Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Bachelorarbeit

Julian Wollenberg

Entwicklung eines präzisen und robusten Systems zum automatisierten Landen von UAV: Analyse, Vergleich und praktische Evaluierung verschiedener Methoden

Fakultät Technik und Informatik
Studiendepartment Fahrzeugtechnik und Flugzeugbau

Faculty of Engineering and Computer Science
Department of Automotive and Aeronautical Engineering

Julian Wollenberg

Entwicklung eines präzisen und robusten Systems zum automatisierten Landen von UAV: Analyse, Vergleich und praktische Evaluierung verschiedener Methoden

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung

im Studiengang Bachelor of Science Mechatronik
am Department Fahrzeugtechnik und Flugzeugbau
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. Tim Tiedemann
Zweitgutachter: Prof. Dr. Thomas Lehmann

Eingereicht am: 13. November 2023

Julian Wollenberg

Thema der Arbeit
Entwicklung eines präzisen und robusten Systems zum automatisierten Landen von UAV: Analyse, Vergleich und praktische Evaluierung verschiedener Methoden

Stichworte
Drohnen, UAV, automatisiertes landen

Kurzzusammenfassung
In dieser Arbeit werden verschiedene Methoden verglichen, ein Präzisionslandesystem für Unmanned Aerial Vehicles mit optischen Positionierungsverfahren umzusetzen. Diese werden mithilfe einer Simulationsumgebung entwickelt und anschließend in simulierten und realen Flugversuchen evaluiert.

Julian Wollenberg

Title of the paper
Development of a precise and robust system for the automated landing of UAV: analysis, comparison, and practical evaluation of various methods

Keywords
Dones, UAV, Precision Landing, Autonomous Landing

Abstract
This thesis compares different methods of implementing a precision landing system for UAVs using optical positioning methods. These are developed using a simulation environment and then evaluated in simulated and real flight tests.

Inhaltsverzeichnis

Tabellenverzeichnis

Abbildungsverzeichnis

Abkürzungsverzeichnis

CM4 Compute Module 4

FRD Front-Right-Down

GNSS Global Navigation Satellite System

GPS Global Positioning System

GCS Ground Control Station

IoT Internet of Things

LiDAR Light Detection and Ranging

LHIND Lufthansa Industry Solutions

NED North-East-Down

RTK Real Time Kinematics

RTL Return to Launch

ROS Robot Operating System

SITL Software in the Loop

STM Statemachine

UWB Ultra-Wideband

UAV Unmanned Aerial Vehicle

WSL Windows Subsystem for Linux

1. Einleitung

Die rasante Entwicklung unbemannter Flugzeuge (Drohnen), im englischen Unmanned Aerial Vehicles (UAVs), hat in den letzten Jahren zu einer Vielzahl innovativer Anwendungen geführt, sodass diese Technologie von der Luftbildfotografie über Such- und Rettungseinsätze bis hin zur Landvermessung und der Agrarwirtschaft, unverzichtbar geworden ist. Eine Schlüsselkomponente für die weitere Entwicklung von UAVs, ist ein zunehmender Automatisierungsgrad. Durch einen vollständig automatisierten Betrieb erschliessen sich heute aus wirtschaftlicher Sicht eine Vielzahl neuer Einsatzfelder. Beim Einsatz vieler gleichzeitig fliegender Drohnen oder zyklisch betriebener UAVs ist ein System, welches kein menschliches Eingreifen erfordert sinnvoll. Damit kann die Anzahl der eingesetzten Drohnen nach oben skaliert werden, ohne dass der Aufwand zum Betrieb des Systems in gleichem Maße skaliert. Dadurch erschließen sich neue Einsatzmöglichkeiten, z.B. in der automatisierten Überwachung von Waldflächen zur Waldbrandfrüherkennung oder dem zyklischen Aufnehmen von meteorologischen Daten für die Wettervorhersage und Klimaforschung.
Dazu muss vorher jedoch eine entscheidende Herausforderung gelöst werden. Anders als Start und Flug einer Drohne, welche vergleichbar einfach automatisierbar sind, stellt die Landung eine große technische Herausforderung dar. Die Drohne muss nicht nur ohne menschliches Eingreifen auf dem Boden aufsetzen, sondern präzise auf einer Plattform landen, von welcher aus sie geladen, aufbewahrt und sicher vor äußeren Eingriffen geschützt werden kann. Dafür stehen Präzision der Landung und Zuverlässigkeit des Systems an erster Stelle. In dieser Arbeit steht die Analyse, der Vergleich und die praktische Evaluierung verschiedener Lösungen zur Realisierung eines solchen Landesystems im Mittelpunkt.

1.1. Motivation

Diese Arbeit ist Bestandteil eines Projekts der Lufthansa Industry Solutions (LHIND) für ein meteorologisches Forschungsinstituts. Ziel ist die Entwicklung eines Messsystems für meteorologische Daten, wie Windgeschwindigkeit, Windrichtung, Lufttemperatur und Luftdruck, mittels kleiner und leichter Drohnen. Diese sollen mehrmals täglich zeitgleich örtlich verteilt Messungen durchführen. Durch die Konzeption von mindestens 30 gleichzeitig operierender Drohnen ist ein manueller Betrieb stark erschwert. Im Projekt wurde eine modulare Landebox entwickelt, die den automatisierten Betrieb der Drohnen ermöglichen soll. Im Gegensatz zu einer einfachen unpräzisen Landung im Gelände ist es erforderlich, dass die Drohnen über die Fähigkeit verfügen, auf der Landeplattform dieser Boxen selbstständig zu landen. Durch die Projektvorgaben hinsichtlich Kosteneffizienz und leichter Bauweise, kann bei dieser Drohne kein RTK-GNSS (Real Time Kinematics Global Navigation Satellite System) zum Einsatz kommen. Da die typischerweise in Drohnen eingesetzten GNSS Systeme (im Folgenden auch GPS Systeme genannt) die erforderliche Genauigkeit in der Positionierung nicht erreichen, soll stattdessen ein optisches Verfahren zum Lokalisieren der Landeplattform verwendet werden.

1.2. Aufgabenstellung und Aufbau der Arbeit

In dieser Arbeit soll die Frage beantwortet werden, welche Methoden geeignet sind, um ein automatisiertes Landesystem für Drohnen auf Basis optischer Positionierungssysteme zu realisieren. Zusätzlich wird überprüft, ob eine Simulation als Entwicklungswerkzeug in einem Projekt zur Realisierung eines solchen Systems sinnvoll eingesetzt werden kann.
Kapitel 2 gibt einen Überblick über die Grundlagen und den aktuellen Stand der technischen Entwicklung. Das dritte Kapitel definiert die Anforderungen für das hier zu entwickelnde System und verweist auf existierende Arbeiten zu diesem Thema. Daraus wird ein Konzept zur Lösung der gestellten Aufgabe erarbeitet und in drei zu untersuchende Verfahren aufgeteilt.
Im vierten Kapitel wird die Implementierung der drei Verfahren beschrieben und in Kapitel 5 erfolgt eine ausführliche Evaluation, sowohl mittels simulierter als auch realer Flugversuche. Die Arbeit schließt mit einem Fazit und einem Ausblick auf weiterführende Entwicklungsmöglichkeiten

2. Grundlagen

Im Folgenden werden die zum Einsatz kommenden Hard- und Softwarekomponenten beschrieben sowie auf die zum Verständnis dieser Arbeit notwendigen theoretischen Grundlagen eingegangen.

2.1. Modellbildung und Position der Kamera im Raum

Hier werden die Berechnungen dargestellt, welche notwendig sind, um von der Position eines Punktes auf einem Bildsensor seine Position im dreidimensionalen Raum zu ermitteln.

2.1.1. Das Lochkameramodell

Das Lochkameramodell ist eine vereinfachte Modellierung einer Kamera, bei der alle Strahlen durch einen einzigen Fokuspunkt gehen. Es beschreibt den Zusammenhang von Punkten im dreidimensionalen Raum und ihrer Projektion auf eine Ebene. Der Fokuspunkt c bildet den Ursprung eines kartesischen Koordinatensystems und die Ebene $Z = f$ wird als Bildebene bezeichnet. Das Lochkameramodell bildet den Zusammenhang eines Punktes im Raum mit den Koordinaten $X = (X, Y, Z)^T$ und seiner Abbildung auf der Bildebene. Diese Abbildung ergibt sich durch den Schnittpunkt der Linie, welche den Punkt im euklidischen Raum mit dem Fokuspunkt c verbindet, mit der Bildebene. Der Zusammenhang ist in Abbildung 2.1 dargestellt.
Durch Betrachtung der Dreiecke mithilfe des Strahlensatzes ergibt sich mit

$$(X, Y, Z)^T \rightarrow (\frac{fX}{Z}, \frac{fY}{Z})^T \tag{2.1}$$

der Zusammenhang von Weltkoordinaten im dreidimensionalen Raum zu Bildkoordinaten im zweidimensionalen Raum. Im Folgenden wird die Achse, welche durch den Fokuspunkt geht und normal zur Bildebene steht, die Bildachse genannt. Der Schnittpunkt der Bildachse mit

der Bildebene sei der Bildmittelpunkt p und der Abstand der Bildebene zum Fokuspunkt die Brennweite f. Der Zusammenhang zwischen Koordinaten im Raum und Kamerakoordinaten

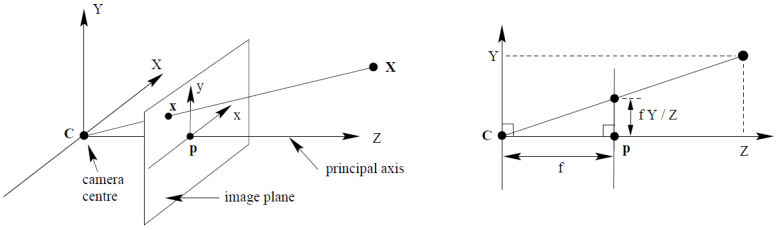


Abbildung 2.1.: Das Lochkameramodell. Quelle: [1] S. 154

lässt sich auch mittels homogener Koordinaten darstellen.

$$\begin{bmatrix} fX \\ fY \\ Z \end{bmatrix} = \begin{bmatrix} f & 0 & 0 & 0 \\ 0 & f & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix} \tag{2.2}$$

Das Bildkoordinatensystem liegt im Bildmittelpunkt. In der Praxis ist es sinnvoll, den Koordinatenursprung in eine Bildecke zu legen. Die Darstellung in homogenen Koordinaten ergibt sich dann wie folgt:

$$\begin{bmatrix} fX + Zp_x \\ fY + Zp_y \\ Z \end{bmatrix} = \begin{bmatrix} f & 0 & p_x & 0 \\ 0 & f & p_y & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix} \tag{2.3}$$

Die Matrix

$$K = \begin{bmatrix} f & 0 & p_x \\ 0 & f & p_y \\ 0 & 0 & 1 \end{bmatrix} \tag{2.4}$$

wird als Kamerakalibriermatrix oder auch interne Kameramatrix bezeichnet.
Die Umrechnung eins Punktes X von Kamerakoordinaten (Position relativ zur Kamera) in Bildkoordinaten kann über den Zusammenhang

$$x = K[1|0]X_{cam} \tag{2.5}$$

erfolgen. [1, S.153 ff.] [2, S. 4 ff.]

2.1.2. Position der Kamera im Raum

Um die Lage eines Punkts im Raum nicht nur relativ zur Kamera, sondern in absoluten Weltkoordinaten beschreiben zu können, muss die Lage des Kamerakoordinatensystems im Raum mathematisch beschrieben werden. Welt- und Kamerakoordinatensystem hängen über Rotation und Translation zusammen. In homogenen Koordinaten lässt sich dies durch Transformationsmatrizen darstellen: [3]

$$ {}^0p = \begin{bmatrix} {}^0r_p \\ 1 \end{bmatrix} = \begin{bmatrix} {}^0R_1 & {}^0r_1 \\ (0\,0\,0) & 1 \end{bmatrix} \begin{bmatrix} {}^1r_p \\ 1 \end{bmatrix} = {}^0T_1\ {}^1p \tag{2.6} $$

Zeichen	Beschreibung
${}^0p$	Punkt im KS0 in homogenen Koordinaten
${}^0r_p$	Ortsvektor zum Punkt P im 0 Koordinatensystem (KS0)
${}^0R_1$	Rotationsmatrix
${}^0r_1$	Ortsvektor vom Ursprung KS0 zu KS1
${}^1r_p$	Ortsvektor zum Punkt P im 1 Koordinatensystem (KS1)
${}^0T_1$	Transformationsmatrix
${}^1p$	Punkt im KS1 in homogenen Koordinaten

Angewandt auf die Umrechnung von Kamera- auf Weltkoordinaten ergibt sich der Zusammenhang wie folgt:

$$ X_{cam} = [R\ t]X \tag{2.7} $$

Dabei ist R die Rotationsmatrix, welche die Drehung der Kamera relativ zum Raum beschreibt und der Vektor t die Verschiebung der beiden Koordinatensysteme abbildet. Diese Transformationsmatrix wird auch als extrinsische Kameramatrix bezeichnet.

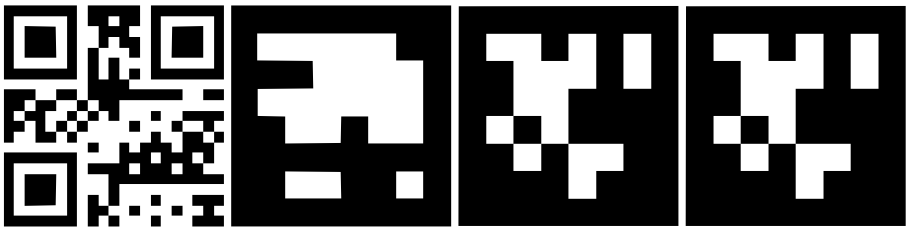
Die Bestimmung der Lage eines Punkts X im Raum auf der Bildebene lässt sich also durch folgenden Zusammenhang bestimmen:

$$ x = K[R\ t]X \tag{2.8} $$

2.2. Fiducial Marker

2.2.1. Allgemein

Fiducial Marker (engl. fiducial: Bezug, Bezugswert) sind optische Marker, welche als Größen- und Positionsreferenz in Kamerabildern platziert werden. Ihre Aufgabe ist nicht primär die Übertragung von Daten, wie z. B. bei QR-Codes und Barcodes, sondern die Lokalisierung von Referenzpunkten in einem Kamerabild. Fiducial Marker werden in verschiedenen technischen Bereichen eingesetzt, z.B. als Referenz in Mikroskopbildern [4] [5] oder als Referenzpunkte für die Platzierung von SMD Komponenten auf Platinen [6, S. 24 f].
Eine weitere, weit verbreitete Anwendung von fiducial Markern findet sich im Bereich der Robotik. Dort werden Marker eingesetzt, um einzelne oder Schwärme von Robotern zu lokalisieren (Marker befinden sich auf den Robotern), , oder damit Roboter sich anhand von Markern auf dem Boden in einem Raum orientieren können. Die in diesem Bereich eingesetzten Marker, erlauben sowohl die Erkennung ihrer Position, als auch ihrer Pose, d.h. Ausrichtung im Raum. Ist die Markergröße bekannt, kann mit einem einzelnen Marker bereits eine Abschätzung der relativen Lage von Marker zu Kamera getroffen werden. Durch die Verwendung mehrerer Marker kann die Genauigkeit der Lokalisierung erhöht werden. Dafür werden Marker verwendet, welche über eine identifizierbare Nummer (ID) verfügen. Solche in der Robotik verwendeten Marker sind beispielsweise APrilTag[7], Aruco[8] oder ARTag[9], siehe Abbildung 2.2.
Da in dieser Arbeit Aruco Marker verwendet werden sollen, wird im Weiteren besonders auf diese eingegangen. Ein Vergleich verschiedener fiducial Marker Systeme findet sich in Alaitzakis et al. [10] und Jurado-Rodriguez et al.2023 [11]



(a) (b) (c) (d)

Abbildung 2.2.: Darstellung verschiedener optischer Marker,
(a) QR-Code, (b) ARTag, (c) APrilTag, (d) Aruco

2.2.2. Aruco Marker

Aruco Marker, erstmals vorgestellt von Garrido-Jurado et al. 2014 [8], ist eine Marker Bibliothek basierend auf ARTag und ARToolkit. Anders als seine Vorgänger erlaubt das System dem Nutzer, eigene Bibliotheken zu erstellen und zu verwenden. Anstatt alle möglichen Marker der Standardbibliothek zu detektieren, kann eine kleinere Bibliothek ausgewählt oder selber erstellt werden. Innerhalb einer Bibliothek sind nur die Marker mit der größten Hamming Distanz (Maß für Unterschiedlichkeit von Codes) enthalten, was die Detektierbarkeit verbessert.
Alle Marker sind als $(n + 2) \times (n + 2)$ Raster definiert, wobei alle äußeren Zellen immer schwarz sind. Die $n \times n$ Zellen im Inneren werden zur Kodierung des Markers verwendet. Damit ist eine Gesamtanzahl von n^2 Markern möglich. In dieser Arbeit wird die vordefinierte Aruco Bibliothek mit $n = 4$ und 50 verschiedenen Markern zurückgegriffen.

Markererkennung

Die Detektion eines Markers erfolgt in mehreren Schritten, wie in [8, S. 6, f] beschrieben. Im ersten Schritt wird im Graustufenbild nach markanten Konturen gesucht. Dafür kann ein Kantendetektionsalgorithmus wie z.B. der *Canny Edge Detector* [12] oder das *local Adaptive Thresholding* Verfahren eingesetzt werden. Gesucht wird nach Bereichen, in denen sich die Helligkeit des Bildes stark zu einem benachbarten Bereich ändert.
Nach der Schwellenwertsegmentierung werden die Konturen mithilfe des Suzuki-Abe-Algorithmus extrahiert. Die meisten gefundenen Konturen sind irrelevant, daher wird eine polygonale Approximation mit dem Douglas-Peucker-Algorithmus durchgeführt. Konturen, die nicht zu 4-Eck-Polygonen approximiert werden können, werden verworfen, und nur die äußeren Konturen bleiben erhalten.
Im nächsten Schritt wird wird das Innere des Markers analysiert und der Code bestimmt. Dafür wird zuerst die perspektivische Verzerrung entfernt. Das resultierende Bild wird mithilfe des Otsus Algorithmus schwellenwertbasiert segmentiert. Dieser Algorithmus ist besonders gut geeignet, um Bilder zu binarisieren, welche im Helligkeitshistogramm über zwei gut differenzierbare Bereiche verfügen. Der binarisierte Bildausschnitt wird in ein Raster eingeteilt ($(n+2) \times (n+2)$, basierend auf der Verwendeten Bibliothek) und für jedes Feld wird, basierend auf der Mehrzahl der Pixel, bestimmt ob der Wert 0 (schwarz) 1 (weiß) ist. Sind alle Werte des Randes 0, wird das Bild weiter analysiert.
Schließlich muss festgestellt werden, welcher der erkannten Marker tatsächlich zum Wörterbuch und welche nur zur Umgebung gehören. Nachdem der Code eines Markers erkannt ist, erhält man vier verschiedene mögliche IDs für die vier möglichen Ausrichtungen. Wenn eine von

ihnen im Wörterbuch gefunden wird, gilt der Kandidat als gültiger Marker.
Im letzten Schritt kann aus den Außenkonturen des Markers die Ausrichtung des Markers gegenüber der Kamera ermittelt werden.

In Abbildung 2.3 sind die einzelnen Schritte der Markererkennung dargestellt.

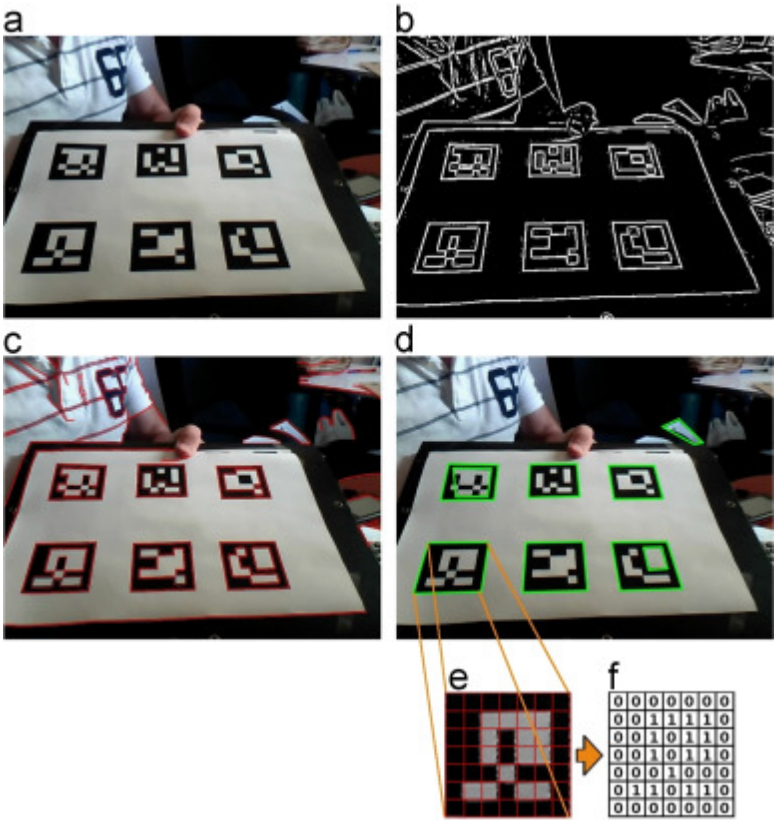


Abbildung 2.3.: Auswertung von Bildern zur Erkennung von Aruco Markern
(a) Originalbild. (b) Ergebnis der lokalen Schwellenwertsegmentierung. (c) Konturerkennung. (d) Polygonale Approximation und Entfernung irrelevanter Konturen. (e) Beispiel eines Markers nach perspektivischer Transformation. (f) Bit-Zuweisung für jede Zelle.
Quelle [8]

2.3. Flightcontroller Software und Kommunikation

2.3.1. Ardupilot

Ardupilot ist eine Open-Source-Softwareplattform für autonome Fluggeräte, Fahrzeuge und Schiffe. Sie wurde entwickelt, um unbemannte Fahrzeuge, wie Drohnen, Flugzeuge, Autos, Boote und Unterwasserfahrzeuge ferngesteuert oder automatisiert zu steuern. Ardupilot bietet eine Vielzahl an Funktionen, darunter Waypoint-basierte Flugmissionen, Telemetrie und Fernsteuerung über Funk, die Einbindung verschiedener Sensoren (GPS, Airspeed, LiDAR, etc.) und Kommunikation mit einem *Companion Computer*
Ardupilot ist auf einer großen Anzahl von Hardwareplattformen lauffähig, was den Einsatz in den unterschiedlichsten Fluggeräten erlaubt.
Die Software hat eine aktive Entwicklergemeinschaft und wird auf der ganzen Welt in verschiedenen Anwendungen eingesetzt, von Luftbildfotografie über Landwirtschaft bis hin zu Forschung und Rettungseinsätzen. Ardupilot ist bekannt für seine Zuverlässigkeit und Flexibilität, was sie zu einer beliebten Wahl für Hobby und professionelle Anwender macht.
Im Folgenden wird in dieser Arbeit nur noch auf den Multirotor-Teil (*Arducopter*) eingegangen und alle Informationen beziehen sich, sofern nicht anders angegeben, auf diese Firmware.

