Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Bachelorarbeit

Ole Schönherr

Entwicklung und Erprobung einer Multi-Roboter-Steuerung für den Aufbau eines Roboterschwarms

Fakultät Technik und Informatik
Department Maschinenbau und Produktion

Faculty of Engineering and Computer Science
Department of Mechanical Engineering and Production Management

Ole Schönherr

Entwicklung und Erprobung einer Multi-Roboter-Steuerung für den Aufbau eines Roboterschwarms

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung

im Studiengang Maschinenbau / Entwicklung und Konstruktion
am Department Maschinenbau und Produktion
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg.

Erstprüfer/in: Prof. Dr. Stephan Schulz
Zweitprüfer/in: Prof. Dr.-Ing. Hans-Joachim Schelberg

Abgabedatum: 13.08.2020

Ole Schönherr

Thema der Arbeit

Entwicklung und Erprobung einer Multi-Roboter-Steuerung für den Aufbau eines Roboterschwarms

Stichworte

Kamera, ArUco, ESP32, Arduino, Regelungstechnik, Autonomes Laden, HOOU, PIA, Vorsteuerung, Roboterschwarm, ISP-Programmierung, Fernwartung

Kurzzusammenfassung

Diese Arbeit befasst sich mit der Entwicklung eines zuverlässig und präzise fahrenden PIA Roboters und dessen Ansteuerung. Im Zuge dessen wird das bestehende System analysiert und ein verbessertes Systemdesign, neue Schaltpläne und eine neues PCB-Design für den Einsatz des neuen Microcontrollers ESP32 und der beiden Motorcontroller vom Typ Baby Orangutan erstellt.

Um mehrere Roboter komfortabel auf einem einheitlichen Firmware-Stand zu halten, werden Over-the-Air-Funktionalitäten für den ESP32 sowie für die beiden Motorcontroller umgesetzt.

Die Softwareänderungen, welche bedingt durch die neue Hardware benötigt werden, werden erläutert und Verbesserungen der Bestandsfunktionen vorgestellt. Neben einer Vorsteuerung für Motordrehzahlen wird auch ein weiterer Regelkreis implementiert, welcher die Roboterorientierung mittels einer inertialen Messeinheit stabilisiert.

Die bestehenden Protokolle werden dahingehend angepasst, dass mehrere Roboter individuell angesteuert werden können. Dies bildet die Grundlage zum Aufbau eines Roboterschwarms.

Abschließend wird ein Algorithmus vorgestellt, welcher mittels ArUco-Marker einen anderen Roboter identifizieren, um anschließend an ihm anzudocken. Die so verbunden Roboter können eine Formationsfahrt vollziehen, ohne sich relativ zueinander zu bewegen.

Ole Schönherr

Title of Thesis

Development and testing of a multi-robot controller for the construction of a robot swarm

Keywords

camera, ArUco, ESP32, Arduino, control engineering, autonomous loading, HOOU, PIA, feedforward, robot swarm, ISP programming, remote maintenance

Abstract

This thesis deals with the development of a reliable and precise driving PIA robot and its control. In the course of this work, the existing system is analysed and an improved system design, new circuit diagrams and a new PCB design for the use of the new microcontroller ESP32 and the two motor controllers of the Baby Orangutan type are created.

In order to keep several robots comfortably on a uniform firmware level, over-the-air functionalities are implemented for the ESP32 and the two motor controllers.

The software changes required due to the new hardware are explained and improvements of the existing functions are presented. In addition to a feedforward control for engine speeds, another control loop will be implemented, which stabilizes the robot orientation by means of an inertial measurement unit.

The existing protocols will be adapted to allow individual control of several robots. This forms the basis for building a robot swarm.

Finally, an algorithm is presented which uses ArUco markers to identify another robot in order to dock with it. The robots connected in this way can perform a formation drive without moving relative to each other.

Inhaltsverzeichnis

Formelzeichen und Abkürzungen

A	Kanal A eines Quadraturencoders
a	Mittlerer Abstand zum theoretischen Zielpunkt
a_0	Parameter der einfachen Vorsteuerung
$a_{0,x}$	Parameter der mehrfachen Vorsteuerung für Fahrtrichtung x
$a_{0,y}$	Parameter der mehrfachen Vorsteuerung für Fahrtrichtung y
$a_{0,yaw}$	Parameter der mehrfachen Vorsteuerung für yaw-Drehung
a_1	Parameter der einfachen Vorsteuerung
$a_{1,x}$	Parameter der mehrfachen Vorsteuerung für Fahrtrichtung x
$a_{1,y}$	Parameter der mehrfachen Vorsteuerung für Fahrtrichtung y
$a_{1,yaw}$	Parameter der mehrfachen Vorsteuerung für yaw-Drehung

Akku	Akkumulator
a_n	Abstand zum theoretischen Zielpunkt
AR	Augmented Reality
ARM	Advanced RISC Machines
ArUco	Augmented Reality University of Córdoba
AVR	8-Bit-Mikrocontroller-Familie des Herstellers Atmel
bal_0	x-Geschwindigkeitsanteil im Intervall [0, 1]
bal_1	y-Geschwindigkeitsanteil vom y-, yaw-Anteil im Intervall [0, 1]
BEE	PIA6 Roboter ohne besondere Ausstattung
b_x	x-Geschwindigkeitsanteil im Intervall [0, 1]
b_y	y-Geschwindigkeitsanteil im Intervall [0, 1]
b_{yaw}	yaw-Geschwindigkeitsanteil im Intervall [0, 1]
CRC	cyclic redundancy check
dt	Zeitintervall
e_{AR}	Durch ArUco-Marker ermittelter Fehler
EE	Endeffektor
$Error_{x,n}$	Abweichung zwischen Soll- und Ist-Position in x-Richtung des Roboters n
$Error_{y,n}$	Abweichung zwischen Soll- und Ist-Position in y-Richtung des Roboters n
$Error_{yaw,n}$	Verdrehung zwischen Soll- und Ist-Position des Roboters n
ESP32	ESP32 SoC der Firma espressif
$e(t)$	Regelabweichung
e_{VR}	Durch VR-Tracker ermittelter Fehler
f	Frequenz
$Fit_{lin}(S)$	Fit einer linearen Funktion durch verschiedene Streckenpunkte
FOTA	Firmware-Over-the-Air
$f_{PID,PIA004}$	PID-Update-Frequenz PIA004 Roboter
$f_{refresh}$	Aktualisierungsfrequenz
f_{Update}	PID-Update-Frequenz
GPIO	general purpose input/output
HIVE	PIA6 Roboter mit verbauter Ladestation
HOOU	Hamburg Open Online University
IMU	Inertial Measurement Unit
ISP	In-System-Programmer
K_D	Verstärkungsfaktor des D-Glieds
K_I	Verstärkungsfaktor des I-Glieds

KiCAD	Software zu Erstellung von Schaltplänen und PCB-Designs
K_P	Verstärkungsfaktor des P-Glieds
$K_{VS}(w)$	Vorsteuerungsfunktion
l	Länge
LiDAR	Light Detection and Ranging
LiPo	Lithium-Polymere
MISO	Master In Slave Out
MOSI	Master Out Slave In
MVS	Motor Vorsteuerung
$MVS_{balance0}$	x-Geschwindigkeitsanteil im Intervall [0, 255]
$MVS_{balance1}$	y-Geschwindigkeitsanteil vom y-, yaw-Anteil im Intervall [0, 255]
MVS_{sum0}	Summe aller Robotergeschwindigkeiten
MVS_{sum1}	Summe der Robotergeschwindigkeiten in x- und y-Richtung
MVS_x	Motor Vorsteuerung für Fahrtrichtung x
MVS_y	Motor Vorsteuerung für Fahrtrichtung y
MVS_{yaw}	Motor Vorsteuerung für yaw-Drehung
N	Anzahl der Messungen
n_{enc}	Encoder Auflösung
$n_{getriebe}$	Übersetzungsverhältnis des Motorgetriebes
$Offset_{x,n}$	Verschiebung in x-Richtung vom EE-Koordinatensystem zum Roboterkoordinatensystem n
$Offset_{y,n}$	Verschiebung in y-Richtung vom EE-Koordinatensystem zum Roboterkoordinatensystem n
$Offset_{yaw,n}$	Verdrehung vom EE-Koordinatensystem zum Roboterkoordinatensystem n
ω	Drehzahl
ω_{Diff}	Drehzahldifferenz zwischen gesetzter- und Ist-Drehzahl
ω_{Ist}	Ist-Drehzahl
ω_{min}	Kleinste zu messende Drehzahl
ω_{Set}	Gesetzte Drehzahl
OpenCV	Open Computer Vision
OTA	Over-the-Air
$P_{comp,XY}$	Kompensationsproportionalitätsfaktor für x- und y-Komponente
$P_{comp,YAW}$	Kompensationsproportionalitätsfaktor für die Verdrehung
PIA	Platform for Intelligent Automation
$p_{Ist,n}$	Ist-Position der Messung n

p_m	Mittlerer erreichter Zielpunkt aus N Fahrten
$\overrightarrow{p_n}$	Messpunkt der Fahrt n
p_{Soll}	Zielposition
$p_{Soll,n}$	Soll-Position der Messung n
PWM_{PID}	Stellgröße des PID-Reglers
PWM_{Sum}	Stellgröße des Motorreglers
PWM_{VS}	Stellgröße der Vorsteuerung
QR	Quick Response
r_e	Fehlerradius
$\overrightarrow{r_{EE,n}}$	Verschiebungsvektor vom EE-Koordinatensystem ins Koordinatensystem des Roboters n
$r_{e,n}$	Fehlerradius der Fahrt n
ROS	Robot Operating System
R_s	Schutzwiderstand der Pegelwandlerschaltung
RST	Reset
S	Strecke
S_0	Strecke zum Zeitpunkt $t = 0$ s
SAMD	Microchip-Technology
SCK	Clock
$S_{Fit}(t)$	Durch die Messpunkte gefittete Funktion bezüglich der zum Zeitpunkt t zurückgelegten Strecke
S_{Ist}	Ist-Strecke
SoC	System on a Chip
S_{Soll}	Soll-Strecke
$S_{theo}(t)$	Theoretisch nach der Zeit t zurückgelegte Strecke
t	Zeit
t_{comp}	Kompensationszeitraum
$t_{digitalRead}$	Zeit zum Auslesen von 10^5 Pinzuständen mittels *digitalRead()*
$T_{EE,n}$	Transformationsmatrix vom EE-Koordinatensystem ins Koordinatensystem des Roboters n
t_n	Bestimmter Zeitpunkt
$t_{Resgister}$	Zeit zum Auslesen von 10^5 Pinzuständen mittels Portmanipulation
$t_{tot,0.05}$	Totzeit bei einer Geschwindigkeit von $v = 0{,}05\,\frac{\mathrm{m}}{\mathrm{s}}$
$u(t)$	Stellgröße
UART	Universal Asynchronous Receiver Transmitter
UDP	User Datagram Protocol

USB	Universal Serial Bus
B	Kabal B eines Quadraturencoders
v	Geschwindigkeit
$\dot{v}$	Anfahrbeschleunigung
$v(t_n)$	Geschwindigkeit zum Zeitpunkt t_n
V_{CC}	Voltage Common Collector
$\overrightarrow{v_{err,n}}$	Geschwindigkeit des Roboters n auf Grund seiner Positionsabweichung
$V_{ESP,0}$	Spannung einer logischen NULL der ESP-Microcontroller-Familie
$V_{ESP,1}$	Spannung einer logischen EINS der ESP-Microcontroller-Familie
$\overrightarrow{v_{ges,n}}$	Gesamtgeschwindigkeit des Roboters n
$\overrightarrow{v_{HIVE}}$	Geschwindigkeitsvektor eines HIVE Roboters
v_{ist}	Ist-Geschwindigkeit
V_{lH}	Spannung einer logischen EINS
V_{lL}	Spannung einer logischen NULL
$\overrightarrow{v_n}$	Geschwindigkeitsvektor eines Roboters n
v_{max}	Maximale Geschwindigkeit
VR	Virtual Reality
v_x	Geschwindigkeit in Roboterrichtung x
v_y	Geschwindigkeit in Roboterrichtung y
WLAN	Wireless Local Area Network
w_n	Führungsgröße des Motors n
$w(t)$	Führungsgröße
x_1	Charakteristischer Wert eins zu Bestimmung der Quadraturencoderdrehrichtung
x_2	Charakteristischer Wert zwei zu Bestimmung der Quadraturencoderdrehrichtung
$\dot{x_{EE}}$	Geschwindigkeit des EE-Koordinatensystems in x-Richtung
$\dot{x}$	Geschwindigkeit in x-Richtung
$x(t)$	Regelgröße
$ya\dot{w}$	Yaw-Drehgeschwindigkeit
$ya\dot{w}_{EE}$	Drehgeschwindigkeit des EE-Koordinatensystems
$\dot{y_{EE}}$	Geschwindigkeit des EE-Koordinatensystems in y-Richtung
$\dot{y}$	Geschwindigkeit in y-Richtung

Tabellenverzeichnis

Abbildungsverzeichnis

1 Einleitung

Im Rahmen der Bachelorarbeit soll das F&E-Vorhaben PIA (Platform for Intelligent Automation) der Hamburg Open Online University (HOOU) erweitert werden. Die Weiterentwicklung des Roboters PIA zu einem präzisen fahrenden Roboter ermöglicht neue Anwendungen wie das autonome Andocken an Ladestationen zur Errichtung einer automatisierten Energieversorgung. Die Roboter PIA sollen als Einzelsysteme und als Schwarm in einem breiten Spektrum eingesetzt werden können, welches bei der schulischen Ausbildung beginnt und in der Forschung an Hochschulen endet. Die Erweiterung des Roboters PIA als Teil eines verteilten Systems ermöglicht den simultanen Betrieb von mehreren Robotern und ebnet den Weg zum Roboterschwarm. Dafür ist die Erstellung eines Frameworks zum Betrieb von mehreren PIA-Robotern notwendig, wie die eindeutige Adressierung, die stabile Datenverbindung und weitere OTA-Funktionalitäten bis hin zu FOTA (Firmware Over-the-Air). Im Rahmen dieser Arbeit wird die Präzision der omnidirektionalen Bewegung des Roboters PIA grundlegend überarbeitet. Beginnend bei einer Analyse der Ausgangssituation werden die Aspekte mechanische Bewegung, elektronische Ansteuerung und Regelung betrachtet. Dafür werden Messprozesse entwickelt, um eine Verifizierung und Validierung der Entwicklung zu ermöglichen. Es soll ein Framework entwickelt werden, das einen gleichzeitigen Betrieb mit Telemetrie von mehreren Systemeinheiten PIA ermöglicht und eine individuelle Steuerung und Regelung jedes Roboters zur Verfügung stellt. Die Funktionalität soll am gleichzeitigen Betrieb von mehreren Systemeinheiten PIA evaluiert und messtechnisch bewertet werden. In einem Gesamtsystemtest werden mehrere Roboter PIA zeitgleich interagieren und der Schwarm als verteiltes System gesteuert.

1.1 Struktur der Arbeit

Zu Beginn der Arbeit werden in Kapitel 2 technische Grundlagen erläutert, welche für das bessere Verständnis der Arbeit erforderlich sind. Hierzu gehören Grundlagen zu Mecanumrädern, PID-Reglern samt Vorsteuerung und messtechnische Grundlagen wie die Lokalisierung mittels ArUco-Marker oder die Funktionsweise eines Virtual-Reality-Trackers.

In Kapitel 3 wird der Ist-Zustand des Roboters PIA004 untersucht und dokumentiert, um ein Konzept zur Verbesserung des Roboters zu erarbeiten. Hierbei werden die Aspekte Hardwaredesign, Kommunikation und Fahrverhalten untersucht. Anschließend werden verschiedene Optionen zur Verbesserung erarbeitet und bewertet.

Die zuvor erarbeiteten Konzepte werden in Kapitel 4 umgesetzt. Zunächst wird die Umsetzung der einzelnen Teillösungen beschrieben und anschließend die Integration der Teillösungen zu einem Gesamtsystem. Zu den Lösungen gehören die Umsetzung eines Firmware-Over-The-Air-Progammers, ein Closed-Loop-Motorcontroller inklusive Vorsteuerung, eine Kursstabilisierung basierend auf den Daten einer inertialen Messeinheit, sowie eines Interfaces zur Steuerung mehrerer Roboter. Abschließend wird ein Anwendungsbeispiel beschrieben, mit dessen Hilfe die Funktionalität des Roboters erprobt werden kann.

In Kapitel 5 wird der neu entstandene Roboter auf seine Funktionalitäten untersucht und bewertet. Zuerst wird die Einzelfahrleistung des Roboters PIA6-1 untersucht und bewertet. Danach werden die Roboter PIA6-1 (siehe Abbildung 1) und PIA6-2 Formationsfahrten durchführen, um die Funktionsweise der umgesetzten Multi-Roboter-Steuerung zu analysieren und zu bewerten.

Im letzten Kapitel wird ein Fazit aus der Arbeit gezogen und ein Ausblick in die Zukunft gegeben.

Abbildung 1: Roboter PIA6-1

2 Technische Grundlagen

Dieses Kapitel behandelt die verschiedenen Technischen Grundlagen, welche nötig sind, um ein besseres Verständnis bezüglich der Umsetzung der Arbeit zu erlangen.

2.1 Mecanum-Rad

Hauptmerkmal des PIA Roboters ist, dass er mittels so genannter Mecanum-Räder angetrieben wird (siehe Abbildung 2). Diese ermöglichen es ihm durch koordiniertes Ansteuern aller vier Räder omnidirektionale Bewegungen auf dem Untergrund zu vollführen [1]. Hierbei sind statt eines Mantels mehrere Walzen unter einem Winkel von $\pm 45°$ so an der Felge montiert, dass jeweils nur eine der Walzen Kontakt mit dem Untergrund hat.

Abbildung 2: Mecanum-Rad

Um eine omnidirektionale Bewegung durchführen zu können, müssen die Räder so montiert werden, dass die walzen, welche Kontakt mit dem Untergrund haben, einen Kreis ergeben. Somit ist es nötig 2 paarweise verschiedene Räder zu haben. Jeweils zwei mit Walzen im Winkel von $+45°$ und zwei im Winkel von $-45°$. Ist dies der Fall so lässt sich die Drehzahl aus der vorgegebenen Fahrtrichtung berechnen.

Hierzu kann die Formel:

$$\begin{cases} \omega_0 = \frac{1}{r}\left(v_x - v_y - \left(l_x + l_y\right)\omega\right), \\ \omega_1 = \frac{1}{r}\left(v_x + v_y + \left(l_x + l_y\right)\omega\right), \\ \omega_2 = \frac{1}{r}\left(v_x + v_y - \left(l_x + l_y\right)\omega\right), \\ \omega_3 = \frac{1}{r}\left(v_x - v_y + \left(l_x + l_y\right)\omega\right). \end{cases} \quad (1)$$

verwendet werden [2]. Eingangsvariablen stellen hierbei die Geschwindigkeiten v_x, v_y und ω dar. Die Variablen ω_i entsprechen der jeweiligen Raddrehzahl, wobei das Rad mit dem Index null der Raddrehzahl des hinteren rechten Rades entspricht. Die weiteren Räder sind fortlaufend im Uhrzeigersinn durchnummeriert. Die Längen l_x und l_y entsprechen dem Abstand des Rades zum Robotermittelpunkt in x und y-Richtung, r entspricht dem Radius des Rades.

2.2 PID-Regler und Vorsteuerung

PID-Regler sind heutzutage die Grundlage für nahezu alle Regelkreise. So fährt zum Beispiel auch der Roboter PIA4 mittels PID-Regelkreises.

Grundsätzlich korrigiert ein Regelkreis eine Regelabweichung $e(t_n)$ die sich aus der Differenz zwischen Führungsgröße $w(t_n)$ und Regelgröße $x(t_n)$ berechnet [3].

$$e(t) = w(t_n) - x(t_n) \tag{2}$$

Hierbei entspricht die Regelgröße $x(t)$ der Differenz aus Stellgröße $u(t)$ und Störgröße $d(t)$. Die Störgröße setzt sich zusammen aus äußeren Einflüssen oder physikalischen Eigenschaften der Regelstrecke.

$$x(t) = u(t_n) - d(t_n) \tag{3}$$

Ein PID-Regelkreis setzt sich aus einem P-, I- und D-Glied zusammen. Das P-Glied steuert hierbei eine Stellgröße proportional zum Fehler bei.

$$u_P(t_n) = K_P \cdot e(t_n) \tag{4}$$

Ein I-Glied liefert für einen diskreten PID-Regelkreis eine Stellgröße welche proportional zur integrierten Regelabweichung ist.

$$u_I(t_n) = K_I \cdot \sum (e(t_n) \cdot dt) \tag{5}$$

Die Zeit dt entspricht hierbei der Zeit zwischen zwei Berechnungen der Stellgröße.

Das D-Glied liefert eine Stellgröße, welche proportional zur Änderungsgeschwindigkeit der Regelabweichung $e(t)$ ist. Für den diskreten Fall berechnet sich dieser Anteil wie folgt.

$$u_D(t_n) = K_D \cdot \frac{\big(e(t_n) - e(t_{n-1})\big)}{dt} \tag{6}$$

Kombiniert man alle drei Regelglieder, so kommt man zu folgendem Ergebnis für einen diskreten PID-Regler:

$$u(t_n) = u_{PID}(t_n) = u_P + u_I + u_D \tag{7}$$

Die Stellgröße kann dann mit Hilfe der Verstärkungsfaktoren beeinflusst werden. Setzt man einen der Verstärkungsfaktoren auf null, so hat das jeweilige Glied keinen Einfluss mehr auf die Stellgröße. Somit sind P-, I-, PI- und PD-Regler auch mit der allgemeinen Form eines PID-Reglers beschreibbar.

Ein keiner D-Regler lässt sich ebenso mittels eines PID-Reglers beschreiben. Da jedoch sein einziger Antrieb Messrauschen wäre, kommt diese Art eines Reglers in der Praxis nicht zum Einsatz.

Neben den Verstärkungsfaktoren spielt auch die Frequenz f_{Update}, mit welcher der Microcontroller die Berechnungen des Reglers ausführt, eine Rolle für das Verhalten.

Hieraus lässt sich dann die Samplezeit bestimmen.

$$t_{sample} = \frac{1}{f_{Update}} \tag{8}$$

Diese Samplezeit kann mitunter zu Instabilitäten führen. Wird diese zum Beispiel zu groß gewählt, können große Fehler durch Diskretisieren entstehen, was dazu führen kann, dass der Regler instabil wird. Wählt man die Samplezeit zu gering, können Instabilitäten durch zu extreme Stellgrößen entstehen [4].

Dieses Grundkonzept des PID-Reglers kann durch eine Vorsteuerung ergänzt werden. Diese liefert eine Stellgröße, welche proportional zur Führungsgröße ist.

$$u_{VS}(t_n) = K_{VS}(w(t_n)) \tag{9}$$

Die Funktion $K_{VS}(w)$ kann hierbei jede beliebige Funktionalität aufweisen und beschreibt die im Mittel zu erwartenden Stellgröße für eine bestimmte Führungsgröße. Somit errechnet sich die Stellgröße für einen PID-Regler mit Vorsteuerung wie folgt:

$$u(t_n) = u_{PID}(t_n) + u_{VS}(t_n) \tag{10}$$

Dieses Verfahren bietet den Vorteil, dass der Regler besser auf Sprungantworten reagieren kann, da die zu erwartende Stellgröße ohne Zeitverlust eingestellt werden kann. Der PID-Regler muss in diesem Fall lediglich Regelabweichungen vom Mittelwert kompensieren, welche durch äußere Einflüsse oder durch ungenaue Vorsteuerungswerte entstehen [5].

2.3 Augmented Reality-Marker

Augmented Reality-Marker (AR-Marker) sind aus dem Bereich der *Computer Vision*. Ein Bereich, in dem mit Hilfe von Kameras Bilder aufgenommen werden und mittels Computer ausgewertet werden, so das verschiedene Informationen gewonnen werden können. Der wohl bekannteste unter den AR-Markern ist der QR-Code, mit welchem binäre Daten bereitgestellt werden können.

Ein weniger bekannter und dennoch sehr nützlicher Marker stellt der ArUco-Marker dar, welcher an der *University of Córdoba* entwickelt wurde. Dieser Marker zeichnet sich durch seine quadratische Form und einem schwarzen Rahmen auf weißem Hintergrund aus. In seinem Inneren beinhaltet er eine binär-Kodierte ID, welche unabhängig der Rotation des Markers eindeutig bleibt. Der Vorteil dieser Marker im Vergleich zu QR-Codes besteht darin, dass er auf größere Entfernungen auch mit schlechterer Auflösung deutlich zu erkennen und identifizieren ist [6].

Die verschiedenen Marker und ihre ID werden in einem sogenannten Dictionary zusammengefasst. So beinhaltet ein Dictionary alle eindeutigen binären Codes und die Größe des ArUco-Markers.

Ist die Kantenlänge eines ArUco-Markers bekannt, kann mit Hilfe der ID und der mittels der Kamera bestimmten Länge der einzelnen Kanten bestimmt werden, wo sich die Kamera relativ zum ArUco-Marker befindet.

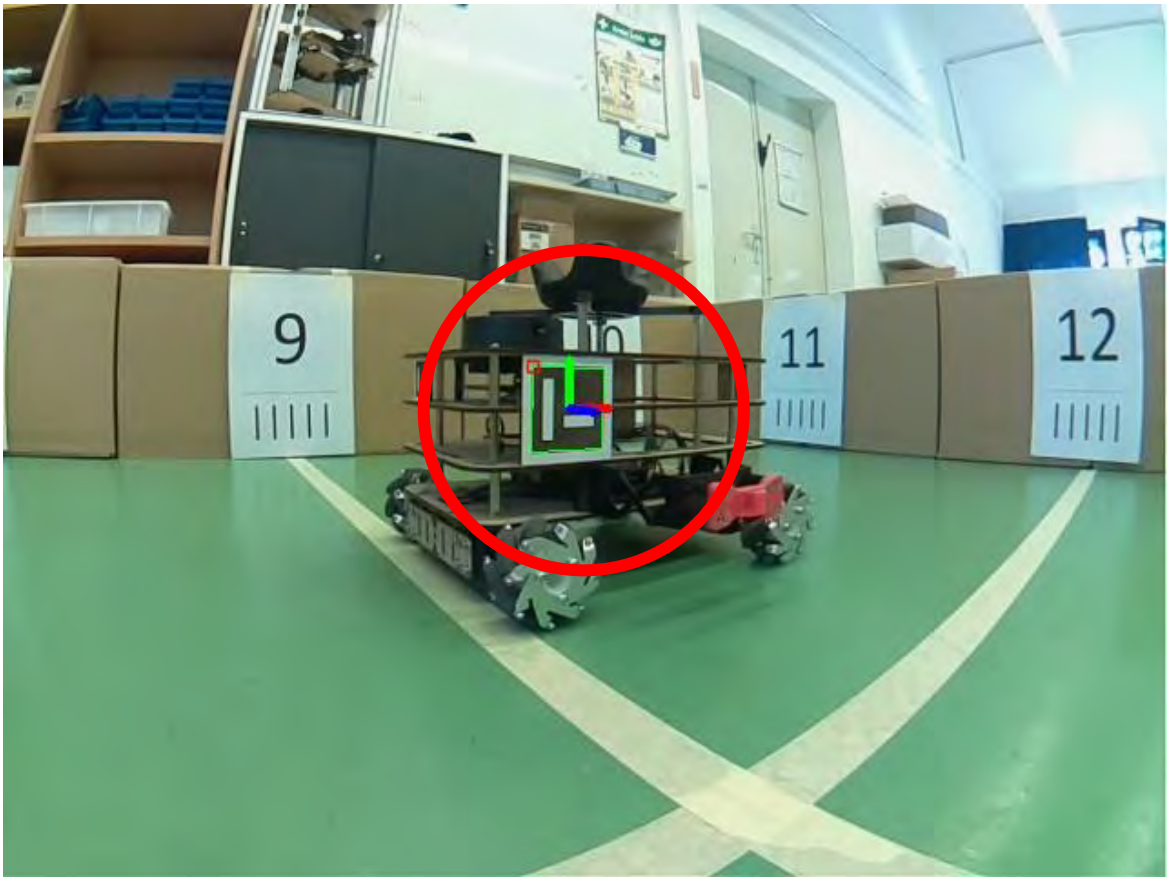

Abbildung 3: Roboter Aruco Detection

In Abbildung 3 ist ein ArUco-Marker zu sehen, welcher an einem PIA-Roboter montiert ist. Ebenso ist das Marker-Koordinatensystem dargestellt, welches im Dictionary für diese Marker ID abgelegt ist.

Weitere intrinsischen Kamera-spezifischen Parameter sind von Nöten, um eine präzise Ortung eines ArUco-Markers vornehmen zu können. Diese können jedoch bei der Kalibrierung mittels eines Schachbrettmusters ermittelt werden.

Sind alle Voraussetzungen gegeben, kann ein ArUco-Marker dafür verwendet werden, einen anderen Roboter zu identifizieren und relativ zur eigenen Position zu lokalisieren.

Mit Hilfe der *Open Computer Vision* (OpenCV) Bibliothek lässt sich die Identifizierung und Lokalisierung leicht und effizient auch auf einem RaspberryPi, einem auf ARM-Technologie basiertem Mini-Computer, umsetzten.

2.4 VR-Tracker (HTC VIVE)

Zur Beurteilung der durch den Roboter gefahrenen Trajektorie ist ein VR-Tracker zu verwenden. Bei dem zur Verfügung stehenden Produkt handelt es sich um einen Tracker bestehend aus zwei *Basisstationen* des Herstellers HTC. Diese Basisstationen werden auch Lighthouse genannt, da sie einen Infrarot Laser aussenden.

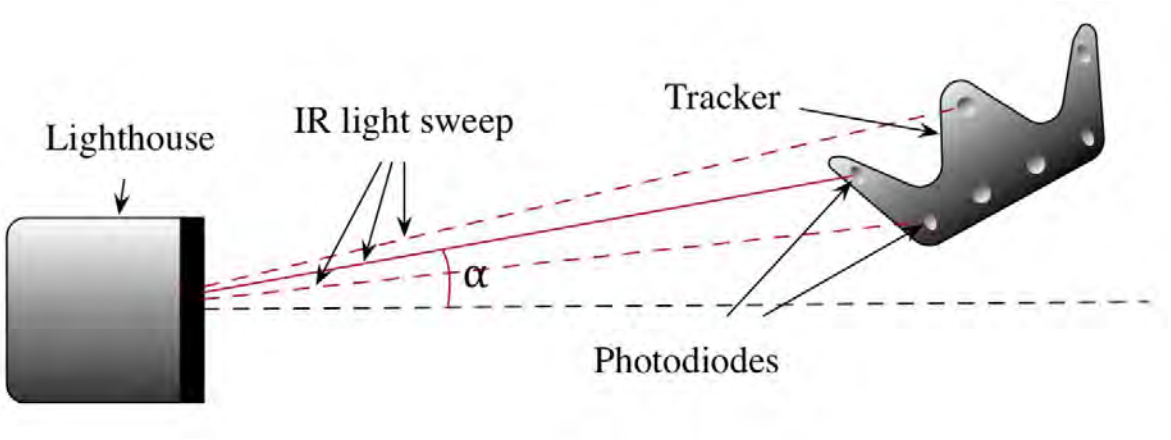

Abbildung 4: Funktionsweise HTC VIVE Tracker [7]

In Abbildung 4 zu sehen ist eine schematische Darstellung eines Lighthouses sowie eines VR-Trackers. Dieser ähnelt in seiner Darstellung dem in dieser Arbeit zu verwendenden *VIVE Tracker (2018)* der Firma HTC. Der „IR light sweep“ beschreibt die Ebene, auf welche der Laserstrahl gestreut wird. Diese Laserebenen werden über die Zeit beispielsweise um den Winkel α variiert. Diese Darstellung repräsentiert lediglich eine der Varianten des Aussendens, geschieht aber in der Praxis unter drei verschiedenen Winkeln.

An dem Tracker zu sehen sind mehrere Photosensoren, welche die Laserebene zu unterschiedlichen Zeitpunkten detektieren. Durch die Zeitdifferenz lässt sich die Position und Orientierung des VR-Trackers zur Basisstation berechnen.

Die zu erwartende absolute Positionsgenauigkeit in einer dynamischen Anwendung liegt im Zentimeterbereich, wobei die Streuung im Submillimeterbereich liegt. Um dies auch für den in dieser Arbeit verwendeten Aufbau zu zeigen werden zwei Tests durchgeführt. Zum einen wird die Streuung über einen Zeitraum von ca. 20 Sekunden mit einer Frequenz von $f = 50\,\mathrm{Hz}$ gemessen, zum anderen wird die absolute Positionierungsgenauigkeit mittels neun verschiedener Messpunkte innerhalb des Messbereichs ermittelt.

Berücksichtig wird hierbei lediglich die x- und y-Ebene des VR-Tracker-Koordinatensystems zum Zeitpunkt $t = 0\,\mathrm{s}$.

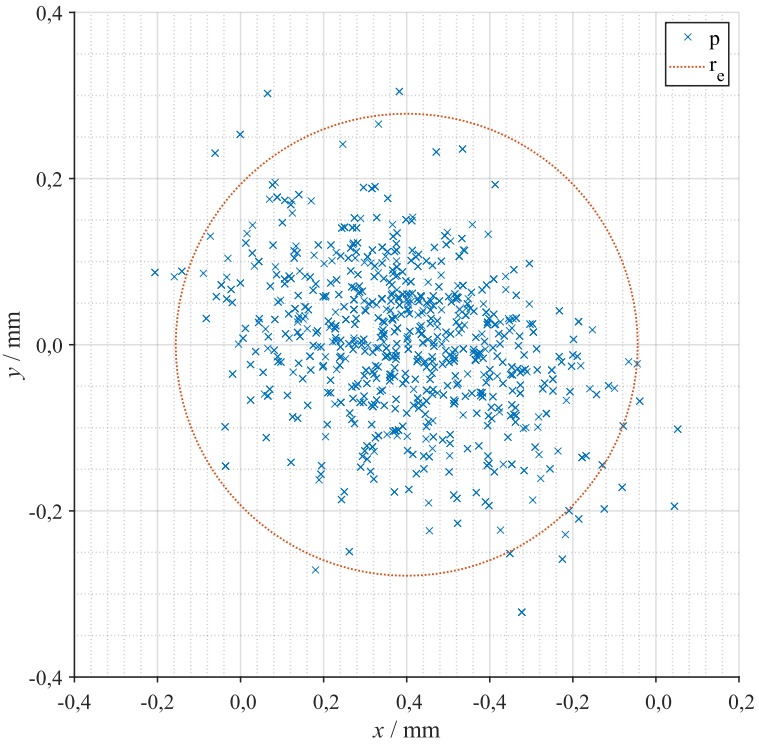

Abbildung 5: VR-Tracker-Streuung (XY-Diagramm)

Wie in Abbildung 5 zu sehen ist, liegen die Messwerte im statischen Fall alle im Submillimeterbereich. Der eingezeichnete Radius von $r_e = 0{,}28$ mm errechnet sich wie folgt:

$$r_e = 2 \cdot \frac{\Sigma|r_{e,n}|}{\sqrt{N}} \tag{11}$$

Wobei der Fehlerradius $r_{e,n}$ eines jeden Punkts durch

$$r_{e,n} = |\overrightarrow{p_n}| \tag{12}$$

beschreiben wird. Hierbei entspricht $\overrightarrow{p_n} = \begin{pmatrix} x \\ y \end{pmatrix}$ dem jeweiligen Messpunkt.

Um die absolute Genauigkeit des Messaufbaus bestimmen zu können, wird zunächst ein Testfeld erstellt. Hierbei wird mittels eines dünnen Fadens ein Gitter von 3x3 Messpunkten erstellt. Diese werden dann mit Hilfe eines Bosch Professional GLM 50 Laser Entfernungsmessers eingemessen. An diesen als absolut angenommen Messpunkten wird der VR-Tracker positioniert und die Abweichung berechnet.

