

Thema

Erweiterung bestehender Drohnen um eine Autonomflugfähigkeit

Studienarbeit T3100

des Studienganges Eletrotechnik

an der Dualen Hochschule Baden-Württemberg, Stuttgart

von

Markus Rein

Abgabedatum: 22.01.2023

Bearbeitungszeitraum	21.10.22 – 22.01.2023
Matrikelnummer, Kurs	6983030, TEL20GR5
Dualer Partner	Infineon Technologies, Neubiberg
Gutachter*in der Dualen Hochschule	Prof. Dr.-Ing. Johannes Moosheimer

Erklärung

Ich versichere hiermit, dass ich meine Projektarbeit mit dem Thema: „Analyse der Anwendungsmöglichkeiten bei der Vermittlung von Datenpaketen zwischen Mikrocontrollern und Terminalcomputern“ 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.

.....
Ort

.....
Datum

.....
Unterschrift

Kurzfassung

..... Short summary of the thesis ...

Inhaltsverzeichnis

1	Problemstellung und geplantes Vorgehen
2	Grundlagen im Themenbereich autonomer Drohnenflug
2.1	Peripherie an der Drohne
2.2	Robot Operating System
3	Verfügbare Technologien
3.1	Vorraussetzungen aus dem ersten Projektteil
3.2	Vorraussetzungen aus dem zweiten Projektteil
3.3	Marktreife Anwendungen
3.4	Pixhawk Software
4	Aufarbeitung bestehendes Drohnenprojekt
4.1	Inbetriebnahme
4.2	Einbindung von Funktionalität mittels Docker-Containern
4.3	Experiment: Positionsbestimmung der Drohne mit GPS
4.4	Ultraschallsensoren am RPI
5	Einführung Navigationsalgorithmus
5.1	Einrichtung der Software
5.2	Verarbeitung von MAVLink-Nachrichten mit ROS
5.3	Obstalce Avoidance

Literaturverzeichnis

Abbildungsverzeichnis

2.1	Einteilung Automationsklassen von Unmanned Aerial Vehicle (UAV)
3.1	Systemübersicht mit allen physischen Bauteilen[WH22a, Kapitel 4.3]
3.2	Systemübersicht in Hardware und im Simulator
4.1	Konfiguration und Verwendung von <code>mavlink-routerd</code>
4.2	Vergleich Bewegungsprofil verschiedener Geräte. GPS-Modul der Drohne in Grün, zwei Smartphones in Blau und Orange.
4.3	Vergleich Höhenprofil verschiedener Geräte bei Bewegung bergab.
4.4	Demonstration der Ultraschallsensoren
4.5	Messung mit einem Ultraschallsensor. Zu sehen: oben ungefilterte Messwerte; Median Filter 1 pflegt Sprünge in Messdaten direkt in Filter-Array ein aber berechnet Durchschnitt; Median Filter 2 berechnet erst Durchschnittswert und pflegt diesen in ein Filter-Array ein; Moving Average berechnet immer den Durchschnittswert der vergangenen Messwerte.
5.1	Aufgliederung Bestandteile der Softwareentwicklung
5.2	Simulation der Drohne unter Windows
5.3	Graphische Anwendung <i>rqt</i> aus ROS unter Windows
5.4	First-Person View, Kamera der simulierten Drohne
5.5	<i>topic-explorer</i> zeigt Daten zur Punktwolke

Tabellenverzeichnis

2.1	Vergleich ROS und ROS2
4.1	Ergebnisse aus [WH22b, Kapitel 6.13], größerer Zahlenwert steht für höhere Übertragungsrate

Abkürzungsverzeichnis

GCS Ground Control Station.

GPS Global Positioning System.

GUI Graphical User Interface.

HIL Hardware in the Loop.

IMU inertiale Messeinheit.

LBA Luftfahrt-Bundesamt.

MAVLink *MAVLink*.

Pixhawk® 4 Microcontroller, mit Software und Peripherie zur Flugsteuerung.

RAM Arbeitsspeicher.

ROS Robot Operating System, Framework mit Schnittstellen für Sensor/Aktor-Verknüpfungen.

RPI Raspberry Pi (Einplatinencomputer).

TCP Transmission Control Protocol.

UAS Unmanned Aircraft System.

UAV Unmanned Aerial Vehicle.

UDP User Datagram Protocol.

1 Problemstellung und geplantes Vorgehen

Ziel dieser Arbeit ist es, das Projekt „Erweiterung einer bestehenden Drohne um eine Autonomflugfähigkeit“ von Harald Wirth und Dominik Helfenstein, siehe [WH22a][WH22b], aufzuarbeiten und fortzuführen. Dazu steht eine Modelldrohne „Holybro S500“ zur Verfügung. Diese wurde zusätzlich mit einem Raspberry Pi (Einplatinencomputer) (RPI) ausgestattet, der eine Kommunikation mit der Drohne ermöglicht und zusätzliche Rechenaufgaben übernehmen kann. Dieser Aufbau wird verwendet, um Objekte in der Flugbahn der Drohne zu erkennen und auszuweichen. Bisher können der im Flug befindlichen Drohne über WLAN GPS-Zielkoordinaten zugespielt werden. Anschließend fliegt sie in Richtung des Ziels und verwendet währenddessen Ultraschall, um Objekte in nächster Umgebung zu detektieren. Die Umsetzung was anschließend passieren soll, ist allerdings noch nicht abgeschlossen. Es herrscht folgender Zustand:

- In den Tests flog die Drohne in Richtung eines ebenen Hindernisses. Sie hielt in sicherer Entfernung an und wurde automatisch gelandet. Bei weiteren Tests an natürlichen Strukturen (Büsche) konnten diese nicht erkannt werden und die Drohne musste manuell abgefangen werden.
- Für weitere Flüge soll das sog. Magneten-Prinzip zur Anwendung kommen (noch nicht implementiert). Bei diesem wird eine Kursänderung um sich im Weg befindliche Hindernisse durchgeführt. [schänes BILD zum Verständnis] Wird das Ziel erreicht, landet die Drohne an gegebenem Punkt. Für produktive Anwendungszwecke reicht dieses aber nicht aus denn eine Zielführung ist nicht garantiert.

Um das Projekt erfolgreich abzuschließen, wird eine verbesserte Hinderniserkennung und Routenplanung entwickelt. Dieses Projekt sieht vor, dass die Drohne sowohl statische als auch dynamische Hindernisse erkennt und als Karte dokumentiert. Somit wird Navigation durch unbekanntes Terrain ermöglicht. Weiterhin soll die Drohne gezielt Markierungen ansteuern können.

Im ersten Teil dieser Arbeit wird das bestehende Projekt tiefgründig analysiert. Grundlegende Konzepte werden aufgegriffen und erweitert. Bestandteil hiervon ist das erfolgreiche Anwenden der Tests und Funktionalität des ersten Projektes. Dazu wird die Software angepasst und auch „ROS“, was Vorgesehen war aber nie zum Einsatz kam, verwendet um eine Basis für weitere Entwicklungen zu schaffen.

Ein weiterer Meilenstein ist es, die Sicherheit für weiteres Vorgehen zu schaffen. Sämtliche

1 Problemstellung und geplantes Vorgehen

Software soll ab diesem Punkt im Simulator getestet werden, um Ausfälle und Schäden zu vermeiden. Auch soll die Software zur Hindernis- und Zielerkennung im Simulator getestet werden.

Der zweite Teil sieht vor, ein Kamerasystem zu installieren, um Hindernisse und Ziele zu erkennen. Dieses kommuniziert mittels ROS mit der Drohne selbst und steuert diese so. Die Software der Drohne dokumentiert eine notwendige grundlegende Vorgehensweise und wird als eine Basis für eigene Entwicklungen benutzt.

Für die Erkennung von Hindernissen ist ein schneller (echtzeitfähiger), zuverlässiger Algorithmus notwendig. Es sind umfangreiche Tests notwendig.

Zuletzt wird der Navigationsalgorithmus der Drohne erweitert und angepasst.

2 Grundlagen im Themenbereich autonomer Drohnenflug

Ein UAS besteht aus einem unbemannten Luftfahrzeug und dessen Ausrüstung. Die Steuerung kann entweder von einem Menschen oder von einem integrierten oder ausgelagerten Computer durchgeführt werden. Dadurch kann eine Drohne auch ein teil- oder vollautonomes Luftfahrzeug darstellen.

(Bundesministerium für Digitales und Verkehr¹)

Der englische Fachbegriff für Drohnen lautet „Unmanned Aircraft System (UAS)“. Er stammt vorrangig aus dem militärischen Bereich und beschreibt den Flugkörper selbst und etwaige Zusatzgeräte (Waffen). Heutzutage wird vermehrt der Begriff „UAV“ verwendet, so auch fortwährend in dieser Arbeit. Aus dem Modellbau sind UAV in Form von fernsteuerbaren Flugzeugen, Helikoptern und Multicoptern bekannt. Drohne wird häufig als Synonym für letztere genutzt. Eine häufige Anwendung spielt dabei der autonome Flug, bei dem Anweisungen vom Bediener kommen, die automatisiert durchgeführt werden. So kann eine Drohne auch zu gewerblichem Einsatz kommen, bspw. in den folgenden Bereichen: Landwirtschaft, Wettervorhersage, Vermessung, Bevölkerungsschutz und Transport.

Zur Einstufung des automatisierten Verhaltens von Drohnen definiert die European Cockpit Association AISBL 6 „Level“ (Klassen), veranschaulicht in Grafik 2.1. Verfügbare UAV implementieren bereits automatische Funktionen der unteren Level, wie bspw. Regelung zur Flugstabilität und dem Halten einer Flugrichtung. Interessant sind die oberen Level, beginnend bei Level 3:

Level 3 - Bedingte Automation: Das UAV kann bestimmte Teilaufgaben in definierter Umgebung selbsttätig durchführen. Ein Pilot muss immer anwesend sein, um im Störungsfall einzugreifen.

Level 4 - Hohe Automation: Die gesamte Flugsteuerung und Ausfallsicherung wird eigenständig durchgeführt. Kein Pilot wird benötigt, jedoch muss das Verhalten des UAV überwacht werden.

Level 5 - Vollständige Automation (Autonomie): Eigenständige Flugplanung von Start über Flugstrecke bis Landung. Mit derzeitigen Mitteln nicht realisierbar².

¹<https://www.dipul.de/homepage/de/informationen/allgemeines/was-ist-eine-drohne>

²siehe <https://twitter.com/elonmusk/status/1411280212470366213>

2 Grundlagen im Themenbereich autonomer Drohnenflug

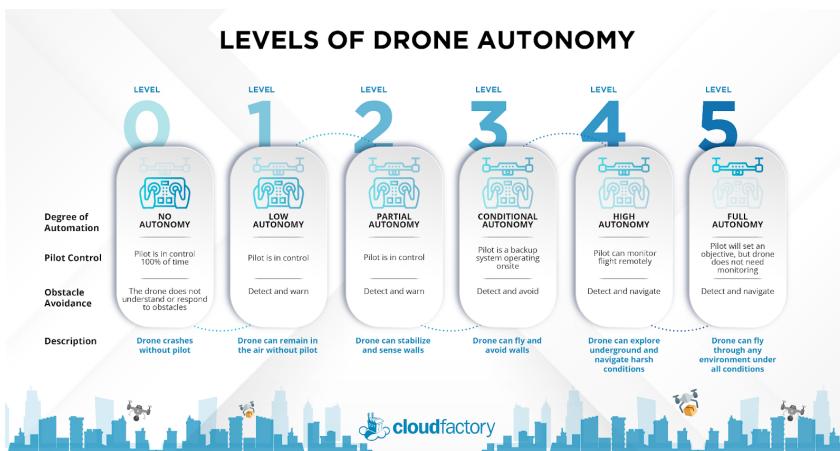


Abbildung 2.1: Einteilung Automationsklassen von UAV: Vom niedrigsten Level (0) zum höchsten Level (5) steigt der Automationsgrad, von [Clo].

Das Luftfahrt-Bundesamt (LBA) erlässt zudem Regelungen, wonach ein autonomer Flug , [...] derzeit nicht zulässig [ist]; Fernpiloten müssen jederzeit eingreifen können. [...] '[Ope].

