 0,3$ mit Gain-Scheduling für K_D	54

6.5	Nicken Zustandsregler Simulation	60
6.6	Nicken Zustandsregler	61
6.7	FFT Reglerausgang	62
6.8	FFT Reglerausgang mit Notchfilter	62
6.9	Gain-Scheduling mit unterschiedlicher Gewichtung des Fehlerintegrals	64
6.10	Nicken Zustandsregler mit Gain-Scheduling	65
6.11	Rollen Simulation mit $KP = 12$; $KD = 1,44$; $KI = 10,62$	67
6.12	Rollen Simulation mit $KP = 14$; $KD = 2,8$; $KI = 1,4$	68
6.13	Rollen Simulation mit $KP = 15$; $KD = 3$; $KI = 3$	69
6.14	Rollen Simulation mit $KP = 15$; $KD = 3$; $KI = 3$ mit fester Sollwertvorgabe	70
6.15	Struktur kaskadierter PID-Regler zum Balancieren	70
6.16	Rollen Simulation mit $KP = 15$; $KD = 3$; $KI = 3$ mit dynamischer Sollwertvorgabe	71
6.17	Rollen Experiment mit $KP = 15$; $KD = 3$; $KI = 3$ mit fester Sollwertvorgabe	72
6.18	Rollen Experiment mit $KP = 15$; $KD = 3$; $KI = 3$ mit dynamischer Sollwertvorgabe	73
6.19	Rollen Zustandsregler ohne Modellierung des Aktors	76
6.20	Rollen Zustandsregler mit Modellierung des Aktors	78
6.21	Rollen Zustandsregler mit Modellierung des Aktors und Trajektorie	79
6.22	Systemtest Fahrzeug Balancieren	81
6.23	Systemtest Fahrzeug Linie fahren	82
7.1	Simulation Fahrzeug Kurvenfahren durch passive Trajektorie	87
7.2	Systemtest Fahrzeug Kurvenfahren mit fester Roll-Trajektorie	89
7.3	Struktur kaskadierter PID-Regler zum Kurvenfahren	90
7.4	Simulation Fahrzeug Kurvenfahren durch aktive Trajektorie	90
7.5	Systemtest Fahrzeug Kurvenfahrt mit reglerbasierter Roll-Trajektorie	91
7.6	Systemtest Fahrzeug enge Linkskurve mit reglerbasierter Roll-Trajektorie	92
7.7	Systemtest Fahrzeug Slalom mit reglerbasierter Roll-Trajektorie	93
8.1	Übersicht Software	95
8.2	Architektur der Targetanwendung	98
8.3	Klassendiagramm der Targetanwendung (vereinfacht)	100
8.4	Histogramm Jitter Test Low-Latency-Kernel mit relativer Zeitmessung	102
8.5	Histogramm Jitter Test Realtime-Kernel mit relativer Zeitmessung	103
8.6	Histogramm Jitter Test Realtime-Kernel mit absoluter Zeitmessung	103
8.7	Klassendiagramm von <i>CStateEstimation</i> (vereinfacht)	105
8.8	Klassendiagramm von <i>CController</i> (vereinfacht)	107
8.9	Klassendiagramm von <i>CHardware</i> (vereinfacht)	113
8.10	SPI Kommunikation <i>ICM20948</i> [15, S. 31]	116
B.1	IMU Daten mit IMU-DLP mit $f_G = 5$ Hz	7
B.2	IMU Daten mit IMU-DLP mit $f_G = 5$ Hz FFT	8
B.3	IMU Daten mit IMU-DLP mit $f_G = 23$ Hz	9
B.4	IMU Daten mit IMU-DLP mit $f_G = 23$ Hz FFT	10
B.5	IMU Daten mit IMU-DLP mit $f_G = 11$ Hz und Butterworth-Tiefpass mit $f_G = 40$ Hz	11

B.6 IMU Daten mit IMU-DLP mit $f_G = 11 \text{ Hz}$ und Butterworth-Tiefpass mit $f_G = 40 \text{ Hz}$ FFT	12
---	----