Hierzu wird die Formel (11) herangezogen. Der Fehlerradius $r_{e,n}$ errechnet sich in diesem Fall jedoch aus der Differenz zwischen Soll- und Ist-Position $\overrightarrow{p_{Ist,n}}$ und $\overrightarrow{p_{Soll,n}}$:

$$r_{e,n} = |\overrightarrow{p_{\mathrm{Ist,n}}} - \overrightarrow{p_{Soll,n}}| \tag{13}$$

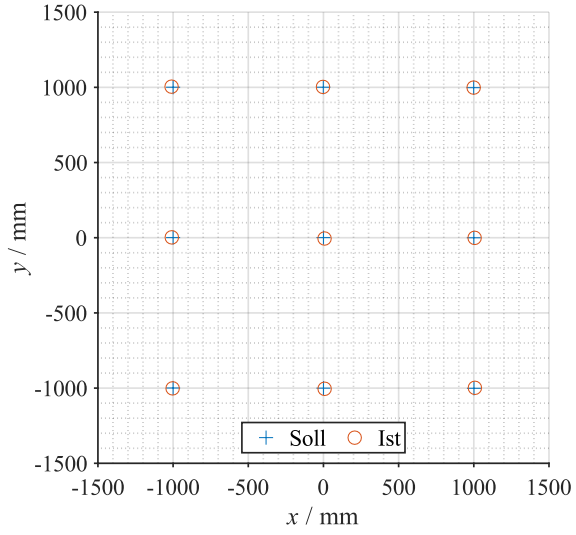

Abbildung 6: Absolute Genauigkeit VR-Tracker, Soll- gegenüber Ist-koordinaten

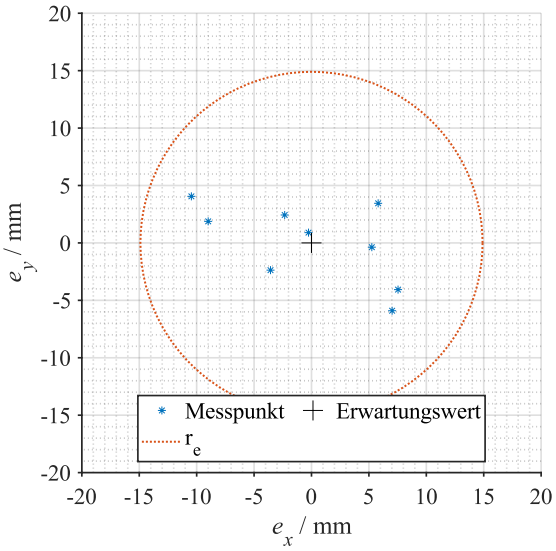

Abbildung 7: VR-Tracker Soll-Ist-Fehler, sowie der Fehlerradius r_e

Wie in Abbildung 6 und Abbildung 7 zu sehen ist, ist der Fehlerradius $r_e = 15$ mm und liegt somit im zu erwartenden Bereich.

Zwar lassen sich diese Daten theoretisch mittels Kalibrierung soweit verbessern, dass der VR-Tracker auch absolute Ergebnisse im Submillimeterbereich liefert, jedoch wurde das bislang nur in Simulationen gezeigt [8] und würde den Rahmen dieser Arbeit sprengen.

Da der zu erwartende relative Messfehler bei Fahrten auf einer ähnlichen Strecke jedoch gering ausfallen wird, werden Messdaten im Millimeterbereich angegeben, da qualitative Aussagen dadurch dennoch möglich sind. Für eine quantitative Aussage wäre es jedoch erforderlich, eine Kalibrierung der VR-Tracker durch zu führen oder ein anderes Messverfahren zu verwenden.

3 Ist-Zustand

Um den Ist-Zustand des PIA-Projekts besser beurteilen zu können, wird mit Hilfe verschiedener Analysen die Präzision, mit welcher der Roboter eine Pose erreicht bestimmt werden. Hierzu wird ein Virtual-Reality-Tracker an dem Roboter montiert. Dieser liefert mit Hilfe der Software *SteamVR* und der Python-Bibliothek *openVR* die Möglichkeit, die genaue Pose des Roboters zu bestimmen. Eine andere anfängliche Methode stellt die Verwendung eines Whiteboards dar, auf welchem der Roboter mit einem an der Seite montierten Stift, verschiedene Muster abfährt. Dies bietet einen guten ersten Eindruck, lässt aber keine Aussagen über das zeitliche Verhalten sowie über die Yaw-Genauigkeit zu, weshalb im Weiteren ausschließlich Daten des VR-Trackers verwendet werden.

Da nicht nur das Fahrverhalten des Roboters zu beurteilen ist, sondern auch das Protokoll, über welches die Roboter kommunizieren, ist es ebenso nötig, die Softwarestruktur des Roboters zu verstehen und zu analysieren. Als Referenz soll der Roboter mit der Bezeichnung PIA004 mit dem Softwarestand vom 18.09.2019 dienen. Alle relevanten Software- und Hardwarestrukturen werden dokumentiert und verschiedene Konzepte hinsichtlich ihrer Umsetzbarkeit, Kosten, Nutzen und Schwächen diskutiert.

3.1 Hardwaredesign des Roboters PIA004

Der PIA004-Roboter ist ein verteiltes System, welches aus mehreren Komponenten besteht. Diese verschiedenen Komponenten sind auf verschiedene Layer aufgeteilt, sodass unterschiedlichste Konfigurationen leicht umsetzbar sind.

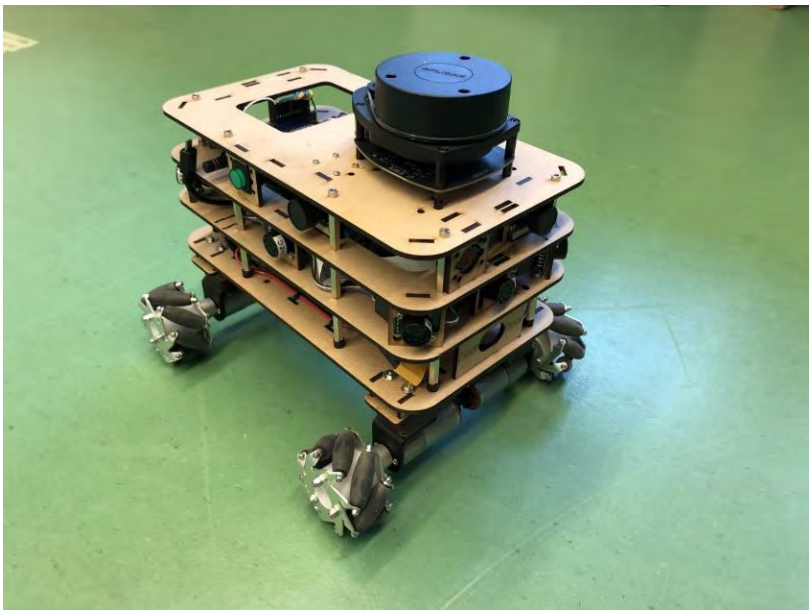

Abbildung 8: Foto PIA004

In Abbildung 8 zu sehen ist der Roboter PIA004, welcher ein Roboter mit der höchsten Ausbaustufe darstellt. Er verfügt über alle in PIA Robotern verbauten Sensoren.

Das erstellte Systemdesign gibt einen Überblick über die verwendeten Komponenten und ihrer Layer Zugehörigkeit.

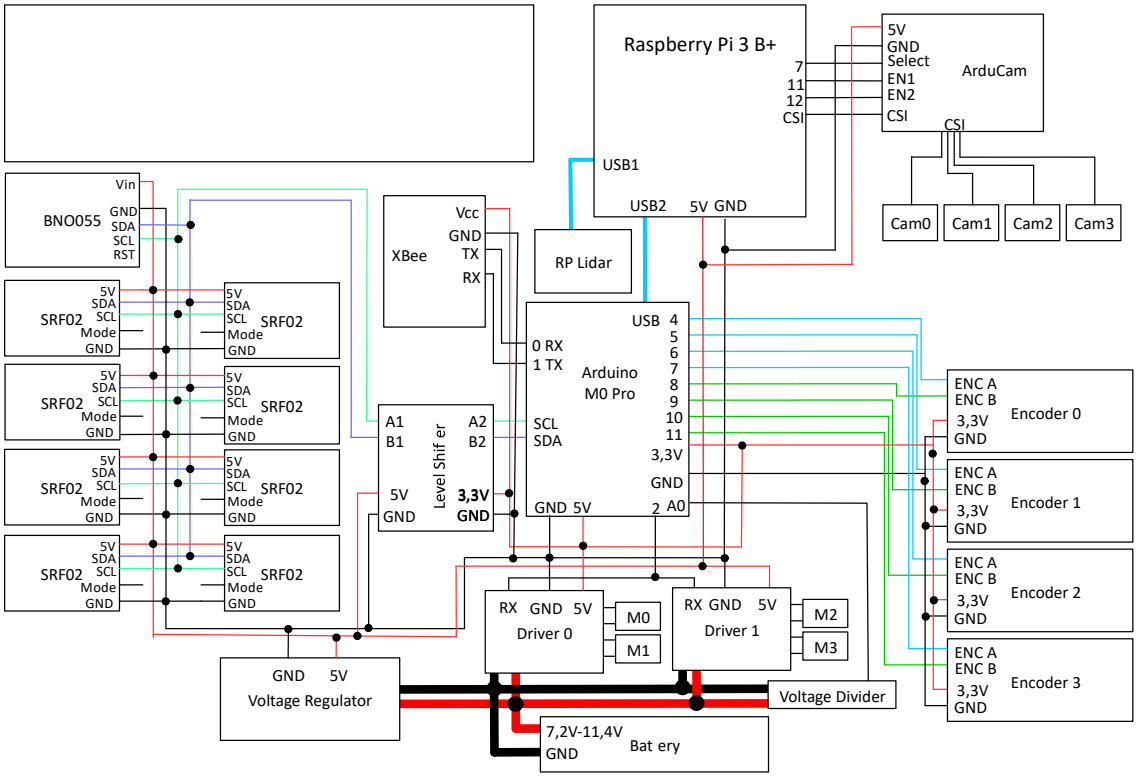

Abbildung 9: Systemdesign PIA004 – Config 3 [9]

Wie dem Systemdiagramm in Abbildung 9 zu entnehmen ist, besteht der PIA004 Roboter aus insgesamt fünf Layern.

Layer 0: In diesem Layer sind die Motoren und ihre Quadraturencoder verbaut. Die vier Motoren dienen dem Antrieb von vier Mecanum-Rädern. Diese ermöglichen es dem Roboter omnidirektionale Bewegungen aus zu führen. Diese verschiedenen Bewegungsrichtungen ergeben sich aus einem Zusammenspiel der verschiedenen Drehzahlen und Drehrichtungen des jeweiligen Rads (siehe Kapitel 2.1). Um diese über den Microcontroller Arduino M0 Pro mit einer bestimmten Drehzahl ansteuern zu können, muss die Drehzahl gemessen werden. Dies geschieht mittels der Quadraturencoder.

Layer 1: Dieser Layer beherbergt die Leistungselektronik. Neben dem Energieträger, einem 3-Zellen Lithium Polymere Akkumulator, kurz 3S-LiPo, sind auch zwei sogenannte Step-Down-DC-DC-Converter verbaut, welche eine Spannung von 5 V bzw. 3,3 V erzeugen.

Hinzu kommen zwei Motorcontroller vom Typ Qik 2s9v1 des Herstellers Pololu. Diese regeln die elektrische Leistung, welche an die Motoren weitergegeben wird. Dies geschieht auf Grund der vom Microcontroller gegebenen digitalen Signalen.

Layer 2: Dieser Layer ist der letzte Layer der Basis-Konfiguration (Config 1), welche nur eine reduzierte Version der dargestellten Config 3 darstellt. Hier befindet sich ein Arduino M0 Pro, welcher für die Erfassung von Sensordaten, sowie für die Ansteuerung der Motoren verantwortlich ist. Zu den ausgewerteten Sensoren gehören die vier Quadraturencoder der Motoren, eine inertiale Messeinheit BNO055 des Herstellers Bosch (IMU) sowie acht Sonar

Sensoren vom Typ SRF02. Da einige dieser Sensoren Daten mittels 5V-Logik übertragen, ist hier auch noch ein so genannter 4-bit Lvl-Shifter des Herstellers Adafruit verbaut.

Um die erhobenen Daten auch auf anderen Geräten zur Verfügung zu haben, befindet sich auf dieser Ebene ebenfalls ein XBeeS6B des Herstellers Digi, welcher für eine WLAN-Verbindung sorgt.

In der Config 3 befindet sich auf diesem Layer zusätzlich ein Raspberry Pi 3b+. Er sorgt für die Auswertung von Kamera-Bildern und die Daten des LiDARs welche auf Layer 3 und 4 verbaut sind.

Layer 3: Auf diesem Layer befinden sich vier Kameras vom Typ rb-camera-WW der Firma Joy-it. Diese werden mittels eines ArduCam boards verwaltet, da der Raspberry Pi 3b+ sonst nur in der Lage wäre *eine* Kamera zu verwenden.

Ebenso sind auf dieser Ebene ArUco-Marker montiert, sodass der Roboter auch von anderen identifiziert und lokalisiert werden kann.

Layer 4: Dieser Layer ist der Abschluss Layer. Er beherbergt einen 2D Laserscanner vom Typ RPLIDAR A1-M8-R5 des Herstellers Slamtec. Dieser Layer kann zu Testzwecken noch um einen VIVE-Tracker 2018 der Firma HTC erweitert werden.

3.2 Kommunikation

Im Zuge der Bachelor-Thesis werden Konzepte zur Weiterentwicklung des Pia-Projekts zu einem Roboterschwarm erarbeitet und umgesetzt. Hierzu muss die aktuelle Roboter Software und Hardware Architektur analysiert und verstanden werden, um eine best mögliche Umsetzung zu gewährleisten.

3.2.1 Paketstruktur

Die Software des PIA-Projekts ist sehr umfangreich und ermöglicht schon zum Zeitpunkt des Beginns dieser Arbeit viele Diagnosen und Tests. Die Kommunikation des verteilten Systems stützt sich dabei auf verschiedene implementierte Pakete. Der Roboter wird auf diese Art gesteuert, kann aber auch seine Sensordaten anderen an das Netzwerk angeschlossenen Computern oder Microcontrollern zur Verfügung stellen.

Pakete des PIA-Projekts haben grundsätzlich eine sehr ähnliche Struktur. Sie beginnen mit einer ID, gefolgt von der Zeit des Erstellens des Paketes in Millisekunden. Darauf folgen die zu übermittelnden Daten. Um die Gültigkeit eines übertragenen Pakets zu gewährleisten, endet das Paket mit einer CRC-Checksumme.

Ein so strukturiertes Paket bietet also zum aktuellen Zeitpunkt keine Möglichkeit der Zuweisung. Dem Paket selbst kann keine Quelle und kein Ziel entnommen werden.

3.2.2 Schnittstellen

Die beschriebenen Pakete werden mittels serieller Schnittstellen roboterintern verschickt beziehungsweise empfangen. Abnehmer sind ein *XBeeS6B* und, falls verbaut, ein *Raspberry Pi 3b+*. Ebenso kann der Roboter direkt mit einem USB-Kabel mit einem PC verbunden werden. Da dies nur zum Programmieren dient und im Betrieb keine sinnvolle Alternative darstellt, soll dies im Weiteren nicht vertieft betrachtet werden.

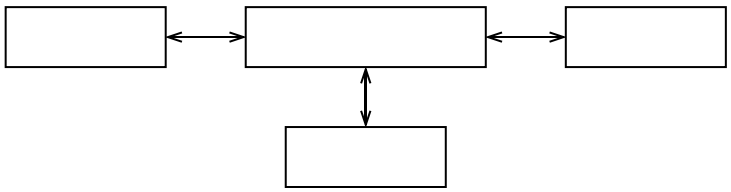

Abbildung 10: Serielle UART Kommunikation

Daten, die über die serielle Schnittstelle an den Raspberry übertragen werden, können dort entpackt werden und beispielsweise in ein ROS-Netzwerk eingespeist werden.

An den XBee gesendete serielle Daten werden über WLAN an andere XBees verschickt. Hierzu müssen die XBees mittels der XCTU-Software richtig konfiguriert werden. Um eine Kommunikation mit anderen Robotern und XBees zu gewährleisten, müssen alle XBees im gleichen WLAN angemeldet werden. Ebenso ist als Protokoll UDP auf dem Port 9750 zu wählen. Damit alle Roboter die Nachrichten bekommen können, müssen sie ihre Nachrichten als Broadcast senden, hierzu ist als Ziel-IP die 255.255.255.255 zu wählen.

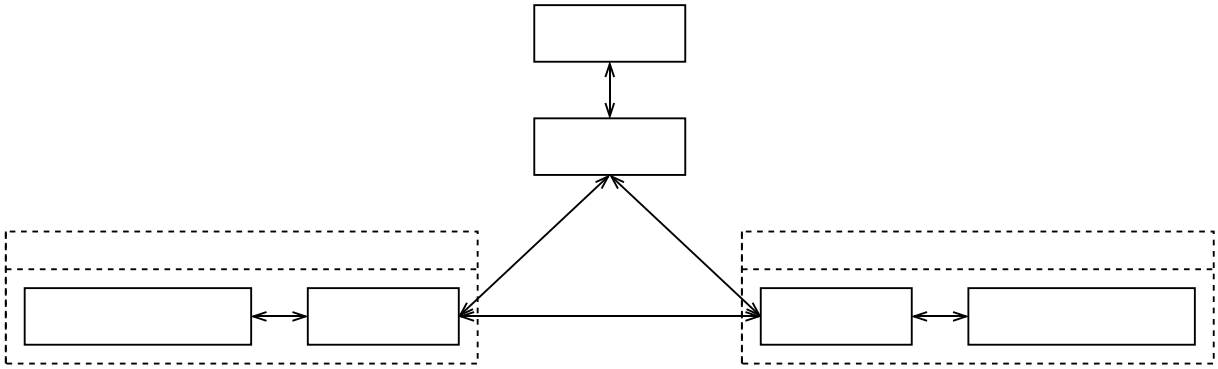

Abbildung 11: XBee Kommunikation

Die so von XBees empfangen Daten, werden von ihm wieder in serielle Daten umgewandelt, sodass XBees als eine Art virtuelles Kabel dienen.

Die seriellen Daten des XBee-outputs können vom Empfänger, dem Arduino M0 Pro interpretiert werden. Somit ist eine drahtlose Kommunikation zwischen zwei Komponenten des verteilen Systems möglich. Da die Daten als Broadcast verschickt werden, erhält jeder XBee dieselben Daten. Es ist somit nicht möglich jedem Roboter individuell Daten zu kommen zu lassen. Ebenso lässt sich die Quelle der Daten auf diese Weise nicht ermitteln.

Mittels der Software Wireshark wird die Kommunikation zwischen XBee-Modulen anschließend analysiert. Dieses Programm ist in der Lage den Netzwerkverkehr aufzuzeichnen. Einzige Voraussetzung hierfür ist, dass der Netzwerkverkehr entweder über den Wireshark ausführenden PC läuft oder dass die zu untersuchenden Pakete auch an den PC adressiert sind.

Da die Konfiguration der XBees bekannt ist, lassen sich die eigehenden Daten nach UDP-Paketen auf dem Port 9750 filtern. Diese werden mittels Broadcast an andere Teilnehmer verschickt, weshalb sie auch auf dem PC zu empfangen sind.

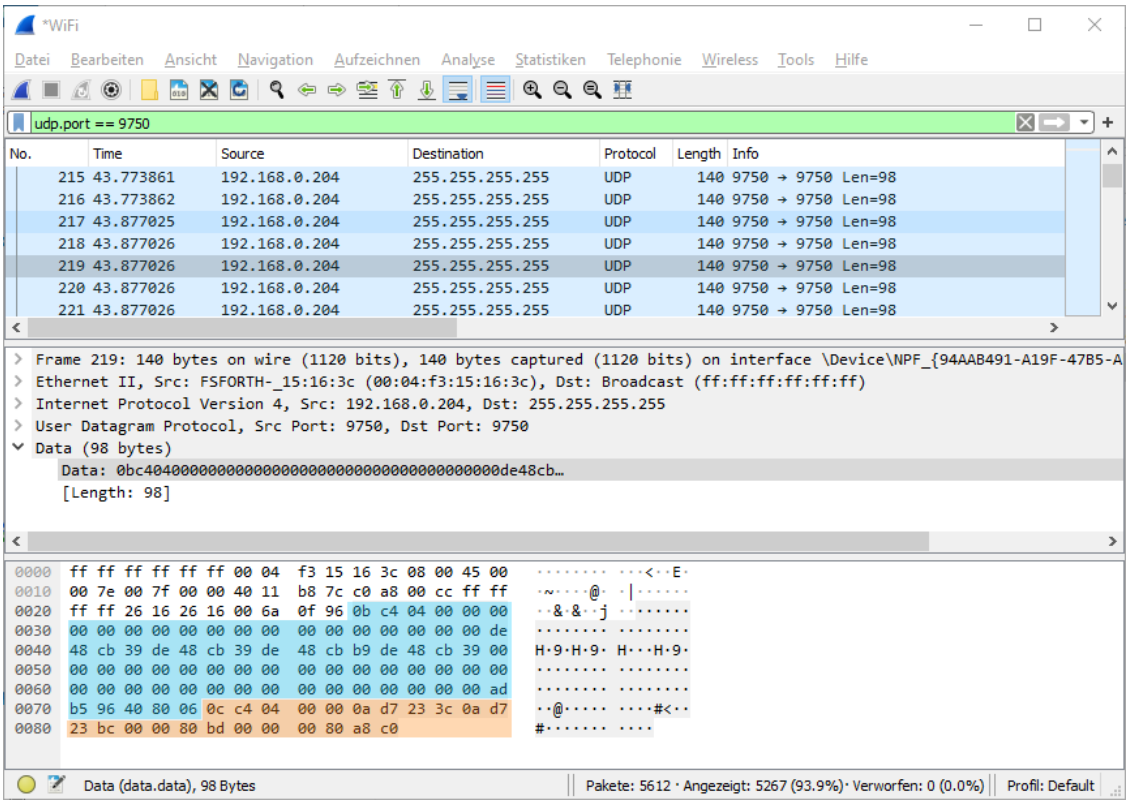

Abbildung 12: Wireshark

Abbildung 12 zeigt das Fenster des Programms Wireshark. Im oberen Bereich sind alle eingehenden und ausgehenden Daten auf dem UDP-Port 9750 zu sehen.

Im mittleren Bereich ist der Aufbau des im oberen Bereich ausgewählten Pakets zu sehen. Gegliedert ist dieser Bereich in die verschiedenen Layer nach dem OSI-Modell [10].

Der Bereich, welcher analysiert werden soll, sind die Daten, welche innerhalb des UDP-Pakets verschickt werden. Diese sind im untersten Bereich zu sehen. Der unmarkierte Abschnitt beschreibt die Adressierung des Pakets. Der Bereich, welcher blau- bzw. orangemarkiert ist, repräsentiert die übertragenen Bytes im Hexadezimalsystem. Im betrachteten Fall sind dies 98 Bytes.

Betrachtet man die Daten genauer fällt einem auf, dass der blaumarkierte Bereich den Daten eines Diagnostics-Pakets entspricht, gefolgt durch ein IMU-Paket. Interessant ist hier, dass der XBee zwar nur komplette Pakete zu versenden scheint, aber nicht immer in der Lage ist, diese Pakete voneinander zu trennen. Es ist also davon aus zu gehen, dass der XBee Daten erst dann zu einem Paket zusammenfasst, wenn über einen gewissen Zeitraum keine weiteren Daten über die serielle Schnittstelle gelangen. Diese Zeit wird mittels der Option „RO Packetization Timout“ im Konfigurationsprogramm XCTU konfiguriert.

Da der XBee die zu verschickenden Daten nicht verschlüsselt oder mit einer Checksumme versieht, ist es ohne weiteres möglich mit Hilfe eines Python-Skripts oder ähnlichem, die XBee-Kommunikation zu emulieren. Somit können Pakete auch an bestimmte IPs verschickt werden und es kann der Sender eines Pakets ermittelt werden. Dies gibt die Möglichkeit, mit mehreren Robotern zeitgleich verschiedene Trajektorien ab zu fahren.

Eine Abwägung der Vor- und Nachteile findet sich in Kapitel 3.4.

3.3 Fahrverhalten von Roboter PIA004

Um das Fahrverhalten des aktuellen Roboters beurteilen zu können wird eine Reihe von Tests benötigt. Hierbei werden zunächst die Kursstabilität und die Positionsgenauigkeit bestimmt, um Ansatzpunkte zur Verbesserung des bestehenden Systems zu finden.

Ein Messverfahren, welches entwickelt wurde, stellt eine Fahrt auf einem Whiteboard dar. Hierbei wird ein Whiteboard Marker an der Seite des Roboters befestigt. Anschließend wird eine geplante Trajektorie abgefahren.

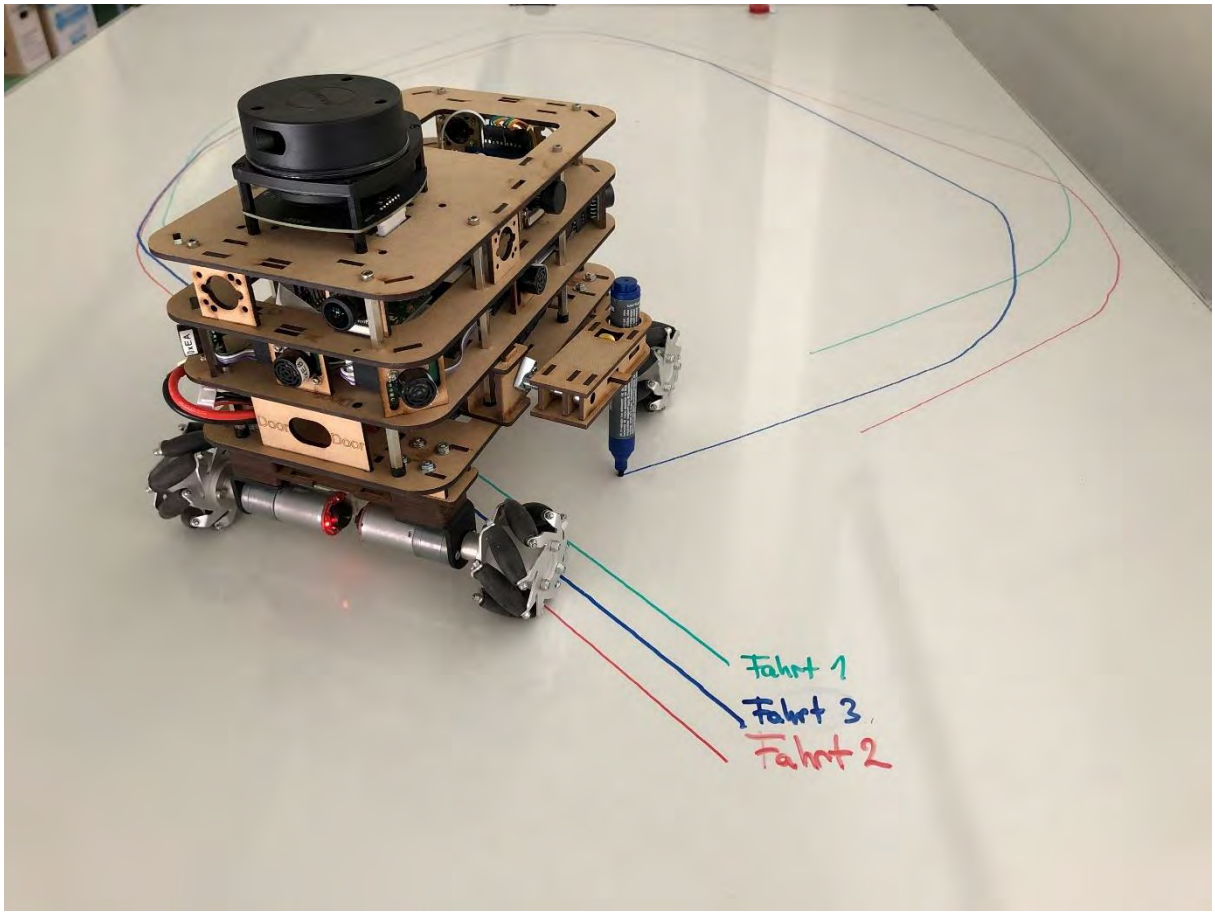

Abbildung 13: Messverfahren Whiteboard PIA004, Kreis Trajektorie r = 0,5 m

Auf Abbildung 13 ist der Roboter PIA004 am Ende der dritten Fahrt einer Kreis Trajektorie zu sehen. Der Kreisradius soll $r = 0{,}5$ m betragen. Klar zu sehen ist, dass der Roboter sich nicht auf einer Kreisbahn bewegt hat.

Um hier jedoch weitere Aussagen treffen zu können, wird dieses Messverfahren wieder verworfen und stattdessen ein VR-Tracker verwendet. Dieser lässt es nämlich zum einen zu, den Mittelpunkt des Roboters zu tracken, zum anderen ist auch eine zeitliche Auflösung der Position möglich.

3.3.1 Test 1: x-Richtung bei konstanter Strecke und Geschwindigkeit

In einem ersten Test fährt der Roboter PIA004 eine Strecke von $S = 1{,}00\text{m}$ mit einer Geschwindigkeit $v_x = \pm 0{,}05\,\frac{\text{m}}{\text{s}}$. Somit ergibt sich eine theoretische Fahrtzeit von $t = 20\text{s}$. Zur Auswertung werden alle Fahrten mit einer Geschwindigkeit von $v_x < 0\,\frac{\text{m}}{\text{s}}$ um 180° gedreht, sodass sich die Zielpunkte decken und bei $p_{Soll} = \begin{pmatrix}1000\\0\end{pmatrix}\text{mm}$ liegen sollten.

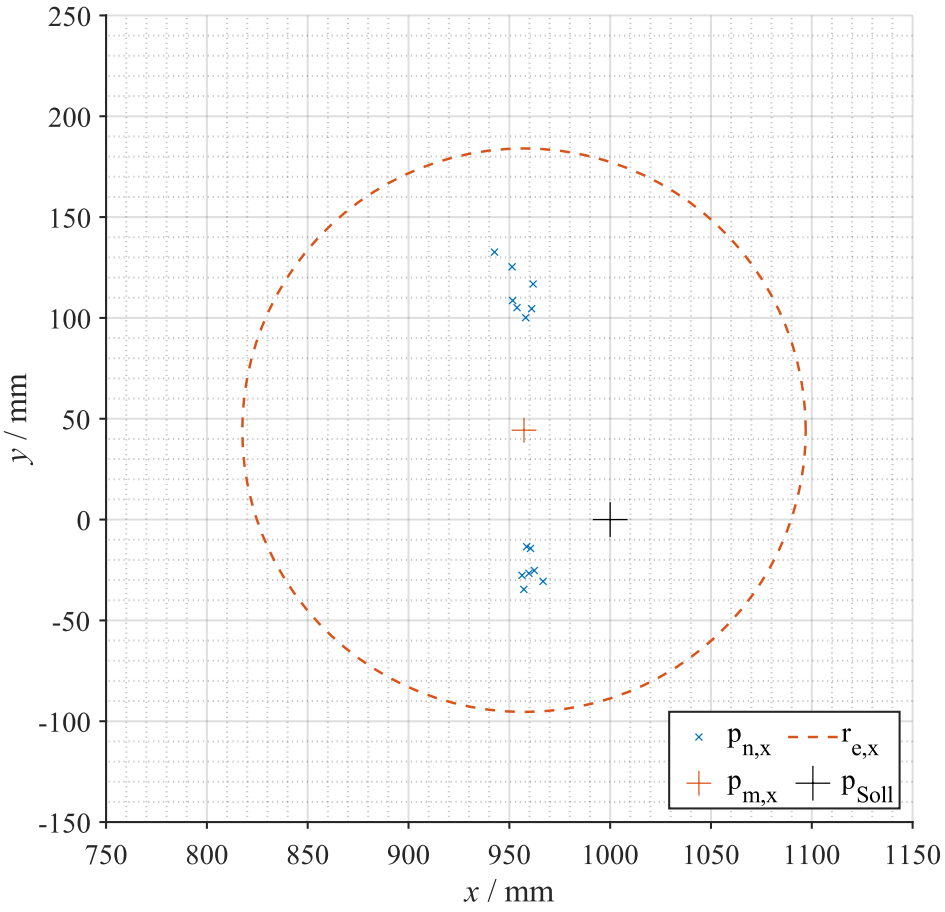

Abbildung 14: Fehlerradius für Fahrten in X-Richtung für den Roboter PIA004

Wie Abbildung 14 zeigt, sind die Endpunkte der verschiedenen Fahrten zu sehen. Diese errechnen sich wie folgt:

$$\overrightarrow{p_n} = \begin{pmatrix}x_{n,end}\\y_{n,end}\end{pmatrix} \tag{14}$$

Der mittlere Endpunkt $p_m = \begin{pmatrix}957\\44\end{pmatrix}\text{mm}$ errechnet sich nach der Formel:

$$\overrightarrow{p_m} = \overline{p_n} \tag{15}$$

Zu dem Punkt $\overrightarrow{p_m}$ kann nun zu jedem Punkt $\overrightarrow{p_n}$ der sogenannte Fehlerradius ermittelt werden:

$$r_{e,n} = |\overrightarrow{p_n} - \overrightarrow{p_m}| \tag{16}$$

Der zu erwartende Fehlerradius $r_e = 140$ mm wird mit Hilfe der Formel (17) ermittelt.

$$r_e = 2 \cdot \frac{\Sigma|r_{e,n}|}{\sqrt{N}} \tag{17}$$

Hierbei entspricht N der Anzahl aller Fehlerradien.

Betrachtet man die Abbildung 14 jedoch genauer fällt auf, dass sich zwei Gruppen deutlich voneinander unterscheiden. Um dieses Phänomen weiter zu untersuchen werden die tatsächlichen Fahrtverläufe dargestellt und weiter untersucht.

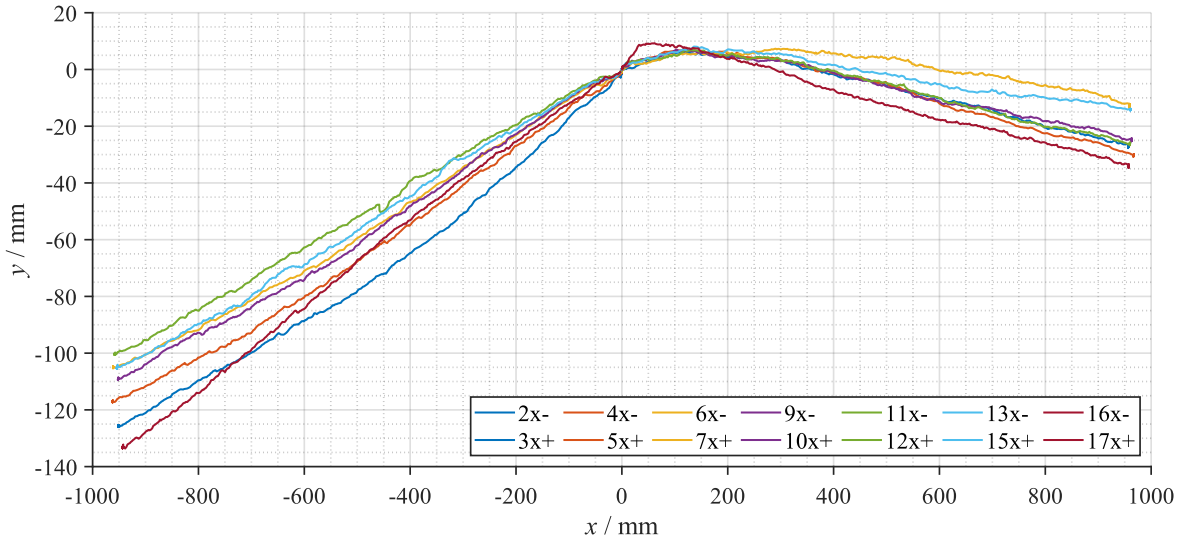

Abbildung 15: Verfahrweg PIA4 mit Zielvektor $\hat{x} = \pm$[1000mm, 0mm, 0deg]

In Abbildung 15 zu sehen, sind alle aufgenommenen Fahrten in positive sowie negative Roboter-x-Richtung. Die y-Achse ist dabei um den Faktor 10 vergrößert, so dass Effekte besser sichtbar werden.