Der bisherige Stand des Projektes steht im Übergang von Automationsklasse 3 auf Klasse 4. Allerdings fehlen generalisierte Anwendungsfälle (später gezeigt) sodass die Drohne noch nicht wie gewünscht, Hindernissen ausweicht (das bisherige Verhalten beschränkt sich auf sicheres Landen beim Erkennen von Hindernissen).

Wie in Kapitel 3 erläutert, wird die Modelldrohne Holybro S500 verwendet. Um diese Fliegen zu dürfen, bestehen in Deutschland mehrere Voraussetzungen³:

- Drohnenführerschein: Unterkategorie A2 (UAS bis 4kg, 5m Abstand zu Menschen, kein Überfliegen von Menschenansammlungen, nur auf Sichtweite fliegen)
- Höhenmesser: die maximale Flughöhe von 120m darf nicht überschritten werden
- Fernidentifizierung: Pflicht zum Mitführen eines ADS-B Transmitters⁴
- Haftpflichtversicherung: Pflicht zum Versichern und Kennzeichnen der Drohne
- „Filmen, fotografieren oder das Anfertigen von Tonaufnahmen ist verboten, wenn: [...] die Aufnahmen für Gesichtserkennung oder andere automatisierte Prozesse verwendet werden“

Somit der Einsatz der Drohne im öffentlichen Raum streng untersagt. Tests beschränken sich auf Flüge nahe am Boden, fernab der Zivilisation.

³<https://lba-openuav.de/onlinekurs/lehrmaterial>

⁴https://de.wikipedia.org/wiki/Automatic_Dependent_Surveillance

2.1 Peripherie an der Drohne

Im Flugcontroller integriert sind mehrere Sensoren (Beschleunigung, Gyroskop, Magnetometer, Barometer). Neben diesen besitzt das Modell einen dedizierten Global Positioning System (GPS)-Empfänger. Nachfolgend ist dargestellt, wie diese Sensoren ausgewertet werden können.

2.1.1 Navigation mittels GPS

Die Navigation der Drohne erfolgt vorrangig über GPS. Mit solchem System können die aktuelle Position und Geschwindigkeit festgestellt werden. Für ein reales System muss ein eventueller Ausfall der GPS-basierter Navigation in Betracht gezogen werden, denn GPS setzt klare Sicht zum Himmel und eine Verbindung zu mindestens 4 Satelliten voraus. [Quelle]

Fällt das GPS aus, ist die Drohne jedoch noch nicht völlig blind. Beschleunigungssensor, Gyroskope und Kompass bilden eine inertiale Messeinheit (IMU), mit deren Daten mittels eines *Extended Kalman Filter* die ungefähre Position berechnet werden kann. Das Ergebnis kann durch das Überlagern der Messwerte mehrerer Sensoren verbessert werden, bspw. indem sich an einem Kamerabild mit den bereits bekannten Abständen zu Objekten orientiert wird.

Ein weiteres Problem ist die totale Abhängigkeit von GPS in Bezug auf Zielfindung. Selbst wenn die Drohne direkt über dem Zielpunkt fliegt, aber GPS ausfällt, kann sie keine Erfolg verzeichnen. Für die Flugplanung wäre folglich ein zweiter Sensor von notwendig, der ohne GPS zum Zielpunkt navigieren kann. Für das Aufwinden des Ziels stehen mehrere Technologien zur Verfügung:

Ortung eines Sender (Ultraschall oder Infrarot) welcher durchgängig Signale sendet. Die Drohne bewegt sich dann immer auf den Sender zu.

Erkennung einer Struktur oder Bildes (Marker) mittels Kamera.

2.1.2 Hinderniserkennung durch Ultraschallsensoren

Mit einem Ultraschallsensor kann der Abstand zu Objekten gemessen werden.

Das Prinzip der Ultraschallortung besteht aus 3 Schritten:

1. Aussenden des Messimpulses. Daraufhin generiert der Sensor Ultraschall-Impulse.
2. Empfangen der Messimpulse. Der Sensor setzt ein Signal wenn ein Echo empfangen wird.
3. Auswerten der Signallaufzeit. Die Impulse bewegen sich mit Schallgeschwindigkeit im Medium zum Messobjekt und zurück.

Verwendet werden Ultraschallsensoren an der Drohne, welche Messimpulse in die Umgebung aussenden. Für eine exakte Auswertung der Signallaufzeit müssten Umgebungstemperatur, Luftdruck und Luftfeuchte bekannt sein. Den größten Einfluss spielt dabei die Temperatur. Druck und Feuchte wirken nur mit jeweils maximal auf das Ergebnis ein 5% bzw. 2%. Die Formel zur Bestimmung der Entfernung eines Objektes lautet ist gegeben in Formel 2.2. Benötigt wird die Schallgeschwindigkeit c_{20} bei 20°C und der Temperaturkoeffizient α_{20} [Gru22, Seite 152].

$$s = \frac{1}{2}c \cdot t \quad (2.1)$$

$$= \frac{1}{2}c_{20}(1 + \alpha_{20}(\vartheta - 20^\circ\text{C}))t \quad (2.2)$$

Stereokamera Verwendet mehrere Kameras aus parallelverschobenen Bildern Tiefeninformationen zu gewinnen. also Abstand zu Punkten im Bild zu erkennen.

Optical Flow Auswertung der Bewegung von Objekten in Videoabläufen. Kann schlecht zwischen Bewegung der Kamera und Bewegung der Objekte unterscheiden. Ungenau, da Kameras immer eine Verzerrung besitzen.

2.2 Robot Operating System

Das ROS wird hier kurz angeschnitten, mehr Informationen in [WH22b, Kapitel 2.2]. Das wichtigste ist in Tabelle 2.1 aufgeführt. Außerdem ist ROS stark an Linux Ubuntu geknüpft und läuft kaum auf anderen Betriebssystemen. Das Vorgängerprojekt sah vor, ROS2 zu verwenden [WH22b, Kapitel 6.6]. In [Dro23] (siehe Abschnitt 3.4.2) wird Linux Ubuntu 20.04 in Verbindung mit *ROS Noetic* (Version 1) verwendet. Um kompatibel zu bestehender Software zu bleiben wird für dieses Projekt vorerst ein Container mit *ROS Noetic* (Version 1) verwendet. Im DockerHub⁵ stehen diverse Images zur Verfügung. Aufgrund weiterer Vorhaben im Bereich Bildverarbeitung wird das Image „noetic-perception-focal“ verwendet.

Tabelle 2.1: Vergleich ROS und ROS2

	ROS	ROS2
Entwicklungsstand	Abgeschlossen, nur noch Sicherheitsupdates	Aktive Entwicklung
Verfügbarkeit	64-bit PC, 32- und 64-bit ARM	64-bit PC und 64-bit ARM

⁵https://hub.docker.com/_/ros/ [Drob]

3 Verfügbare Technologien

Im Kapitel wird auf verschiedene Arbeiten zum Thema autonome Flugplanung und Hinderniserkennung eingegangen. Zuerst werden die beiden Arbeiten des vorhergehenden Jahrgangs betrachtet. Weiterhin wurden bereits Forschungen zum Thema durchgeführt, welche als Ausgangspunkt für diese Arbeit genutzt werden. Außerdem werden Algorithmen zur kamera-basierten Bilderkennung eingeführt, da derartige von verkauffertigen Produkten verwendet werden.

3.1 Voraussetzungen aus dem ersten Projektteil

Im ersten Projektteil, siehe [WH22a], wurden die Grundlagen zum autonomen Drohnenflug erarbeitet. Es wurde der Markt an verfügbaren Drohnen analysiert, um die best-geeignetste Drohne herauszusuchen[WH22a, Kapitel 3]. Zur Entwicklung ausgewählt wurde das Modell „Holybro S500“[WH22a, Kapitel 4.3], welches in dieser Arbeit weiterhin verwendet wird. Der Quadcopter hat folgende Eigenschaften:

- Gewicht: 935g, Gesamtgewicht mit Akku, Sensoren und Bordcomputer: ?
- Traglast: 1,8kg
- Akku: Typ - LiPo; Nennspannung - 14,8V; Kapazität - 5000mAh; Capacity-Racting: 10C; Gewicht - ?
- Flugcontroller: Pixhawk® 4
- Sensoren: Gyroskop, Beschleunigung, Kompass, Barometer, GPS

Um den autonomen Flug planen zu können, wären mögliche Fluggeschwindigkeit und Flugdauer notwendig. Dazu ist keine Angabe vom vorhergehenden Projekt bekannt, die Werte werden in diesem Projekt ermittelt.

Der gesamte Aufbau des Systems ist in Abbildung 3.1 gezeigt. Die Modelldrohne und ihre zusätzliche Peripherie gliedert sich folgendermaßen:

Flugcontroller: Kontrolle der originalen Drohne

RPI: Bordcomputer, kommuniziert direkt mit dem Flugcontroller und den Ultraschallsensoren; stellt WLAN-Netzwerk bereit

Ground Control Station (GCS): ¹ Laptop/Smartphone der Flug steuert und Drohne parametriert; zum Einsatz kommt die Software QGroundControl, siehe [WH22a, Kapitel 4.3.5]; der Begriff GCS wird fortwährend als Synonym QGroundControl verwendet

Ultraschallsensoren: 4 Stück; Erfassen von Hindernissen in der Umgebung; ausgerichtet nach vorn/oben/unten

Kamera: verbunden mit RPI; nicht für dieses Projekt bereitgestellt

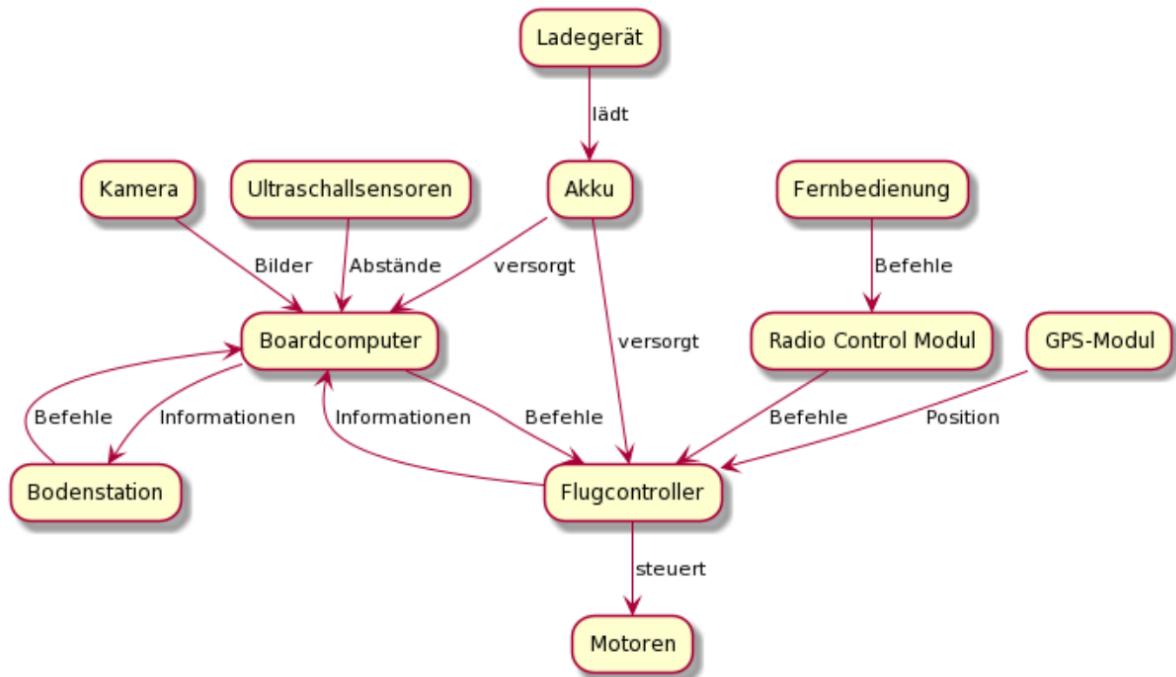


Abbildung 3.1: Systemübersicht mit allen physischen Bauteilen[WH22a, Kapitel 4.3]

Mit den Sensoren können Objekte in der Flugbahn detektiert werden, solange die Bewegung entlang einer der Raumrichtungen: „Oben“, „Unten“ oder „Vorwärts“ stattfindet. Zusammengesetzte Bewegungen dürfen nicht durchgeführt werden, auch darf sich die Drohne nicht seitwärts oder rückwärts bewegen.