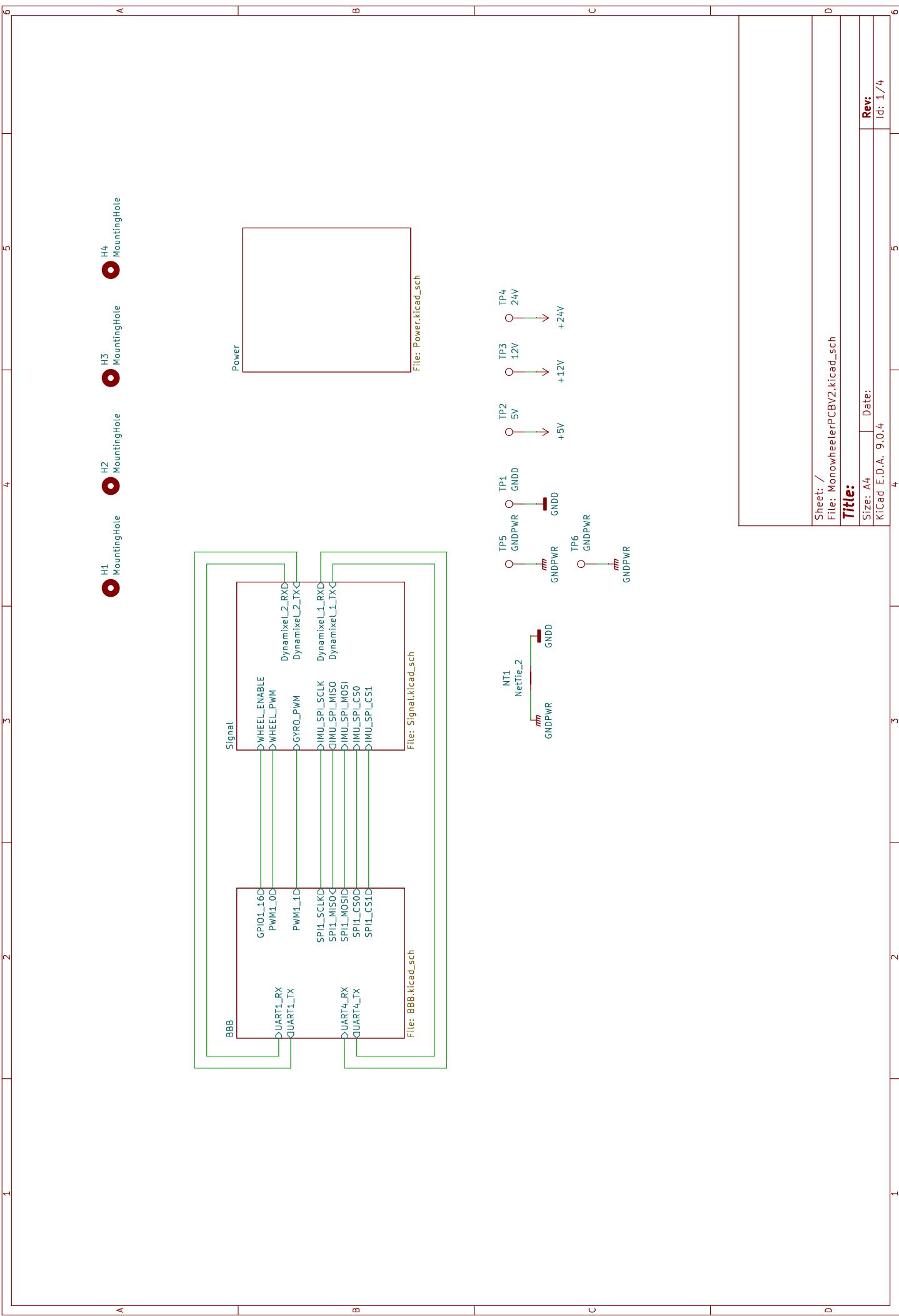
Tabellenverzeichnis

4.1	PT1-Fit Wheel Dynamixel mit $K_P = 800, K_D = 50, K_I = 0$	15
4.2	PT1-Fit Wheel Dynamixel mit $K_P = 800, K_D = 50, K_I = 1000$	16
4.3	PT1-Fit Gyro Dynamixel mit $K_P = 100, K_I = 1980$	17
4.4	PT1-Fit Gyro Dynamixel mit $K_P = 500, K_I = 5000$	18
4.5	PT1-Fit Gyro Dynamixel mit $K_P = 300, K_I = 3000$	20
4.6	Übersicht elektrischer Komponenten und Anschlüsse	24
4.7	Pinbelegung BBB	24
5.1	Übersicht der SNR-Werte der IMU Daten	46
6.1	Maximal zulässige Werte der Zustände des Teilsystems Nicken	59
6.2	Maximal zulässige Werte der Zustände des Teilsystems Rollen	75

Quelltextverzeichnis

1	FFT Tiefpassfilter	36
2	SNR Bestimmung in Python	37
3	FFT Analyse in Python	39
4	Nicken Gain-Scheduling	54
5	Bestimmung der Steuerbarkeit	58
6	Diskreter LQR-Entwurf in Python	58
7	Rollen Trajektorie zum Aufrichten	79
8	Automatisierung der Entwicklungsumgebung	96
9	Ausschnitt Implementierung Timer	104
10	Ausschnitt Implementierung Zustandsschätzung	105
11	Ausschnitt Implementierung IIR-Filter zweiter Ordnung	106
12	Ausschnitt Implementierung IIR-Filter als SOS-Filter	106
13	Ausschnitt Implementierung PID-Regler	107
14	Ausschnitt Implementierung des Zustandsreglers Nicken	108
15	Ausschnitt Implementierung der Zustandsübergänge Nicken	109
16	Ausschnitt Implementierung der Interpolation Nicken	109
17	Ausschnitt Implementierung der Regler der physikalischen Größen in der Funktion <i>update()</i> aus <i>CController</i>	110
18	Ausschnitt Implementierung der Rampe für sanftes Anfahren	110
19	Ausschnitt Implementierung der Trajektoriengenerierung in der Funktion <i>update()</i> aus <i>CController</i>	111
20	Ausschnitt Implementierung des Interface-Managers	115
21	Ausschnitt Implementierung der Funktion <i>burstRead()</i> aus <i>CICM20948</i> . . .	117
22	Ausschnitt Implementierung der Funktion <i>writeReg()</i> aus <i>CICM20948</i> . . .	118
23	Ausschnitt Implementierung der Funktion <i>readImu()</i> aus <i>CICM20948</i> . . .	119
24	Ausschnitt Implementierung der Klasse <i>CMotorPWM</i>	120
25	Ausschnitt Implementierung der Funktion <i>readFifo()</i> aus der Klasse <i>CAD-CMMAP</i>	121
26	Ausschnitt Implementierung der Funktion <i>readADC()</i> aus der Klasse <i>CAD-CMMAP</i>	122
27	Ausschnitt Implementierung der Funktion <i>run()</i> aus <i>CCommComp</i>	123
28	Ausschnitt Implementierung der Funktion <i>init()</i> aus <i>CServer</i>	124
29	Ausschnitt Implementierung der Funktion <i>waitForClient()</i> aus <i>CServer</i> . . .	125
30	Ausschnitt Implementierung der Funktion <i>transmitMessage()</i> aus <i>CServer</i> . .	125
31	Ausschnitt Implementierung des Hintergrundthreads der Python-Gui	126
32	Ausschnitt Implementierung des Timer-Callbacks der Python-Gui	127
33	Ausschnitt Implementierung des <i>send_inital_message()</i> -Funktion der Python-Gui	127
34	Ausschnitt Implementierung des <i>recv_message()</i> -Funktion der Python-Gui . .	128
35	Ausschnitt Implementierung der <i>CContent</i> -Klasse der Python-Gui	128
36	Ausschnitt Implementierung des <i>record()</i> -Funktion der Python-Gui	129

