

Initialsensorenbasierte Positionsregelung eines Indoor-Quadroopters

T3100

für die Prüfung zum
Bachelor of Science

im Studiengang Informatik
an der DHBW Karlsruhe

von

Michael Maag

17.05.2022

Bearbeitungszeitraum

6 Monate

Matrikelnummer

6170558

Gutachter der DHBW Karlsruhe

Prof. Dr. Marcus Strand

Michael Maag
Freidorfstraße 14
97957 Wittighausen

Eigenständigkeitserklärung

Ich versichere hiermit, dass ich meine Ausarbeitung T3100 mit dem Thema “Initial-sensorenbasierte Positionsregelung eines Indoor-Quadrokopters“ selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Ich versichere zudem, dass die eingereichte elektronische Fassung mit der gedruckten Fassung übereinstimmt.

Wittighausen, 17.05.2022

Unterschrift

Inhaltsverzeichnis

Abbildungsverzeichnis	I
Code-Verzeichnis	III
1 Einleitung	1
2 Problemstellung	2
3 Methodik	3
3.1 Begriffe	3
3.2 Vorgehen	3
3.3 Eingesetzte Software	3
3.3.1 ROS	3
3.3.2 Ubuntu - Betriebssystem	4
3.3.3 Visual Studio - IDE	4
4 Regelsysteme	5
4.1 Regelkreis	5
4.2 Arten von Reglergliedern	6
4.2.1 P-Glied	6
4.2.2 I-Glied	6
4.2.3 D-Glied	7
4.2.4 PID-Regler	8
4.2.5 PT-Glied	8
4.2.6 Tt-Glied	9
4.3 Stabilität	9
5 Positionsregelung von Quadrokoptern	10
5.1 Quadrokopter als System	10
5.1.1 Orientierungsmerkmale	10
5.1.2 Geometrien	10
5.1.3 Freiheitsgrade	12
5.1.4 Pose	12
5.2 Positionsregelung von Quadrokoptern mittels Pose	13
5.3 Erzeugung von Posen aus Beschleunigungsdaten	14
5.3.1 Berechnung	14
5.3.2 Signalaufarbeitung	14

6	Eingesetzte Hardware.....	21
6.1	COEX Drohne.....	21
6.1.1	Control Stack	21
6.1.2	Sensorik	21
6.1.3	Aufbau des Bausatzes.....	22
6.1.4	Inbetriebnahme.....	22
6.1.5	Mögliche Lösung der Aufgabenstellung.....	23
6.1.6	Troubleshooting.....	24
6.2	Parrot Drohne	25
6.2.1	Sensorik	26
6.2.2	Interaktion mittels <i>ROS</i>	26
6.2.3	Inbetriebnahme auf separatem Rechner.....	27
6.2.4	Troubleshooting.....	27
7	Implementierung.....	28
7.1	Software-Architektur	28
7.1.1	Package-Konzept	28
7.1.2	<i>Domain Package</i>	29
7.1.3	<i>DroneController Package</i>	29
7.1.4	<i>Adapter Package</i>	29
7.1.5	<i>Controller Package</i>	29
7.1.6	<i>parrot Package</i>	29
7.1.7	<i>PosControl Package</i>	29
7.2	Abhängigkeitsgraph	30
7.3	Domain Layer.....	30
7.4	Application Layer - <i>DroneController Package</i>	32
7.5	Adapter Layer	35
7.6	PlugIn Layer	35
7.6.1	<i>parrot Package</i>	35
7.6.2	<i>Controller Package</i>	36
7.6.3	<i>PosControl Package</i>	37
7.6.4	<i>calling Package</i>	37
7.6.5	<i>threading Package</i>	37
7.6.6	<i>coex Package</i>	39
8	Ergebnis	42
8.1	Analyse realer Flugdaten	42
8.1.1	Datenbasis	42
8.1.2	Verlauf des Fluges.....	44
8.1.3	Lecks der Datenübertragung	45

8.2	Signalverarbeitung	45
8.2.1	Ermittlung der dauerhaften Nullpunkt-Abweichung.....	45
8.2.2	Glättung der Eingangsdaten	45
8.2.3	Kalibrierung der Posenberechnung.....	45
8.3	Auswertung.....	47
9	Fazit und Ausblick.....	49
9.1	Erweiterungen	49
9.1.1	Weitere Ansätze der Signalverarbeitung / Pose-Berechnung	49
9.1.2	Interne Sensoren	49
9.1.3	Externe Sensoren	50
9.1.4	Externe Sensoren und Mapping.....	50
Literaturverzeichnis		

Abbildungsverzeichnis

1	Allgemeines Schema eines Regelkreises nach IEC60050-351 [10]	5
2	Sprungantwort eines P-Glieds	6
3	Sprungantwort eines I-Glieds	7
4	Sprungantwort eines D-Glieds	8
5	Sprungantwort eines PID-Glieds	8
6	Sprungantwort eines PT-Glieds	9
7	Orientierungsmerkmale eines Flugobjekts [28]	10
8	wirkende Kräfte am Quadrocopter in x -Anordnung (vgl. [2])	11
9	Geometrien von Quadrocoptern [1]	11
10	wirkende Kräfte für gerollten Quadrocopter (vgl. [24])	13
11	Originalkurve des Szenarios	16
12	Einfluss von Signalrauschen	17
13	Einfluss von Signalrauschen	18
14	Einfluss von Ausreißern	19
15	Einfluss von konstanter Nullpunkt-Abweichung	20
16	Aufbau des Bausatzes für die Drohne <i>Clover 4.20</i>	22
17	rqt-Graph der <i>ArDrone 2.0</i>	26
18	Architektur des Regelungssystems	28
19	Klassendiagramm des Programms	30
20	Klassendiagramm des <i>calling Packages</i>	38
21	Klassendiagramm des <i>coex Packages</i>	40
22	Testflug: StatusID, Flags und Höhenprofil	43
23	Testflug Signalverarbeitung: Aufarbeitung a_z	46
24	Testflug Signalverarbeitung: Kalibrierung	47
25	Testflug Signalverarbeitung: Aufarbeitung a_z	48

Formelverzeichnis

1	Übertragungsfunktion des P-Glieds	6
2	Übertragungsfunktion des I-Glieds	7
3	Übertragungsfunktion des D-Glieds	7
4	Übertragungsfunktion des PID-Glieds	8
5	Übertragungsfunktion des PT-Glieds	9
6	mathematische Grundlage der <i>PoseBuildable</i> -Kalibrierung	46

Code-Verzeichnis

1	Befehl zum Öffnen des <i>OverrideRCIn</i> -Headers	25
2	Definition des Struct MD5Sum für das Template <i>OverrideRCIn</i>	25

1 Einleitung

“Über den Wolken muss die Freiheit wohl grenzenlos sein.“

aus dem Lied *Über den Wolken* von Reinhard Mey, 1973 [14]

Schon lange sehnten sich Menschen danach, fliegen zu können. Eine der bekanntesten Geschichten ist die Mythologie über Ikarus, der auf der Flucht aus dem Labyrinth auf der Insel Kreta mit selbstgebauten Flügeln der Sonne zu nahe kam und daraufhin ins Meer stürzte. (vgl. [15])

Leonardo da Vinci deutete Ende des 15. Jahrhunderts in den *Pariser Manuskripten* die Funktionsweise eines Helikopters beziehungsweise einer Luftschaube an. (vgl. [9])

Diverse Paket-Lieferdienste planen, Quadrocopter in Ballungsgebieten zum Versand von Paketen einzusetzen. (vgl. [16])

Zur Wegplanung ist eine korrekt berechnete Pose (Position und Orientierung) des Quadrocopters notwendig. Hierzu können diverse Sensoren eingesetzt werden, welche an den Fügeräten eingesetzt werden.

In dieser Projektarbeit wurden Initialsensoren als mögliche Herangehensweise ausgewählt. Als Grundlage für die Auswahl kann die einfache Aufarbeitung der Daten genannt werden. Zudem wurden bereits erfolgreich Projekte mit diesem Konzept umgesetzt. (vgl. [3] und [4])

Anmerkung: Für diese Ausarbeitung werden fachliche Begrifflichkeiten vorausgesetzt, sofern diese nicht innerhalb der Ausarbeitung erklärt werden. Sind Begriffe für Lesende unklar, sind diese an geeigneter Stelle nachzuschlagen. Auf eine voranstehende Erklärung aller genutzten und nicht näher erklärten Begriffe wird in dieser Ausarbeitung verzichtet, um den Rahmen dieser Arbeit einhalten zu können.

2 Problemstellung

Fluggeräte jeder Ausführung können durch Umwelteinflüsse von ihrer Position abgetrieben werden (vgl. [17]). Während der Versuchsdurchführung von Studierenden der DHBW Karlsruhe an einem Quadrokopter hat sich gezeigt, dass sich das Halten einer Position für Piloten mit geringer Erfahrung als schwierig erweist. Beschädigungen der in Laborversuchen eingesetzten Hardware ist zu vermeiden. Die Versuchsdurchführung *Höhenregelung*¹ soll für die Studierenden dahingehend vereinfacht werden, alsdass der eingesetzte Quadrokopter die horizontale Bewegung selbstständig regelt.

Zu entwickeln ist eine Positionsregelung auf Basis der verfügbaren Beschleunigungswerte der Drohne. Optional kann die Regelung um ein bildgestütztes System erweitert werden.

Für die Positionsregelung können unterschiedliche Modi entwickelt werden:

- Halten der Position nach einer manuellen Positionsänderung
- Anfliegen von vorgegebenen Positionen. Hierbei ist ein Überschwingen möglichst zu vermeiden.

Durch die Anschaffung eines neuen Quadroopters erweitert sich die Aufgabenstellung um den Aufbau des Bausatzes und die Inbetriebnahme des Quadroopters, an dem die Positionsregelung implementiert werden soll. Die Anpassung der Versuchsbeschreibung an die geänderte Hardware ist gewünscht.

¹Bei dem Laborversuch *Höhenregelung* sollen Studierende eine ROS-Node erstellen, welche eine konstante Flughöhe des Quadroopters ermöglicht. Hierzu wird als Rückführungsgröße des Regelkreises die Abstandsmessung zwischen Quadroopter und der darunterliegenden Ebene eingesetzt.

3 Methodik

3.1 Begriffe

In diesem Kapitel sollen Begrifflichkeiten für diese Projektarbeit definiert werden, welche im allgemeinen Sprachgebrauch mehrdeutig belegt sind.

Drohne	Quadrokopter
Controller	Regelkreis zum regeln der Pose

3.2 Vorgehen

Die Problemstellung dieser Projektarbeit (siehe Kapitel 2 *Problemstellung* (Seite 2)) sieht vor, eine Positions- beziehungsweise Posenregelung für einen Quadrokopter zu implementieren.

Hierzu wurde das System Quadrokopter analysiert und aus den von der Hardware übersendeten Nachrichten, diejenigen Informationen ausgewählt, welche zur Lösung der Problemstellung beitragen. Darüber hinau wird ermittelt, welche Intraktionen mit der Hardware jeweils nötig sind, um eine Regelung aufzubauen zu können.

Die Ermittlung einer Position aus Daten von Initialsensoren entspricht einer doppelten Integration (vgl. Kapitel 5.3.1 *Berechnung* (Seite 14)). In Kapitel 5.3.2 *Signalaufarbeitung* (Seite 14) zeigt sich der Einfluss von Signalfehlern auf diese Berechnung.

Aus den gesammelten Erkenntnissen wurde anschließend eine Architektur und daraus eine Implementierung entwickelt, welche den Anforderungen dieser Projektarbeit grundsätzlich genügt.

Zum Abschluss dieser Dokumentation wird das erstellte Programm auf reale Daten angewandt (siehe Kapitel 8 *Ergebnis* (Seite 42)).

3.3 Eingesetzte Software

In diesem Kapitel sollen die für diese Projektarbeit eingesetzten Softwares genannt, um eine Reproduktion der Ergebnisse gewährleisten zu können.

Anmerkung: Die Ausführungen der eingesetzten Software beziehen sich auf den Umgang mit der Drohne *ArDrone 2.0*. Für die Interaktion mit der Drohne *Clover 4.20*, welche zum Projektbeginn eingesetzt wurde, wurde aktuellere Software eingesetzt.

3.3.1 ROS

Das *Robot Operating System (ROS)* ist eine *Open Source* Bibliothek, welche dem Nutzer eine modulare Architektur ermöglicht. Hierbei kommunizieren *Nodes* mittels *Messages*

miteinander.(vgl. [22])

Die *ROS* Versionen werden jeweils mit Namen versehen, wobei die Anfangsbuchstaben der Versionen der alphabetischen Nummerierung entsprechen. In diesem Projekt wurde die *ROS*-Version *Indigo* eingesetzt. Diese Version wird auf Grund der Anforderungen des *Driver-Package* *ardrone_autonomy* eingesetzt. Der Einsatz der aktuellsten *ROS*-Version *noetic* in Zusammenspiel mit *Ubuntu 20.04* konnte das gewünschte Ergebnis nicht erzielen.

Die *ROS-Nodes* wurden mittels *catkin* kompiliert. Für eine korrekte Kompilierung müssen *CMakeList.txt*-Dateien den Befehl `add_compile_options(-std=c++11)` beinhalten.

3.3.2 Ubuntu - Betriebssystem

Als Betriebssystem wird das *UNIX*-basierte Betriebssystem *Ubuntu* auf einer *virtuelle Maschine* in der Version 14.04 eingesetzt. Die Auswahl dieser Version gründet auf den Anforderungen der *ROS Indigo*-Version.(vgl. [23])

Die *virtuelle Maschine* wird durch die Software *VMWare Workstation 15 Pro* virtualisiert.

3.3.3 Visual Studio - IDE

Aus der Präferenz des Autors heraus wurde der Code mittels *Visual Studio 2019 (Community Edition)* erstellt, getestet und anschließend in den *catkin workspace* migriert.

4 Regelsysteme

In der Norm *DIN IEC 60050-351 (Internationales Elektrotechnisches Wörterbuch – Teil 351: Leittechnik)*, welche Begriffe der Regelungstechnik definiert, wird der Begriff *Regelung* wie folgt beschrieben: [10]

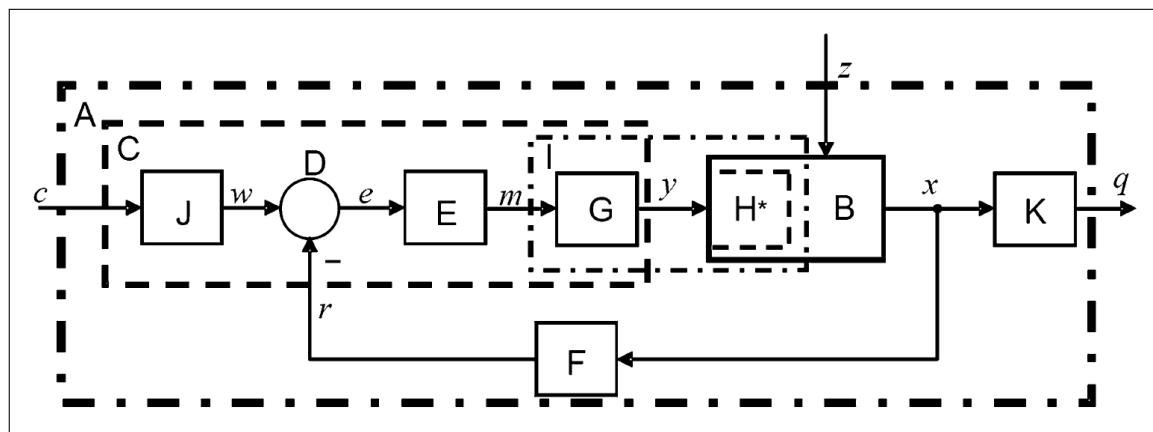
„Vorgang, bei dem fortlaufend eine variable Größe, die Regelgröße, erfasst, mit einer anderen variablen Größe, der Führungsgröße, verglichen und im Sinne einer Angleichung an die Führungsgröße beeinflusst wird.“

Anmerkung: Kennzeichen für das Regeln ist der geschlossene Wirkungsablauf, bei dem die Regelgröße im Wirkungsweg des Regelkreises fortlaufend sich selbst beeinflusst.“

In diesem Kapitel sollen regelungstechnische Grundlagen beschrieben werden.

4.1 Regelkreis

Im Allgemeinen grenzen sich die Konzepte *steuern* und *regeln* durch die Eigenschaft der Rückkopplung des Systems ab. Gesteuerte Systeme wirken lediglich auf Aktoren, wobei geregelte Systeme die Abweichung des *Ist*-Zustandes vom *Soll*-Zustand ermitteln und eine geeignete Veränderung erzielen sollen.(vgl. [10])



A	Regelungssystem	K	Bildung der Aufgabengröße
B	Regelstrecke	c	Zielgröße
C	Regeleinrichtung	w	Führungsgröße
D	Vergleichsglied	e	Regeldifferenz
E	Regelglied	m	Reglerausgangsgröße
F	Messglied	y	Stellgröße
G	Steller	z	Störgröße
H*	Stellglied	x	Regelgröße
I	Stelleinrichtung	q	Aufgabengröße
J	Führungsgrößenbildner	r	Rückführgröße

Abbildung 1: ALLGEIMEINES SCHEMA EINES REGELKREISES NACH ORN IEC60050-351 [10]

4.2 Arten von Reglergliedern

In diesem Kapitel sollen grundlegende Bausteine der Regelungstechnik beschrieben werden. Diesbezüglich erhebt dieses Kapitel keinen Anspruch auf Vollständigkeit. Auf eine detaillierte Beschreibung, welche unter anderem das Zeitverhalten betrachten, soll hier außen vor gelassen werden. Als weitere Einschränkung soll sich die Betrachtung der Bausteine ausschließlich auf zeitdiskrete Systeme beziehen.

4.2.1 P-Glied

Die Abkürzung P in der Bezeichnung *P-Glied* steht für *proportional*. Hierbei wird eine die Eingangsgröße um einen Faktor k_P verstärkt. (vgl. [6])

$$y_k = k_P * x_k \quad (1)$$

ÜBERTRAGUNGSFUNKTION DES P-GLIEDS

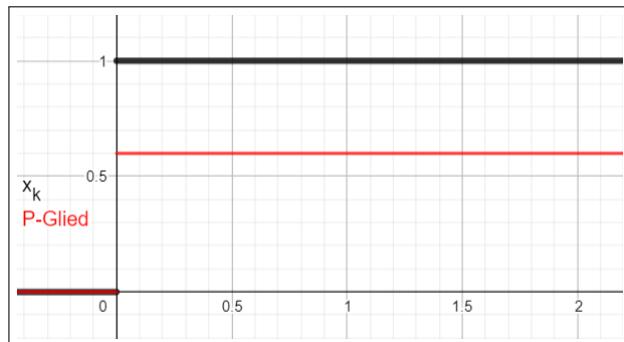


Abbildung 2: SPRUNGANTWORT EINES P-GLIEDS

Abbildung 5 zeigt charakteristischen Sprungantwort des *P-Glieds*. Hier wurde der Parameter k_P mit dem Wert 0,6 gewählt, um die Sichtbarkeit in der Abbildung zu erhöhen.