Für die Wahl der Sensoren standen ökonomische und technische Gründe im Vordergrund: anderweitige Sensoren sind entweder teuer oder verbrauchen sehr viel Strom, was die Flugdauer der Drohne drastisch senken würde[WH22a, Kapitel 4.3.8].

Die Drohne wurde mithilfe der Software QGroundControl eingerichtet[WH22a, Kapitel 4.3.5] und auf Flugfähigkeit getestet[WH22a, Kapitel 4.3.7]. Zum Einsatz kam eine RC-Fernsteuerung,

¹Bodenstation

welche nicht diesem Projekt nicht zur Verfügung steht. Als Alternative Steuerung kann entweder nur per GPS Navigation geflogen werden, oder ein virtueller Joystick von QGroundControl verwendet werden.

3.2 Vorraussetzungen aus dem zweiten Projektteil

Im zweiten Projektteil [WH22b] die Software zum autonomen Flug eingeführt. Dabei traten folgende Probleme auf: das „Meta-Betriebssystem“ ROS, eingesetzt zum Auslesen der Ultraschallsensoren und Steuerung des Flugcontroller, ließ sich nicht auf dem RPI installieren. Es wurde die Software „Docker“ eingeführt, um ROS per Virtualisierung nutzen zu können[WH22b, Kapitel 6.5].

Außerdem wurde die Kamera[WH22b, Kapitel 6.8] ausgelesen, die Ultraschallsensoren getestet[WH22b, Kapitel 6.9], und die Navigation mittels GPS verifiziert[WH22b, Kapitel 6.10].

Mit dem bestehenden System wurden Flugtests durchgeführt. Alle Tests wurden, mit einer Ausnahme, erfolgreich abgeschlossen. Kaum möglich ist das erkennen nicht-flächiger Hindernisse mit Ultraschallsensoren[WH22b, Kapitel 6.10]. Die gesamte bereitgestellte Software wird in Abschnitt 4.1 nachvollzogen, überprüft, verbessert und angepasst.

Als Bestandteil der Arbeit wurde ein Simulator, um „ein[en] Algorithmus zur Hindernisumgehung, der auf die reale Hardware übertragen werden soll [zu entwickeln]. Dieser Algorithmus soll in der Simulation implementiert und getestet werden.“[WH22b, Kapitel 8]. Somit wird die Entwicklung und das Testen der Funktionalität vereint und kann fließend erfolgen. Der Ansatz wird für diese Arbeit übernommen.

3.3 Marktreife Anwendungen

Der Drohnenmarkt gehört so gut wie der Firma „DJI“ allein, nach [Jür]. Die Drohnen der Firma verwenden zur Hinderniserkennung verschiedene SLAM-Algorithmen[Cio21], siehe ???. Dabei kommen immer mehrere Sensoren zum Einsatz deren Messdaten überlagert werden. Nach [Cio21] können folgende Sensoren verwendet werden:

Ultraschall Sensor: Beschrieben in Abschnitt 2.1.2

Infrarot Sensor: Sendet Infrarot-Signale aus und misst anhand von Stärke von Reflektionen die Entfernung zu Objekten; funktioniert nicht bei Sonnenlicht

Time of Flight Sensor: Beleuchtung einer Szene und Messung der Laufzeit der Lichtwellen; erfasst gesamtes Bild

LIDAR: Beleuchtung eines Punktes und Messung der Laufzeit der Lichtwellen; erfasst einen Punkt

Stereokamera: Verwendung mehrerer Kameras zur Erkennung und Messung von Objekten in Bildern; beschrieben in Abschnitt 2.1.2

Kameras befinden sich zumeist nur nach vorn (oder unten) gerichtet, arbeiten also in Flugrichtung. Ultraschall- und Infrarot Sensoren werden rundherum eingesetzt um Abstände zu weiteren Objekten abzuschätzen. Andere Sensoren kommen nicht zum Einsatz (oder sind aus Konkurrenzwecken nicht dokumentiert).

Außerdem bei Hobby-Anwendern im Internet weit verbreitet², sind Projekte basierend auf „Intel RealSense“-Produkten. Derartige Kameras stellen ein hochauflösendes Bild mit Tiefeninformationen zur Verfügung. Dazu werden Stereo- und Infrarot Kameras und ausgeklügelte Algorithmen verwendet. Jedoch sind dem Autor keine Erfolge im für diese Projekt geforderten Umfang bekannt.

Alle Anwendungen außer einfachen Ultraschallsensoren und Kameras sind nach [WH22a, Kapitel 4.3.8] nicht für dieses Projekt geeignet und werden nicht weiter in Betracht gezogen.

3.4 Pixhawk Software

Die Software des Flugcontrollers „Pixhawk“, Ursprünglich entwickelt von Lorenz Meier et al., bietet umfangreiche Anleitungen zur Entwicklung neuer Software. Diese werden im Projekt aufgegriffen und verwendet.

3.4.1 Simulator

In [WH22b, Kapitel 6.5] wurde erarbeitet, dass die Simulationssoftware „Gym Pybullet Drone“³ am einfachsten zu bedienen sei. In den Anleitungen des Pixhawk⁴ ist diese allerdings nicht aufgeführt. Stattdessen empfohlen wird die Software „Gazebo“. Mit ihr können beliebige Drohnen in beliebigen Umgebungen simuliert werden und auch die Integration von ROS ist vorgesehen. Eine solche Konfiguration ist vereinfacht dargestellt in Bild 3.2a. Allerdings wird angenommen, dass ROS direkt auf dem Flugcontroller zusammen mit der PX4-Software ausgeführt wird. Die realen Bedingungen hingegen entsprechen dem Aufbau in Bild 3.2b. Da es nicht notwendig sein sollte, die Software des Flugcontrollers zu verändern, ist es auch nicht notwendig offiziell unterstützte Software zu verwenden. Weiterhin muss die Entwicklung auf einem Windows® PC stattfinden. *Gym Pybullet Drone* vergleicht sich selbst mit Alternativen, unter anderem „AirSim“⁵ von Microsoft. Dieses wird auch von *Pixhawk* unterstützt und steht als ausführbare Datei bereit. Auch kann der Flugcontroller direkt mit *AirSim* verbunden werden,

²Bspw. <https://www.youtube.com/watch?v=p8frNNYQNV4>, <https://www.youtube.com/watch?v=f0HoyJbYCPQ>

³<https://github.com/utiasDSL/gym-pybullet-drones>[PZZ+21]

⁴<https://docs.px4.io/main/en/simulation/>[Droa]

⁵<https://github.com/microsoft/AirSim>[Mic23]

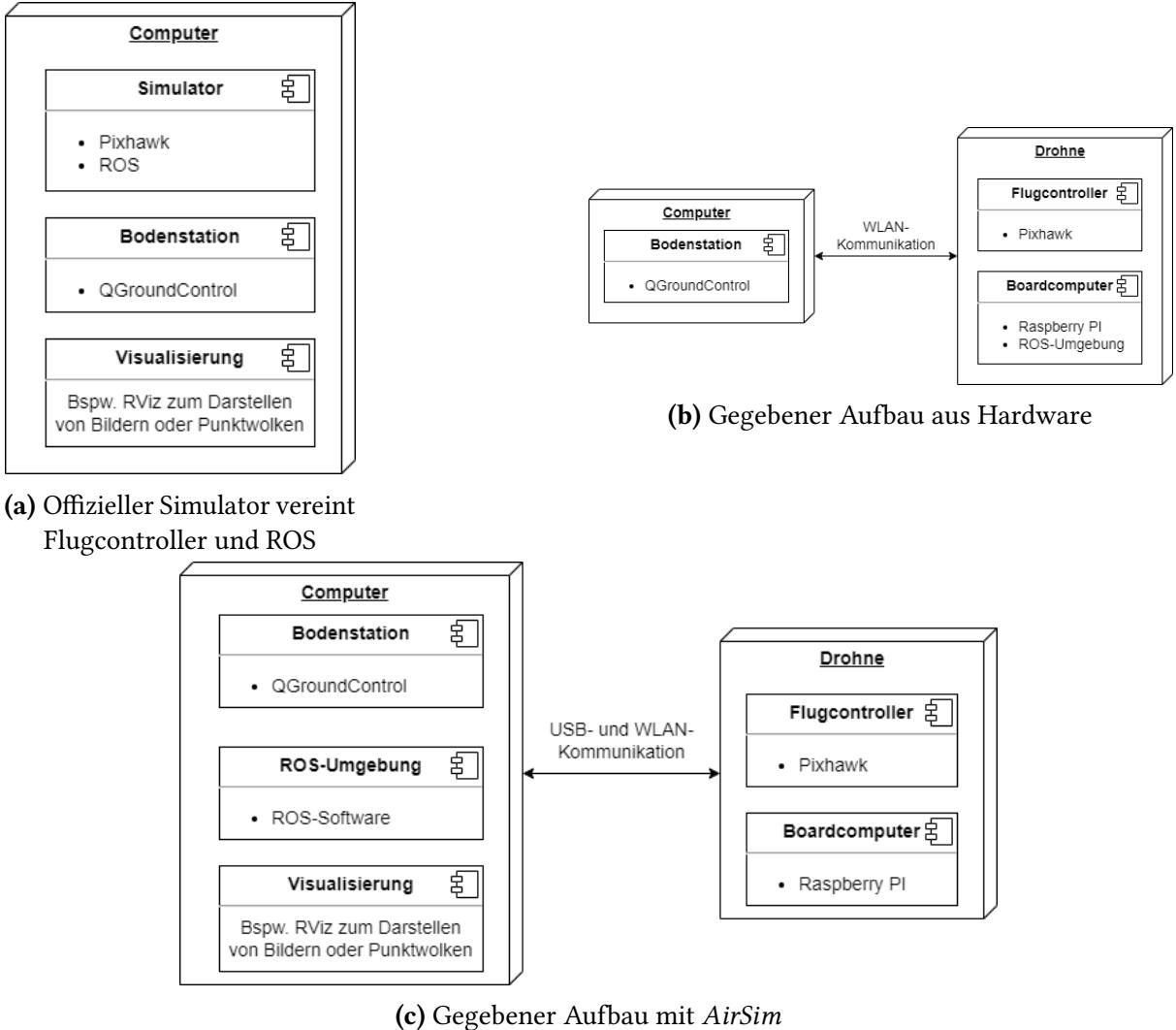


Abbildung 3.2: Systemübersicht in Hardware und im Simulator

sodass ein Hardware in the Loop (HIL) Aufbau wie in Bild 3.2c bereit steht. Nachteilig ist, dass das Programm eingestellt werden soll. Aufgrund der umfangreichen Anleitungen zur Einrichtung bereit[Mic23, siehe ,documentation‘] soll es trotzdem während dieses Projektes zur Entwicklung verwendet werden.

3.4.2 PX4 „Obstacle Detection and Avoidance“

Weitere Arbeiten bezüglich des Autonomen Fluges auf die dieses Projekt aufbauen kann, wurden von Lorenz Meier in Auftrag gegeben. Als Resultat entstanden, unter anderem zwei

Master-Arbeiten und, das Verzeichnis⁶. Dort sind Algorithmen zur Hinderniserkennung, Wegplanung und Navigation analysiert und notwendige ROS-Knotenpunkte bereits definiert. Sie müssen nur noch eingebunden werden.

Zur Implementation kamen bisher nur *Intel RealSense* Kameras in Verbindung mit leistungsstarker Hardware (bspw. PC, Nvidia Jetson) zum Einsatz. Die ROS-Knotenpunkte benötigen eine Tiefenkarte, welche direkt von der Kamera bereitgestellt wird. Ein anderer Anwendungsfall ist nicht beschrieben.

Somit besteht die Aufgabe dieses Projektes darin, entsprechende Tiefenkarte zu generieren, dem Wegplanungsalgorithmus zuzuführen und schließlich dem Flugcontroller das Ergebnis einzuspielen.