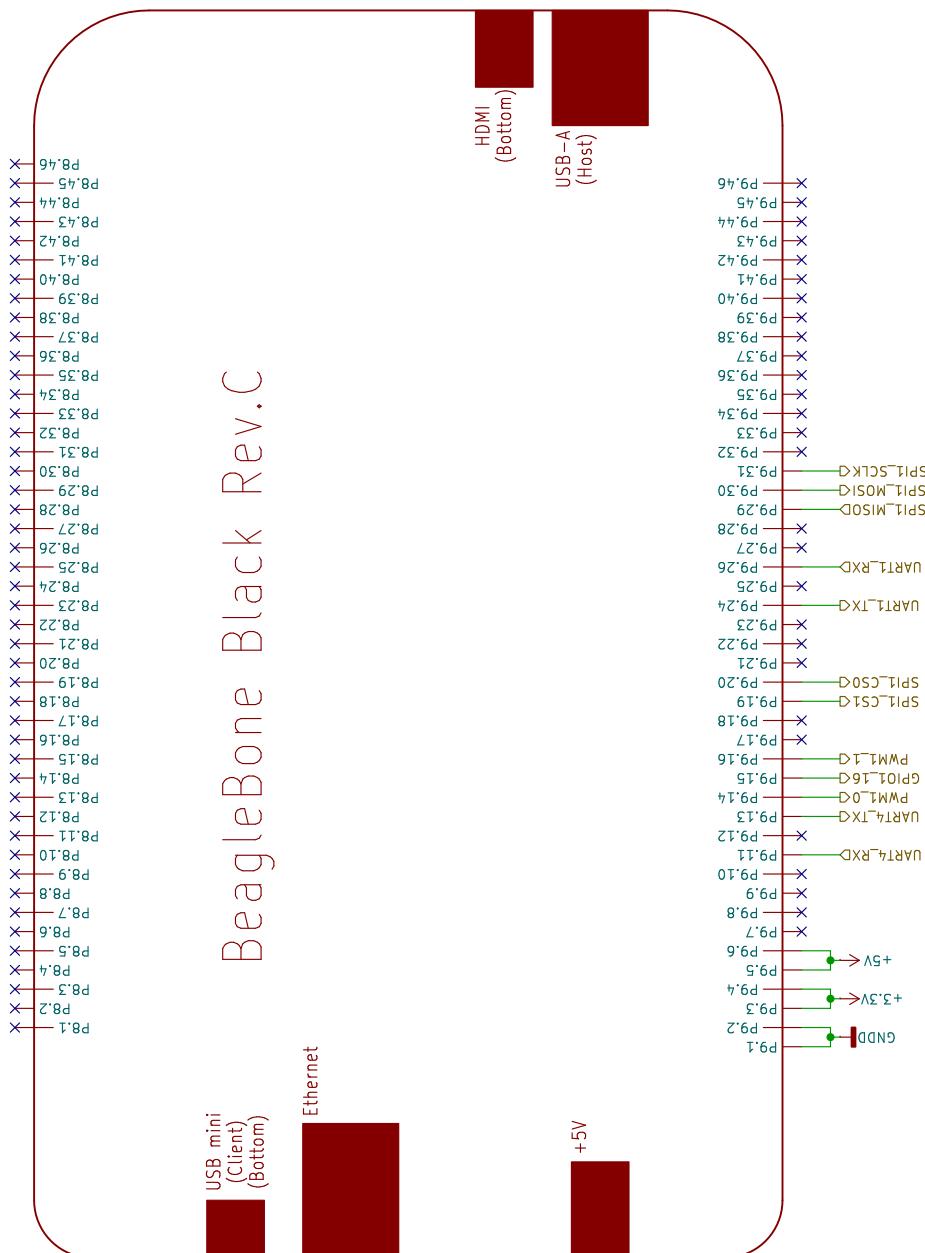
A Schaltplan

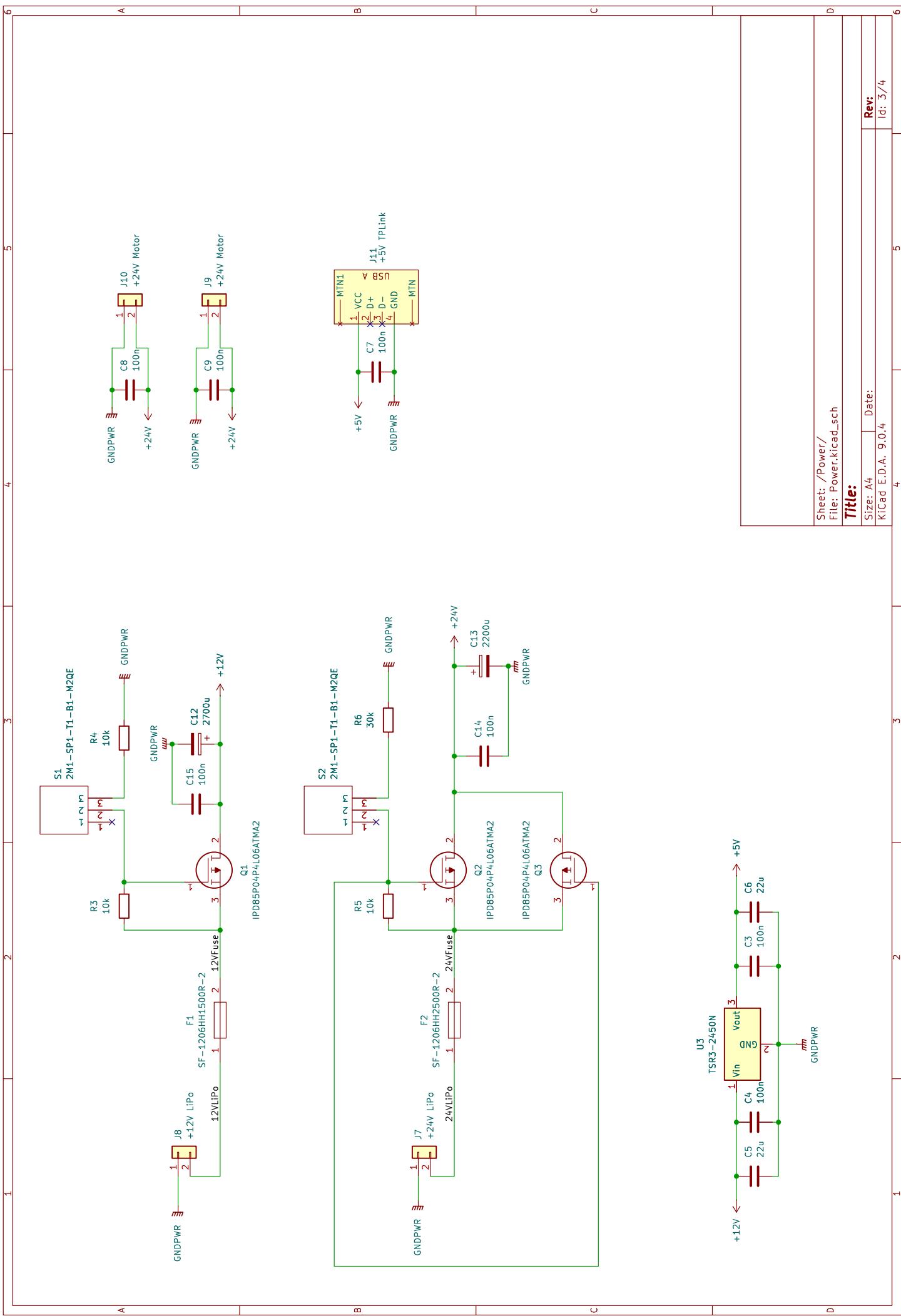