In einem geschlossenen Regelkreis führt der Einsatz eines *P-Glieds* ohne weitere Regelbausteine zu einer bleibenden Regelabweichung. Aus den Zusammenhängen $e = w - r$ und $r = e * k_P^2$ lässt sich die dauerhafte Regelabweichung in Abhängigkeit des Verstärkungsfaktors bestimmen: $e = \frac{w}{1-k_P}$.

Entgegen steht der Vorteil eine schnellen und dauerhaften Reaktion auf eine Regeldifferenz.

4.2.2 I-Glied

Wie sich aus dem Namen des *Integral-Glieds* ableiten lässt, bildet dieser Baustein ein Integral über dem Eingangssignal. Hierdurch können physikalische Umrechnungen oder

²Für diesen Ansatz werden sämtliche Bausteine nach E (*Regelglied*) (siehe Abbildung 1 (Seite 5)) ignoriert. Daraus folgt: $r = m$.

Prozesse abgebildet werden. Als Beispiele kann an dieser Stelle der Füllstand eines Tanks oder die Ermittlung einer Geschwindigkeit aus Beschleunigungsdaten genannt werden. (vgl. [6])

$$y_k = y_{k-1} + k_I * x_k * \Delta t \quad (2)$$

ÜBERTRAGUNGSFUNKTION DES I-GLIEDS

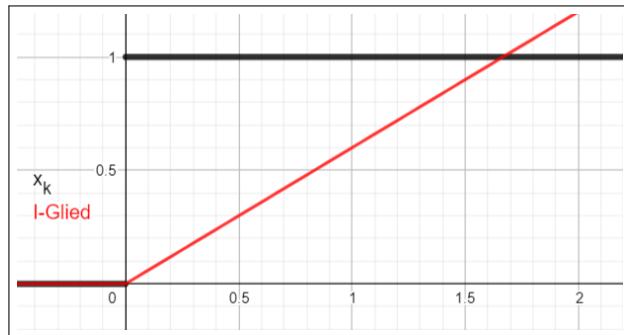


Abbildung 3: SPRUNGANTWORT EINES I-GLIEDS

Da das *I-Glied* Regeldifferenzen aufsummiert, läuft die Regeldifferenz im geschlossenen Regelkreis asymptotisch gegen den Wert 0.

4.2.3 D-Glied

In der betrachteten Literatur spielen D-Glieder keine signifikante Rolle. (vgl. [6] und [7]) Dennoch sollen sie hier als Grundlage für *PID*-Glieder beschrieben werden.

Bei einem *D-Glied* handelt es sich um ein differentielles Glied. Somit lässt sich hiermit die Veränderung des Eingangssignals ermitteln. (vgl. [18])

Das Ausgangssignal des *D-Glieds* kann durch folgende Formel berechnet werden, welche sich aus der Diskretisierung der Differenzationsfunktion ergibt.

$$y_k = k_D * \frac{x_k - x_{k-1}}{\Delta t} \quad (3)$$

ÜBERTRAGUNGSFUNKTION DES D-GLIEDS

Durch den Einfluss eines *D-Glieds* kann auf schnelle Änderungen einer Größe reagiert werden. Die Regeldifferenz der Sprungantwort ist nach dem anfänglichen Ausschlag mit dem Wert 1 (100%) anzunehmen.

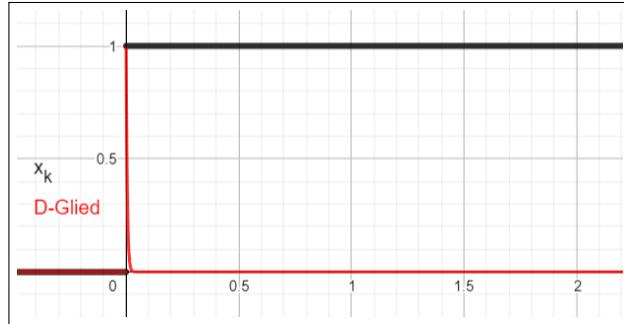


Abbildung 4: SPRUNGANTWORT EINES D-GLIEDS

Abbildung 4 zeigt die Sprungantwort eines *D-Glieds*. Diese ist ein Ausschlag, welcher für den Zeitschritt anhält, in dem sich die ansteigende Flanke des Sprungs befindet.

4.2.4 PID-Regler

Das *PID-Glied* bildet die Vereinigung der vorab genannten Regelbausteine. Als Ausgangssignal wird die Summe über die einzelnen Regelbausteine gebildet:

$$y_k = y_{k(P)} + y_{k(I)} + y_{k(D)} \quad (4)$$

ÜBERTRAGUNGSFUNKTION DES PID-GLIEDS

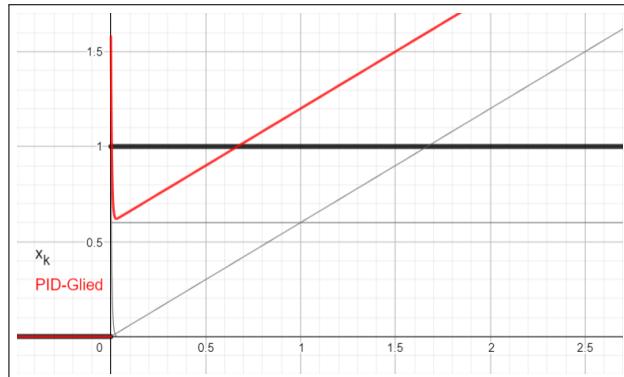


Abbildung 5: SPRUNGANTWORT EINES PID-GLIEDS

Abbildung 5 zeigt die Sprungantwort eines *PID-Glieds*. Hierbei setzt sich das Signal als Summe des P-, des I- und des D-Glieds zusammen, welche in grau angedeutet sind.

Das *PID-Glied* vereinigt die Vorteile der Einzelbausteine. Die genannten Nachteile werden durch eines der anderen Regelglieder ausgehebelt.

Daher wird das *PID-Glied* als universal-Regelbaustein eingesetzt, wobei geeignete Verstärkungsfaktoren ausgewählt und optimiert werden müssen.

4.2.5 PT-Glied

Um reale Systeme abzubilden, kann das *PT-Glied* eingesetzt werden. Als Beispiele können hier die Temperatur in einem Gas ein eingebrachter Energie oder die Drehzahl

eines Motors (massebehaftet) genannt werden.

Die Einbindung des *PT-Glieds* erfolgt die Vor- beziehungsweise Nachschaltung an Berechnung der idealisierten Größe.

$$y_k = y_{k-1} + (k_{PT1} * x_k - y_{k-1}) * \frac{\Delta t}{T_1} \quad (5)$$

ÜBERTRAGUNGSFUNKTION DES PT-GLIEDS

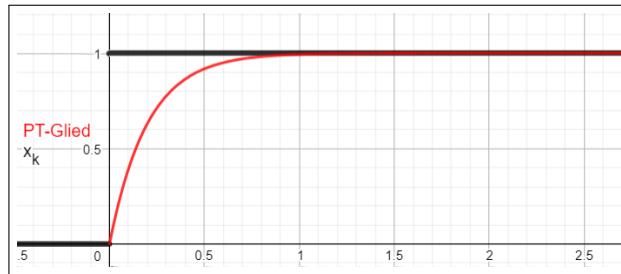


Abbildung 6: SPRUNGANTWORT EINES PT-GLIEDS

4.2.6 Tt-Glied

Bei einem *Totzeit-Glied* handelt es sich um Baustein, welcher vorwiegend zur Abbildung realer Systeme genutzt wird. Als Beispiel kann hier die Füllmenge eines Förderbands genannt werden, wobei der Füllstandssensor nach der Befüllleinrichtung angeordnet werden muss.

Totzeiten erhöhen die Instabilität von Systemen. (vgl. [20])

Da dieser Baustein nur zur Vollständigkeit genannt werden soll, wird von einer Vertiefung der Thematik an dieser Stelle abgesehen.

4.3 Stabilität

Die Stabilität eines Regelkreises beschreibt, ob der Regelkreis zum Schwingen neigt. Hierfür wird die Übertragungsfunktion des offenen Regelkreises analysiert. Das Schwingen eines zu regelnden Systems ist zu vermeiden. (vgl. [6] und [19])

An dieser Stelle soll auf eine Beschreibung der Analyse der Stabilität von Regelkreisen verzichtet werden, um den Rahmen dieser Projektarbeit einhalten zu können. Als Schlagworte für interessierte Lesende sei hier das *Bode-Diagramm* und die *Ortskurve* genannt (vgl. [6] und [19]).

5 Positionsregelung von Quadrooptern

5.1 Quadroopter als System

5.1.1 Orientierungsmerkmale

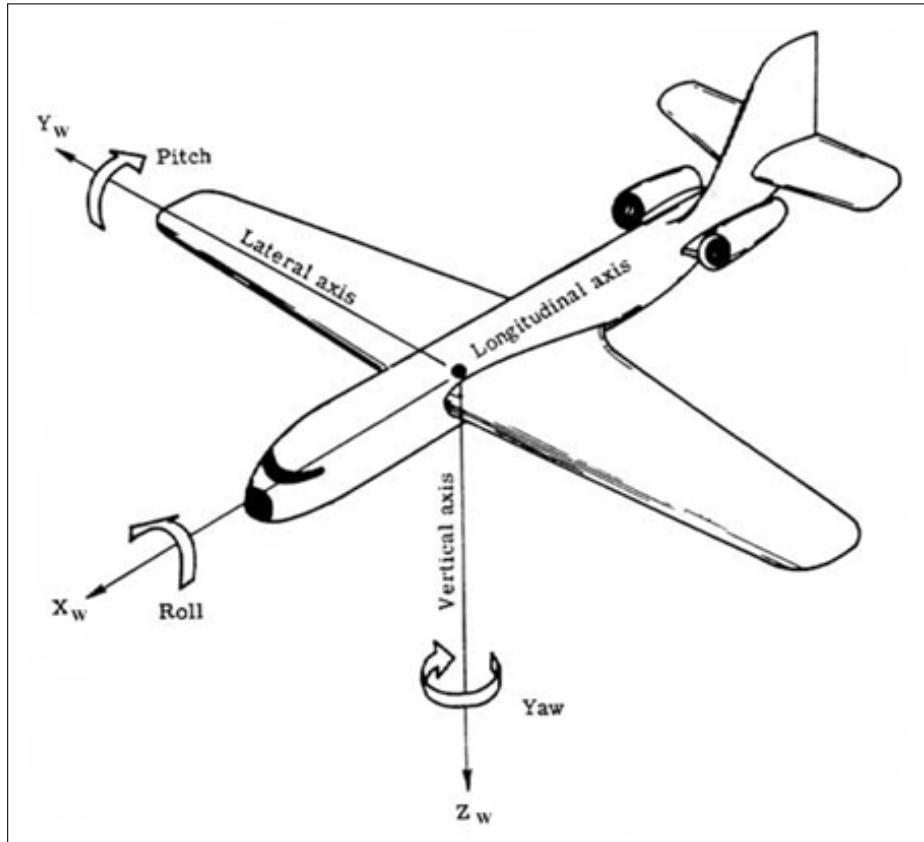


Abbildung 7: ORIENTIERUNGSMERKMALE EINES FLUGOBJEKTS [28]

Abbildung 7 zeigt die Orientierung eines beliebigen Flugobjektes am Beispiel eines Flugzeugs. Die Bezeichner beziehen sich auf ein kartesisches Koordinatensystem und beschreiben jeweils die Rotation um eine Raumachse.

5.1.2 Geometrien

Bei Quadrooptern handelt es sich um Fluggeräte mit vier Rotoren, welche horizontal angebracht sind. Der Auftrieb wird somit unmittelbar durch die Rotoren induziert.

Das Drehmoment der Rotoren des Quadroopters muss sich aufheben können, um eine unkontrollierbare Rotation um die z-Achse vermeiden zu können. Um die Agilität des Quadroopters beizubehalten zu können, sind die Drehrichtungen der Rotoren zu alternieren. (vgl. [1]) Abbildung 8 (Seite 11) deutet die Kräfte und aus der Rotationsgeschwindigkeit der Rotoren ableitbare Momente eines Quadroopters an. Dies soll als Einleitung für das Kapitel 5.1.3 *Freiheitsgrade* (Seite 12) dienen und wird an dieser Stelle nicht näher vertieft.

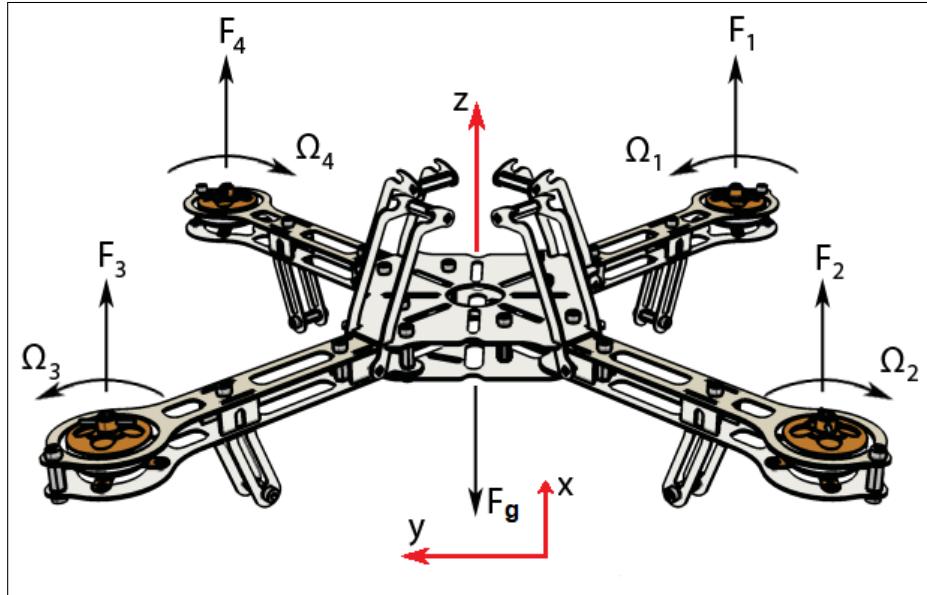


Abbildung 8: WIRKENDE KRÄFTE AM QUADROKOPTER IN x-ANORDNUNG (VGL. [2])

Abbildung 8 zeigt die wirkenden Kräfte an einem Quadrokopter in **x**-Anordnung. Die Achsen des Koordinatensystems sind in **rot** markiert. Alle mit *F* markierten Pfeile deuten Kräfte an. Die mit Ω markierten gebogenen Pfeile zeigen die Rotationsgeschwindigkeit der einzelnen Rotoren.

Die Rotoren von Quadrooptern können in zwei verschiedenen Anordnungen angebracht werden (siehe Abbildung 9 (Seite 11)).

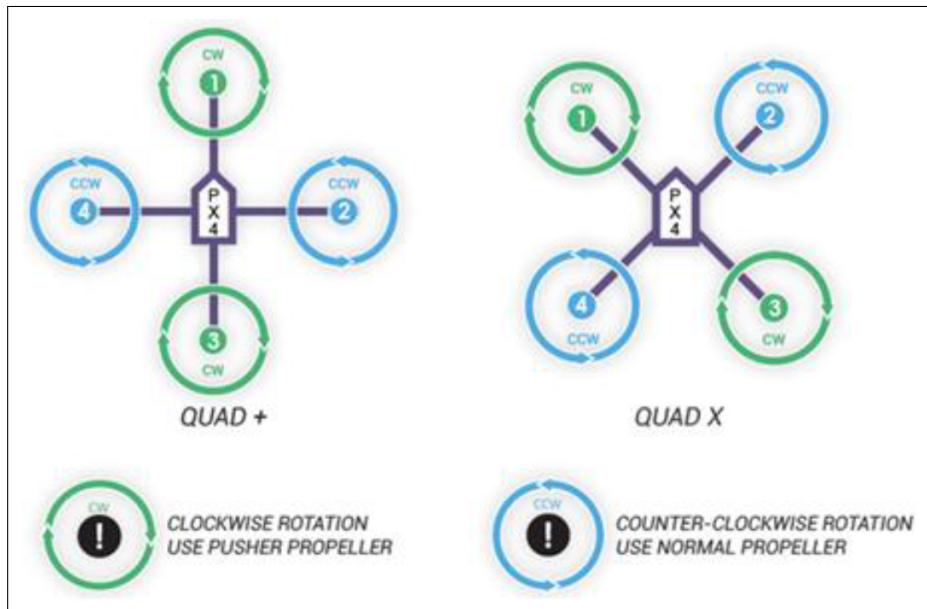


Abbildung 9: GEOMETRIEN VON QUADROKOPTERN [1]

+-Anordnung

In der **+-Anordnung** befinden sich die Rotoren auf den Objekt-Achsen des Quadroko-

pters. Die Nummerierung der Rotoren erfolgt gemäß Abbildung 9 (Seite 11) (links) im Uhrzeigersinn beginnend mit dem Rotor auf der positiven x-Achse.

x-Anordnung

Die **x**-Anordnung scheint nach online Recherchen weiter verbreitet. Eine mögliche Begründung findet sich in der weniger verdeckten Sicht für Frontkameras oder andere Anbaugeräte.

5.1.3 Freiheitsgrade

Einem Quadrokopter können sechs Freiheitsgrade zugewiesen werden, jeweils drei der Position und der Orientierung im Raum.

Nachfolgend wird physikalisch begründet, weshalb nur vier der genannten sechs Freiheitsgrade unabhängig regelbar sind.

Wird eine Kraft in der horizontalen Ebene induziert, beschleunigt der Quadrokopter entlang dieser Kraft. Eine solche Kraft wird durch eine Neigung um die x- beziehungsweise y-Achse hervorgerufen (siehe Abbildung 10 (Seite 13)). Aus den genannten Umständen lässt sich ableiten, dass sich eine geänderte Orientierung um die x- beziehungsweise y-Achse auf die Position des Quadroopters auswirkt.

5.1.4 Pose

Eine Pose ist die Positions- und Lagebeschreibung im Raum. Hierzu wird ein kartesisches Koordinatensystem zur Positionsbeschreibung genutzt. Die Lage beziehungsweise Orientierung wird durch die Rotation um Achsen beschrieben. Hierfür wurden zwei herangehensweisen definiert (siehe [12]):

- Euler-Winkel (Orientierung um verändertes Koordinatensystem)
- Roll-Pitch-Yaw (Orientierung zu den Achsen des Basiskoordinatensystems)

lokale Pose

Als lokale Pose wird diejenige Pose bezeichnet, welche sich auf den letzten definierten Nullpunkt (*Basiskoordinatensystem*) der Bewegungsabfolge bezieht.