Positions- und Lageregelung

Zur Regelung Lage- und Position im Raum, verfügt Ardupilot über eine Reglerkaskade von aufeinander aufbauenden Regelkreisen. Hier wird nicht weiter auf die technischen Grundlagen von Regelkreisen eingegangen. Dafür empfiehlt sich die Lektüre des Buchs *Regelungstechnik 2: Mehrgrößensysteme, Digitale Regelung* [14] sowie die Masterarbeit von Patrik Vogl [15]
In Abbildung 2.4 (b) ist der Regelkreis zur Lageregelung für eine Achse vereinfacht dargestellt. Der Sollwinkel (*Target Angle*) wird mit dem tatsächlichen Winkel (*Actual Angle*) verglichen und über einen *Square Root Controller* in eine Soll-Drehrate konvertiert. Diese wird mit der tatsächlichen Rollrate verglichen. Zusätzlich wird der Sollwinkel über eine Vorkopplung (*Feed Forward*) direkt auf die Regeldifferenz der Drehrate (*Error*) addiert.
Aus der Regeldifferenz errechnen sich anschließend über einen PID-Regler die Stellgrößen für die Motorregler.
In Abbildung 2.4 (a) ist der Regelkreis für die Positionsregelung vereinfacht dargestellt. Die Differenz aus Soll- und tatsächlicher Position (*Position Error*) wird über einen *Square Root Controller* in eine Sollgeschwindigkeit umgerechnet. Diese wird mit der tatsächlichen Ge-

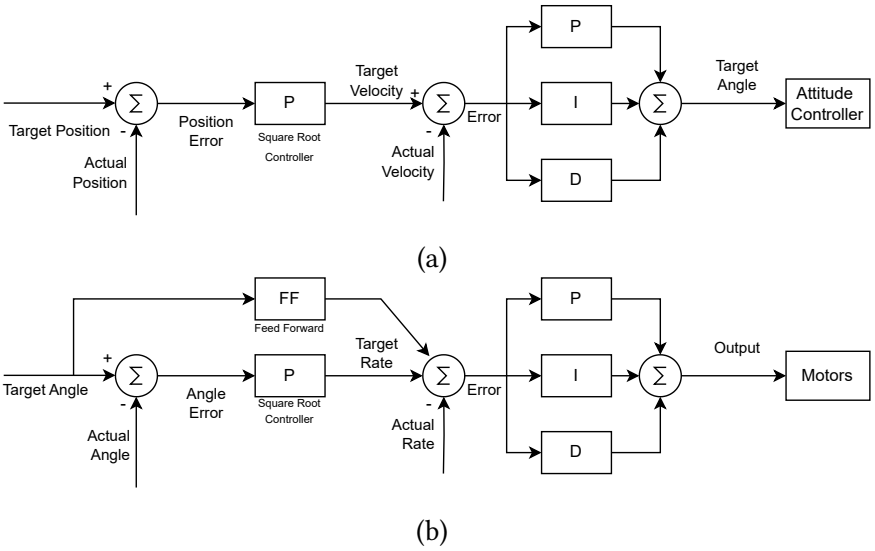


Abbildung 2.4.: Schematische Darstellung der Ardupilot Reglerkaskade zur Positionsregelung.
(a) Lageregelung, (b) Positionsregelung
Quelle: angelehnt an Schaubilder aus Ardupilot Dev Docs [13]

schwindigkeit verglichen und die Differenz über einen PID-Regler in einen Sollwinkel für den Lageregler umgerechnet.

Flugmodi

Ardupilot verfügt über eine Reihe verschiedener Flugmodi, in denen sich das Fluggerät befinden kann. Diese bestimmen das Verhalten bezüglich der Steuerimpulse, sowie den Grad der Stabilisierung und der Autonomie der Drohne. Im Folgenden werden einige relevante Flugmodi vorgestellt [16].

Stabilize

Im *Stabilize* Mode wird die *Roll* und *Pitch* Achse der Drohne stabilisiert und Steuerbefehle von der Fernsteuerung auf diese Achsen werden als Sollwinkel in die Regelung gegeben. Die *Yaw* Achse wird so stabilisiert, dass die Drehgeschwindigkeit auf die vorgegebene Sollgröße geregelt wird. *Throttle* ist in diesem Modus nicht geregelt und wird von der Fernsteuerung aus manuell gesteuert.

Alt Hold

Wie *Stabilize*, nur mit geregelter Flughöhe. Der *Throttle* Stick auf der Fernsteuerung bestimmt die Sollgröße (Steig- und Sinkrate, Mittelstellung bedeutet Flughöhe halten).

Loiter

Vollständige Positionsstabilisierung. Über GNSS oder lokale Positionierungssysteme ermittelte Position sowie Ausrichtung wird gehalten. Eingaben über die Fernsteuerung werden als Sollgeschwindigkeiten interpretiert.

Land

Automatisiertes Landen der Drohne. Wird dieser Modus aktiviert, sinkt die Drohne mit einer definierten Geschwindigkeit bis zu einer Höhe von 10 m über Grund und sinkt dann mit verringerter Geschwindigkeit weiter bis zum Aufsetzen auf dem Boden. Sobald der Bodenkontakt detektiert wird, werden die Motoren gestoppt und die Drohne *disarmed*.
Wenn das *Precision Landing* Verfahren aktiviert ist, wird statt an der aktuellen Position auf einem *Landing Target* gelandet (siehe Abschnitt 3.3.3, S. 25).

RTL

Der *Return to Launch (RTL)* (zum Start zurückkehren) Modus navigiert die Drohne von ihrer aktuellen Position zurück zum Startpunkt und landet dort. Dafür fliegt die Drohne zuerst auf eine definierte Flughöhe (Standard ist $15\,\mathrm{m}$) und dann auf direktem Weg zum Startpunkt zurück.

Auto

Im *Auto* Modus wird eine vorher erstellte Flugmission, bestehend aus Wegpunkten (*Waypoints*) und *Do-Commands*, abgeflogen. Flugmissionen können über eine Ground Station Software erstellt werden und müssen anschließend an das Fluggerät übertragen werden. Die Missionen starten mit einem Wechsel in den *Auto* Flugmodus.

Guided

Guided ist kein klassischer Flugmodus und kann nicht von der Fernsteuerung aus aktiviert werden. Dieser Modus wird verwendet, um dem Flightcontroller über MAVLink Positions- oder Geschwindigkeitsziele zu senden. Das erlaubt ein Steuern der Drohne über eine Telemetrieverbindung zu einer Ground Station oder über einen Companion Computer an Bord der Drohne. Werden keine Befehle gesendet, wird die aktuelle Position und Höhe gehalten.

Ground Control Station Software

Zur Überwachung des Flugs wird eine Ground Control Station (GCS) Software eingesetzt. Für Ardupilot wird dafür meist *Mission Planner* verwendet. Mission Planner kann sowohl zur Konfiguration von Ardupilot, zum Planen von Flugmissionen, dem Flashen von Hardware, dem Visualisieren von Fluglogs als auch im Betrieb zur Visualisierung der Flugtelemetrie eingesetzt werden.
Die Kommunikation mit dem Flightcontroller erfolgt über *MAVLink* . Mission Planner unterstützt dabei eine Vielzahl von Übertragungswegen, unter anderem direkte serielle Verbindungen (z.B. über USB oder ein am PC angeschlossenes Telemetriemodul) oder über eine Netzwerkverbindung (udp/tcp).

2.3.2. MAVLink

MAVLink ist ein Messaging-Protokoll zur Kommunikation zwischen Drohnen, zwischen Bodenstation und Drohne sowie zwischen Komponenten innerhalb einer Drohne. Das Protokoll folgt einem hybriden Ansatz aus Publisher/Subscriber Prinzip (Daten-Streams werden unter einem Topic veröffentlicht) und Point-to-Point Verbindungen, z.B. für die Übertragung von Flugmissionen oder das Abrufen der Flightcontroller Parameter.
Nachrichten werden in XML-Dateien definiert. Jede XML-Datei definiert den Nachrichtensatz, der von einem bestimmten MAVLink-System unterstützt wird (*Dialekt*). Der Referenz-Nachrichtensatz, der von den meisten GCS und Flightcontroller Softwares implementiert wird, ist in common.xml definiert. MAVLink ist gut für die Übertragung von Daten bei begrenzter Bandbreite geeignet. Der aktuelle Standard MAVLink 2 erzeugt nur jeweils 14 bit overhead pro gesendetem Paket. [17]
Die Liste an übertragbaren Nachrichten (MAVLink Common Message Set, common.xml) findet sich auf der MAVLink Website [18].

Nachrichten

Im folgenden wird auf eine Auswahl für diese Arbeit relevanter Nachrichten weiter eingegangen.

LANDING_TARGET

MAVLink Message zum Senden von Positionsdaten eines *Landing Targets*. Wird bei Verwendung des integrierten Landealgorithmus des Flightcontrollers verwendet.
Es stehen zwei Möglichkeiten zum Senden der Daten zur Verfügung:

Target in Bildkoordinaten: Position des *Landing Targets* als Winkel *angle_x* und *angle_y* (rad) relativ zur Kameraachse sowie die Längen *size_x* und *size_y* (Meter) als Abstand auf der Bildebene (Kamerasensor) vom Mittelpunkt des Sensors. Zusätzlich muss der Abstand von der Kamera zum Marker (*distance*) mit gesendet werden.

Target in FRD oder NED Koordinaten: Mit MAVLink 2 besteht zusätzlich die Möglichkeit, die Position eines *Landing Targets* direkt in FRD oder NED Koordinaten zu übermitteln. Dafür werden folgende Daten übertragen:

- die Position (x, y und z) des *Landing Targets*
- das gewählte Koordinatensystem (siehe 2.4)
- die Ausrichtung des Markers als Quaternion
- art des *Landing Targets*
- *Valid Flag* zur Angabe, ob die gesendete Position gültig ist

Zu beachten ist, dass Ardupilot nur die Verwendung des *MAV FRAME BODY FRD* Koordinatensystems für die Angabe der Target Koordinaten unterstützt. [19]

Auf die Verwendung des Precision landing Verfahrens mittels *Landing Target Messages* wird in Abschnitt 3.3.3, S. 25 weiter eingegangen.

SET_POSITION_TARGET_LOCAL_NED

MAVLink Message zum Senden von Zielpositionen, Geschwindigkeiten und Beschleunigungen in einem definierbaren Koordinatensystem. Welche Informationen in einer spezifischen Nachricht übertragen werden, kann über eine Bitmaske gesetzt werden (*type_mask*) Die vollständige Message-Definition findet sich in Tabelle A.1.

MAV_CMD_CONDITION_YAW

MAVLink Befehl zum Senden einer Zielausrichtung mit definierter Drehgeschwindigkeit und Richtung. Ein Auszug aus der Definition findet sich in Tabelle A.2

2.3.3. Software Bibliotheken

Pymavlink

Pymavlink ist eine *Low-Level*-Bibliothek zur Verarbeitung von MAVLink-Nachrichten.
Pymavlink ist in Python geschrieben und bietet eine Schnittstelle zur Erstellung, Verarbeitung und Interpretation von MAVLink-Nachrichten. Dadurch ist es möglich, MAVLink-Kommunikation in eigene Anwendungen zu integrieren. Pymavlink wird in verschiedenen

MAVLink-Systemen eingesetzt, darunter redGCS (z.B. MAVProxy) und Entwickler-APIs (z.B. DroneKit), um die Kommunikation zwischen verschiedenen Teilen eines unbemannten Systems zu ermöglichen [20].

DroneKit

DroneKit ist eine *High-Level*-Bibliothek zum Steuern von UAVs über MAVLink und baut auf der Funktionalität von Pymavlink auf. Über DroneKit kann der Zustand der Drohne überwacht (z.B. *arm/disarm* oder aktueller Flugmodus) sowie aktiv auf diesen Einfluss genommen werden (z.B. Wechsel des Flugmodus). Auch das direkte Senden von MAVLink Nachrichten (siehe Abschnitt 2.3.2, S. 12) ist möglich [21].

2.4. Koordinatensysteme

Die in dieser Arbeit verwendeten Koordinatensysteme orientieren sich in ihrer Benennung und Definition an der MAVLink Definition (siehe Abschnitt 2.3.2)

GLOBAL

Das *GLOBAL* Koordinatensystem (MAVLink : *MAV_FRAME_GLOBAL*) ist ein globales Koordinatensystem basierend auf dem WGS84 Referenzrahmen. Es bildet die Grundlage für globale Navigationssatellitensysteme (GNSS) wie GPS oder Galileo. Dabei wird die Erdkugel in Längen- und Breitengrade unterteilt.
Die Höhe wird in Metern über dem mittleren Meeresspiegel angegeben (positive Koordinatenrichtung zeigt vom Erdmittelpunkt weg).

LOCAL NED

Lokales, kartesisches und erdfestes Koordinatensystem (MAVLink *MAV_FRAME_LOCAL_NED*). Ursprungspunkt muss im *GLOBAL* System gesetzt werden, zum Beispiel am Startpunkt der Drohne. NED steht für *north-east-down*, die X Achse ist nach Norden ausgerichtet, die Y Achse nach Osten und die Z Achse zeigt Richtung Erdmittelpunkt. Höhenangaben (über Grund) haben in diesem Koordinatensystem ein negatives Vorzeichen.

LOCAL OFFSET NED

Lokales, kartesisches Koordinatensystem, dessen Ursprung sich mit dem Fluggerät bewegt, jedoch in seiner Ausrichtung dem *LOCAL NED* System entspricht
(MAVLink : *MAV_FRAME_LOCAL_OFFSET_NED*).

BODY FRD

Lokales, kartesisches Koordinatensystem, dessen Ursprung und Ausrichtung sich mit dem Fluggerät bewegt (MAVLink : *MAV_FRAME_BODY_FRD*). Die F Achse (*front*) zeigt in Flugrichtung, die R Achse davon ausgehend nach rechts (*right*) und die D Achse nach unten (*down*).

Eine vollständige Liste der in MAVLink definierten Koordinatensysteme findet sich unter [18].

FRD ATTITUDE CORRECTED

Lokales, kartesisches Koordinatensystem, dessen Ursprung sich mit dem Fluggerät bewegt. Die Ausrichtung um die globale Hochachse folgt dem *BODY FRD* System, die durch die Achsen F und R aufgespannte Ebene bleibt jedoch immer parallel zur Ebene durch N und E (*LOCAL NED*). Damit ergibt sich ein Koordinatensystem, welches unabhängig von der Fluglage der Drohne ist, sich jedoch mit ihrer Nord-Süd Ausrichtung und ihrer Position mitbewegt. Dieses Koordinatensystem ist nicht Teil der MAVLink Definition.

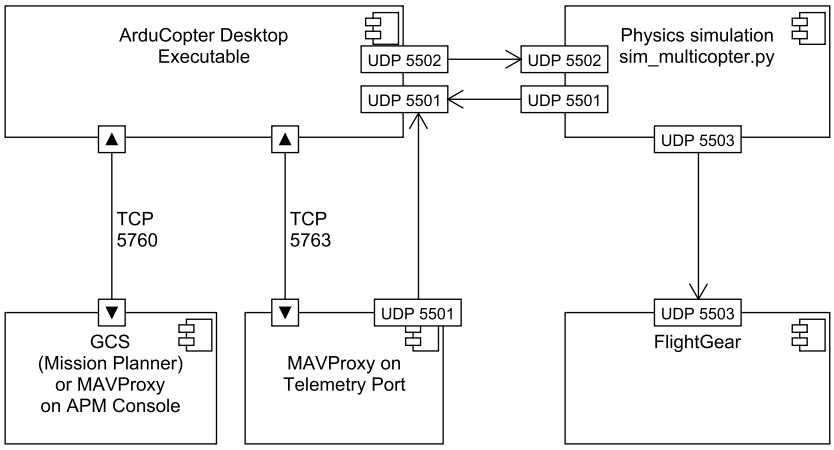


Abbildung 2.5.: Schematische Darstellung Ardupilot SITL unter Windows
Quelle: angelehnt an Schaubild aus Ardupilot Dev Docs [13]

2.5. Simulations-Framework

Zur Entwicklung der im Rahmen dieser Arbeit betrachteten Algorithmen und Verfahren soll eine Simulation zum Einsatz kommen. Da auf der realen Drohne Ardupilot als Flightcontrollerfirmware zum Einsatz kommt, wird Ardupilot als *Software in the Loop (SITL)* Simulation genutzt.
Die Ardupilot SITL besteht aus dem kompilierten Ardupilot Code sowie einem Physiksimulator. Optional lässt sich zusätzlich eine grafische Visualisierung mittels FlightGear integrieren (siehe Abbildung 2.5).
Da der Code unter Windows nicht ohne Weiteres kompiliert werden kann, muss auf eines der beiden folgenden Verfahren zurückgegriffen werden.
In Mission Planner kann im Tab *SIMULATION* eine SITL Instanz gestartet werden. Dadurch wird eine ausführbare Datei (ArduCopter.exe) heruntergeladen. Diese wird im Ordner *C:\Users\%USERNAME%\Documents \Mission Planner\sitl* gespeichert und kann anschließend auch außerhalb von Mission Planner über die Konsole gestartet werden. In der Konsole können auch weitere Argumente übergeben werden. Für diese Arbeit wird die Simulation wie folgt ausgeführt:

```
ArduCopter.exe -Mquad -53.6255937,9.9845971,10.4,100 -s1 --uartA tcp:0 --
    defaults "C:\Users\$USERNAME$\Documents\Mission Planner\sitl\
    default_params\copter.parm" --home=53.6255937, 9.9845971,10.4,100
```

Die zweite mögliche Methode besteht darin, die SITL Simulation in einer virtualisierten Linux Umgebung auszuführen. Dafür bietet es sich an, das Windows Subsystem for Linux (WSL) zu verwenden. In der Linux Umgebung muss das Ardupilot *Build Environment*, wie in den Developer Docs [13] beschrieben, eingerichtet sein. Gestartet wird die Simulation dann über die Datei *sim_vehicle.py*. Diese Methode bietet die Möglichkeit, auch nicht offiziellen oder selbst modifizierten Ardupilot Code zu simulieren.

2.6. Entwicklungshardware

Für die Entwicklung der hier beschriebenen Landeverfahren wurden zwei Drohnen aufgebaut. Beide Drohnen basieren auf den gleichen Elektronik- und Antriebskomponenten, sind jedoch einmal für den Einsatz in Innenräumen und einmal für den Außenbereich ausgelegt.

Allgemeines Konzept

Die Drohne ist ein Quadrokopter, verfügt also über 4 Antriebsmotoren und Propeller. Sie ist mit 8 Zoll Propellern ausgestattet, welche von bürstenlosen Drohnenmotoren der Baugröße 2004 angetrieben werden. Als Akku kommen selbstverschweißte Samsung 50s Lithium-Ionen Becherzellen der Größe 21700 zum Einsatz. Das Akkupack hat eine Kapazität von $5\,\mathrm{Ah}$ und eine Nennspannung von $11{,}1\,\mathrm{V}$. Als Flightcontroller wird ein Matek H743 slim und als Motorregler ein Vierfach-Drohnenregler mit $35\,\mathrm{A}$ maximaler Dauerbelastbarkeit verwendet.
Der Frame der Drohne besteht aus CFK Rohren und im SLS 3D Druck Verfahren gefertigten Nylonelementen, wie Motorhaltern und Klemmungen für die Rohre.

Mit diesem Konzept sind, je nach dem Eigengewicht der Drohne und der transportierten Nutzlast, Flugzeiten von über $70\,\mathrm{min}$ erreichbar.
Der Companion Computer ist mittig unter der Drohne angebracht.

Die Indoor Drohne

Zum Schutz der Propeller kommen bei dieser Version die Schaumstoffteile einer *Parrot AR.Drone* zum Einsatz. Alle weiteren Elemente dieser Drohne werden durch eigene Teile ersetzt.
Die Elektronik wird an speziell gefertigten Halteplatten, die an die originalen Anbindungen des Parrot Frames geschraubt sind, befestigt. Mit dieser Ausstattung erreicht die Drohne ca. $45\,\mathrm{min}$ Flugzeit.

Um in Innenräumen ohne GPS die Position halten zu können, verfügt diese Drohne über einen *optical Flow* Sensor. Dieser kann mithilfe eines zusätzlichen Distanzsensors zum Boden die Relativgeschwindigkeit zwischen Drohne und Grund messen, wodurch die Flightcontroller Firmware über Integration der Geschwindigkeit die Position im NED (*LOCAL NED*) Koordinatensystem berechnen kann. Dieses Verfahren reicht in seiner Genauigkeit nicht aus, um eine absolute Position genau anzufliegen (Drift des Integrals), ist aber ausreichend genau, um die Drohne an einer Position im Raum zu halten und auch kleinere Störungen (Windverwirbelungen, leichtes Wegschieben der Drohne von Hand) auszugleichen. Dafür ist es wichtig, dass der Boden unter dem *optical Flow* Sensor über einen ausreichenden Kontrast verfügt. Aus diesem

Abbildung 2.6.: Bilder der zur Entwicklung und Evaluation verwendeten Drohnen
links *indoor* Variante, rechts *outdoor* Variante

Grund sind auf der zum Testen verwendeten Landeplattform schwarze Punkte um den kleinen Marker herum angebracht (siehe Abbildung 5.9).