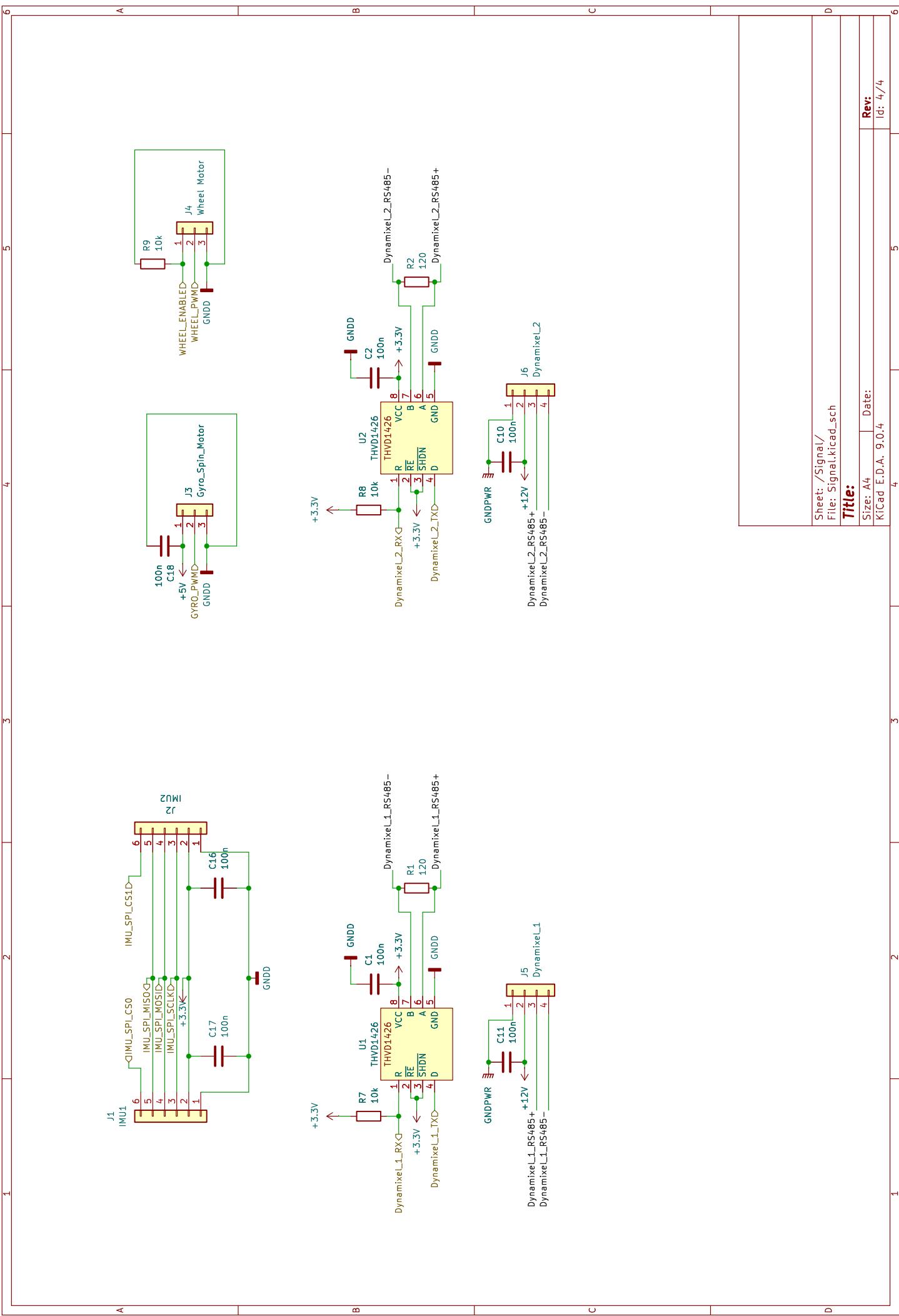
Sheet: /BBB/
File: BBB.kicad_sch
Title:
Size: A4
KiCad E.D.A. 9.0.4