Anmerkung: In dieser Projektarbeit wird der Nullpunkt während unmittelbar beim Start des Quadroopters gesetzt. somit verändert sich die lokale Pose regelmäßig.

globale Pose

Unter einer globalen Pose wird die Pose innerhalb eines Welt-Koordinatensystems verstanden. Hierbei kann eine lokale Pose in eine globale Pose transformiert werden, sofern

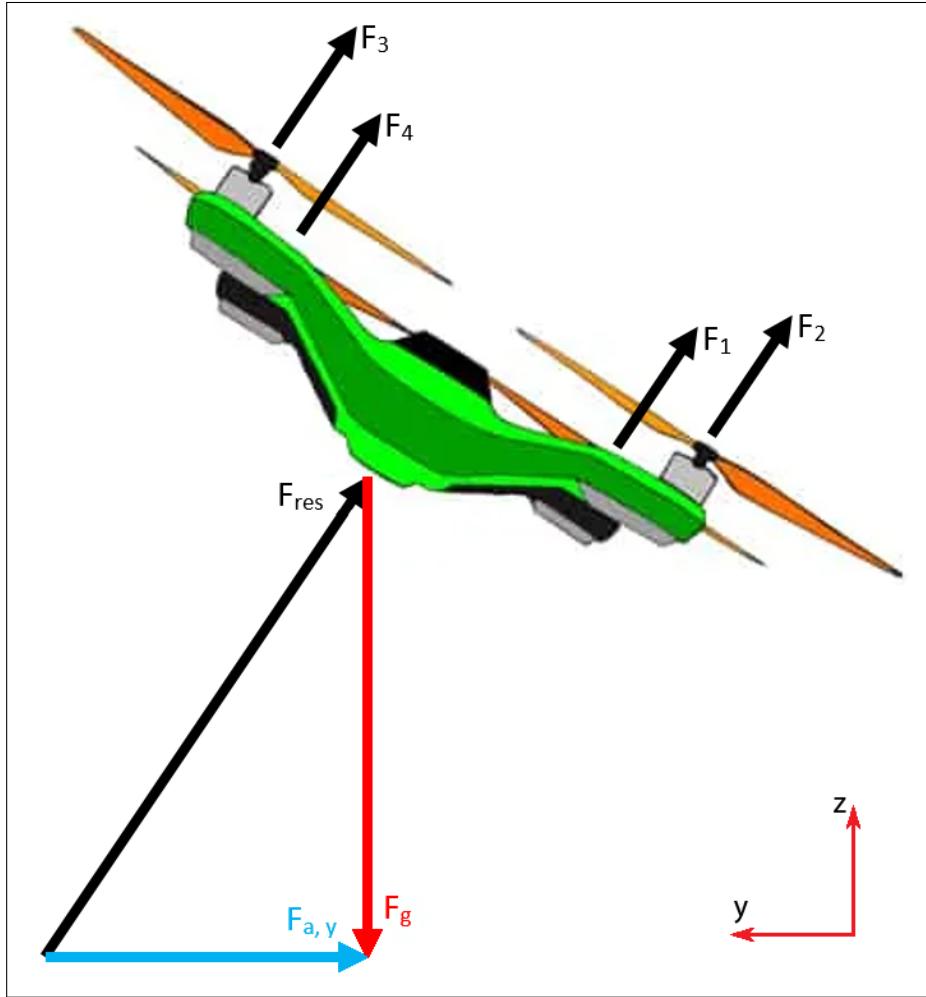


Abbildung 10: WIRKENDE KRÄFTE FÜR GEROLLTEN QUADROKOPTER (VGL. [24])

Abbildung 10 zeigt die auf einen Quadrocopter wirkenden Kräfte, wenn sich dieser in einer gerollten Orientierung befindet. Hier entspricht der Betrag der Kraft entlang der z-Achse der Gewichtskraft.

Die Kraft F_{res} wird mit der Gewichtskraft F_g überlagert. Die hieraus resultierende Kraft kann in drei Kräfte aufgeteilt werden, welche parallel zu den kartesischen Achsen angeordnet sind. Hieraus berechnet sich die Beschleunigung, welche auf den Quadrocopter wirkt.

der Nullpunkt der lokalen Pose innerhalb des Welt-Koordinatensystems bekannt ist. Hierfür wird externe Sensorik und ein geeigneter Algorithmus (zum Beispiel Markov-Localization oder Sequential Monte Carlo (vgl. [13])) benötigt.

5.2 Positionsregelung von Quadrooptern mittels Pose

Die Freiheitsgrade einer Drohne können grundsätzlich separat geregelt werden. Jedoch ist die Kraft in z-Richtung abhängig von der Ausrichtung (Roll und Pitch) des Quadroopters. Dies ist aus Abbildung 10 (Seite 13) ersichtlich. Der Quadrocopter wird als Folge einer Veränderung der Roll- und Pitch-Winkel an Höhe verlieren. Der Hö-

henregler wird diese Veränderung nachführen. Um ein Nachregeln des Quadroopters bei einer Orientierungsänderung in der Horizontalen abschwächen zu können, kann der veränderte Schubbedarf aus den gemessenen oder berechneten Roll- und Pitch-Winkel abgeleitet und zu dem Schubsignal aufsummiert werden.

Es ist zu beachten, dass die Parallelität des Quadroopter-Koordinatensystems zum lokalen beziehungsweise globalen Koordinatensystem bei einer Rotation um die z-Achse verloren geht. Hierzu ist die translativen Bewegung entgegengesetzt dieser Rotation zu transformieren. Somit kann die Position bezogen auf das lokale beziehungsweise globale Koordinatensystem ermittelt werden. (vgl. [11])

5.3 Erzeugung von Posen aus Beschleunigungsdaten

5.3.1 Berechnung

Für die nachfolgend beschriebene Berechnung der Position einer Pose wird angenommen, dass die Orientierung des Quadroopters bekannt ist. Es ist zu berücksichtigen, dass die Beschleunigungssensorik fest im Quadroopter verbaut ist. Der resultierende Kraft-Vektor ist in ein Koordinatensystem zu transformieren, dessen xy-Ebene parallel zu der Start-Orientierung des Quadroopters ausgerichtet ist.

Anschließend kann der Kraft-Vektor entlang der Achsen des Quadroopter-Koordinatensystems aufgeteilt werden und die Beschleunigungen auf den Quadroopter zweifach integriert werden. Dieses Vorgehen gelingt lediglich bei idealen Daten. Da in Störungen auf das Signal wirken können, sind geeignete Methoden der Signalverarbeitung anzuwenden.

5.3.2 Signalaufarbeitung

Nachfolgend soll auf die Einflüsse einer Messung eingegangen werden, welche sich die Qualität der ermittelten Pose auswirken. Hierzu werden verschiedene Störgrößen am Beispiel eines Szenarios betrachtet und mögliche Gegenmaßnahmen gezeigt.

An dieser Stelle sei angemerkt, dass bei dem Vorgehen der Ansatz für online-Datenverarbeitung genutzt wurde. Hierbei fließen ausschließlich Daten ein, welche maximal den aktuell betrachteten Zeitstempel aufweisen.

veranschaulichendes Szenario

Im gewählten Szenario wird die z-Achse beziehungsweise Flughöhe als Beispiel betrachtet. Die Kurve setzt sich aus idealisierten Teilstücken zusammen (siehe Abbildung 11 (Seite 16), oben):

- Initialisierung: horizontale Gerade

- Abflug: negativer Cosinus (1/2 Schwingung)
- Schweben: horizontale Gerade
- Landung: zwei entgegengesetzte Parabeln³
- Abschalten: horizontale Gerade

Die Beschleunigungsdaten wurden aus der genannten Kurve mittel zweifacher zeitdiskreter Differentiation gebildet (siehe Abbildung 11 (Seite 16), unten).

Durch dieses Vorgehen ist sichergestellt, dass ein Vergleich zwischen original Kurve und den rekonstruierten Verläufen gebildet werden kann. In diesem Zusammenhang wurde sichergestellt, dass eine Rekonstruktion ohne aufgeprägte Störgrößen möglich ist. (vgl. [7])

³Die Parabeln sind so angeordnet, dass der Berührungszeitpunkt zeitlich in der Mitte der Landungsphase liegt. Da die Parabeln einen identischen Streckfaktor besitzen, entspricht der Übergang zwischen den Parabeln einer knickfreien Kurve.

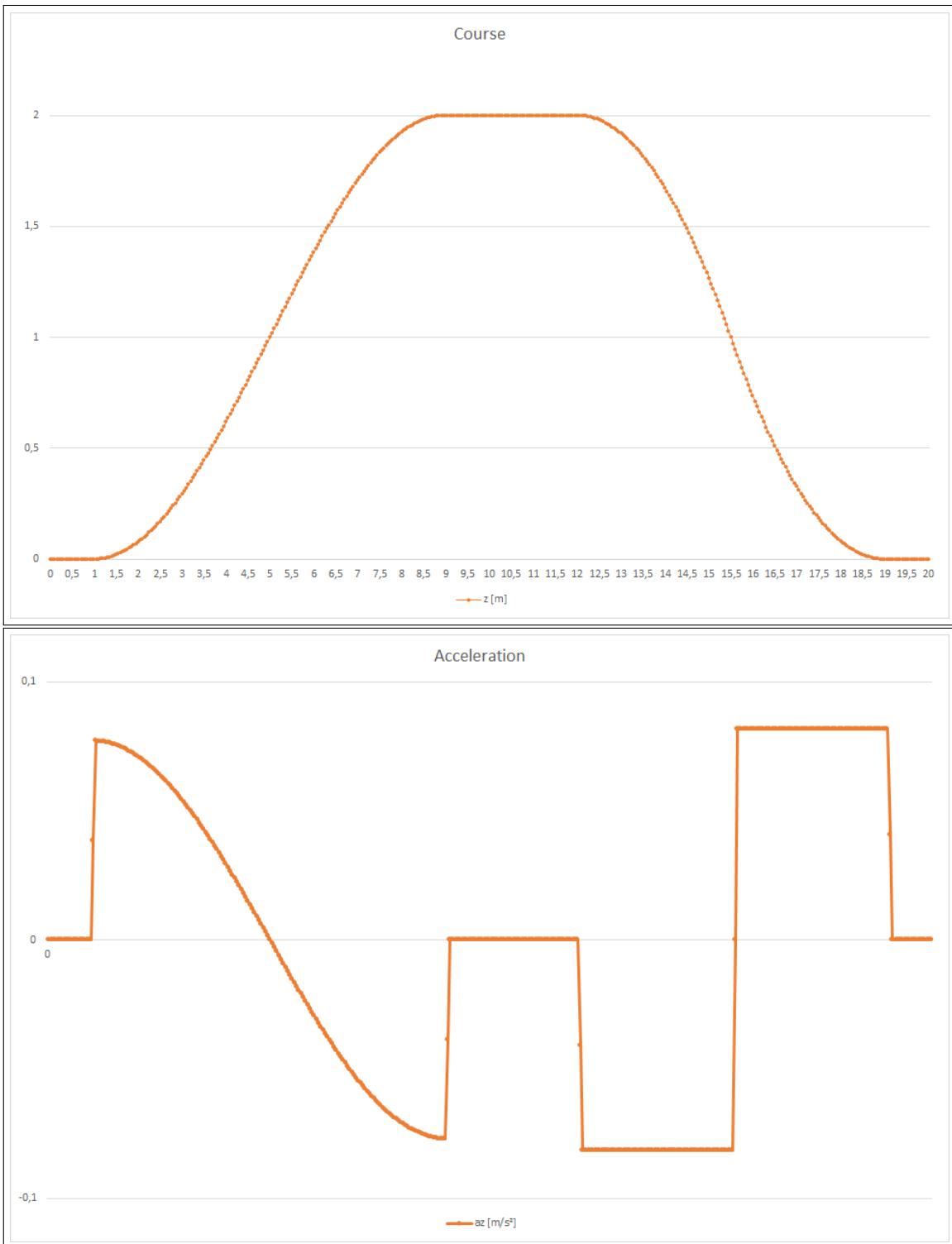


Abbildung 11: ORIGINALKURVE DES Szenarios

In Abbildung 11 sind der idealisierte Höhneverlauf (oben) und die daraus abgeleitete Beschleunigung (unten) aufgetragen. Sofern nicht anderweitig beschrieben, wird im Folgenden eine Abtastrate von 20 Datenpunkten pro Sekunde angenommen.

Rauschen

Rauschen eines Signals kann durch die Quantisierung von analogen Daten auftreten

(vgl. [7]) (Seite 100, Abbildung 4.5) oder in dem Anwendungsfall einer Beschleunigungsmessung auch von Vibrationen des Systems beeinflusst werden. Es ist darauf zu achten, dass Überläufe von digitalen Speichertypen vermieden werden, da diese einen signifikanten Anteil an rauschenden Signalen ausmachen können. (vgl. [7])

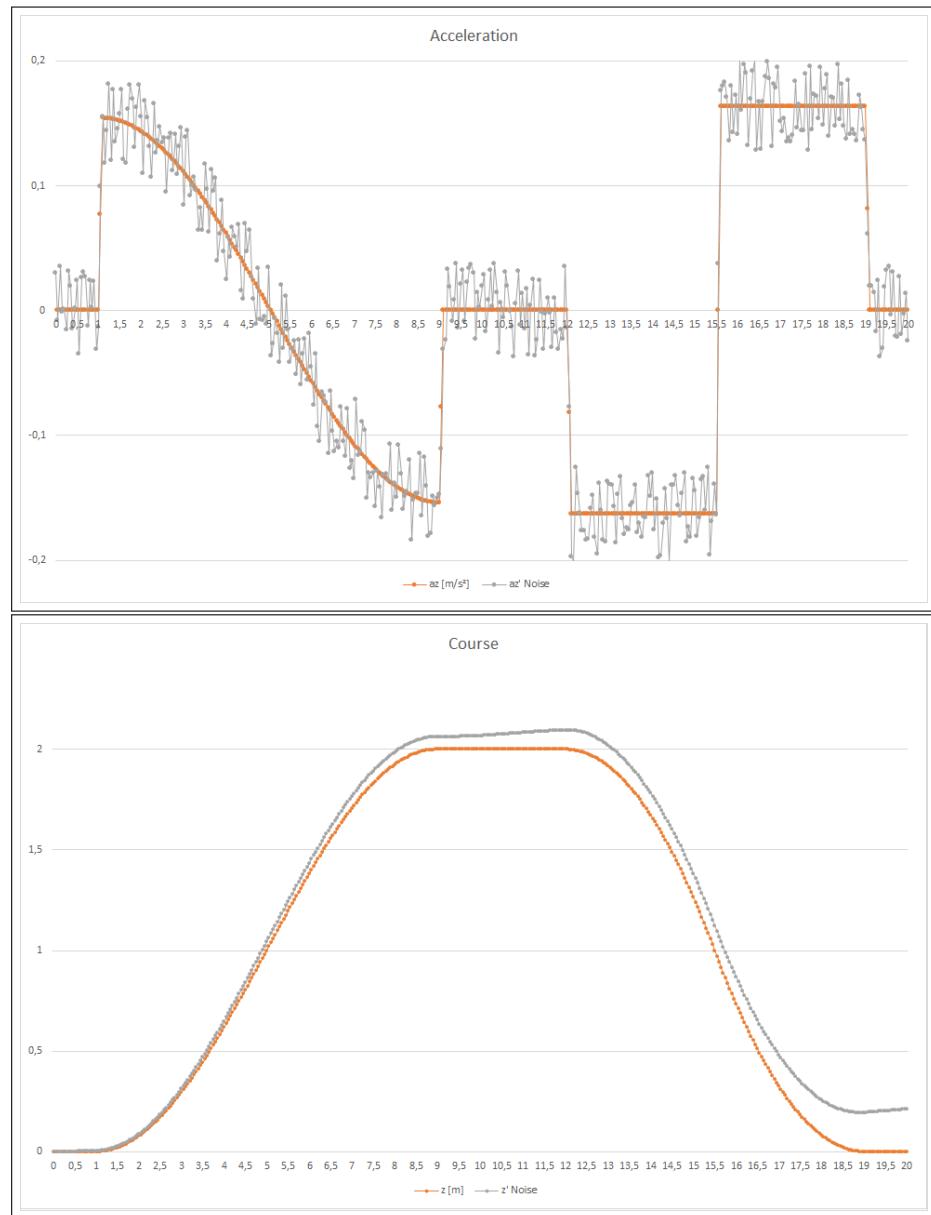


Abbildung 12: EINFLUSS VON SIGNALRAUSCHEN

Um das aufgeprägte Rauschen zu erzeugen wurden Zufallszahlen im Bereich $[-1, 1]$ erzeugt und diese mit dem Faktor einer angenommenen absoluten Genauigkeit (0.375% für einen Wertebereich $[-15 \text{ m/s}^2, 15 \text{ m/s}^2]$) und einer angenommenen relativen Genauigkeit (0.5%) verrechnet.

Entsprechend der Literatur (vgl. [7]) kann ein Tiefpass-Filter zur Glättung von Rauschen eingesetzt werden. Zur Berechnung des Datensatzes wurde eine vereinfachte Variante eines Tiefpass-Filters 1. Ordnung als online-Verfahren herangezogen (vgl. [21]).

In Abbildung 13 (Seite 18) zeigt sich, dass ein Tiefpass-Filter nicht den gewünschten Erfolg zeigt. Alternativ wurde ein Median-Filter angewandt, welcher zu geeigneteren Ergebnissen führt.

Zudem zeigt im Vergleich unterschiedlicher Übertragungsraten⁴, dass eine höhere Übertragungsrate mit vergleichbarem Rauschen eine bessere Rekonstruktion ermöglicht.