Die Outdoor Drohne

Diese Drohne wurde als erster Prototyp für das hier entwickelte System aus automatisierbarer Schwarmdrohne und Landeplattform entwickelt. Sie bildet auch die Basis für die von einer Drittfirma entwickelten Landebox zum Aufladen und Aufbewahren der Drohne.
Die Drohne hat die Abmessungen 490 x 490 x 126 mm (Aufstandspunkte der Landebeine 280 x 280 mm) und wiegt abflugbereit ca. 420 g. Sie ist, wie die Indoor Variante mit CFK Rohren für die Arme aufgebaut. Diese werde in der Mitte von zwei CFK Platten gehalten. Zur Positionsbestimmung ist ein GPS mit integriertem Kompass verbaut.

Alle CNC gefrästen Elemente sowie alle 3D gedruckten Teile wurden im Rahmen dieser Arbeit konstruiert und aufgebaut. Die Auswahl der Komponenten sowie die Auslegung des Antriebskonzept wurde von Helge Hackbarth bei Lufthansa Industry Solutions durchgeführt. In Abbildung 2.6 sind beide Drohnen dargestellt.

3. Konzeption

3.1. Anforderungen und Vorgaben

Diese Arbeit ist im Rahmen meiner Werkstudententätigkeit bei Lufthansa Industry Solutions (LHIND) entstanden. Die hier beschriebenen Vorgaben und Anforderungen richten sich nach dem Projekt, in welches sich diese Arbeit einfügt.
Im LHIND Projekt wird eine kleine und leichte Drohne entwickelt, welche mit unterschiedlicher Sensorik und einem Kamerasystem ausgestattet werden kann. Die Drohne soll zusätzlich selbständig auf einer Landeplattform landen können. Dafür wird parallel eine Landebox entwickelt, in welcher die Drohne geladen und zwischen den Flügen geschützt aufbewahrt wird. Ziel des Projekts ist es, ein vollständig automatisiertes System zu entwickeln, welches für den Betrieb der Drohne keine menschliche Interaktion benötigt. Dies ist notwendig, da die mit dem system ausgestatteten Drohnen als Schwarm agieren sollen, einen hohen Automatisierungsgrad wirtschaftlich zu betreiben ist.
Da sich diese Arbeit in ein größeres Projekt eingliedert, sind bestimmte Vorgaben gesetzt. Diese werden nicht innerhalb der folgenden Analysen verglichen, sondern als Anforderungen für diese Arbeit übernommen.

Art der Lokalisierung

Da die Drohne keine große Nutzlastkapazität besitzt, muss die Hardware, welche für das Landesystem notwendig ist, möglichst leicht sein. Für die Lokalisierung der Drohne relativ zur Landeplattform, soll ein optisches System mit Fiducial Markern verwendet werden. Zum Einsatz kommen hier Arcuo Marker. Dafür kann eine Gimbal Kamera verwendet werden, wenn diese schon auf der Drohne verbaut ist. Andernfalls wird eine zusätzliche Kamera verbaut, welche starr an der Drohne befestigt und senkrecht nach unten ausgerichtet ist. Letzteres System soll auch für die Entwicklung verwendet werden.

Aruco Marker

Die Landebox für dieses Projekt ist so ausgeführt, dass im landebereiten Zustand eine Fläche von 200 cm x 100 cm für die Platzierung von Markern zur Verfügung steht. Der Bereich, auf dem tatsächlich gelandet werden kann, hat eine Fläche von 100 cm x 100 cm. Es wurde festgelegt, dass zum Landen zwei Marker unterschiedlicher Größe verwendet werden sollen. Es soll aus einer Höhe von $25\,\mathrm{m}$ angeflogen werden. Der große Marker soll aus dieser Höhe bereits detektierbar sein. Der kleine Marker hat den Zweck, lange, d.h. aus möglichst geringer Entfernung erkennbar zu sein, um eine präzise Landung zu ermöglichen. Zu diesem Zweck hat der große Marker ein Seitenmaß von $40\,\mathrm{cm}$, der kleine von $10\,\mathrm{cm}$. Der Abstand der Mittelpunkte der beiden Marker beträgt $60\,\mathrm{cm}$. Gelandet wird auf dem kleinen Marker.

Onboard Computing Hardware

Da für diese Plattform bei LHIND schon ein Software Framework besteht, soll der entwickelte Code auf einem Raspberry Pi Computer lauffähig sein. Den besten Kompromiss aus Gewicht, Rechenleistung und Arbeitsspeicher bietet zum Zeitpunkt der Erstellung dieser Arbeit das Raspberry Pi Compute Module 4 (CM4). Dieses wird kombiniert mit einem Carrier Board, auf welchem bereits eine Kamera installiert ist. Weiteres zur verwendeten Hardware und der für diese Arbeit gebauten Drohnen findet sich in Abschnitt 2.6 auf Seite 18.

Flightcontroller Firmware

Auf der Drohne soll die Ardupilot Firmware zum Einsatz kommen, weshalb diese auch auf der Entwicklungsdrohne verwendet wird. Es ist jedoch auch von Vorteil, wenn das hier entwickelte System einfach auf andere Plattformen portiert werden kann.

Präzision

Zum automatisierten Betrieb des Systems, ist es wichtig, dass die Landungen alle präzise auf der Landeplattform stattfinden. Diese ist mit einem Zentriermechanismus ausgestattet, sodass Landungen innerhalb einer Fläche von $100\,\mathrm{cm}$ x $100\,\mathrm{cm}$ cm zulässig sind. Ausserhalb dieser Fläche ist eine Landung nicht mehr erfolgreich, was vermieden werden soll. Dies bedeutet, dass die Landung maximal $35\,\mathrm{cm}$ vom Soll-Landepunkt entfernt erfolgen darf. Auch die Ausrichtung der Drohne, also der Winkel zwischen Drohne und Marker um die Hochachse, darf nur innerhalb enger Grenzen liegen, damit eine Landung als erfolgreich gelten kann. Hierfür wird ein Schwellenwert von $\pm\ 5°$ festgelegt.

Robustheit

Für den Betrieb unter Realbedingungen Robustheit des Systems wichtig. In dieser Arbeit soll

vor allem auf den Betrieb bei Wind optimiert werden. Die maximale Windgeschwindigkeit, bei der noch geflogen werden soll, beträgt $10\,\mathrm{m/s}$.

Entwicklungsframework und Simulation

Zum Einsatz kommen soll ein von Helge Hackbarth bei LHIND entwickeltes Framework, welches für diese Arbeit erweitert werden soll (siehe Abschnitt 4.1). Dieses verfügt über eine Kamerasimulation und lässt sich an die in Abschnitt 2.5 beschriebene ArdupilotSoftware in the Loop Simulation anbinden.
Die Simulation wird hauptsächlich als Entwicklungswerkzeug eingesetzt, um Code schnell testen zu können und die prinzipielle Funktionsweise des Systems zu überprüfen. Da die physikalischen Parameter der simulierten Drohne nur begrenzt editierbar sind, ist der Anspruch bei der Nutzung der Simulation nicht, eine direkte Übertragbarkeit auf das reale System abzubilden. Die Parametrierung der Systeme muss an der realen Hardware erfolgen.

Alle Anforderungen sind in Tabelle 3.1 aufgelistet. Eine Darstellung der beiden optischen Marker findet sich in Abbildung 3.1.

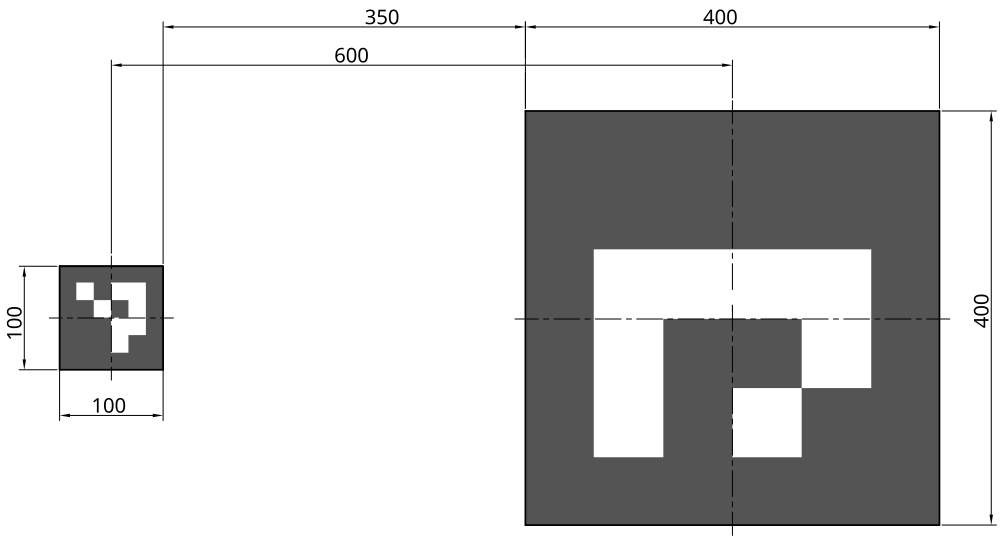


Abbildung 3.1.: Darstellung der Marker zum automatisierten Landen (alle Längen in mm)

Nummer	Art	Bezeichnung	Wert	Einheit
1		**Lokalisierungssystem**		
1.1	F	Art der Lokalisierung	optische Marker	
1.2	F	Art der Marker	Aruco	
1.3	F	Anzahl der Marker	2	
1.4	F	Maße Marker klein	10x10	cm
1.5	F	Maße Marker groß	40x40	cm
1.6	F	Abstand Mittelpunkte Marker	60	cm
1.7	F	Aruco Marker Typ	4x4	
1.8	F	ID Marker klein	0	
1.9	F	ID Marker groß	1	
2		**Hardware**		
2.1	F	Companion Computer	Raspberry Pi cm4	
2.2	F	Carrier Board	WaveShare Nano Base Board (C)	
3		**Landeplattform**		
3.1	F	Fläche zur Platzierung von Markern	200x100	cm
3.2	F	Fläche zur Landung	100x100	cm
4		**Kamerasystem**		
4.1	F	Horizontales *Field of view*	65	°
4.2	M	Auflösung Kamerastream	1024x768	Pixel
4.3	M	Bildfrequenz Kamerastream	15	fps
5		**Abgeleitete Anforderungen**		
5.1	M	Maximale Abweichung vom Soll-Landepunkt	35	cm
5.2	M	Maximale Abweichung Ausrichtung	± 5	°
5.3	Z	Quote erfolgreicher Landungen	100	%
5.4	M	Maximal zulässige Windgeschwindigkeit	10	m/s
5.5	W	Einfach anpassbar auf neue Systeme		

Tabelle 3.1.: Anforderungsliste für die Entwicklung des Landesystems
F = Festforderung, M = Mindestforderung, Z = Ziel, W = Wunsch

3.2. Stand der Forschung

In der wissenschaftlichen Literatur finden sich viele Arbeiten, welche sich bereits mit dem Thema des automatisierten Landens von Drohnen beschäftigt haben. Da sich diese Arbeit nur auf optische Verfahren bezieht, werden im Folgenden auch nur Arbeiten aus diesem Bereich vorgestellt.
Springer 2020 [22] nutzt eine Kombination aus WhyCon und AprilTag Markern, welche durch eine stabilisierte Gimbal-Kamera erkannt werden. Die Arbeit ist in ROS (Robot Operating System) aufgebaut und implementiert einen Regelkreis auf einem *Companion Computer*, um Steuerungsbefehle an die Drohne zu senden.
Dupree und Zhu 2021 [23] nutzen das in Ardupilot integrierte Landeverfahren für IR Sensoren, verwenden dafür jedoch *Fiducial Marker* (AprilTag). Sie setzen eine feste Kamera ein und verwenden einen *Companion Computer* mit ROS.
Nguyen et al. 2020 [24] verwenden Infrarot *Beacons* als Landepunkt und implementieren verschiedene Flugphasen für die Landung, u.a. den *Final Approach*. Bei dieser letzten Phase der Landung, wird die Position ab Unterschreitung einer bestimmten Höhe nicht mehr korrigiert, sondern mit einer vergleichbar zum vorherigen Abstieg hohen Sinkrate bis zum Aufsetzten auf dem Boden geflogen.
Anders als die meisten Arbeiten, setzt Tiziano [25] nicht auf einen geschlossenen Regelkreis zur Steuerung der Drohne, sondern gibt der Drohne die Position des Markers als Positionsziel in GPS Koordinaten vor.

3.3. Vorstellung der Landealgorithmen

Im folgenden Abschnitt wird das allgemeine Konzept zur Umsetzung der automatisierten Landung beschrieben, welches in dieser Arbeit zum Einsatz kommen soll. Im Weiteren werden drei Methoden zur Realisierung des Landealgorithmus vorgestellt. Diese haben sich basierend auf der Recherche und den vorgegebenen Anforderungen als Interessant herausgestellt.

3.3.1. Allgemein

Der Landealgorithmus soll sich in die normale Handhabung der Drohne integrieren. Da auf der verwendeten Drohne Ardupilot als Firmware verwendet wird, überwacht der Algorithmus den Flugmodus der Drohne und greift nur ein, wenn die Drohne in den Modus *Return to Launch (RTL)* oder *LAND* geschaltet wird. Dadurch bleibt die Grundfunktionalität der Drohne unverändert, es wird lediglich eine Präzisionslandung anstelle einer herkömmlichen Landung

durchgeführt, sofern sich eine Landeplattform in der Nähe des *Home*-Punktes befindet.
Um Gewicht bei Drohnen ohne Kameragimbal zu sparen, wird eine feste nach unten ausgerichtete Kamera verwendet. Dafür ist es notwendig, die Koordinaten der erkannten Marker in das *FRD ATTITUDE CORRECTED* Koordinatensystem zu transformieren. Wie bei Nguyen et al. 2020 [24], soll die Landung in unterschiedliche Flugphasen unterteilt werden. Bis ein Marker sicher erkannt wurde, soll das System nicht in die Steuerung der Drohne eingreifen. Solange die Software keinen gültiger Marker erkennt, wird ein Landeanflug auf den eingestellten *Home* Punkt durchgeführt. Der Marker befindet sich an diesem und wird innerhalb der Anflugs erkannt. Ist ein Marker erkannt, soll das jeweilige Landeverfahren aktiv werden.
Während des gesamten Landevorgangs sollte die Drohnenausrichtung kontinuierlich an die des Markers angeglichen werden. Falls zu einem bestimmten Zeitpunkt der Landung absehbar ist, dass diese Anpassung nicht mehr innerhalb der verbleibenden Zeit erfolgen kann, sollte die Landung unterbrochen und erst fortgesetzt werden, wenn die Ausrichtung von Drohne und Marker übereinstimmt.

3.3.2. Positionsbasiertes Landen

Dieser Ansatz zur Realisierung eines automatisierten Landesystems basiert darauf, die Position eines Markers zu bestimmen und diese Position an der Flightcontroller als Zielposition zu übermitteln. Die Update-Rate zum Senden der Markerpositionen kann dabei von mehreren Sekunden (Tiziano [25] nutzt eine Update-Rate von 1 Hz) bis mehreren Updates pro Sekunde gewählt werden. Für dieses Verfahren muss der Marker in einem für den Flightcontroller bekannten Koordinatensystem an die Drohne übertragen werden. Dafür kann das globale Koordinatensystem (MAVLink *GLOBAL*) wie bei Tiziano oder das *LOCAL NED* System verwendet werden.

3.3.3. Ardupilot Precision Landing Verfahren

Ardupilot besitzt einen eigenen Algorithmus zum Landen auf einer Zielposition (*Precision Landing*). Die Daten der Position müssen von einem Companion Computer oder ähnlichen Gerät mittels der MAVLink *LANDING_TARGET* Message übermittelt werden (siehe Abschnitt 2.3.2, S. 12). Dazu ist eine Sendefrequenz von mindestens 5 Hz notwendig.
Entwickelt wurde dieses Verfahren für ein IR-Beacon System, bei dem eine Infrarot Kamera an Bord des Fluggeräts einen Infrarot Lichtsender (Beacon) am Boden erkennt und die Position des Beacons im Kamerabild (Kamerakoordinaten) über MAVLink an den Flightcontroller sendet. Dieses Verfahren benötigt wenig Rechenleistung und kann auf einem Mikrocontroller realisiert

werden.
Der Algorithmus lässt sich jedoch auch für die in dieser Arbeit betrachteten optischen Verfahren verwenden (siehe Dupree und Zhu 2021 [23]). Vorteilhaft dafür ist die in MAVLink 2 hinzugefügte Option, die Position des *Landing Targets* in Weltkoordinaten anzugeben. Ardupilot unterstützt dabei aber nur das *MAV_FRAME_BODY_FRD* Koordinatensystem, also die Position des Markers relativ zur Kamera/Drohne in einem Koordinatensystem, welches an der Drohne ausgerichtet ist und seinen Ursprung mit dem Fluggerät mitbewegt (siehe Abschnitt 2.4). Weitere Einschränkungen sind, dass die *yaw* Ausrichtung der Drohne nicht über dieses Verfahren gesteuert wird. Die Angabe einer Ausrichtung des Markers in Quaternion in der MAVLink *Landing target message* erzielte bei Tests keinerlei Auswirkung.
Auf Ardupilot-Seite lässt sich das Precision Landing Verfahren Parametrieren. Ein Auszug aus den dafür relevanten Parametern ist in Tabelle 3.2 dargestellt.

Parameter	Beschreibung
PLND_ENABLE	Aktivieren des Ardupilot Precision Landings
PLND_TYPE	Art des Precision Landings, z.B. Marker oder IR-Lock
PLND_EST_TYPE	Aktivierung eines Kalman Filters für die Markerposition
PLND_LAG	Latenz der Markererkennung
WPNAV_SPEED_DN	Abstiegsgeschwindigkeit bei Flugmissionen und bei der Landung bis zu einer Höhe von $10\,\mathrm{m}$ in $\mathrm{cm/s}$
LAND_SPEED	Abstiegsgeschwindigkeit in der finalen Landephase in $\mathrm{cm/s}$

Tabelle 3.2.: Relevante Parameter für das Ardupilot Precision Landing Verfahren [26]

Zusätzlich stehen Parameter zum Einstellen von Offsets der Kamera und Landeposition, dem Verhalten im Falle eines Markerverlusts sowie Maximal- und Minimalhöhen zur Verfügung.
Zur Überwachung des Verfahrens dienen die über MAVLink gesendeten Statusmeldungen, die in der Ground Control Station Software eingesehen werden können (MAVLink Konsole). Sobald Ardupilot *Landing Target Message* gesendet und diese als gültig eingestuft werden, wird dies über die Konsole mit der Nachricht „PrecLand: Target Found“ bestätigt.
Nach zwei Sekunden wird erneut überprüft, ob noch gültige Messages empfangen werden. Ist dies der Fall, wird auf der Konsole „PrecLand: Init Complete“ ausgegeben. Der Precision Landing Algorithmus ist damit einsatzbereit. Wird ab jetzt in den *LAND* Flugmodus gewechselt, wird ein Precision Landing durchgeführt. Kommt es innerhalb der zwei Sekunden zu einem Verlust des Markers, wird „PrecLand: Init Failed“ ausgegeben. Wird der Marker nach erfolgreicher Initialisierung verloren, wird dies durch die Meldung „PrecLand: Target Lost“ bestätigt [27].

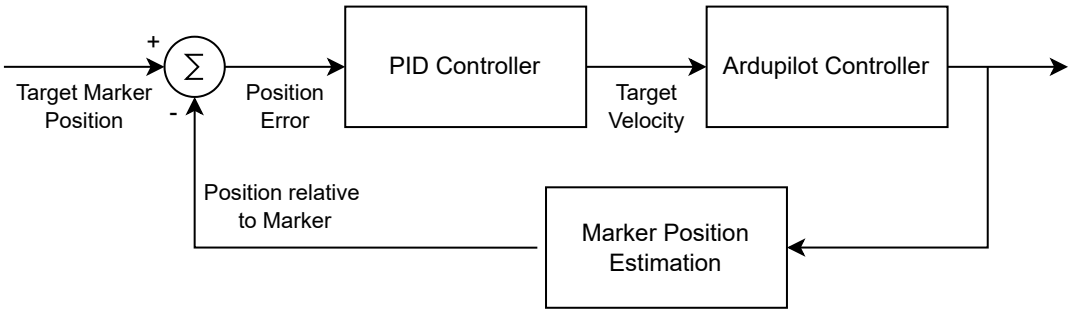


Abbildung 3.2.: Schematische Darstellung des Regelkreises für das geschwindigkeitsbasierte Landeverfahren

3.3.4. Geschwindigkeitsbasiertes Landen

Das dritte Verfahren, welches hier beschrieben und evaluiert werden soll, basiert darauf, der Drohne direkt Steuerungsbefehle in Form von Soll-Geschwindigkeiten vorzugeben. Dazu müssen die Steuerungssignale von der Position der Drohne relativ zum Marker abgeleitet werden. Dafür werden auf dem *companion computer* Regelkreise für die Achsen F und R (*FRD ATTITUDE CORRECTED* Koordinatensystem, siehe 2.4, S. 14) mit PID Reglern implementiert. Die Sollgröße der Regler ist immer null. Die Position relativ zum Marker gehen dadurch direkt als Regeldifferenz in die Regler ein.
Die Vertikalgeschwindigkeit wird nicht mittels eines Regelkreises gesteuert, sondern stufenweise dem Flightcontroller als Sollwert vorgegeben.
Wie auch im positionsbasierten Verfahren soll die letzte Phase des Abstiegs ungeregelt und verhältnismäßig schnell erfolgen.
Eine schematische Darstellung des Regelkreises findet sich in Abbildung 3.2.
Dieses Verfahren ermöglicht neben dem Einsatz mit Ardupilot auch die Verwendung anderer Drohnensysteme. Auch die Anwendung in Systemen, welche keine direkte Schnittstelle für die Integration eines companion computers liefern, ist möglich. In diesen Systemen kann statt der Vorgabe einer definierten Geschwindigkeit ein Steuerimpuls wie mittels einer Fernsteuerung genutzt werden.