Rev:
Id: 2/4

BeagleBone Black Rev.C







B Signalanalyse

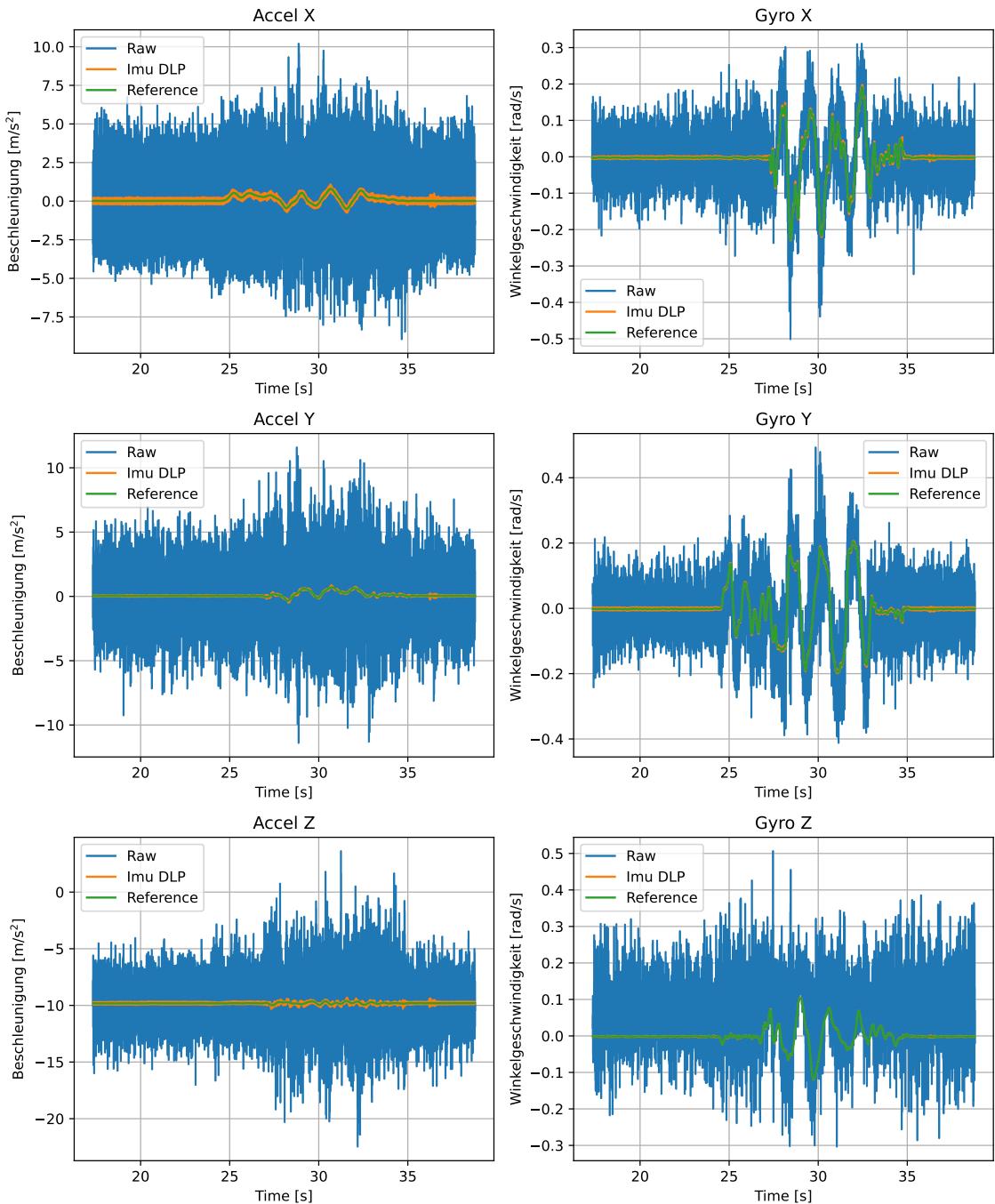


Abbildung B.1: IMU Daten mit IMU-DLP mit $f_G = 5$ Hz

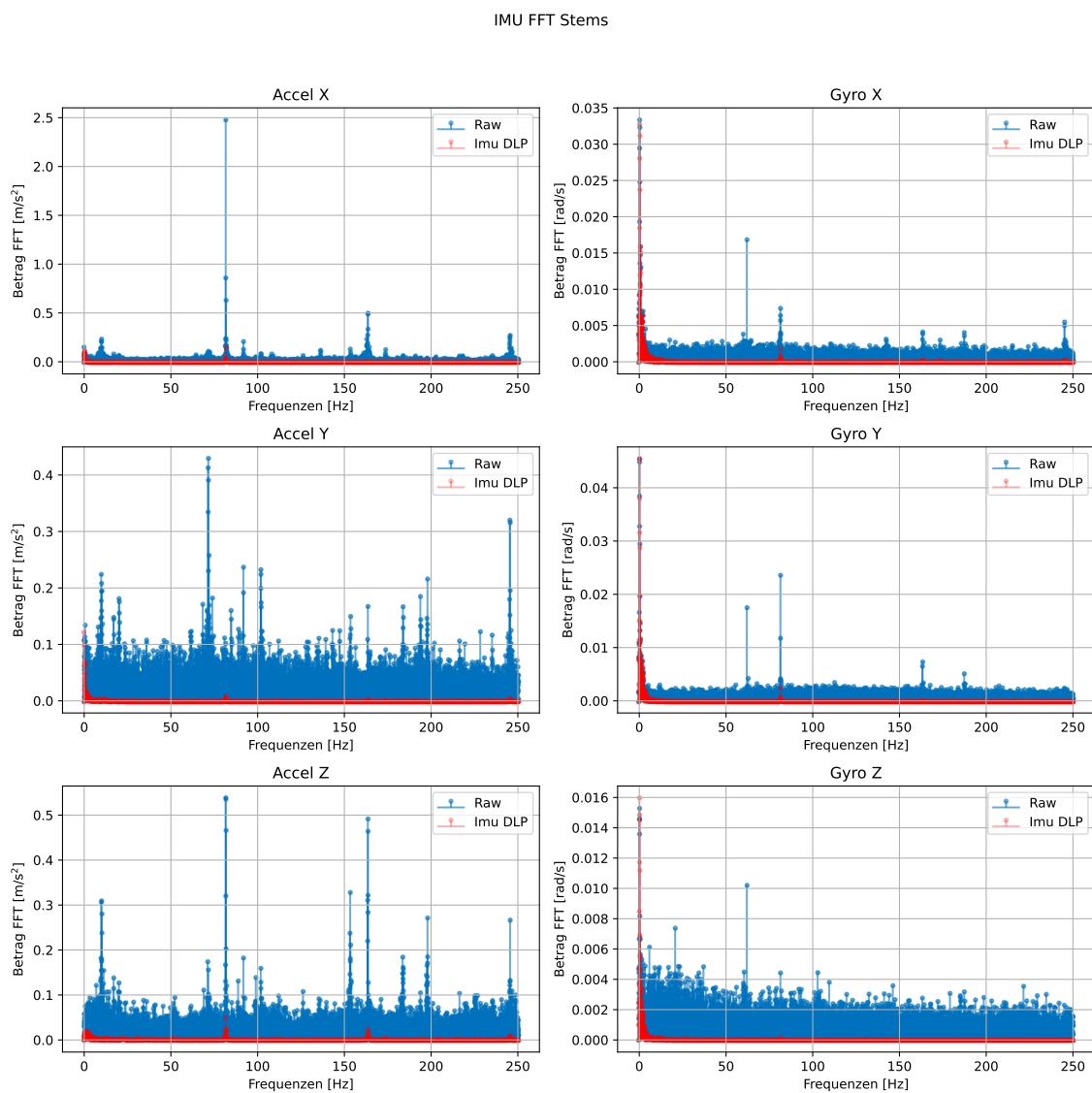


Abbildung B.2: IMU Daten mit IMU-DLP mit $f_G = 5 \text{ Hz}$ FFT