So ist in der Abbildung 15 zu erkennen, dass der Roboter PIA004 nicht entlang einer Geraden fährt.

3.3.2 Test 2: x-Richtung bei konstanter Geschwindigkeit und variabler Strecke

In diesem Test wird der Roboter konstant mit der Geschwindigkeit $v_x = 0{,}05 \frac{\text{m}}{\text{s}}$ angesteuert. Über Variation der Fahrtzeit werden unterschiedliche Strecken eingestellt.

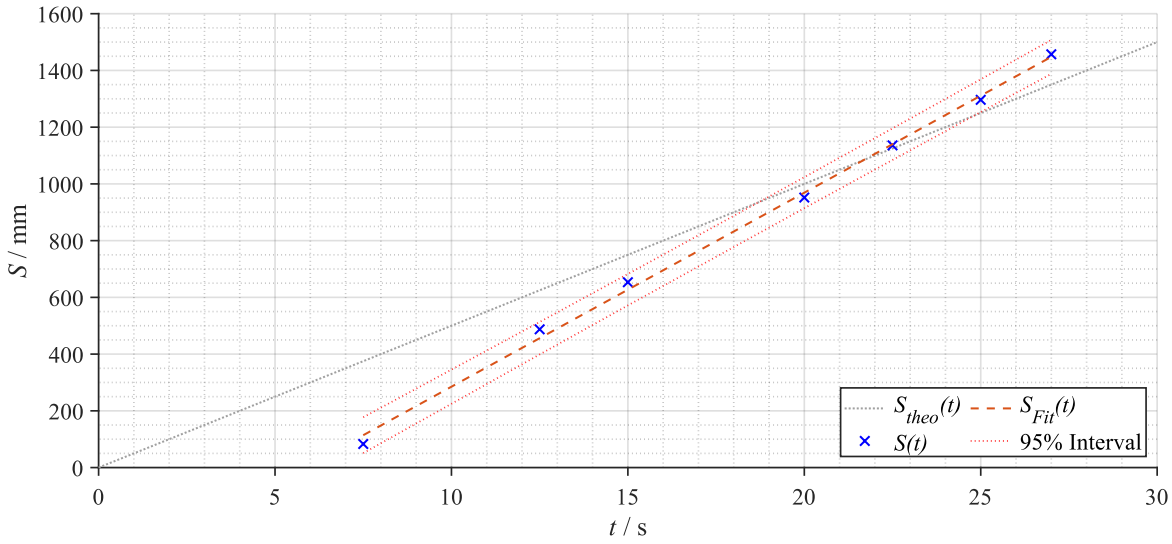

Abbildung 16: Gegenüberstellung theoretische/tatsächliche Strecke bei konstanter Geschwindigkeit

Die theoretisch zurückgelegte Strecke

$$S_{theo}(t) = v_x \cdot t \tag{18}$$

weicht wie in Abbildung 16 zu sehen, von der praktisch gemessenen Strecke ab. Fittet man die gemessenen Werte mittels einer Graden, repräsentiert die Steigung die gefahrene Geschwindigkeit.

$$S_{Fit}(t) = v_{ist} \cdot t + S_0 \tag{19}$$

Auffällig ist, dass bis zu einer Zeit von ca. fünf Sekunden keine Strecke zurückgelegt wird. Berechnet man den Schnittpunkt der orangenen Ausgleichsgeraden mit der X-Achse, erhält man den Zeitpunkt ab welchem der Roboter sich im Mittel anfängt zu bewegen.

$$t_{tot,0.05} = \frac{S_0}{v_{Ist}} \approx 5{,}8\ \text{s} \tag{20}$$

Nach dieser Zeit bewegt sich der Roboter nahezu konstant mit einer Geschwindigkeit von $v_{Ist} \approx 0{,}07 \frac{\text{m}}{\text{s}}$. Dies entspricht 134% der eingestellten Geschwindigkeit.

Der Schnittpunkt beider Geraden stellt den Punkt dar, an dem die mittlere Geschwindigkeit über den gesamten Zeitraum der Soll-Geschwindigkeit entspricht. Dieser Punkt liegt bei einer Fahrtzeit von $t = 21{,}7\text{s}$. Dies entspricht einer zurückgelegten Strecke von $s = 1{,}086\text{m}$. Diese Strecke entspricht in etwa der Strecke des ersten Tests, was darauf schließen lässt, dass dies der Kalibrierungsfall für das System war.

Um dem in Abbildung 16 sichtbaren Effekt weiter auf den Grund zu gehen werden stellvertretend für alle Fahrten die Daten der **Fahrt 3x+** des Tests aus Kapitel 3.3.1 weiter analysiert.

Hierzu wird die zurückgelegte Strecke über die Zeit in einem Diagramm dargestellt. Ebenso werden die verschiedenen signifikanten Zeitpunkte markiert, welche den Steuersignalen und der Kurve entnommen werden können. Der Zeitpunkt t_0 entspricht dem Zeitpunkt, an welchem dem Roboter der Befehl gesendet wird, dass er sich mit einer Geschwindigkeit von $v_x = 0{,}05\,\frac{\mathrm{m}}{\mathrm{s}}$ bewegen soll. Zum Zeitpunkt t_1 beginnt der Roboter tatsächlich mit seiner Fahrt. Der Zeitpunkt t_2 entspricht dem Ende der Fahrt.

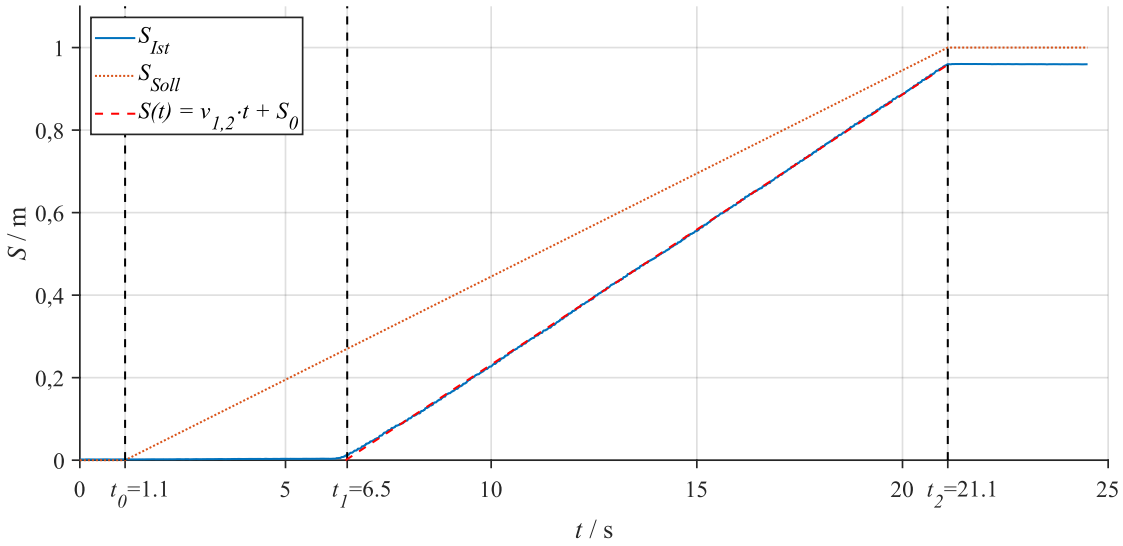

Abbildung 17: PIA004 Totzeit (st-Diagramm)

Wie in Abbildung 17 zu sehen ist, beträgt die Totzeit in dieser Messung $t_{tot} \approx 5{,}4\mathrm{s}$, was circa 25% der gesamten Fahrtdauer von $t_{Fahrt} = t_2 - t_0 = 20\mathrm{s}$ entspricht. Um den Zielpunkt im gegebenen Zeitfenster dennoch zu erreichen, muss deshalb die Geschwindigkeit für die verbleibenden 15 Sekunden um ca. 33% höher sein. Mit Hilfe einer Ausgleichsgrade ist die Geschwindigkeit für den Zeitraum t_1 bis t_2 zu bestimmen. Diese beträgt $v_{1,2} = 0{,}07\,\frac{\mathrm{m}}{\mathrm{s}}$, was sich mit den Analysen bezüglich der Abbildung 16 deckt.

Betrachten wir nun stellvertretend für alle Motoren von Motor0 den PID-Output, sowie die Soll- und Ist-Drehzahl, so ergibt sich folgendes Diagramm:

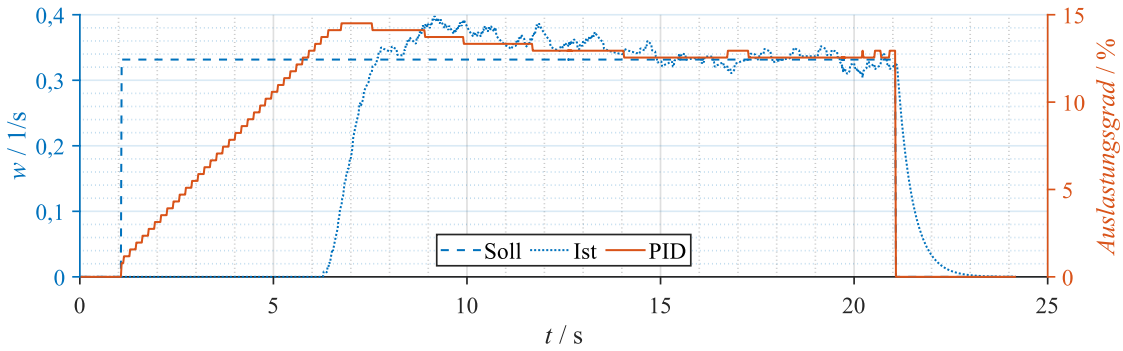

Abbildung 18: Totzeit PID-Analyse-PIA004

In Abbildung 18 ist zu erkennen, dass der Regler zwar direkt auf den geänderten Soll-Wert reagiert, der Motor aber erst ab etwa 12% Auslastungsgrad des Motorcontrollers zu drehen beginnt. Bis dieser Wert erreicht ist, erfordert es bei den aktuellen PID-Parametern eine Zeit von $t_{tot} = 5{,}22\text{s}$.

3.3.3 Test 3: x-Richtung bei konstanter Strecke mit variabler Geschwindigkeit

In einem Folgetest wird untersucht, ob die Zielgeschwindigkeit einen Einfluss auf die Totzeit hat. Hierzu werden verschiedene Geschwindigkeiten gewählt, um eine Strecke von $s_x = 1{,}00\text{m}$ zurück zu legen.

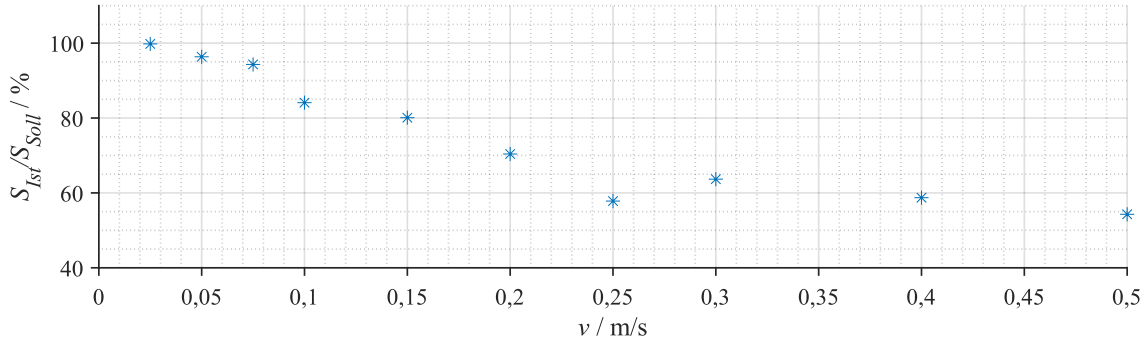

Abbildung 19: x-Richtung (sv-Diagramm)

Wäre die Totzeit unabhängig von der gesetzten Drehzahl, würde der Roboter schon bei einer Geschwindigkeit von $v = 0{,}2\,\frac{\text{m}}{\text{s}}$ keine Strecke mehr zurücklegen, weil die Fahrzeit für eine Strecke von einem Meter nur fünf Sekunden beträgt. Abbildung 19 zeigt jedoch, dass dies nicht der Fall ist.

Wäre die Totzeit direkt proportional zur Geschwindigkeit, würde eine konstante Strecke die Folge sein. Auch dies ist Abbildung 19 nicht zu entnehmen. Eine klare Gesetzmäßigkeit zwischen Totzeit und Geschwindigkeit ist aufgrund der erhobenen Daten nicht erkennbar.

Des Weiteren ist es theoretisch mit den erhobenen Daten möglich, eine Fahrtdauer für einen bestimmten Zielpunkt zu bestimmen. Dies soll aber nicht Ziel der Arbeit werden, da der Roboter in einem Schwarm eingesetzt werden soll und somit der Zeitpunkt des Erreichens einer bestimmten Koordinate durch andere Roboter beeinflusst werden kann.

Insgesamt zeigen alle Daten auf, dass der Roboter für Fahrten mit $v = 0{,}05\,\frac{\text{m}}{\text{s}}$ und $s_x = 1{,}00\text{m}$ zwar eine Strecke von einem Meter zurück legt, eine bestimmte Position kann aber nicht genau angesteuert werden. Hinzu kommt, dass dies bei abweichenden Parametern mitunter zu immensen Fehlern führt.

3.3.4 Test 4: y-Richtung bei konstanter Strecke und Geschwindigkeit

Der folgende Test soll das Fahrverhalten des Roboters in Y-Richtung untersuchen. Hierzu wird der Roboter mit einer Geschwindigkeit von $v_y = \pm 0{,}05 \frac{\mathrm{m}}{\mathrm{s}}$ für eine Zeit $t = 20\mathrm{s}$ fahren. Dies resultiert in einer Teststrecke von $S = 1{,}00\ \mathrm{m}$.

Bei diesem Test sind ähnliche Resultate zu erwarten wie zuvor in Kapitel 3.3.1, da die gleichen Motoren und Regelparameter auch für eine Fahrt in Y-Richtung verwendet werden. Somit ist hier das träge Verhalten nicht weiter zu beachten. Ebenso wird hier nicht weiter beleuchtet, was außerhalb der gewählten Parameter passiert, da hier ähnliche Effekte wie in Test 2 und 3 zu erwarten sind.

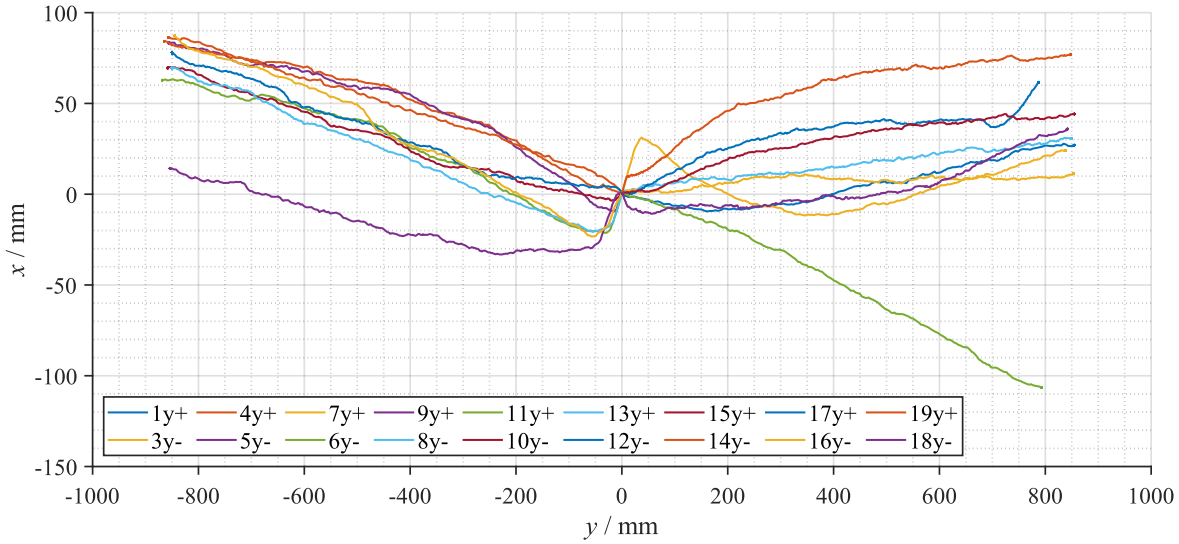

Abbildung 20: y-Richtung [s = konst.] (XY-Diagramm)

Alle Effekte aus Abbildung 15 sind auch in Abbildung 20 zu sehen. So Fährt der Roboter auch in y-Richtung nicht entlang einer Geraden. Jedoch hat die Streuung im Vergleich zur Fahrt in x-Richtung deutlich zugenommen. Dies hängt vermutlich damit zusammen, dass die Mecanumräder bei Fahrten in y-Richtung gegeneinander laufen (vergleiche Formel (1) aus Kapitel 2.1). Somit ist mehr Drehmoment erfordert, um den Roboter in Bewegung zu versetzten.

Weitere Auswertung werden an dieser Stelle bezüglich der Fahrt in y-Richtung nicht angestellt, da sie bezüglich der Arbeit nicht zielführend wären.

3.3.5 Test 5: Yaw-Stabilität

Der Test 5 beschäftigt sich mit der Rotation um die Z-Achse. Hierbei werden die Fahrtdaten aus Test 1 und Test 4 verwendet und analysiert, welche Drehung der Roboter über eine Strecke von $S = 1{,}00\text{m}$ vollführt.

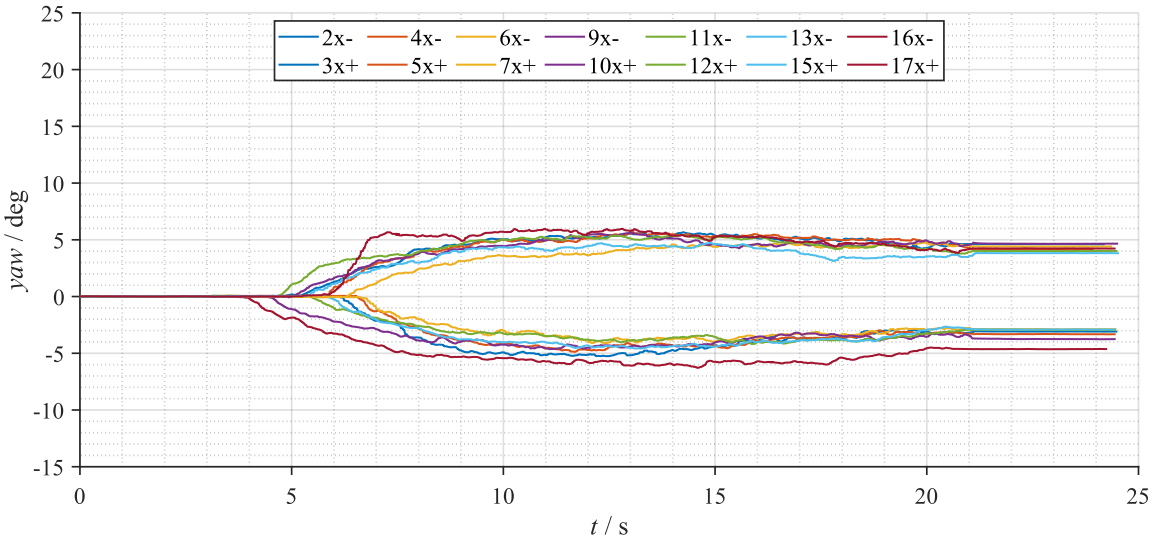

Abbildung 21: X-Richtung [S = konst.] (Winkel-Diagramm)

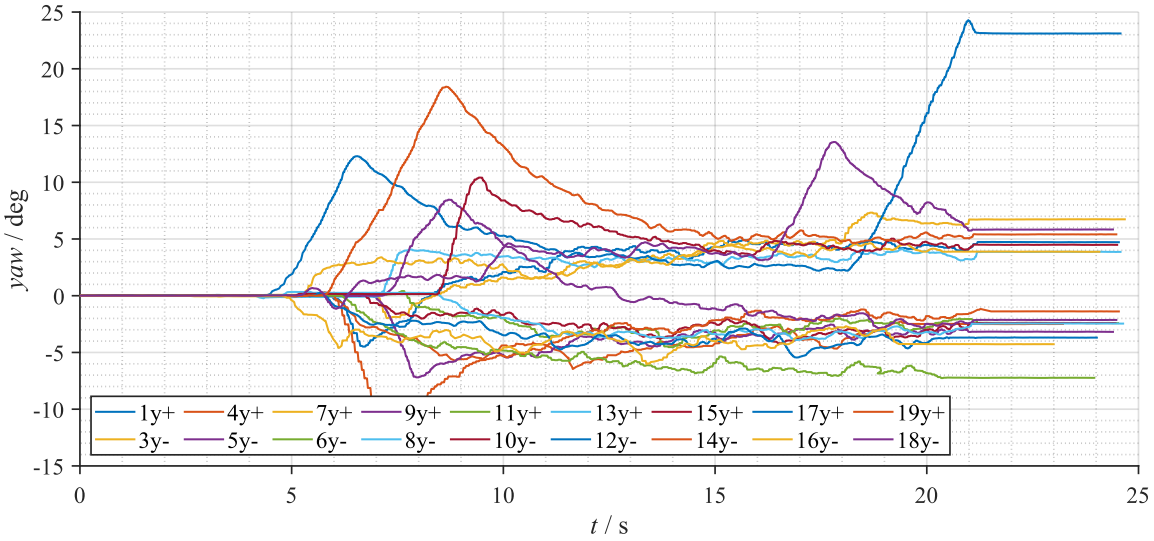

Abbildung 22: Y-Richtung [S = konst.] (Winkel-Diagramm)

Der größte Anteil der Orientierungsabweichung entsteht wie in Abbildung 21 und Abbildung 22 deutlich zu erkennen ist, direkt zum Beginn der Fahrten. Die Abweichungen sind bei Fahrten entlang der Y-Achse deutlich größer als bei Fahrten entlang der X-Achse. Dies wird besonders deutlich, betrachtet man die Soll- und Ist-Drehzahlen der Motoren, anhand der Fahrt 9.

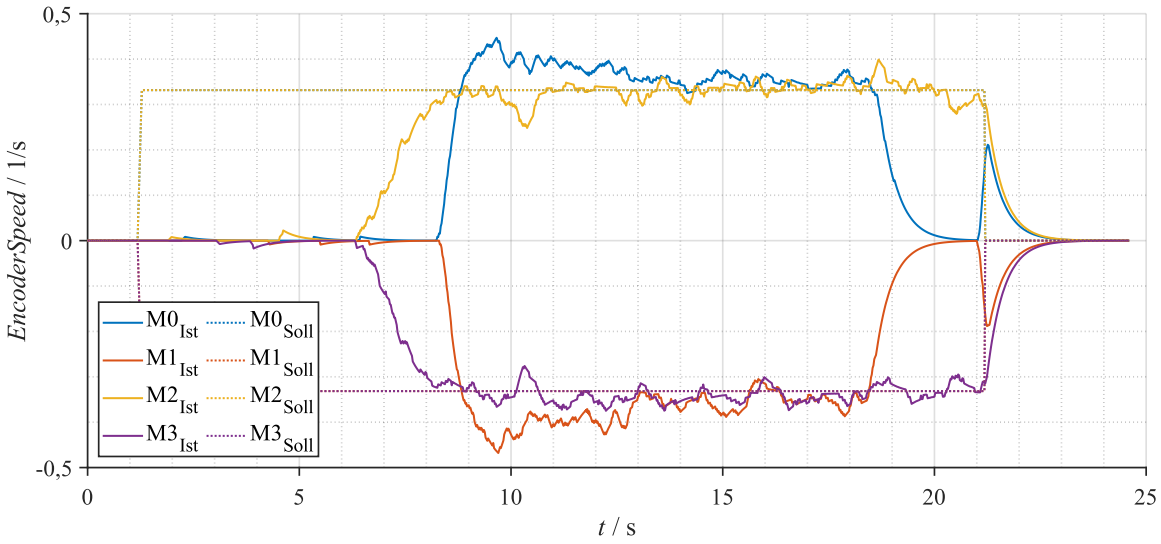

Abbildung 23: PID-Output (Fahrt y9)

In Abbildung 23 ist deutlich zu erkennen, dass der Zeitpunkt, zu welchem die Motoren sich zu drehen beginnen, paarweise stark abweichen können.

Warum die Motoren M0 und M1 im Zeitraum von ca. 18,5s - 20,0s stoppen kann damit erklärt werden, dass das erforderliche Moment zum Drehen der Räder nicht konstant ist. Da die Motoren M0 und M1 sich gegenseitig unterstützen, hat das Blockieren eines Rades zur Folge, dass der andere Motor diese Last mit übernehmen muss. Die Folge ist, dass beide Motoren stehen bleiben, bis der PID-Regler den Strom soweit erhöht hat, dass sich beide Motoren wieder drehen.

3.4 Konzeptbildung

In diesem Abschnitt sollen die Vor- und Nachteile verschiedener Konzepte diskutiert werden, um die bestmögliche Lösung für die oben erkannten Probleme und Anforderungen zu finden. Zum einen sollen drei Hardwarekonzepte hinsichtlich ihrer Vor- und Nachteile beurteilt werden.

Zum anderen sollen zwei Varianten zu Umsetzung einer Multi-Roboter-Steuerung gegenübergestellt werden. Hierbei soll besonders auf potenzielle Limitierungen des jeweiligen Konzepts geachtet werden.

In einem dritten Teil sollen Maßnahmen beschrieben werden, welche zur Verbesserung des Fahrverhaltens des Roboters beitragen sollen.

3.4.1 Hardwaredesign

Das Hardwaredesign des Roboters PIA004 besteht wie in Kapitel 3.1 zu sehen ist im Kern aus einem Arduino M0 Pro als Hauptrechneneinheit des Basis-Roboters, sowie viel Motoren mit quadraturencondern welche über zwei Qik-Motorcontroller von Pololu angetriebenwerden. Die Daten bekommt der Basisroboter entweder über eine serielle Verbindung oder über WLAN mittels eines XBees.

Da der Arduino M0 Pro nicht mehr bezogen werden kann, sind hier verschiedene Alternativen zum bestehenden Konzept aufgeführt und bewertet:

Variante 1a (2x Qik-MC, 1x Maple Mini, XBee):

In Variante a wird lediglich der Hauptcontroller (Arduino M0 Pro) durch einen Maple ersetzt. Dieser ist grundsätzlich ein STM32F103 und bietet ausreichend Schnittstellen und genug Speicher um das bestehende Projetzt für ihn an zu passen, und wäre vom Umfang der Änderungen eher gering. Der XBee stellt jedoch einen sehr hohen Kostenpunkt dar. Ebenso schränkt er die potentielle Erweiterbarkeit doch deutlich ein. Das Risko bei der Umsetzung ist relativ gering, da sich der Aufbau nur minimal ändert.

Variante 1b (2x Qik-MC, MCP23017, 1x ESP32):

In Variante b soll der Hauptcontroller (Arduino M0 Pro) sowie der XBee durch einen ESP32 ersetzt werden. Der ESP32 verfügt über eine interne WLAN Schnittstelle und benötigt somit kein Externes WLAN Modul, was vollen Zugriff auf die WLAN Schnittstelle zulässt sowie die Kosten drastisch senkt. Da der ESP32 aber für die vier Quadraturencoder alleine 8 seiner GPIOs verwenden muss, bleiben wenige GPIOs für andere Hardware wie beispielsweise Sensoren, weshalb seine GPIOs z.B. durch einen MCP23017 mittels der SPI- oder I2C-Schnittstelle um zwei 8bit-Register erweitert werden könnte. Diese wären nicht für Echtzeitanwendungen geeignet. Sie könnten jedoch Funktionalitäten übernehmen, welche dieses nicht erfodern,

beispielsweise debug-LED's betreiben oder ähnliches. Der Änderungsaufwand hierbei ist schon deutlich höher im Vergleich zur Variante a, da hierbei die gesamte XBee Kommunikation auf dem ESP32 umgesetzt werden muss. Ebenso muss der Code für alles, was mittels der MCP23017 angeschlossen wird, überarbeitet werden. Ein Risiko besteht hierbei im Programmieraufwand, um alle Funktionalitöten des alten Systems im neuen umzusetzen.

Variante 1c (2x Baby Orangutan, 1x ESP32):

In Variante c wird die Aufteilung der ersten Ebene des Roboters am stärksten verändert. So sind hier die meisten Anpassungen von Nöten. Hierbei werden Arduino M0 Pro und XBee durch den ESP32 ersetzt. Um sich aber nicht in den GPIOs zu limitieren, soll in dieser Variante ein anderer Motocontroller verwendet werden. Die hier verwendeten Motorcontroller sind die Baby Orangutan 328p von Pololu. Diese besitzen einen ATMEGA328p als Recheneinheit. Ebenso sind am Motorcontroller einige der GPIOs zugänglich, so dass jeder Motorcontroller auch die Quadraturencoder der angeschlossenen Motoren auslesen kann. Somit sind diese Motorcontroller in der Lage die komplette Closed-Loop-Radsteuerung zu gewährleisten. Ein großer Teil der zeitkritischen Aufgaben ist damit vom ESP32 auf die Motorcontroller übergegangen und ebenso sind 8 GPIOs am ESP32 weniger belegt. Der Änderungsaufwand in dieser Variante ist mit Abstand am größten, da hierbei nicht nur der ESP32 programmiert werden muss, sondern auch noch die Motorcontroller selbst. Diese Variante ist aber zum einen die günstigste Variante und zum anderen ist siezusätzlich am flexibelsten für potentielle zukünftige Änderungen. In dieser Variante bestehen Risiken darin, dass wie schon in Variante b der ESP32 die XBee Aufgaben übernehmen muss. Ebenso ist zu den Baby Orangutan nicht klar, auf der schwächeren Rechenhardware ein Closed-Loop-Regler zu realisieren ist.

Mit Hilfe einer Entscheidungsmatrix soll nun die beste Variante ermittelt werden. Hierbei werden den verschiedenen Varianten in vier Kategorien Schulnoten von 1-6 zu gewiesen. Die geringste Summe entspricht der umzusetzenden Variante.

Tabelle 1: Entscheidungsmatrix Hardware Design

	Variante 1a	*Variante 1b*	*Variante 1c*
Kosten	4	3	1
Erweiterbarkeit	3	3	1
Änderungsaufwand	1	2	3
Risiko	2	3	3
Summe	10	11	8

Somit ist nach Tabelle 1 die **Variante 1c** umzusetzen. Hierbei sollte aber schon früh abgeschätzt werden, ob die Risiken beherrschbar sind.

3.4.2 Multi-Roboter-Steuerung

In diesem Abschnitt sollen zwei Varianten diskutiert werden, mit deren Hilfe es möglich sein soll, mehrere Roboter zeitgleich individuell anzusteuern. Hierzu werden die beiden Varianten bezüglich ihrer Vor- und Nachteile gegenübergestellt.

Variante 2a: UDP-Pakete als Unicast

Wie in Kapitel 3.2 zu sehen, senden die XBees, so wie sie konfiguriert sind, ihre Daten immer als UDP-Broadcast. Somit erhalten alle XBees im WLAN dieselben Daten. Ein Lösungsansatz wäre also, statt des XBee-Senders einen UDP-Socket zu verwenden. Diesen kann man individuell konfigurieren und somit ist es möglich, verschiedenen IPs unterschiedliche Daten zu senden. Da die Roboter alle eine eigene IP bzw. sogar mehrere individuelle IPs besitzen, wäre es somit möglich, jedem Roboter individuelle Daten zu schicken.

Ein Nachteil bei dieser Variante besteht darin, dass sollte ein Roboter nur mit einem XBee ausgestattet sein, wäre er nicht in der Lage, selbst Nachrichten an bestimmte Ziel-Systeme zu schicken, könnte aber individuelle Daten empfangen. Da die neuen Roboter jedoch alle mit einem ESP32 ausgestattet werden, ist dieser Nachteil tatsächlich nicht so gravierend, da alte Roboter weiterhin im Schwarm mitfahren könnten, den Schwarm aber nicht koordinieren könnten.

Maßnahmen zur Umsetzung wären hierbei, dass der ESP angepasst werden muss, um anderen Robotern Befehle zu schicken. Ebenso muss im HOOU-Package ein neues Interface erstellt werden, welches in der Lage ist mit den Robotern mittels UDP-Socket zu kommunizieren.

Variante 2b: Erweiterung der HOOU-Pakete um eine Roboter-ID

In dieser Variante wird allen HOOU-Paketen eine Ziel- und Quell-ID beigefügt werden. Somit ist eine eindeutige Zuweisung möglich und es können auch individuelle Antworten gegeben werden. Da alle Pakete in dieser Variante weiter als Broadcast versendet werden, entsteht somit ein sehr hohes Datenaufkommen, was zu Schwierigkeiten führen kann. Da die Paketgröße festgelegt ist, wäre in dieser Variante auch keine Abwärtskompatibilität möglich, es sein denn, alle älteren Roboter würden mit einer neuen Firmware ausgestattet werden.

Ein weiteres Problem besteht darin, dass sich in dieser Variante Daten beim Empfangen mittels eines XBees überschneiden können, da es nur einen Buffer gibt, welcher von allen IPs Daten empfängt. Nimmt man nun viele verschiedene Quellen an, kann es passieren, dass ein HOOU-Paket, welches vom XBee auf zwei UDP-Pakete aufgeteilt wurde, nicht mehr korrekt zusammengesetzt wird. Somit wäre eine Zuweisung nicht mehr möglich.

Im Folgenden werden Vor- und Nachteile der beiden Varianten aufgelistet, um die beste Umsetzung zu ermitteln.

Tabelle 2: Abwieglung Vor- und Nachteile zweier Varianten zur Umsetzung einer Multi-Roboter-Steuerung

Variante 2a: UDP-Pakete als Unicast	*Variante 2b: HOOU-Paket Erweiterung*
+ Weniger Broadcast-Traffic + Abwärtskompatibilität, keine Änderung am HOOU-Protokoll + Weniger Kosten, ein XBee kann eingespart werden. - PC muss sich im gleichen Netzwerk befinden - Roboter-zu-Roboter-Kommunikation mit XBee schwer um zu setzten	+ Roboter-zu-Roboter-Kommunikation nur mit XBees möglich - Hoher Broadcast-Traffic - viele Rechenoperationen auf Microcontroller - evtl. Datenkollision

Nach Abwägung der Vor- und Nachteile nach Tabelle 2 ist im Folgenden die **Variante 2a** um zu setzten. Somit können die HOOU-Pakete wie bisher weiter bestehen und da der XBee durch einen ESP32 ersetzt wird, ist auch die gerichtete Roboter-zu-Roboter-Kommunikation möglich.

3.4.3 Auswertung des Fahrverhaltens von PIA004

In Kapitel 3.3 wird das Fahrverhalten des Roboters PIA004 dargestellt. Ein Hauptproblem bei den Fahrten besteht darin, dass eine hohe Totzeit zu Beginn jeder Fahrt besteht und der PID-Regler viel zu langsam auf Änderungen reagiert. Diese soll bei der Überarbeitung der Motorregelung mittels dreier Maßnahmen verbessert werden. Zum einen kann das Auslesen der Quadraturencoder weniger fehleranfällig gestaltet werden und zum anderen wird die Auflösung um den Faktor 4 vergrößert. Dies sorgt für eine genauere Bestimmung der Geschwindigkeit. Ebenso soll die PID-Looptime deutlich reduziert werden. Aktuell beträgt die PID-Frequenz $f_{PID,PIA004} = 100\,\mathrm{HZ}$ und diese soll deutlich erhöht werden. Um auf große Änderungen schneller reagieren zu können, soll ebenso eine Vorsteuerung umgesetzt werden. Diese soll die mittlere anzunehmende Last vorsteuern. Der PID-Regler muss somit lediglich kleinere Abweichungen vom Mittelwert kompensieren, welche durch äußere Einflüsse entstehen.