4. Implementierung

4.1. Entwicklungs-Framework

Das Framework für die Implementierung der Algorithmen dieser Arbeit wurde von Lufthansa Industry Solutions entwickelt und baut auf der Funktionalität von Dronekit auf. Es beinhaltet unter anderem folgende Features:

- Kompilieren des Python Codes mit *Cython*
- openGL Simulation für Kameras und Lidar Sensoren
- Kommunikation der Module über MQTT
- Statemachine Implementation
- Hilfsfunktionen zum Berechnen geometrischer Zusammenhänge
- Hilfsfunktionen zum Umrechnen von GPS Koordinaten
- High Level Funktionen zum Steuern der Drohne
- Funktionen zum Steuern eines Kameragimbals

Im Rahmen dieser Arbeit wurden folgende Funktionen von mir ergänzt:

- Neuimplementierung des Statemachine Frameworks
- Erkennung und Verarbeitung von Aruco Markern in einem Kamera-Stream
- Skripte zum Kalibrieren von Kameras in openCV
- Echtzeitdarstellung von Werten auf Liveplot-Dahsboard
- grafische Auswertung von Logfiles und Methoden zum vereinfachten Loggen von Daten
- Skripte zur statistischen Auswertung von Flugdaten

4.2. Allgemeine Softwarearchitektur

4.2.1. Statemachine Framework

Das Steuern der einzelnen Flugphasen soll über eine Statemachine realisiert werden. Das dafür verwendete Framework soll objektorientiert sein, muss in der Lage sein, eigene, persistente Daten in den States zu halten und soll es erlauben, die Frequenz, mit der die States evaluiert werden, einzustellen.

Jeder *State* wird als eigene Klasse definiert, welche bestimmte Methoden und Attribute von einer Elternklasse erbt. Die vererbten Methoden *run()* und *next()* müssen zwingend von jedem State realisiert werden, *onEntry()* und *onExit()* sind optional zu implementieren. *run()* und *next()* werden zyklisch bei jedem Durchlauf der Statemachine ausgeführt. *run()* enthält den Code, welcher das Verhalten innerhalb der States bildet, *next()* enthält die Evaluierung der Bedingungen, welche zu einem Wechsel des States führen. An *next()* muss zusätzlich ein Objekt *States* übergeben werden, welches alle in der Statemachine befindlichen States als attribute enthält. Dadurch können die State-Wechsel über einen return der Form *States.nextState*, und nicht wie bei vielen Statemachine Implementierungen in Python, über Strings ausgeführt werden. Die Unterscheidung in *run()* und *next()* Methoden dient hauptsächlich der Übersichtlichkeit des Codes. Die Methoden *onEntry()* und *onExit()* werden entsprechend ihrer Benennung beim Wechsel in einen State und beim Verlassen des States ausgeführt.
Dem Konstruktor können die Attribute *cargo*, *endstate* und *name* übergeben werden. Alle diese Elemente sind optional und werden für den Fall, dass sie nicht übergeben werden, mit Standardwerten initialisiert. *name* kann ein String zur Benennung des States übergeben werden, der verwendet wird, wenn in die Konsole oder ein Logfile Informationen über den Zustand der Statemachine geschrieben werden sollen. Wird kein String dafür übergeben, so wird der Name der Klasse im Code hierfür verwendet. *endstate* ist ein Bool zum Markieren, ob ein State einen Endzustand in der Statemachine markiert. Das wichtigste Attribut ist *cargo*. Hier kann dem State ein Objekt übergeben werden, über welches er mit der restlichen Software kommunizieren kann. Alle Informationen, die zur Evaluation der State-Wechsel notwendig sind sowie alle Aktionen, die aus einem State heraus ausgeführt werden können, müssen in diesem Objekt übergeben werden.
In Code-Ausschnitt 4.1 ist der Code für die *State* Elternklasse sowie ein Beispiel-State *exampleState* und die Klasse *States* dargestellt.

Die Klasse *StateMachine* ist für das Ausführen der Statemachine verantwortlich. Ihr werden im Konstruktor das *States* Objekt (siehe oben), der Anfangs-State sowie *loop_delay* zum Erreichen einer bestimmten Ausführfrequenz in der Einheit $\frac{1}{Hz} = s$ übergeben.

```
class State:
"""
parent class for state
"""
def __init__(self, endstate = False, name = None):
    self.endstate = endstate
    self.name = name
    if self.name == None:
        self.name = self.__class__.__name__
def run(self):
    assert 0, "run method not implemented"
def next(self, states):
    assert 0, "next method not implemented"
def onEntry(self):
    pass
def onExit(self):
    pass

class States:
    def __init__(self, cargo):
      self.exampleState = exampleState(cargo)
      self.nextExampleState = nextExampleState(cargo)
class exampleState(State):
    def __init__(self, cargo, endstate = True, name = 'Example State'):
        super().__init__(endstate, name)
        self.cargo = cargo
    def onEntry(self):
        # do something on entry
        pass
    def onExit(self):
        # do something on exit
        pass
    def run(self):
        # do something everytime the state gets evaluated
        pass
    def next(self, states: States):
        if cargo.exmple_condition is True:
            return states.nextExampleState
```

Quellcode 4.1: Code-Beispiel zum Statemachine Framework, *State* Elternklasse, Klasse *States* und Implementierung eines States *ExampleState*

Mit der Methode *runAll()* wird die Statemachine gestartet und bleibt dann bis zum Erreichen des End-States in einer While-True Schleife.
Bei jedem Durchlauf wird zuerst evaluiert, ob die *next()* Methode des aktuellen States eine State-Änderung zurück gibt. Ist dies der Fall, werden die entsprechenden *onExit()* und *onEntry()* Methoden ausgeführt und das Attribut *self.currentState* auf den neuen State gesetzt. Anschließend wird die *run()* Methode des aktuellen States aufgerufen.
Um eine feste Durchlaufzeit zu erreichen, wird vom *loop_delay* die bis hierhin benötigte Zeit zum Ausführen der Statemachine abgezogen. Über *time.sleep(wait_time)* wird der Code dann bis zum nächsten Loop-Durchlauf angehalten. Zum Ausführen der Statemachine müssen *stateMachine* und das *States* Objekt importiert werden. Wenn mehrere Statemachines im Code vorliegen, ist es sinnvoll, die States mit einem aussagekräftigen Namen zu importieren.
Im Code-Ausschnitt 4.2 wird eine Statemachine *groundTesting* ausgeführt:

```
1 from StateMachine_lib import StateMachine
2 from mission_ground_testing import States as groundTestingStates
3
4 states = groundTestingStates(missionProfile)
5 groundTestingSTM = StateMachine(states, states.initializing, loop_delay
      =0.066)
6 groundTestingSTM.runAll()
```

Quellcode 4.2: Ausführen der Statemachine *groundTesting*

Der vollständige Code für die Klasse *StateMachine* ist in Code-Ausschnitt 4.3 dargestellt, in Abbildung 4.1 ist das Klassendiagramm zum Statemachine Framework abgebildet.

```
class StateMachine:
  """
  class definition for state machine
  """
  def __init__(self, states, initialState, loop_delay):
      self.states = states
      self.currentState = initialState
      self.newState = initialState
      self.currentState.onEntry()
      self.loop_delay = loop_delay

  def runAll(self):
      while True:
          start_time = time.time()
          self.newState = self.currentState.next(self.states)
          if not self.currentState == self.newState and not self.newState
              == None:
              if not self.newState.cargo is None:
                  if not self.newState.cargo.logger is None:
                      self.newState.cargo.logger.info("{\"statemachine
                          \":{\"state\":\"" + self.newState.name + "\"}}")
              self.currentState.onExit()
              self.newState.onEntry()
              self.currentState = self.newState
          self.currentState.run()
          if self.currentState.endstate:
              print('endstate reached')
              break
          execution_time = time.time() - start_time
          wait_time = self.loop_delay - execution_time
          if wait_time > 0:
              time.sleep(wait_time)
```

Quellcode 4.3: Implementierung der Klasse *StateMachine*

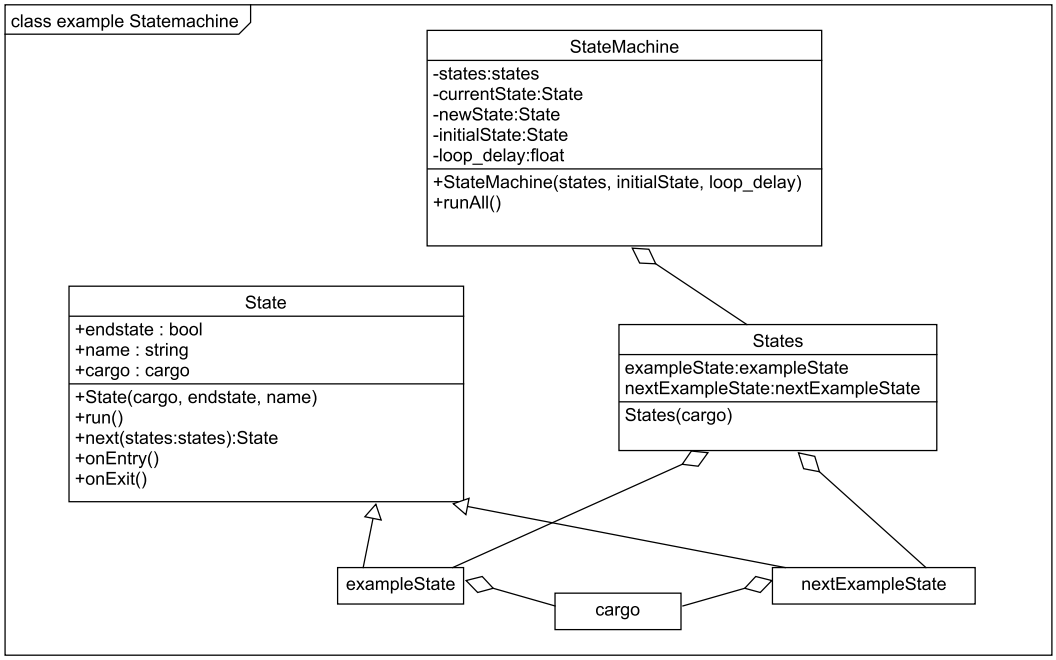


Abbildung 4.1.: Klassendiagramm zum Beispielcode des Statemachine Frameworks

4.2.2. Erkennung und Verarbeitung der Marker im Kamerabild

Die Erkennung der Marker im Kamera-Stream soll in einem eigenen Thread, unabhängig vom restlichen Code erfolgen. Seine Aufgabe ist die Bestimmung der Markerpositionen im Bild (Bildkoordinaten) sowie die sich aus der Pose der Marker und den extrinsischen und intrinsischen Kameraparametern ergebenden Rotationsmatrizen und Verschiebungsvektoren. Die Verknüpfung der Daten mit dem restlichen Code erfolgt über einen MQTT Stream, welcher für jeden erkannten Marker die oben beschriebenen Daten übergeben bekommt.
Zusätzlich kann ausgewählt werden, ob die Markererkennung auf einem realen Kamerabild oder dem simulierten Kamera-Stream läuft.
Zum Überwachen der Flüge kann das analysierte Kamerabild inklusive der erkannten Marker lokal angezeigt oder über einen RTSP Stream an einen Streaming Server gesendet werden.

Erkennung der Marker

Die Erkennung der Marker erfolgt in der Klasse *ArucoMultiTracker*. Ihr werden die zu suchenden Marker, Kameraparameter, MQTT Informationen, welche Videoquelle verwendet werden soll sowie Flags zum Anzeigen des Kamerabilds und des Videostreams übergeben. Über die Methode

track() wird die Erkennung ausgeführt. Über das optionale Flag *loop* kann gesteuert werden, ob die Markererkennung nur einmalig oder in einer While-True-Schleife laufen soll. Wird zweiteres gewählt, sendet *track()* die gefundenen Daten selbständig an den MQTT Broker. Der Ablauf des Codes ist in Abbildung 4.2 dargestellt.

Weitere Verarbeitung der Markerdaten

Die Liste der erkannten Marker inklusive Timestamp wird im Objekt *cargo* der Statemachine (*missionProfile*) gespeichert. Für jeden Marker, der in der Statemachine verwendet werden soll, wird eine Instanz der Klasse *ArucoMarker* erzeugt. Diese Klasse kapselt alle notwendigen Berechnungen, um den Marker in der Statemachine verwenden zu können. Die Klasse *ArucoMarker* ist in Abbildung 4.3 dargestellt.

Für die in dieser Arbeit betrachteten Landeverfahren muss die Markerposition in unterschiedlichen Koordinatensystemen vorliegen. Die Transformation von Bildkoordinaten in das FRD System erfolgt durch die OpenCV Methode *aruco.estimatePoseSingleMarkers()* in der Klasse *ArucoMultiTracker*. Die Theorie zu dieser Koordinatentransformation findet sich in Abschnitt 2.1, S. 3. Diese Methode liefert eine Rotationsmatrix $R_{marker-kamera}$ und einen Transformationsvektor T zurück, welche auch genutzt werden können, um über folgenden Zusammenhang die Koordinaten im LOCAL OFFSET NED System zu berechnen.

$$R_{marker-kamera} = R^{T}_{kamera-marker} \tag{4.1}$$

$$\text{Position Kamera} = -R_{marker-kamera} \cdot T \tag{4.2}$$

Da die erkannte Pose der Marker jedoch, je nach Größe des Markers im Kamerabild, erhebliche Störungen aufweisen kann, ist zusätzlich noch eine Methode implementiert, welche die Lagedaten der Drohne zur Berechnung der Markerposition verwendet.

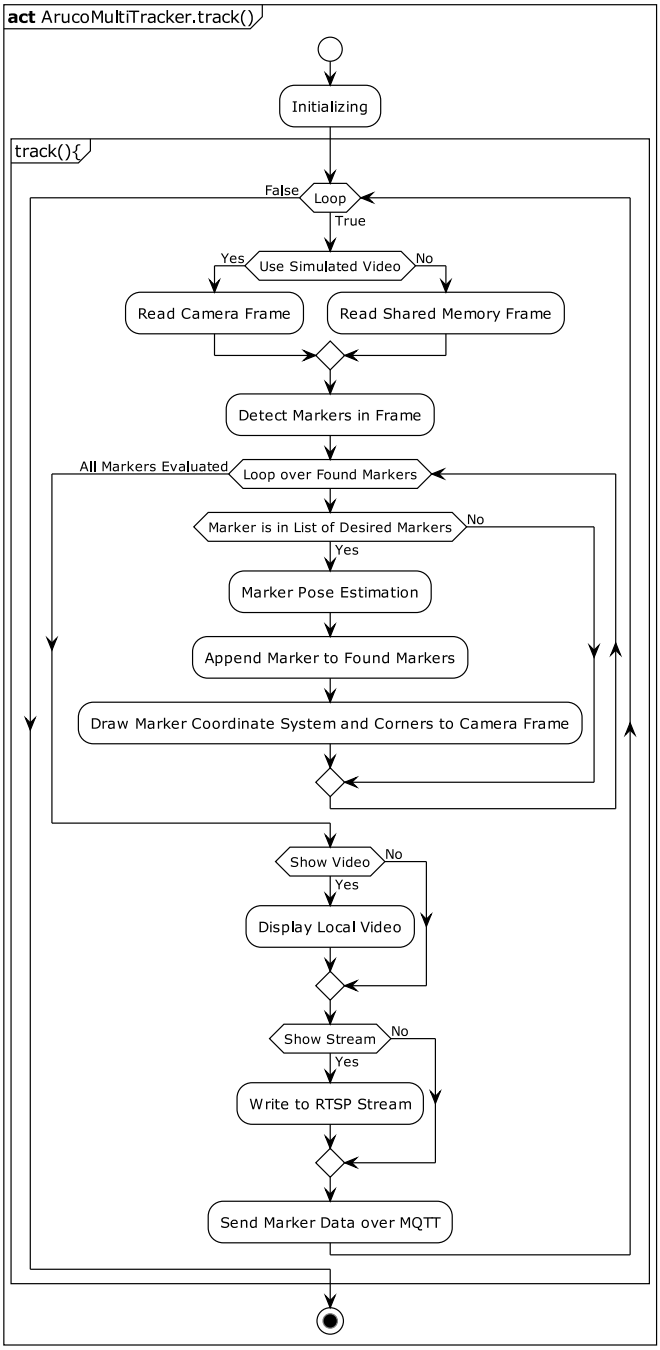


Abbildung 4.2.: Aktivitätsdiagramm *ArucoMultiTracker.track()*

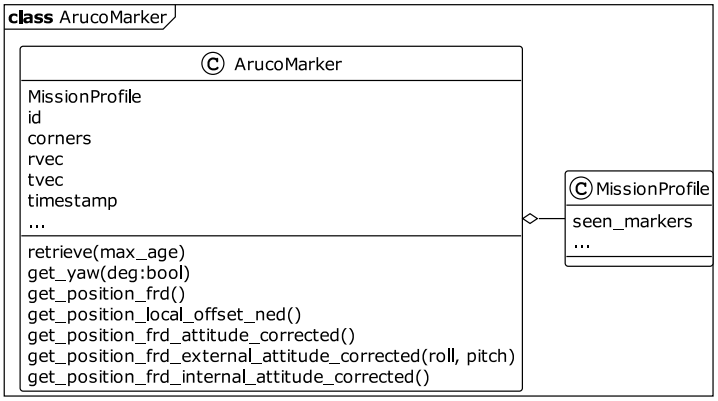


Abbildung 4.3.: Klassendiagramm *ArucoMarker*

4.2.3. Logging

Das Logging bedient sich der standard Python Logging Bibliothek, welche von der Methode *log_data()* gefüllt wird. Die Log-Einträge werden in reinen Text für Statusmeldungen und in JSON formatierte Einträge für das kontinuierliche Loggen veränderlichen Größen unterschieden. Zweitere liegen in Form doppelt verschachtelter JSON Strings vor:
{"TOPIC_NAME":{"data_1": value_1, "data_2": value_2, "data_n": value_n}}
Diese Formatierung erleichtert die grafische Aufbereitung der Daten zum Zweck der Flugauswertung. Der vollständige Code der Methode *log_data()* ist in Code-Ausschnitt 4.4 abgebildet.

4.2.4. Kommunikation der Threads über MQTT

MQTT ist ein TCP/IP Nachrichtenprotokoll, das häufig im Bereich Internet of Things (IoT) zum Einsatz kommt. Es basiert auf dem Publisher-Subscriber Prinzip und kann zum Senden von Daten in einem lokalen Netzwerk oder über das Internet verwendet werden. Es zeichnet sich durch seine leichte Implementierbarkeit und seinen geringen Nachrichten-*Overhead* aus. [28] MQTT kann nicht nur zur Kommunikation zwischen Hardwareinstanzen genutzt werden, sondern ist auch innerhalb einer Software zur Kommunikation zwischen Prozessen einsetzbar. Dies ermöglicht auch, die Prozesse auf mehrere Geräte im Netzwerk zu verteilen, beispielsweise das Auslagern rechenintensiver Programmteile von dem Companion Computer einer Drohne auf einen Leistungsstärkeren PC am Boden.

```
1  def log_data(self, topic=None, names=None,
2               values=None, text=None, debug_level='info'):
3    msg = {}
4    if len(names) != len(values):
5        print('error while logging, lengths do not match')
6    if not isinstance(topic, str) and not topic is None:
7        print('error while logging, topic musst be a string')
8    else:
9        dict = {}
10       for name, value in zip(names, values):
11           dict[name] = round(value, 4)
12       msg[topic] = dict
13       msg = json.dumps(msg)
14   if topic is None and values is None and names is None
15       and text is not None: # just log plain text
16       msg = text
17
18   if debug_level == 'info':
19       self.logger.info(msg)
20   elif debug_level == 'warning':
21       self.logger.warning(msg)
22   elif debug_level == 'critical':
23       self.logger.critical(msg)
24   elif debug_level == 'debug':
25       self.logger.debug(msg)
```

Quellcode 4.4: Code zur Implementierung des Loggings

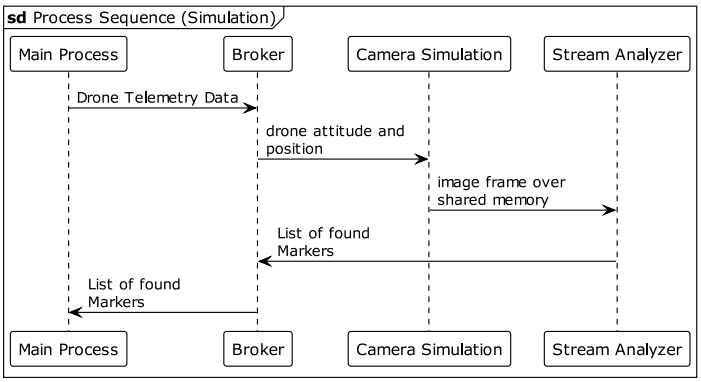


Abbildung 4.4.: Ablaufdiagramm Datenfluss in der Simulation

Das im Rahmen dieser Arbeit verwendete Framework von LHIND verwendet MQTT für die Kommunikation der parallel laufenden Prozesse verwendet. Dazu stehen eine Reihe von Hilfsfunktionen zur Verfügung, die das Einrichten einer MQTT Verbindung erleichtern.

Im Ablaufdiagramm 4.4 ist die Kommunikation zwischen den Prozessen dargestellt. Der Prozess zur Kamerasimulation ist dabei nur aktiv, wenn die Software mit Kamera- und SITL Simulation und nicht auf realer Drohnenhardware läuft.

4.3. Implementierung der Landealgorithmen

Die Implementierung der drei Landeverfahren erfolgte zu Teilen durch Helge Hackbart bei LHIND. Folgende Programmteile wurden nicht oder nicht vollständig von mir programmiert:

- State alignYaw
- State turnAroundScanForMarker
- State finalLanding
- State descendMarker im geschwindigkeitsbasierten Landeverfahren

Alle weiteren, nicht gesondert gekennzeichneten Elemente sind im Rahmen dieser Arbeit entstanden und wurden von mir eigenständig implementiert.