Im Verzeichnis stehen 2 Algorithmen zur Verfügung:

- Local Planner
- Global Planner

In dieser Arbeit wird der *Local Planner* eingegangen, näheres in [Tan18]. Dieser wird als Startpunkt empfohlen, da er [Dro23]:

- weniger Rechenaufwand benötigt
- weitgehend getestet wurde
- mit ungenauerem Sensordaten zurecht kommt

⁶<https://github.com/PX4/PX4-Avoidance>[Dro23]

4 Aufarbeitung bestehendes Drohnenprojekt

In diesem Kapitel wird das bestehende Projekt aus nachvollzogenen und überarbeitet. Inhalte sind das Testen bestehender Vorlagen aus [WH22a] und [WH22b], sowie das Aufsetzen einer neuen Arbeitsumgebung wie in beschrieben in [HD22], siehe ??.

4.1 Inbetriebnahme

Zur Aufarbeitung des Projektes steht ist die Drohne bestehend aus Rahmen, Flugcontroller, Motoren mit Propellern, Ultraschallsensoren und Akku bereit. Nicht vorhanden sind die RC-Fernsteuerung und der auf die Drohne montierte RPI. Ursprünglich als Bordcomputer verwendet wurde ein RPI 4 Model B mit 2GB Arbeitsspeicher (RAM). Gleiches kann auch mit dem RPI 3 Model B+ bewerkstelligt werden. Dieser besitzt im Gegensatz nur 1GB RAM und unbedeutend weniger Rechenleistung. Die gelieferte SD-Karte enthält das Betriebssystem Raspbian OS in der Ausführung als 32-bit Betriebssystem. Sowohl RPI Model 4 als auch Model 3 besitzen zwar 64-bit Prozessoren, allerdings wird bei bis zu 4GB RAM das 32-bit Betriebssystem empfohlen um diesen effektiver auszunutzen. Der verwendete RPI startet somit problemlos. Beim Systemstart wird automatisch der WLAN-Hotspot aktiviert und der MAVLINK-Server (mavlink-routerd) gestartet.

Verbindung Flugcontroller mit RPI

Durch das automatische Setup des RPI sollte die Drohne mit dem Einsticken des Stromes bereit zum Flug sein. In [WH22a, Kapitel 4.3.6] ist der erste Schritt beschrieben als "Test der Verbindung". Der RPI wird mit dem Flugcontroller per Serieller Schnittstelle verbunden. Dazu wurde eigens ein Kabel entwickelt, welches die UART-Pins des RPI mit dem TELEM2-Port des Flugcontrollers verbindet. Es ist zu beachten, dass keine Dokumentation bezüglich der Belegung der Kabeladern (per Jumper an den RPI zu Stecken) gegeben ist. Ein Ausmessen des Kabels ergab:

- Schwarz: Masse -> Pin 06 des RPI
- Braun: UART-Receive (Rx) des Flugcontrollers -> Pin 08 (Tx) des RPI
- Weiß: UART-Transmit (Tx) des Flugcontrollers -> Pin 10 (Rx) des RPI

Anschließend soll die Verbindung wie in [WH22a, Kapitel 4.3.6] beschrieben, die MAVLINK-Konsole geöffnet werden. Jedoch ist das Programm „mavproxy“ auf dem aktuellen Image nicht vorhanden. Um das weitere Vorgehen zu vereinfachen wurde das Programm nachinstalliert¹. Das Starten des Konsolenprogrammes ist nur möglich, wenn der *MAVLink* (MAVLink)-Server (Prozess *mavlink-routerd*) nicht läuft (ansonsten ist die Serielle Schnittstelle blockiert). Mit dem Befehl:

```
mavproxy.py --master=/dev/serial0 --baudrate=921600
```

kann schließlich die Kommunikation des RPI mit der Drohne verifiziert werden.

Für das weitere Vorgehen wird das allzeit vorliegende Programm „mavlink-routerd“ verwendet. Es arbeitet als Server im Hintergrund und verbreitet MAVLink-Nachrichten im Netzwerk. Auch *mavproxy* stellt solch eine Funktionalität bereit, allerdings wird aufgrund von zu erwartenden Leistungseinbußen davon abgeraten². Die Konfiguration des *mavlink-routerd* wird wie in 4.1 dargestellt angepasst. Grundsätzlich kann jedes Netzwerkgerät nun eine Transmission Control Protocol (TCP)-Verbindung zum Flugcontroller aufbauen. Außerdem können spezielle Geräte die MAVLink-Nachrichten auch per User Datagram Protocol (UDP)-Protokoll erhalten, müssen aber gesondert eingestragen werden. Eine Auswertung zur Verwendung des geeigneten Protokolls folgt in Abschnitt 4.1. In Bild 4.1c ist das Vorgehen zur Verbindung über einen Netzwerknoten und das erneute Ausführen des Funktionstests mit *mavproxy* dargestellt.

Verbindung Flugcontroller mit PC

Als GCS zur Kommunikation mit dem Flugcontroller wird das Programm QGroundControl, wie in [WH22a, Kapitel 3.4] vorgeschlagen, verwendet. Mit diesem können beliebige Drohnen konfiguriert, parametriert und geflogen werden. Der Flugcontroller kann dazu per Micro-USB mit dem PC verbunden werden und das Programm findet selbigen automatisch.

Per Knopfdruck kann hier die Funktion *arm* ausgeführt werden und die Propeller beginnen zu Drehen. Das Programm stellt sogleich fest, dass keine Fernsteuerung verbunden ist und verfällt in den *manuellen Modus*. Nach einigen Sekunden wird die Drohne automatisch wieder *disarmed* und die Propeller gestoppt um Schäden zu vermeiden.

Um den Flugcontroller über das drahtlose Netzwerk des RPI anzusteuern, muss die GCS manuell konfiguriert werden. Dazu muss eine Einstellung unter *Comm Links* vorgemerkt werden, in welcher die IP-Adresse des RPI und der *TcpServerPort* des MAVLINK-Servers (siehe Bild 4.1b) eingepflegt werden.

¹https://ardupilot.org/mavproxy/docs/getting_started/download_and_installation.html[Ard]

²https://ardupilot.org/mavproxy/docs/getting_started/forwarding.html[Ard]

4 Aufarbeitung bestehendes Drohnenprojekt

<pre>pi@raspberrypi:~ \$ cat /etc/mavlink-router/main.conf [General] TcpServerPort=3000 #5760 ReportStats=true MavlinkDialect=common DebugLogLevel=info [UartEndpoint DROHNEuart] Device = /dev/serial0 Baud = 921600 [UdpEndpoint DROHNEudp] Mode = Normal Address = 192.168.4.12 Port = 14550</pre>	<pre>pi@raspberrypi:~ \$ cat /etc/mavlink-router/main.conf [General] TcpServerPort=5760 ReportStats=true MavlinkDialect=common DebugLogLevel=info [UartEndpoint DROHNEuart] Device = /dev/serial0 Baud = 921600 [UdpEndpoint DROHNEudp] Mode = normal Address = 192.168.1.216 #192.168.4.10 Port = 14550</pre>
---	--

(a) Alte Konfiguration von mavlink-routerd (b) Neue Konfiguration von mavlink-routerd

```
Verbindung herstellen → pi@raspberrypi:~ $ mavproxy.py --master=udp:127.0.0.1:14550
Connect udp:127.0.0.1:14550 source_system=255
Log Directory:
Telemetry log: mav.tlog
Waiting for heartbeat from 127.0.0.1:14550
MAV> Detected vehicle 1:1 on link 0
online system 1
MANUAL> Mode MANUAL
fence breach
Received 848 parameters
Saved 849 parameters to mav.parm
arm throttle
MANUAL> Got COMMAND_ACK: COMPONENT_ARM_DISARM: ACCEPTED
AP: Failsafe mode activated
ARMED
UNKNOWN> Mode UNKNOWN
Arming checks disabled
disarm
UNKNOWN> Got COMMAND_ACK: COMPONENT_ARM_DISARM: ACCEPTED
AP: Disarmed by external command
AP: Failsafe mode deactivated
DISARMED
MANUAL> Mode MANUAL
```

Drohne aktivieren → Drohne deaktivieren →

(c) Verbindung mittels mavproxy auf dem RPI

Abbildung 4.1: Konfiguration und Verwendung von mavlink-routerd

Benchmark zur Verbindung von PC zum Flugcontroller

Zur Durchführung des Projektes steht keine physikalische RC-Fernsteuerung zur Verfügung, sondern es können nur Wegpunkte gesetzt oder ein Konsolen-Controller am PC verwendet werden. Für letzteren Fall ist es entscheidend, dass die Daten über das MAVLink-Protokoll zum Flugcontroller gesendet werden. Dabei ist die Verbindung über das Netzwerk des RPI langsamer als die Verwendung einer echten Fernsteuerung. Um trotzdem die optimale Reaktionsfähigkeit

zu erreichen, soll verglichen werden, ob die Verbindung über TCP oder UDP schneller ist³. Die entscheidenden Kriterien sind:

- Latenz: Reaktionsfähigkeit des Flugcontrollers
- Bandbreite: Auslastung des Datenkanals

Um eine UDP-Verbindung herzustellen, muss die IP-Adresse des Zielrechners in der *mavlink-routerd*-Konfigurationsdatei eingetragen sein, siehe Bild 4.1a unterer Abschnitt. Diese wurde auf den zur Entwicklung verwendeten PC angepasst. Die Packete wurden empfangen, was mit dem Programm „Wireshark“ überprüft wurde. Trotzdem konnte die GCS keine Verbindung über UDP aufbauen. Anschließend wurde die IP-Adresse auf ein Smartphone geändert. Auf diesem hat die GCS beim Öffnen sofort automatisch eine Verbindung hergestellt.

Für das Benchmark wurde ein Python-Skript geschrieben, welches mit beide Transportprotokolle ausprobieren und die geforderten Zeiten ausgeben soll. Dieses wurde sowohl auf einem Windows- als auch auf einem Linux-Rechner ausgeführt. In beiden Fällen konnten keine Verbindung über UDP aufgebaut werden bzw. keine Packete empfangen werden. Die Schlussfolgerung lässt zwei Möglichkeiten zu:

- Die Python-Bibliothek ist teilweise defekt, weshalb keine Packete über UDP empfangen werden konnten.
- Sowohl Windows als auch Linux verfügen über eine Firewall und blockieren sämtliche Packete. Anstrengungen die Firewall abzuschalten wurden unternommen, jedoch ohne Erfolg.

Somit kann eine Verbindung zum Flugcontroller nur über TCP aufgebaut werden und der Benchmark erübrigts sich.

4.2 Einbindung von Funktionalität mittels Docker-Containern

Weitere Funktionen der Drohne sollen mit ROS bereitgestellt werden. Da dieses nicht auf dem Betriebssystem des RPI lauffähig ist, wird es als Docker Containern bereitgestellt.

Aufbauend auf den Projekten [WH22a], [WH22b] sollen bestehende Docker-Container wieder verwendet werden. Eine erste Inspektion mit `docker images` zeigt, dass lediglich das `hello-world` Beispiel bereits heruntergeladen wurde und im Programmspeicher bereit liegt. Weiterhin zeigt der Befehl `docker ps -a` dass auch nur dieses Beispiel bisher ausgeführt wurde. Im bestehenden Quellcode enthalten sind verschiedene Tests. Schon der Test 1 importiert einen Container aus „arm64v8/ros:galactic“. Das besagte Abbild ist für eine 64-bit ARM

³<https://stackoverflow.com/questions/47903/udp-vs-tcp-how-much-faster-is-it>

Architektur gedacht und damit auf dem gelieferten Betriebssystem nicht lauffähig. In weiteren Tests 3 und 4 wurde dieser Mangel erkannt und versucht zu korrigieren.

Test 1 dient der Demonstration von ROS im Zusammenspiel mit Docker und wird in dieser Arbeit nicht betrachtet, da er keine erkennbare Funktion erfüllt.