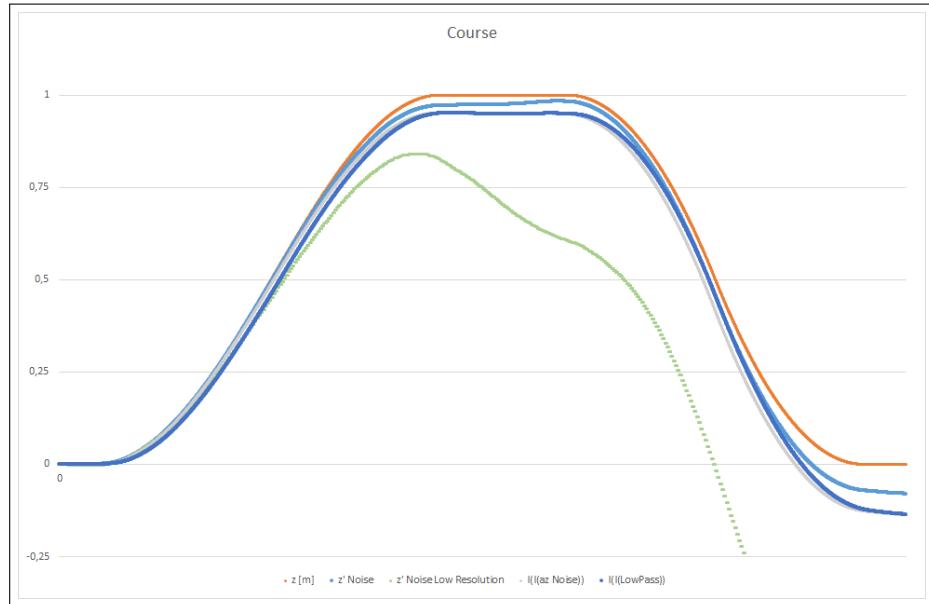


Abbildung 13: EINFLUSS VON SIGNALRAUSCHEN

Die Rekonstruktion der Kurve „z’ Noise“ wird durch einen Median-Filter für eine Wertemenge von 5 Werten als online-Verfahren durchgeführt.

Für die Tiefpass-Filter wurde ein Filter-Faktor von $\frac{31}{32}$ gewählt. Ein höherer Wert ermöglicht eine stärkere Filterung der Eingangsdaten.

Das zeitliche Intervall des als „Low Resolution“ markierten Datensatz ist um genau eine Größenordnung größer, als das Intervall des Vergleichsdatensatzes.

Ausreißer

1977 wurde von Tukey beschrieben, dass ein gleitender Median für die Glättung einer Datenreihe eingesetzt werden kann. (vgl. [8]) Im Buch wird ein Median über drei Werte vorgeschlagen. Eine Erweiterung der Anzahl der einbezogenen Werte ist möglich, wobei zu berücksichtigen ist, dass die Median-Funktion bei einer geraden Anzahl an Werten den Durchschnitt der mittleren Werte bildet.

⁴Für den Beispiel-Datensatz wurde hierfür ein kürzeres Zeitintervall gewählt.

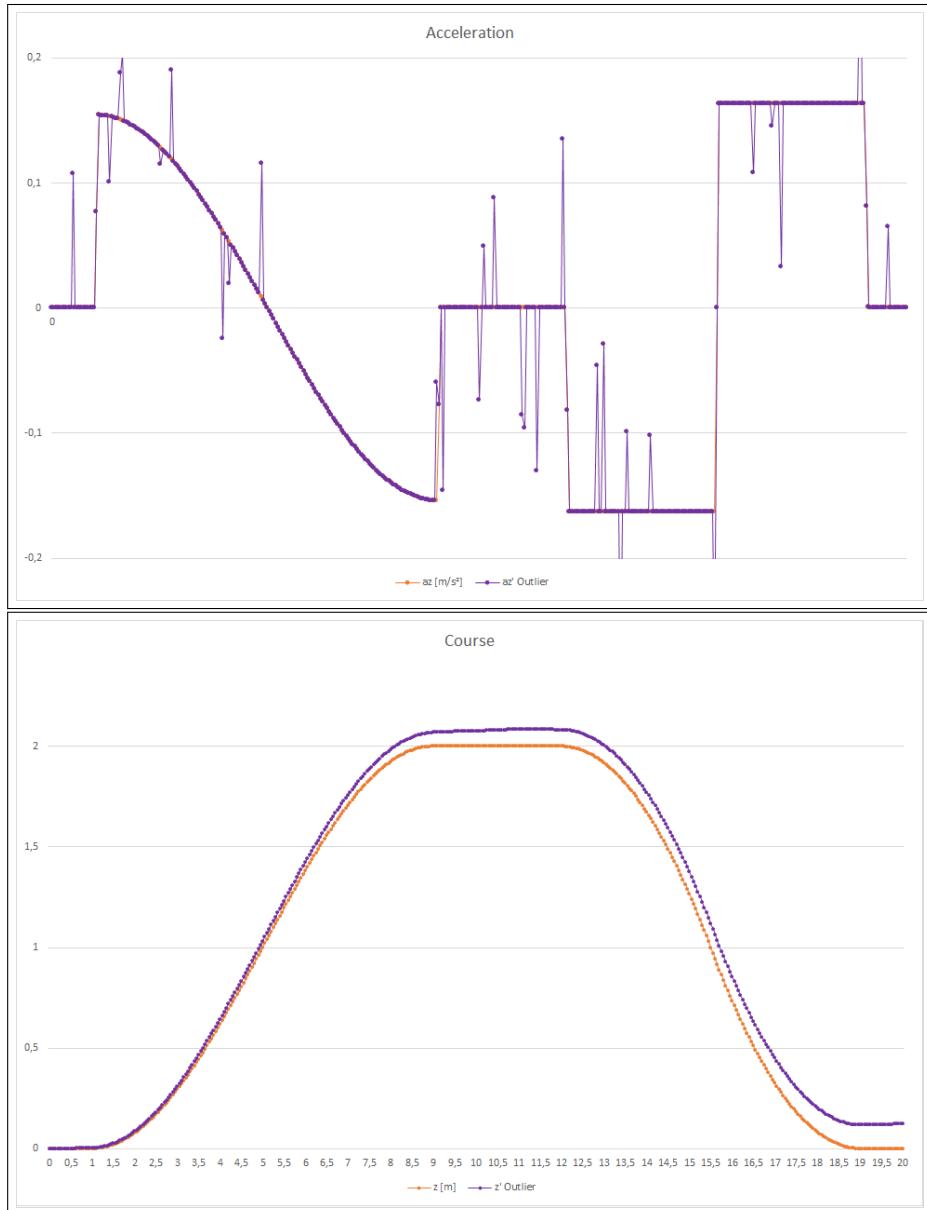


Abbildung 14: EINFLUSS VON AUSREISSERN

Die Höhe der aufgeprägten Ausreißer wurde mittel einer Datenreihe von Zufallszahlen im Bereich $[-1, 1]$, verrechnet mit dem beliebigen Faktor 0.125, generiert. Für die zeitliche Lage wurde eine Datenreihe von Zufallszahlen im Bereich $[0, 1]$ mit einem Schwellwert von 0.925 verglichen. Daraus folgt, dass etwa 7.5% der Datenwerte einen zusätzlichen Wert aufgeprägt bekommen.

konstante Nullpunkt-Abweichung

In Abbildung 15 (Seite 20) soll aufgezeigt werden, dass sich eine geringfügig Abweichung vom Nullpunkt signifikant auf den zweifach integrierten Wert auswirkt.

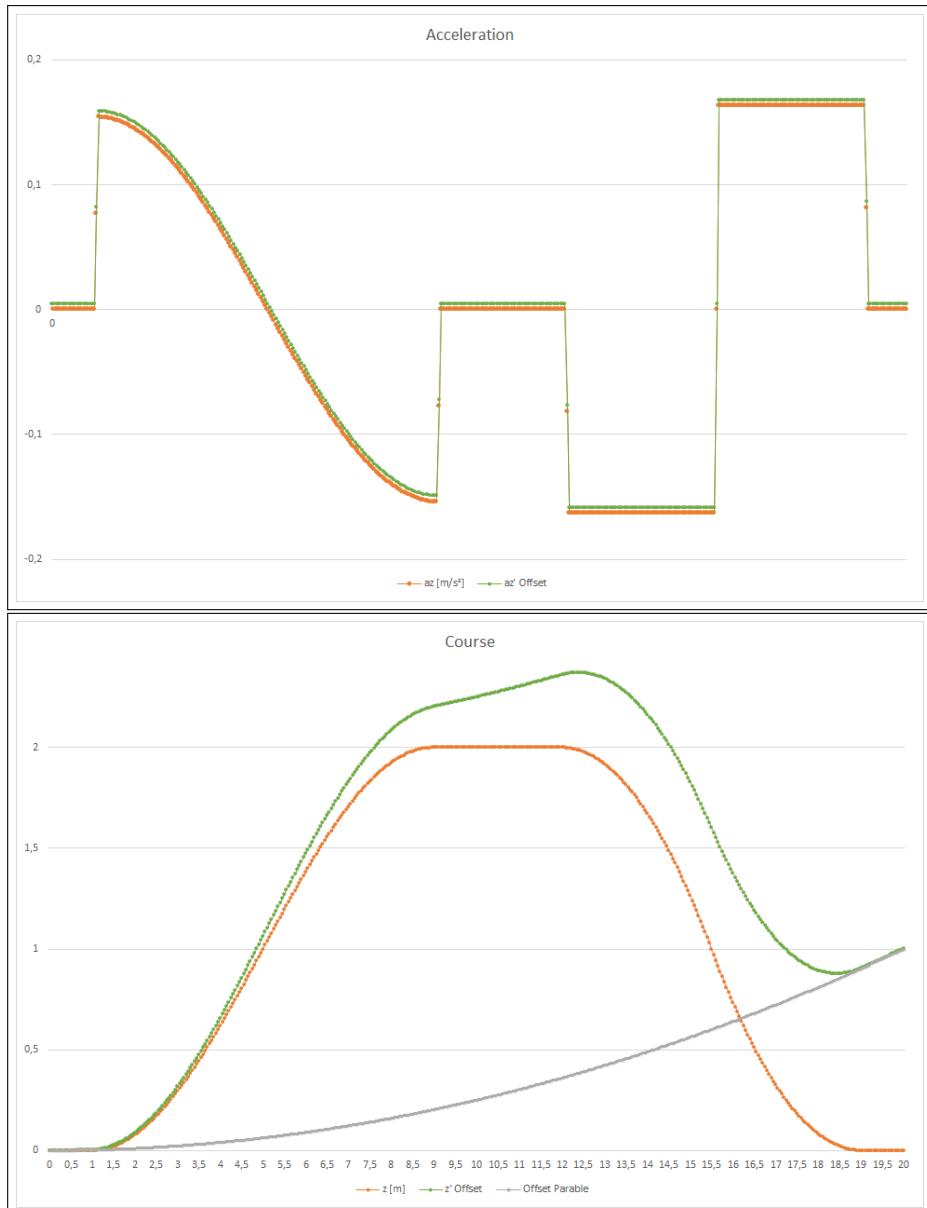


Abbildung 15: EINFLUSS VON KONSTANTER NULLPUNKT-ABWEICHUNG

Die hier gezeigte Abweichung der Beschleunigungsdaten beträgt 0.0025 m/s^2 . Die Superposition der rekonstruierten Kurve entspricht einer in grau dargestellten Parabel mit einem Streckfaktor von $\frac{\Delta a}{2}$.

6 Eingesetzte Hardware

In dieser Projektarbeit wurden zwei unterschiedliche Quadrokopter eingesetzt.

Die zu Beginn der Projektarbeit eingesetzte Quadrokopter des Herstellers *COEX* wird in diesem Kapitel ebenfalls beschrieben. Vorrangig sollen hier Erkenntnisse über das Verhalten dieser Drohne und mögliche Lösungswege der Problemstellung erläutert werden.

Nach Austausch der Hardware wurde die *ArDrone 2.0* des Herstellers *Parrot* genutzt. Dieser Quadrokopter wurde bereits erfolgreich durch die DHBW Karlsruhe im Zuge der Labor-Vorlesung *Robotik* eingesetzt.

6.1 COEX Drohne

Bei dem für die DHBW Karlsruhe neu angeschafften Quadrokopter handelt es sich um das Modell *Clover 4.20* des Unternehmens *Copter Express (COEX)*.

Das Quadrokopter-Familie *Clover* wurde vom Hersteller zur Ausbildung und Forschung an Quadrokoptern entwickelt. Das Modell besitzt einen Rahmen, welche die Rotoren bei Kollisionen schützen soll.

Interner Flight Controller, ROS Kommunikation via Pie 4.

Probleme bei Inbetriebnahme - Beispielprogramm des Herstellers bringt nicht das erwartete Ergebnis.

6.1.1 Control Stack

Als *Flight Controller* wird das Modell *PX4 Racer* genutzt. Die Firmware, sowie weitere Software zur Interaktion mit der Drohne *Clover 4.20* werden von *Dronecode Foundation* bereitgestellt.

Die Anbindung an *ROS* wird durch einen *Raspberry Pie 4* realisiert. Hierbei wird der On-Board Computer als *roscore* genutzt.

6.1.2 Sensorik

- Gyroskop
- Laser-Abstandsmessung zum Boden
- GPS

- Bodenkamera

6.1.3 Aufbau des Bausatzes

Aufbau

Der Aufbau und die Verkabelung des Quadroopters *Clover 4.20* wurden entsprechend der Hersteller-Vorgaben (siehe [25]) durchgeführt.

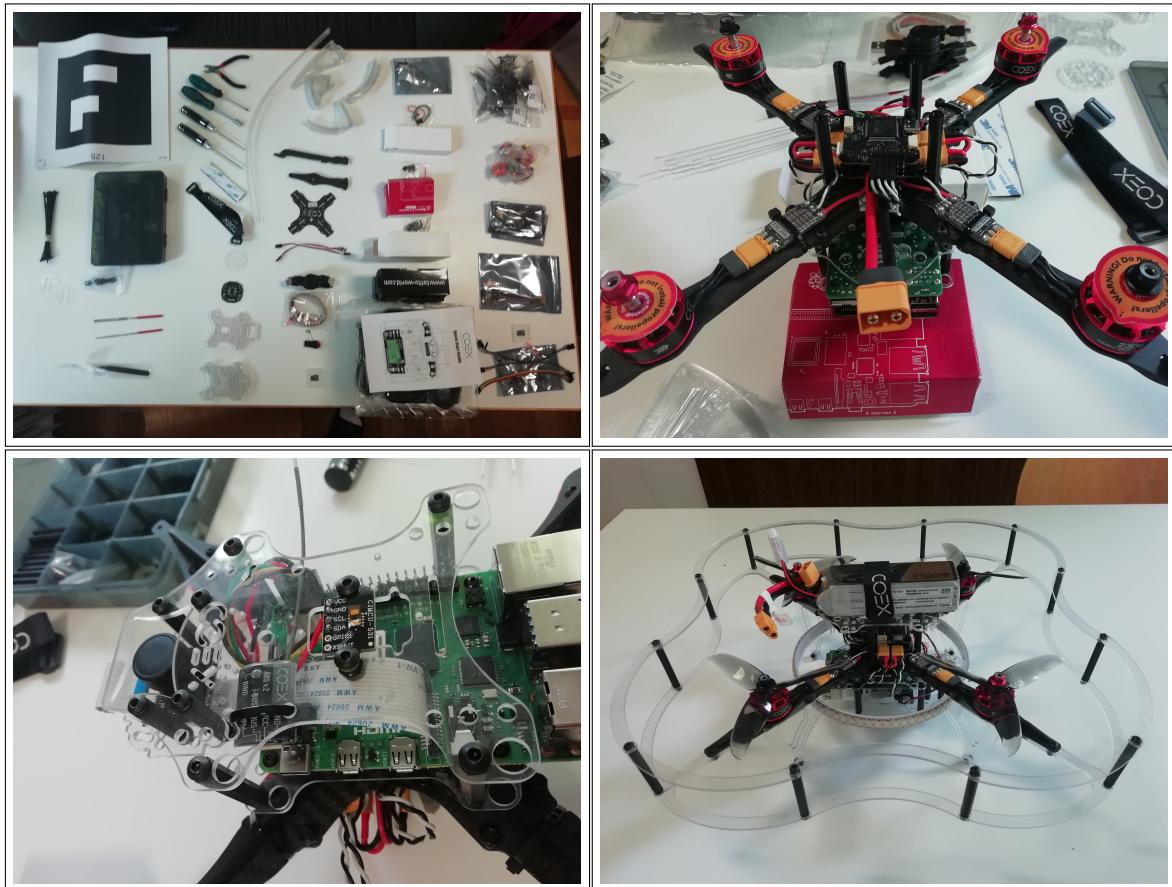


Abbildung 16: AUFBAU DES BAUSATZES FÜR DIE DROHNE *CLOVER 4.20*

Abbildung 16 zeigt etappenweise den Aufbau des Bausatzes für die Drohne *Clover 4.20*. Hierbei ist die zeitliche Anordnung in der Collage zeilenweise zu interpretieren.

6.1.4 Inbetriebnahme

Konfiguration des Flight Controllers

Die Konfiguration wurde entsprechend der Hersteller-Empfehlung mittels die Software *QGroundControl* durchgeführt. Sämtliche Anweisungen der Hersteller-Vorgaben (siehe [26]) wurden befolgt.

Das für den *Flight Controller* aufgespielte Image entspricht der Version v0.22.

Testflug

Nach einer vollständigen Konfiguration des *Clover 4.20* wurde ein Beispielprogramm zum Testen des Quadroopters genutzt. Dieses Programm wird vom Hersteller des *Flight Controller* zur Verfügung gestellt [27].

Das Programm sieht vor, den Quadroopter nach dem Start auf einer Höhe von 2 m in einem Hover-Modus zu halten. Um die Sicherheit dieses Versuches zu erhöhen, wurde die Flughöhe dieses Testflugs reduziert. Im Verlauf des Testfluges zeigte sich, dass die vorgegebene Höhe nicht gehalten wurde. Der Quadroopter wurde durch eine baulichen Begrenzung davon abgehalten, weiter an Höhe zu gewinnen.

Eine Rekalibrierung der Sensorik konnte diesem Umstand nicht entgegenwirken. Der Ansatz, diverse andere Topics zur Steuerung des *Clover 4.20* einzusetzen, schlug ebenfalls fehl.⁵ Der Austausch der Hardware hin zu einem bewährten Quadroopter wurde durch diese Flugversuche mit dem Quadroopter *Clover 4.20* hervorgerufen.

6.1.5 Mögliche Lösung der Aufgabenstellung

COEX-Package

Für die Interaktion mit der *COEX*-Drohne wurden diverse Klassen erstellt, um einzelne Aspekte der Interaktion mit der Drohne umsetzen zu können (siehe Kapitel 7.6.6 *coex Package* (Seite 39)).

Nach dem Wechsel auf die andere Drohne wurde die Aktualisierung dieses Package nicht weiter verfolgt. Sofern eine Einbindung der *COEX*-Drohne in die Ergebnisse dieser Projektarbeit durchgeführt werden soll, muss dieses Package entsprechend angepasst werden.

geeignete Topics

Mit dem *ROS-Topic* `setpoint_raw/local` kann eine Regelung in der XY-Ebene umgesetzt werden. Die Höhenregelung kann mit dem *ROS-Topic* `set_attitude/thrust` eingeführt werden. Die Umsetzung mit den genannten *ROS-Topics* entspricht der Lösung der Aufgabenstellung entsprechend Kapitel 2 *Problemstellung* (Seite 2).