4.3.1. Allgemeine Statemachine Architektur

Für alle drei zu untersuchenden Verfahren wird jeweils eine Statemachine im in Abschnitt 4.2.1 beschriebenen Framework erstellt. Diese kann durch das Argument *-mission* beim Aufruf der *main.py* ausgewählt werden.

Im folgenden Abschnitt werden alle Elemente beschrieben, welche in allen drei Landeverfahren enthalten sind.

State *WaitForRTLorLand*

In diesem State wird zyklisch der Flightmode der Drohne überwacht. Ändert sich dieser in den Modus *LAND* oder *RTL*, wird in den State *waitForMarker* gewechselt.
Dieser State ist immer aktiv, solange sich die Drohne in ihrem normalen Flug befindet. Da hier keine weiteren Instruktionen ausgeführt werden, werden auch wenig Ressourcen verbraucht.

State *waitForMarker*

In diesem State wird zyklisch überprüft ob, die Markererkennung einen der beiden Marker im Kamerabild detektiert hat, und es wird entschieden, ob die Detektion ausreichend stabil ist. Dafür wird bei jeder Detektion eines Markers ein Zähler inkrementiert. Dies geschieht für beide Marker unabhängig. Wird ein Marker in einem Frame nicht erkannt, wird der jeweilige Zähler wieder zurückgesetzt.
Erreicht einer der beiden Zähler den Wert 15, also wurde dieser Marker in 15 aufeinanderfolgenden Kamera-Frames erkannt, wird der Übergang in den nächsten State eingeleitet.

State *alignYaw*

In diesem State wird die Drohne am Marker ausgerichtet. Dafür wird der jeweils sichtbare Marker verwendet. Sind beide sichtbar, wird der kleine Marker favorisiert.
Der Befehl zum Drehen der Drohne wird über den MAVLink Befehl *MAV_CMD_CONDITION_YAW* übermittelt. Diesem wird die Winkeldifferenz zum Marker, die gewünschte Drehrichtung und Drehgeschwindigkeit übergeben. Die Drehrichtung ergibt sich aus dem Vorzeichen des Winkels, die Drehgeschwindigkeit wird in 3 Stufen gesteuert:

- default yaw Drehgeschwindigkeit der Drohne für Winkel $\geq 15°$
- $12\,°/s$ für Winkel $\geq 5°$
- $6\,°/s$ für Winkel $< 5°$

Wenn die Ausrichtung der Drohne mit der des Markers übereinstimmt (Toleranz einstellbar), wird ein Zustandsübergang ausgelöst. In welchen State gewechselt wird, ist abhängig vom verwendeten Landeverfahren.

4.3.2. Implementierung „positionsbasiertes Landen“

Diese Statemachine implementiert das in Abschnitt 3.3.2 beschriebene Verfahren auf der Basis von Positionszielen. Im Folgenden werden alle speziell für dieses Verfahren genutzten States beschrieben.

State *initializing*

Anfangs-State, kann verwendet werden um Parameter der Drohne zu setzen. Wird nach einem Durchlauf wieder verlassen, Zustandsübergang zu *ground*.

State *ground*

Dieser State wird immer dann eingenommen, wenn sich die Drohne am Boden befindet, also nicht *armed* ist. In der aktuellen Implementierung wird kein Code zyklisch ausgeführt.

State *descendMarker*

Hier erfolgt die Implementierung des Landealgorithmus. Bei jedem Programmdurchlauf wird entschieden, welcher Marker verwendet werden soll. Bei einer Flughöhe über $3\,\mathrm{m}$ wird der große Marker präferiert, bei einer Flughöhe unter $3\,\mathrm{m}$ der kleine. Wird der präferierte Marker nicht erkannt, wird, sofern möglich, der jeweils andere Marker verwendet.
Als Markerposition wird die Position relativ zur Drohne im *FRD ATTITUDE CORRECTED* Koordinatensystem verwendet (siehe Abschnitt 2.4). Sofern der große Marker aktiv ist und die *Yaw*-Ausrichtung der Drohne mit der des Markers übereinstimmt, wird die Markerposition noch so korrigiert, dass sie mit der des kleinen Markers übereinstimmt.
Der Drohne wird anschließend die Markerposition mittels der MAVLink Message *SET_POSITION_TARGET_LOCAL_NED* im oben genannten Koordinatensystem übergeben.
Die Flughöhe wird bis zu einem einstellbaren Schwellenwert vom Flightcontroller übernommen und darunter die Höhe aus der Markererkennung verwendet. Diese hat sich in der Entwicklung als die verlässlichere Quelle für die Flughöhe herausgestellt, sofern der Marker sicher im Bild erkannt wird.
Die Sinkgeschwindigkeit leitet sich aus der Flughöhe ab, kann jedoch nicht wie im geschwindigkeitsbasierten Verfahren (siehe Abschnitt 4.3.4) als Sollgeschwindigkeit vorgegeben werden. Dies liegt daran, dass Ardupilot nicht parallel Positions- und Geschwindigkeitsbefehle umsetzt. Daher wird die Sinkgeschwindigkeit nicht direkt gesteuert, sondern als unterschiedlich weit entfernte Position unterhalb der Drohne mit der N und E Position des Markers zusammen übermittelt. Dadurch ist die absolute Sinkgeschwindigkeit abhängig von der Parametrierung

der Drohne, lässt sich jedoch relativ zur Flughöhe anpassen, wodurch ein zuerst schnellerer Abstieg und am Ende der Landung langsamerer Abstieg erreicht wird.

Die Ausrichtung der Drohne wird kontinuierlich gesteuert, der Algorithmus dazu ist dem State *alignYaw* entnommen.

Die Frequenz, mit der die Positionsdaten des Markers an den Flightcontroller gesendet werden, ist einstellbar, jedoch maximal auf einen Wert, der der Durchlaufzeit der Statemachine entspricht.

Zustandsübergänge

- Sobald eine einstellbare Höhe über dem Marker erreicht ist, wechselt die Statemachine in den State *finalLanding*. Als sinnvoll hat sich hier eine Höhe von $40\,\mathrm{cm}$ über dem Marker erwiesen.
- Stimmt die Ausrichtung der Drohne auf ± 10 Grad nicht mit der des Markers überein, wird in den State *alignYaw* gewechselt.
- Wird der Marker über einer Höhe von $5\,\mathrm{m}$ verloren, wird in in den State *turnAroundScanForMarker* gewechselt.
- Unter einer Höhe von $5\,\mathrm{m}$ wird bei Verlust der Marker in den State *waitForMarker* gewechselt und der Flugmodus *RTL* aktiviert, wodurch ein erneuter Landeanflug durchgeführt wird.
- Ist der aktuelle Flugmodus nicht *Guided*, d.h. das Landeverfahren wurde von außen unterbrochen, wechselt die Statemachine in:
  - *waitForMarker*, wenn der Flugmodus *RTL* oder *LAND* ist.
  - *WaitForRTLorLand*, wenn der Flugmodus nicht *RTL* oder *LAND* ist.

State *turnAroundScanForMarker*

Wird der Marker bei höheren Windgeschwindigkeiten verloren, kann es helfen, die Drohne auf der Stelle um 360° zu drehen.
In diesem State wird dies durchgeführt. Wird der Marker innerhalb der für eine Umdrehung benötigten Zeit gefunden, erfolgt ein Übergang in den State *descendMarker*. Wird kein Marker gefunden, erfolgt ein Übergang in den State *waitForMarker* und es wird in den *RTL* Flugmodus geschaltet, wodurch ein erneuter Anflug ausgelöst wird.

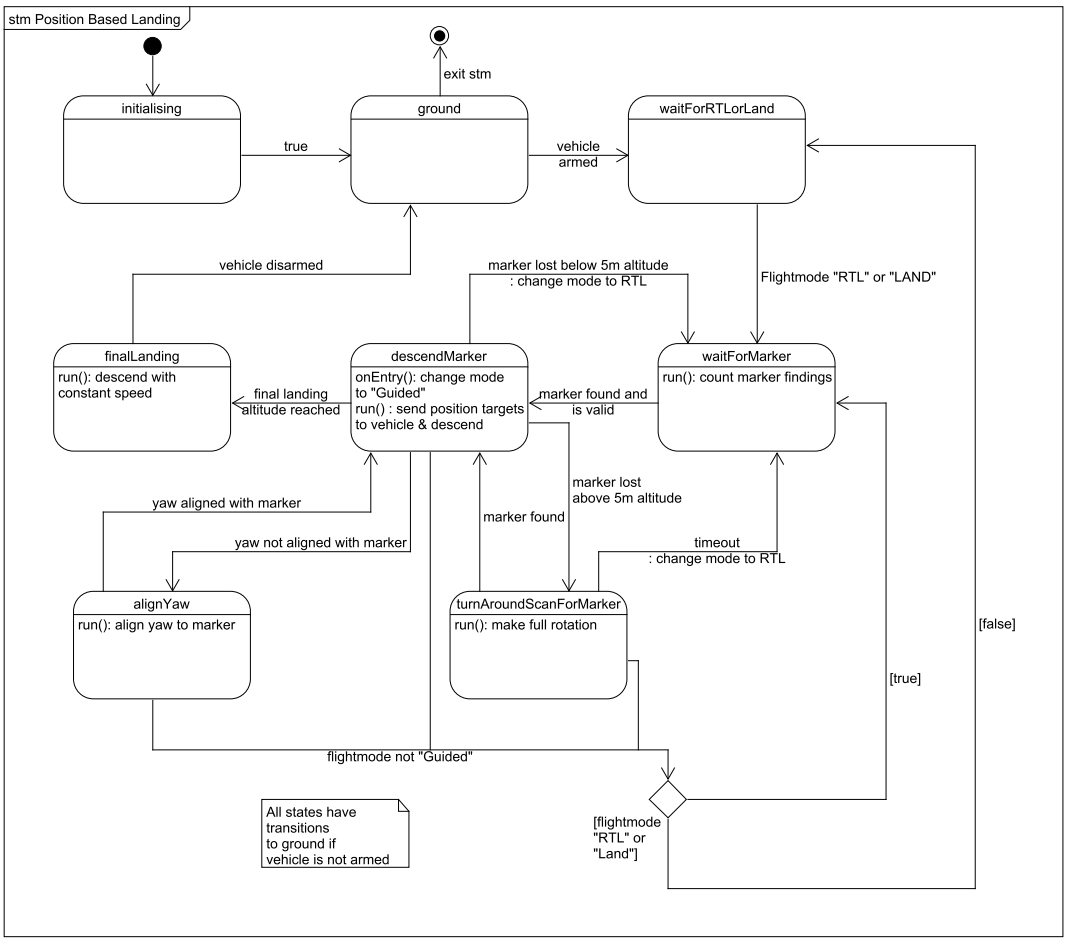


Abbildung 4.5.: Statemachine zum positionsbasierten Landealgorithmus

State *finalLanding*

In diesem State wird mit konstanter Sinkgeschwindigkeit geflogen, bis die Drohne gelandet ist. Die Sinkgeschwindigkeit ist einstellbar; als sinnvoll hat sich eine Geschwindigkeit von $40\,\mathrm{cm/s}$ herausgestellt.
Ardupilot detektiert selbständig ein Aufsetzen auf dem Boden und *disarmed* die Drohne. Als zusätzliche Sicherheit ist ein *Timeout* integriert, der nach einer bestimmten Zeit in den *LAND* Flugmodus schaltet. Die Länge des Timeouts berechnet sich aus der Flughöhe und der Landegeschwindigkeit, sodass sie dem Vierfachen der theoretisch zum Landen benötigten Zeit entspricht.
Sobald die Drohne nicht mehr *armed* ist, wird in den State *ground* gewechselt.
Alle States, mit Ausnahme von *initializing* und *ground* verfügen über einen Zustandsübergang zum State *ground*, sobald die Drohne nicht mehr *armed* ist. Eine Übersicht aller States und Zustandsübergänge finden sich in Abbildung 4.5.

4.3.3. Implementierung „Ardupilot Precision Landing Verfahren“

Diese Statemachine implementiert das in Abschnitt 3.3.3 beschriebene Verfahren zum präzisen Landen mithilfe des Ardupilot eigenen *Precision Landing* Algorithmus. Im Folgenden werden alle speziell für dieses Verfahren genutzten States beschrieben. Eine Übersicht über das Verfahren ist in Abbildung 4.6 zu sehen.

State *landing*

Dieser State implementiert das oben beschriebene Landeverfahren. Die Auswahl des verwendeten Markers erfolgt dabei analog zum positionsbasierten Landeverfahren (State *descendMarker*). Die Markerkoordinaten werden im Front-Right-Down (FRD) Koordinatensystem (siehe 2.4, S. 14), also relativ zur Kamera, verarbeitet.
Wird ein Marker erkannt, wird seine Postion über die MAVLink Message *LANDING_TARGET*, wie im Abschnitt 2.3.2, S. 12 beschrieben, gesendet. Die Distanz zum Marker wird aus den FRD Koordinaten mittels Pythagoras Theorem berechnet. Die Art des Markers wird als *Fiducial Marker* gesetzt.
Zusätzlich wird die Ausrichtung der Drohne über die Funktion *align_yaw_to_marker()* gesteuert, welche analog zum Verfahren in State *alignYaw* funktioniert.

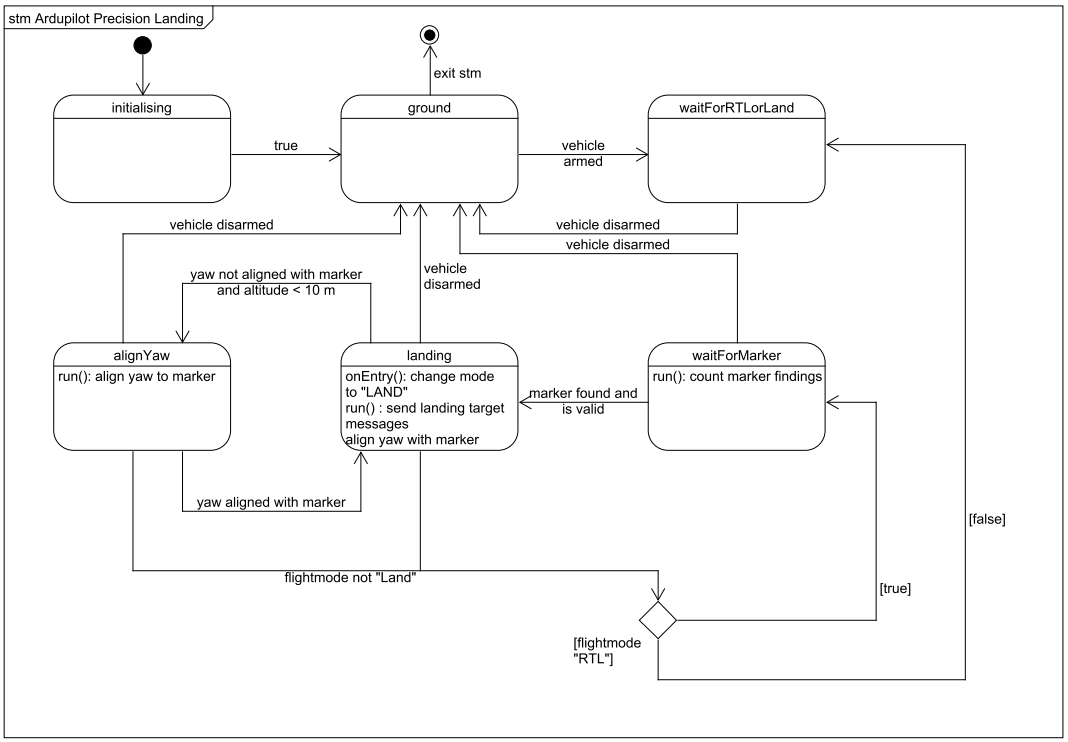


Abbildung 4.6.: Statemachine zum Ardupilot Precision Landing Landealgorithmus

Zustandsübergänge

- Stimmt die Ausrichtung der Drohne mit der des Markers nicht überein ($\pm$ 10°), wird unter einer Flughöhe von 10 m in den State *alignYaw* gewechselt.
- Ist die Drohne nicht *armed*, wird in den State *ground* gewechselt.
- Ist der aktuelle Flugmodus nicht *LAND*, d.h. das Landeverfahren wurde von außen unterbrochen, wechselt die Statemachine in:
  - *waitForMarker*, wenn der Flugmodus *RTL* ist
  - *WaitForRTLorLand*, wenn der Flugmodus nicht *RTL* ist

Wie auch im ersten Landeverfahren verfügen alle States, mit Ausnahme von *initializing* und *ground*, über einen Zustandsübergang zum State *ground*, sobald die Drohne nicht mehr *armed* ist.

Parametrierung Ardupilot

Auf Ardupilot Seite wurden die Parameter wie folgt gewählt:

Parameter	Wert	Bedeutung
PLND_ENABLE	1	aktiviert
PLND_TYPE	1	Companion Computer
PLND_EST_TYPE	0	*RawSensor*
PLND_LAG	0,066	66 ms Signalverzögerung
PLND_STRICT	1	Landung Wiederholen
PLND_RET_BEHAVE	1	Bei Verlust des Markes in Richtung der letzten erkannten Markerposition fliegen
PLND_ALT_MIN	0,75	Höhe unter der bei Verlust des Markers direkt gelandet wird, kein neuer Anflug

In Tests zeigte sich die Filterung des Signals mittels des Ardupilot Kalman Filters (*PLND_EST_TYPE*) als nicht sinnvoll, da die Drohne merkbar unruhiger über dem Marker schwebte. Die Signalverzögerung (vor Übermittlung an Ardupilot) wurde zu 66 ms bestimmt (siehe Abschnitt 5.2.1). Die Parameter *PLND_STRICT*, *PLND_RET_BEHAVE* und *PLND_ALT_MIN* wurden so eingestellt, dass die Drohne bei Verlust des Markers die Landung wiederholen und in Richtung der letzten bekannten Markerposition fliegen soll, sofern die Flughöhe noch größer als 0,75 m ist. Andernfalls wird an der aktuellen Position gelandet.

4.3.4. Implementierung „geschwindigkeitsbasiertes Landen“

Diese Statemachine implementiert das in Abschnitt 3.3.4 beschriebene Verfahren zum präzisen Landen mithilfe eines eigenen Regelkreises auf dem Companion Computer. Die Struktur der Statemachine dieses Verfahrens entspricht der des positionsbasierten Algorithmus, weshalb hier nicht weiter auf die einzelnen States eingegangen wird. Lediglich der State *ground* fehlt hier, was jedoch keine Auswirkungen auf das Landen an sich hat. Der für das Verfahren relevante Code befindet sich im State *descendMarker*, worauf im Folgenden weiter eingegangen wird. Eine Übersicht über die Statemachine dieses Verfahrens ist in Abbildung 4.6 zu sehen.