Test 3 betrachtet das Zusammenwirken mehrerer Container über Netzwerkenpunkte. Das Sende- und Empfangsverhalten wurde untersucht um möglichst effizienten Datenaustausch bereitzustellen. Der Test soll mit dem Befehl `docker-compose up` gestartet werden. Zum Zeitpunkt der Ausarbeitung dieser Arbeit schlägt dies erst einmal fehl, denn das vorgesehene Ubuntu Image für den Server („ubuntu:impish“, zu finden in Zeile 1 in `server/Dockerfile`) aus dem Jahr 2021 ist nicht mehr verfügbar (es wurde kein Image mit "Langzeitsupport" verwendet, also war das Image nur 6 Monate verfügbar). Ein sehr ähnlicher Fehler tritt beim Aufbau des Client auf. Für den Client ist eine ROS2 Distribution vorgesehen, wobei ROS2 vornehmlich für 64-bit Betriebssysteme entwickelt wird und offiziell keine Images für die 32-bit ARM Architektur bereit stellt (die Plattform ärm“ fällt in Tier 3, siehe ⁴; allerdings sind 32-bit arm Images für ältere ROS1 Umgebungen verfügbar). Da der Test überhaupt nichts mit ROS zu tun hat, kann dies getrost verändert werden. Beide Dockerfiles können mit einer simplen Anpassung auf ein unterstütztes Ubuntu (bspw. `ubuntu:focal`) lauffähig gemacht werden⁵. Als Ergebnis wird die Datei „`results.json`“ neu berechnet. Die Ergebnisse sind aufgeführt in Tabelle 4.1.

Tabelle 4.1: Ergebnisse aus [WH22b, Kapitel 6.13], größerer Zahlenwert steht für höhere Übertragungsrate

	requests_http	requests_turbo-gears2	requests_bottle	websocket
Aktuelle Messwerte	965	830	912	3136
Messwerte aus alter Dokumentation	7857	6567	7516	28230

Dort ist sichtbar, dass mittels „websocket“ die meisten Daten übertragen werden können. Zustätzlich eingepflegt wurden die Messergebnisse aus dem alten Projekt, verfügbar auf GitHub⁶. Diese weisen wesentlich größere Werte auf, vermutlich wurde um die Tests durchzuführen

⁴<https://www.ros.org/reps/rep-2000.html>

⁵die Verwendung der aktuellen Version „ubuntu:jammy“ ist nicht möglich, da dieses eine neuere Python Version verwendet, mit der die Tests nicht laufen

⁶<https://github.com/dippa-1/autonomous-drone/blob/test/4-sensors-with-ros/test3-container-communication/client/results.json>

eine leistungsfähigere Hardware verwendet. Mit dem Wissen, dass die Container nicht für den RPI gedacht und auch nicht auf diesem ausgeführt wurden, lässt sich schlussfolgern, dass diese auf einem PC durchgeführt wurden.

Überhaupt ist es fraglich Daten über das „http“-Protokoll zu übertragen, da ROS eigene Mechanismen zum Datenaustausch besitzt.

Test 4 ist unvollständig. Er ist in keiner Ausarbeitung dokumentiert. Die Dateien von Server und Client sind dieselben wie in Test 3. Es sollte wohl die Kommunikation über ROS erprobt werden, dazu kam es allerdings nicht.

Ultraschallsensoren benötigen weiterhin eine Schnittstelle um über ROS Daten zu verbreiten. Eine Schnittstelle wurde ansatzweise entwickelt und steht als „ros2_ultrasonic_sensor“ auf GitHub⁷ zur Verfügung. Das Repository enthält einen angepassten Quellcode basierend auf der Beispielimplementation von ROS2⁸. Es wird beispielhaft ein Ultraschallsensor ausgelesen und der Messwert publiziert.

[Da zum Testen eine funktionsfähige ROS Umgebung notwendig wäre, kann ich hiermit derzeit nichts anfangen.]

4.3 Experiment: Positionsbestimmung der Drohne mit GPS

In [WH22b, Kapitel 6.10, 7.4 und folgende] werden GPS-Daten von der Drohne und Bodenstation ausgelesen. Anschließend wird die Drohne angewiesen zu den Koordinaten der Bodenstation zu fliegen.

Um die Zuverlässigkeit der Navigation zu überprüfen wurde folgender Versuch durchgeführt: Die Drohne wird im eingeschaltenen Zustand manuell bewegt. Dabei wird das GPS Signal aufgezeichnet. Gleichzeitig wird ein weiteres GPS Gerät mitgeführt, welches ebenfalls die Bewegung aufzeichnet. Anschließend werden die Aufzeichnungen miteinander verglichen.

Zur Durchführung wird auf dem Bordcomputer (RPI) das Programm *mavproxy* gestartet. Es legt während es aktiv ist, automatisch eine Log-Datei mit diversen Daten zur Drohne an. Die GPS Daten können nach Beenden des Programmes mit dem Programm *mavtogpx* (in *mavproxy*-Suite enthalten) entschlüsselt werden. Neben der Drohne werden zwei Smartphones angewiesen, ihre aktuelle Position zu tracken. Anschließend wird ein kleiner Spaziergang mit

⁷https://github.com/dippa-1/ros2_ultrasonic_sensor

⁸<https://docs.ros.org/en/galactic/Tutorials/Beginner-Client-Libraries/Writing-A-Simple-Py-Publisher-And-Subscriber.html>

4 Aufarbeitung bestehendes Drohnenprojekt

allen Geräten gemacht. Um möglichst diverse Ergebnisse zu erzielen in 3 Kategorien: in der Ebene, Bergauf, Bergab.

Zur Auswertung wurden die Daten in GoogleMaps hochgeladen. Die Bilder 4.2 zeigen die zurückgelegte Strecke. Auf den Karten sind jeweils farbige Spuren der Smartphones und der Drohne eingezeichnet. Bei näherer Betrachtung (reinzoomen in Maps ist möglich, hier dargestellt ist immer der größtmögliche Ausschnitt) ist zu sehen, dass die Spuren teilweise um mehrere Meter voneinander abweichen. Diese Diskrepanz vergrößert sich teilweise mit der zurückgelegten Strecken, was bedeutet, dass eine aus größerer Entfernung losgeschickte Drohne das Ziel unter Umständen weit verfehlt.

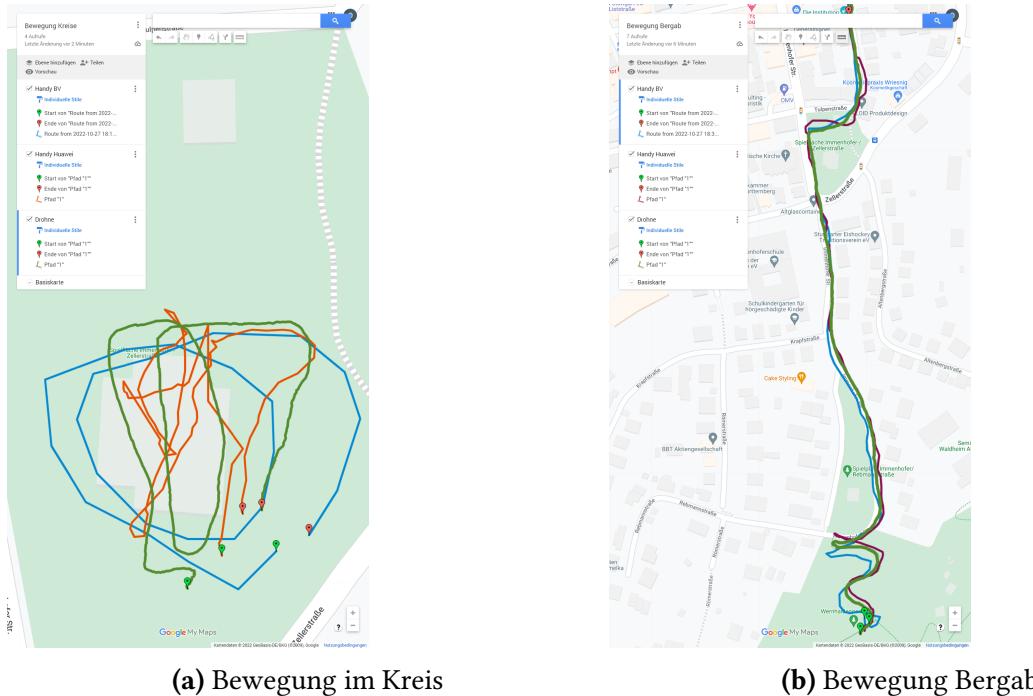


Abbildung 4.2: Vergleich Bewegungsprofil verschiedener Geräte. GPS-Modul der Drohne in Grün, zwei Smartphones in Blau und Orange.

Weiterhin kann aus den GPS Daten Höhe, Geschwindigkeit und Neigung über die Zeit beobachtet werden. Die Drohne nimmt insgesamt mehr Messpunkte als die Smartphones auf, was ihr eine größere Genauigkeit und Zuverlässigkeit verleihen sollte. Auch gibt es bei den Smartphones teilweise Aussetzer bei der Aufzeichnung, die auf Signalverlust oder Software-Probleme zurückzuführen sind. In Bild 4.3 zu sehen ist das Höhenprofil beim Bewegen der Drohne und eines Smartphones. Hätte die Drohne schwerwiegende Abweichungen während des Fluges könnte dies fatale Folgen haben. Auch ist das Profil der Drohne insgesamt ruhiger und glatter, was für gute Messwerte spricht.

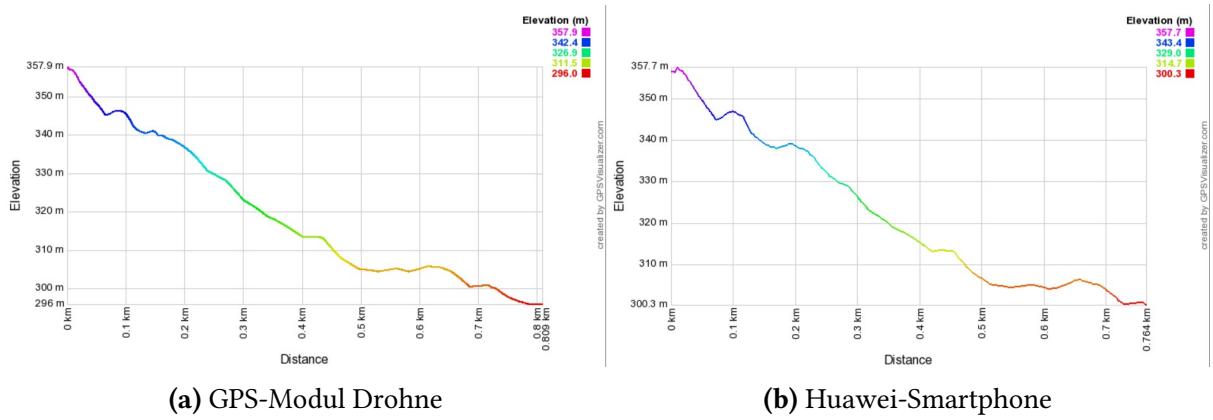


Abbildung 4.3: Vergleich Höhenprofil verschiedener Geräte bei Bewegung bergab.

Um die Genauigkeit des GPS zu verbessern gibt es verschiedene Möglichkeiten⁹. In der Praxis eingesetzt bei Landwirtschaftlichen Maschinen (zentimetergenaue Fahrzeugführung) wird die zweite auf der Website aufgeführte Methode: über das Internet wird ein Korrektursignal abgerufen und dem GPS-Modul zugespielt. Diese Methode könnte auch einfach mithilfe des RPI und einem GSM-Modul (Mobiles Internet über 3G) implementiert werden.

4.4 Ultraschallsensoren am RPI

Zuerst sollen die Ultraschall-Sensoren mit dem RPI verbunden und getestet werden, um die Wiederverwendbarkeit zu bewerten.

In [WH22a, Kapitel 4.3.9] ist beschrieben dass die Signale der Sensoren mit 5V anliegen, aber auf 3,3V herabgesetzt werden müssen um die Pins des RPI nicht zu beschädigen. Die Beschreibung sieht vor dazu einen Spannungsteiler einzusetzen. Bei genauem betrachten des Schaltplans fällt auf, dass der Spannungsteiler falsch gebaut wurde, und nicht die gewünschte Funktion erfüllt. Um die Pins des RPI nicht doch zu beschädigen muss eine alternative Lösung gefunden werden.

Weiterhin wurde ein Python-Script verwendet um die Pins am RPI zu schalten/lesen. Zum anstoßen einer Messung soll das Trigger-Signal für $1\mu s$ auf HIGH (logisch 1) gesetzt werden. Um die Zeitverzögerung zu implementieren, wurde die Funktion „wait“ verwendet. Eine Messung mit dem Oszilloskop zeigt, dass besagtes *wait* im Bereich von 200us liegt.