Alternativ zu der genannten Lösung können Stellgrößen als Befehle der Funkfernsteuerung immitiert werden. Das *ROS-Topic* `/mavros/rc/override` erlaubt das Überschreiben der RC-Kanäle, somit können die Eingaben durch den Regelkreis übernommen werden. Eingaben Nutzender sind durch eine entsprechend implementierte Daten-Brücke weiterhin möglich.

⁵Eine Auswahl an Herangehensweisen ist im Git-Repository im Ordner Text_coex hinterlegt.

hilfreiche Literatur

Nachfolgend sollen Internetseiten genannt werden, welche die Einarbeitung in den Umgang mit der *COEX*-Drohne vereinfachen können.

- <https://clover.coex.tech/en/wifi.html>
- https://clover.coex.tech/en/simple_offboard.html
- https://docs.px4.io/master/en/ros/mavros_offboard.html
- https://docs.px4.io/master/en/flight_modes/offboard.html
- https://mavlink.io/en/services/manual_control.html
- http://wiki.ros.org/mavros#mavros.2FPlugins.manual_control
- https://mavlink.io/en/messages/common.html#SET_POSITION_TARGET_LOCAL_NED

Spezifische Verweise sind im Quellcode des *COEX*-Package hinterlegt.

6.1.6 Troubleshooting

Platinenfehler

Es zeigte sich nach Messungen, dass die Spannungsversorgung des LED-Streifens verpolt an die entsprechende Platine angebracht wurde. Als Folge wird der Verpolungsschutz des LED-Streifns aktiv, wodurch die Spannung aller Komponenten des *Control Stack* auf etwa 0.6V abfällt.

Der LED-Streifen zählt nicht zu des essentiellen Bauteilen des Quadrokopter. Der Quadrokopter kann ohne die Nutzung des LED-Streifens betrieben werden.

Bus-System des RC Empfängers

Der RC-Empfänger gibt die Funk-Signale über einen Bus an den *Flight Controller* weiter. Hierbei ist zu berücksichtigen, dass der RC-Empfänger die beiden Bus-Protokolle *i-Bus* und *s-Bus* versenden kann. Der Inhalt beider Protokolle ist identisch. Der Unterschied besteht in der Signal-Invertierung.

RC-Sender: FS-i6

RC-Empfänger: FS-X8B

Lösung

BIND-Knopf beim Einschalten für etwa 3 Sekunden gedrückt halten. Der Bus-Mode wird hiermit umgeschalten.

md5-Sum des Topics OverrideRCIn

Nach erfolgreicher Kompilierung wird nachfolgender Laufzeitfehler ausgegeben, wenn sich eine Instanz der Klasse `ros::Subscriber` oder der Klasse `ros::Publisher` auf das *ROS-Topic /mavros/rc/override* anmeldet:

[ERROR] [1643616625.226584828]: Client [/mavros] wants topic /mavros/rc/override to have datatype/md5sum [mavros_msgs/OverrideRCIn/73b27a463a40a3eda1f9fbb1fc86d6f3], but our version has [mavros_msgs/OverrideRCIn/fd1e1c08fa504ec32737c41f45223398]. Dropping connection.

Lösung

Definition von

```
struct MD5Sum<::mavros_msgs::OverrideRCIn_<ContainerAllocator>>
```

aus

```
sudo nano /opt/ros/noetic/include/mavros_msgs/OverrideRCIn.h
```

Code 1: BEFEHL ZUM ÖFFNEN DES OVERRIDERCIN-HEADERS

von dem *Raspberry Pi 4* kopieren und in der Definition des lokalen *ROS OverrideRCIn*-Headers ersetzen. Ein Versuch, den *Raspberry Pi 4* einem Update zu unterziehen, ist fehlgeschlagen. Somit ist eine unmittelbare Synchronisation der Nachrichten-Typen aufwändig.

```
template<class ContainerAllocator>
struct MD5Sum<::mavros_msgs::OverrideRCIn_<ContainerAllocator>>
{
    static const char* value()
    {
        return "73b27a463a40a3eda1f9fbb1fc86d6f3";
    }

    static const char* value(const ::mavros_msgs::OverrideRCIn_<ContainerAllocator>&)
    {
        return value();
    }

    static const uint64_t static_value1 = 0x73b27a463a40a3edULL;
    static const uint64_t static_value2 = 0xa1f9fbb1fc86d6f3ULL;
};
```

Code 2: DEFINITION DES STRUCT `MD5Sum` FÜR DAS TEMPLATE `OVERRIDERCIN`

6.2 Parrot Drohne

Um die Problematik der COEX Drohne zu umgehen, steigt diese Projektarbeit auf die Drohne um, welche die Idee für diese Studienarbeit ergeben hat. Hierbei handelt es sich um die Drohne *ArDrone 2.0*. Nachfolgend soll die Drohne und die eingebaute Sensorik näher beschrieben werden.

6.2.1 Sensorik

- Beschleunigungssensorik
- Magnetometer
- Ultraschall-Abstandsmessung zum Boden
- Frontkamera
- Bodenkamera

6.2.2 Interaktion mittels *ROS*

Treiber *ardrone_autonomy*

Für die Ansteuerung der Drohne *ArDrone 2.0* existiert eine Treiber, welche die Initialisierung und die Kommunikation mit der Drohne anbietet. Die *ArDrone 2.0* entspricht hierbei einem Client. Der *ROS*-Core wird auf dem Host-Rechner ausgeführt. Als *ROS*-seitige Schnittstelle werden verschiedene *ROS-Topics* und *ROS-Services* angeboten, welche nachfolgend näher beschrieben werden sollen.

Topics

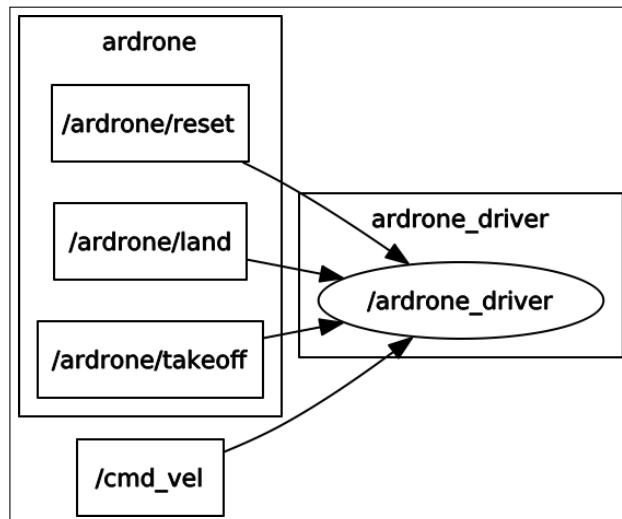


Abbildung 17: RQT-GRAPH DER ARDRONE 2.0

Die hier gezeigten *ROS-Topics* beschränken sich auf die *ROS-Topics* zum steuern des Quadroopters *ArDrone 2.0*.

ROS-Topic *NavData*

Das *ROS-Topic* *NavData* fasst alle für die verfügbaren Daten des Quadroopters in einem Nachricht zusammen. Hieraus können geeignete Informationen entnommen werden.

ROS-Topic `cmd_vel`

Entsprechend der Dokumentation werden über das *ROS-Topic cmd_vel* die linearen Geschwindigkeiten und die Winkelgeschwindigkeit übertragen. Eine Vorgabe der *Roll-Pitch-Yaw*-Winkel ist nicht vorgesehen. (vgl. [29])

Entsprechen alle Attribute der `geometry_msgs::Twist`-Nachricht dem Wert 0, schaltet die *ArDrone 2.0* in einen Hover-Modus um und versucht, die aktuelle Position zu halten. Zu beachten ist, dass diese Funktion nicht für die Aufgabenstellung herangezogen werden kann, da für den studentischen Laborversuch *Höhenregelung* Daten abweichend des Werts 0 über das *ROS-Topic cmd_vel* übermittelt werden.

Video-Streams

Die Übertragung der Video-Daten der beiden Kameras soll an dieser Stelle lediglich erwähnt werden.

6.2.3 Inbetriebnahme auf separatem Rechner

Der in Kapitel 6.2.2 *Interaktion mittels ROS* (Seite 26) Treiber kann entsprechend folgender Anleitung installiert werden:

<https://ardrone-autonomy.readthedocs.io/en/latest/installation.html>

Für das in dieser Projektarbeit entwickelte Programm werden folgende *ROS*-Knoten benötigt:

- `roslaunch ardrone_autonomy ardrone.launch`
- `rosrun PosControl AutoController`
alternativ: `rosrun PosControl ManualController`
- `rosrun keyboard KeyReader`

6.2.4 Troubleshooting

Akkus

Die mit der *ArDrone 2.0* zur Verfügung gestellten Akkus konnten selbst in vollständig geladenen Zustand keine zufriedenstellende Flugdauer garantieren. Hierfür wurden zwei neue Akkus von einem Dritthersteller bezogen. Die Lieferung der bestellten Akkus erfolgte in geringem zeitlichen Abstand zum Projektende.⁶

⁶ „geringer zeitlicher Abstand“ entspringt in diesem Kontext einem Zeitraum von drei Wochen. In diesem Zeitraum lag die Priorität des Autors auf der schriftlichen Ausarbeitung dieser Projektarbeit.

7 Implementierung

7.1 Software-Architektur

7.1.1 Package-Konzept

Der Aufbau der Architektur entspricht den Konzept der *Clean Architecture*. Diesem Prinzip folgend zeigen sämtliche Abhängigkeiten des Codes auf *weiter innen liegende* Schichten. Die Farbgebung in Abbildung 18 (Seite 28) orientiert sich an den Inhalten der Vorlesung Advanced Software-Engineering der DHBW Karlsruhe⁷.

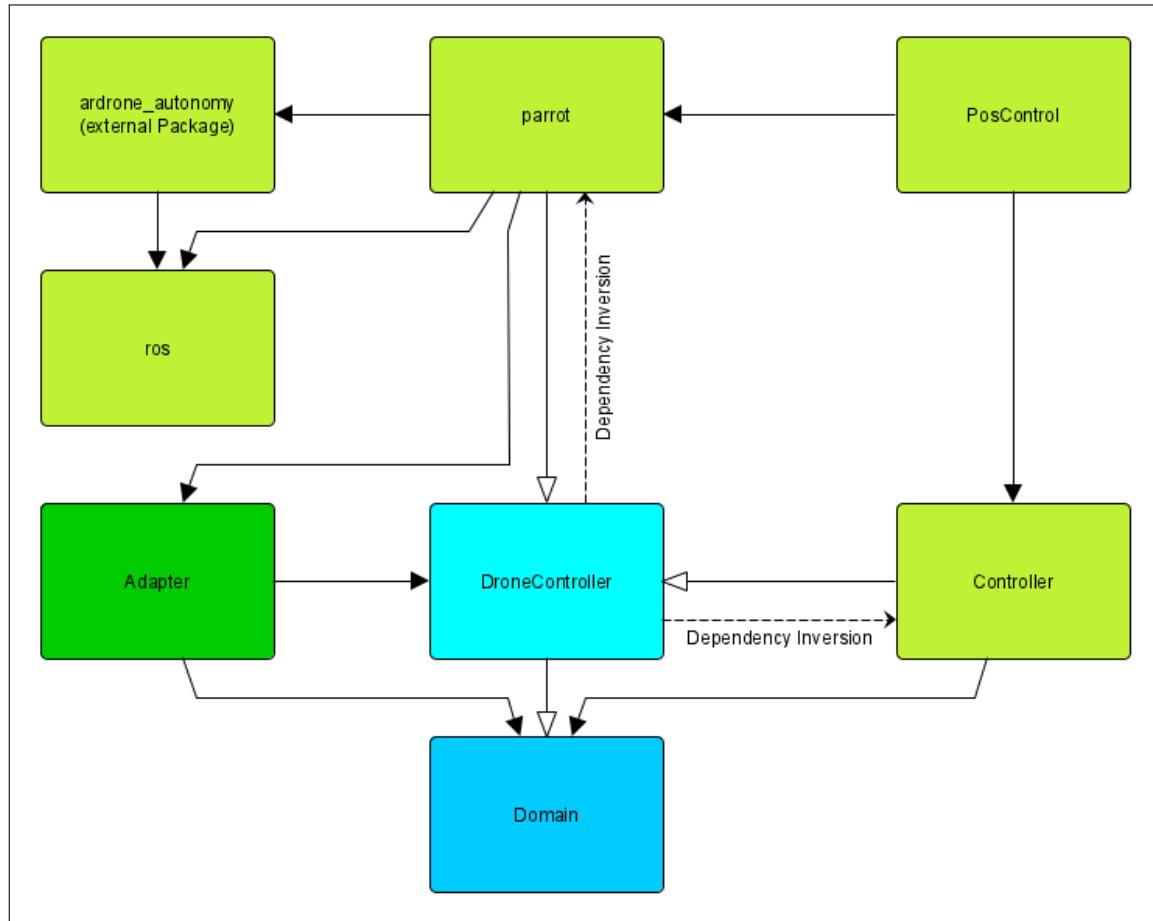


Abbildung 18: ARCHITEKTUR DES REGELUNGSSYSTEMS

Abbildung 18 zeigt das Konzept der Architektur. Die Bedeutungen der Pfeile orientieren sich an UML-Klassendiagrammen. Die Farbgebung unterscheidet die verschiedenen Schichten der *Clean Architecture*: *Domain-Layer* (blau), *Application-Layer* (hellblau), *Adapter-Layer* (grün), *PlugIn-Layer* (hellgrün).

⁷An dieser Stelle sei darauf hingewiesen, dass Abbildung 18 (Seite 28) in der Ausarbeitung für die genannte Vorlesung auftaucht.

7.1.2 Domain Package

Im *Domain Package* sind grundlegende Klassen abgelegt. Diese werden von den nachfolgend beschriebenen Schichten eingebunden. Gemeinsam mit dem *DroneController Package* bildet es das Herzstück der Anwendung. Eine eindeutige Zuordnung der Klassen in diese beiden *Packages* ist diskutabel.

7.1.3 DroneController Package

Das Entwurfsmuster des *Application-Layer* entspricht einer *Bridge*. Ziel ist es, sowohl die eingesetzte Drohne, als auch die Implementierung des Reglers mit überschaubarem Aufwand austauschen zu können.

Hierin werden die grundlegenden Aufgaben für die Regelung einer Drohne übernommen. Eine detailliertere Beschreibung findet sich im Kapitel Kapitel 7.4 *Application Layer - DroneController Package* (Seite 32).

7.1.4 Adapter Package

Im *Adapter Package* wird die Signalverarbeitung der Rohdaten übernommen und anschließend in einem aktuellen Zustand zur Weiteren Verarbeitung bereitgestellt.

7.1.5 Controller Package

Das *Controller Package* einspricht beinhaltet Regelungsglieder, welche für eine beliebige Regelung im Allgemeinen genutzt werden können. Die Interaktion wird durch die Fassade Klasse `PoseController` an das Aufgabenfeld dieser Projektarbeit angepasst.

7.1.6 parrot Package

Die Interaktion mit der in dieser Projektarbeit eingesetzten Hardware wird vom gleichnamigen *parrot Package* übernommen. Hier sind verschiedene Klassen enthalten, welche mittels *ROS ROS-Topics* mit der Drohne kommunizieren und sowie Messdaten von der Drohne zur Verarbeitung bereitstellen, sowie Steuerungsbefehle an die Drohne übersenden.

7.1.7 PosControl Package

Das *PosControl Package* kann als Dach-Software angesehen werden. hierin wird die Methode `main()`-Methode aufgerufen und die übergeordnete verwaltende Klasse initialisiert.

7.2 Abhängigkeitsgraph

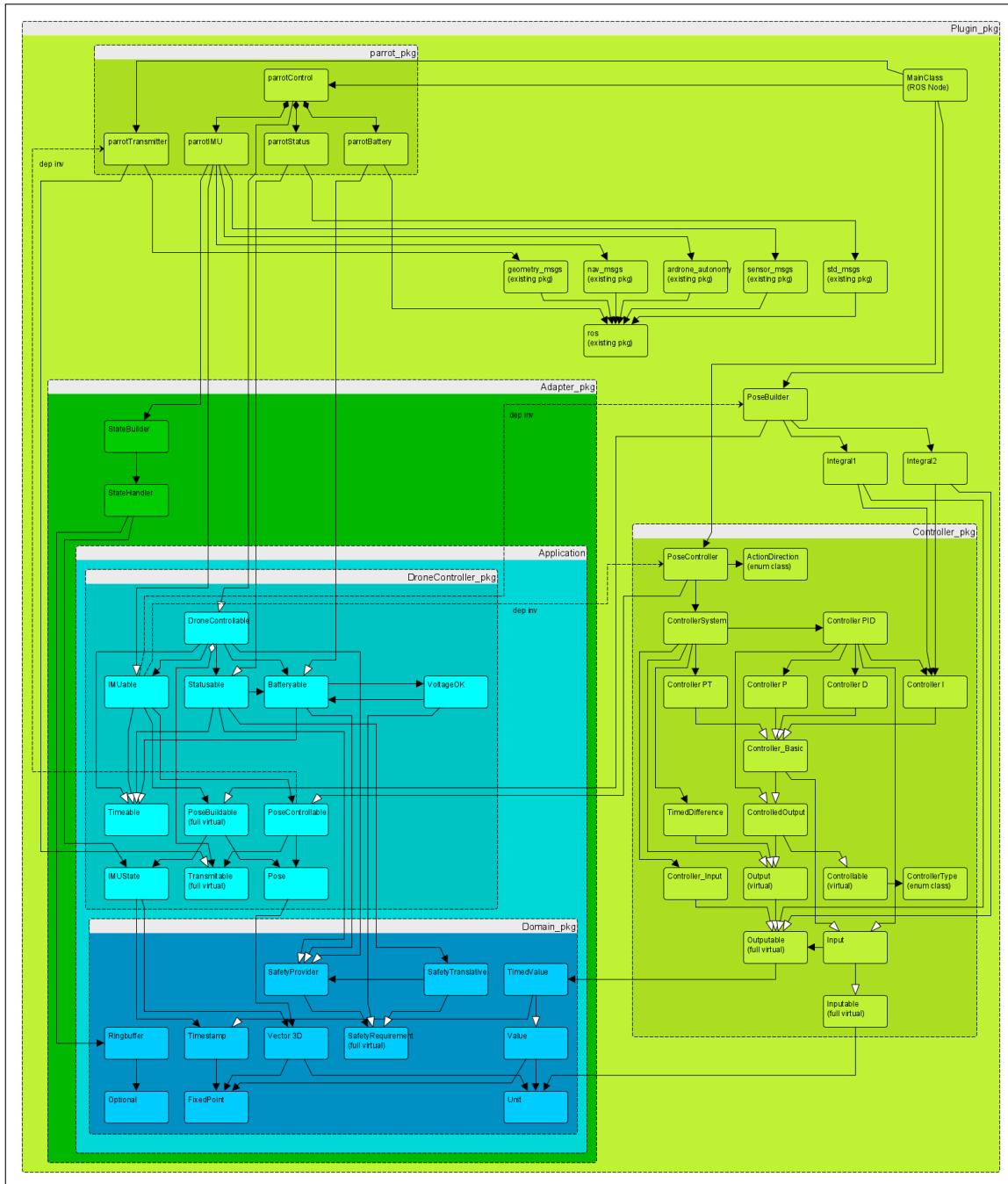


Abbildung 19: KLASSENDIAGRAMM DES PROGRAMMS

Zur Veranschaulichung der Abhängigkeiten dieses Projekts soll Abbildung 19 dienen.