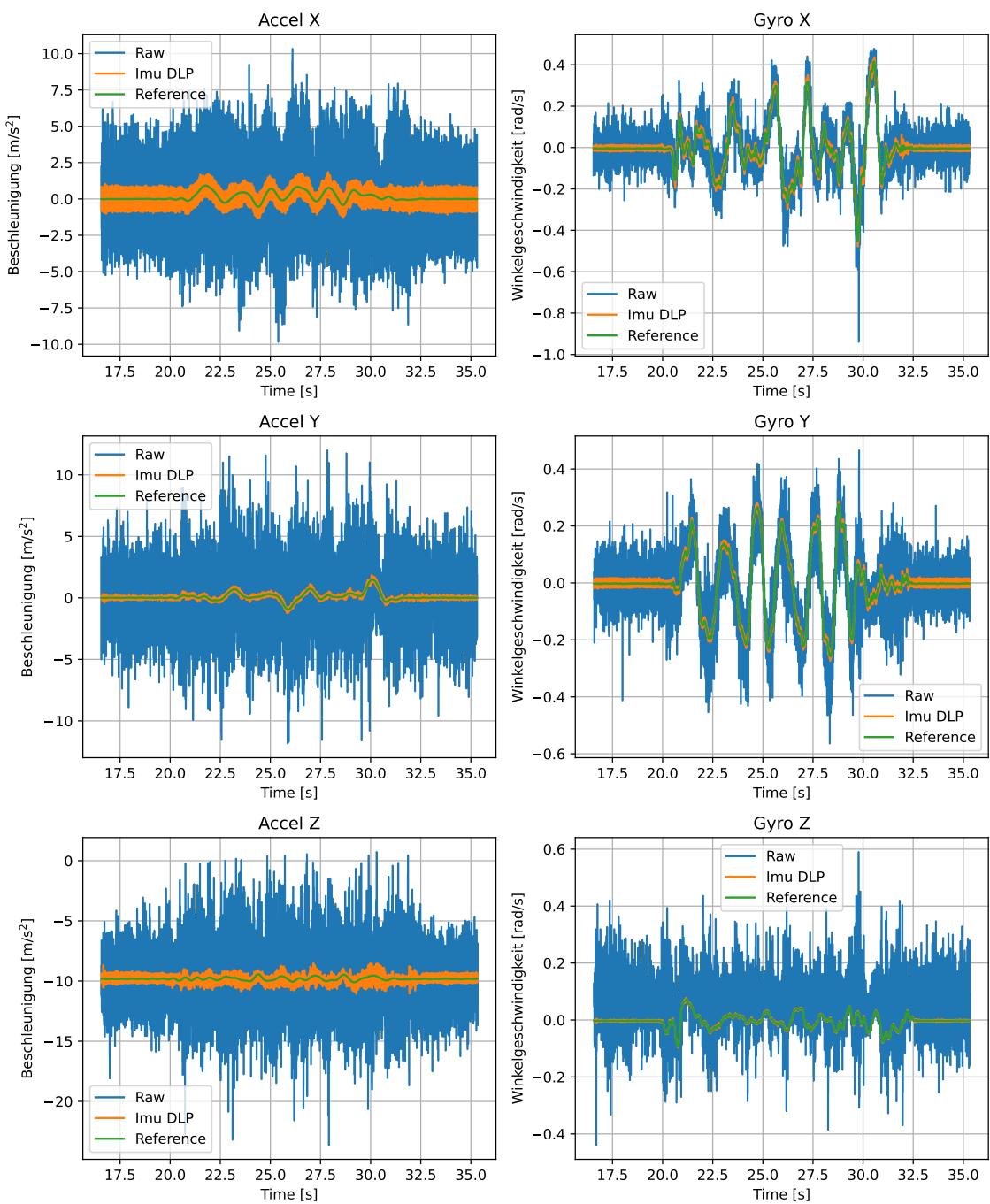


Abbildung B.3: IMU Daten mit IMU-DLP mit $f_G = 23$ Hz

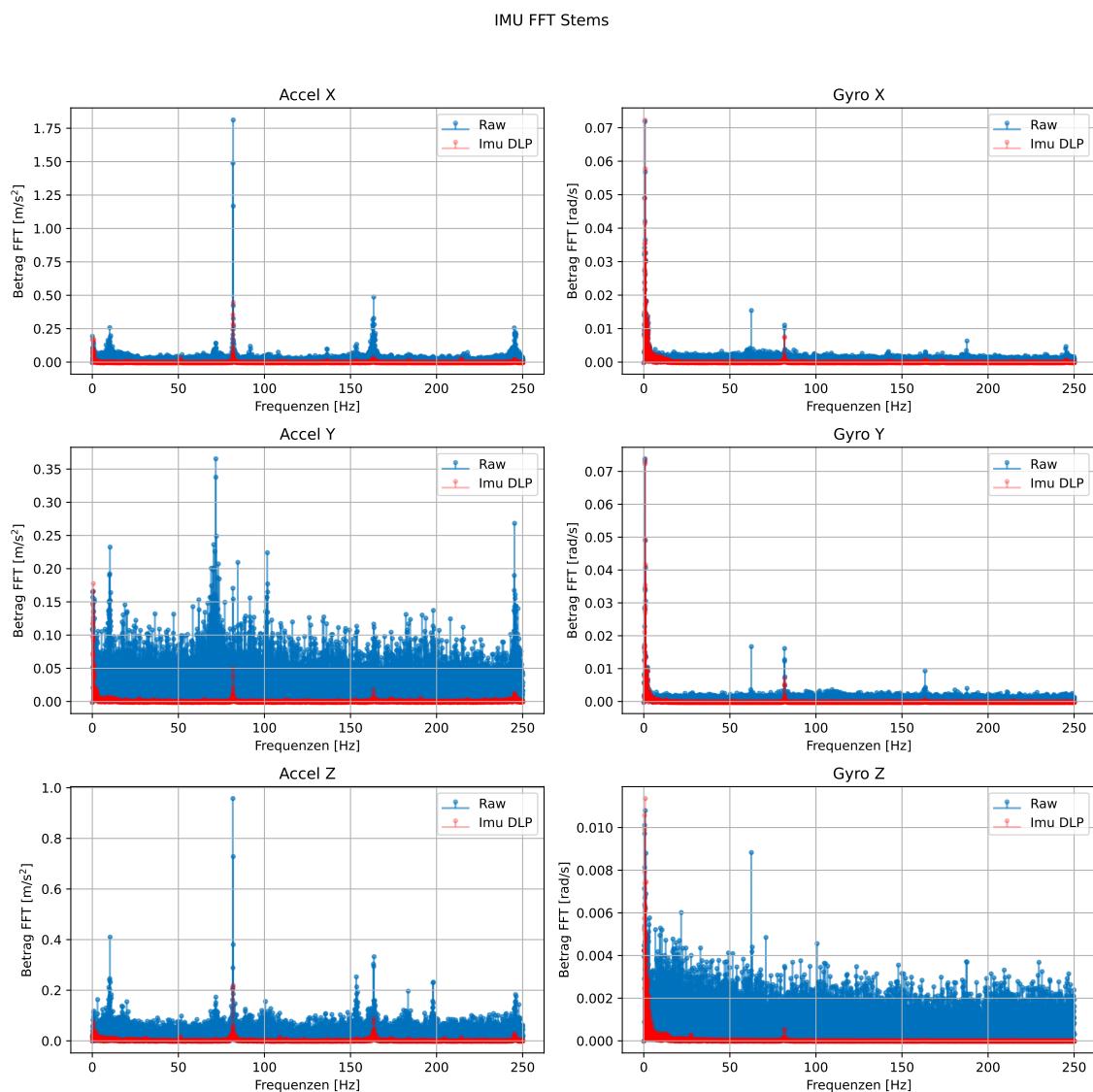


Abbildung B.4: IMU Daten mit IMU-DLP mit $f_G = 23 \text{ Hz}$ FFT

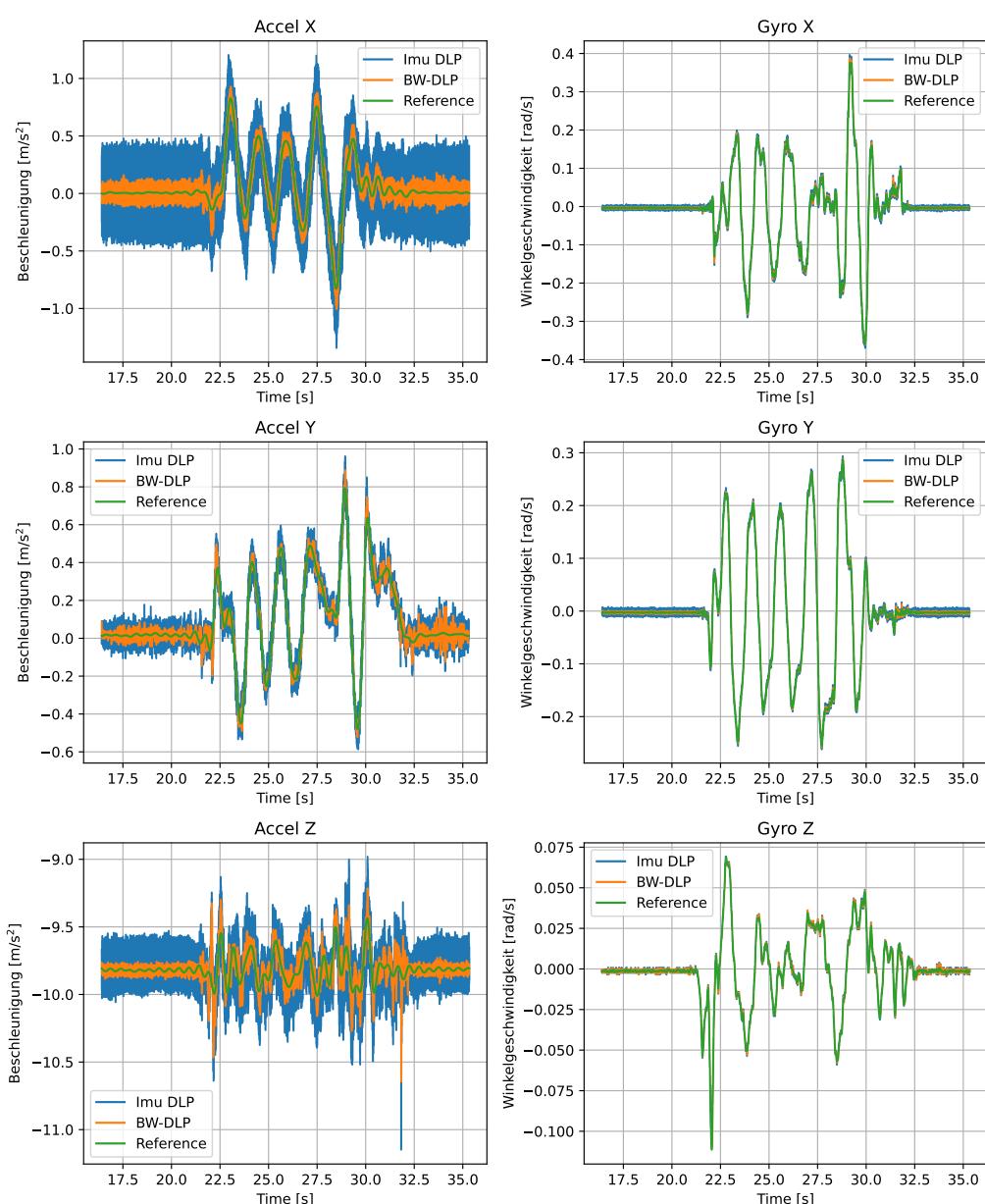