Eine weitere Maßnahme soll sein, dass ein Regelkreis hinzugefügt werden soll, welcher die im Roboter verbaute IMU verwendet, um die Fahrtrichtung des Roboters zu korrigieren. Ein Großteil der Abweichungen bei längeren Fahrten rührt daher, dass der Roboter sich beim Anfahren doch deutlich dreht. Dieser Winkelfehler lässt den Roboter dann über die gesamte Fahrt in eine falsche Richtung fahren. Kompensiert man das jedoch mit Hilfe der IMU, sind bessere Ergebnisse zu erwarten.

4 Umsetzung

Im Folgenden sollen alle Konzepte umgesetzt werden, für die sich in Kapitel 3.4 entschieden wurde. Da eine Änderung des Hardwaredesigns auch einen Einfluss auf die Programmierung hat, soll mit diesem Abschnitt begonnen werden. Der hierbei entstehende Roboter stellt die Version 6 des Pia-Roboters dar, weshalb er im Weiteren nur noch mit PIA6 bezeichnet wird.

4.1 Entwicklung einzelner Teilsysteme

Da die Entwicklung von PIA6 in vielen Bereichen doch gravierende Änderungen im Systemdesign sowie in der Programmierung beinhaltet, müssen Teilkomponenten im Vorfeld designt bzw. programmiert werden. Einzelne zu untersuchende Teilsysteme bilden hierbei:

- Umsetzung einer ISP-Lösung zum FOTA-Flashen der Motorcontroller-Firmware
- Programmierung einer Closed-Loop-Motorcrontroller-Firmware für Baby Orangutan Motorcontroller von Pololu inclusive der Implementierung einer Vorsteuerung
- Portierung des gesamten HOOU-Robot Projekts von der SAMD-Architektur auf die ESP32-Architektur

Sind die Teilsysteme für sich funktionstüchtig, werden Sie zu einem Komplettsystem kombiniert.

4.1.1 ISP-Programmierung (FOTA)

Bevor eine Firmware für die Motorcontroller entwickelt werden kann, muss ein Weg gefunden werden diese zu programmieren. Dem Datenblatt des Pololu Baby Orangutan [11] ist zu entnehmen, dass auf dem Motorcontroller ein Atmel Mega328p [12] (ATMega328p), so wie eine Dual DC-Motor Treiber von der Firma Toshiba mit der Bezeichnung TB6612FNG [13] verbaut sind. Der verbaute Microcontroller findet sich auch in vielen Arduinos wieder. So z.B. auf dem Arduino UNO, Arduino Nano und dem Arduino Pro Mini. Diese Arduinos können in der Regel über USB oder mit Hilfe eines separaten USB-zu-Seriell-Adapters programmiert werden, was jedoch einen vorinstallierten Bootloader erfordert. Bei dem Motorcontroller, welcher ohne Firmware und Bootloader ausgeliefert wird, ist einzig die Programmierung über einen ISP-Programmer vorgesehen.

Auf jedem Roboter der Version sechs sollen zwei dieser Motorcontroller verbaut werden. Jeden dieser Controller muss man einzeln programmieren. Dies bedeutet, dass für jede Änderung an der Motorcontroller Firmware zunächst ein ISP-Programmer verbunden werden muss, bevor die Firmware auf den Motorcontroller gespielt werden kann. ISP-Programmer können käuflich erworben werden. Es ist jedoch auch möglich einen anderen Arduino mit dem Beispiel ArduinoISP aus dem ArduinoIDE zu bespielen und diesen als ISP-Programmer zu verwenden. Um sich mit der Materie vertraut zu machen, wird dieses in einem ersten Schritt versucht. Ein ArduinoUNO soll mittels eines anderen mit dem Beispiel Blink bespielt werden. Folgt man den

Anweisungen des Beispiels, ist dies auch ohne Probleme möglich. Dies bietet also bereits eine Möglichkeit die Motorcontroller mit einer Firmware zu programmieren. Da eine Anforderung an den neuen PIA-Roboter jedoch ist, dass er fernwartbar ist, muss ein Weg gefunden werden, dieses Vorgehen auch über eine drahtlose Verbindung zu ermöglichen.

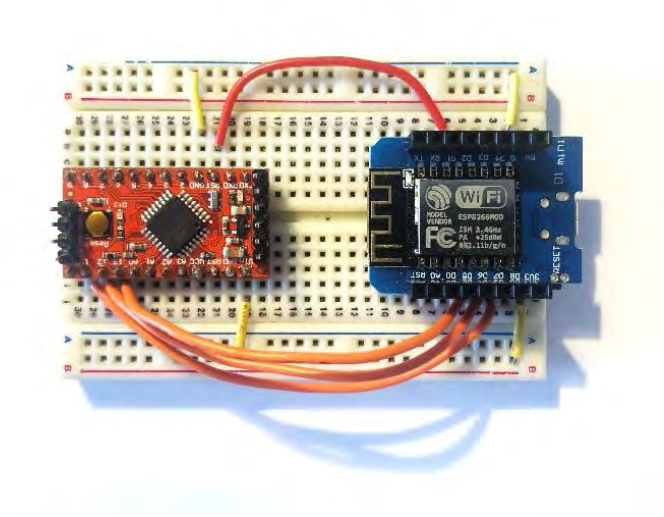

Abbildung 24: ESP8266 an 3v3-ATMega328p um mittels ISP-Programmer den Arduino mit einer neuen Firmware zu programmieren

Ein interessantes Projekt hierbei stellt das „ESP8266AVRISP“ dar. Hierbei wird ein ESP8266 als ISP-Programmer programmiert. Dies ermöglicht es, mit Hilfe des Programms AVRDUDE einen AVR-Microcontroller zu programmieren. Bei dem durchgeführten Test wird ein WeMOS D1, welcher einen ESP8266 Microcontroller besitzt, sowie einen Arduino Pro Mini verwendet, da dieser genauso wie der ESP8266 auf 3,3V läuft. Folgt man den Anweisungen ist es so unter Linux möglich, den Arduino Pro Mini über das WLAN zu programmieren. Dieses ist aus Windows jedoch ist es nicht möglich.

Da eine Programmierung aber möglichst aus jedem Betriebssystem heraus möglich sein soll, reicht es nicht aus, dieses Projekt an die ESP32 Hardware an zu passen.

Eine weitere Hürde die Überwunden werden muss, ist die Tatsache, dass der ESP32 ebenso wie der ESP8266 keine 5V toleranten Pins hat.

Aus diesem Grund muss ein Weg gefunden werden, um den ESP vor Spannungen größer 3,3V zu schützen. Dies darf aber nicht dazu führen, dass der ATMega328p eine logische 1 nicht mehr erkennt. So muss die Spannung für eine logische 0 im Bereich $-0{,}5\text{V} \leq V_{IL} \leq 1{,}5\text{V}$ liegen. Für eine logische 1 liegt dieser Bereich bei $3{,}0\text{V} \leq V_{IH} \leq 5{,}5\text{V}$ ist der ATMega328p mit einer Spannung von $V_{CC} = 5\text{V}$ betrieben [12]. Da der ESP8266 sowie der ESP32 bei einer logischen 1 die Spannung am gewählten Pin auf $V_{ESP,1} = 3{,}3\text{V}$ und bei einer logischen 0 auf $V_{ESP,0} = 0\text{V}$ setzt, muss lediglich verhindert werden, dass die maximale Spannung überschritten wird. Dies ist am einfachsten mit folgender Schaltung zu gewährleisten:

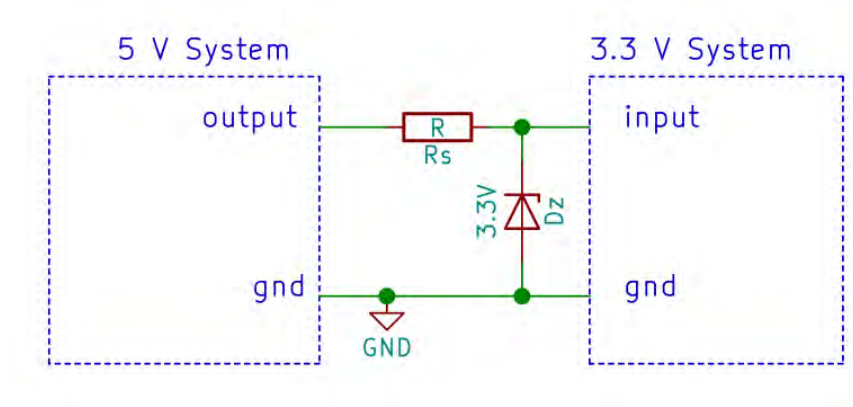

Abbildung 25: Pegelwandler [14]

Als Widerstand wird hier eine Größe von $R_s = 220\ \Omega$ gewählt und soll den Verluststrom über die Zehnerdiode reduzieren. Bei richtiger Konfiguration des ATMega328p ist diese Schaltung nur für die Pins RST und MISO nötig, da die anderen Pins nur als Input fungieren sollten. Da diese Pins im Betrieb aber andere Zustände annehmen könnten, werden aus Sicherheitsgründen auch MOSI und SCK eine solche Schaltung erhalten.

Mit dieser Schaltung sollte es im Weiteren also möglich sein, auch 5V Microcontroller mittels eines ESP8266 zu programmieren, was es zu untersuchen gilt. Hierzu wird der im ersten Versuch verwendete Arduino UNO mit Hilfe des Pegelwandlers und eines WeMOS D1 programmiert.

Da zu diesem Zeitpunkt klar ist, dass es möglich ist, mit einem 3,3V ISP-Programmer einen 5V Microcontroller zu programmieren, soll in einem nächsten Schritt eine passende Firmware für den ESP32 geschrieben werden. Zunächst muss eine Lösung gefunden werden, die es auch möglich macht, den ESP als OTA-ISP-Programmer aus Windows heraus zu verwenden. Die Software AVRDUDE ist zwar Open Source und somit wäre es potenziell möglich den Fehler zu beheben, dies würde aber den zeitlichen Rahmen der Bachelorarbeit sprengen, weshalb nach einer Alternative gesucht werden muss. Der hier gewählte Ansatz sieht vor, dass die Firmware mittels ArduinoIDE in eine *.hex Datei nach dem Intel Hex Format [15] exportiert wird. Diese kann dann über einen auf dem ESP32 laufenden Webserver in seinen Speicher geladen werden. Sobald die Datei auf dem ESP32 liegt, wird diese ausgelesen, in binäre Daten umgewandelt und mittels der ISP-Schnittstelle hochgeladen.

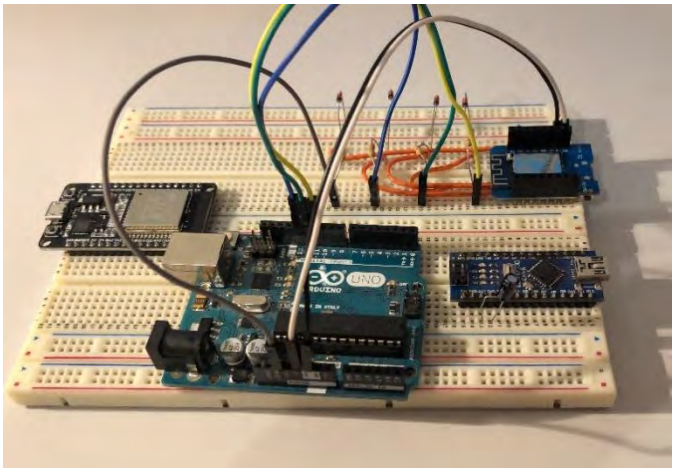

Abbildung 26: ESP8266 ISP-Verbindung zu einem ATMega328p inclusive 3,3V Schutzschaltung

Dies bietet die Möglichkeit der Fernwartung, ist nahezu von jedem PC und sogar von Tablets aus möglich und erfordert keine weiteren Tools als das ArduinoIDE.

Als Ausgangspunkt hierfür dient der Beispielsketch FSBrowser.ino der Webserverbibliothek. Hier sind grundlegende benötigte Funktionen bereits konfiguriert und aktiviert. So ist es bereits mit diesem Sketch möglich ein Dateisystem zu initialisieren sowie Daten hoch- und runterzuladen. Es wird die Funktion *handleFileuploadHex* angelegt. Diese sorgt seitens des Webservers dafür, dass eine *.hex Datei hochgeladen werden kann. Diese Datei erhält intern immer den Namen FW.hex. Sobald sie vollständig hochgeladen ist, soll eine weitere Funktion namens *doFlash* aufgerufen werden. Diese sorgt dafür, dass die binären Daten der Datei FW.hex extrahiert und dann auf den ATMega328p hochgeladen werden.

Um die Funktion *doFlash* programmieren zu können ist es von Nöten zu verstehen, wie ein ISP-Programmer funktioniert. Hierzu wird der Quellcode des ESP8266 so manipuliert, dass die SPI-Befehle, welcher der ESP an den ATMega328p sendet ebenso über die serielle Schnittstelle ausgegeben werden. Hier fällt schon auf, dass immer Gruppen von vier Bytes an den ATMega328p versendet werden. Dieser antwortet je nach Befehl auf dem dritten oder vierten gesendeten Byte. Ebenso muss der ATMega328p zur Programmierung resettet werden. Dies passiert dadurch, dass der Reset-Pin auf LOW gezogen wird.

Eine Programmiersequenz beginnt bei einem ATMega328p dann mit folgenden 4 Bytes:

```
0xAC  0x53  0x00  0x00
```

Zieht man das ATMega328p Datenblatt heran, erklärt Kapitel *27.8.3 Serial Programming Instruction set*, dass dieser Befehl den ATMega328p in den Programmingmodus versetzt. Antwortet der ATMega328p auf diesen Befehl mit `0x53` auf dem dritten Byte war die Aktion erfolgreich. Tut er dies nicht, muss er erneut dazu resettet werden. Dieses Vorgehen, wird in der Funktion *setProgMode* umgesetzt.

Als nächstes werden drei Befehle geschickt:

```
0x30  0x00  0x00  0x00

0x30  0x00  0x01  0x00

0x30  0x00  0x02  0x00
```

Auf diese antwortet der ATMega328p mit: `0x1E 0x95 0x0F` jeweils auf dem vierten Byte des gesendeten Befehls. Diese Sequenz liest die Signatur des Microcontrollers aus. Diese entspricht der im ATmega328p Datenblatt angegebenen Signatur. Auch hierfür wird eine Funktion namens *isValidChipSig* angelegt, welche überprüft, ob die Signatur mit einer angegebenen Übereinstimmt.

Als nächstes wird der aktuelle Programmspeicher des ATMega328p gelöscht. Dies geschieht mit dem Befehl:

```
0xAC  0x80  0x00  0x00
```

Dieses Vorgehen erhält die Funktion *eraseFlash*.

Daraufhin wird der Microcontroller resettet und erneut die Signatur ausgelesen. Danach beginnt der eigentliche Programmiervorgang. Um diesen Vorgang besser zu verstehen muss hierzu die

Speicherorganisation des ATMega328p verstanden werden. Der Speicher wird mittels WORDs und PAGEs organisiert. So besteht jedes WORD aus zwei Bytes, eines davon wird Most-Significant-Byte (MSB) genannt, das andere Least-Significant-Byte (LSB). Eine PAGE setzt sich aus 64 dieser WORDs zusammen. Und der gesamte Flash-Speicher umfasst 256 PAGEs. Somit ergibt sich eine Gesamtgrößte des Flashspeichers von $2 * 64 * 256\ \text{B} = 31768\ \text{B}$.

Beim Programmiervorgang werden nacheinander alle WORDs einer PAGE in den Speicher des Microcontrollers gelegt. Dies geschieht in den mit den Befehlen:

```
0x48  adrMSB adrLSB MSB
```

```
0x40  adrMSB adrLSB LSB
```

`adrMSB` beschreibt hierbei die oberen 8 Bit der Adresse und `adrLSB` die unteren 8 Bit der 16 Bit Adresse. Diese Adresse beginnt bei 0 und zählt für jedes Wort um einen nach oben.

Die Daten, welche das MSB und das LSB bilden, sind wie zuvor beschrieben in der FW.hex im Speicher des ESP32 hinterlegt. Die Besonderheit einer HEX-Datei besteht darin, dass eigentlich binäre Daten auf eine lesbare Weise als Textdatei dargestellt werden. Bytes, welche außerhalb des lesbaren ASCII Bereichs liegen, können in diesem Format dargestellt werden, weil die hexadezimale Dartellung als Text (zwei Zeichen [0-9,A-F]) verwendet wird [16]. Somit macht eine HEX-Datei binäre Daten für den Menschen lesbar, muss aber zur weiteren maschinellen Verwendung wieder für den Computer verständlich gemacht werden. Eine HEX-Datei bietet noch weitere Vorteile, so gibt es z.B. eine Checksumme, welche es ermöglicht zu überprüfen ob es sich um einen validen Datensatz handelt. Die genaue Funktionsweise soll hier aber nicht weiter diskutiert werden.

Das Hochzählen, die Adressierung und das Hochladen werden in der Funktion *flashNextTupel* umgesetzt.

Sind 64 WORDs einer Seite geladen, muss diese in den Programmspeicher geschrieben werden. Hierfür wird die Funktion *incPage* angelegt. Der ausgeführte Befehl lautet hier:

```
0x4C adrMSB adrLSB 0x00
```

Die Adressen `adrMSB` und `adrLSB` sind die gleichen, welche zuvor für das Schreiben des letzten Words verwendet wurden. Sie lassen sich aber auch berechnen, wenn man die PAGEs zählt und man diese mit ihrer Größe multipliziert.

Ist so die PAGE in den Speicher geschrieben worden, wird die nächste PAGE geladen, bis der gesamte Programmcode aus der HEX-Datei hochgeladen wurde. Da die letzte PAGE nicht zwangsweise vollständig mit Daten befüllt werden kann, müssen die letzten freien WORDs noch mit dem Wert `0xFF` beschrieben werden.

Das Programmieren ist abgeschlossen, sobald die letzte PAGE, welche Programmcode enthält in den Speicher geschrieben wird. Nachdem dies geschehen ist, kann die SPI-Schnittstelle deaktiviert werden. Ebenso wird der Reset-Pin wieder auf HIGH gesetzt. Ab diesem Zeitpunkt läuft der ATMega328p bereits mit seiner neuen Firmware. Das Ergebnis hieraus ist der Sourcecode der Bibliothek *OS_FS_AVRISP*.

Nachdem der Programmablauf geklärt ist, muss noch entschieden werden welche Peripherie am ESP32 verwendet werden soll.

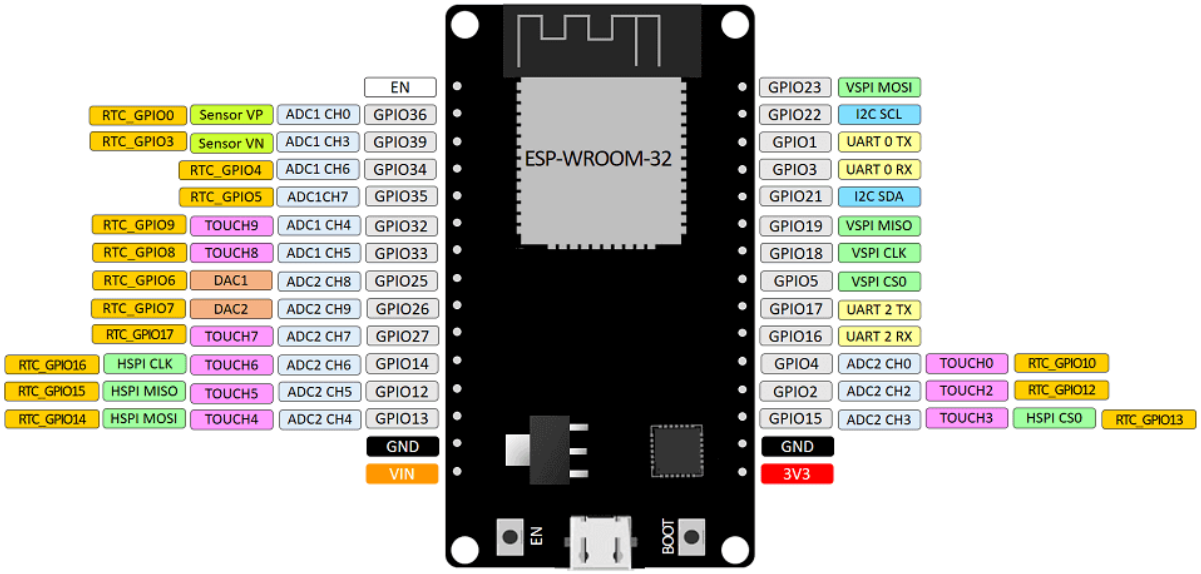

Abbildung 27: ESP32-Pinout [17]

Wie in Abbildung 27 zu sehen ist, verfügt der ESP32 über zwei Hardware SPI-Schnittstellen. Die SPI-Library des ESP32 verwendet standartmäßig die HSPI-Schnittstelle. Somit werden auch für den ISP-Programmer jene Pins gewählt, welche zur HSPI-Schnittstelle gehören. Demzufolge sind dies GPIO12 – GPIO14. Der GPIO15 wird nicht verwendet, da der „Chip Select“-Pin (CS0) nicht benötigt wird. Anders als bei einer Standard SPI Verbindung, wo über den „Chip Select“-Pin das Zielgerät ausgewählt wird, muss beim Programmieren eines AVR-Microcontrollers der Reset-Pin für die Dauer des Programmiervorgangs auf LOW gezogen werden. Da das Ziel des Projekts ist, gleich zwei Motorcontroller mit verschiedener Firmware zu programmieren, ohne sie dafür aus dem System entfernen zu müssen, muss als nächstes ein Weg gefunden werden diese separat voneinander zu programmieren. Hierzu werden die beiden Reset-Pins der Ziel-Microcontroller mit zwei verschiedenen GPIOs des ESP32 verbunden. Der Reset-Pin von Motorcontroller 0 (MC0) erhält dabei den GPIO27 und der Motrocontroller 1 (MC1) erhält den GPIO33. Im Folgenden nun eine Verbindungstabelle:

Tabelle 3: ESP32-Baby Orangutan-Verbindungstabelle

Typ	ESP32	MC0	MC1
MISO	GPIO12	PB4	PB4
MOSI	GPIO13	PB3	PB3
SCK	GPIO14	PB5	PB5
Reset (M0)	GPIO27	PC6	
Reset (M1)	GPIO33		PC6

An dieser Stelle gibt es einen weiteren Aspekt, welcher betrachtet werden muss. So ist auf den Baby Orangutan Motorcontrollern der Pin PB3 mehrfach verwendet. Er dient nicht nur zum Programmieren, sondern ebenso der Ausgabe des PWM-Signals zur Steuerung der H-Brücke. Dies wird mit folgender Schaltung umgesetzt.

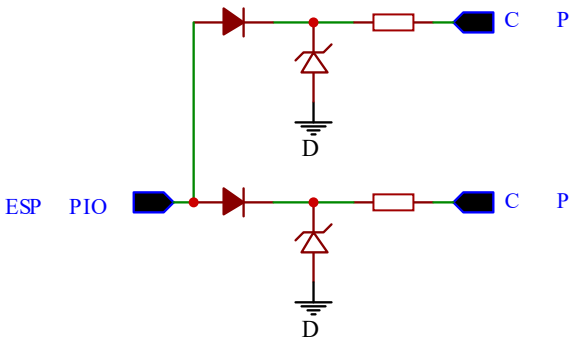

Abbildung 28: Trennung-PB3

PWM-Signale des Pins PB3 treiben somit nicht mehr die H-Brücke des jeweils anderen Motorcontroller. Ebenso bietet diese Schaltung einen Schutz für den ESP32 gegenüber des 5V Ausgangssignals. Nach der Umsetzung der Software kann das gesamte Teilsystem getestet werden, hierfür wird die Schaltung auf einem Breadboard gesteckt und es werden verschiedene Beispiele auf die auf die Testcontroller geladen. Der Prozess des Hochladens einer Firmeware mittels Webinterface wird im Folgenden mit Firmware-Over-The-Air (FOTA) bezeichnet.

Nachdem dieses erfolgreich funktioniert, kann sich der Programmierung des Closed-Loop-Motorcontrollers zugewandt werden.

4.1.2 Closed-Loop-Motorcontroller-Firmware

Die Closed-Loop-Motorregelung ist im Grunde keine Neuerung für den PIA-Roboter. Hier soll also im Folgenden lediglich erläutert werden, welche Änderungen mit welchem Ziel vorgenommen werden.

Im ersten Schritt werden Aspekte bezüglich des Einlesens der Quadraturencoder diskutiert. Es gibt mehrere Möglichkeiten den Zustand des jeweiligen Encoder Kanals zu bestimmen. Zum einen kann der Zustand mittels der Funktion *digitalRead* eingelesen werden, zum anderen ist es möglich ein komplettes Register in ein Byte zu lesen. Dieses beinhaltet dann die Pin-Zustände aller auf dem Register befindlichen Pins. Nachteil hiervon ist aber, dass der Quellcode mitunter schwieriger zu lesen ist.

Da eine maximale Performance aus dem Motorcontroller geholt werden soll, werden diese beiden Methoden hinsichtlich ihrer Geschwindigkeit untersucht. So wird ein Pin 100000 Mal ausgelesen und in einer Variablen abgelegt. Hierfür wird die benötigte Zeit ermittelt. Um eine bessere Vergleichbarkeit zu haben, wird beim Registerauslesen dieses auch noch so verarbeitet, dass nur noch der Wert eines Pins in der Variablen steht. Die Funktion *digitalRead* benötigt für das Einlesen etwa $t_{digitalRead} = 233\ \text{ms}$, liest man einen Pin jedoch direkt aus dem Register, so benötigt man für dieselbe Anzahl nur etwa $t_{Register} = 44\ \text{ms}$. Somit ist das Einlesen der Pins um den Faktor fünf schneller, liest man die Pins direkt aus dem Register. Dieses Verhältnis wird sogar noch besser, legt man alle einzulesenden Pins auf dasselbe Register, hierdurch ist das Einlesen mittels Register in unserem Fall sogar mehr als 20mal so schnell. Auf Grund dieser Erkenntnis wird auf die Verwendung von digitalRead verzichtet und die etwas kryptischere Schreibweise in Kauf genommen.

Als nächstes muss untersucht werden, wieso es in der aktuellen PIA-Software gelegentlich zu Lesefehlern beim Auslesen des Quadraturencoders kommt.

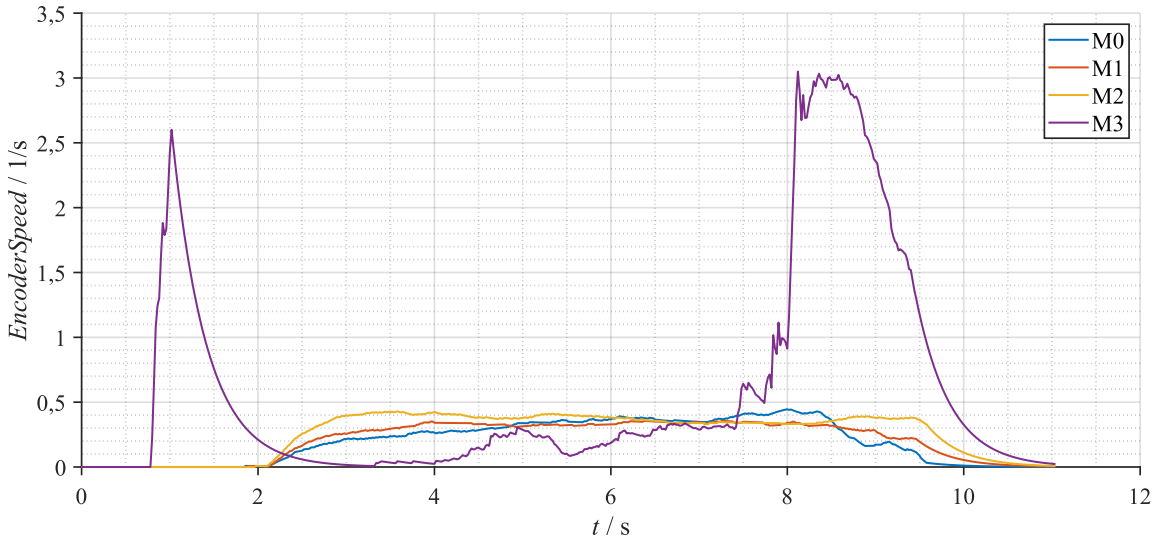

Abbildung 29:PIA004 Encoder-Lesefehler

Wie in Abbildung 29 zu sehen, kann es mitunter zu gravierenden Lesefehlern kommen, was zur Folge hat, dass der Roboter ein unwillkürliches Verhalten an den Tag legt. Um der Ursache hierfür auf den Grund zu gehen wird die aktuelle Methode etwas genauer beleuchtet und überprüft werden, ob es weniger fehleranfällige Alternativen gibt.

Aktuell wird an jeder steigenden Flanke des Kanal A überprüft, welchen Zustand der Kanal B hat. Ist Kanal B LOW, wird die Encoder Position um eins reduziert. Ist er HIGH, wird die Position um eins erhöht.

Eine andere, bessere Variante ist es aber, überprüft man jeden Wechsel jedes Kanals [18]. Also nicht nur den Wechsel von LOW zu HIGH auf Kanal A, sondern auch den Wechsel von HIGH zu LOW. Und dies tut man auch auf dem Kanal B. Zum einen erhält man so eine vier Mal so hohe Auflösung der aktuellen Encoder Position, zum anderen kann sogenanntes Pendeln nicht zu Fehlern führen. Pendelt der Kanal A zwischen dem Zustand LOW und HIGH, weil der Magnet gerade exakt auf der Grenze zwischen HIGH und LOW steht, zählt die einfache Logik, wie auf PIA004 umgesetzt, für jedes Pendeln je nach Zustand von Kanal B um einen hoch bzw. runter. Dies führt mitunter zu den in Abbildung 29 zu sehenden Effekten.

Auf den ersten Blick hat Variante zwei zwar den Nachteil, dass man vier Mal so oft überprüfen muss, ob sich ein Pin-Zustand geändert hat. Bei näherer Betrachtung stellt man aber fest, dass auch in Variante eins im gleichen Intervall überprüft werden muss, da es sonst Lesefehlern kommen kann, wenn der Kanal B schon nicht mehr den Zustand hat, welchen er beim Wechsel des Kanals A von LOW zu HIGH hatte.

Der aktuelle Code zum Auslesen des Encoders wird durch einen so genanntes Interrupt angestoßen. Dies ist empfehlenswert, da ein Interrupt die Ausführung des Main-Loops unterbrechen kann. Somit wird der Encoder in einem konstanten Intervall ausgelesen, auch wenn das Hauptprogramm sich grade in einer zeitintensiven Operation befindet.

	x_1	A	B	x_2
t_n	–	0	0	–
t_{n+1}	0	0	1	1
t_{n+2}	0	1	1	1
t_{n+3}	0	1	0	1
t_{n+4}	0	0	0	1

Abbildung 30: Auswertungsbeispiel für Quadraturencoder

Da die Logik es Auslesens der Encoder auf Bitebene einige Optimierungen beinhaltet und dadurch etwas schwerer zu lesen ist, soll an dieser Stelle mithilfe einer Logiktabelle der Algorithmus erklärt werden.

Abbildung 30 zeigt ein Beispiel dafür, wie mittels einer XOR-Operation die Drehrichtung eines Quadraturencoders ermittelt werden kann. Hierzu werden exemplarisch die verschiedenen Encoderschritte einer Drehrichtung zeilenweise nacheinander aufgeführt. Zur Auswertung werden die charakteristischen Werte x_1 und x_2 brechnet. Dies geschieht mit Hilfe der Formeln (21) und (22). Für eine positive Drehrichtung ist somit $x_1 = 0$ und $x_2 = 1$. In der anderen Drehrichtung ist dies genau umgekehrt. Ändert sich hingegen nichts, gilt $x_1 = x_2$. Somit reichen zwei Bedingungen aus, um alle acht Möglichkeiten ab zu decken.

$$x_1 = A_{t_{n+1}} \oplus B_{t_n} \quad (21)$$

$$x_2 = A_{t_n} \oplus B_{t_{n+1}} \quad (22)$$

Die gewählte Formulierung bedingt jedoch, dass Pins für einen Encoder so gewählt werden müssen, dass sie im Register nebeneinander liegen. Dies hat den Vorteil. Dass nur mit wenigen Rechenoperationen bis zu vier Encoder auf einem 8-bit Register ausgewertet werden können.

Um möglichst effizient die Quadraturencoder aus zu lesen, werden deshalb die Encoder an dem Baby Orangutan Board auf die PC0-PC3 gelegt.

Weitere Änderungen sehen zum einen eine deutlich reduzierte „Looptime“ vor. So berechnet der neue Motorregler die PID-Werte mit einer Frequenz von 1000 Hz statt zuvor mit 100 Hz. Somit sollte der Regler im Stande sein auf Änderungen schneller zu reagieren. Ebenso sollten die Änderungen der Werte zwischen zwei Berechnungen kleiner ausfallen, dadurch soll erreicht werden, dass sich der Regelkreis nicht so schnell aufschwingt.

Dies hat jedoch zu zur Folge, dass die Auflösung beim Einlesen des Quadraturencoders sinkt. So ist die kleinste zu messender Drehzahl:

$$\omega_{min} = 1 \cdot \frac{1}{f_{Update}} \cdot \frac{1}{n_{Getriebe} \cdot n_{enc}} = 1{,}03 \frac{1}{\mathrm{s}} \quad (23)$$

Mit einer Getriebeübersetzung von $n_{getriebe} = 44$ und einer Encoderauflösung $n_{enc} = 22$.

Der Roboter PIA004 und vorherige Versionen nutzen zur Regelung der Geschwindigkeit der Einzelräder einen einschleifigen Eingrößenregelkreis. Dieser nutzt die Ist-Drehzahl, welche mittels der Quadraturencoder für jedes Rad bestimmt wird als Regelgröße, welche mittels eines PI-Reglers auf eine Soll-Drehzahl geregelt wird. Software seitig ist zwar ein PID-Regler umgesetzt, mittels der gesetzten Parameter wird aber lediglich das P- und I-Glied des Reglers verwendet. Die Ausgangsgröße stell einen PWM Wert dar, welcher den Auslastungsgrad der H-Brücke beschreibt.

Betrachtet man den Einfluss der verschiedenen Regelglieder, so fällt auf, dass der P-Anteil auf dynamische Änderungen reagiert, jedoch keinen Einfluss auf das Halten einer bestimmten Drehzahl hat. Bei einem Lastwechsel reagiert des P-Glied wesentlich schneller auf die Änderung, mit zunehmender Zeit übernimmt aber zusehends das I-Glied Anteile an der Ausgangsgröße.

Da eine stabile Geschwindigkeit erst dann erreicht wird, wenn das I-Glied lang genug integriert hat, ist die in Kapitel 3.3.2 beschriebene Totzeit nur dadurch zu reduzieren den I-Term deutlich zu erhöhen, oder das Regelintervall zu verkürzen. Eine Erhöhung des I-Terms kann mitunter dazu führen, dass sich das System stark aufschwingt. Aus diesem Grund wurde in einem ersten Schritt das Regelintervall von 100Hz auf 1kHz erhöht, somit kann der Regler schneller auf Änderungen reagieren.

Eine weitere Verbesserung des Ansprechverhaltens bietet eine Vorsteuerung.

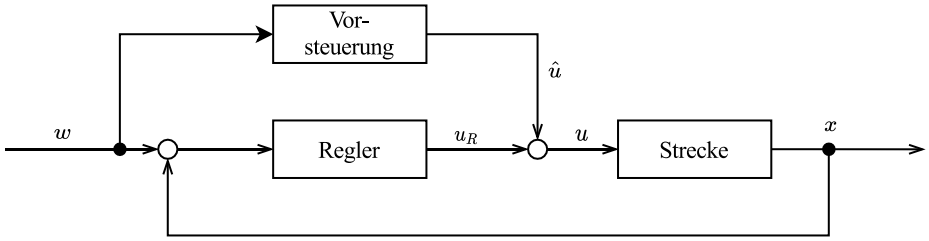

Abbildung 31: Prinzip der Vorsteuerung [5]

Hierbei wird die Regelgröße nicht nur durch den Regler beeinflusst, sondern auch durch eine parallelgeschaltete Vorsteuerung. Diese liefert eine Ausgangsgröße welche einzig vom Sollwert abhängig ist.