7.3 Domain Layer

Der *Domain Layer* stellt als Teil der *Clean Architecture* die Ebene dar, in der allgemein gültige Typen oder Definitionen abgelegt werden können. Nachfolgend werden die Klassen beschrieben, die in diesem Projekt zu dem *Domain Layer* gehören. Sofern nicht anders betitelt, handelt es sich bei den Klassen um Klassen vom Typ *Value-Object*.

Optional

Die Klasse Klasse `Optional` soll dem Ringbuffer ermöglichen, das Nichtvorhandensein von Einträgen darstellen zu können. Diese Klasse ist als *template* implementiert und enthält neben dem Speicherplatz für die generische Instanz einen bool'schen Wert als Validierung. Hiermit wird das Arbeit mit *Null-pointern* im Kontext desr Klasse Klasse `Ringbuffer` vermieden. *Anmerkung:* Mit der Verwendung von c++17 ist eine vergleichbare Klasse Teil der Standard-Bibliothek. Die Sprachversion des Projekts ist c++11.

Ringbuffer

Die Klasse Klasse `Ringbuffer` soll einen Ringspeicher abbilden. Hier wird nicht -wie der Name vermuten lässt- ein Ringspeicher im Sinne einer ringförmig verketteten Liste implementiert. Diese Klasse Kapselt eine *Standard Template Library (STL)* vom Typ Klasse `std::vector<T>`, wobei bei Überschreiten der maximalen Anzahl an Elementen das vordere Elemente entfernt wird.

Anmerkung: Die Implementierung der Klasse Klasse `std::vector<T>` sieht vor, neue Elemente an das Ende anzuhängen.

TimedValue

Die Klasse Klasse `TimedValue` soll einen mit einem Zeitstempel versehenen Wert abbilden. Dies wird durch das Erben von den Klassen `Timestamp` und Klasse `Value` umgesetzt.

Timestamp

Mit der Klasse Klasse `Timestamp` wird ein Zeitstempel eingeführt. Alle *ROS*-Nachrichten beinhalten durch die Kapselung der Klasse `std_msgs::Header`-Klasse einen Zeitstempel.

Unit

Mit Klasse `Unit` werden Einheiten umgesetzt, um eine korrekte Übergabe von Klasse `Value`-Instanzen zur Laufzeit zu gewährleisten.

Value

Die Klasse Klasse `Value` bildet einen Wert ab. Sie besteht aus einer Klasse `Unit` und einem dazugehörigen Zahlenwert.

Vector3D

Bei der Klasse Klasse `Vector3D` handelt es sich um die Abbildung eines Vektors im dreidimensionalen Raum. Zusätzlich wird dem Vektor eine Einheit zugewiesen. Zu-

dem werden in dieser Klasse grundlegende mathematische Operationen für Vektoren implementiert.

SafetyProvider

Die Klasse `SafetyProvider` ist in ihrer Funktionalität abgeschlossen. Instanzen der Klasse `SafetRequirement` werden an einen *SafetyProvider* übergeben. auf Anfrage wird geprüft, ob die übergebenen Instanzen der Klasse `SafetRequirement` die erwartungen erfüllen. Ist dies nicht der Fall, wird die Methode `safetyTriggered` der hinterlegten Instanzen der Klasse `SafetReceiver` aufgerufen.

SafetReceiver

Die *full virtual* Klasse `SafetReceiver` fordert die Implementierung der Methode `safetyTriggered`. Hierin definiert die erbende Klasse, wie in Folge einer Verletzung der Sicherheitsanforderungen reagiert werden soll. Allgemein ist hierfür eine Landung des Quadroopters vorgesehen.

SafetRequirement

Im Sinne des *Open/Closed-Principle* wird die Klasse `SafetRequirement` als virtuelle Klasse eingeführt. Hierdurch werden Eigenschaften und Status beziehungsweise Flags des Quadroopters für Sicherheitsfunktionalitäten zugänglich gemacht.

SafetyTranslative

Eine Instanz der Klasse `SafetyTranslative` erhält eine Referenz auf eine Instanz der Klasse `SafetyProvider`. Ziel dieser Klasse ist es, die Sicherheitsfunktionalität eines anderen *SafetyProvider* als *SafetRequirement* einbinden zu können.

7.4 Application Layer - *DroneController* Package

Batteryable

Die Klasse `Batteryable` hat die Aufgabe, den Akku-Stand des Quadroopter zu überwachen. Die Klasse bietet die virtuellen Methoden Methode `getVoltage()` und Methode `getPercentage()` an, die ableitende Klasse kann die Methoden-Definition in Abhängigkeit der von der Hardware zur Verfügung gestellten Daten überschreiben.

DroneControlable

Um die Interaktion mit der Hardware gekapselt zu gestalten und zudem die korrekte Initialisierung der beteiligten Klassen zu gewährleisten, wird die Klasse `DroneControlable` als Basis für eine Fassade eingesetzt. Die Klasse selbst erhält Pointer auf die für die

Funktionalität des Quadroopters benötigten Klassen (Klasse `Batteryable`, Klasse `PoseControlable`, Klasse `IMUState`, Klasse `Statusable` und Klasse `Transmitable`). Im Konstruktor der Klasse `DroneControlable` werden die Sicherheitsfunktionalitäten der verwalteten Instanzen miteinander verknüpft.

IMUable

Die Klasse `IMUable` wurde als Basis-Klasse für die Aufnahme der *Inertial Measurement Unit (IMU)* Daten und die Weiterleitung zur Berechnung und Regelung der Pose implementiert.

Neben der Sicherheitsfunktionalität (Ermittlung von Zusammenstößen anhand eines Beschleunigungsschwellwerts) werden als Attribute Pointer auf Instanzen der Klasse `PoseBuildable` und Klasse `PoseControlable` eingesetzt. Hierbei handelt es sich im *Dependency Injections*.

IMUState

Die Klasse Klasse `IMUState` beinhaltet Attribute, welche die Daten der *Inertial Measurement Unit (IMU)* abbildet.

Anmerkung: Mit der Änderung der Hardware ergab sich, dass die Klasse `IMUState` Daten der lokalen Orientierung enthält. Der Quadroopter *Clover 4.20* sendet die Rate der Winkeländerung.

Im Sinne der Auswertung und Parameter-Optimierung wurde hier zusätzlich der gemessene Abstand zum Untergrund als Attribut aufgenommen. Dieser dient nur zu Referenzzwecken und wird, entsprechend der Aufgabenstellung (siehe Kapitel 2 *Problemstellung* (Seite 2)), nicht zur Berechnung der Pose eingesetzt.

Pose

Die Klasse `Pose` bildet die in Kapitel 7.4 *Application Layer - DroneController Package* (Seite 33) definierten Eingeschäften ab. Hierzu wenden Instanzen der Klasse `Vector3D` eingesetzt. Zudem erhält jede berechnete Pose einen Zeitstempel. Hierüber kann in einer Weiterführung des Projekts ein Flugweg zeitlich korrekt wiedergegeben werden. Für eine zukünftige Erweiterung sind bereits Attribute zur Erfassung der Unsicherheit der berechneten Pose vorhanden. Die Unsicherheiten können für Mapping-Algorithmen genutzt werden.

PoseBuildable

Die Klasse `PoseBuildable` bietet eine Basis für *Dependency Inversion* einer Factory für die Klasse `Pose`. Sie ist in diesen Paket angesiedelt, um der Klasse `IMUable` Zugriff zu ermöglichen. Eine Implementierung der Funktionalität wird in der PlugIn-Schicht übernommen.

PoseControlable

Ebenfalls als Basis für eine *Dependency Inversion* wird die Klasse `PoseControlable` genutzt. Nach der Berechnung der aktuellen Pose aus den aufgearbeiteten Beschleunigungsdaten wird die Pose an die Implementierung der Klasse `PoseControlable` übertragen. Hier werden die Stellgrößen für die Regelung des Quadroopters ermittelt und anschließend die Methode `transmitAction` der Klasse `Transmitable` aufgerufen.

Statusable

Die Aufgabe der Klasse `Statusable` ist die Interaktion mit den Status des Quadroopters und das senden von Befehlen, welche den Zustandsautomaten des Quadroopters beeinflussen. Als Maßgebliche Befehle werde Start- und Lande-Befehle übertragen.

Timable

Um eine allgemeine Zeit innerhalb des Systeme definieren zu können, wurde die Klasse `Timeable` implementiert. Die Methoden erlauben eine Ermittlung der vergangenen Zeit ab Programmstart beziehungsweise erster eingetragener Nachricht.

Anmerkung: Die Zeitstempel der mittels *ROS* übertragenen Nachrichten enthalten die Zeit entsprechend Unix Epoch Time

Transmitable

Die Klasse `Transmitable` bietet als *full virutal* Klasse die Methode `bool transmitAction(double roll, double pitch, double yaw, double thrust)` zur Implementierung an. Von dieser Klasse ableitende Klassen übernehmen die Aufgabe, Steuerungsbefehle an die Hardware zu übersenden.

ImpactOK und ImpactWatcher

Als Sicherheitsfunktionalität wird der Klasse `IMUable` eine Instanz der Klasse `ImpactWatcher` übergeben. Diese erhält zur Prüfung der Sicherheit Beschleunigungsdaten, welche durch die Klasse `StateBuilder` aufgearbeitet wurden. Überschreitet der Betrag der Beschleunigungswerte einen Schwellwert, wird die Sicherheitsfunktionalität ausgelöst.

Anmerkung: Die vorgeschlagene Vorgehensweise kann nur angewandt werden, wenn der *ROS-core* auf dem Rechner (nicht auf dem Quadroopter) ausgeführt wird.

PercentageOK und VoltageOK

Die Ableitungen der Klasse `SafetRequirement` überprüfen, ob die als Pointer übergebene Instanz der Klasse `Batteryable` die geprüfte Eigenschaft (Spannung oder Akkustand) einhält.

TimeoutOK und WatchDog

Wie aus der Namensgebung ableitbar sind diese Klassen dafür gedacht, das System über Ausfälle der Datenübertragung zu informieren. Der derzeitige Stand prüft den zeitlichen Unterschied zwischen eingegangenen Nachrichten, angewandt in der Instanz der Klasse `Statusable`. Als Verbesserung für diese Funktionalität kann ein nebenläufiger Timer implementiert werden, welcher bei Ablauf die Sicherheitsfunktionalität auslöst. Eine eingehende Nachricht würde den Timer zurücksetzen.

Rotor und Thrustable

Die Klasse `Rotor` wurde versuchsweise hinzugefügt. Unter Umständen kann hier die auf den Quadrokopter wirkenden Kräfte ermittelt werden und somit die Regelung in einer Umgebung ohne äußere Einflüsse verbessert werden.

Als Interaktion mit einem Quadrokopter wird die virtuelle Klasse `Thrustable` eingeführt. Hier werden die vier Rotoren (Instanzen der Klasse `Rotor`) und die zugehörige Geometrie des Quadroopters hinterlegt. Aus den Drehzahlen der Rotoren lässt sich die wirkende Gesamtkraft ermitteln.

Anmerkung: Diese Klassen finden im aktuellen Programm keine Verwendung.

7.5 Adapter Layer

StateBuilder

Innerhalb der Klasse `StateBuilder` werden große Teile der Signalverabreitung übernommen. Hierzu werden verschiedene Instanzen der Klasse `StateHandler` mit entsprechenden Aufgaben implementiert. Als Funktionalität bietet die Klasse `StateBuilder` vor Allem das Glätten von Signalen, sowie die Anpassung einer dauerhaften Nullpunkt-Abweichung an.

StateHandler

Die Klasse `StateHandler` erbt von der Klasse `Ringbuffer` für das *Template* der Klasse `IMUState`. Zudem werden verschiedene Methoden als Grundlage für die Signalverabreitung angeboten (Methode `IMUState getAvgState()`, Methode `IMUState getMedianState()` und Methode `IMUState getVariance()`).

7.6 Plugin Layer

7.6.1 *parrot* Package

Die Klassen in diesem *Package* entsprechen der vollständigen Implementierung der virtuellen Klassen des *DroneController-Package*. Weitere Details sind dem Code zu entnehmen.

Folgende Klassen sind verfügbar:

- Klasse `parrotBattery`
- Klasse `parrotControl`
- Klasse `parrotIMU`
- Klasse `parrotStatus`
- Klasse `parrotTransmitter`

7.6.2 Controller Package

Die in dieser *Package* umfangreiche Vererbungshierarchie wurde umgesetzt, um das parallel hierzu auszuarbeitende *Software Engineering 2*-Projekt ausführlicher umsetzen zu können. Eine schlankeres *Package* wäre möglich.

Nachfolgend sollen die implementierten Klassen dieses *Package* näher erläutert werden:

Controllable

Die virtuelle Klasse `Controllable` hält als Attribut eine Ausprägung der Enumeration `ControllerType` und deklariert zudem die Methoden für die Funktionalität, einen Regelparameter k verwalten zu können. Diese Funktionalität wird in der Klasse `Controller_Basic` definiert.

ControlledOutput

Mit der Klasse `ControlledOutput` werden die Funktionalitäten der Klassen Klasse `Controllable` und Klasse `Output` vereint.

Controller_Basic

Die Klasse `Controller_Basic` bildet die Grundlage für die in der Implementierung vorhandenen Regelbausteine, ausgenommen der Klasse Klasse `ControllerPID`.

Controller_x

Klassen, welche mit „Klasse `Controller_`“ beginnen und einen Baustein (vgl. Kapitel 4.2 *Arten von Reglergliedern* (Seite 6)) benennen, implementieren die Funktion des jeweiligen Bausteins.

ControllerSystem

Instanzen der Klasse `ControllerSystem` kapseln Instanzen verschiedener Regelbausteine (Klasse `Controller_`). Hierbei können diverse Regelbausteine als Reihenschaltung

zusammengefasst werden.

Anmerkung: In der Implementierung wird diese Klasse nicht aktiv genutzt.

ControllerType

Bei *ControllerType* handelt es sich um eine Enumeration. Hier können Parameter der Regelbausteine über eine kapselnde Instanz der Klasse **ControllerSystem** angepasst werden.

Anmerkung: Im Projektfortschritt hat sich keine Möglichkeit ergeben, dieses Funktionalität einzusetzen.

Inputable und Input

Die beiden Klassen ermöglichen einen Daten-Eingang für ein Objekt. Hierbei Besitzt die Klasse **Input** einen Pointer auf eine Instanz der Klasse **Outputable**.

Outputable und Output

Die beiden Klassen bilden den Output eines Objekts an. Hierbei ist die Methode **TimedValue getOutput()** eine virtuelle Methode und muss in einer von der Klasse **Outputable** erbenden Klasse definiert werden.

TimedDifference

Die Klasse **TimedDifference** wurde eingeführt, um ein Datum und einen Zeitstempel zu vereinen. Dies ist für die Berechnung der Regelbausteine notwendig.

7.6.3 PosControl Package

7.6.4 calling Package

Dieses *Package* wurde eingeführt, um die Erzeugung neuer *ROS-Messages* zu umgehen. Ferner bildet die Struktur dieser Klassen einen *pull Observer* ab. Die aufgerufene Methode erhält den Pointer auf die aufrufende Klasse und kann hiermit eine Entscheidung über weitere Aktionen treffen.

Anmerkung: Dieses *Package* entspricht nicht den Grundsätzen einer *ROS*-Programmierung und ist daher nicht weiter zu nutzen. Die Beschreibung dient an dieser Stelle dem Verständnis des Einsatzes dieser Klassen im *coex Package* und wird darüber hinaus nicht weiter eingesetzt.

7.6.5 threading Package

Das Package **threading** bietet die Möglichkeit wiederkehrende Aufgaben, abseits von separaten *ROS-Nodes*, bearbeiten zu können. Dieser Zusatz erlaubt ein monolithisches

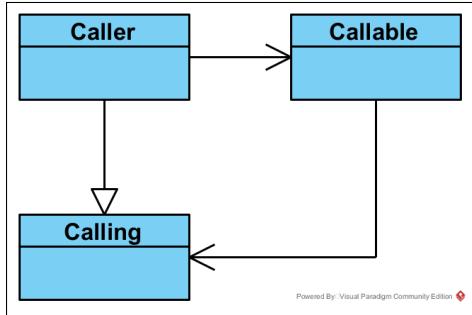


Abbildung 20: KLASSENIDIAGRAMM DES CALLING PACKAGES

Programm zu entwerfen.

Thread

Diese Klasse bietet die Basis für die Thread-Implementierung, indem die Klasse `std::thread` gekapselt wird. Hier wird mit Aufruf der Methode `start()` eine neue Instanz auf den zugehörigen *Pointer* initialisiert. Zusätzlich implementiert die Klasse `Thread` einen Sperr-Mechanismus, um synchrones Schreiben auf das Run-Attribut zu vermeiden.

rosThread

Die Klasse Klasse `rosThread` erweitert die Klasse Klasse `Thread` um eine generische Variable `T Payload`, welcher von den erbenden Klassen zum Versand genutzt werden kann. Die Methode Methode `T runOnce(T Payload)` ist als *full virtual* implementiert, somit wird ein Überschreiben durch erbende Klassen erzwungen. Die Klasse Klasse `ros::Rate` ermöglicht eine frequentiell pausierte Abarbeitung des der auszuführenden Methode `T runOnce()`.

Anmerkung: Idealerweise sollte eine Instanz der Klasse Klasse `ROS::NodeHandler` ebenfalls in dieser Klasse integriert sein. Tests während der Entwicklung zeigten, dass dies nicht umsetzbar ist. Eine Begründung hierfür konnte nicht gefunden werden.

AutoPublisher

Wie der Name der Klasse `AutoPublisher` erahnen lässt, wird hier ein Klasse `ros::Publisher` implementiert. Mit dem Aufruf der Methode `runOnce()` wird wird der in Klasse `rosThread` gespeicherte `T Payload` mit der Instanz der Klasse `ros::Publisher` versandt.

AutoClient

Die Klasse `AutoClient` kapselt eine Instanz der Klasse `ros::ServiceClient` und bietet somit die Option, Service-Anfragen regelmäßig senden zu können.