Abbildung B.5: IMU Daten mit IMU-DLP mit $f_G = 11 \text{ Hz}$ und Butterworth-Tiefpass mit $f_G = 40 \text{ Hz}$

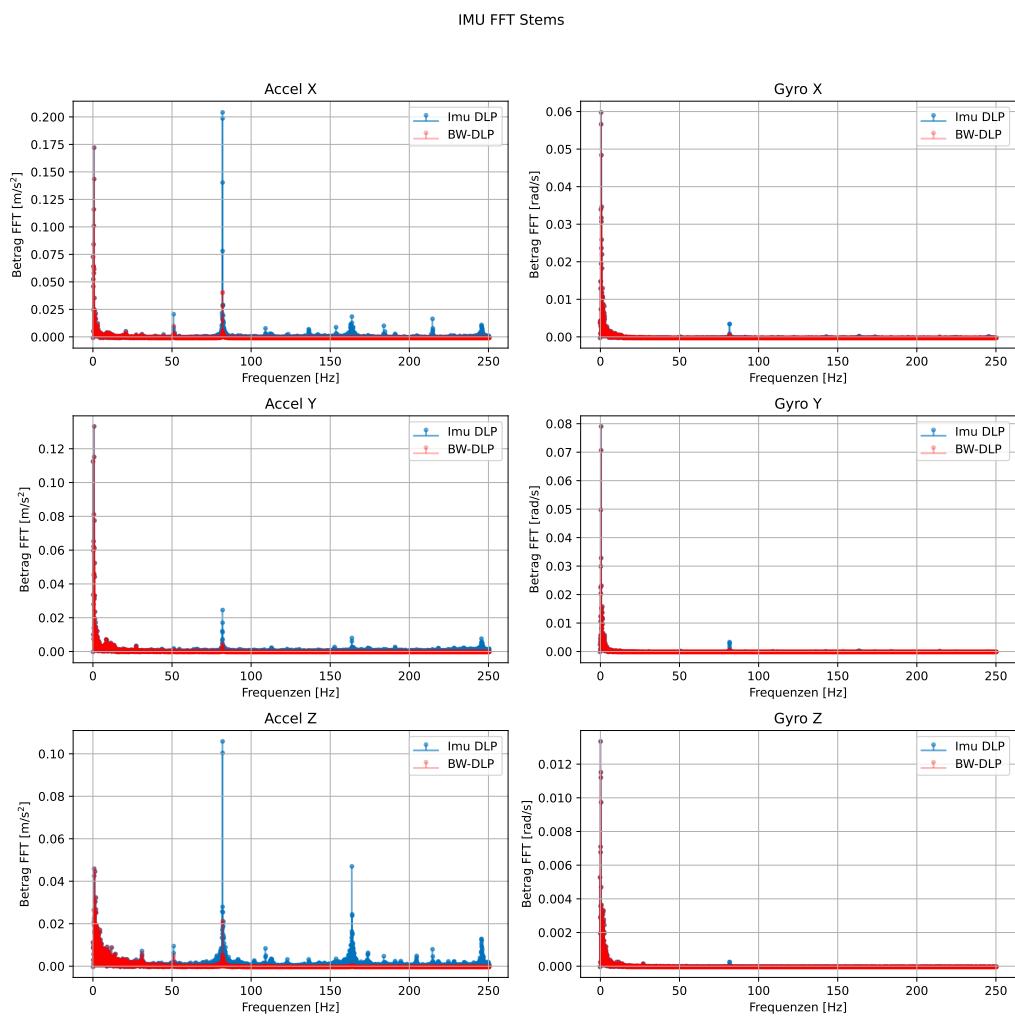


Abbildung B.6: IMU Daten mit IMU-DLP mit $f_G = 11$ Hz und Butterworth-Tiefpass mit $f_G = 40$ Hz FFT