Um diese Abhängigkeit mathematisch beschreiben zu können, werden mittels des einfachen PID-Reglers mehrere Encoder-Geschwindigkeiten eingestellt. Der Motor dreht hierbei noch ohne Last, die so ermittelten Werte werden im folgenden Diagramm ins Verhältnis gesetzt.

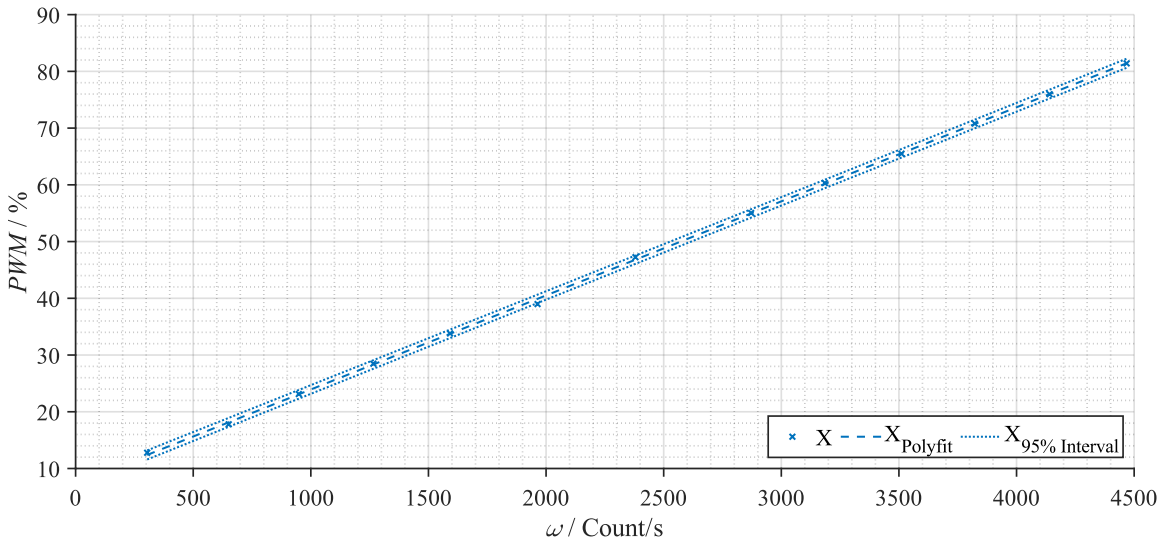

Abbildung 32: Vorsteuerungswert Motor ohne Last

Wie in Abbildung 32 zu sehen, gibt es einen linearen Zusammenhang zwischen Encoder Geschwindigkeit und dem PWM-Output, welches nötig ist, um diese Geschwindigkeit zu erreichen. Die Einheit $\left[\frac{\text{Count}}{\text{s}}\right]$ beschreibt wie viele Encoder Schritte pro Sekunde gezählt werden. Es handelt sich hierbei aber nicht um eine Ursprungsgrade. Aufgrund der Fortführung für negative Geschwindigkeiten ist eine abschnittsweise Definition für die Vorsteuerungsfunktion von Nöten.

$$MVS(w_n) = \begin{cases} w_n < 0 & \rightarrow & -a_0 - a_1 \cdot abs(w_n) \\ w_n = 0 & \rightarrow & 0 \\ w_n > 0 & \rightarrow & a_0 + a_1 \cdot w_n \end{cases} \quad (24)$$

Der Wert w_n ist proportional zur Soll-Drehzahl und wird entspricht dem vom Motorcontroller einzustellenden Wert. Der Funktionswert MVS entspricht dem vorzusteuernden Wert. Die

Parameter für a_0 und a_1 können der Abbildung 32 entnommen werden. So ist für den Leerlauf $a_0 = 18{,}732\ \%$ und $a_1 = 0{,}0432\ \frac{\%}{\text{Count/s}}$.

Versucht man jedoch Werte für den Fall unter Last zu ermitteln, stellt man fest, dass die Werte für die Vorsteuerung sich je nach Bewegungsform des Roboters doch deutlich unterscheiden.

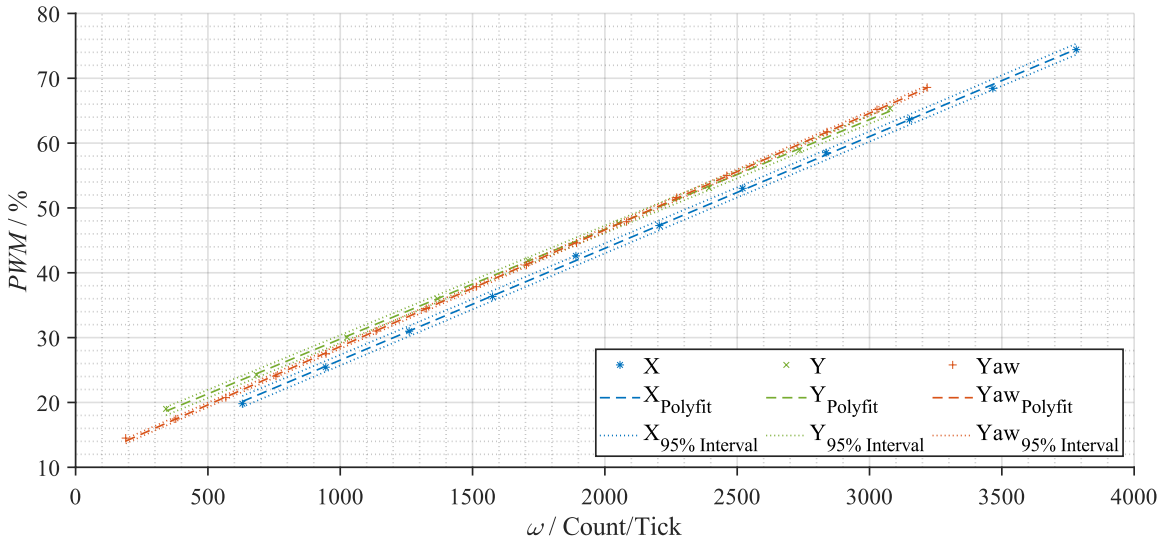

Abbildung 33: Vorsteuerung verschiedene Lastfälle

Wie in Abbildung 33 zu sehen, erfordert eine Fahrt in X-Richtung generell die geringste Leistung. Fahrten in Y-Richtung scheinen generell mehr Leistung zu erfordern, um den Roboter in Bewegung zu versetzen, dann steigt die Leistung mit zunehmender Drehzahl aber ähnlich wie bei Fahrten in X-Richtung. Rotationen um die Z-Achse erfordern bei niedrigen Rotationsraten eine Leistung, welche zwischen einer Fahrt in X- und Y-Richtung liegt. Bei hohen Drehraten übersteigt sie sogar die benötigte Leistung einer Fahrt in Y-Richtung.

Aus diesem Grund wird das Model zur Vorsteuerung der Motoren um zwei Vorsteuerungen erweitert. Somit ergeben sich in Anlehnung an Formel (24) folgende Vorsteuerungsfunktionen:

$$MVS_x(w_n) = \begin{cases} w_n < 0 & \rightarrow & -a_{0,\mathrm{x}} - a_{1,\mathrm{x}} \cdot abs(w_n) \\ w_n = 0 & \rightarrow & 0 \\ w_n > 0 & \rightarrow & a_{0,\mathrm{x}} + a_{1,\mathrm{x}} \cdot w_n \end{cases} \tag{25}$$

$$MVS_y(w_n) = \begin{cases} w_n < 0 & \rightarrow & -a_{0,\mathrm{y}} - a_{1,\mathrm{y}} \cdot abs(w_n) \\ w_n = 0 & \rightarrow & 0 \\ w_n > 0 & \rightarrow & a_{0,\mathrm{y}} + a_{1,\mathrm{y}} \cdot w_n \end{cases} \tag{26}$$

$$MVS_{yaw}(w_n) = \begin{cases} w_n < 0 & \rightarrow & -a_{0,\mathrm{yaw}} - a_{1,\mathrm{yaw}} \cdot abs(w_n) \\ w_n = 0 & \rightarrow & 0 \\ w_n > 0 & \rightarrow & a_{0,\mathrm{yaw}} + a_{1,\mathrm{yaw}} \cdot w_n \end{cases} \tag{27}$$

Die Parameter hierbei sind analog zu dem *ohne-Last*-Fall.

$$a_{0,x} = 9{,}28\,\% \qquad a_{1,x} = 0{,}017\,\frac{\%}{\text{Count/s}}$$

$$a_{0,y} = 12{,}87\,\% \qquad a_{1,y} = 0{,}017\,\frac{\%}{\text{Count/s}}$$

$$a_{0,yaw} = 10{,}63\,\% \qquad a_{1,yaw} = 0{,}018\,\frac{\%}{\text{Count/s}}$$

Empfehlenswert hierbei ist es, die so ermittelten Werte in einen Wertebereich von 0-255 zu Skalieren, da sie so mittels nur eines Bytes an die Motorcontroller übertragen werden können.

Mit Hilfe dieser Vorsteuerungsfunktionen muss nun nur noch ein Mischer für die verschiedenen Versteuerungsfunktionen umgesetzt werden.

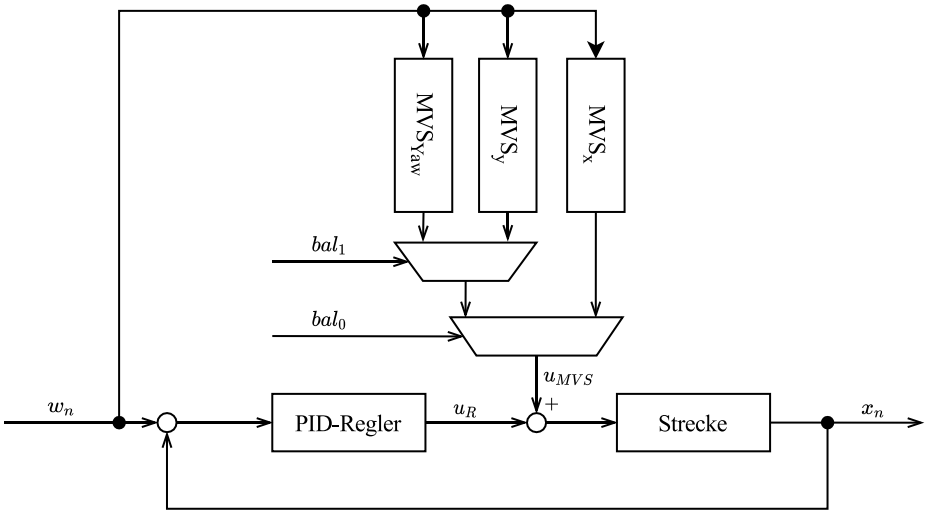

Abbildung 34: Multiple-Vorsteuerung

Wie der Abbildung 34 zu entnehmen ist, sollen für das Eingangssignal w_n drei verschiedene Vorsteuerungswerte berechnet werden. Diese werden dann mittels der Größen bal_0 bzw. bal_1 gemischt. In einer ersten Annahme wird davon ausgegangen, dass die verschiedenen Komponenten linear ineinander übergehen.

Es muss nun im Folgenden noch ein Weg gefunden werden zu ermitteln, zu welchen Anteilen die einzelnen Vorsteuerungen ein zu gehen haben.

So setzt sich die Motorgeschwindigkeit eines Motors dadurch zusammen, dass die verschiedenen Bewegungsarten zunächst skaliert und dann je nach Motor entweder addiert oder subtrahiert werden. Diese Berechnung findet innerhalb der Funktion *calcMixing* statt, weshalb an dieser Stelle auch das Gewichten der einzelnen Vorsteuerungsanteile geschehen muss. In einem Blockschaltbild lässt sich das wie folgt darstellen.

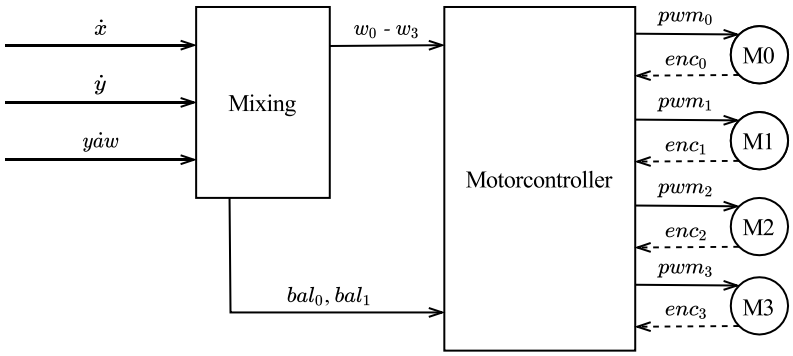

Abbildung 35: Blockschaltbild Motorregelung

So ist in Abbildung 35 zu sehen, dass die Geschwindigkeiten $\dot{x}$, $\dot{y}$ und $y\dot{a}w$ im Mixing in die Motorgeschwindigkeiten w_0 bis w_3 umgewandelt werden. Gleichzeitig werden noch die beiden Gewichtungsfaktoren bal_0 und bal_1 bestimmt.

In einem ersten Schritt wird unterschieden, ob es sich um eine X-Bewegung handelt und im zweiten Schritt, wird zwischen Y- und Yaw-Bewegung unterschieden. Um eine Referenzgröße zu haben, wird zunächst das absolute Maximum bestimmt. Hierzu werden die vorskalierten Werte für die X-, Y- und Yaw-Bewegung betragsweise addiert:

$$MVS_{sum0} = abs(\dot{x}) + abs(\dot{y}) + abs(y\dot{a}w) \tag{28}$$

Von diesem Wert wird der X-Anteil bestimm:

$$MVS_{balance0} = \frac{abs(\dot{x})}{MVS_{sum0}} \cdot 255 \tag{29}$$

Die Multiplikation mit 255 rührt daher, dass dieser Parameter mittels eines Bytes vom ESP32 an den Motorcontroller übermittelt werden soll. Somit bedeutet ein Wert von $MVS_{balance0} = 255$, dass es sich bei der Bewegung ausschließlich um eine Bewegung in X-Richtung handelt. Niedrigere Werte bedeuten, dass die Bewegung auch Anteile einer Y- bzw. Yaw-Bewegung beinhaltet.

Sollte $MVS_{sum0} = 0$ sein, wird zur Vermeidung von Zero-Division-Fehlern $MVS_{balance0} = 255$ gesetzt.

Um zwischen einer Yaw- bzw. Y-Bewegung zu unterscheiden wird die Vorgehensweise wie zuvor wiederholt, als Referenz wird diesmal aber nur das absolute Maximum von Y und Yaw herangezogen.

$$MVS_{sum1} = abs(\dot{y}) + abs(y\dot{a}w) \tag{30}$$

Den Y-Anteil beschreibt somit:

$$MVS_{balance1} = \frac{abs(\dot{y})}{MVS_{sum1}} \cdot 255 \tag{31}$$

Motorcontrollerseitig werden die Werte wieder in Gewichtungsfaktoren umgewandelt:

$$b_x = bal_0 = \frac{MVS_{balance0}}{255} \tag{32}$$

$$bal_1 = \frac{MVS_{balance1}}{255} \tag{33}$$

$$b_y = (1 - bal_0) \cdot bal_1 \tag{34}$$

$$b_{yaw} = (1 - bal_0) \cdot (1 - bal_1) \tag{35}$$

Ein Vorsteuerungswert berechnet sich somit schlussendlich mittels folgender Formel:

$$MVS(w_n, b_x, b_y, b_{yaw}) = b_x \cdot MVS_x(w_n) + b_y \cdot MVS_y(w_n) + b_{yaw} \cdot MVS_{yaw}(w_n) \tag{36}$$

Eine Ausführliche Analyse der dadurch erzielten Fahrleistungen befindet sich im Kapitel 5.1.

4.1.3 IMU-Kursstabilisierung

Eine weitere Optimierung der Fahrleistungen des PIA-Roboters soll durch eine Yaw-Stabilisierung mittels der im Roboter verbauten IMU geschehen. So ist in Kapitel 3.3.5 zu sehen, dass zeitlich versetztes Andrehen der einzelnen Räder oder Kontaktverlust mit dem Boden mitunter zur Rotation des Roboters führen kann. Um die daraus resultierenden Fahrfehler kompensieren zu können, soll die im Roboter verbaute IMU verwendet werden, um ein ungewolltes Rotieren des Roboters zu kompensieren.

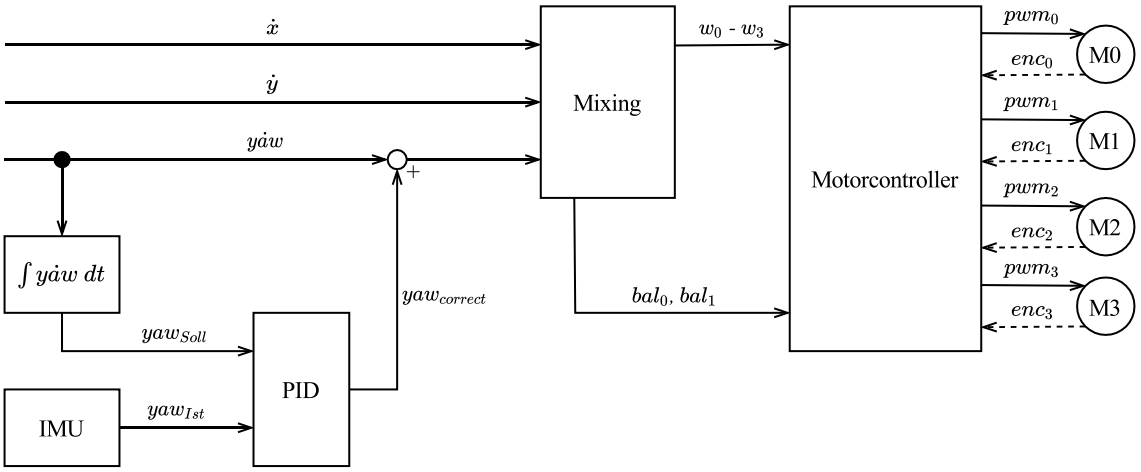

Abbildung 36: Blockschaltbild IMU-Stabilisierung

Hierzu wird, wie in Abbildung 36 zu sehen ist, die Soll-Yaw-Geschwindigkeit integriert und mit dem von der BNO055 gelieferten Yaw-Wert verglichen. Ein PID-Regler errechnet aus der Differenz der beiden Werte dann eine Rotationsgeschwindigkeit, welche auf die Soll-Geschwindigkeit aufaddiert wird. Dies ermöglicht es Fehler bezüglich der Orientierung des Roboters zu kompensieren. Damit dies jedoch gut funktioniert muss die Sensordrift kompensiert werden. Dies geschieht bei der BNO055 mittels des integrierten Controllers, welcher die Daten aufbereitet. So ist die BNO055 in der Lage mehrere Sensordaten mit einander zu verknüpfen und auf zu integrieren.

Eine weitere Möglichkeit besteht darin, die von der IMU gelieferte YAW-Rotationsgeschwindigkeit zu verwenden und diese mit der Soll-Rotationsgeschwindigkeit zu vergleichen. Dies führt aber zu erheblich schlechteren Ergebnissen, da nicht sichergestellt werden kann, dass jeder neue IMU-Wert ausgelesen wird.

4.2 Kombination aller Teilsysteme

In diesem Kapitel wird beschrieben wie aus den verschiedenen zuvor erarbeiteten Teilkomponenten ein komplett einsatzfähiger Roboter wird.

4.2.1 Hardware

Da der Roboter PIA6 viele neue Hardware Komponenten mit sich bringt, ist dies am besten mittels eines Systemdiagrams darzustellen.

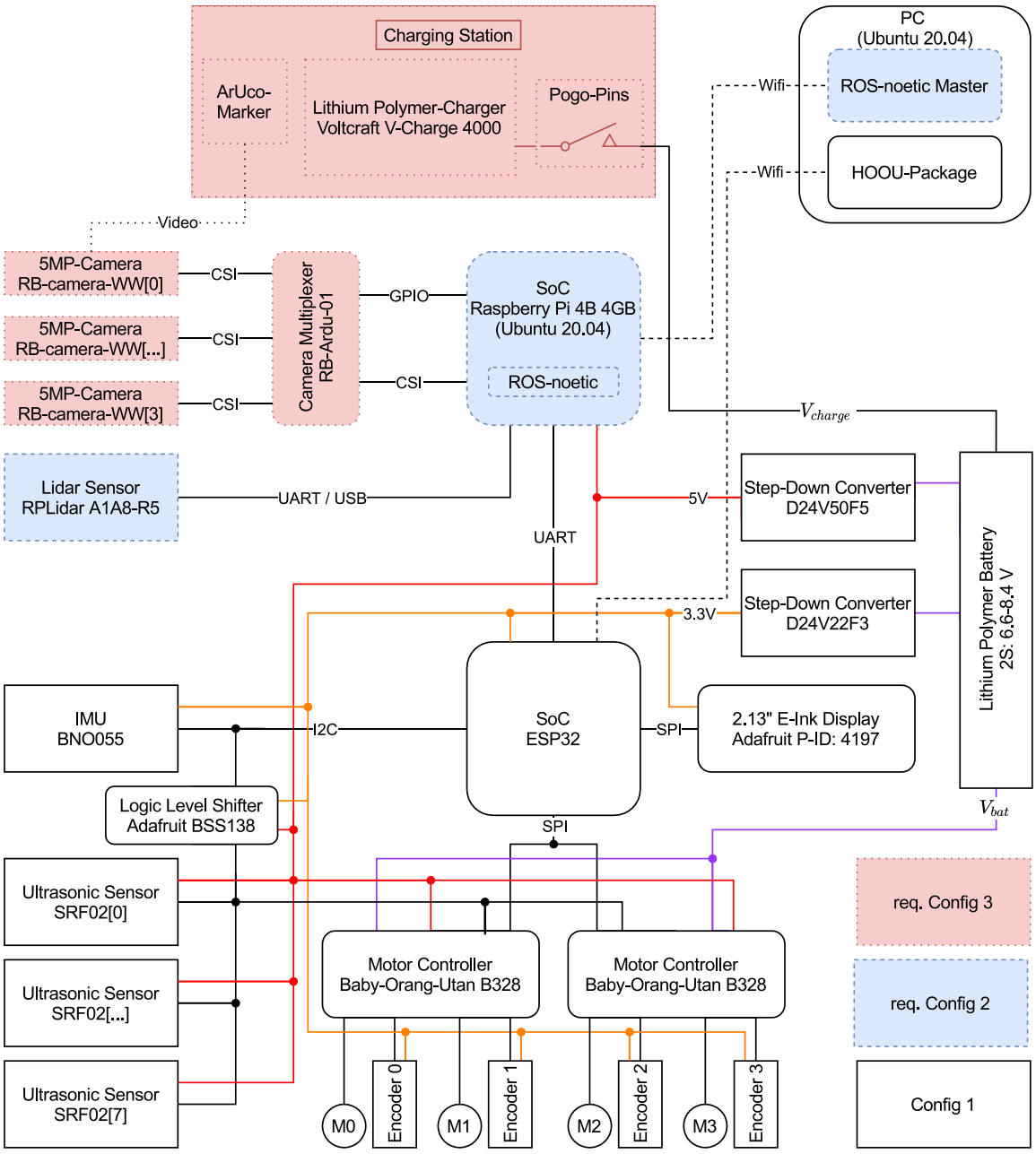

Abbildung 37: Systemdiagram PIA6 incl. Ladeinfrastruktur [19]

Das in Abbildung 37 dargestellte Systemdiagramm stellt das durch Finn Weithoff erstellte Systemdiagram des Roboters PIA6 dar. Die meisten der angebrachten Änderungen beziehen

sich auf die Verwendung des ESP32 und der beiden Baby Orangutan Motorcontroller. Dieses verändert den Aufbau der vorigen Layer 1-2.

Da viele neue Komponenten verwendet werden und die Produktionszeit einzelner PIA6 Roboter gesenkt werden soll, soll ein Platinendesign erarbeitet werden, welches eine einfache und schnelle Produktion ermöglicht.

Auf dem Weg zur Platine müssen die einzelnen Schaltkreise im Vorfeld erstellt und getestet werden, um sie im Zweifel noch überarbeiten zu können, bevor die Platine bestellt wird. Hierzu werden mehrere Prototypen angefertigt.

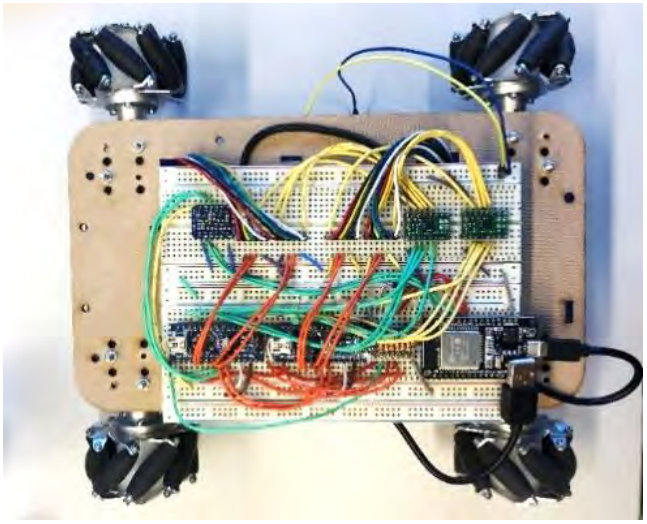

Abbildung 38: Breadboard - Prototyp ESP32 und zwei Arduino-Nano mit jeweils einem Pololu DRV8833 für einen vollen Programmiertest und erste Fahrversuche

Ein erster Prototyp testet die Ansteuerung und Programmierung der Motorcontroller mittels zweier ATMega 328p. Dies soll ermöglichen, dass der Entwicklungszyklus bis hin zu einer fertigen Platine auf ein mögliches Minimum verkürzt wird. So bildet der in Abbildung 38 zu sehenden Prototyp die Grundlage zur Erprobung der Umsetzung des in Kapitel 4.1.1 beschrieben FOTA-Programmers. Ebenso können die Portierung des gesamten HOOU-PIA-Projekts von der SAMD-Architektur auf die ESP32-Architektur erprobt werden und einfache Testfahrten durchgeführt werden.

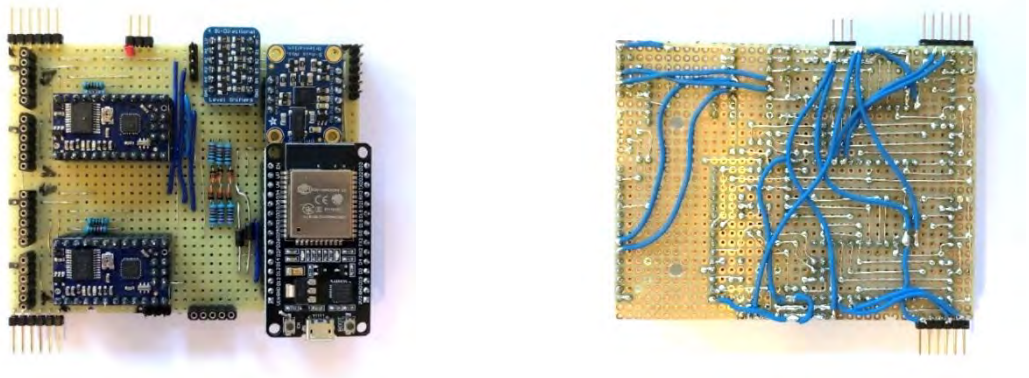

Abbildung 39: PIA6 Prototyp Platine bestückt mit zwei Baby Orangutan einer BNO055 und einem ESP32 sowie 4-Channel-Level-Shifter

Die in Abbildung 39 dargestellte Platine ist der erste Prototyp zur Erprobung aller Schnittstellen so wie der kompletten Sensorik eines PIA6-Roboters. Auf dieser Platine sind zum ersten Mal zwei Pololu Baby Orangutan verbaut ebenso wie ein ESP32. Die Platine bietet eine Anschlussmöglichkeit von vier Motoren inclusive ihrer Quadraturencoder. Ebenso ist für beide Motorcontroller eine serielle Schnittstelle zugänglich, um besser Debuggen zu können. Des Weiteren sind Konnektoren vorhanden, die es ermöglichen, sämtliche Sonars sowie einen RaspberryPi zu verbinden. Mit diesem Prototyp wird das Gesamtsystem des PIA6 Roboters erprobt.

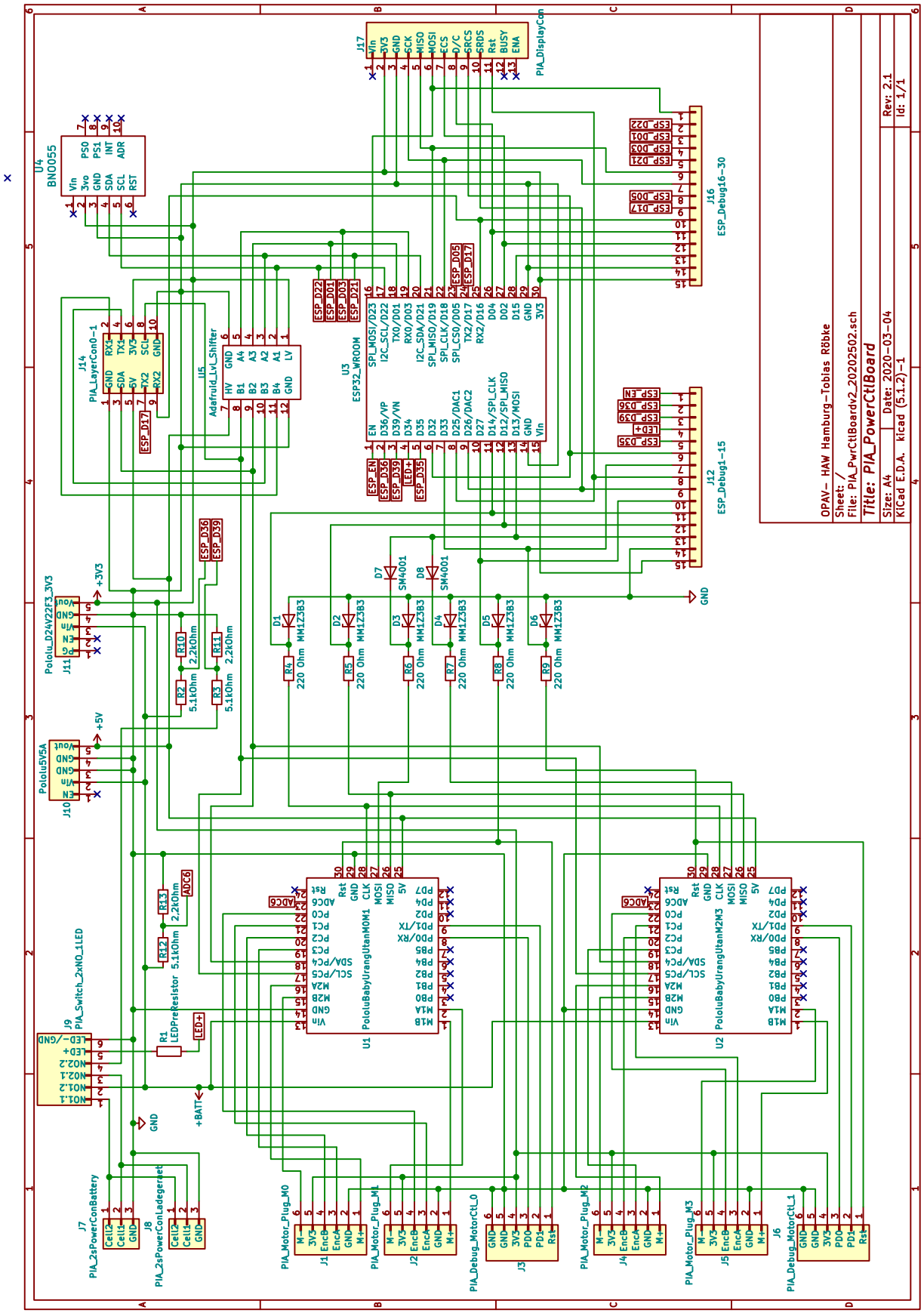

Abbildung 40: Schaltplan PIA6 [4]

In Abbildung 40 zu sehen ist der Schaltplan welcher aus den zuvor gewonnen Erkenntnissen entstanden ist. Dieser wurde dankenswerterweise von Tobias Röbke in KiCAD übertragen und in PCB design überführt so, so dass weitere PIA6-Roboter kostengünstig und schnell produziert werden können.

Bei dem PCB design wurde darauf geachtet, dass es weiterhin eine Entwicklungsplattform darstellt. So sind z.B. alle GPIOs des ESP32 noch einmal durch eine Buchsenleiste zugänglich gemacht worden. Dadurch ist es möglich bei kleineren Fehlern im Design die Platine alternativ zu beschalten.

Ebenso wurde für die beiden Baby Orangutan ein UART Interface zugänglich gemacht. Dies dient der besseren Testbarkeit der Motor Controller im Entwicklungszustand.

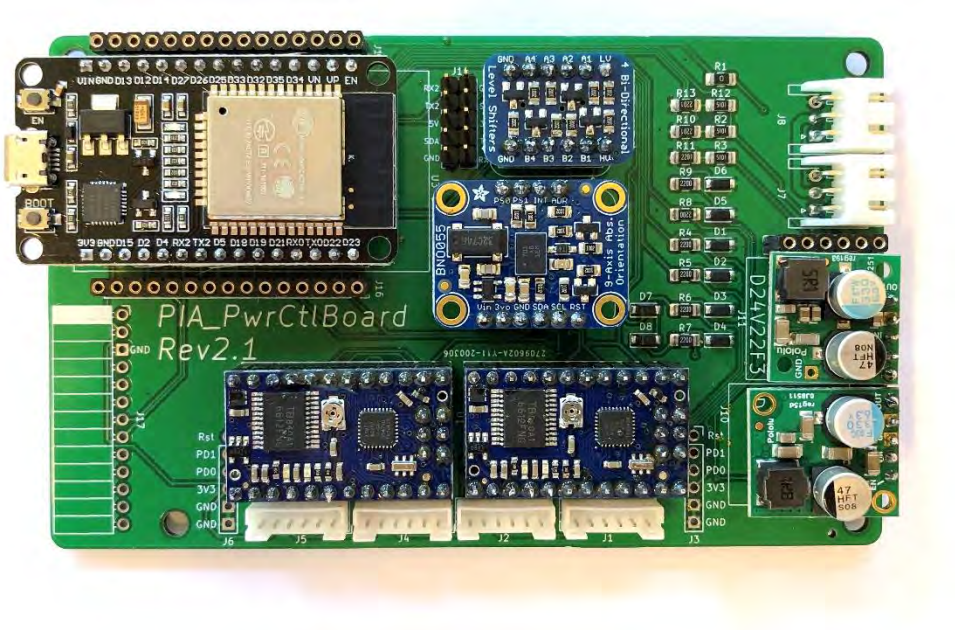

Abbildung 41: PIA6 Platine (PIA_PwrCtrBoard Rev.2.1) bestückt mit zwei Baby Orangutan, einem ESP32, einer BNO055, einem 4-Channel-Level-Shifter sowie der beiden Spannungsregulierer Pololu D24V50F5 (5V) und Pololu D24V22F3 (3V3)

Die in Abbildung 41 zusehende Platine entspricht dem aktuellsten Stand der in dieser Arbeit behandelten Entwicklung. Diese Platine dient im Folgenden als Kern der verwendeten Roboter.

Im Vergleich zu der in Abbildung 39 zusehenden Prototyp-Platine wurde diese noch um Spannungswandler und einen Anschluss für ein eInk-Display ergänzt. Um die Anforderungen des autonomen Ladens zu erfüllen wurde eine zweite JST-Buchse vorgesehen, so dass jede einzelne Zelle des 2S-Lipo geladen werden kann. Dieser Ladevorgang kann durch den ESP32 überwacht werden und im Notfall abgebrochen werden.

Ebenso ist vorgesehen, dass die beiden Motorcontroller die Akkuspannung ebenso messen können, falls in einer späteren Entwicklung bei der Motorsteuerung auf die anliegende Spannung reagiert werden soll.