7.6.6 coex Package

Dieses Kapitel beschreibt die Klassen, welche zur Interaktion mit der zuerst eingesetzten Hardware genutzt wurden.

Anmerkung: Die Implementierung dieses *Packages* wurde vor der konzeptionellen Änderung des *Application-Layers* umgesetzt. Die Beschreibung der Klassen dieses Paketes dient der Einarbeitung nachfolgender Studienarbeiten. Auf Grund der Änderungen im Code kann dieses *Package* nicht Kompiliert werden. Abbildung 21 (Seite 40) soll eine Portierung des Programms auf den Quadrokopter *Clover 4.20* unterstützen.

coexBattery

Wie der Klassename vermuten lässt überwacht diese Klasse die Batteriestand des Quadroopters.

Diese Klasse entspricht der Klasse [parrotBattery](#) der *ArDrone 2.0*.

coexControl

Die Klasse [coexControl](#) ist als Fassade eingesetzt und bietet Anwendenden Zugriff auf diverse Funktionen der gekapselten Klassen.

Diese Klasse entspricht der Klasse [parrotControl](#) der *ArDrone 2.0*.

coexMC

Die Abkürzung *MC* im Klassennamen steht für *Manual Control* und deutet hiermit sowohl das genutzte *ROS-Topic* als auch den zugrundeliegenden Nachrichtentyp an. Laut Entwickler-Literatur (siehe Kapitel 6.1.5 *Mögliche Lösung der Aufgabenstellung* (Seite 24)) werden Steuerungsdaten der vier Freiheitsgrade an den Quadroopter gesendet.

Diese Klasse erbt von der Klasse [coexTransitable](#)

Diese Klasse entspricht der Klasse [parrotTransmitter](#) der *ArDrone 2.0*.

coexOrientation

Es war geplant, die Pose des Quadroopters in dieser Klasse berechnen zu lassen. Im weiteren Projektverlauf hat sich ergeben, hierfür eine separate Klasse zu erstellen.

Diese Klasse entspricht der Klasse [parrotIMU](#) der *ArDrone 2.0*.

coexRC

Die Klasse [coexRC](#) ist als Fassade für die beiden nachfolgenden implementiert.

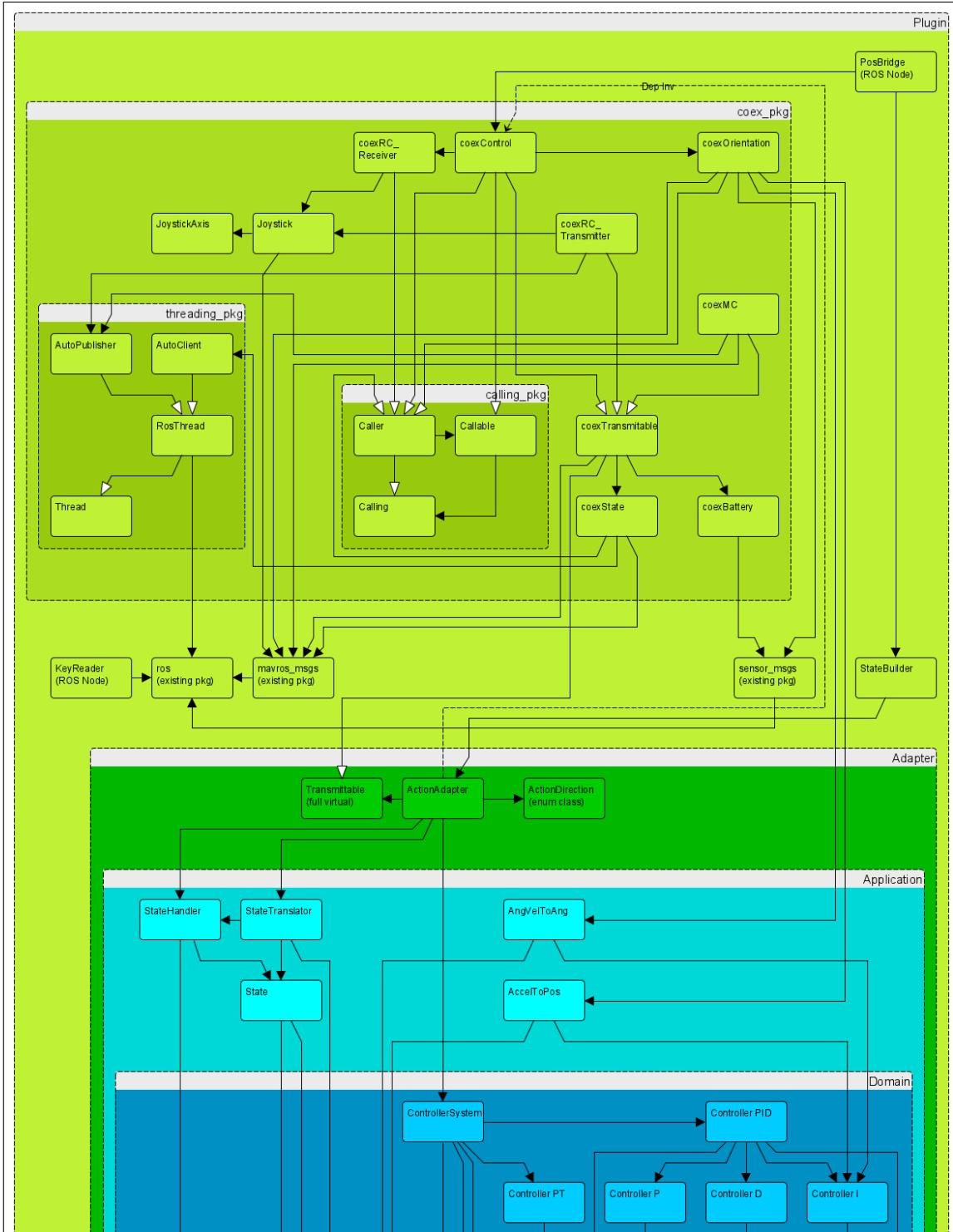


Abbildung 21: KLASSENIDIAGRAMM DES COEX PACKAGES

Zur Veranschaulichung der Abhängigkeiten dieses Pakets soll Abbildung 21 dienen.

coexRC_Receiver

Die Klasse `coexRC_Receiver` liest vom ROS-Topic `/mavros/rc/in` und somit die Eingaben der Funkfernsteuerung. Eine nähere Erläuterung, welche Daten übermittelt werden, findet sich als Kommentar im Code.

coexRC_Transmitter

Als *RC-Transmitter* wird die Klasse bezeichnet, welche Nachrichten an das *ROS-Topic /mavros/rc/override* veröffentlicht (vgl. Kapitel 6.1.5 *Mögliche Lösung der Aufgabenstellung* (Seite 23)).

Diese Klasse erbt von der Klasse [coexTransitable](#)

coexState

Die Klasse [coexState](#) übernimmt die Interaktion mit dem Zustandsautomaten, welcher im *Flight Controller* des Quadroopters realisiert ist. Anfragen zur Änderung eines Zustand beziehen sich maßgeblich auf den Start und die Landung eines Flugs.

Diese Klasse entspricht der Klasse [parrotStatus](#) der *ArDrone 2.0*.

coexTransitable

Diese Klasse erbt von der Klasse [Transitable](#). Hierin wird die Steuerung innerhalb dieses *Packages* als eine normierte *Manual Control*-Nachricht mit dem Wertebereich [-1, 1] definiert.

Joystick

Hier wird ein Hilfskonstrukt für die Transformation von Daten der Funkfernsteuerung eingeführt. Die Klasse implementiert vier Instanzen der Klasse [JoystickAxis](#).

JoystickAxis

Hier werden die Rohdaten der Kanäle der Funkfernsteuerung in normierte Daten zu überführt. Zudem können normierte Werte in den Wertebereich der Funkfernsteuerung transformiert werden.

8 Ergebnis

Um ein Ergebnis bewerten zu können, wird nachfolgend einer Auswahl aussagekräftiger Daten eines Fluges gezeigt.

Hierbei wurden zwei Starts in dem Datendatzen durchgeführt. Zur besseren Interpretation wurden Daten entfernt, welche entsprechend der Implementierung irrelevant sind.

Ein Mitschnitt, welcher mittels *rosbag* entstand, ist verfügbar. Die genannten Daten wurden mittels geeigneter Software (`rosrun BagPlotter parrot` und `rosbag play <File>.bag`) in menschenlesbare *.txt* Dateien überführt. Diese Daten wurden anschließend zur Ermittlung der von der Implementierung geschätzten Pose genutzt. Dieser Vorgang wurde aus Sicherheitsgründen in einer Simulation (*PosControl/Simulant.cpp*) durchgeführt.

Im Zuge dieses Projekts wurden diverse Sicherheitsfunktionen zum Quadrokopter hinzugefügt. Da sich diese Projektarbeit aus interne Sensoren beschränkt, werden diese als Basis der Sicherheitsfunktionen eingesetzt. Aus den übertragenen Daten kann ein Timeout⁸, sowie ein Zusammenstoß mit einem Objekt ermittelt werden. Als weitere Funktionalität wird ein Absturz auf Grund einer zu geringen Akku-Ladung verhindert. Jedes Sicherheitsfunktion löst das Landemanöver der *ArDrone 2.0* aus.

8.1 Analyse realer Flugdaten

8.1.1 Datenbasis

Dieses Kapitel zeigt die verschiedenen Daten des Flugs, um diese in der nachfolgenden Analyse einordnen zu können.

Der gezeigte Datensatz enthält zwei Flüge, welche naheinander aufgenommen wurden. Aus diesen Daten wurden irrelevante Einträge zwischen den Flügen entfernt. Durch diesen Eingriff wurde der berechnete Verlauf des zweiten Fluges nicht beeinflusst, da die Status des Quadroopters zu geeigneten Zeitpunkten zurückgesetzt werden.

Der vom Zustandsautomaten der *ArDrone 2.0* einnehmbare ZustandIDs sind im Package *ardrone_autonomy* wie folgt definiert:

⁸Der *Timeout* wird in diesem Projekt mit dem Eingang einer neuen Nachricht ermittelt. Geeigneter wäre das Starten eines Timers beim Eingang einer Nachricht. Verstreicht die Zeit des Timers soll das Landemanöver initialisiert werden.

StatusID	Status-Text
0	Unknown
1	Init
2	Landed
3	Flying
4	Hovering
5	Test
6	Taking off
7	Goto Fix Point
8	Landing
9	Looping

Aus den Flugversuchen zeigte sich, dass StatusID 6 (*Taking off*) nicht gesendet wird. Stattdessen wird StatusID 7 (*Goto Fix Point*) unmittelbar nach der *Takeoff*-Message eingenommen. StatusID 8 (*Landing*) wird durch StatusID 9 (*Looping*) abgebildet.



Abbildung 22: TESTFLUG: STATUSID, FLAGS UND HÖHENPROFIL

Um die Aussagekraft der Graphik zu gewährleisten wurde die Ordinatenachse lediglich bis zum Wert 7 abgebildet. Hierdurch wird die StatusID der Landephase ausgespart.

Aus den Daten kann eine *ground truth* für die Position der z-Achse entnommen werden. Diese Daten stammen von dem auf den Untergrund gerichteten Ultraschall-Sensor. Eine Genaue Abschätzung der Korrektheit dieser Daten lässt sich nicht aus Vergleichsdaten belegen. Die Genauigkeit des Untraschall-Sensors kann jedoch als höher angenommen werden als die der berechneten Pose.

8.1.2 Verlauf des Fluges

Die Flüge in Abbildung 22 (Seite 43) folgen jeweils dem gleichen Scheman:

1. Der Quadrokopter befindet sich in StatusID 2 (*Landed*)

2. Die Methode `reset()` der Klasse `IMUable` wird aufgerufen.

Dies geschieht mit Initialisierung einer Instanz der Klasse `IMUable` oder durch eine User-Eingabe.

3. Die *valid*-Flags der Klasse `StateBuildable` und Klasse `PoseBuildable` werden zurückgesetzt.

4. Die Klasse `StateBuildable` wird in den *Offsetting*-Zustand versetzt.

Aus den eingehenden Daten der *IMU*-Nachrichten wird die dauerhaften Nullpunkt-Abweichung geschätzt.

5. In der Klasse `PoseBuildable` wird die Kalibrierung eingeleitet. Hierzu werden die eingehenden Instanzen der Klasse `IMUState` in einer Liste abgelegt.

6. Aus der User-Eingabe folgt der Start-Befehl. Die Klasse `Statusable` sendet den Befehl an den Quadrokopter die StatusID wechselt auf den Wert 7 (*Goto Fix Point*).

Die Kalibrierung der Klasse `PoseBuildable` wird abgeschlossen: Hierzu werden die in der Liste gespeicherten Werte zur Berechnung einer Pose genutzt. Auf Grund der verstrichenen Zeit kann entsprechend Kapitel 5.3.2 *Signalaufarbeitung* (Seite 19) die verbliebene dauerhaften Nullpunkt-Abweichung geschätzt werden. Dieser Vorgang wird iterativ durchgeführt.

Parallel zur Kalibrierung beginnt der Quadrokopter *ArDrone 2.0* mit dem Start-Vorgang. Dieser sieht vor, die Rotor-Drehzahl zügig zu erhöhen, bis der Quadrokopter abhebt und auf eine vom Hersteller vorgegebene Höhe ansteigt.

Mit dem Erreichen einer empirisch definierten Drehzahl⁹ wird das *valid*-Flag der Klasse `PoseBuilder` gesetzt und ein Rücksetzen der Position ausgelöst.

7. Der Flug endet mit der User-Eingabe des *Landen*-Befehls.

Mit dem Erreichen des Wertes 0 für die Rotor-Drehzahl und die Höhe wird das *valid*-Flag der Klasse `PoseBuilder` zurückgesetzt.

⁹Für die *ArDrone 2.0* ist die Drehzahl in der Klasse `parrotIMU` als Konstante `Magic_TakeoffRotorSpeed` definiert.

8.1.3 Lecks der Datenübertragung

Aus den Daten, aufgetragen in Abbildung 22 (Seite 43), und weiteren Testflügen zeigt sich, dass die Verbindung zwischen dem Host-PC und dem Quadrokopter anfällig für Verbindungsunterbrechungen ist. Während diesen Unterbrechungen werden für einen Zeitraum von etwa einer Sekunde keine Nachrichten übermittelt.

Die entwickelte Software sieht für diese Fälle keine Interpolation der Daten vor. Hieraus entstehen Sprünge in den berechneten Daten und durch die Integration dieser veränderten Daten ein Drift der Position. Hier können als Beispiel Abbildung 24 (Seite 47) und Abbildung 25 (Seite 48) jeweils für die Zeitbereiche um 17 Sekunden und 61 Sekunden herangezogen werden.

8.2 Signalverarbeitung

8.2.1 Ermittlung der dauerhaften Nullpunkt-Abweichung

In Abbildung 15 (Seite 20) wird eine Abweichung der Beschleunigungswerte vom Nullpunkt deutlich. Die eingesetzte Software wirkt dem in der Klasse `StateBuilder` entgegen, indem über eine definierte Anzahl an Eingangswerten der Klasse `IMUState` ein Mittelwert gebildet wird. Um an dieser Stelle aussagekräftigere Ergebnisse zu erhalten, wird zudem die Varianz über den zwischengespeicherten Datensatz gebildet. Der Mittelwert mit der geringste Varianz aus einer definierten Anzahl an Stichproben wird als Nullpunkt-Abweichung angenommen. Hierbei ist zu beachten, dass die Attribute der Klasse `IMUState` separat betrachtet werden.

8.2.2 Glättung der Eingangsdaten

Als Parameter werden für die Berechnung der dauerhaften Nullpunkt-Abweichung (siehe Kapitel 8.2.1 *Ermittlung der dauerhaften Nullpunkt-Abweichung* (Seite 45)) jeweils ein Median-Fenster und ein Mittelwert-Fenster der Größe 1 angesetzt, somit findet keine Veränderung durch diese Filter statt. Es zeigte sich in den Auswertungen, dass ein Median-Fenster bis zur Größe von 15 Werten keinen signifikanten Einfluss auf die berechnete Pose besitzt. Jedoch wirkt sich eine Vergrößerung des Mittelwert-Fensters negativ auf die berechnete Pose aus. Die Abweichung von erwarteten Werten nimmt deutlich zu.

8.2.3 Kalibrierung der Posenberechnung

Aus Simulationen mit der in Kapitel 7 *Implementierung* (Seite 28) beschrieben Software zeigte sich, dass eine einfache Abschätzung der Nullpunkt-Abweichung unach-

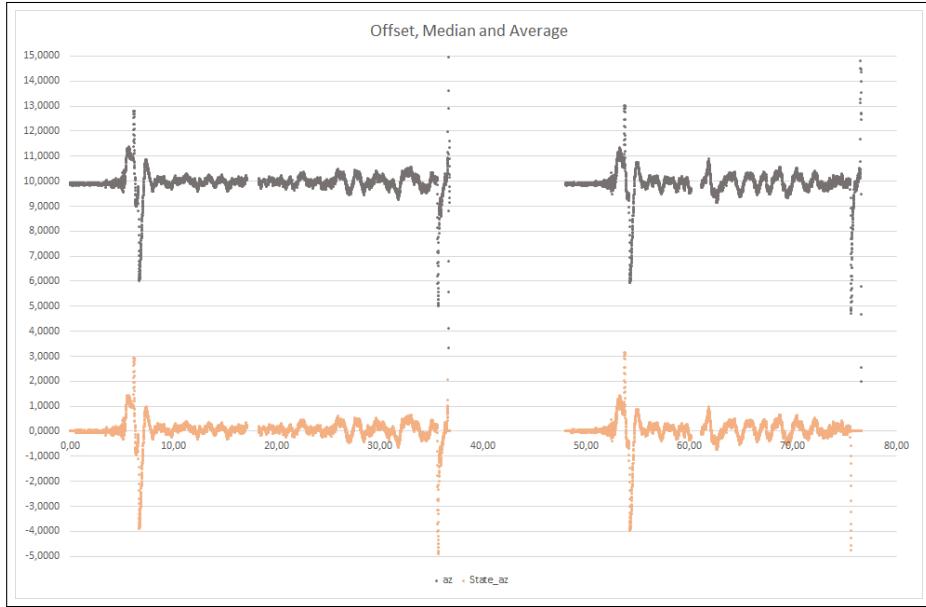


Abbildung 23: TESTFLUG SIGNALVERARBEITUNG: AUFARBEITUNG a_z

Abbildung 23 zeigt die erfolgreiche Ermittlung der dauerhaften Nullpunkt-Abweichung für die Beschleunigungswerte entlang der z-Achse.

Es zeigt sich, dass die Nullpunkt-Abweichung aus den Eingangsdaten der Quadrokopter-IMU berechnet und als Grundlage für eine weitere Aufarbeitung der Daten eingesetzt werden kann.