Eine ähnliche Ungenauigkeit tritt beim Auslesen des ECHO-Signals auf. Die Bibliothek des RPI wird verwendet um die Pins zu Pollen, was den Prozessor unnötig auslastet. Messungen mit konstant eingespeister Einschaltzeit (zu messendes Signal von Signalgenerator erzeugt) ergaben, dass immer 3 von 4 Messwerten gleich, der vierte aber eine Abweichung von ca. 8%

⁹<https://ardupilot.org/copter/docs/common-rtk-correction.html>

hatte.

Somit kann mit Python keine genaue Messung der Ultraschallsensoren durchgeführt werden.

Verwendung eines Arduino zum Auslesen der Sensoren Zur Lösung der Probleme kann bspw. zusätzlich ein Arduino verwendet werden, der die Sensordaten korrekt ausliest und digitalisiert an den RPI weiterleitet. Dies ist eine zuverlässige Lösung, erhöht aber auch gleichzeitig den Stromverbrauch. Um die Lösung einfach zu halten, wird diese trotzdem angewandt.

Zum Senden digitaler Messwerte muss ein Protokoll verwendet werden. Die UART-Pins des RPI sind bereits mit dem Flugcontroller verbunden. Für den Anwendungszweck bietet sich das I2C-Protokoll an. Es erlaubt eine direkte Verbindung des Arduino mit dem RPI, denn es werden vom Arduino keine 5V aktiv geschalten sondern nur der Leitungsbau auf Masse heruntergezogen.

Die Ultraschallsensoren haben laut Datenblatt [ele23] eine Reichweite von ca. 2cm bis 4m und eine Genauigkeit von ca. 3mm. Im Datenblatt wird empfohlen, zwischen aufeinanderfolgenden Messungen mindestens 60ms abzuwarten, um Fehleinstreuungen durch weitere Ultraschallechos zu vermeiden. Um weitere Fehler zu vermeiden wird diese Zeit vorerst auch zwischen den Messungen der 4 Sensoren eingehalten. Eine weitere Verzögerung entsteht durch das Messen selbst (maximal $\frac{380\text{cm}}{0.5 \times 0.034\frac{\text{cm}}{\text{s}}} \approx 22352\mu\text{s}$), diese wird vorsorglich von der jeweiligen Wartezeit abgezogen. Die minimale Periode der Sensordaten beträgt somit $4 \times 60\text{ms} = 240\text{ms}$. Grob gesagt, entspricht die Frequenz der Sensordaten somit 4Hz.

Ein weiteres Problem ist das durch die Ultraschallsensoren verursachte Rauschen der Messwerte. Bei Messungen im Raum mit konstantem Abstand zu Objekten ergaben sich Abweichungen zwischen 2 Messungen von ca. 1% – 2%, siehe Bild 4.4a, 4.4b. Im linken Bild sind nacheinander die Messwerte mehrerer Ultraschallsensoren aufgeführt, das rechte Bild zeigt den zeitlichen Verlauf von Messdaten. Selbst bei Stillstand (Anfang und Ende der Messung) sind Schwankungen in den Messwerten zu sehen.

Auch haben die Ultraschallsensoren ein Problem mit Bewegungen. In Bild 4.4b wurde der Sensor zuerst in Richtung Decke gehalten (Beginn bei ca. 8s auf der x-Achse). Dann wurde er waagerecht gedreht, in den Raum zeigend. Kann der Sensor keinen Wert erfassen, kommt es auf dem Arduino zu einem Timeout und es wird 0 zurückgegeben (ca. bei 11s). Zu beachten sind die starken Abweichungen während den Bewegungen: es kommt immer wieder zu starken Einbrüchen und Anstiegen (bspw. bei 14s) durch teilweisen Verlust des Echo-Signals.

Durch den Feldversuch wird ersichtlich, dass Ultraschallsensoren nur funktionieren, wenn sie nahezu senkrecht auf Flächen gerichtet werden. Somit können auch keine Hindernisse (wie bspw. Wände) detektiert werden, auf die sich die Drohne schräg zubewegt. Da der maximale Entfernungswert nur auf glatten Flächen erreicht werden kann, wird maximale Distanz auf 3.8m festgelegt. Nicht messbare Entfernung (zu nahe oder zu große Entfernung) werden beim weiteren Vorgehen durch die Entfernung von 4m ersetzt.

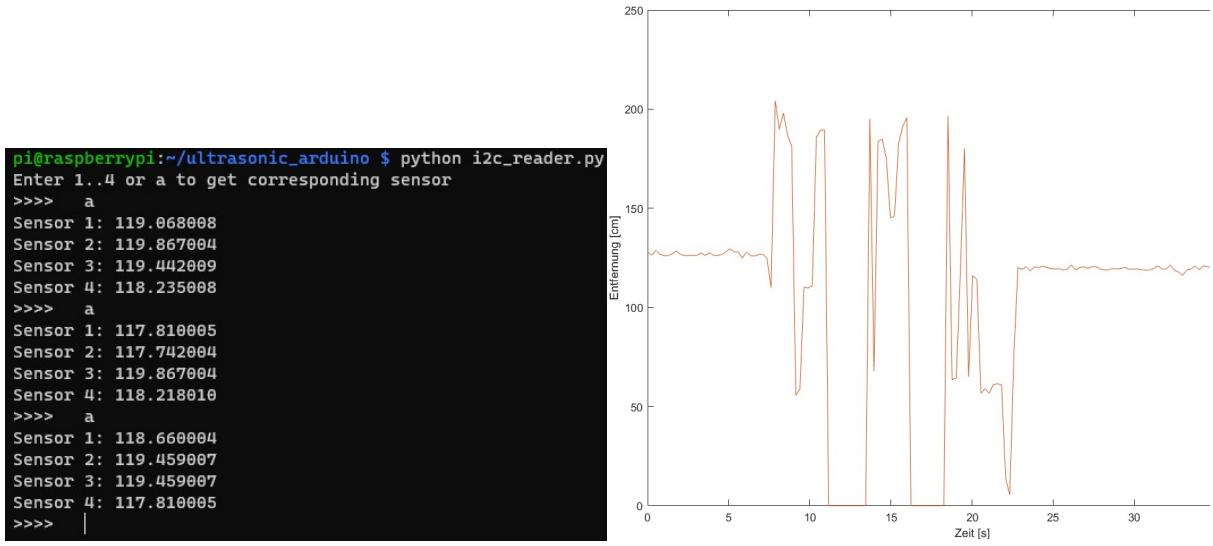


Abbildung 4.4: Demonstration der Ultraschallsensoren

Um die Messwerte der Sensoren sinnvoll verarbeiten zu können ist weiteres Filtern notwendig. Eine einfache Implementation für einen Filter sind Moving Average Filter, oder speziell für diesen Anwendungszweck, wie in [Sup18] gezeigt: Median Filter. Bei dem zuletzt beschriebenen Verfahren werden manuell Grenzen festgelegt, welche Messwerte gefiltert werden müssen. Für die Messungen wurden Abweichungen kleiner 1% ignoriert und größer 3% gefiltert. Vom beschriebenen Vorgehen zum Filtern wurde aber abweichen: anstatt des Medians wurde jeweils der Mittelwert der letzten Messwerte, gespeichert in einem Filter-Array, errechnet. In Bild 4.5 ist eine weitere Messung dargestellt. Die besten Resultate (wenige Sprünge, steile Anstiege) liefert der „Median Filter 1“ und soll im Projekt weiterhin verwendet werden.

Weitere Experimente mit dem Sensor im praktischen Einsatz zeigten, dass es besser ist, Messwertänderungen unter 2% zu ignorieren und ab 5% zu filtern.

[TODO: neue Bibliothek entwerfen, am besten gleich mit ROS]

[TODO: vielleicht gibt es schon eine "direkte Einspeisung", müsste ich mich mal belese]

[TODO: Alternativ kann der Flugcontroller auch einen EKF berechnen]

4 Aufarbeitung bestehendes Drohnenprojekt

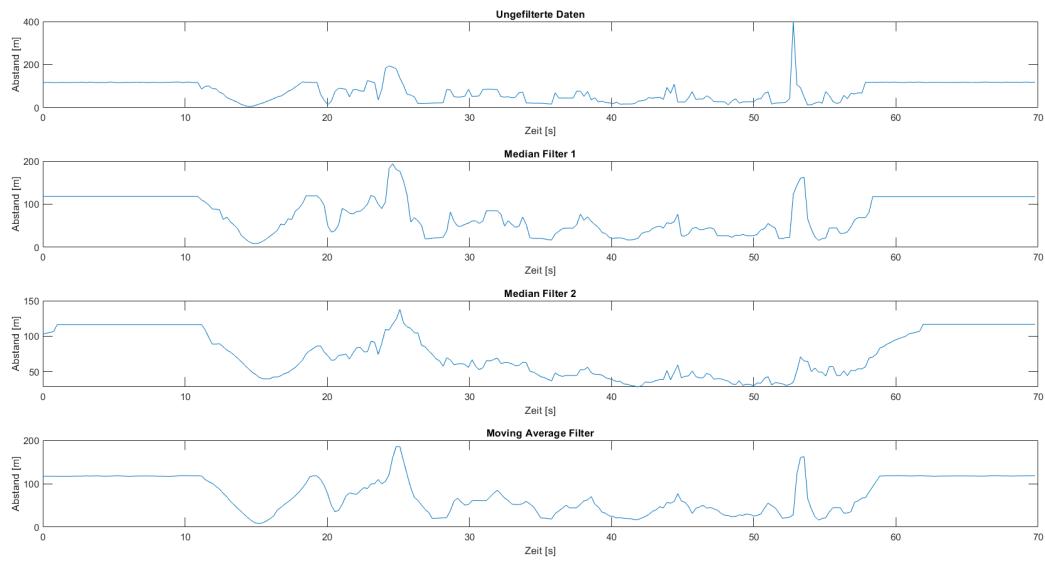


Abbildung 4.5: Messung mit einem Ultraschallsensor. Zu sehen: oben un gefilterte Messwerte; Median Filter 1 pflegt Sprünge in Messdaten direkt in Filter-Array ein aber berechnet Durchschnitt; Median Filter 2 berechnet erst Durchschnittswert und pflegt diesen in ein Filter-Array ein; Moving Average berechnet immer den Durchschnittswert der vergangenen Messwerte.

5 Einführung Navigationsalgorithmus

Für weitere Entwicklungen wird der Aufbau wie in Bild 3.2c verwendet. Bild 5.1 beschreibt die notwendigen Funktionsblöcke. Zuerst wird der Simulator *AirSim* eingerichtet. Weitere Entwicklungen finden jeweils in einem eigenen Container statt, um sie anschließend auf den Bordcomputer übertragen zu können. Die MAVLink-Nachrichten werden vom ROS-Knotenpunkt weiter verbreitet. Mit diesen und dem Bildern aus dem Simulator wird die Hinderniserkennung betrieben. Die generierte Tiefenkarte wird vom *Obstacle Avoidance* Modul verarbeitet und die Navigationsinformationen dem Flugcontroller wieder zugeführt.

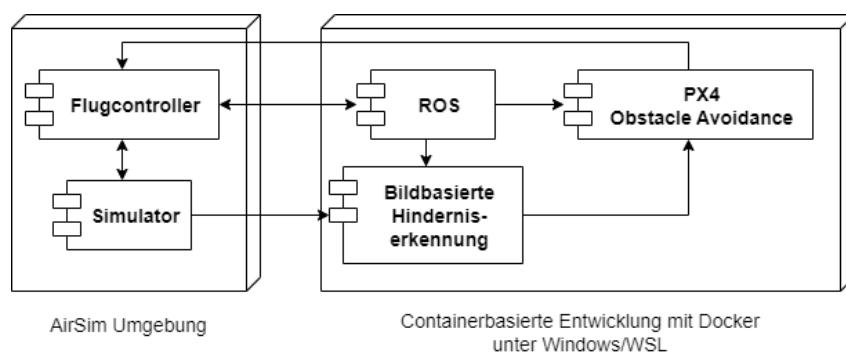


Abbildung 5.1: Aufgliederung Bestandteile der Softwareentwicklung

5.1 Einrichtung der Software

Der erste Schritt ist das Verbinden des Flugcontrollers mit der GCS und das Durchführen eines Updates der Software des Flugcontrollers. Für das Projekt läuft nun PX4, Version 1.13.2. Anschließend wird die GCS angewiesen keine automatische Verbindung (*AutoConnect*) zu *Pixhawk* Geräten durchzuführen, um die Kabelverbindung für die Simulationssoftware frei zu halten.