Implementierung des Regelkereises

Gleich dem positionsbasierten Verfahren, wird die Markerposition relativ zur Drohne im *FRD ATTITUDE CORRECTED* Koordinatensystem verwendet (siehe Abschnitt 2.4). Wie auch in den anderen Verfahren, wird bei jedem Durchlauf des Codes entschieden, welcher Marker für die

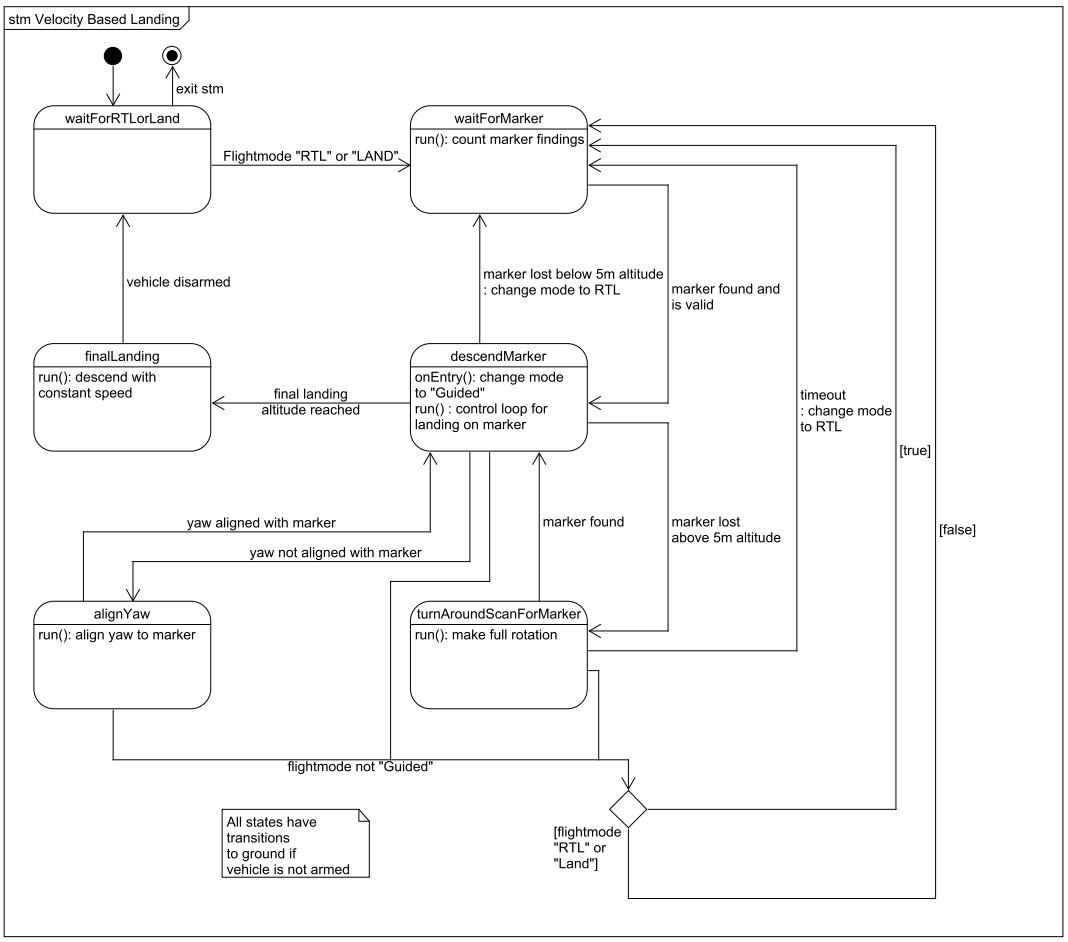


Abbildung 4.7.: Statemachine zum geschwindigkeitsbasierten Landealgorithmus

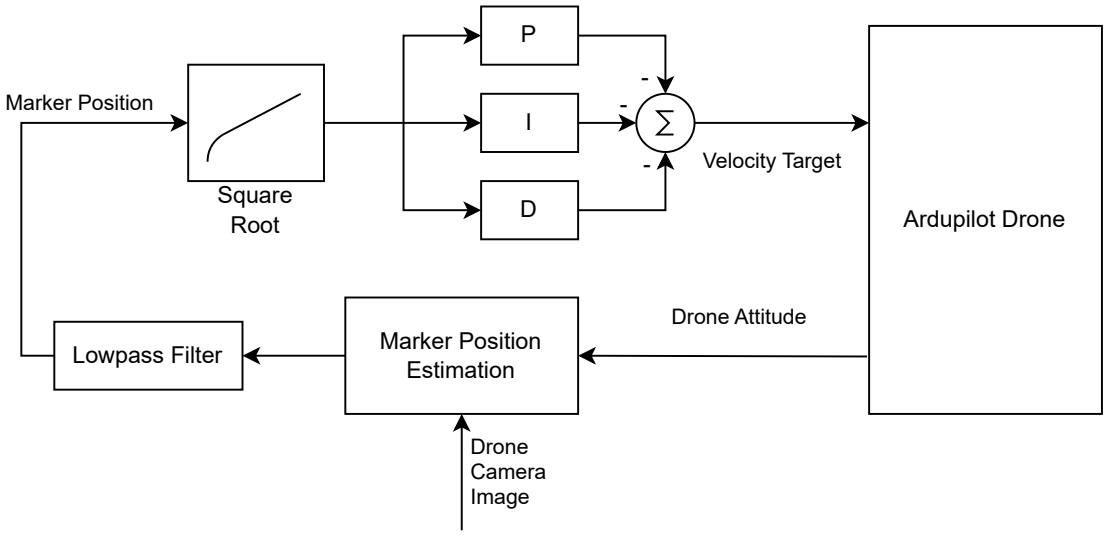


Abbildung 4.8.: Schematische Darstellung des Regelkreises für das geschwindigkeitsbasierte Landeverfahren

Landeregelung verwendet werden soll. Hier wird immer der kleine Marker bevorzugt, solange dieser im Bild ist und erkannt wird. Der große Marker wird nur verwendet, wenn der kleine nicht sichtbar ist. Stimmt die Ausrichtung der Drohne mit der der Marker um $\pm\ 7{,}5°$ überein, wird die Position des großen Markers so korrigiert, dass sie der des kleinen entspricht. Der Abstieg erfolgt dann über dem kleinen Marker, selbst wenn dieser noch nicht detektierbar ist. Im nächsten Schritt wird die Markerposition durch einen Tiefpassfilter in Form eines einfachen exponentiell gleitenden Mittelwerts gefiltert. Anschließend folgt der PID Regler. Da die anzufliegende Position immer die Position des Markers ist, ist der *Setpoint* der Regelung immer null. Das bedeutet, dass die Regelabweichung (Abstand zum Marker) direkt als *Error* in die Regelung eingeht.
Im ersten schritt wird die Markerposition für Werte unter $1\,\mathrm{m}$ verändert, indem statt der Position die Wurzel berechnet und weitergegeben wird.

$$\text{Markerposition} = \begin{cases} \text{Markerposition} & \text{für } x \geq 1 \\ \sqrt{\text{Markerposititon}} & \text{für } x < 1 \end{cases} \tag{4.3}$$

Dies führt zu einer Anhebung der Werte ¡ 1 und damit zu einer Verstärkung der Regelung für kleine Abweichungen vom Marker. Zusätzlich wird bei Nulldurchgängen der Markerposition der I-Wert der Regelung zurückgesetzt, um das Problem des *Integral Windups* zu begrenzen.

Anschließend werden die proportional (P), integral (I) und differential (D) Komponenten berechnet, mit den entsprechenden Faktoren multipliziert, invertiert und als neue Sollgeschwindigkeit an den Flightcontroller übermittelt.
Dieser Regelkreis ist separat für die F und R Achse definiert. Für die Sinkgeschwindigkeit ist kein geschlossener Regelkreis implementiert. Diese wird wie in den anderen Verfahren, stufenweise als Sollgröße dem Flightcontroller vorgegeben. Die Regelung der Ausrichtung der Drohne erfolgt wie bei den anderen beiden Landeverfahren.
Die Ermittlung der PID-Parameter erfolgte experimentell während der Implementierung an der simulierten Drohne und anschließend in Flugtests an der realen Drohne.
Als sinnvoll haben sich folgende Werte ergeben:

Parameter	Wert
Proportionalfaktor	0,13
Integralfaktor	0,015
Differentialfaktor	0,13

Der Regelkreis für das geschwindigkeitsbasierte Verfahren ist in Abbildung 4.8 dargestellt.

5. Evaluation

5.1. Evaluation mittels SITL Simulation

5.1.1. Aufbau und Durchführung

Dieser Flugversuch erfolgt in der zur Entwicklung der drei Landeverfahren genutzten SITL Simulation (siehe Abschnitt 2.5). In Abbildung 5.1 ist ein Bildschirmfoto einer solchen Landung zu sehen.
Für jedes der drei Verfahren werden ca 1.000 Flüge simuliert. Dabei wird der Wind zwischen $0\,\mathrm{m/s}$ und $10\,\mathrm{m/s}$ in Schritten von $1\,\mathrm{m/s}$ zufällig variiert. Die Windrichtung wird von $0°$ bis $360°$ in Schritten von $1°$ zufällig variiert. Die Verteilung von Windrichtung und Windgeschwindigkeit im simulierten Datensatz ist in Abbildung 5.2 dargestellt.
Der simulierte Marker, auf dem gelandet wird, ist im Ursprung des NED Koordinatensystem platziert und $90°$ im Uhrzeigersinn gegen Norden verdreht.
Zur Umrechnung in das Marker-Koordinatensystem wird jeder Datenpunkt im Drohnen NED System über folgenden Zusammenhang umgerechnet:

$$X = -N \tag{5.1}$$

$$Y = E \tag{5.2}$$

$$Ausrichtung_{NED} = Ausrichtung_{Marker} - 90° \tag{5.3}$$

Geflogen wird auf $20\,\mathrm{m}$ Höhe zu einem ca. $10\,\mathrm{m}$ entfernten Punkt. Dieser ist so gelegt, dass die Drohne dabei ihre Ausrichtung um $90°$ ändert. Dort angekommen wird in den *RTL* Flugmodus gewechselt, wodurch die Drohne auf $25\,\mathrm{m}$ steigt und anschließend zum Home-Punkt zurück fliegt. Dort begibt sie sich in einen Sinkflug. Sobald ein Marker erkannt wird, übernimmt das jeweilige Landeverfahren.

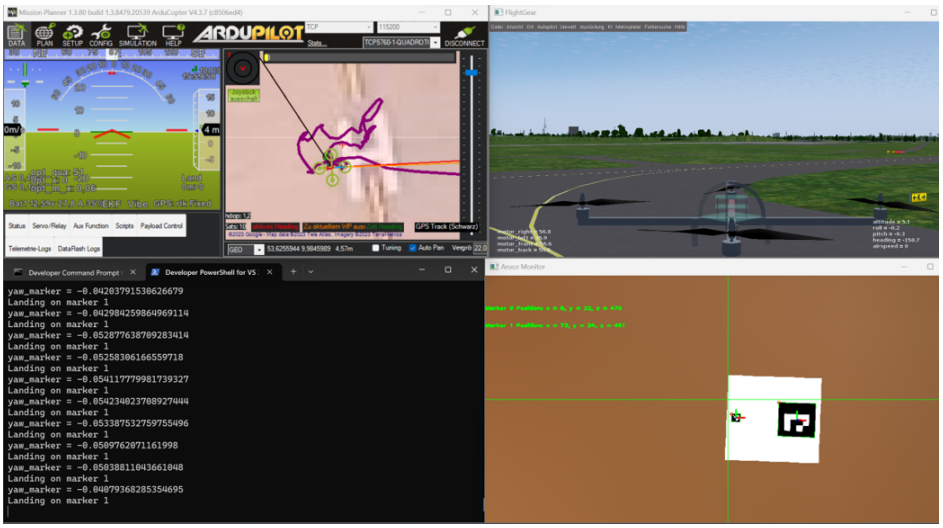
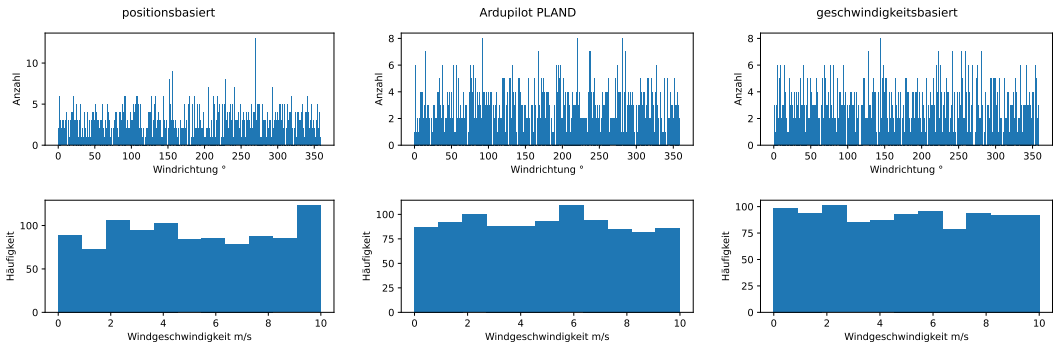


Abbildung 5.1.: Bildschirmfoto Flugerprobung SITL Simulation

5.1.2. Ergebnisse

In Abbildung 5.3 sind alle Landungen relativ zum Marker abgebildet. Zur Bewertung der mittleren Genauigkeit, wird für jede Landung der absolute Abstand der Kamera zum Mittelpunkt des kleinen Markers mittels Pythagoras-Theorem berechnet. Über alle Flüge gemittelt ergibt dies die mittlere absolute Abweichung vom Marker.
Das positionsbasierte Verfahren (1) schneidet hier mit einer mittleren Abweichung von $2{,}71\,\mathrm{cm}$ am besten ab. Auch das geschwindigkeitsbasierte Verfahren (3) erreicht mit $2{,}78\,\mathrm{cm}$ Abweichung einen ähnlich guten Wert. Das Ardupilot Precision Landing Verfahren (2) hat die geringste Genauigkeit mit einer mittleren Abweichung von $5{,}19\,\mathrm{cm}$.
Die Streuung der Landungen ist bei Verfahren 3 am geringsten (Standardabweichung $1{,}25\,\mathrm{cm}$) und bei Verfahren 2 am höchsten (Standardabweichung $9{,}93\,\mathrm{cm}$). In Abbildung A.2 ist die Verteilung der Landungen dargestellt. Auffällig ist hier die Cluster-Bildung an bestimmten Positionen, die bei dem positionsbasierten und geschwindigkeitsbasierten Verfahren auftritt. Diese ist vermutlich auf die Simulation zurückzuführen und nicht auf die Funktionsweise der Verfahren in der Realität übertragbar.
Die mittlere absolute Winkelgenauigkeit ist bei allen Verfahren sehr hoch mit einer maximalen Abweichung der Ausrichtung von $1{,}19°$ beim positionsbasierten Verfahren und einer Abweichung von $0{,}85°$ beim Verfahren Ardupilot Precision Landing. Dieses hat jedoch auch die höchste Standardabweichung von $2{,}23°$. Das geschwindigkeitsbasierte Verfahren schneidet hier mit einem Wert von $0{,}96°$ am besten ab. In Abbildung A.1 ist die Verteilung der Landeaus-



(a) *(b)* *(c)*

Abbildung 5.2.: Verteilung von Windrichtung (oben) und Windgeschwindigkeit (unten) in den simulierten Daten. (a) positionsbasiertes Verfahren, (b) Ardupilot Precision Landing Verfahren, (c) geschwindigkeitsbasiertes Verfahren

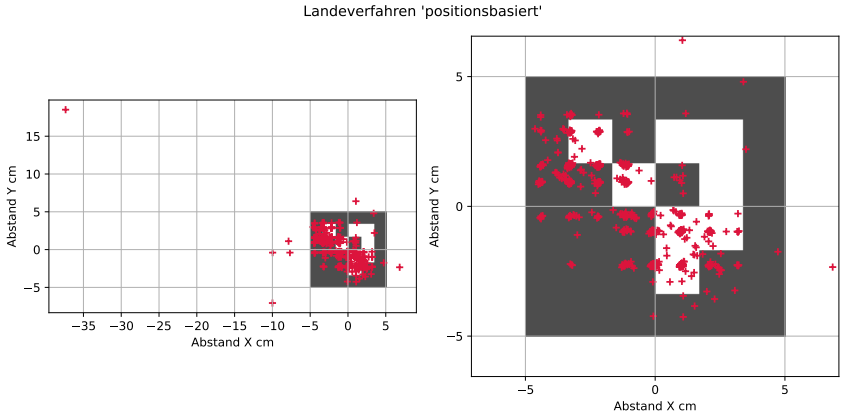
richtungen dargestellt.

Alle drei Verfahren sind bis $10\,\mathrm{m/s}$ Windgeschwindigkeit funktionsfähig. Das Verfahren 2 zeigt als einziges eine deutliche Korrelation zwischen Windgeschwindigkeit und Landegenauigkeit (r = $0{,}91$). Hier ist zu beobachten, dass die Streuung der Landungen für Windgeschwindigkeiten $> 7\,\mathrm{m/s}$ massiv ansteigt (siehe Abbildung 5.4 (b), links). Auch zeigt sich hier ein Zusammenhang zwischen Windrichtung und der Richtung, in welche die Landung vom Sollpunkt versetzt erfolgt (Landerichtung), in Form von zwei Peaks bei $70°$ und $270°$ (siehe Abbildung 5.4 (b), rechts).

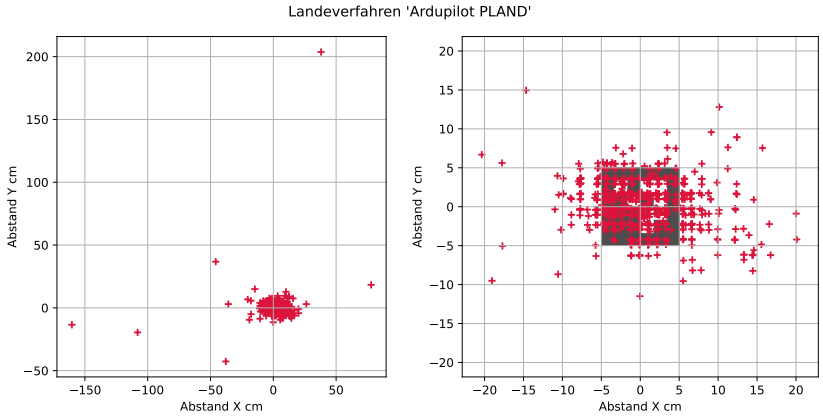
Die beiden anderen Verfahren zeigen weder eine signifikante Korrelation zwischen Windrichtung und Landerichtung, dem mittleren Abstand der Marker zum Soll-Landepunkt und der Windgeschwindigkeit, noch einen Zusammenhang der Streuung der Landungen und der Windgeschwindigkeit. Der Peak in der Streuung der Landungen für das Verfahren 1 bei $0\,\mathrm{m/s}$ Windgeschwindigkeit (Siehe Abbildung 5.4 (a), links) lässt sich durch die einzelne Landung bei (-37,5 18,5) cm erklären.

Insgesamt erreichen alle drei Verfahren eine hohe Quote für erfolgreiche Landungen. Am schlechtesten schneidet das positionsbasierte Verfahren mit einem Wert von $99{,}01\,\%$ ab. Das beste Ergebnis erreicht das geschwindigkeitsbasierte Verfahren mit einer Quote von $100\,\%$ erfolgreichen Landungen.

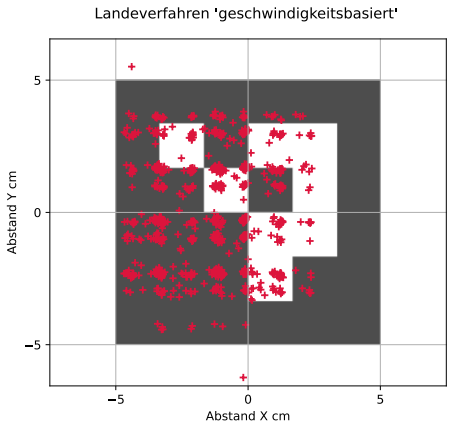
In Tabelle 5.1 sind die Ergebnisse der simulierten Flugversuche dargestellt.



(a)

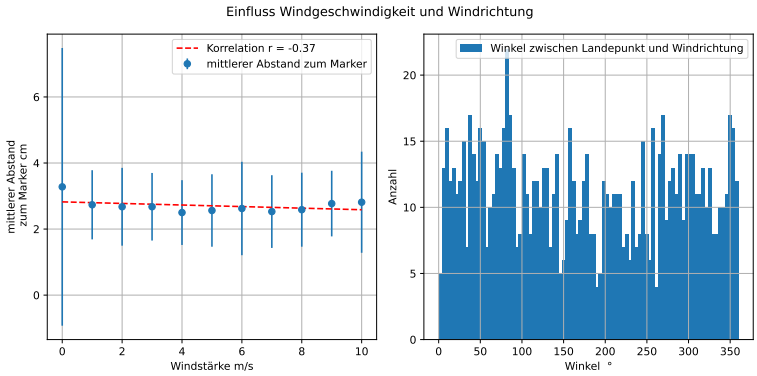


(b)

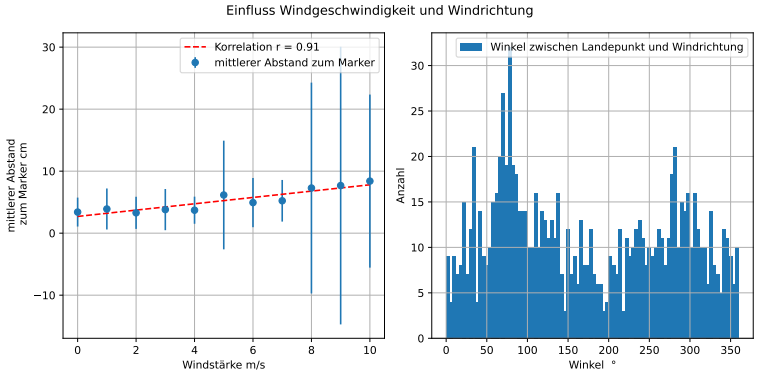


(c)

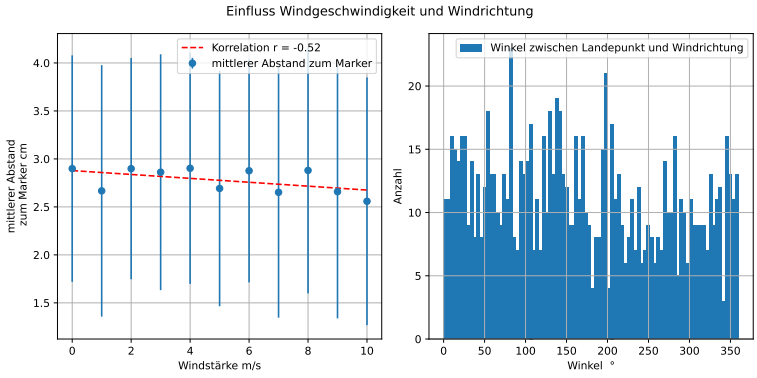
Abbildung 5.3.: Darstellung der Landungen der simulierten Flüge, für (a) und (b) jeweils gesamt (links) und Ausschnitt um den Marker (rechts)
(a) positionsbasiert, (b) Ardupilot Precision Landing, (c) geschwindigkeitsbasiert



(a)



(b)



(c)

Abbildung 5.4.: Windeinfluss bei den simulierten Flügen, (a) positionsbasiert, (b) Ardupilot Precision Landing, (c) geschwindigkeitsbasiert
links: Zusammenhang zwischen Windgeschwindigkeit und dem mittleren Abstand zum Soll-Landepunkt. Fehlerbalken: Standardabweichung für jeweilige Windgeschwindigkeit
rechts: Histogramm der Winkeldifferenz zwischen Windrichtung und Richtung, in welche die Landung vom Soll-Punkt versetzt ist.