Kapitel 8.2.1 *Ermittlung der dauerhaften Nullpunkt-Abweichung* (Seite 45) nicht vollständig ausreicht. Der Einfluss bereits kleiner Nullpunkt-Abweichung wurde in Kapitel 5.3.2 *Signalaufarbeitung* (Seite 19) erläutert. Aus Erkenntnissen, aufgezeigt in Abbildung 15 (Seite 20), lässt sich nachfolgender mathematische Zusammenhang ableiten:

$$\Delta P_{Direction} = \frac{\Delta a_{Direction}}{2} * \Delta t^2 \quad (6)$$

MATHEMATISCHE GRUNDLAGE DER *POSEBUILDABLE*-KALIBRIERUNG

Entsprechend dieser Formel berechnet die Klasse **PoseBuilder** eine nach der Aufarbeitung durch die Klasse **StateBuilder** verbleibende Nullpunkt-Abweichung der Beschleunigungswerte. Diese werden zur Berechnung der Pose von den Eingangdaten subtrahiert.

In Abbildung 24 (Seite 47) kann gezeigt werden, dass sich eine zusätzliche Kalibrierung der Beschleunigungsdaten positiv auf die Berechnung der Pose¹⁰ auswirken kann. Die durch die Klasse **PoseBuildable** berechnete Geschwindigkeit entlang der z-Achse entspricht für den zweiten Flug etwa der aus dem gemessenen Höhenverlauf abgeleiteten Geschwindigkeit.

¹⁰In der genannten Abbildung werden die berechneten Geschwindigkeiten aufgetragen.

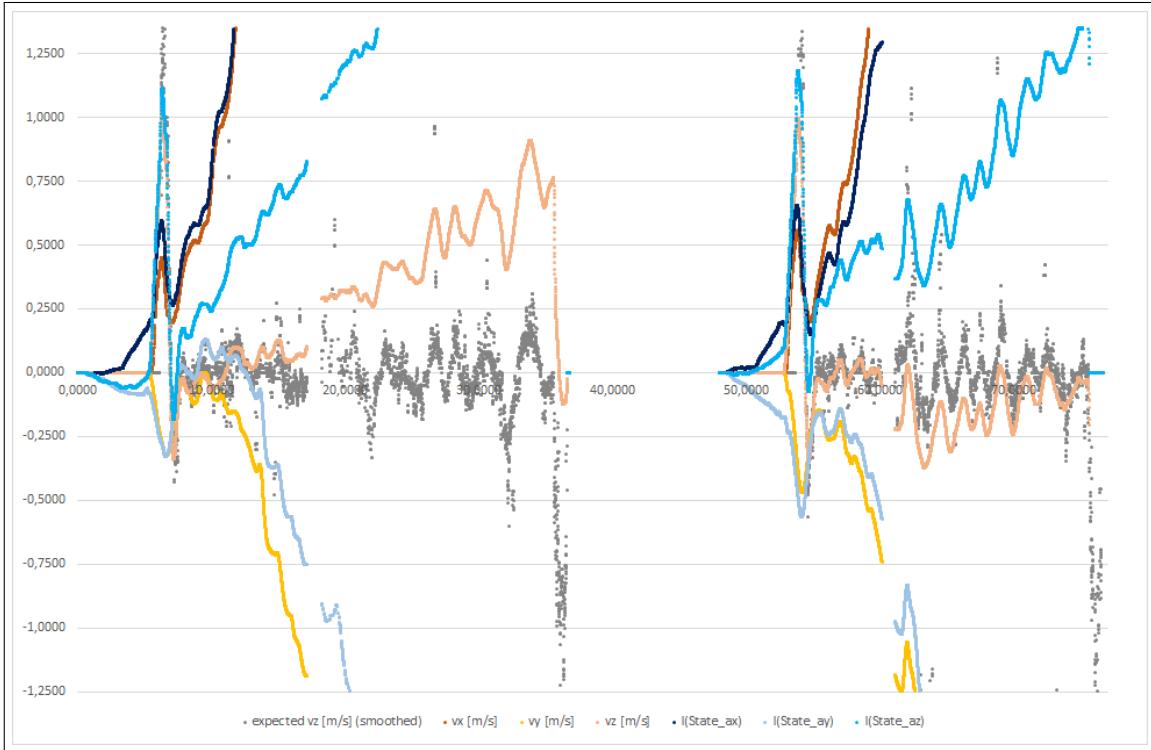


Abbildung 24: TESTFLUG SIGNALVERARBEITUNG: KALIBRIERUNG

An der aufgetragenen Geschwindigkeit vz lässt sich deutlich erkennen, welchen Einfluss die Kalibrierung der Klasse `PoseBuildable` hat. Die berechnete Geschwindigkeit (beige) weicht nur mäßig von der erwarteten Geschwindigkeit (grau) ab.

Anmerkung: Da die IMU-Nachrichten in einer Folgen mit kurzen Zeitschritten eintreffen, wurde die erwartete Geschwindigkeit als gleitender Mittelwert über 10 Werte geglättet.

8.3 Auswertung

Aus der Berechnung der Position in z-Richtung im Vergleich mit den gemessenen Daten (siehe Abbildung 25 (Seite 48)) lässt sich zeigen, dass der Ansatz grundsätzlich korrekt gewählt wurde.

Da die dieser Auswertung zugrundeliegenden Daten mittel der *Hover*-Funktionalität der *ArDrone 2.0* (siehe Kapitel 6.2.2 *Interaktion mittels ROS* (Seite 27)) aufgezeichnet wurden, ist lediglich eine geringe Abweichung in x- und y-Richtung zu erwarten. Diese kann lediglich durch die empirische Erkenntnisse, nicht durch gelegbare Daten, untermauert werden. Im Zuge dieser Projektarbeit konnte keine Begründung für die in Abbildung 25 (Seite 48) ersichtlichen Abweichungen der Position in x- und y-Richtung vom lokalen Nullpunkt gefunden werden. Da auch eine sehr geringe Kalibrierung der Klasse `PoseBuildable` durchgeführt wird (vgl. Abbildung 24 (Seite 47)), ist davon auszugehen, dass die Abweichungen eine tiefergehende Begründung verlangen. Auch der Aufruf der Methode `reset()` für die beiden Ausrichtungen nach Änderungen der StatusID zu dem Wert 3 (*Flying*) oder dem Wert 4 (*Hovering*) konnte den Verlauf der berechneten Position in der horizontalen Ebene nicht zu einem zufriedenstellenden

Ergebnis führen.

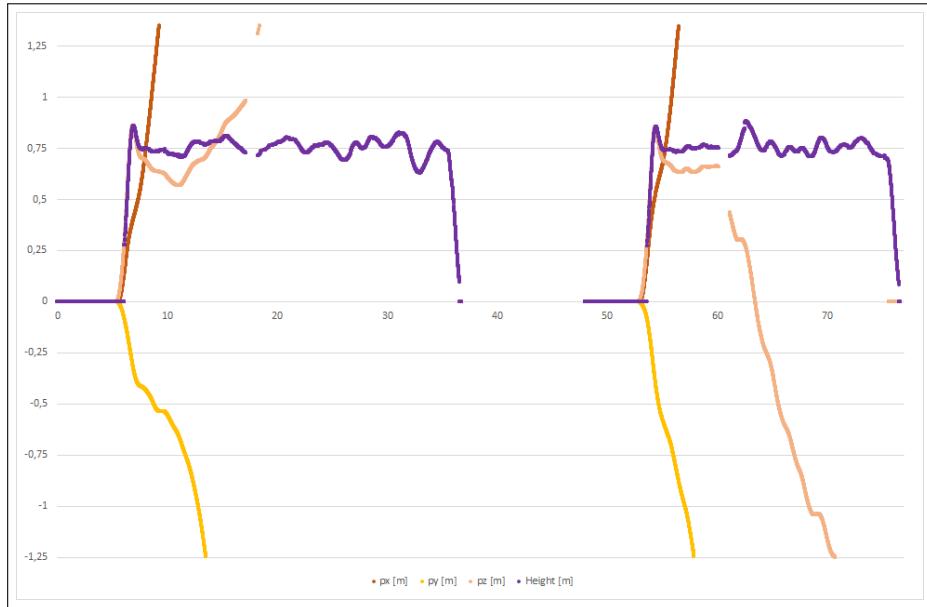


Abbildung 25: TESTFLUG SIGNALVERARBEITUNG: AUFARBEITUNG a_z

Abbildung 25 zeigt die von dem eingesetzten Programm errechneten Position des Quadroopters *ArDrone 2.0* und als Vergleichswert die vom Ultraschall-Sensor gemessenen Abstand zum Untergrund.

Unter der Annahme, dass das Programm zur Berechnung der Pose korrekt implementiert wurde, kann abgeleitet werden, dass die in der *ArDrone 2.0* verbauten Initialsensoren in Verbindung mit der Diskretisierung der gemessenen Werte eine Abweichung der berechneten Pose von der tatsächlichen Pose zu erwarten ist. Diese Abweichung kann als Unsicherheit beziehungsweise Wahrscheinlichkeitsverteilung in die berechnete Pose aufgenommen werden.

9 Fazit und Ausblick

Im Zuge der Analyse der Flugdaten (siehe Kapitel 8.1 *Analyse realer Flugdaten* (Seite 42)) in Kombination mit der Rekonstruktion eines idealisierten Flugverlaufs (siehe Kapitel 5.3 *Erzeugung von Posen aus Beschleunigungsdaten* (Seite 14)) zeigte sich, dass sich eine Rekonstruktion der Pose aus Beschleunigungsdaten als schwierig herausstellt. Hieraus folgt, dass die Problemstellung (siehe Kapitel 2 *Problemstellung* (Seite 2)) nicht beziehungsweise nicht vollständig gelöst werden konnte.

Nach erfolgreicher Berechnung der Pose kann die Implementierung des Controllers getestet und die Regelparameter bestimmt und optimiert werden.

9.1 Erweiterungen

Dieses Kapitel soll beschreiben, welche weiteren Sensoren eingesetzt werden können, um die Genauigkeit berechneten der Pose zu erhöhen.

Da es sich bei den möglichen Erweiterungen um eine Fülle von Varianten handelt, sollen diese jeweils nur knapp beschrieben oder lediglich namentlich genannt werden.

9.1.1 Weitere Ansätze der Signalverarbeitung / Pose-Berechnung

Geänderte Integral-Berechnung

Das Integral wird in Klasse `Controller_I` in Form eines riemannschen Integrals berechnet, die eingehenden Daten jeweils mit der zuvor vergangenen Zeit multipliziert werden. Bei riemannschen Integralen wird hier der Funktionswert mit der nachfolgenden Höhe (Hier: Zeit) verrechnet.

Ein Ansatz zur besseren Berechnung wäre somit, andere Varianten der Integral-Berechnung heranzuziehen.

Anmerkung: Aus Berechnung mit einem Mittelwert-Filter, welcher zwei Werte verrechnet, ergab sich, dass ein Trapez-Integral nicht geeignet erscheint.

9.1.2 Interne Sensoren

Höherwertige Initialsensoren

Sollten etwaige Berechnungsfehler oder Problematiken im strukturellen Ablauf eines Flugen -bezogen auf die gesetzten Flags- behoben sein, können höherwertige Sensoren als weitere Verbesserungsmöglichkeit dienen. Hierbei müssen Befestigung am Quadrokopter, Spannungsversorgung dieser IMU, sowie der Datenaustausch mit dem Host-Rechner durchdacht werden.

Magnetometer

Im Sinne einer Erweiterung könnte ein Verfahren entsprechen [4] eingesetzt werden. Ein tiefergehendes Konzept in Bezug auf die Fortführung dieser Projektarbeit wird an dieser Stelle nicht genannt.

9.1.3 Externe Sensoren

Boden- und Frontkamera

Mit den Kameras können markante Objekte und hierdurch Veränderungen der Position erfasst werden.

9.1.4 Externe Sensoren und Mapping

Mittels Mapping-Algorithmen kann die lokale oder globale Pose des Quadroopters bestimmt werden. Ist die Karte der Umgebung bereits bekannt, kann der Quadroopter mit geeigneten Vorgehensweisen (zum Beispiel Monte Carlo Algorithmus) die globale Pose selbstständig bestimmen.

Vorwärts-Berechnung

Sind bereits Posen von markanten Objekten bekannt, können diese zur genaueren Schätzung der Pose des Quadroopters herangezogen werden. Hierbei sind jeweils die Unsicherheiten der bekannten Objekte zu aktualisieren, sofern deren Posen genauer geschätzt werden können.

Rückwärts-Berechnung bei Ringschluss

Wird auf dem Weg eine bekanntes Objekt identifiziert, kann von diesem Objekt aus die vergangenen Posen zurück gerechnet werden und gegebenenfalls die Pose anderer bekannter Objekte genauer geschätzt werden. Somit wird das Mapping deutlich verbessert. Als Anhaltspunkt soll hier [5] dienen, worin diese Technik für ein verbessertes Mapping in Bezug auf 3D-Laser-Messungen in Mienen eingesetzt wurde.

Literaturverzeichnis

- [1] Pozo D., Romero L., Rosales J., Quadcopter stabilization by using PID controllers, veröffentlicht 2014
- [2] Fresk E., Nikolakopoulos G., Full Quaternion Based Attitude Control for a Quadrotor, veröffentlicht 19.07.2013
- [3] Morton L., Baille L., Ramirez-Iniguez R., Pose Calibrations for Inertial Sensors in Rehabilitation Applications, veröffentlicht 2013
- [4] Yang Z., et al., Hand-Finger Pose Estimation Using Inertial Sensors, Magnetic Sensors and a Magnet, veröffentlicht 2021
- [5] Bosse M., Zlot R., Efficient Large-scale Three-dimensional Mobile Mapping for Underground Mines, veröffentlicht 2013
- [6] Heinrich B. (Hrsg.), et al., Kaspers/KOfner Messen - Steuern - Regeln, 8., überarbeitete und ergänzte Auflage. Auflage veröffentlicht 2009 ISBN 978-3-8348-0006-0
- [7] Wendemuth A., Grundlagen der digitalen Signalverarbeitung, veröffentlicht 2005 ISBN 3-540-21885-8
- [8] Tukey J., Exploratory Data Analysis, veröffentlicht 1977 ISBN 0-201-07616-0
- [9] Leishmann G., Principles of Helicopter Aerodynamics, veröffentlicht 2006 ISBN 0-521-85860-7
- [10] Internationales Elektrotechnisches Wörterbuch – Teil 351: Leittechnik (IEC 60050-351:2006), veröffentlicht 06/2009
- [11] Richter M., Vorlesung Robotik 1, Einheit 4, DHBW Karlsruhe, Kurs TINF19B1, veröffentlicht 13.03.2014, verändert 08.03.2022 **BESCHREIBUNG! Published Date!**
- [12] Richter M., Vorlesung Robotik 1, Einheit 5, DHBW Karlsruhe, Kurs TINF19B1, veröffentlicht 13.03.2014, verändert 22.03.2022
- [13] Prof. Dr. Strand M., Vorlesung Robotik 2, Einheit 8 - Positionsschätzung, DHBW

Karlsruhe, Kurs TINF19B1,
veröffentlicht 27.04.2022

- [14] Mey R., Reinhard Mey Textsammlung 14.Auflage,
online, <https://www.reinhard-mey.de/texte-fuer-alle/>
veröffentlicht 13.12.2017, abgefragt 08.02.2022
- [15] Ikarus,
online, <https://griechische-mythologie.fandom.com/wiki/Ikarus>
veröffentlicht -unbekannt-, abgefragt 11.05.2022
- [16] Frauenhofer IML, ZF-ZUKUNFTSSTUDIE 2016 - Die letzte Meile,
online, <https://publica-rest.fraunhofer.de/server/api/core/bitstreams/6e295b15-7a64-4e19-8895-30f7c650307e/content>
veröffentlicht 24.11.2016, verändert 19.12.2016, abgefragt 11.05.2022
- [17] ellwangen2010, Ballon steuern?,
online, <https://www.ballonfahrten.com/ballon-steuern/>
veröffentlicht 20.02.2010, abgefragt 07.11.2021
- [18] D-Glied,
online, http://testcon.info/FB_DE_D-Glied.html
veröffentlicht -unbekannt-, abgefragt 13.04.2022
- [19] Stabilität von Regelkreisen (Frequenzkennlinienverfahren),
online, <https://homepages.thm.de/hg13555/Datenbank/aat/index.php/grundlagen-regelungstechnik/45-stabilitaet-von-regelkreisen-frequenzkennlinienverfahren.html>
veröffentlicht -unbekannt-, abgefragt 15.04.2022
- [20] Die Krux mit der Totzeit,
online, <https://www.technik-und-wissen.ch/krux-mit-totzeit-regelungstechnik-mit-smith-praediktor.html>
veröffentlicht -unbekannt-, abgefragt 13.05.2022
- [21] Jentsch M., Hochpass-Filter & Tiefpass-Filter programmieren,
online, <https://www.jentsch.io/hochpass-filter-tiefpass-filter-programmieren/>
veröffentlicht 12.02.2022, abgefragt 13.05.2022
- [22] ROS - Robot Operating System,
online, <https://www.ros.org>
veröffentlicht -unbekannt-, abgefragt 28.11.2021
- [23] ROS Indigo Igloo,

online, <http://wiki.ros.org/indigo>
veröffentlicht -unbekannt-, verändert 08.01.2018, abgefragt 16.03.2022

[24] How quadcopters work & fly: An intro to multirotors,
online, <https://www.droneybee.com/how-quadcopters-work/>
veröffentlicht -unbekannt-, verändert 20.11.2017, abgefragt 01.04.2022

[25] Clover 4.2 assembly,
online, https://clover.coex.tech/en/assemble_4_2.html
veröffentlicht -unbekannt-, abgefragt 09.05.2022

[26] Initial setup,
online, <https://clover.coex.tech/en/setup.html>
veröffentlicht -unbekannt-, abgefragt 09.05.2022

[27] MAVROS Offboard control example,
online, https://docs.px4.io/master/en/ros/mavros_offboard.html
veröffentlicht -unbekannt-, verändert 02.02.2021, abgefragt 16.03.2022

[28] AR.Drone Developer Guide,
Kapitel *AR.Drone 2.0 Overview*, Seite 5 ff. online, <https://jpchanson.github.io/ARdrone/ParrotDevGuide.pdf>
veröffentlicht 21.05.2012, abgefragt 17.03.2022

[29] Sending Commands to AR-Drone,
online, <https://ardrone-autonomy.readthedocs.io/en/latest/commands.html>
veröffentlicht -unbekannt-, abgefragt 11.05.2022

Anmerkung: Wird hier ein Veröffentlichungsdatum als “-unbekannt-“ markiert, so konnte diese Angabe weder auf der entsprechenden Webseite, noch in deren Quelltext ausfindig gemacht werden.