Um den Flugcontroller mit AirSim zu verwenden, muss dieser in den HIL-Modus versetzt werden. Ausführliche Anleitungen sind verfügbar unter¹. Bild ?? zeigt die virtuelle Drohne im Simulator. Dabei wird der Flugcontroller zurückgesetzt und muss erneut konfiguriert werden,

¹https://microsoft.github.io/AirSim/px4_setup/#setting-up-px4-hardware-in-loop[Mic23]

über den TELEM2-Port MAVLink-Nachrichten zu verbreiten. Nun kann QGroundControl auf zwei Arten betrieben werden:

- Per Netzwerkzugriff über den RPI auf den Flugcontroller zugreifen. (selbe Variante wie in Abschnitt 4.1; wird hier verwendet; wird später verwendet, um die MAVLink-Nachrichten an ROS weiterzuleiten)
- Wenn der Simulator gestartet ist, verbreitet er über UDP-Protokoll MAVLink-Nachrichten (allerdings war unter Windows kein Verbindungsaufbau möglich, siehe Abschnitt 4.1)

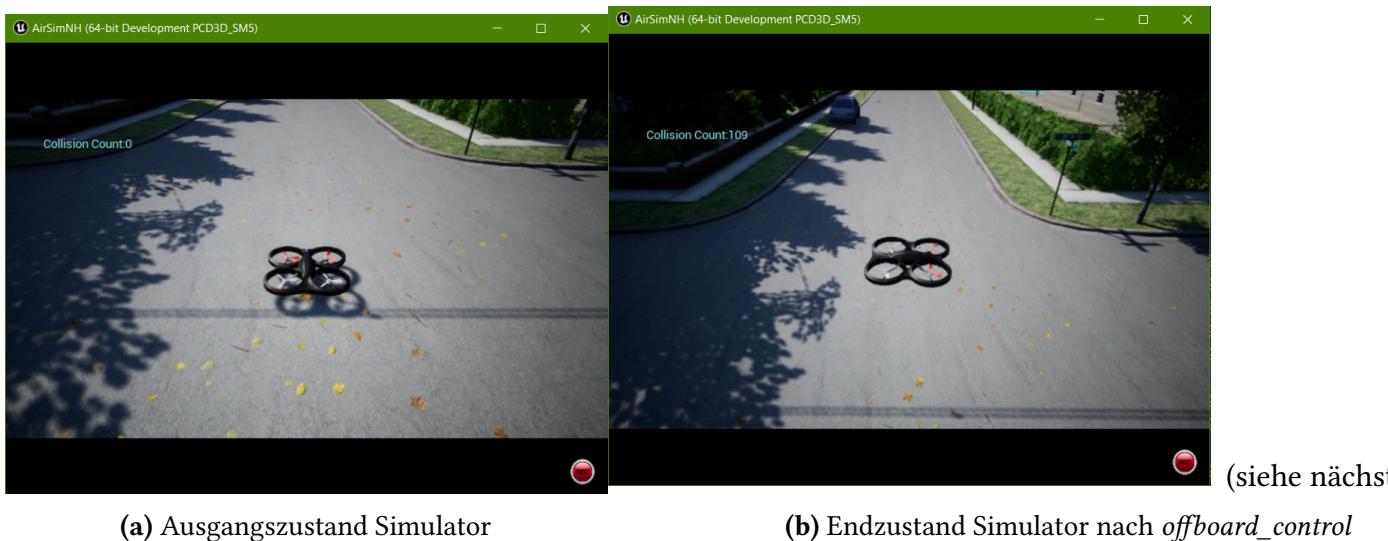


Abbildung 5.2: Simulation der Drohne unter Windows

5.2 Verarbeitung von MAVLink-Nachrichten mit ROS

In diesem Kapitel wird der Funktionsblock „ROS“ aus dem Bild 5.1 entwickelt. Um ROS laufen zu lassen, sind mehrere Bestandteile notwendig. Nach gängiger Praxis wird für jede Funktion ein eigener Container angelegt. Alle Container verwenden dasselbe Image, sodass dieses nur einmalig gebaut wird (mithilfe der *Dockerfile*) und anschließend jeden Container mit eigenem Kommando startet. Diese sind in den folgenden Paragraphen aufgegliedert. Die vollständige Zusammenstellung besteht aus:

- roscore: ROS-Umgebung
- rqt: ROS-Überwachung
- mavros: MAVLink-zu-ROS-Übersetzung
- offboard_control: Beispielanwendung

Alle Container werden mit dem Netzwerk des Host-Computers verknüpft, so müssen keine Ports manuell freigegeben werden.

roscore

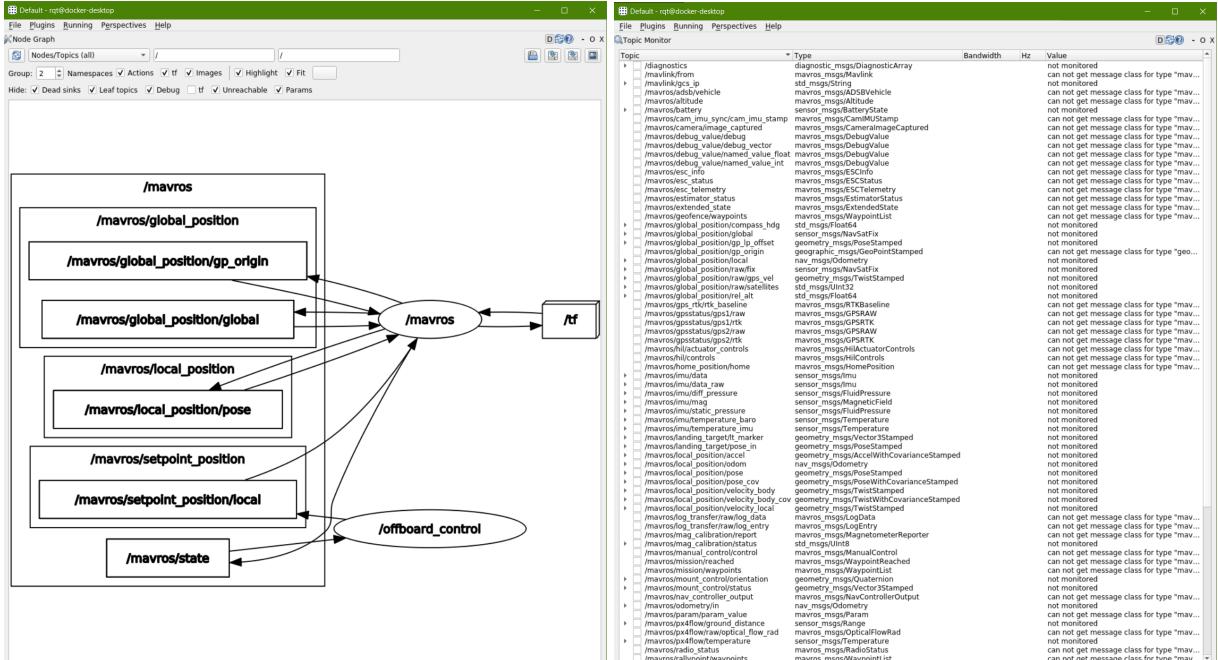
Um auf die Funktionalität von ROS zugreifen zu können, muss das Programm „roscore“ ähnlich einem Server laufen. Dieses ist in jeder ROS-Umgebung enthalten und wird einmalig gestartet.

rqt

Mit dem Tool „rqt“ können ROS-Nachrichten mitgelesen werden. In diesem Anwendungsfall wird das korrekte Ausführen der ROS-Container manuell überwacht.

Um die Topics per Graphical User Interface (GUI) nachverfolgen zu können, wurde diese nach [Fis] eingerichtet und das Programm „VcXsrv“ unter Windows nachinstalliert. Zwar wäre es möglich, Log-Dateien sämtlicher Nachrichten in Textform anzuschauen, doch so etwas umständliches ist kaum während des Betriebes der Drohne möglich.

Das gewählte ROS-Image enthält grundlegend keine GUI-Anwendungen, weshalb diese als Plugins zusätzlich installiert werden (innerhalb der *Dockerfile*). Zur Verfügung stehen bspw. der *rqt-graph* (Bild 5.3a) und der *topic-explorer* (Bild 5.3b)



(a) *rqt-graph* zeigt die Verknüpfungen aller Knoten

(b) *topic-explorer* zeigt alle verbreiteten Nachrichten

Abbildung 5.3: Graphische Anwendung *rqt* aus ROS unter Windows

mavros

Zusätzliche Schritte, beschrieben in [Dro23] und ², werden auf dem Image „noetic-perception-focal“ ausgeführt um das Programm „mavros“ zu installieren. Mit Start des Containers startet auch *mavros*. Dieses kann per Netzwerkzugriff auf den RPI zugreifen. In der entsprechenden Konfigurationsdatei (*Dockerfile*) wird die derzeitige IP-Adresse des RPI und der Zugriff über TCP-Protokoll eingetragen. Spätere Anwendungen auf dem RPI selbst müssen diese anpassen (es kann per UDP auf *localhost* zugegriffen werden solange *mavlink-routerd* läuft **oder** *mavlink-routerd* wird komplett deaktiviert und *mavros* erhält direkten Zugriff auf die Serielle Schnittstelle).

Wird *mavros* im Container gestartet, erscheinen sogleich eine Vielzahl von Warnungen und Fehlern. Eine Warnung erscheint besonders häufig: „RTT too high for timesync“. Diese sagt aus, dass die Zeitverzögerung der Verbindung über Netzwerk zu groß sei. Eine Problemlösung ist diese Warnung abzustellen, beschrieben in ³. Die Prozedur wird auch für dieses Projekt angewandt, direkt während der Einrichtung von Docker. Die Folgen sind zum derzeitigen Zeitpunkt nicht abzuschätzen, wahrscheinlich wird eine Steuerung der Drohne über Netzwerk aufgrund von Verzögerungen nicht möglich sein.

offboard_control

Die Beispieldokumentation stammt aus der offiziellen Dokumentation ⁴. Es wird die C++-Variante genommen, da eine höhere Performance im Vergleich zu Python erwartet wurde. Um die virtuelle Drohne, ohne RC-Fernsteuerung zu fliegen, muss die GCS gestartet sein, und der Joystick-Modus aktiviert sein⁵. Zusätzlich wurden die Timeouts „RC Loss Failsafe Trigger“, „Data Link Loss Failsafe Trigger“ auf 10s erhöht, um eine Kommunikation über den RPI zu ermöglichen.

Die Performance beim Start stellt jedoch das Programm insgesamt in Frage. Im Idealfall verharrt die Drohne in ruhiger Endlage wie in Bild 5.2b dargestellt. Teilweise fliegt sie aber schon beim Start quer durch die Gegend. Nebenbei wird durch die GCS eine Kartenaufzeichnung erzeugt, die nicht der Bewegung im Simulator entspricht. Hindernisse in der Simulation werden nicht von der GCS erkannt, stattdessen wird weiterhin eine Bewegung in entsprechende Richtung angenommen obwohl keine Bewegung stattfindet. Es ist deshalb möglich, dass die Drohne so in einem Hindernis stecken bleibt. Andernfalls bewegt sie sich irgendwann wieder an ihren Ausgangspunkt zurück.

²https://docs.px4.io/main/en/ros/mavros_installation.html[Droa]

³<https://discuss.ardupilot.org/t/rtt-too-high-for-timesync-with-sitl-mavros/38224>

⁴https://docs.px4.io/main/en/ros/mavros_offboard_cpp.html[Droa]

⁵<https://github.com/PX4/PX4-Autopilot/issues/18389>

Bild 5.3a wurde aufgenommen als die Beispielanwendung lief. Es ist zu erkennen, dass das Programm Daten zum Status ([/mavros/state](#)) abfragt und Daten zum Positionsgeber ([/mavros/setpoint_position/local](#)) generiert. Diese Daten stehen im direkten Austausch zum *mavros*-Knotenpunkt, der wiederum mit dem RPI kommuniziert.