	Einheit	positionsbasiert	Ardupilot PLAND	geschwindigkeitsbasiert
Anzahl Flüge		1.004	1.011	1.012
mittlere Position in X	cm	-1,09	-0,20	-1,36
mittlere Position in Y	cm	-0,08	0,29	0,003
mittlere absolute Abweichung	cm	2,71	5,19	2,78
mittlere Abweichung Ausrichtung	°	0,06	-0,49	0,55
mittlere Abweichung absolute Ausrichtung	°	1,19	0,85	0,91
Standardabweichung Landeposition	cm	1,69	9,93	1,25
Standardabweichung Ausrichtung	°	1,65	2,23	0,96
Quote erfolgreiche Landung Position	%	99,90	99,30	100
Quote erfolgreiche Landung Winkel	%	99,12	99,80	100
Quote erfolgreiche Landung	%	99,01	99,10	100

Tabelle 5.1.: Ergebnisse der Evaluation mittels Simulation

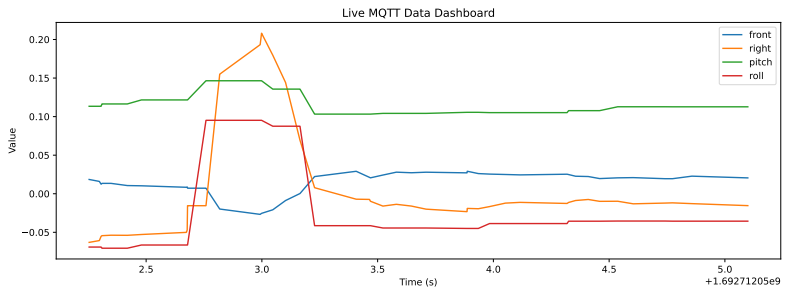


Abbildung 5.5.: Abschätzung der Verzögerung zwischen Lagedaten der Drohne und Markerposition aus Bilderkennung

5.2. Evaluation mittels realer Flugversuche

5.2.1. Verzögerung Markererkennung ermitteln

Zur Parametrierung der Landealgorithmen ist eine Abschätzung der Verzögerung durch die Markererkennung und sonstige Datenverarbeitung notwendig. Zu diesem Zweck wurde die Ausrichtung der Drohne und die Position des Markers im Kamerakoordinatensystem aufgezeichnet. Die Drohne wurde per Hand über dem Marker gehalten und eine schnelle Rollbewegung um die Längsachse ohne Änderung der Drohnenposition ausgeführt. Anschließend wurde mittels des in Abbildung 5.5 dargestellten Diagramms die Verzögerung zwischen den Lagedaten der Drohne und den Daten aus der Bilderkennung mit $66\,\mathrm{ms}$ ermittelt.

5.2.2. Flugevaluation im Innenbereich

Diese Flüge dienen der Evaluation der drei zu untersuchenden Verfahren unter kontrollierten Bedingungen mit der realen Hardware.

Aufbau und Durchführung

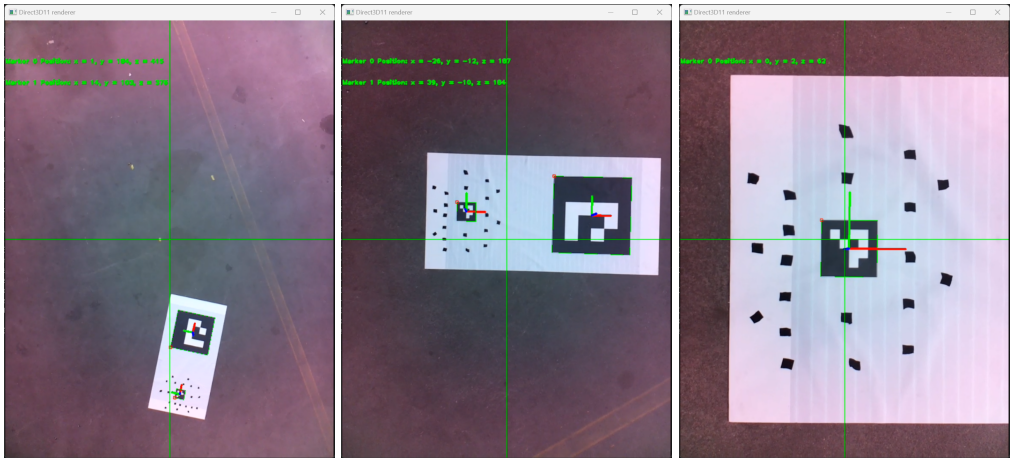
Der Flugversuch findet in einer Lagerhalle auf einer Fläche von ca. $10\,\mathrm{m}$ x $10\,\mathrm{m}$ statt. Die Deckenhöhe beträgt ca. $7\,\mathrm{m}$. Für jeden Flug wird die Drohne auf eine Höhe von $5\,\mathrm{m}$ geflogen, $90°$ im Uhrzeigersinn gedreht und anschließend $2\,\mathrm{m}$ nach vorne geflogen. Zum Starten der Landung wird in den *LAND* Flugmodus geschaltet, danach übernimmt das jeweilige Landeverfahren. Nach dem Aufsetzen auf dem Boden wird der Abstand der Kamera zum Mittelpunkt des kleinen Markers in den Markerkoordinaten gemessen. Der Winkel der Landung, also die Ausrichtung der Drohne relativ zum Marker, kann aufgrund des fehlenden Kompasses nicht

Abbildung 5.6.: Aufbau Flugversuch im Innenbereich

genau bestimmt werden. Es wird aber eine optische Kontrolle durchgeführt und große Abweichungen vom Soll-Winkel werden notiert. Für jedes der drei zu untersuchenden Verfahren werden 20 Flüge durchgeführt.
Zum Einsatz kommt die für Tests in Innenräumen gebaute Drohne (siehe Abschnitt 2.6).
In Abbildung 5.6 ist der Aufbau zu sehen und in Abbildung 5.7 Screenshots aus dem Kamera-Stream der Landekamera zu verschiedenen Phasen der Landung.

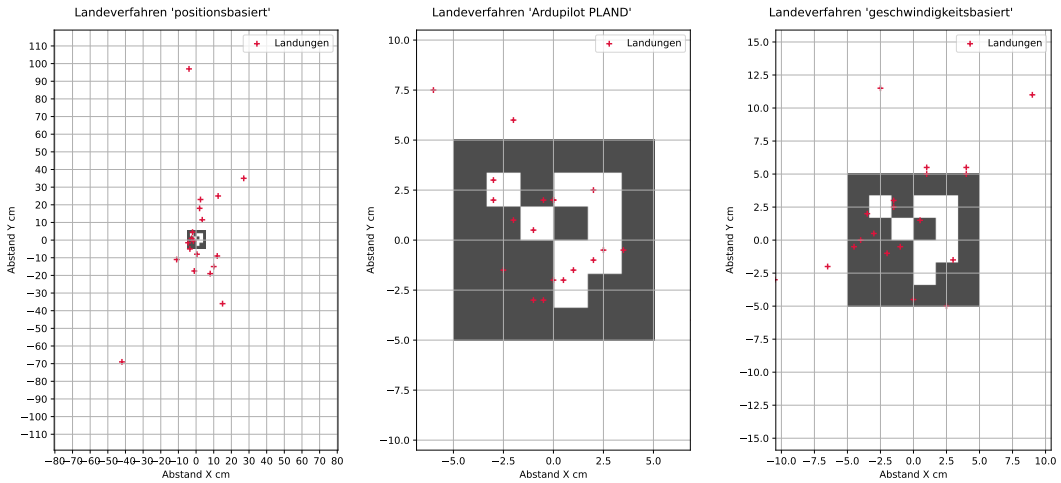
Ergebnisse

Alle 60 Flüge konnten ohne Eingreifen des Piloten durchgeführt werden, und die Marker wurden bei keinem Flug verloren. Die Ausrichtung der Drohne zum Marker war bei allen Flügen kleiner als die erlaubte maximale Ausrichtung von $\pm\ 5°$. In Abbildung 5.8 sind die Positionen der Landungen (Position der Kamera) relativ zum Marker für alle drei Verfahren dargestellt.
Am besten schneidet das Ardupilot Precision Landing Verfahren (2) mit einer mittleren absoluten Abweichung vom Landepunkt von $3{,}39\,\mathrm{cm}$, gefolgt vom geschwindigkeitsbasierten Verfahren (3) mit einer Abweichung von $5{,}39\,\mathrm{cm}$ ab. Das positionsbasierte Verfahren (1) erreicht hier lediglich einen Wert von $28{,}98\,\mathrm{cm}$.
Auch bei der Standardabweichung liegt das Verfahren 2 mit einem Wert von $2{,}13\,\mathrm{cm}$ vor dem Verfahren 3 ($3{,}33\,\mathrm{cm}$) und dem Verfahren 1 ($32{,}5\,\mathrm{cm}$). Dies bedeutet für die Verfahren 2 und 3 in einer Quote von $100\,\%$ erfolgreiche Landungen. Das Verfahren 1 kommt nur auf eine Quote von $75\,\%$.
In Tabelle 5.2 finden sich die Ergebnisse der Flugversuche im Innenbereich.



(a) (b) (c)

Abbildung 5.7.: Screenshots aus dem Kamera-Stream der Landekamera zu verschiedenen Phasen der Landung



(a) (b) (c)

Abbildung 5.8.: Darstellung der Landungen für den Flugversuch im Innenbereich
(a) positionsbasiertes Landeverfahren, (b) Ardupilot Precision Landing Verfahren, (c) geschwindigkeitsbasiertes Landeverfahren

	Einheit	positionsbasiert	Ardupilot PLAND	geschwindigkeitsbasiert
Anzahl Flüge		20	20	20
mittlere Position in X	cm	0,45	-0,15	-0,78
mittlere Position in Y	cm	-4,93	0,48	1,75
mittlere absolute Abweichung	cm	28,98	3,39	5,39
Standardabweichung Landeposition	cm	32,5	2,13	3,33
Quote erfolgreiche Landung	%	75	100	100

Tabelle 5.2.: Ergebnisse der Evaluation im Innenbereich

5.2.3. Flugevaluation im Außenbereich

Bei dieser Versuchsreihe liegt der Fokus auf der Landung unter Realbedingungen. Besonders die Faktoren Wind und ein Anflug aus größerer Höhe, welche im Innenbereich nicht evaluiert werden können, sollen so getestet werden.

Aufbau und Durchführung

Die Auslegung der Marker ist in 5.9 zu sehen. In $1\,\mathrm{m}$ Höhe ist ein Windmesser aufgestellt. Für jeden Flug wird die Drohne manuell gestartet und mindestens 10 Meter weit von den Markern weg geflogen. Dort wird der *RTL* Modus aktiviert, wodurch die Drohne auf $25\,\mathrm{m}$ Höhe steigt und anschließend zurück zum Startpunkt fliegt (siehe Abschnitt 2.3.1, S. 10). Während des Abstiegs übernimmt dann das jeweilige Landeverfahren. Die Flüge finden auf einer freien Wiese statt. Das Wetter ist sonnig bis halbschattig, die Temperatur beträgt $25\,^{\circ}\mathrm{C}$ und der Wind ist böig mit Geschwindigkeiten bis $7\,\mathrm{m/s}$. Nach jeder Landung werden folgende Parameter dokumentiert:

- Uhrzeit
- Information über etwaiges Eingreifen des Piloten bei der Landung
- Abstand der Kamera zur Mitte des kleinen Markers
- Ausrichtung der Drohne durch Auslesen des integrierten Kompasses
- Windgeschwindigkeit bei der Landung
- Windrichtung mittels externem Kompass
- Fluglogs (Ardupilot und eigene)

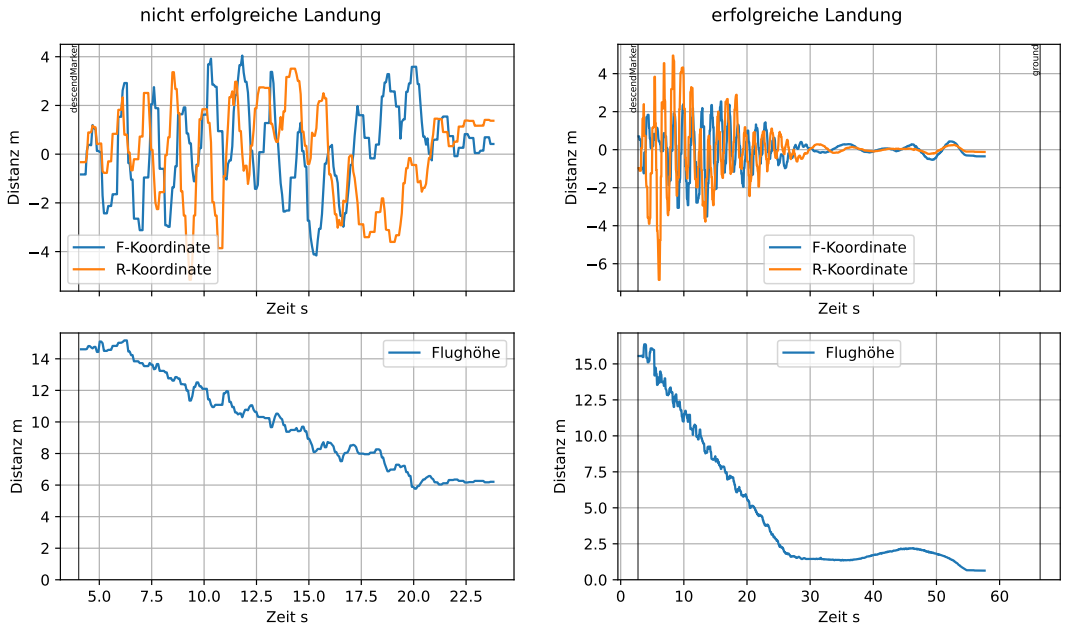
Abbildung 5.9.: Aufbau Flugversuch im Außenbereich

	Einheit	Flug 1	Flug 2
Landeposition X	cm	−1,5	11,5
Landeposition Y	cm	−3	18,5
Abstand zur Markermitte	cm	3,35	21,78
Ausrichtung zum Marker	°	1	−5
Windgeschwindigkeiten	m/s	3,5	1,5

Tabelle 5.3.: Ergebnisse der Evaluation im Außenbereich für das positionsbasierte Verfahren

Ergebnisse positionsbasiertes Verfahren

Für das positionsbasierte Verfahren (1) konnten nur vier Flüge durchgeführt werden, wovon nur zwei zu einer erfolgreichen Landung führten. Nach diesen Flügen wurde der Test abgebrochen, da die Anflüge auf den Marker zu unsicher und zeitintensiv waren.
In Abbildung 5.10 sind die FRD-Koordinaten der Drohne während des Anflugs auf den Marker für einen nicht erfolgreichen (links) und einen erfolgreichen Flug dargestellt. Zu beobachten ist bei größeren Flughöhen ein massives Aufschwingen und Überschießen über den Marker, was zu einem Verlust des Markers führen kann. Mit sinkender Flughöhe verringert sich dieser Effekt, wie in Abbildung 5.10 (b), oben, zu sehen. Ab einer Höhe von unter 2 m ist kein Aufschwingen mehr zu beobachten.
Die Messwerte der beiden erfolgreichen Flüge sind in Tabelle 5.3 dargestellt.



(a) *(b)*

Abbildung 5.10.: F und R Koordinaten (oben) sowie Flughöhe (unten) für eine nicht erfolgreiche (a) und einer erfolgreiche (b) Landung mit dem positionsbasierten Verfahren

Ergebnisse Ardupilot Precision Landing Verfahren

Für das zweite Verfahren wurden 3 Flüge durchgeführt, von denen zwei zu einer erfolgreichen Landung führten. Danach wurde auch diese Testreihe beendet, da ähnlich wie beim ersten Verfahren, die Flüge sehr zeitintensiv waren und sich die Drohne beim Landeanflug stark aufschwingt. In Abbildung 5.11 sind ein nicht erfolgreicher Flug (a) und ein erfolgreicher Flug (b) dargestellt.

Bei beiden Flügen schwingt die Position der Drohne mit einer Amplitude von ca. $4\,\mathrm{m}$ für beide horizontale Achsen. Bei dem nicht erfolgreichen Flug ist zu sehen, wie die Drohne auf einer Höhe von $10\,\mathrm{m}$ in den State *alignYaw* wechselt. Da die Datenaufzeichnung nur im State *landing* aktiv war, fehlen hier die FRD-Koordinatenwerte. Nachdem die Drohne ausgerichtet ist, setzt sie ihren Sinkflug fort, verliert den Marker jedoch auf einer Höhe von ca. $8\,\mathrm{m}$. Auch bei der erfolgreichen Landung wird der Marker zwischenzeitlich verloren (siehe Abb. 5.11 a, ab ca. $20\,\mathrm{s}$ bis $25\,\mathrm{s}$), jedoch reagiert das Ardupilot Landeverfahren so, dass der Marker wieder im Kamerabild erscheint. Mit sinkender Höhe nehmen die Oszillationen ab, sind jedoch erst ab ca $2{,}5\,\mathrm{m}$ Höhe verschwunden. Die Landung erfolgt an der Koordinate ($8\,\mathrm{cm}$, $2\,\mathrm{cm}$) mit einer

Winkelabweichung vom Marker von 3° und ist damit gültig.
Die Messwerte der beiden erfolgreichen Flüge sind in Tabelle 5.4 dargestellt.

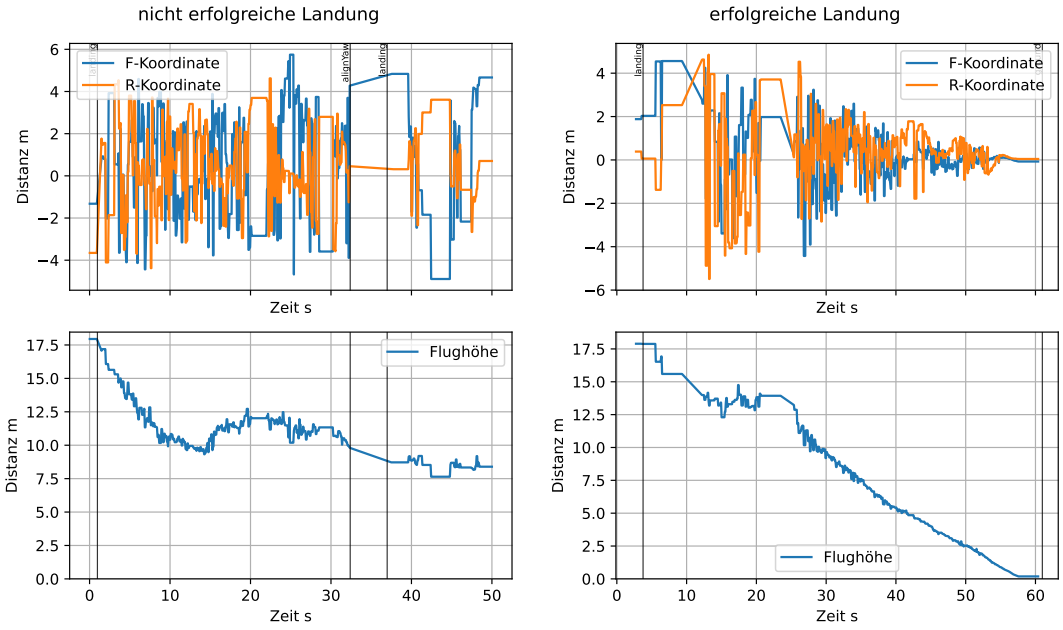
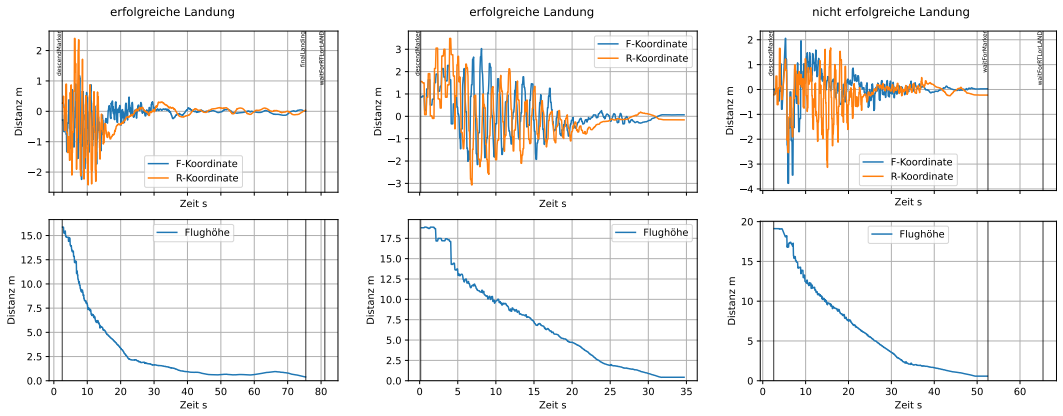


Abbildung 5.11.: F und R Koordinaten (oben) sowie Flughöhe (unten) für eine nicht erfolgreiche (a) und eine erfolgreiche (b) Landung mit dem Ardupilot Precision Landing Verfahren im Außenbereich

	Einheit	Flug 1	Flug 2
Landeposition X	cm	8	45
Landeposition Y	cm	2	10
Abstand zur Markermitte	cm	8,25	46,1
Ausrichtung zum Marker	°	3	1
Windgeschwindigkeiten	m/s	2,5	2,6

Tabelle 5.4.: Ergebnisse der Evaluation im Außenbereich für das Ardupilot Precision Landing Verfahren



(a) *(b)* *(c)*

Abbildung 5.12.: F und R Koordinaten (oben) sowie Flughöhe (unten) für eine erfolgreiche Landung mit 2 $\mathrm{m/s}$ Wind (a), eine erfolgreiche Landung mit 6 $\mathrm{m/s}$ Wind (b) und einer nicht erfolgreichen Landung (c) mit dem geschwindigkeitsbasierten Verfahren

Ergebnisse geschwindigkeitsbasiertes Landeverfahren

Für das dritte Verfahren wurden insgesamt 20 Flüge in zwei Testreihen durchgeführt. Die beiden Testreihen unterscheiden sich durch die Ausrichtung des Markers und wurden ca. eine Stunde zeitversetzt durchgeführt. Bei insgesamt drei Flügen wurde manuell eingegriffen, wobei nur bei einem das Verlieren des Markers der Grund war. Bei den anderen beiden Flügen war der Drift im Höhenwert der Drohne so stark, dass diese teilweise im State *finalLanding* wieder zu steigen begann. Dieser Fehler lässt sich nicht dem Landeverfahren anrechnen, weshalb diese Datenpunkte für die Auswertung aus dem Datensatz entfernt wurden. Für die statistische Auswertung werden deshalb für die erste Testreihe 9 Flüge und für die zweite 8 Flüge ausgewertet. In Abbildung 5.12 sind exemplarisch drei Flüge dargestellt. In (a) und (b) sind zwei erfolgreiche Landungen bei 2 $\mathrm{m/s}$ und 6 $\mathrm{m/s}$ Windgeschwindigkeit abgebildet. Zu beobachten ist auch hier ein Überschwingen über die Marker zu Beginn der Landung. Dieser Effekt ist bei der Landung mit mehr Wind deutlicher ausgeprägt. In (c) ist die nicht erfolgreiche Landung dargestellt. Zu sehen ist hier, dass der Marker erst kurz vor dem Aufsetzen auf der Plattform verloren wurde und nicht, wie bei den anderen Verfahren in größerer Höhe.