Abbildung 42: PIA004 links und PIA6 rechts

Die in Abbildung 42 zu sehenden Roboter sind zum einen links der Referenzroboter PIA004 und zum anderen rechts der finalen Roboter PIA6-1. Einige hardwareseitige Änderungen wurden an PIA6-1 ebenso vorgenommen, so wurde unter anderem das Fahrwerk überarbeitet, so dass es stabiler und flacher ist. Diese Änderungen werden jedoch in dieser Ausarbeitung nicht weiter beleuchtet.

4.2.2 Software

Mit dem ESP32 als neuem Hauptcontroller des Basis-Roboters muss die gesamte Firmware an den neuen Controller angepasst werden. Hierzu müssen alle verwendeten Schnittstellen angepasst werden. Alle Dateien sind im Anhang einzusehen.

Um das Projekt übersichtlicher zu gestalten, wurde es in mehrere Dateien untergliedert. Somit ist der Programmablauf eindeutiger zu erkennen und verschiedene Module sind in separaten Dateien zusammengefasst.

In der Hauptprojekt-Datei werden die spezifischen Dateien inkludiert und deren Funktionen verwendet.

Ein jedes Arduino Programm besteht mindestens aus einer „setup“ und einer „loop“ Funktion. Im „setup“ werden alle Module initialisiert und einmalige Einstellungen vorgenommen. Und in der „loop“ Funktion werden wiederkehrende Funktionen aufgerufen. Da nicht jede Funktion in jedem Programmdurchlauf aufgerufen werden soll, wurde ein Mechanismus angelegt, welcher den Aufruf von Funktionen zu einem bestimmten Zeitpunkt gewährleistet.

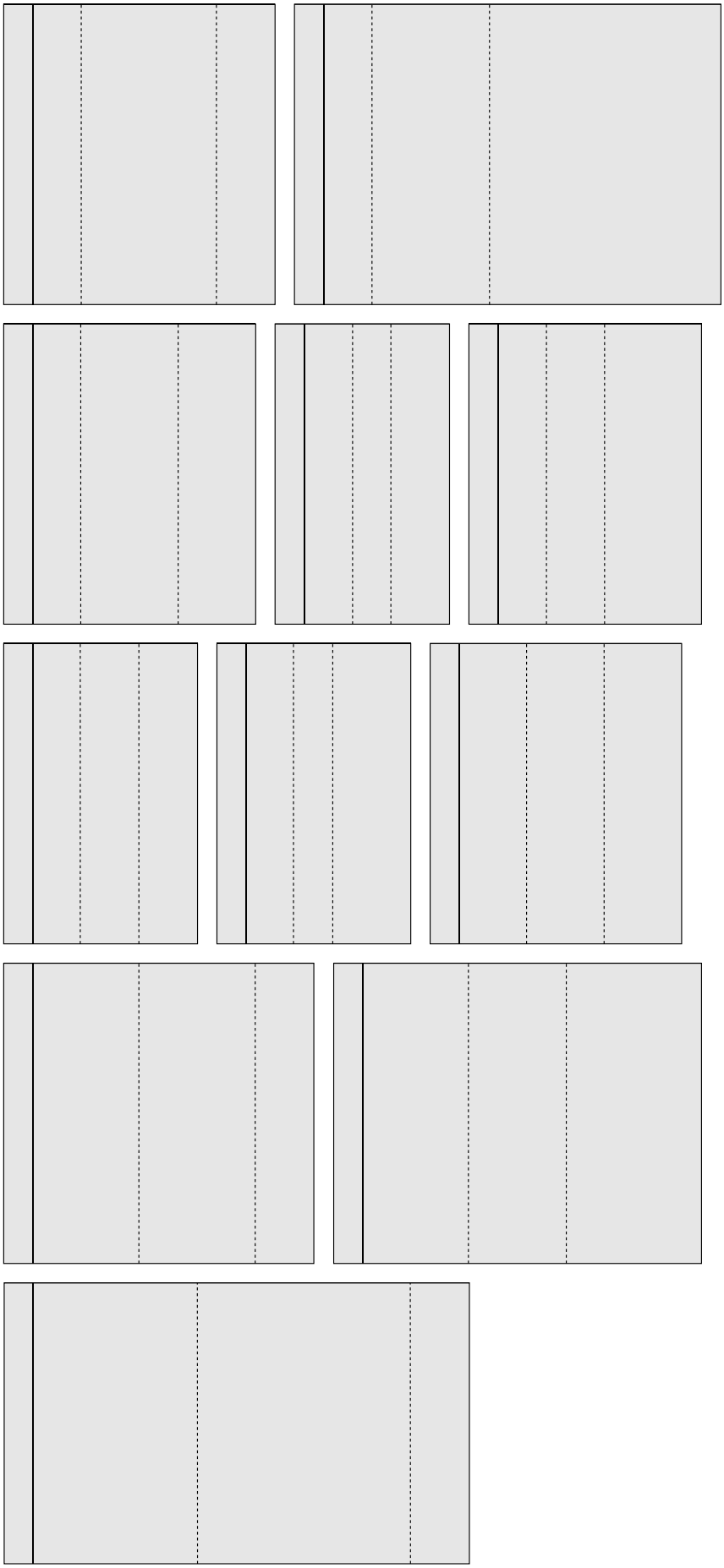

Abbildung 43: ESP32 Firmware-Architektur im Überblick

In Abbildung 43 ist die neue ESP32-Firmware-Architektur dargestellt. Viele der vorgenommenen Änderungen beziehen sich auf das Detektieren von Sensorfehlern. Da mit den zwei Motorcontrollern zwei weitere Endpunkte an der I2C-Schnittstelle angeschlossen sind, wurde eine Funktion implementiert, welche überprüfen kann, ob eine bestimmte I2C-Adresse erreichbar ist. Dies kann nicht nur bei der Überprüfung der Motorcontroller verwendet werden, sondern auch bei der Initialisierung der Sonar-Sensoren und der IMU. Somit ist es in der neuen Firmware möglich, einzelne nicht funktionierende Sensoren besser zu identifizieren.

Die meisten Anpassungen bei der neuen Firmware betreffen die Motorsteuerung und die Kommunikation. Motorsteuerung betreffende Änderungen befinden sich nun in der Datei _Move.cpp. Alle Kommunikationsrelevanten Aufgaben werden in der Datei _COMM.cpp behandelt.

So wird in der _Move.cpp mittlerweile nicht mehr die in Kapitel 4.1.2 beschriebene Closed-Loop-Motorsteuerung umgesetzt, dafür aber die Kapitel 4.1.3 beschriebene IMU-Kursstabilisierung. Um die Kommunikation mit den Motorcontrollern zu gewährleisten, wurde eine eigene Bibliothek *HOOU_Baby_Orangutan* zur Steuerung der selbst programmierten Motorcontroller erstellt und in der Datei _Move.cpp eingebunden. Somit ist es mittels einfacher Funktionen möglich die Motorcontroller zu konfigurieren und verschiedene Drehzahlen einzustellen.

Die Kommunikation via WIFI des ESP32 hat sich gegenüber dem Arduino M0 Pro doch signifikant verändert. So hat der Arduino M0 Pro nur ein Protokoll verwendet, und zwar das UART-Protokoll. Der ESP32 hingegen benutzt das UART-Protokoll sowie einen UDP-Socket zur Kommunikation. Damit UDP-Pakete zu jeder Zeit empfangen werden können wird hierzu die AsyncUDP-Bibliothek verwendet. Diese ermöglicht es UDP-Pakete zu empfangen und zu versenden, und dieses nicht nur als Broadcast, sondern auch direkt an eine bestimmte IP. Somit ist mittels dieser Bibliothek schon eine Roboter-zu-Roboter-Kommunikation sichergestellt.

Eine weitere größere Änderung geht mit dem Umstieg auf den ESP32 einher. So ist es möglich den ESP32 OTA zu programmieren. Damit ist eine Fernwartung eines jeden Roboters möglich. Um auch die Motorcontroller fernwarten zu können wurde, wie in Kapitel 4.1.1 beschrieben, eine neue Funktionalität namens FOTA umgesetzt. Hierzu werden ein Dateisystem sowie ein Webinterface benötigt. Um dies zu gewährleisten wurde das ESP32 Beispiel *FSBrowser* so angepasst, dass der Upload und das Programmieren der beiden verbauten Baby Orangutan Motorcontroller ermöglicht wurde. Die zu diesem Zweck verwendete selbst erstelle Bibliothek nennt sich *os_fs_avrisp*.

Viele weitere Änderungen wurden vorgenommen, diese jedoch weiter zu erläutern, würde den Umfang dieser Arbeit sprengen. Der gesamte Sourcecode ist dem digitalen Anhang beigefügt und kann dort eingesehen werden.

4.3 Kommunikation mittels UDP-Socket als Multi-Roboter-Steuerung

Grundlage bezüglich des Konzepts zur Multi-Roboter-Steuerung bildet die Verwendung von UDP-Sockets anstelle einer UART-Verbindung. Seitens des PIA6-Roboters wurde durch die Verwendung eines ESP32 an Stelle eines XBees eine Roboter-zu-Roboter-Kommunikation so schon ermöglicht.

Seitens des PIA-Python-Packages, zur Ansteuerung der Roboter von einem anderen PC oder dem im Roboter verbauten Raspberry, müssen jedoch noch Änderungen vorgenommen werden.

Als Grundlage hierfür dient das *System Interface Serial*. Dieses nutzt eine UART-Schnittstelle zur Kommunikation und wird auch weiterhin roboterintern seine Verwendung finden. Das neue *System Interface UDP* wird die gleichen Funktionen besitzen, jedoch nicht die UART-Schnittstelle verwenden, sondern UDP-Sockets.

Abschließend wird ein Beispiel erstellt, um die Kommunikation im Test erproben zu können.

4.3.1 PIA-Python-Package

Alle Tests und Demos sind in Python umgesetzt, ebenso wird das PIA-Python-Package als Verknüpfung zwischen Microcontroller und dem im Roboter verbauten RaspberryPi verwendet. Hierzu gibt es bereits verschiedene System Interfaces. Das einzige Funktionierende stellt hierbei jedoch *System Interface Serial* dar.

Das aktuelle PIA-Python-Package besteht aus Klassen und Funktionen welche in Abbildung 44 dargestellt sind. Wie bereits in Kapitel 3.2.2 erwähnt, kann mit dem System Interface Serial keine Zuordnung von Nachrichten zu Robotern gemacht werden.

Um dies zu ermöglichen wird ein neues System Interface angelegt, welches keine UART Schnittstelle verwendet, sondern einen UDP-Socket. Dies lässt es zu, ein Ziel für jede einzelne Nachricht zu definieren. Ebenso wird es dadurch möglich den Ursprung bei eingehenden Daten zu ermitteln.

Um den Zielort flexibel angeben zu können, bekommen alle Funktionen einen optionalen Funktionsparameter, damit die Funktionen abwärtskompatibel bleiben und bestehende Programme und Tests weiterhin funktionieren.

So ändert sich zum Beispiel die Funktionssignatur für die zum Fahren verwendete Funktion von:

```
def SendMovePacket(self, x, y, yaw, ts=None):
```

zu

```
def SendMovePacket(self, x, y, yaw, ts=None, ip='255.255.255.255'):
```

Gibt man also in einem Programm keine IP an, so werden die Befehle wie zuvor als Broadcast an alle Roboter gesendet. Spezifiziert man jedoch die IP, so kann ein Roboter individuell

angesteuert werden. Diese Art Änderung wurde für alle Funktionen ergänzt. Ebenso wird zu jedem eingehenden Paket die IP des Senders mitgespeichert. Diese Änderungen haben auch Einfluss auf weitere Module, weshalb diese entsprechend angepasst sind.

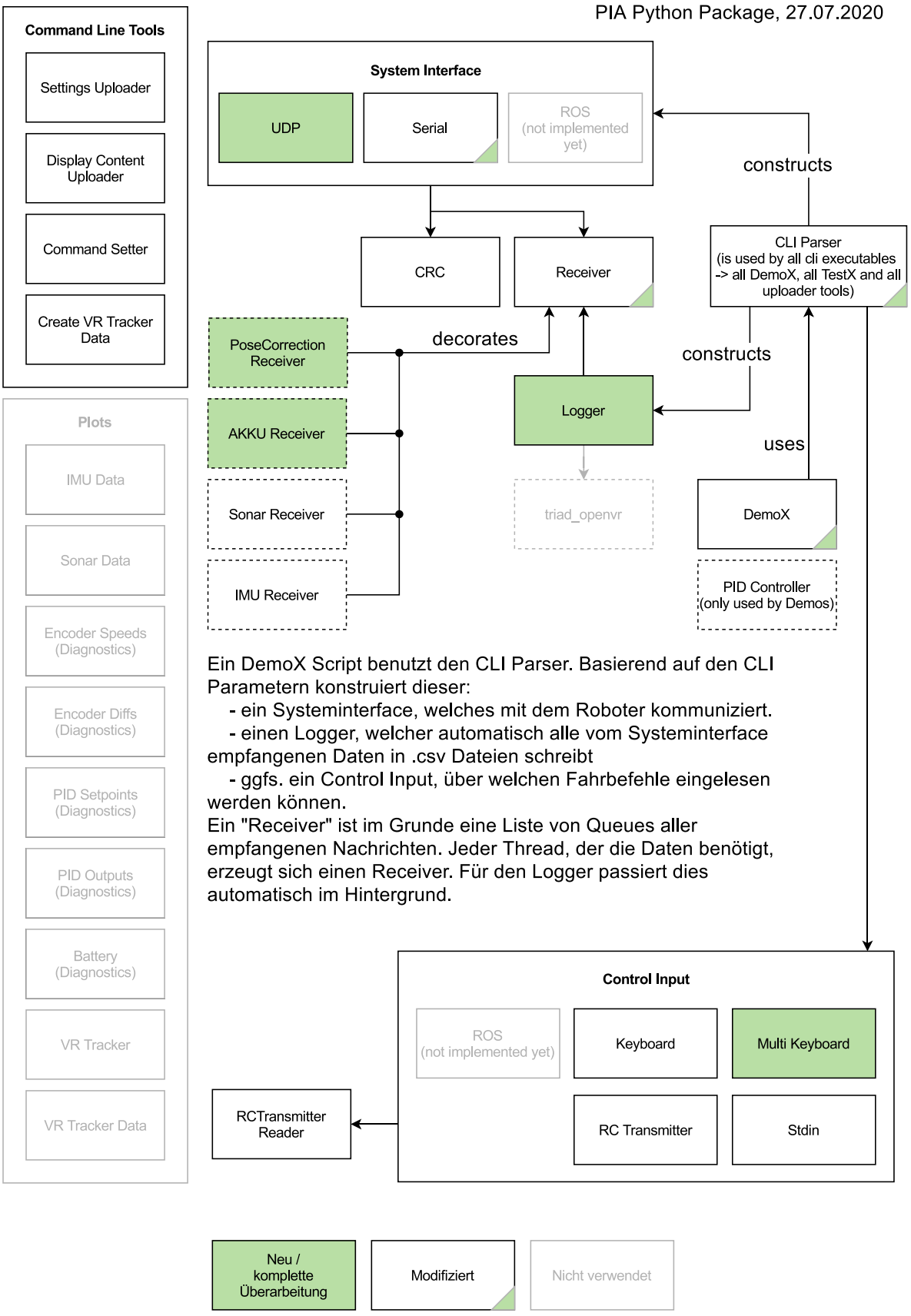

Abbildung 44: Änderungen am PIA Python Package zur Verwendung als Multi-Roboter-Steuerung [20]

In Abbildung 44 zu sehen ist die Struktur des PIA-Python-Package. Markiert sind alle Pakete, welche durch die Änderungen zur Multi-Roboter-Steuerung angepasst werden mussten.

So musste der Logger dahingehend angepasst werden, dass pro eingehender IP die Daten individuell abgelegt werden. So gibt es nach den Änderungen für jede IP und jeden Sensor eine eigene Datei, in welche alle dies bezüglichen Daten abgelegt werden.

Ebenso ist es nunmehr möglich die Logger so zu konfigurieren, dass es möglich ist, verschiedenen Robotern mittels ihrer IP VR-Tracker zuzuordnen. Diese werden dann durch den Logger unter der IP des dazugehörigen Roboters abgelegt.

Das Control Input Multi Keyboard ermöglicht es bis zu vier Roboter individuell über die Tastatur zu steuern. Dies funktioniert aber ausschließlich in Kombination mit dem System Interface UDP.

Um alle Funktionalitäten für das mobile Laden abdecken zu können, wird für das Testprogramm welches in Kapitel 4.3.2 beschrieben wird, auch die Akku-Spannung, sowie ein Korrekturwert zwischen Soll- und Ist-Position bereitgestellt. Hierzu stehen zwei weitere Klassen zur Verfügung, welche die Receiver Klasse des jeweiligen System Interface dekorieren.

4.3.2 Testprogramm zur Multi-Roboter-Steuerung (HIVE-Control)

Zur Erprobung der Multi-Roboter-Steuerung wurde ein Konzept erdacht, bei dem mehrere Roboter eine Formationsfahrt durchführen, damit ein oder mehrere Roboter, welche im folgenden BEE genannt werden, durch einen weiteren Roboter mit den Namen HIVE geladen werden kann. Sobald die Roboter verbunden sind, muss lediglich der HIVE gesteuert werden und alle Steuerbefehle werden für die BEEs so transformiert, dass die relative Position zum HIVE gleichbleibt.

Um eine hohe Flexibilität in der Programmierung zu gewährleisten und alle Roboter auf die gleiche Weise ansteuern zu können, wird ein End-Effektor Koordinatensystem (EE) als Bezugskoordinatensystem definiert. Anders als in allen vorigen Testprogrammen, welche für den Roboter PIA004 entwickelt wurden, wird nun aber nicht mehr ein Roboter gefahren, sondern ein virtueller Punkt, welcher durch das EE beschrieben wird. Beschreibt man nun die Roboter in diesem Koordinatensystem, muss lediglich die Transformation von einer EE-Bewegung in eine Roboterbewegung berechnet werden.

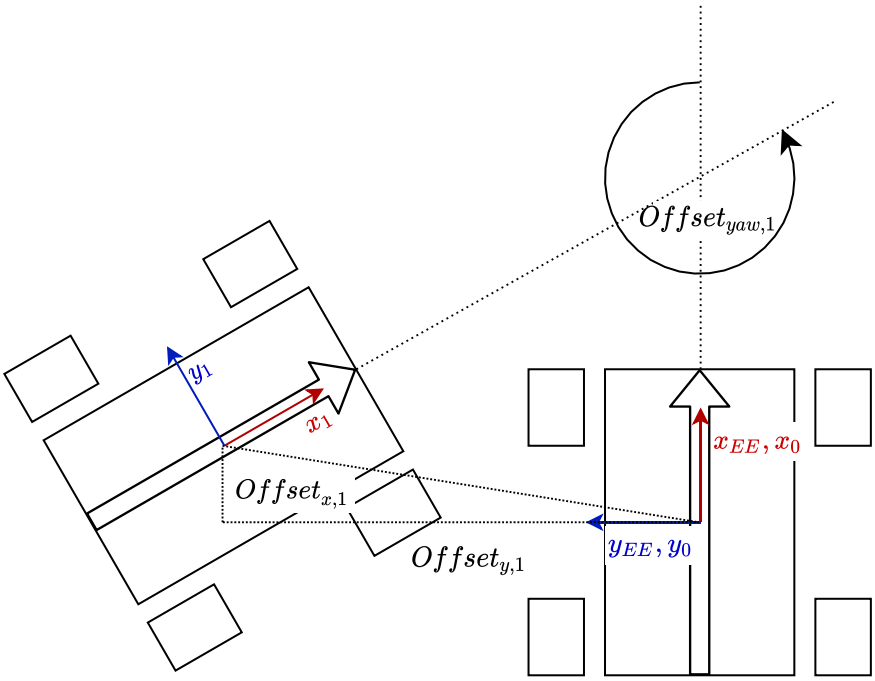

Abbildung 45: Veranschaulichung des Offsets anhand von zwei Robotern bezüglich des Endeffektor-Koordinatensystems

Um diese Transformation berechnen zu können, müssen die einzelnen Roboter bezüglich des EE beschrieben werden. Die Konvention, nach welcher dies im Programm passiert, ist in Abbildung 45 zu sehen.

Dargestellt sind zwei Roboter und 3 Koordinatensysteme, ebenso wie die je Roboter anzugebenden Offsets. Roboter-0 hat keinerlei Verschiebung oder Rotation zum EE. Somit sind seine Offsets null. Er ist als HIVE definiert und jede Bewegung des EE entspricht der Bewegung des HIVE. Obwohl also das EE über die Tastatureingabe gesteuert wird, sieht es so aus, als würde der HIVE gesteuert werden. Obwohl für diesen Spezialfall keinerlei Transformationen berechnet werden müssen, wird dies dennoch getan, da so alle Roboterbewegungen auf die gleiche Weise bestimmt werden können.

Der Roboter-1 besitzt ein Offset sowohl in x- und y-Richtung wie auch eine Verdrehung um Yaw. Dieser Roboter wird im weiteren als BEE-Bezeichnet.

Die berechnete Transformation wird für alle Roboter gleich beschrieben. So besitzt jeder Roboter ein Offset sowohl in x- als auch in y-Richtung und ist zusätzlich noch um eine Verdrehung bezüglich des EE. Die vom Roboter-n zu fahrenden Geschwindigkeiten errechnen sich wie folgt.

$$\overrightarrow{v_n} = T_{EE,n} \cdot \begin{pmatrix} \dot{x_{EE}} \\ \dot{y_{EE}} \\ \dot{yaw}_{EE} \end{pmatrix} + T_{EE,n} \cdot \overrightarrow{r_{EE,n}} \cdot \dot{yaw}_{EE} \tag{37}$$

mit der Transformationsmatrix

$$T_{EE,n} = \begin{pmatrix} \cos(Offset_{yaw,n}) & -\sin(Offset_{yaw,n}) & 0 \\ \sin(Offset_{yaw,n}) & \cos(Offset_{yaw,n}) & 0 \\ 0 & 0 & 1 \end{pmatrix} \tag{38}$$

und dem Vektor

$$\overrightarrow{r_{EE,n}} = \begin{pmatrix} \cos(90°) & -\sin(90°) & 0 \\ \sin(90°) & \cos(90°) & 0 \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} Offset_{x,n} \\ Offset_{y,n} \\ 0 \end{pmatrix} = \begin{pmatrix} -Offset_{y,n} \\ Offset_{x,n} \\ 0 \end{pmatrix} \quad (39)$$

Die Rotation welche in Formel (39) beschrieben ist, ist nötig, um aus den Offsetvektoren die Tangente an dem zu fahrenden Kreis zu bestimmen.

Diese Berechnung muss nun lediglich für jeden Roboter bei jeder Geschwindigkeitsänderung getätigt werden und zwei ideale Roboter, für die das Offset exakt bestimmt ist, würden sich nicht mehr relativ zueinander bewegen. Da dieses Programm jedoch für reale Roboter geschrieben wird, müssen Fehler durch äußere Einflüsse oder eine ungenaue Offsetbestimmung kompensiert werden.

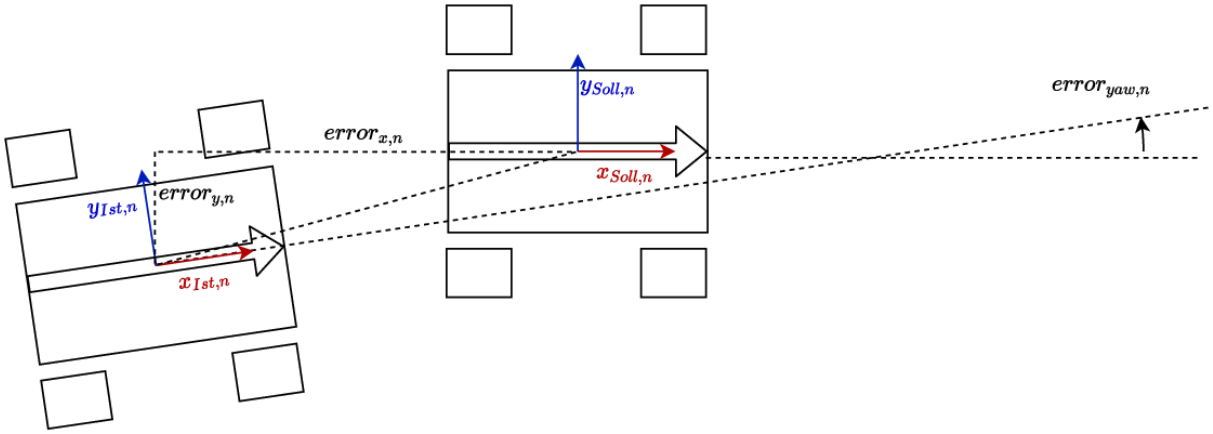

Abbildung 46: Veranschaulichung des Offset-Fehlers anhand eines Roboters, welcher aus seiner Soll-Position verschoben wurde

In Abbildung 46 zu sehen ist ein und derselbe Roboter. Seine Ist-Position weicht von der Soll-Position ab. Diese Abweichung wird mit der im Roboter verbauten Kamera mittels AR-Marker berechnet. Hierzu wird auf dem Raspberry Pi 4 des PIA6 Roboters das Skript *ArucoPositionPublisher.py* ausgeführt. Dieses detektiert die Position und Pose eines im Skript festgelegten Markers.

Ein neues HOOU-Packet namens POSE_CORRECTION überträgt den so ermittelten Abstand der beiden Roboter zueinander an das HIVE-Control-Skript, welches daraus die Abweichung von der Soll-Position ermittelt.

Der so ermittelte Fehler soll mit Hilfe eines einfach P-Reglers kompensiert werden. Die daraus resultierenden Geschwindigkeiten werden auf die durch Gleichung (37) errechneten Werte aufaddiert.

$$\overrightarrow{v_{err,n}} = \begin{pmatrix} Error_x \cdot P_{comp,XY} \\ Error_y \cdot P_{comp,XY} \\ Error_{yaw} \cdot P_{comp,YAW} \end{pmatrix} \cdot \frac{1}{t_{comp}} \quad (40)$$

Die Konstante $t_{comp} = 1\ \mathrm{s}$ beschreibt den Zeitraum, in welchem der gemessene Fehler um den Faktor $P_{comp,XY} = P_{comp,YAW} = 0{,}5$ verringert werden soll. Somit ergibt sich eine Gesamtgeschwindigkeit pro Roboter von:

$$\overrightarrow{v_{ges,n}} = \overrightarrow{v_n} + \overrightarrow{v_{err,n}} \quad (41)$$

Da die Geschwindigkeiten so mitunter zu extremen Beschleunigungen führen können, welche die Roboter in der Realität nicht fahren können, werden diese Geschwindigkeiten abschließend an eine *RoboSpeed* Klasse übergeben. Diese Klasse erfüllt den Zweck, dass die maximalen Beschleunigungen des Roboters nicht überschritten werden. Hierzu werden für die verschiedenen Fahrrichtungen des Roboters maximale Beschleunigungswerte bei der Initialisierung angegeben. Mittels der *setTarSpeed* Funktion kann nun eine Zielgeschwindigkeit gesetzt werden.

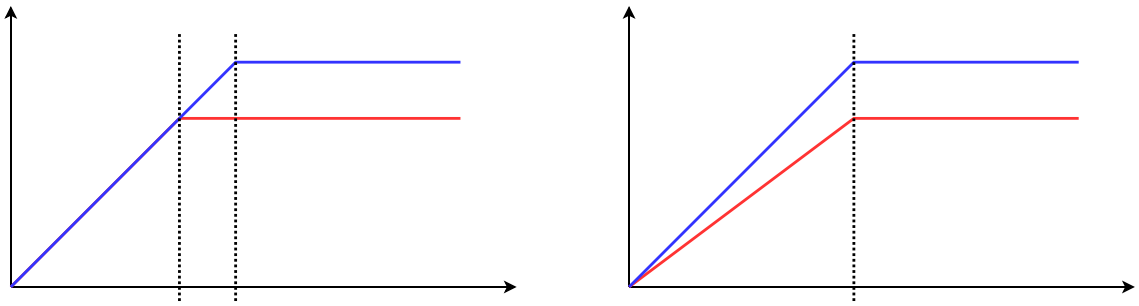

Abbildung 47: Konstante Beschleunigung versus konstante Beschleunigungszeit

Wie in Abbildung 47 dargestellt, würde ein Roboter, welcher gleiche Beschleunigungswerte für jede Achse hat, unterschiedliche Zielgeschwindigkeiten zu verschiedenen Zeitpunkten erreichen. Dies führt zu einem Fehler in der Positionierung, weil der Roboter für eine gewisse Zeit in die falsche Richtung fahren würde. Deshalb ist es von Nöten die beiden Achsen zu Synchronisieren. Dies geschieht in der *RoboSpeed* Klasse. Hierzu wird für jeden Roboter und jede Achse die maximal benötigte Zeit berechnet, die es erfordert, um die gesetzte Zielgeschwindigkeit zu erreichen. Das Maximum aller Roboter wird anschließend wiederrum verwendet, um die Beschleunigungswerte bezüglich jeder Achse jedes Roboters zu berechnen. Dies geschieht in der *calc_syncA* Funktion. Die achsen spezifischen Beschleunigungswerte werden anschließend verwendet, um die Geschwindigkeit des nächsten Regelintervalls zu berechnen.

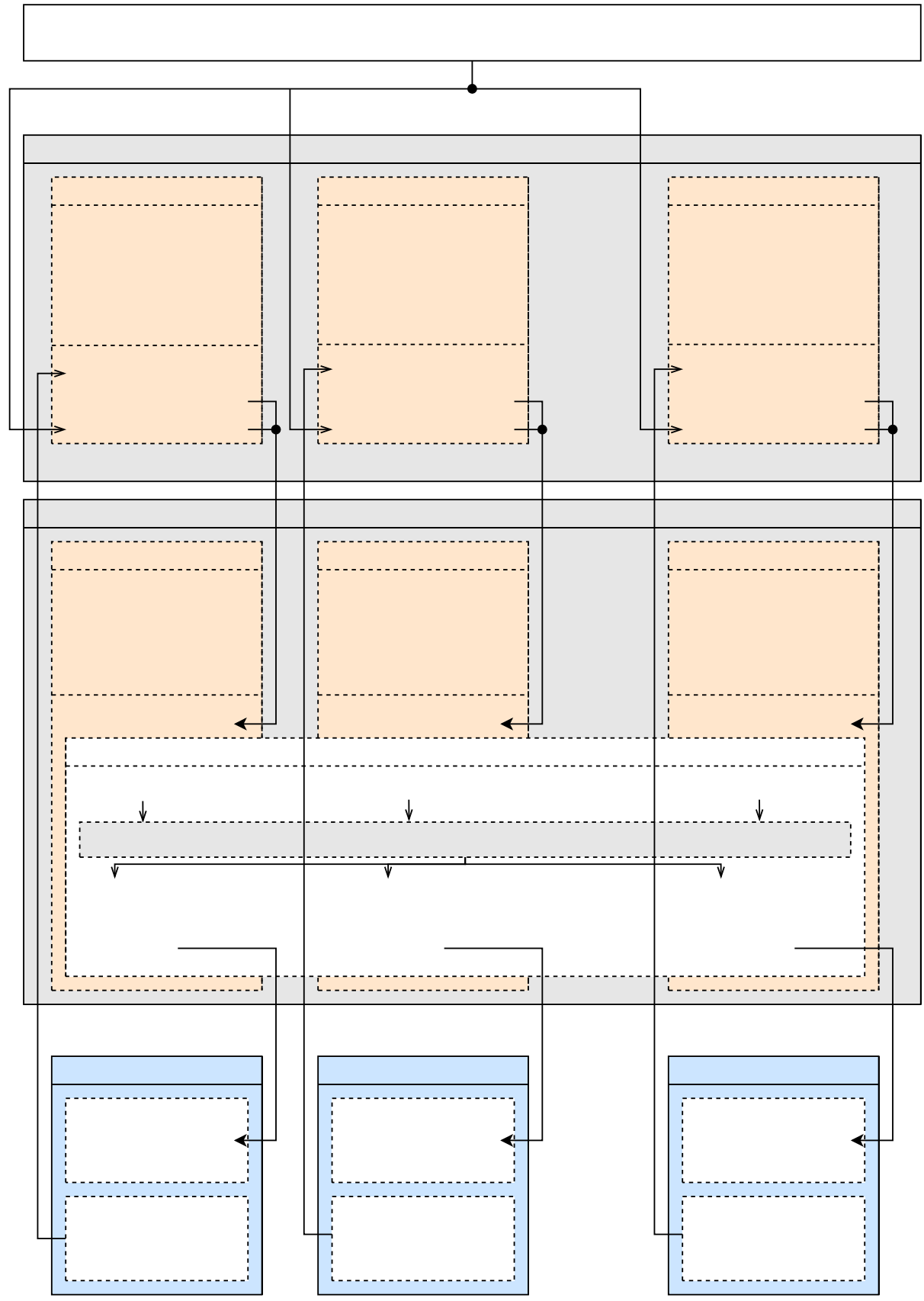

Abbildung 48: Programm Architektur HIVE-Controll-Script. Darstellung der Klassenabhängigkeiten und Übergabeparameter

In Abbildung 48 ist der Programmablauf des kompletten Testprogramms zu sehen. Jeder Roboter hat seine eigene Instanz im Programm. Jede Roboterinstanz besteht aus zwei Klassen: *Robo* und *RoboSpeed*. Im oberen Bereich einer jeden Klasseninstanz zu sehen sind die zur

Initialisierung benötigten Konstanten. Darunter aufgeführt sind die wichtigsten Funktionen. Andere Funktionen, welche zum Verständnis der Funktionsweise nicht beitragen, wurden hier nicht aufgeführt, können jedoch im Anhang eingesehen werden.

Dargestellt ist das Programm für beliebige Benutzereingaben. Um später jedoch vergleichbare Ergebnisse zu haben, um die verschiedenen Stufen des Programms besser beurteilen zu können, wurde auch noch ein anderes Programm so modifiziert, dass es möglich ist, eine vordefinierte Trajektorie mittels des oben beschrieben Programms ab zu fahren. Hierbei wird aber die *RoboSpeed*-Klasse übergangen, da sich sonst die Testvoraussetzungen durch unterschiedliche Abstände verändern könnten.

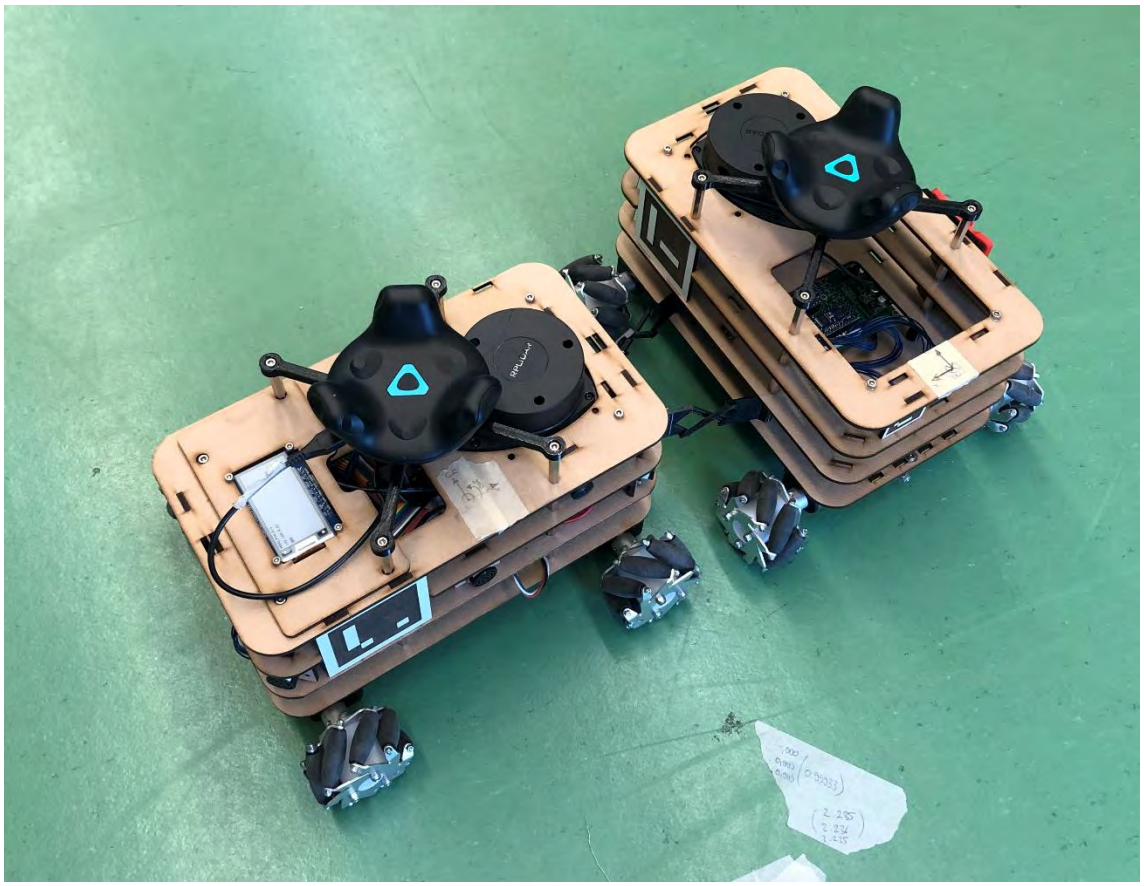

Abbildung 49: HIVE-BEE Kombination für den Multi-Roboter-Steuerungstest

Abbildung 49 zeigt die im Test verwendete HIVE-BEE Kombination. Links im Bild befindet sich der BEE- und rechts der HIVE-Roboter. Es handelt sich hierbei in beiden Fällen um Roboter der Version 6 wobei einer mit Teilen des geplanten HIVE upgrade versehen wurde.