5.3 Obstacle Avoidance

Aus dem Bild 5.1 sind noch zwei Themengebiete zu bearbeiten. Zuerst wird die *PX4 Obstacle Avoidance??* eingerichtet. Das Modul benötigt eine Punktfolge der Umgebung, geliefert von einer Kamera. Da kein konkreter Sensor hierfür bereitsteht, wird das Bild einer Tiefenkamera aufbereitet. Zur Entwicklung in 3 Schritten vorgegangen:

1. Tiefenkarte, bereitgestellt von Simulation
2. Bildmaterial der Simulation, aufbereitet als Tiefenkarte
3. Reales Bildmaterial als Tiefenkarte

Die Tiefenkarte kann anschließend in das Punktfolgenformat umgewandelt werden. Für die Versuche an dieser Stelle wird die Tiefenkarte direkt der Simulation entnommen. Anschließend wird das Modul eingerichtet und getestet. Tests mit Kamerabildern erfolgen zu einem späteren Zeitpunkt.

Es werden erneut Docker Container angelegt. Dabei werden die bestehenden Container um entsprechende Funktionalität erweitert.

5.3.1 Bereitstellen der Tiefeninformationen in ROS

Die Bildinformationen aus *AirSim* werden direkt als ROS-Nachrichten übernommen. Für diesen Zweck stellt Microsoft ein eigens den „*AirSim ROS-Wrapper*“ bereit (das Programm entstand erst später als der Simulator, Alternativen anderer Programmierer sind erhältlich), siehe⁶. Nach dem Bau und Aufruf des Programmes *airsim_ros_pkgs* stehen eine Vielzahl neuer ROS-Topics bereit. Viele doppeln sich mit den Informationen von *mavros*. Zusätzlich ist im Startscript von *AirSim* das Programm *rviz* enthalten. Es dient der graphischen Darstellung von ROS-Topics wie Bildern und Vektoren (Transformationsvektoren spielen im Kontext eine wichtige Rolle, es muss bspw. vom Koordinatensystem der Kamera auf die Drohne zurückgerechnet werden).

Die Kamera muss in der Konfiguration des Simulators separat angelegt werden. Der Simulator muss vor dem Container gestartet werden, ansonsten kann *airsim_ros_pkgs* keine Verbindung aufbauen. Im Simulator kann auf First-Person View geschalten werden, siehe Bild 5.4a. Die Aufnahmen der Kamera können innerhalb der ROS-Umgebung mit *rviz* angesehen werden.

⁶https://microsoft.github.io/AirSim/airsim_ros_pkgs/ [Mic23]

Beispielhaft wurden Aufnahmen mit normaler Kamera (Bild 5.4b), planarer Kamera (Bild 5.4c) und perspektivischer Kamera (Bild 5.4d) gemacht. Letztere unterscheiden sich folgend: planare Kamera beschreibt die Abstände von Objekten von einer Ebene aus gesehen, perspektivische Kamera beschreibt die Abstände von Objekten von einem Punkt (der Kamera) aus gesehen. Die Bilder werden mit einer Frequenz von $1 - 3\text{ Hz}$ veröffentlicht (entnommen dem *topic-explorer*). Es wird mit einer Kamera vom Typ „DepthPerspective“ verblieben, da derartige nach⁷ am geeigneten sind, Punktwolken zu generieren.



Abbildung 5.4: First-Person View, Kamera der simulierten Drohne

Das Zwischenprogramm um die Punktwolke zu generieren, „depth_image_proc“, wird ebenfalls in einem Container bereitgestellt. Es arbeitet nur solange es benötigt wird, also wenn die publizierte Topic abonniert wird. An dieser Stelle treten 3 umfangreiche Probleme auf:

Problem 1: Konfiguration von AirSim ROS-Wrapper

Das Programm *depth_image_proc* muss konfiguriert werden, die Topics von *airsim_ros_pkgs* zu abonnieren. Zur Bildumwandlung sind 2 Informationen notwendig, die in dargestellten Topics publiziert werden:

Kamerabild: /airsim_node/PX4/mk1/DepthPerspective

Kamerainformationen: /airsim_node/PX4/mk1/DepthPerspective/camera_info

Dabei ist es nicht möglich, die bereitgestellten Bilder und Kamerainformationen auszulesen, denn die von *depth_image_proc* verwendete Bibliothek („image_transport“) geht davon aus, dass beide Topics im selben Namensraum verfügbar sind. Es wird von der Topic des Kamerabildes der letzte Teil (*DepthPerspective*) abgeschnitten und automatisch durch *camera_info* ersetzt. Zur Lösung wurde der Quellcode von *airsim_ros_pkgs* angepasst, die Bildinformationen mit einem weiteren Unternamen (*image*) zu versehen.

⁷https://microsoft.github.io/AirSim/image_apis/#available-imagetype-values[Mic23]

Problem 2: Veröffentlichung von Bildern

Das Programm *depth_image_proc* ist ein effizienter Publisher/Subscriber im ROS-Ecosystem⁸. Es erfolgt nur eine Ausgabe der Punktfolge wenn ein Eingabebild vorliegt. Somit wäre die theoretische Bildrate hier dieselbe wie die der Kamera.

Das *PX4 Obstacle Avoidance* Modul benötigt jedoch Bilder mit einer Rate von $10 - 20\text{Hz}$ um arbeiten zu können. Als Lösung wurde der Quellcode von *airsim_ros_pkgs* angepasst, ein Kopie des letzten Bildes zu behalten und mit 15Hz zu veröffentlichen. Dabei wurden ROS-interne Mechanismen verwendet, einen Timer zu erzeugen. Die erzeugte Punktfolge wird anschließend unter der Topic *points*, wie vom Originalprogramm vorgesehen, veröffentlicht.

Problem 3: Darstellung der Punktfolge mit rviz

Die veröffentlichte Topic der Punktfolge soll von *rviz* dargestellt werden, um einen Eindruck der Darstellung zu erhalten. Jedoch liegt der Fehler vor, dass die Wolke keinen gültigen Header enthält, der die Transformation im Koordinatensystem beschreibt. Somit weiß *rviz* nicht wohin im Koordinatensystem die Wolke gezeichnet werden soll. Der Inhalt der Punktfolgen-Nachricht ist abgedruckt in Bild 5.5.

The screenshot shows the ROS topic-explorer window. On the left, a tree view lists topics under the root node /airsim_node/PX4/mk1. One topic is selected: /airsim_node/PX4/mk1/DepthPerspective/image/theora/parameter_updates. On the right, detailed information about this topic is displayed:

Topic	Type	Rate	Header
/airsim_node/PX4/mk1/DepthPerspective/image/theora/parameter_updates	sensor_msgs/PointCloud2	631.77B/s	15.00 not monitored
	uint8[]		b''
	sensor_msgs/PointField[]		[]
	std_msgs/Header		
	uint32		
	stamp	3188	
	time		
	uint32	0	
	is_dense	False	
	bool	False	
	point_step	0	
	uint32	0	
	row_step	0	
	uint32	0	
	width	0	
	nav_msgs/Odometry		not monitored

Abbildung 5.5: *topic-explorer* zeigt Daten zur Punktfolge

??.

merken <https://link.springer.com/article/10.1007/s11554-021-01175-y>

⁸<http://wiki.ros.org/roscpp/Overview/Publishers%20and%20Subscribers>

Literaturverzeichnis

- [Ard] ArduPilot Dev Team. *Getting Started – MAVProxy Documentation*. URL: https://ardupilot.org/mavproxy/docs/getting_started/index.html (besucht am 15.01.2023).
- [Cio21] A. E. Ciobanu. *Obstacle Avoidance in DJI Drones*. Droneblog. 1. Dez. 2021. URL: <https://www.droneblog.com/dji-drone-obstacle-avoidance/> (besucht am 15.01.2023).
- [Clo] CloudFactory. *Breaking Down The Levels of Drone Autonomy*. URL: <https://blog.cloudfactory.com/levels-of-drone-autonomy> (besucht am 15.01.2023).
- [Droa] Dronecode Stiftung. *PX4 User Guide*. URL: <https://docs.px4.io/main/en/> (besucht am 15.01.2023).
- [Drob] Dronecode Stiftung. *Ros - Official Image | Docker Hub*. URL: https://hub.docker.com/_/ros/ (besucht am 16.01.2023).
- [Dro23] Dronecode Stiftung. *Obstacle Detection and Avoidance*. PX4 Autopilot for Drones, 15. Jan. 2023. URL: <https://github.com/PX4/PX4-Avoidance> (besucht am 16.01.2023).
- [ele23] elecfreaks. *Ultrasonic Ranging Module HC - SR04*. 1. Jan. 2023. URL: <https://cdn.sparkfun.com/datasheets/Sensors/Proximity/HCSR04.pdf> (besucht am 01.01.2023).
- [Fis] S. Fish. *Running ROS with GUI in Docker Using Windows Subsystem for Linux 2 (WSL2) – Marine Robotics Group*. URL: <https://marinerobotics.gatech.edu/running-ros-with-gui-in-docker-using-windows-subsystem-for-linux-2-wsl2/> (besucht am 17.01.2023).
- [Gru22] bibinitperiod Gruden Prof. Dr.-Ing.. *Sensorik Und Messwertverarbeitung*. 2022.
- [HD22] Harald Wirth, Dominik Helfenstein. *Nachfolger-Info_Studienarbeit_autonome_Drohnen*. 14. Juli 2022.
- [Jür] Jürgen Oehler. *Statistik - Drohnen-Verkäufe nach Drohnen-Modell pro Jahr*. URL: <https://www.drohnen.de/38904/drohnen-statistik/> (besucht am 15.01.2023).
- [Mic23] Microsoft Corporation. *Welcome to AirSim*. Microsoft, 14. Jan. 2023. URL: <https://github.com/microsoft/AirSim> (besucht am 15.01.2023).
- [Ope] OpenUAVAdmin. *Datenverbindung und Flugmodi – LBA – OpenUAV*. URL: <https://lba-openuav.de/onlinekurs/lehrmaterial/allgemeine-uas-kunde/kontrolle-und-flugmodi/> (besucht am 15.01.2023).

- [PZZ+21] J. Panerati, H. Zheng, S. Zhou, J. Xu, A. Prorok, A. P. Schoellig. *Learning to Fly—a Gym Environment with PyBullet Physics for Reinforcement Learning of Multi-agent Quadcopter Control*. Jan. 2021. url: <https://github.com/utiasDSL/gym-pybullet-drones> (besucht am 15.01.2023).
- [Sup18] SuperRenegade. *Filtering Sensor Data • Renegade Robotics*. Renegade Robotics. 16. Mai 2018. url: <https://renegaderobotics.org/filtering-sensor-data/> (besucht am 16.01.2023).
- [Tan18] Tanja Baumann. „Obstacle Avoidance for Drones Using a 3DVFH* Algorithm“. Eidgenössische Technische Hochschule Zürich, Frühjahr 2018.
- [WH22a] H. Wirth, D. Helfenstein. „Erweiterung Einer Bestehenden Drohne Um Eine Automobilflugfähigkeit“. DHBW Stuttgart, 23. Jan. 2022.
- [WH22b] H. Wirth, D. Helfenstein. „Erweiterung Einer Bestehenden Drohne Um Eine Automobilflugfähigkeit (2/2)“. DHBW Stuttgart, 10. Juni 2022.

Erklärung

Ich versichere, diese Arbeit selbstständig verfasst zu haben. Ich habe keine anderen als die angegebenen Quellen benutzt und alle wörtlich oder sinngemäß aus anderen Werken übernommene Aussagen als solche gekennzeichnet. Weder diese Arbeit noch wesentliche Teile daraus waren bisher Gegenstand eines anderen Prüfungsverfahrens. Ich habe diese Arbeit bisher weder teilweise noch vollständig veröffentlicht. Das elektronische Exemplar stimmt mit allen eingereichten Exemplaren überein.

Ort, Datum, Unterschrift