C Code

C.1 CPitchController

```

1 double CPitchController::control(double pTheta, double pDotTheta, double
2   ↵  pDynamixel)
3 {
4     switch (mMode)
5     {
6         case Mode::Slow:
7             if (std::abs(pTheta) > mThetaUpperThreshold || std::abs(pDotTheta) >
8                 ↵  mDotThetaUpperThreshold) {
9                 mMode = Mode::TransitionSlow2Fast;
10                mTTransition = 0;
11            }
12            break;
13         case Mode::Fast:
14             if (std::abs(pTheta) < mThetaLowerThreshold && std::abs(pDotTheta) <
15                 ↵  mDotThetaLowerThreshold) {
16                 mTSlow += mTa;
17                 if (mTSlow > mTSlowThreshold) {
18                     mMode = Mode::TransitionFast2Slow;
19                     mTTransition = 0;
20                     mTSlow = 0;
21                 }
22                 break;
23         case Mode::TransitionSlow2Fast:
24             if (mTTransition > mTTransitionMax) {
25                 mMode = Mode::Fast;
26             }
27             break;
28         case Mode::TransitionFast2Slow:
29             if (mTTransition > mTTransitionMax) {
30                 mMode = Mode::Slow;
31             }
32             break;
33     }
34     mThetaSum += - pTheta*mTa;
35
36     double uCMDFast;
37     double uCMDSlow;
38     double blend;
39
40

```

A14 | C Code

```
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);
56         uCMDSlow = -std::clamp((mLQRSlow[0]*pTheta
57                             + mLQRSlow[1]*pDotTheta
58                             + mLQRSlow[2]*mThetaSum
59                             + mLQRSlow[3]*pDynamixel), -mUCMDMax, mUCMDMax);
60         blend = std::clamp(mTTransition/mTTransitionMax, 0.0, 1.0);
61         mUCMD = blend*uCMDFast + (1-blend)*uCMDSlow;
62         mTTransition += mTa;
63         break;
64     case Mode::TransitionFast2Slow:
65         uCMDFast = -std::clamp((mLQRFast[0]*pTheta
66                             + mLQRFast[1]*pDotTheta
67                             + mLQRFast[2]*mThetaSum
68                             + mLQRFast[3]*pDynamixel), -mUCMDMax, mUCMDMax);
69         uCMDSlow = -std::clamp((mLQRSlow[0]*pTheta
70                             + mLQRSlow[1]*pDotTheta
71                             + mLQRSlow[2]*mThetaSum
72                             + mLQRSlow[3]*pDynamixel), -mUCMDMax, mUCMDMax);
73         blend = std::clamp(mTTransition/mTTransitionMax, 0.0, 1.0);
74         mUCMD = blend*uCMDSlow + (1-blend)*uCMDFast;
75         mTTransition += mTa;
76         break;
77     default:
78         mUCMD = 0;
79         break;
80     }
81     return mUCMD;
82 }
```

C.2 CPIController

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

```

4     mDotE = (mE -mEPrev) / mPID.Ta;
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) ,
9                       -mPID.UMax, mPID.UMax);
9     mEPrev = mE;
10    return mU;
11 }
12
13 double CPIDController::control(const double pW, const double pDotW, const double
14                                ↪ pX, const double pDotX)
14 {
15     mE = pW - pX;
16     mDotE = pDotW - pDotX;
17     if (std::abs(mU) < mPID.UMax) {
18         mSumE += mE*mPID.Ta;
19     }
20     mU = std::clamp(mDirection*(mPID.KP*mE + mPID.KD*mDotE + mPID.KI*mSumE) ,
21                      -mPID.UMax, mPID.UMax);
21     mEPrev = mE;
22     return mU;
23 }
```

C.3 CController

```

1 void CController::update(CCommand& pCmd, CStateData& pStateData, CMotorData&
2   ↪ pMotorData)
3 {
4     double phiTargetNew, dotPhiTarget;
5     switch (mMode)
6     {
7         case EMode::Balance:
8             phiTargetNew = mRollTargetBalanceController.control(0, 0,
9                           ↪ pMotorData.mGyroDynamixelPosition, pMotorData.mGyroDynamixelVelocity);
10            dotPhiTarget = (phiTargetNew - pStateData.mPhiTarget)/mTa;
11
12            pStateData.mPhiTarget = phiTargetNew;
13            pStateData.mSpeedTarget = ramp(mTargetVelocity, pStateData.mSpeedTarget,
14                                         ↪ Monowheeler::MAX_ACCEL);
15            pStateData.mDotPsiTarget = mDotPsiTarget;
16            pStateData.mThetaTarget = 0;
17
18            CCommand::ECommands cmd;
19            double cmdData;
20            if (pCmd.getData(cmd, cmdData)) {
21                switch (cmd)
22                {
23                    case CCommand::ECommands::Drive:
24                        mTargetVelocity = std::clamp(cmdData, -Monowheeler::MAX_SPEED,
25                            ↪ Monowheeler::MAX_SPEED);
26                        mDotPsiTarget = 0.0;
```

A16 | C Code

```
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) <
47     ↵ 0.05) {
48     mCornerAcceleration = false;
49     mMode = EMode::Corner;
50     if (mCornerRight) {
51         mDotPsiTarget = -0.15;
52     }
53     else{
54         mDotPsiTarget = 0.15;
55     }
56     break;
57
58 case EMode::Corner:
59     pStateData.mDotPsiTarget = mDotPsiTarget;
60     dotPhiTarget =
61         ↵ mRollTargeYawController.control(pStateData.mDotPsiTarget,
62         ↵ pStateData.mDotPsi);
63     pStateData.mPhiTarget += dotPhiTarget*mTa;
64     pStateData.mSpeedTarget = ramp(mTargetVelocity,
65         ↵ pStateData.mSpeedTarget, Monowheeler::MAX_ACCEL);
66     pStateData.mThetaTarget = 0;
67
68     mCounter++;
69     if (mCounter > 500) {
70         if (mSlalom) {
71             mSlalom = false;
72             mDotPsiTarget = 0.15;
73         }
74         else {
75             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,
85     ↪ pStateData.mDotTheta, pMotorData.mWheelDynamixelPosition);
86
87 pMotorData.mGyroDynamixelTarget =
88     ↪ mRollController.control(pStateData.mPhiTarget, dotPhiTarget,
89     ↪ pStateData.mPhi, pStateData.mDotPhi);
90
91 pMotorData.mWheelTorqueTarget =
92     ↪ mSpeedController.control(pStateData.mSpeedTarget, pStateData.mSpeed);
93 }
```