Das Upgrade umfasst zwei bis drei Ladestationen, an denen BEEs andocken können, um mobil ihren Akku wieder aufzuladen. Das ermöglicht unter anderem gestrandete Roboter wieder einzusammeln oder die Reichweite des Schwarms zu erhöhen.

5 Bewertung

Im Folgenden sollen alle Änderungen und Maßnahmen, welche ergriffen wurden, auf ihre Wirksamkeit untersucht werden. Hierzu werden ähnliche Trajektorien zu denen aus Kapitel 3.3 mit dem Roboter PIA6-1, dem ersten PIA-Roboter der Version 6, gefahren und hinsichtlich der daraus resultierenden Fehler diskutiert. Im Test0 wird hierbei die Wirksamkeit der Vorsteuerung genauer beleuchtet. Bei Test1 werden mehrere identische Fahrten wiederholt, um die Streuung der Zielpunkte vergleichen zu können. Test2 behandelt Fehler, die bei unterschiedlich weiten Fahrten auftreten können und den dabei zu erwartendem Fehler. In Test3 geht es um die Untersuchung der zurückgelegten Strecke bei unterschiedlichen Geschwindigkeiten. Hierbei wird geprüft, welchen Einfluss verschiedene Geschwindigkeiten auf die zurückgelegte Strecke haben. In einem abschließenden Test für das einzelne System, wird der Roboter PIA004 dem Roboter PIA6-1 direkt gegenübergestellt.

Ebenso wird hier das Testprogramm *HIVE-Control,* welches in Kapitel 4.3.2 beschrieben wurde, erprobt und aufgezeigt, welche Aspekte bei einer Formationsfahrt zu beachten sind. Hierbei wird eine feste Trajektorie in vier verschiedenen Szenarien abgefahren. Bei zwei Fahrten haben der HIVE und der BEE keinen physikalischen Kontakt und fahren einmal Open- und einmal Closed-Loop. In den anderen beiden Szenarien wird das mobile Laden mit Open- und Closed-Loop erprobt. Hierzu sind die beiden Roboter durch den Ladestecker verbunden, während sie die gleiche Trajektorie abfahren.

5.1 Fahrverhalten von PIA6-1

In den folgenden Tests wird ein einzelner Roboter hinsichtlich seiner Fahreigenschaften untersucht. Viele der zuvor durchgeführten Maßnahmen wurden getroffen, um die Fahrleistungen des Roboters soweit zu verbessern, dass es mit ihm möglich ist, präzise, geplante Trajektorien abzufahren.

5.1.1 Test0: freies Fahren und Reglerverhalten

In einer ersten Untersuchung soll das allgemeine Reglerverhalten untersucht und bewertet werden. Hierzu wird der Roboter über die Tastatureingabe so gesteuert, dass möglichst viele verschiedene Szenarien mit sich überlagernden Bewegungen gefahren werden. Anschließend werden die so ermittelten Daten analysiert und nach größeren Abweichungen zwischen Führungsgröße und Regelgröße gesucht. Zur Darstellung wird die gesetzte Raddrehzahl $\omega_{Set}(t)$ über die Zeit aufgetragen. Gleiches wird auch mit der gemessenen Raddrehzahl $\omega_{Ist}(t)$ getan. In einem letzten Schritt wird noch die Differenzraddrehzahl $\omega_{diff}(t)$ berechnet.

$$\omega_{Diff}(t) = \omega_{Ist}(t) - \omega_{Set}(t) \tag{42}$$

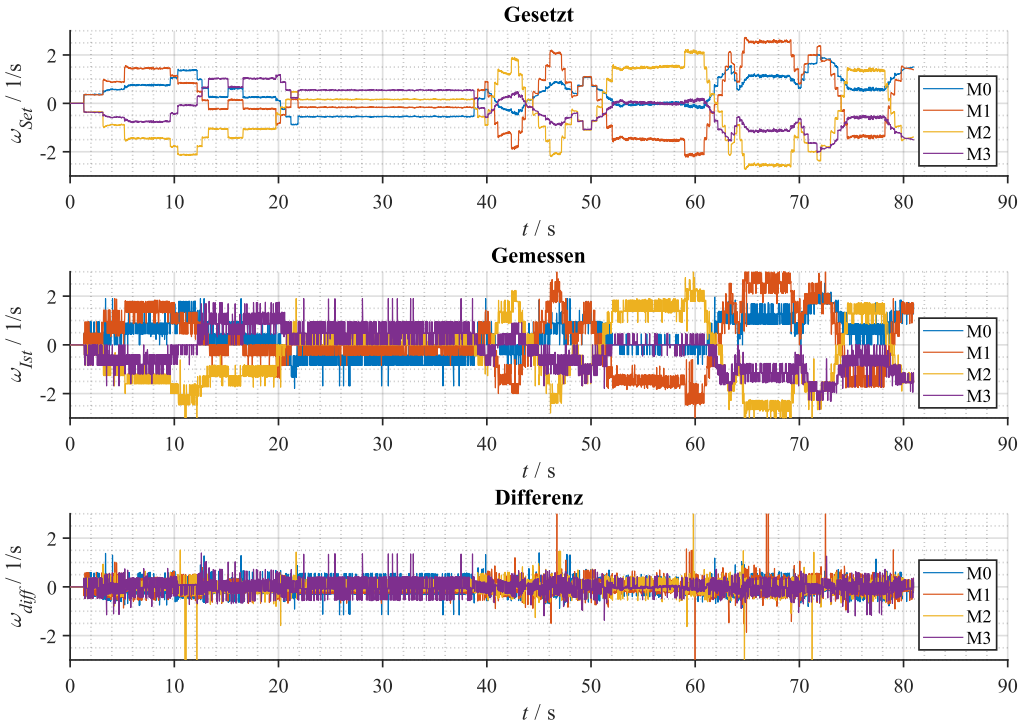

Abbildung 50: Vergleich Soll- zu Ist-Drehzahl des Roboters PIA6

Wie in Abbildung 50 zu sehen, folgt die gemessene Drehzahl der gesetzten Drehzahl mit geringem Fehler, was daran zu erkennen ist, dass die Differenz aus beiden Drehzahlen weitestgehend bei um null liegt. So liegen die mittleren Drehzahl Differenzen bei:

$$\omega_{diff,0} = 0{,}01\frac{1}{s} \pm 0{,}58\frac{1}{s}$$

$$\omega_{diff,1} = 0{,}00\ \frac{1}{s} \pm 0{,}70\frac{1}{s}$$

$$\omega_{diff,2} = -0{,}00\frac{1}{s} \pm 0{,}62\frac{1}{s}$$

$$\omega_{diff,3} = -0{,}01\frac{1}{s} \pm 0{,}59\frac{1}{s}$$

Auffällig ist jedoch, dass eine hohe Streuung der gemessenen Drehzahl zu sehen ist. Dieses ist auf die hohe Updaterate des PID-Loops zurück zu führen. Besonders bei niedrigen Drehzahlen reicht so die Auflösung des Quadraturencoders nicht aus, um in einem Updateintervall die Geschwindigkeit genügend präzise auf zu lösen. So liegt die kleinste zu messende Drehzahl nach Formel (23) bei $\omega_{min} \approx 1\frac{1}{s}$.

Eine weitere Untersuchung, welche diese Fahrt zulässt, ist die Untersuchung der Wirksamkeit der in Kapitel 4.1.2 beschriebenen Vorsteuerung. Da in dieser Fahrt verschiedenste Fahrtrichtungen und Geschwindigkeiten verwendet wurden und diese sich auch überlagern, kann untersucht werden, welchen Anteil die Vorsteuerung und der PID-Regler an der Stellgröße

haben. Eine gut eingestellte Vorsteuerung übernimmt den größten Teil der Last, sodass der PID-Regler lediglich äußere Einflüsse kompensieren muss.

So wird im konkreten Fall die Stellgröße durch PWM_{Sum} beschrieben und zeigt an, zu wieviel Prozent der Motorcontroller ausgelastet ist.

$$PWM_{Sum}(t) = PWM_{VS}(t) + PWM_{PID}(t) \tag{43}$$

Die Stellgröße PWM_{Sum} setzt sich aus dem durch die Vorsteuerung beigesteuerten Anteil PWM_{VS} und dem durch den PID-Regler beigesteuerten Anteil PWM_{PID} zusammen.

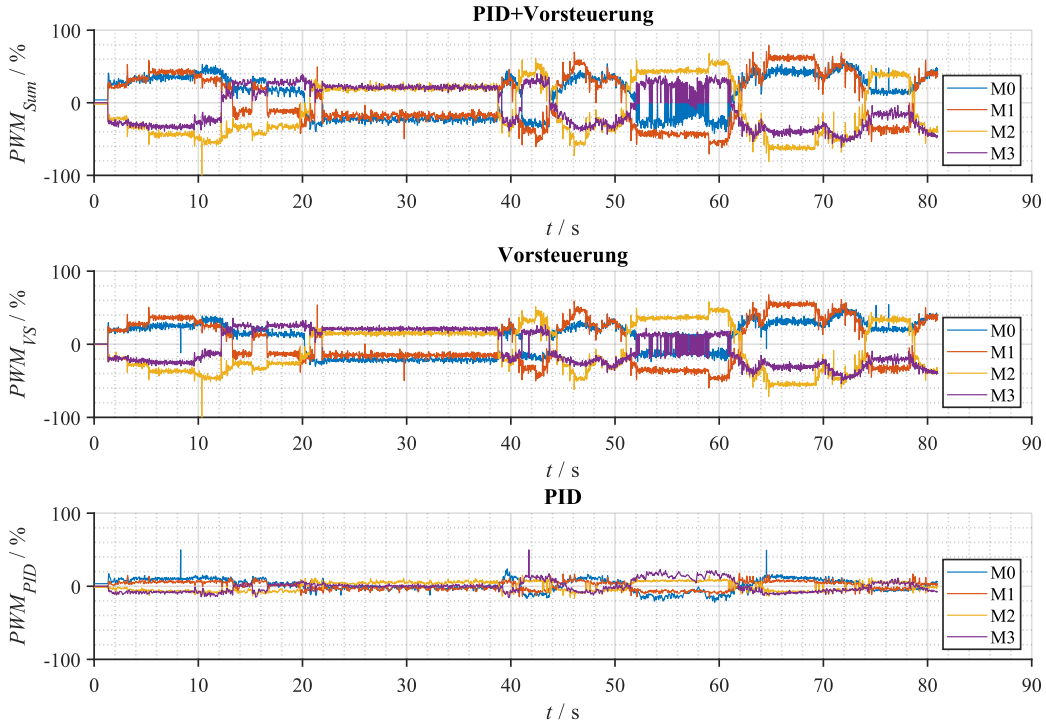

Abbildung 51: Wirksamkeitsuntersuchung Vorsteuerung

In Abbildung 51 ist zu sehen, dass die Last, welche der PID-Regler tragen muss, zu den meisten Zeitpunkten nahezu null ist. Ein Großteil der Last wird durch die Vorsteuerung übernommen.

Die Maßnahme zeigt demnach den gewünschten Effekt, dass die Vorsteuerung den PID-Regler entlastet.

5.1.2 Test1: xy-Richtung bei konstanter Strecke und Geschwindigkeit

In diesem Test wird untersucht, wie groß die Wiederholgenauigkeit des Roboters PIA6-1 ist. Hierbei wird anders als in Kapitel 3.3.1 kein Unterschied zwischen Fahrten in positive bzw. negative x- bzw. y-Richtung gemacht. Um den Effekt der in Kapitel 4.1.3 beschrieben IMU-Kursstabilisierung zu ermitteln, werden diese Fahrten mit und ohne aktivierter Kursstabilisierung gefahren. Die zurückgelegte Strecke beträgt $S = 1000mm$ und wird mit

einer Geschwindigkeit von $v = \pm 0{,}10\,\frac{\mathrm{m}}{\mathrm{s}}$ zurückgelegt. Zu beachten ist, dass bei diesem Test keine Beschleunigung bzw. Verzögerungsphasen im Test-Programm vorgesehen sind. Somit wird zum Startzeitpunkt die Zielgeschwindigkeit gesetzt und nach Ablauf der Fahrtzeit wird diese wieder auf null gesetzt.

Endpunkte p_n werden wie zuvor in Kapitel 3.3 mit Hilfe der Formel (14) angegeben.

Der Punkt p_m beschreibt den mittleren Endpunkt aller Punkte p_n und wird mittels der Formel (15) ermittelt.

Um die Streuung der Endpunkte beschreiben zu können, wird der zu erwartende Fehlerradius zum Punkt p_m angegeben. Hierzu werden die Formeln (16) und (17) herangezogen.

Der Radius r_e berechnet sich nach der Formel (11). Um x- und y-Fahrten in einem Diagramm darstellen zu können, sind die Messergebnisse für Fahrten in y-Richtung um 90° auf die x-Achse gedreht. Und alle Fahrten in negative Achsrichtung wiederum um 180°.

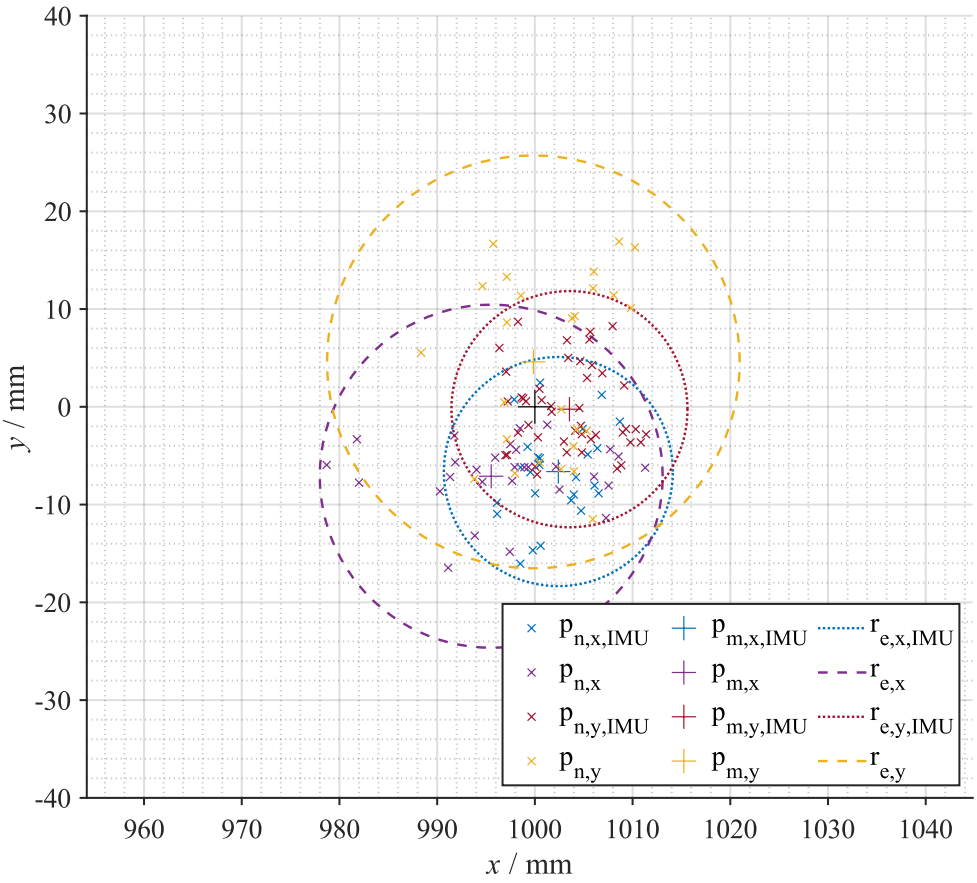

Abbildung 52: Zielpunkte bei Fahrten in X und Y-Richtung mit und ohne IMU-Stabilisierung

Wie in Abbildung 52 zu sehen ist, liegen alle Zielpunkte dicht beieinander. Mit aktivierter IMU-Kursstabilisierung lässt sich die Streuung noch einmal deutlich verringern.

Zwischen Fahrten in x- und y-Richtung ist aber kein signifikanter Unterschied mehr zu erwarten.

Tabelle 4: Auswertungstabelle Test1 von PIA6-1

	x mit IMU	*x*	*y mit IMU*	*y*
p_m/ mm	$\begin{pmatrix}1002\\-7\end{pmatrix}$	$\begin{pmatrix}996\\-7\end{pmatrix}$	$\begin{pmatrix}1004\\0\end{pmatrix}$	$\begin{pmatrix}1000\\5\end{pmatrix}$
r_e/ mm	12	18	12	21

Diese Ergebnisse spiegeln sich auch in Tabelle 4 wider. So liegt der mittlere Zielpunkt nur ca. 1% der zurückgelegten Strecke von dem Zielpunkt entfernt und die Streuung ist mit ca. 1-2 % anzunehmen.

Vergleicht man dieses Ergebnis mit denen aus Kapitel 3.3.1, so wurde die Streuung um eine Zehnerpotenz verringert, was eine signifikante Verbesserung darstellt.

5.1.3 Test2: xy-Richtung bei konstanter Geschwindigkeit und variabler Strecke

In diesem Test wird ermittelt, welchen Einfluss variierende Strecken haben. Bei dem Roboter PIA004 war zu sehen, dass er durch die Totzeit zu Beginn der Fahrt nur für eine Geschwindigkeit und eine Strecke kalibriert werden konnte. Hier soll nun gezeigt werden, dass der Roboter PIA6-1 auch auf beliebigen Strecken abseits des ursprünglichen Kalibrierungsfalls gute Ergebnisse liefert. Die Geschwindigkeit, mit welcher der Roboter die verschiedenen Distanzen zurücklegen soll, beträgt genauso wie im Test1 $v = \pm 0{,}1\,\frac{\mathrm{m}}{\mathrm{s}}$.

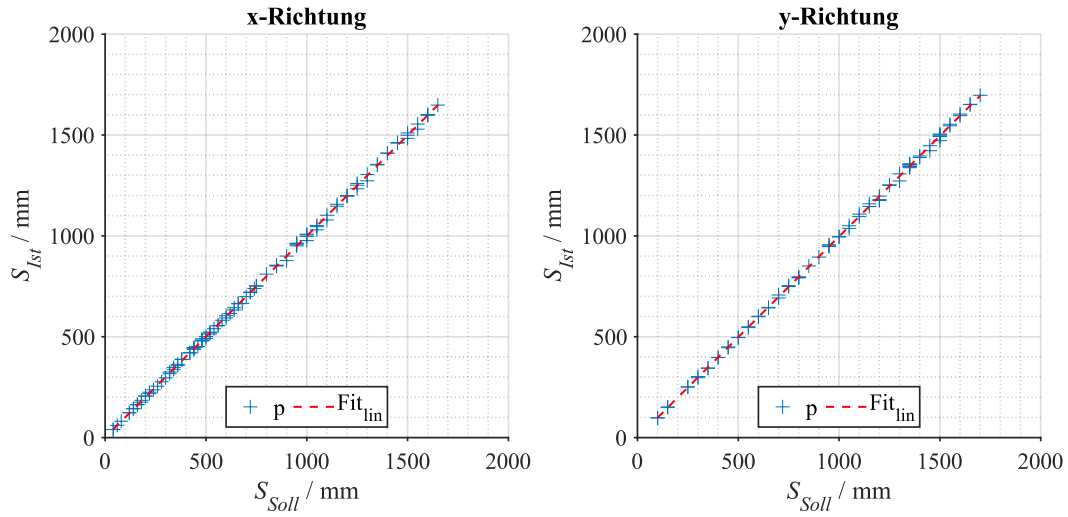

Abbildung 53: Vergleich Soll-Strecke zur Ist-Strecke in X und Y-Richtung bei konstanter Geschwindigkeit

In Abbildung 53 ist die Gegenüberstellung der Strecke S_{Soll} zur Ist-Strecke S_{Ist} zu sehen.

$$S_{Soll} = t \cdot v \tag{44}$$

Wobei t die Zeit beschreibt für die der Roboter die Geschwindigkeit v fährt. Auch hier gibt es idealisiert keine Beschleunigungs- bzw. Verzögerungsphasen bei der geplanten Trajektorie.

$$S_{Ist} = |p_n| \tag{45}$$

Die tatsächlich zurückgelegte Strecke berechnet sich mittels der Norm des Zielpunkts p_n. Die Ausgleichsgraden $Fit_{lin,x}$ und $Fit_{lin,y}$ haben folgende Gleichungen:

$$Fit_{lin,x}(S_{Soll}) = S_{Soll} \cdot 1{,}0 + 1{,}2 \text{ mm} \tag{46}$$

Und

$$Fit_{lin,y}(S_{Soll}) = S_{Soll} \cdot 1{,}0 - 1{,}0 \text{ mm} \tag{47}$$

Der y-Achsenabschnitt von $\approx \pm 1\text{mm}$ stellt keinen signifikanten Fehler da. Somit ist gezeigt, dass der Roboter PIA6-1 beliebige Strecken mit nur sehr geringen Fehlern zurücklegen kann. Dies stellt wiederrum eine deutliche Verbesserung der Fahrleistungen des Roboters PIA6-1 gegen über dem Roboter PIA004 dar.

In der Abbildung 53 wird lediglich auf die zurückgelegte Strecke geachtet, es wird jedoch keine Aussage bezüglich der Streuung mittels des Diagramms sichtbar. Hierzu sind die Daten hinsichtlich eines anderen Aspekts zu untersuchen. Deshalb wird der Abstand vom tatsächlichen zum theoretischen Zielpunkt bestimmt und durch die zurückgelegte Strecke geteilt. Das Ergebnis liefert dann den mittleren Abstand zum Zielpunkt pro gefahrene Strecke.

Berechnet wird hierzu der Abstand a_n zum theoretischen Zielpunkt mittels der Formel:

$$a_n = \left| p_n - S_{Soll} \cdot \frac{v}{|v|} \right| \tag{48}$$

Mit dem Geschwindigkeitsvektor

$$v = \begin{pmatrix} v_x \\ v_y \end{pmatrix} \tag{49}$$

und der Roboter Endposition p_n nach Formel (14) aus Kapitel 3.3. Die Sollstrecke S_{Soll} berechnet sich mit Hilfe der Formel (44).

Der mittlere Abstand a berechnet sich somit:

$$a = \overline{a_n} \tag{50}$$

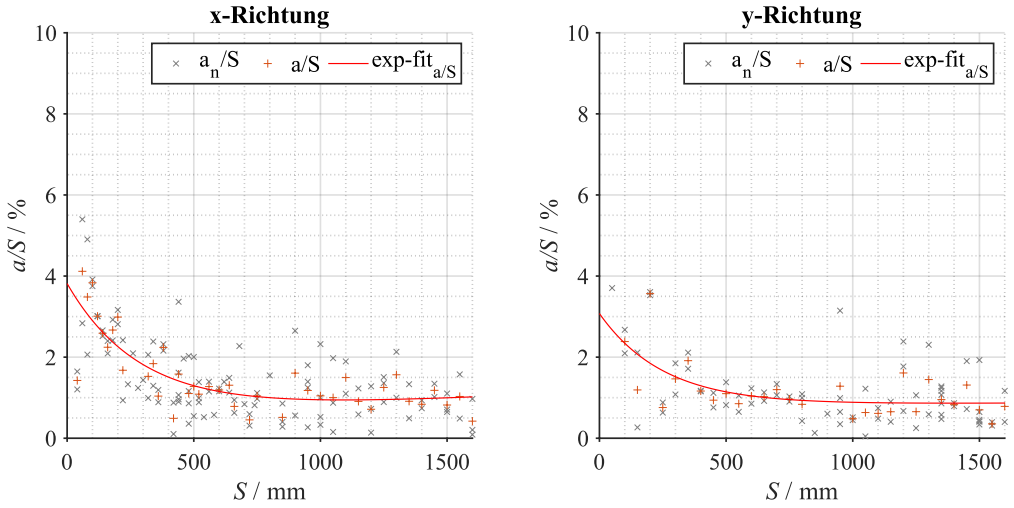

Abbildung 54: Prozentualer Fehlerradius bezogen auf die Soll-Strecke über verschiedene Sollstrecken, bei konstanter Geschwindigkeit

In Abbildung 54 ist zu sehen, dass der Abstand pro zurückgelegte Strecke $\frac{a}{S}$ zwischen theoretischem Zielpunkt und tatsächlichem Endpunkt erst ab einer Strecke von ca. 500mm mit ca. 1% anzunehmen ist. Nimmt man an, dass dieser Wert auch für größere Strecken weiterhin konstant bleibt, lassen sich damit Prognosen bezüglich des mittleren Abstandes anstellen. So würde es z.B. für eine Strecke von $S = 5\text{m}$ bedeuten, dass der mittlere zu erwartende Abstand zum Zielpunkt $a = 5\ \text{cm}$ beträgt solange mit einer Geschwindigkeit von $v = 0{,}1\,\frac{\text{m}}{\text{s}}$ gefahren wird.

Auch dieses spiegelt ein sehr gutes Resultat wider. Demzufolge können auch relativ große Strecken Open-Loop gefahren werden, ohne dass zu große Fehler entstehen. Ist man jedoch auf eine sehr hohe Präzision angewiesen, sind kurze Strecken zu wählen, welche Open-Loop gefahren werden.

5.1.4 Test3: xy-Richtung bei konstanter Strecke und variabler Geschwindigkeit

Im Test 3 geht es darum, zu zeigen, dass die zurückgelegte Strecke unabhängig von der zu fahrenden Geschwindigkeit ist. Hierzu wird eine definierte Strecke von $S = 1000\,\text{mm}$ mit verschiedenen Geschwindigkeiten von $v_{min} = 0{,}05\,\frac{\text{m}}{\text{s}}$ bis $v_{max} = 0{,}55\,\frac{\text{m}}{\text{s}}$ gefahren.

Anders als bei den vorangegangenen Tests wird hier die Strecken nicht mit konstanter Geschwindigkeit gefahren. Stattdessen wird mit einer Beschleunigung von $\dot{v} = 0{,}5\,\frac{\text{m}}{\text{s}^2}$ bis zur Zielgeschwindigkeit beschleunigt. Die Zielgeschwindigkeit wird dann so lange gehalten, bis mit dem Bremsen begonnen werden muss, um die Zielstrecke zu fahren. Gebremst wird wiederrum mit $\dot{v} = -0{,}5\,\frac{\text{m}}{\text{s}^2}$.

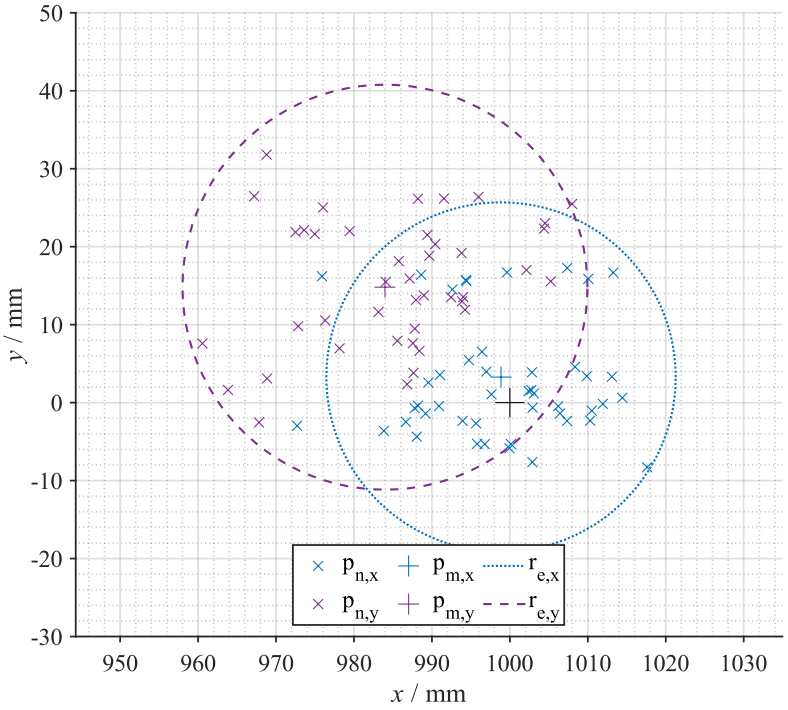

Abbildung 55: Zielpunkte für eine konstante Strecke von einem Meter bei Geschwindigkeiten von 0,05 m/s bis 0,55 m/s

In Abbildung 55 zu sehen sind die verschiedenen Endpunkte bei Fahrten von einem Meter. Hierbei errechnen sich die Endpunkte p_n mit Hilfe der Formel (14), der mittlere Endpunkt p_m mit Hilfe der Formel (15) und der zu erwartende Fehlerradius mit Hilfe der Formeln (16) und (17).

Tabelle 5: Auswertungstabelle für variable Geschwindigkeiten beim Roboter PIA6

	x	y
p_m/ mm	$\begin{pmatrix}998\\3\end{pmatrix}$	$\begin{pmatrix}984\\15\end{pmatrix}$
r_e/ mm	22	26

In Tabelle 5 ist zu sehen, dass die Streuung bei Fahrten in y-Richtung merklich größer ist als bei Fahrten in x-Richtung. Zwar hat die Streuung zugenommen und auch der mittlere Endpunkt liegt bei Fahrten in y-Richtung weiter von dem Zielpunkt entfernt, jedoch ist dies bei dem gewählten Geschwindigkeitsbereich ein sehr gutes Ergebnis.

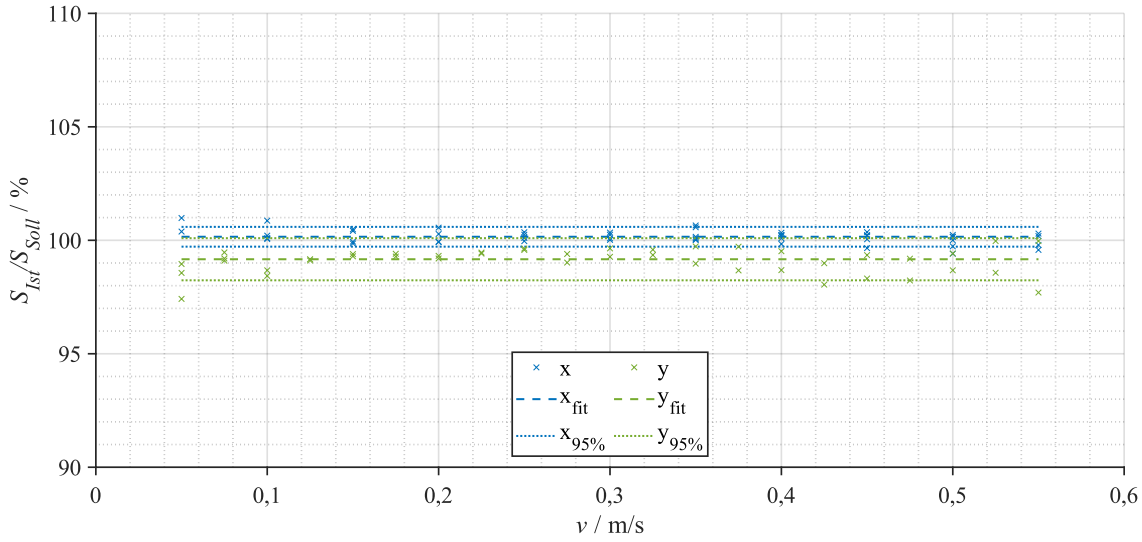

Abbildung 56: Prozentsatz der vom Sollwert zurückgelegten Strecke über verschiedene Geschwindigkeiten, sowie Mittelwert und Konfidenzintervall für x- und y-Fahrten

Da bei den in Abbildung 55 und Abbildung 56 gezeigten Fahrten eine Beschleunigungs- und Verzögerungsphase vorhanden war, gibt es numerische Fehler bei der Bestimmung der der Soll-Strecke. Um diese nicht als zusätzlichen Fehler in die Messung eingehen zu lassen, wird die numerisch berechnete Strecke als Soll-Strecke S_{Soll} angenommen. Hierzu wird die zum Zeitpunkt t_n vorliegende Geschwindigkeit $v(t_n)$ mit dem letzten Zeitintervall dt multipliziert und aufsummiert.

$$S_{Soll} = \sum v(t_n) \cdot dt \tag{51}$$

als Bezug verwendet. Hierbei entspricht das Zeitintervall dem Kehrwert der Aktualisierungsfrequenz der Geschwindigkeit. In diesem Fall $f_{refresh} = 20$ Hz.

$$dt = \frac{1}{f_{refresh}} \tag{52}$$

Vergleicht man die beiden Strecken S_{Soll} und S_{Ist}, erhält man das in Abbildung 56 zu sehende Diagramm. Hierbei ist zu sehen, dass die zurückgelegte Strecke unabhängig von der Geschwindigkeit konstant bleibt. Die Werte für die dargestellten Ausgleichsgraden sind:

$$x_{fit} = \frac{S_{Ist,x}}{S_{Soll,x}} = 100{,}2\ \% \tag{53}$$

Und bei Fahrten in y-Richtung bei:

$$y_{fit} = \frac{S_{Ist,y}}{S_{Soll,y}} = 99{,}2\ \% \tag{54}$$

Alle Tests zeigen eine signifikante Verbesserung der Fahrleistungen des Roboters PIA6-1 im Vergleich zum Roboter PIA004. Anders als zuvor ist der Roboter PIA6-1 in der Lage, mit nahezu jeder Geschwindigkeit jeden beliebigen Punkt anzufahren.

5.1.5 Test4: Vergleich zwischen PIA004 und PIA6-1 mittels dynamischer Fahrt

Um die Unterschiede zwischen PIA004 und PIA6-1 zu verdeutlichen wird ein abschließender Test durchgeführt. Hierbei wird der Roboter PIA004 und PIA6-1 mit der gleichen Kreistrajektorie angesteuert. Die Kreisbahn entsteht dadurch, dass verschiedene Geschwindigkeiten in x- und y-Richtung überlagert werden. Die zu fahrende Geschwindigkeit beträgt hierbei:

$$v = \begin{pmatrix} v_{max} \cdot \sin(\omega \cdot t) \\ v_{max} \cdot \cos(\omega \cdot t) \\ 0\,\frac{\text{rad}}{\text{s}} \end{pmatrix} \tag{55}$$

Mit $v_{max} = 0{,}1\,\frac{\text{m}}{\text{s}}$ muss für einen Radius von $r = 0{,}5$ m eine Winkelgeschwindigkeit von $\omega = 0{,}2\,\frac{\text{rad}}{\text{s}}$ über einen Zeitraum von $t = 31{,}42\text{s}$ gefahren werden.

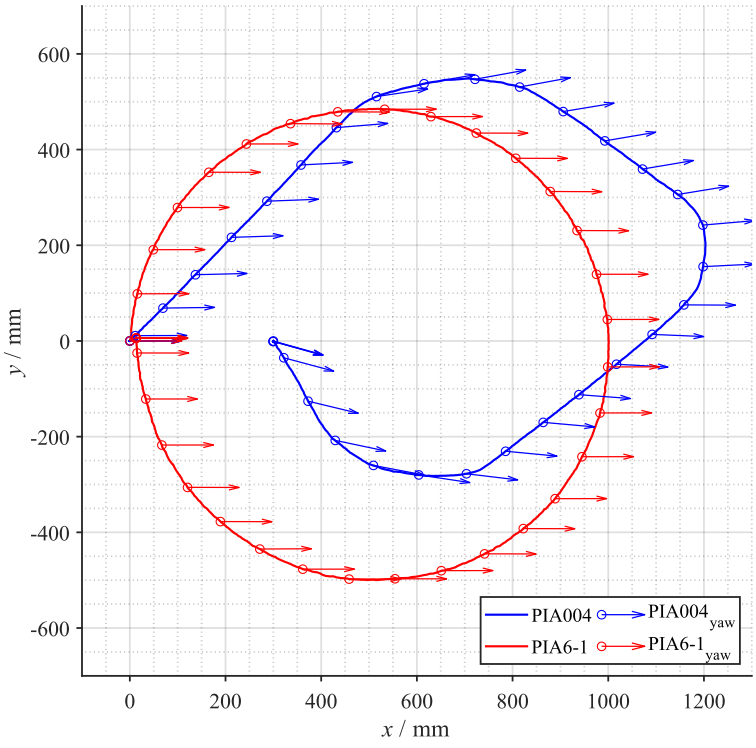

Abbildung 57: Vergleich PIA004 und PIA6-1 bei einer Kreisfahrt

Vergleicht man die Beiden Fahrten aus Abbildung 57 ist deutlich zu erkennen, dass die Fahrt mit dem Roboter PIA6-1 deutlich mehr einen Kreis ähnelt, als es die Fahrt des Roboters PIA004 tut. Nicht nur der Start und Zielpunkt liegen deutlich dichter beieinander, sondern auch die Orientierung wird von dem Roboter PIA6-1 deutlich besser gehalten.

Betrachtet man Jede einzelne Komponente der Roboter-Pose über die Zeit wird diese Verbesserung noch einmal deutlich.

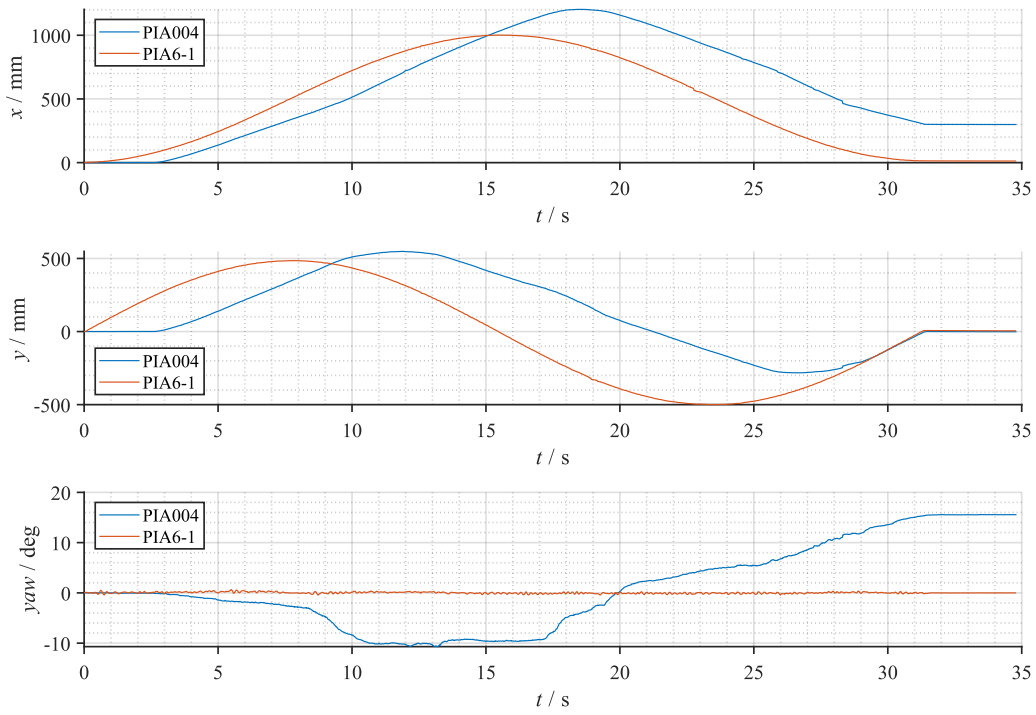

Abbildung 58: Vergleich PIA004 und PIA6-1 bei einer Kreisfahrt aufgeteilt

In Abbildung 58 ist zu erkennen, dass für den Roboter PIA6-1 die Position in x- und y-Richtung über die Zeit Sinus beziehungsweise Cosinus förmig verläuft. Auch entspricht hierbei die eingestellte Amplitude circa dem eingestellten Radius $r = 0{,}5\text{m}$. Diese Beobachtung kann für den Roboter PIA004 nicht geteilt werden. Beispielsweise stellt das Anfahrverhalten in x- und y-Richtung ein Problem dar. Durch das verzögerte Anfahren muss die Geschwindigkeit direkt auf v_{max} springt und dann gehalten werden. Daraus resultiert eine Trajektorie die eher einer Raute als einem Kreis ähnelt.

Am stärksten unterscheiden sich jedoch die beiden Fahrten durch die Orientierung der Roboter. Am Ende der Fahrt von PIA6-1 liegt Verdrehung bei nur $yaw_{PIA6} = 0°$. Bei dem Roboter PIA004 jedoch bei $yaw_{PIA004} = 16°$, dies führt zusätzlich zu Fehlern auf der x- und y-Achse.

Zusammenfassend zeigen alle Tests auf, dass der Roboter PIA6-1 in jeder Hinsicht sehr gute Fahrleistungen erzielt. Er ist bei statischen sowie bei dynamischen Fahrten dem Roboter PIA004 weit überlegen und kann somit auch im Schwarm eingesetzt werden.

5.2 Multi-Robot-Protokoll

Im Folgenden wird mittels des HIVE-Control Programms, welches in Kapitel 4.3.2 beschrieben ist, die Funktionsweise der Multi-Roboter-Steuerung genauer betrachtet. Hierbei liegt das Hauptaugenmerk auf den verschiedenen Aspekten, welche bei der selbstgestellten Aufgabe des mobilen Ladens eine Rolle spielen. Damit ein Roboter während der Fahrt durch einen anderen geladen werden kann, dürfen sich die beiden Roboter nur minimal relativ zueinander bewegen, da sonst die Verbindung zwischen den beiden Robotern unterbrochen werden kann. Es wird gezeigt, dass ein reiner P-Regler ausreichend ist, um eine stabile Verbindung zwischen zwei Robotern des Typs PIA6 zu schaffen.

Im Folgenden bedeutet „physikalischen Verbindung“, dass die beiden Roboter mittels eines magnetischen Steckers verbunden sind. Dieser gewährleistet die Übertragung geringer Kräfte, lässt sich aber allein durch den Antrieb der Roboter lösen. Somit können Roboter diese Verbindung eigenständig herstellen und trennen. Die Roboter werden somit nicht zu einer Einheit, wie es z.B. bei einer Verankerung der Fall wäre.

5.2.1 Test5: HIVE-Control ohne AR-Korrektur

In diesem Test fährt der HIVE-Roboter eine Kreistrajektorie mit einem Radius $r = 0{,}5$ m. Seine x-Achse entspricht hierbei dem Radius und ist nach außen gerichtet. Seitlich von ihm fährt ein BEE-Roboter in deinem Winkel von $Offset_{yaw} = 270°$ eine Trajektorie, sodass die relative Position konstant bleibt. Dieser Test wird einmal durchgeführt mit einem Abstand, sodass der BEE eine physikalische Verbindung zum HIVE mittels des Ladesteckers herstellt. In einer zweiten Fahrt wird der Abstand um 2 cm vergrößert, so dass sich die Roboter nicht berühren. Die Roboter werden manuell zueinander ausgerichtet, so dass Abweichungen möglich sind.

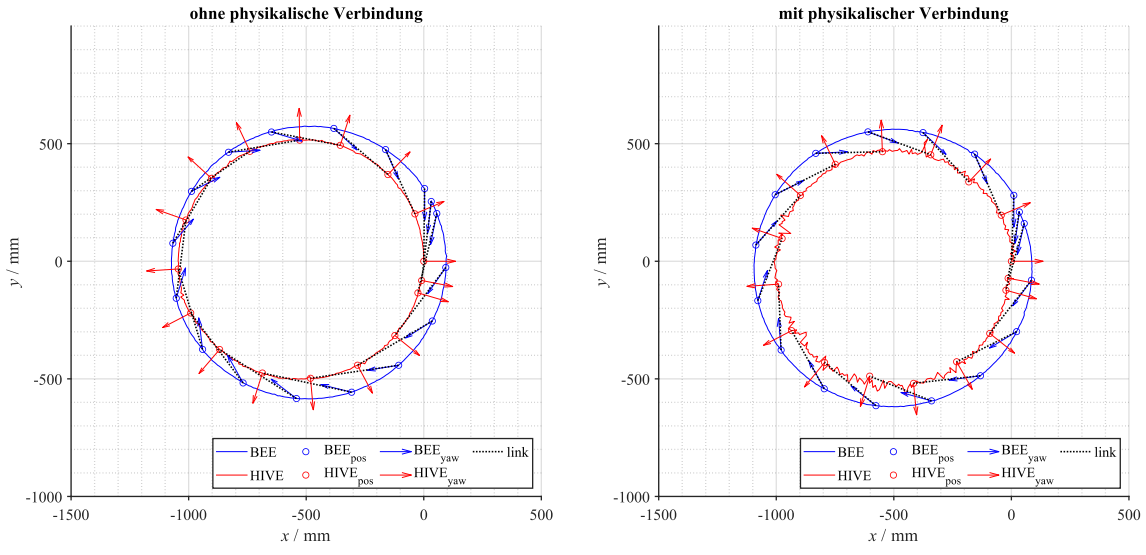

Abbildung 59: Bahnkurve PIA6 HIVE/BEE-Open-loop ohne und mit physikalischer Verbindung

In Abbildung 59 zu sehen sind die beiden kombinierten Bahnkurven der Roboter. Der Pfeil kennzeichnet jeweils die Richtung, in welche die x-Achse des Roboters zeigt. Durch die gepunktete Linie *link* ist gekennzeichnet, welche Positionen bzw. Orientierungen jeweils zueinander gehören.

Zu erkennen ist, dass die Roboter in beiden Fällen Kreise fahren, jedoch sind die Kreise bei der Variante ohne physikalische Verbindung nicht konzentrisch. Dies liegt vermutlich maßgeblich an der anfänglichen Positionierung beider Roboter zueinander. Nur kleine Winkel oder Positionierungsfehler sorgen für größere Abweichungen.

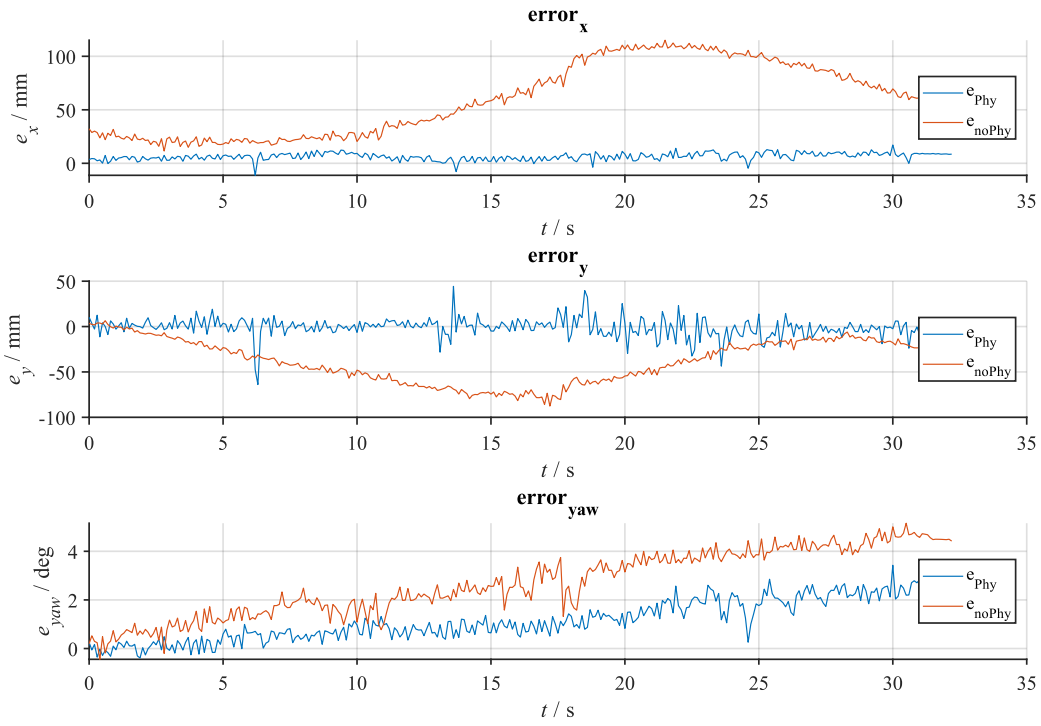

Abbildung 60: Fehlerdarstellung für eine Kreisfahrt mit zwei Robotern, Open-Loop, Offset statisch, mit und ohne physikalische Verbindung

In Abbildung 60 ist eine Gegenüberstellung der verschiedenen Abweichungen des BEEs von seiner Soll-Position zu sehen. Diese folgen der in Abbildung 46 dargestellten Konvention zur Angabe von Positionierungsfehlern. Hierbei wird die Fahrt mit physikalischer Verbindung e_{Phy} und ohne physikalische Verbindung e_{noPhy} in einem gemeinsamen Diagramm dargestellt.

Bei der Fahrt mit Verbindung sind die Positionierungsfehler in x- und y-Richtung laut VR-Tracker nahezu null. Lediglich ein Winkelfehler von ca. 2° stellt sich ein. Verglichen dazu sind die Positionierungsfehler ohne eine Verbindung deutlich größer. Diese schwanken sinusförmig zwischen ca. 0 mm und 100 mm, was dadurch zu erklären ist, dass sich die beiden Roboter auf zwei nicht konzentrischen Kreisbahnen bewegen. Ebenso ist der Winkelfehler ca. doppelt so groß wie bei der Fahrt mit physikalischer Verbindung.

Im Szenario, in dem zwischen den Robotern eine physikalische Verbindung besteht, trennen sich die beiden Roboter über den gesamten Testzeitraum nicht. Lediglich der Winkelfehler kann langfristig zu einer Trennung der beiden Roboter führen.

5.2.2 Test6: HIVE-Control mit AR-Korrektur

In diesem Test entspricht die geplante Trajektorie der aus Kapitel 5.2.1. Der einzige Unterschied besteht darin, dass die Ist-Position mittels eines einfachen P-Reglers korrigiert wird, so dass der Roboter nicht von seiner Soll-Position abweicht. Gezeigt wird hierbei, dass die Ladeverbindung selbst ohne Verankerung zwischen den beiden Robotern dauerhaft bestehen bleibt, so dass auch während der Fahrt geladen werden kann.

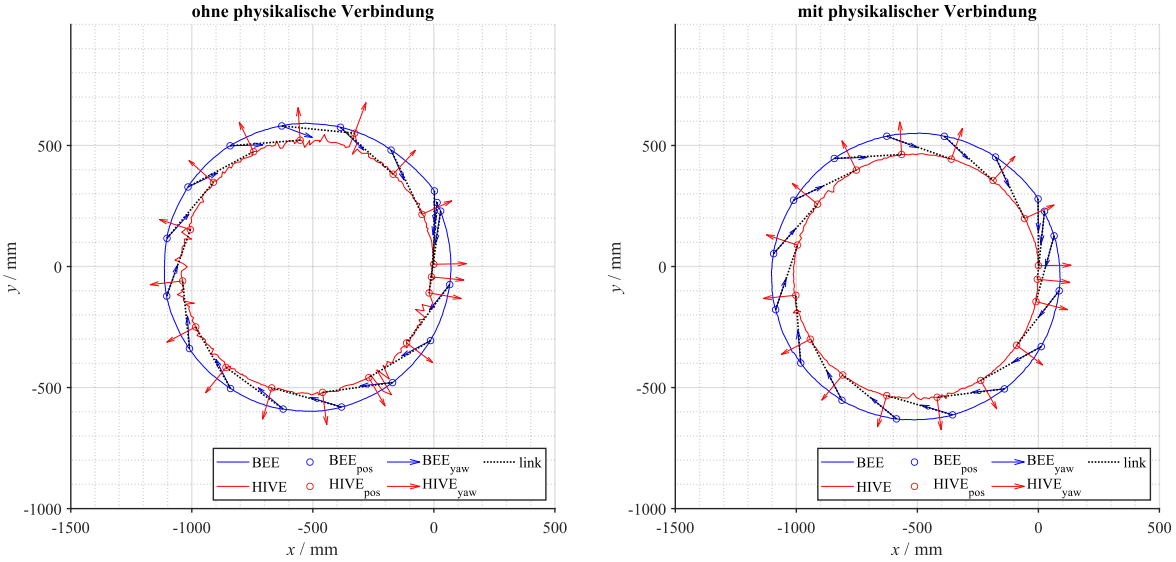

Abbildung 61: Bahnkurve PIA6 HIVE/BEE-Closed-loop mit und ohne Physikalische Verbindung

Wie in Abbildung 61 zu sehen fahren die beiden Roboter im Closed-Loop-Szenario in beiden Fällen konzentrische Kreise. Hierbei ist zusehen, dass, im Fall ohne physikalische Verbindung zwischen den Robotern, der HIVE, welcher eine nicht korrigierte Fahrt macht, nach VR-Tracker-Daten mitunter extreme Abweichungen vom Idealkreis aufweist.

Diese Abweichungen decken sich nicht mit der beobachteten Fahrt und sind demnach auf VR-Tracker-Fehler zurückzuführen. Um weitere Aussagen über die entstandenen Fehler treffen zu können, werden auch hier die Positionierungsfehler des BEE-Roboters bezüglich seiner Soll-Position über die Zeit aufgetragen. Anders als im Test 4 werden hier aber nun nicht die beiden Fahrten gegenübergestellt, sondern die beiden Fehler, welche durch die verschiedenen Sensoren ermittelt worden sind.

Hierbei entspricht e_{VR} dem mittels VR-Tracker ermittelten Fehler, und wird als der tatsächliche Fehler angenommen. Hierdurch lässt sich nun der durch den AR-Marker ermittelte Fehler e_{AR} beurteilen. Dies ist interessant, da die Positionskorrektur des Roboters ausschließlich über den durch die AR-Marker bestimmten Fehler funktioniert.

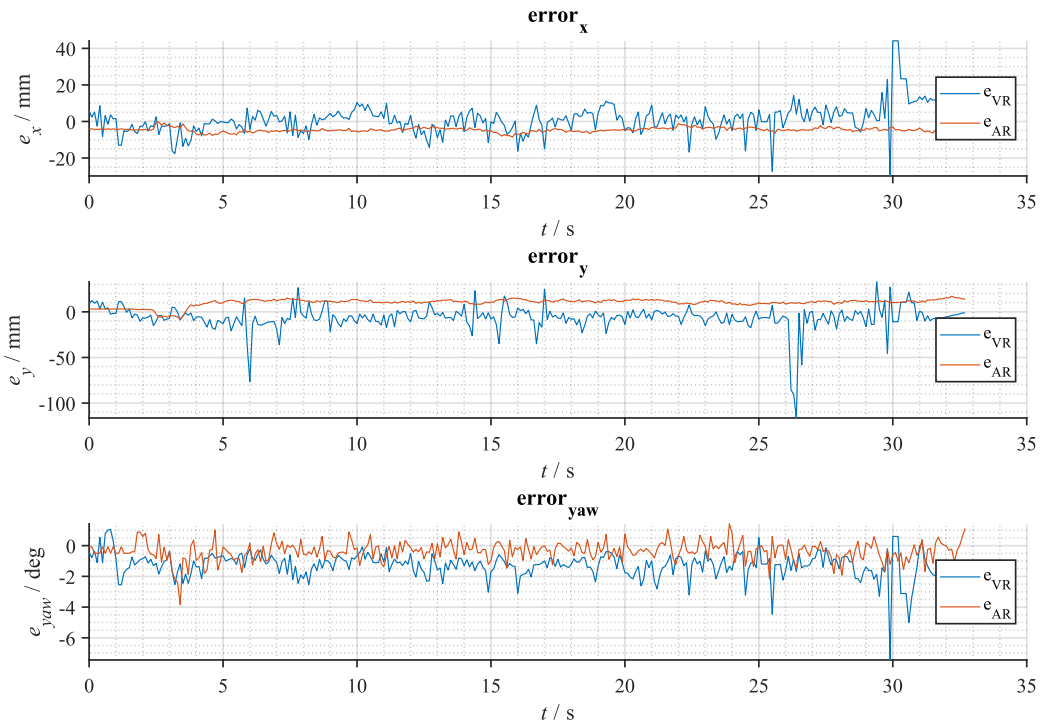

Abbildung 62: Fehlerdarstellung für eine Closed-Loop-Kreisfahrt mit zwei Robotern mit statischem Offset ohne physikalische Verbindung (Testfahrt Regelkreis)

Wie in Abbildung 62 zu sehen, ist der Regelkreis in der Lage, den Fehler über die gesamte Fahrt konstant zu halten. Ebenso decken sich die durch die beiden Messsysteme ermittelten Fehler weitestgehend.

Auffällig ist hier jedoch, dass der mittels VR-Tracker bestimmte Fehler zum Zeitpunkt $t \approx 6\,\mathrm{s}$ und $t \approx 26\,\mathrm{s}$ mitunter groß ist. Dieses ist aber wie zuvor erwähnt auf VR-Tracker-Fehler zurückzuführen. Zum einen, weil der Roboter nicht innerhalb einer Sekunde eine Strecke von 150 mm - 200mm zurücklegen kann, zum anderen, weil der Fehler mittels AR-Marker nicht zu messen war.

Vergleicht man die Ergebnisse mit denen aus Kapitel 5.2.1, ist eine deutliche Verbesserung zu beiden anderen Fahrten zu erkennen. Dadurch ist gezeigt, dass der Regelkreis seinen Zweck erfüllt. Dieses Szenario entspricht dennoch nicht dem geplanten Einsatzziel der Ansteuerungslogik, sondern soll lediglich Probleme bei der Regelung aufzeigen, da im vorangegangenen Test gezeigt werden konnte, dass eine physikalische Verbindung zu signifikant besseren Ergebnissen führt.

In einem letzten abschließenden Test soll nun noch gezeigt werden, dass die beiden Roboter tatsächlich während der Fahrt eine konstante Verbindung herstellen können, obwohl sie sich nicht miteinander verankern.

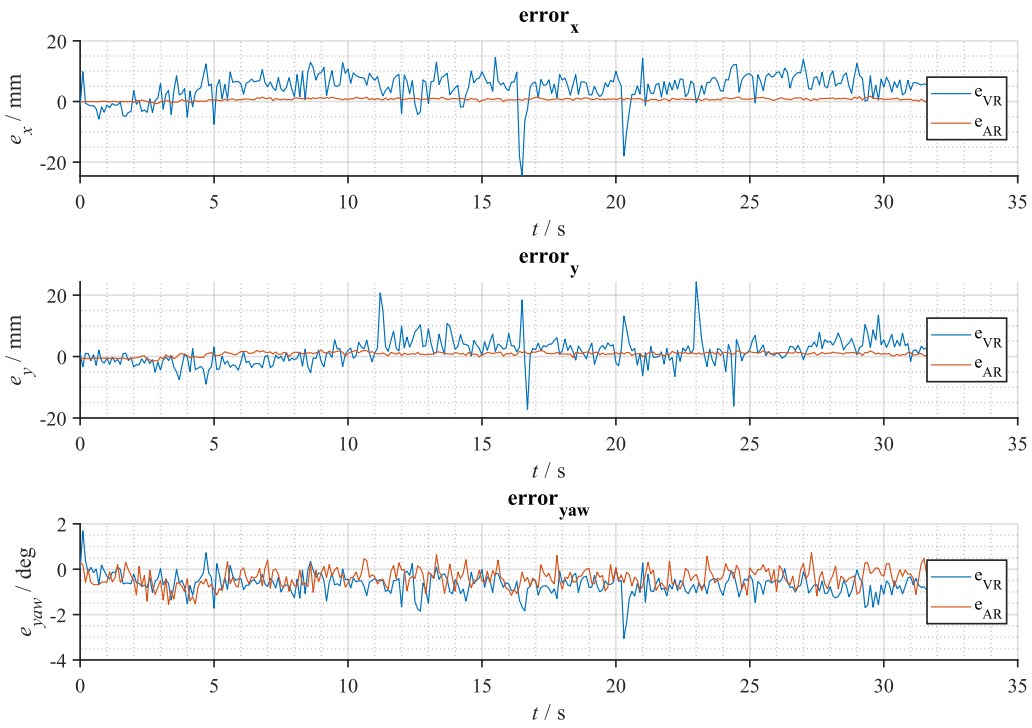

Abbildung 63: Fehlerdarstellung für eine Closed-Loop-Kreisfahrt mit 2 Robotern mit statischem Offset mit physikalischer Verbindung (Ladefall)

Wie in Abbildung 63 zu sehen können in diesem Szenario ebenfalls alle Fehler konstant auf 0 gehalten werden, was die Grundvoraussetzung für mobiles Laden darstellt. Wiederrum sind wie zuvor VR-Tracker-Fehler zu erkennen, diese decken sich aber wie auch zuvor nicht mit den visuellen Beobachtungen sowie den durch den AR-Marker ermittelten Fehlern.

Mit den Tests konnte gezeigt werden, dass die umgesetzte Multi-Roboter-Steuerung auch in der Praxis funktioniert. Sie ermöglicht es präzise, mit mehreren Robotern simultan zu Fahren. Auch das Konzept zur Positionsregelung mittels eines einfachen P-Reglers erweist sich als Erfolg.

6 Schlussbetrachtung

In diesem Kapitel wird eine abschließende Betrachtung getätigt und ein Ausblick für zukünftige Ziele gegeben.

6.1 Fazit

Das Ziel dieser Bachelorarbeit war es, eine Möglichkeit zu finden, mehrere Roboter zeitgleich individuell zu steuern. In ersten Tests wurde schnell klar, dass der Stand des Roboters den Anforderungen nicht entsprach, um ein solches Vorhaben umzusetzen und zu zufriedenstellenden Ergebnissen zu kommen.

Auf Grund dessen wurde der komplette Roboter überarbeitet. Angefangen beim Erstellen eines neuen Systemdesigns, welches in Zusammenarbeit mit Finn Weithoff erstellt wurde. Ebenso wurde basierend auf dem neuen Systemdesign ein neuer Schaltplan erstellt und ein Control-Board-Prototyp entwickelt, welcher alle Komponenten des Systems miteinander verbindet. Auf Grund dieses Prototyps wurde durch Tobias Röbke ein PCB erstellt, welches heute in PIA6 Robotern zum Einsatz kommt.

Die Hauptschwierigkeit bei der Entwicklung des Control-Board-Prototyps bestand darin, dass bedingt durch die herausgelegte Peripherie und des nicht vorhandenen Bootloaders auf den Motorcontrollern, eine Programmierung durch den ESP32 ermöglicht werden musste.

Für das Programmieren der Motorcontroller wurde die Schaltung sowie die benötigte Firmware geschrieben, sodass die Fernwartung des Roboters mittels FOTA ermöglicht wurde.

Durch die Wahl des neuen Motorcontrollers konnte eine effiziente und präzise Motorsteuerung erdacht und umgesetzt werden. Im Zuge dessen wurden mehrere Regelreise überarbeitet, erweitert oder neu hinzugefügt um, ein präzises Fahren, ohne Verzögerung zu ermöglichen.

Um Fernwartung zu ermöglichen wurde die Firmware so erstellt, dass sie eine kabellose Programmierung des ESP32 und der beiden Motorcontroller möglich ist. Dies ist besonders in einem größeren Schwarm von Vorteil.

Die Multi-Roboter-Steuerung selbst ließ sich auf dem verbesserten PIA6 gut umsetzen, nachdem das Protokoll der XBees bekannt war. Komplexer war jedoch die Umsetzung der verschiedenen Klassen zur Steuerung mehrerer Roboter. Durch die modulare Umsetzung der Ansteuerung des Schwarms konnte das Testprogramm leicht um weitere Roboter erweitert werden. Maximal konnten bis zu vier Roboter zeitgleich eine Formationsfahrt durchführen. Hierbei stellte die Anzahl der fertiggestellten Roboter das Limit dar.

Zusammenfassend zeigen die Ergebnisse dieser Arbeit, dass die Roboter individuell präziser fahren. Weiterhin zeigt das Anwendungsbeispiel zur Multi-Robot-Steuerung, dass mit dem aktuellen Hardware-, Software- und Firmwarestand solche und ähnliche Schwarmanwendungen ermöglicht werden.

6.2 Ausblick

Der Roboter PIA6 und die umgesetzten Protokolle lassen es zu mit mehreren Robotern zeitgleich zu fahren. So ist es nun möglich neue Aufgaben für den Roboterschwarm zu erdenken und diese auch umzusetzen. Zu untersuchen gilt es, wodurch bei dieser Art der Umsetzung die maximale Anzahl der Roboter limitiert wird. Ein Ansatzpunkt ist beispielsweise die durch das WLAN bereitgestellte Bandbreite.

In einem nächsten Schritt kann die ROS-Integration überarbeitet werden, um auf die verschiedenen Werkzeuge von ROS Zugriff zu haben. So kann z.B. die vom ESP32 bereitgestellte, recht genaue Odometrie im Zusammenhang mit einem SLAM-Algorithmus verwendet werden.

Bezüglich der Motorcontroller Firmware gilt es zu untersuchen, ob ein gleitender Mittelwert zur Bestimmung der aktuellen Drehzahl eine valide Alternative darstellt, um Drehzahlschwankungen auf Grund der hohen Abtastrate zu kompensieren.

Da auf dem ESP32 ein Webserver läuft, welcher zurzeit lediglich zum Programmieren der Motorcontroller verwendet wird, wäre z.B. die Umsetzung eines Webinterfaces interessant, welches mittels Websockets alle Sensordaten visualisiert und zum Download zur Verfügung stellt.

Alles in allem bietet der neue Roboter nun die solide Grundlage für weitere Forschung in den verschiedensten Bereichen.

7 Literaturverzeichnis

[1] Ilon und . Erland, „WHEELS FOR A COURSE STA LE SELFPROPELLI VEHICLE MOVABLE IN ANY DESIRED DIRECTION ON THE GROUND OR SO E O THER ASE“. USA Patent US 876255, 8 4 975.

[2] H. Taheri, . Qiao und . ha eminezhad, „Kinematic ode l of a Four Mecanum Wheeled obi le Robot,“ *International Journal of Computer Applications (0975-8887),* pp. 8, eq. 20, 03 03 2015.

[3] H. Lutz und W. Wendt, Taschenbuch der Regelungstechnik, Haan-Gruiten: Harri Deutsch, 2005.

[4] L. Tomov und E. Garipov, „Choice of Sample Time in Digital PID Controllers,“ 7 2 7.

[5] W. Prof. Dr.-Ing Weber, Industrieroboter, München: Carl Hanser Verlag, 2009.

[6] S. Garrido-Jurado, R. Muñoz-Salinas, F. Madrid-Cuevas und M. Marín-Jiménez, „Automatic generation and detection of highly reliable fiducial markers under,“ *Pattern Recognition,* Nr. 47, pp. 2208-2292, 2006.

[7] . orges, A. Symington, . Coltin, T. Smith und R. Ventura, „HTC Vive: Analysis and Accuracy Improvement,“ in *International Conference on Intelligent Robots and Systems (IROS)*, Madrid, 2018.

[8] A. Peer, P. Ullich und P. Kevin, „Vive Tracking Alignment and Correction a de Easy,“ in *IEEE Conference on Virtual Reality and 3D User Interfaces (VR)*, Reutlingen, 2018.

[9] H. Hasberg, „Systemdesign: HOOU Robot (PIA) - Config ,“ HOOU-Intranet, 07.05.2019.

[10] S. Dipl.-Ing. (FH) Luber und A. Dipl.-Ing. (FH) Donner, „Was ist das OSI- ode ll?,“ 08 2018. [Online]. Available: https://www.ip-insider.de/was-ist-das-osi-modell-a-605831/. [Zugriff am 29 07 2020].

[11] Pololu, „Pololu aby Orangutan User’s uide ,“ 9 5 2 8. [Online]. Available: https://www.pololu.com/docs/pdf/0J14/baby_orangutan_b.pdf. [Zugriff am 29 07 2020].

[12] Atmel, „ATmega 28 P Datasheet,“ 2 5. [Online]. Available: http://ww1.microchip.com/downloads/en/DeviceDoc/Atmel-7810-Automotive-Microcontrollers-ATmega328P_Datasheet.pdf. [Zugriff am 29 07 2020].

[13] TOSHI A, „T 66 2 F Datasheet,“ 6 2 7. [Online]. Available: https://www.sparkfun.com/datasheets/Robotics/TB6612FNG.pdf. [Zugriff am 29 07 2020].

[14] . W , „5V to . V logic level translation: how to interface a 5V output to a . V input,“ 15 09 2017. [Online]. Available: https://next-hack.com/index.php/2017/09/15/how-to-interface-a-5v-output-to-a-3-3v-input/. [Zugriff am 28 07 2020].

[15] S. ergmans, „Intel HEX format,“ 2 6 2 9. [Online]. Available: https://www.sbprojects.net/knowledge/fileformats/intelhex.php. [Zugriff am 29 07 2020].

[16] T. Horn, „ASCII-, DOS-Latin-1-, Windows-1252- und HTML-Zeichencodes,“ 998. [Online]. Available: https://www.torsten-horn.de/techdocs/ascii.htm. [Zugriff am 29 07 2020].

[17] C. r ieger, „ESP 2,“ 2 9. [Online]. Available: https://elektro.turanis.de/html/prj135/index.html. [Zugriff am 29 07 2020].

[18] Falk, „Drehgeber,“ 25 7 2 9. [Online]. Available: https://www.mikrocontroller.net/articles/Drehgeber. [Zugriff am 29 07 2020].

[19] F. Weithoff, „Systemdiagram PIA6 incl. Ladeinfrastruktur,“ HOOU-Intranet, 03.08.2020.

[20] H. Hasberg, „PIA Python Package,“ HOOU-Intranet, 06.07.2020.

[21] T. Röbke, „PIA Powe rCtl oard,“ HOOU-Intranet, 04.03.2020.

A Anhang

A.1 Digitaler Anhang

a sisverzeichnis der CD

- Arduino
  - [...]
  - digitalRead (Testfirmware: e stimmung der Einlesedauer)
  - HOOU a by Orangutan (Closed Loop otorcontroller Firmware)
  - HOOU ESP 2 (PIA6 Firmware)
  - libraries
    - [...]
    - HOOU a by Orangutan (i bliothek zur Ansteuerung des otorcontrollers)
    - OS CommonWifi (WIFI setup und OTA funktionen des ESP 2)
    - OS FS AVRISP (FOTA i bliothek zum ISP Programmieren der otorcontroller)
- 2 Python
  - [...]
  - HIVE Controll
    - [...]
    - ArucoPositionPublisher.py (Sende Script des AR Fehlers)
    - V .2.6 frontCam small marker.py (Andock Script zum laden am HIVE)
  - Package
    - HOOU Robot
      - [...]
      - Logger.py (VIVE Tracker werden hier mit einer IP versehen)
      - ControlInput ultiKeyboard.py
      - SystemInterface UDP.py
  - Scripts
    - [...]
    - nc
      - [...] (CODE dateien zum fahren beliebiger Tra ektorien)
    - . py (Testscripte f r P IA 4 und PIA6)
- essdaten
  - [...] (Alle essdaten nach Abbildung sortiert)
- Thesis.pdf