In Abbildung 5.13 sind alle Landungen der beiden Testreihen dargestellt. Die Punkte repräsentieren den Landeort im Markerkoordinatensystem. Die Farbe der Punkte gibt die Stärke des Winds zum Zeitpunkt der Landung an. Zusätzlich ist ein Pfeil mit der durchschnittlichen Windrichtung für diese Testreihe im Plot eingezeichnet.

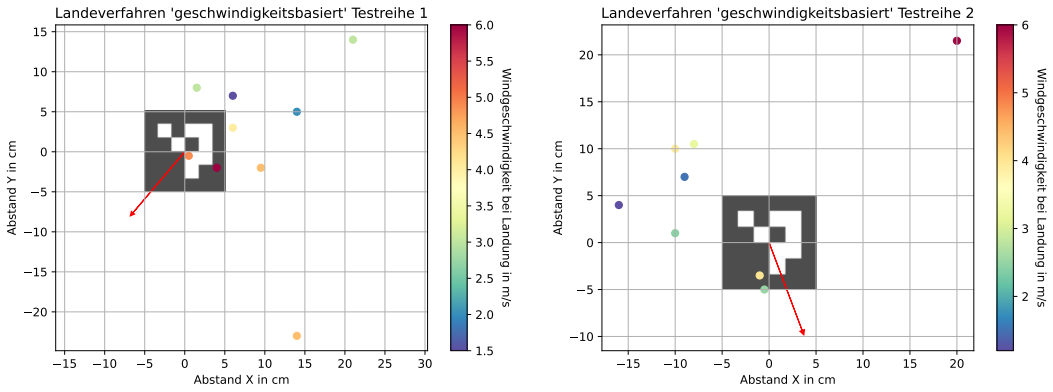


Abbildung 5.13.: Darstellung der Landungen im Außenbereich für beide Testreihen. Die Punkte repräsentieren die Landungen, die Windgeschwindigkeit zum Zeitpunkt der Landung ist über die Farbkodierung der Landepunkte dargestellt. Die mittlere Windrichtung über alle Flüge ist als roter Pfeil eingezeichnet.

In Abbildung 5.13 fällt auf, dass die Landungen im Durchschnitt gegen den Wind versetzt erscheinen. Um diesen Zusammenhang weiter zu untersuchen, sind in Abbildung 5.14 alle Landungen beider Testreihen, im Zusammenhang mit der Windrichtung abgebildet. Auf der Y Achse befindet sich die Richtung gegen den Wind (invertierte Windrichtung) für jede Landung. Auf der X Achse ist der Winkel zwischen dem Ortsvektor der Landung und der Nordrichtung aufgetragen. Hier deutet sich ein linearer Zusammenhang zwischen Windrichtung und dem Ort der Landung an.
Bei beiden Testreihen zeigt sich eine ähnliche durchschnittliche Abweichung vom Soll-Landepunkt von $11{,}78\,\mathrm{cm}$ für die erste und $12{,}92\,\mathrm{cm}$ zweite Testreihe. In den mittleren Landepositionen in X und Y Richtung spiegelt sich der Einfluss des Windes wieder. Im Durchschnitt über beide Testreihen, erreichte das Verfahren eine Abweichung von $12{,}31\,\mathrm{cm}$ gegenüber der Soll-Landeposition und eine Standarabweichung von $8{,}03\,\mathrm{cm}$. Die mittlere Winkelabweichung zum Marker betrug $1{,}76°$ mit einer Standardabweichung von $1{,}86°$. Bis auf die eine Landung, bei der der Marker im Landeanflug verloren wurde, waren alle Landungen erfolgreich. In Tabelle 5.5 sind die Ergebnisse der Flüge dargestellt. Abgesehen von der Erfolgsquote für alle Landungen beziehen sich alle Kenngrößen auf die 17 erfolgreichen Landungen.

	Einheit	Wert
Testreihe 1:		
Anzahl Flüge ohne Eingreifen		9
mittlere Position in X	cm	8,5
mittlere Position in Y	cm	1,06
mittlere absolute Abweichung	cm	11,78
mittlere Abweichung Ausrichtung	°	-1,0
mittlere Abweichung absolute Ausrichtung	°	1,89
Standardabweichung Landeposition	cm	8,47
Standardabweichung Ausrichtung	°	1,94
Testreihe 2:		
Anzahl Flüge ohne Eingreifen		8
mittlere Position in X	cm	-4,31
mittlere Position in Y	cm	5,69
mittlere absolute Abweichung	cm	12,91
mittlere Abweichung Ausrichtung	°	-1,125
mittlere Abweichung absolute Ausrichtung	°	1,63
Standardabweichung Landeposition	cm	7,45
Standardabweichung Ausrichtung	°	1,76
gesamt:		
Anzahl Flüge		18
Anzahl Flüge ohne Eingreifen		17
mittlere Position in X	cm	2,47
mittlere Position in Y	cm	3,24
mittlere absolute Abweichung	cm	12,31
mittlere Abweichung Ausrichtung	°	-1,06
mittlere Abweichung absolute Ausrichtung	°	1,76
Standardabweichung Landeposition	cm	8,03
Standardabweichung Ausrichtung	°	1,86
Quote erfolgreiche Landung (ohne Eingreifen)	%	100
Quote erfolgreiche Landung (alle Flüge)	%	94,44

Tabelle 5.5.: Ergebnisse der Evaluation im Außenbereich

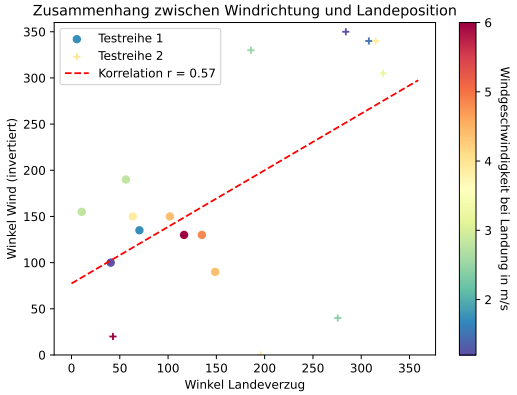


Abbildung 5.14.: Zusammenhang zwischen der invertierten Windrichtung (Richtung gegen den Wind) und dem Winkel zwischen dem Ortsvektor der Landung und Norden

5.3. Interpretation und Bewertung

Die Flugtests, sowohl in der Simulation als auch mit realer Hardware, haben gezeigt, dass alle drei untersuchten Verfahren prinzipiell funktionsfähig sind.
Am wenigsten kritisch hat sich die Ausrichtung der Drohne am Marker herausgestellt. Die geforderte maximale Abweichung von $\pm\ 5°$ wurde bei allen Flügen, bei denen der Marker nicht während der Landung verloren wurde, erreicht. Die maximale Abweichung vom Soll-Landepunkt ($35\,\mathrm{cm}$) wurde von dem Ardupilot Precision Landing Verfahren (2) und dem geschwindigkeitsbasierten Verfahren (3) in den meisten simulierten Flügen und allen im Innenbereich durchgeführten Flügen erreicht. Allgemein war eine, an den Anforderungen gemessen, hohe Präzision bei den Landungen zu beobachten, sofern die Marker im Landeanflug nicht verloren wurden. Eine Ausnahme davon stellte das positionsbasierte Verfahren im Flugtest im Innenbereich dar. Hier war häufig zu beobachten, dass die Drohne kurz vor dem Aufsetzen auf dem Boden noch einmal zu steigen begann. Da in dieser Flugphase (Statemachine State *finalLanding*) keine Korrektur der Position mehr erfolgt, führte dies häufig zu einem starken Abdriften und einer entsprechend unpräzisen Landung. Dieses Problem ist jedoch nicht auf das Landeverfahren zurückzuführen, sondern der Parametrierung des States *finalLanding* zuzuordnen. Im Landeverfahren 3, in dem dieser State auch implementiert ist, wurde als Schwellenwert für das Umschalten in die letzte Flugphase eine Höhe von $40\,\mathrm{cm}$ gewählt und dann mit einer Geschwindigkeit von $40\,\mathrm{cm/s}$ gesunken. Für das Verfahren 1 waren diese Parameter zu $45\,\mathrm{cm}$ und $33\,\mathrm{cm/s}$ gesetzt.

Hier zeigt sich auch ein allgemeines Problem der verwendeten Drohnen. Bei vielen Flügen fiel auf, dass die berechnete Flughöhe der Drohne und die reale Höhe teilweise um mehrere

Meter auseinanderlag. Zusätzlich unterlag die berechnete Flughöhe starken Schwankungen, was zu massiven Höhenänderungen der Drohne ohne Fernsteuerungseingriff führte. Diese Verhalten ist beispielsweise beim Flug in Abbildung 5.11 (a) zu beobachten. Der Effekt ist auf den Temperaturdrift des Barometers zurückzuführen, das von Ardupilot primär für die Berechnung der Flughöhe verwendet wird, konnte jedoch durch aktives Kühlen des Flightcontrollers vor jedem Flug zumindest verringert werden. Ardupilot bietet die Möglichkeit, eine Barometer-Temperatur-Kompensation durchzuführen [29]. Dies war jedoch bei dem verwendeten Flightcontroller (siehe Abschnitt 2.6) nicht erfolgreich durchführbar. Für einen weiteren Einsatz der hier entwickelten Systeme ist es sinnvoll, ein System zu verwenden, welches nicht dieser Problematik unterliegt. Aufseiten der Software wäre es sinnvoll, die Regelung der Sinkgeschwindigkeit nicht dem Flightcontroller zu überlassen, sondern basierend auf dem Höhenwert aus der Markererkennung einen eigenen Regelkreis zum Steuern der Flughöhe auf dem Companion Computer zu implementieren. Zusätzlich sollte ein weiterer Zustandsübergang von *finalLanding* zurück in den *descendMarker* State führen, falls die Drohne weiter zu steigen beginnt. Diese Maßnahmen lassen sich für die Verfahren 1 und 3 umsetzen, im Ardupilot eigenen Landeverfahren kann darauf kein Einfluss genommen werden.

Ein weiteres Problem stellt das stark unterschiedliche Verhalten der Verfahren 1 und 2 zwischen den Flügen im Innen- und Außenbereich dar. Erstaunlich ist, dass beide Verfahren ein sehr ähnliches Verhalten im Außenbereich zeigten, obwohl die verwendeten Algorithmen zur Steuerung unterschiedlich sind. Vermutlich ist der Unterschied im Verhalten nicht auf drinnen und draußen, sondern auf die Flughöhe, aus der der Landeanflug gestartet wird, zurückzuführen. In der Lagerhalle wurde aus ca $4\,\mathrm{m}$ Höhe gelandet, im Außenbereich wurde der Marker zwischen $17\,\mathrm{m}$ und $14\,\mathrm{m}$ erkannt und die Landung eingeleitet. Bei der Landung aus größerer Höhe war dementsprechend die Regelabweichung (horizontaler Abstand zum Marker) auch größer, was zu einer stärkeren Reaktion der Positionsregelung geführt haben könnte. Um hier eine abschließende Aussage über den Grund diese Verhaltens treffen zu können, müssen aber noch weitergehende Testflüge unternommen werden.

Dieses Verhalten offenbart jedoch eine entscheidende Eigenschaft der beiden Verfahren. Bei beiden wird die Positionsregelung durch den Flightcontroller durchgeführt, der Companion Computer liefert lediglich die Eingangsgrößen (Soll-Positionen). Dadurch ist der direkte Einfluss, den man auf die Parametrierung des Landeverfahrens hat, relativ gering. Dies bedeutet, dass die Parametrierung des Landesystems über die Parameter der Positionsregelung von Ardupilot erfolgen müssen. Die verwendeten Drohnen zeigten im normalen Flugbetrieb jedoch keine Defizite in der Parametrierung dieser Regelung. Inwieweit diese für das Landesystem angepasst werden kann, ohne das Flugverhalten in den restlichen Flugphasen zu

verschlechtern, kann noch weiter evaluiert werden. Interessant sind auch die Ergebnisse für die Windanfälligkeit des Systems. In der Simulation hat nur das Verfahren 2 einen signifikanten Einfluss der Windgeschwindigkeit auf die Landegenauigkeit gezeigt. Leider konnte dieses Verhalten durch die fehlenden Flüge im Außenbereich nicht in der Realität überprüft werden. Im Test im Außenbereich unter Realbedingungen zeigte das dritte Verfahren einen erheblichen Einfluss der Windrichtung auf den Ort der Landung. Im Durchschnitt waren die Landungen gegen die Windrichtung versetzt. Eine mögliche Erklärung für dieses Verhalten ist, dass die Windgeschwindigkeit knapp über dem Boden abnimmt. Die Drohne, welche im vorherigen Abstieg, um ihre Position halten zu können, gegen den Wind geneigt ist, kann dies nicht schnell genug ausregeln, wodurch sie ein Stück gegen den Wind versetzt wird. Um dies und den Einfluss der Windgeschwindigkeit abschließend beurteilen zu können, müssten jedoch noch mehr Daten erhoben werden.
Die Vorgabe, dass Landungen bis zu einer Windgeschwindigkeit von $10\,\mathrm{m/s}$ erfolgreich durchgeführt werden sollen, konnte Aufgrund der gegebenen Windverhältnisse nicht verifiziert werden. Das Verhalten bei den Testflügen lässt aber vermuten, dass die Funktionalität auch auch für höhere Windgeschwindigkeiten, als die getesteten $6\,\mathrm{m/s}$ gegeben ist. Bei Windgeschwindigkeiten $> 10\,\mathrm{m/s}$ sind die Marker durch die Neigung der Drohne nicht mehr im Kamerabild, wenn sich die Drohne über dem Marker befindet. Dadurch können die hier beschriebenen Landeverfahren nicht mehr ohne Anpassung der Kamera oder der Anflugroute funktionieren.

Für die Anwendung in dem vorliegenden Projekt bei LHIND bietet sich das geschwindigkeitsbasieten Verfahren an. Dies hat mit den verwendeten Drohnen die besten Ergebnisse, sowohl im Innen- als auch Außenbereich, gezeigt. Zusätzlich lässt es sich auch auf andere Drohnenplattformen adaptieren, wodurch es auch für andere Projekte einsetzbar ist.
Die prinzipielle Funktionsweise der beiden anderen Verfahren ist mit der Einschränkung gegeben, dass hier weitere Anpassungen und Optimierungen durchgeführt werden. Für sie spricht die einfachere Implementierbarkeit, wodurch sie auch mit weniger Aufwand getestet werden können. Zudem ist für ihre Implementierung kein Wissen im Bereich der Regelungstechnik notwendig. Möglicherweise bietet sich ein Einsatz bei größeren Drohnen an, da bei diesen das Systemverhalten träger ist und es möglicherweise nicht zu einem Aufschwingen der Positionsregelung kommt. Dies muss aber noch weiter untersucht werden.

6. Fazit

6.1. Zusammenfassung

Diese Bachelorarbeit widmet sich der Entwicklung eines präzisen und robusten Systems zum automatisierten Landen von UAVs. Dieses Ziel wird vor dem Hintergrund der steigenden Bedeutung der Automatisierung von unbemannten Luftfahrzeugen untersucht. Die Herausforderung besteht darin, UAVs die Fähigkeit zu verleihen, autonom und präzise auf einer Landeplattformen zu landen, um den automatisierten Betrieb der Drohne zu ermöglichen.
Zu diesem Zweck wird nach der Darstellung der Grundlagen, welche zum Verständnis dieser Arbeit notwendig sind, ein Konzept erarbeitet, wie ein solchen Landeverfahren aufgebaut werden kann. Dazu erfolgt eine Analyse der Anforderungen an das System sowie eine Betrachtung des aktuellen Stands der Forschung zu diesem Thema. Diese Arbeit gliedert sich in ein Projekt zur Entwicklung eines Systems aus Drohnen und Landeboxen ein, welches von Lufthansa Industry Solutions durchgeführt wurde. Daraus leiten sich die Anforderungen und Vorgaben für diese Arbeit ab. Der Fokus liegt auf der Präzision und Zuverlässigkeit des Landesystems sowie einem robusten Verhalten gegenüber Wind. Eingesetzt werden soll ein optisches Positionierungssystem mittels Arcuo Markern.
Neben einem allgemeinen Konzept zur Implementierung werden drei Konzepte für die Steuerung der Landung ausgearbeitet, die in dieser Arbeit implementiert und getestet werden.
Das erste Verfahren basiert darauf, die Position der Landeplattform in ein erdfestes Koordinatensystem umzurechnen und der Drohne zyklisch als Zielposition zu übermitteln (positionsbasiertes Verfahren). Die Sinkgeschwindigkeit wird abhängig von der Flughöhe gesteuert. Das zweite Verfahren nutzt das eigene Präzisionslandeverfahren der Flightcontroller Software der Drohne (Ardupilot). Diesem werden die Positionen der Plattform zyklisch übermittelt, wodurch die Drohne in der Lage ist, selbstständig auf dieser Landeplattform zu landen (Ardupilot Precision Landing Verfahren). Das dritte Verfahren implementiert einen Regelkreis auf dem Companion Computer zur Steuerung der Drohne relativ zur Landeplattform und übermittelt der Drohne Zielgeschwindigkeiten als Steuerungsbefehle (geschwindigkeitsbasiertes Verfahren).
Für die Implementierung wird ein von Helge Hackbarth bei LHIND entwickeltes Software

Framework verwendet und für diese Arbeit erweitert. Die drei Landeverfahren werden als *Statemachines* implementiert und während der Entwicklung mittels Software in the Loop Simulation der Drohne kontinuierlich getestet.
Nach der Implementierung werden alle drei Verfahren evaluiert. Zuerst erfolgen jeweils ca. 1.000 Flüge in der Simulation, wo alle drei Verfahren eine prinzipielle Funktionsfähigkeit zeigen. Mit einer Quote erfolgreicher Landungen von $100\,\%$ schneidet das dritte Verfahren hier am besten ab. Am schlechtesten funktioniert das erste Verfahren mit einer Quote von $99{,}01\,\%$, gefolgt vom zweiten Verfahren mit $99{,}1\,\%$. Alle 3 Verfahren zeigen eine gute Resistenz gegen Wind und funktionieren bis zu den getesteten $10\,\mathrm{m/s}$. Lediglich das zweite Verfahren zeigt geringere Präzision und größere Streuung bei Windgeschwindigkeiten über $7\,\mathrm{m/s}$.
Für die Evaluation auf der realen Drohne, werden Flüge im Innen- und Außenbereich durchgeführt. Die Flüge im Innenbereich finden in einer Lagerhalle statt. Die Landungen werden aus einer Höhe von $4\,\mathrm{m}$ initiiert. Jedes Verfahren wird mit 20 Landungen getestet. Die Verfahren 2 und 3 erreichen eine Quote erfolgreicher Landungen von $100\,\%$, Verfahren 1 kommt auf einen Wert von $75\,\%$. Bei den Flügen im Außenbereich wird aus einer Höhe von 25 Metern angeflogen. Für das Verfahren 3 werden 20 Flüge durchgeführt. Für die anderen beiden werden die Tests werden nach 3 und 4 Flügen abgebrochen, da sich die Drohne stark aufschwingt. Die Drohne erreicht mit dem dritten Verfahren eine Quote erfolgreicher Landungen von $94{,}44\,\%$.

Diese Arbeit hat gezeigt, dass alle drei hier untersuchten Verfahren prinzipiell geeignet sind, in einem optischen Präzisionslandesystem für Drohnen eingesetzt zu werden. Dabei bieten alle 3 Verfahren noch Potential zur Optimierung. Welches System auf einer Drohne eingesetzt werden soll, ist abhängig von der verwendeten Drohne sowie den Randbedingungen des Entwicklungsprojekts. Für das hier entwickelte System bei LHIND bietet sich das dritte Verfahren mit selbst implementiertem Positionsregelkreis an. Mit ihm wurden die besten Ergebnisse im Flugversuch unter Realbedingungen erzielt. Um jedoch in einem kommerziellen Drohnensystem eingesetzt zu werden, muss auch hier die Zuverlässigkeit noch weiter erhöht werden.
Der Einsatz einer Simulation als Entwicklungswerkzeug eines solchen Systems hat sich als sehr vorteilhaft herausgestellt. Auch wenn die simulierte Drohne nicht die reale Drohne nachbildet, ist das Systemverhalten ähnlich genug, um die prinzipielle Funktionsweise der realen Drohne abzubilden, was die Entwicklung erheblich vereinfacht und beschleunigt. Die Parametrierung der Systeme muss jedoch an der realen Drohne durchgeführt werden.

6.2. Kritische Reflexion

Trotz der erfolgreichen Implementierung aller drei Landeverfahren gab es bei der Erstellung dieser Arbeit einige Probleme. Bei den Flugversuchen mit den realen Drohen waren diese häufig nicht in der Lage, ihre Höhe zu halten (wie in Abschnitt 5.3 beschrieben). Dieser Effekt war teilweise so stark, dass die Drohne stieg obwohl, das Landeverfahren einen Sinkflug vorgab. Die Suche nach den Ursachen ergab ein thermisches Driften des Barometers. Dieses Problem war mit der vorliegenden Hardware nicht lösbar, eine Verbesserung brachte lediglich das aktive Kühlen der Drohne durch einen Lüfter zwischen den Flügen. Für die weitere Entwicklung sollte ein anderer Flightcontroller verwendet werden, bei dem diese Problematik nicht vorliegt. Auf Seite der Software kann ebenfalls Verbesserung erreicht werden, indem die Sinkgeschwindigkeit nicht als Sollwert an den Flightcontroller übermittelt, sondern mit einem eigenen Regelkreis auf dem Companion Computer gesteuert wird.
Wie sich beim letzten Flugversuch unter Realbedingungen herausgestellt hat, sind die Verfahren 1 und 2 noch nicht optimal parametriert. Da sich dieses Verhalten nicht während der Entwicklung in der Simulation und auch nicht bei den Landungen aus geringeren Höhen abzeichnete, blieb am Ende dieser Arbeit kein Raum mehr für eine weitere Optimierung der beiden Verfahren. Um diese Verfahren weiter zu optimieren, muss identifiziert werden, mit welchen Ardupilot Parametern Einfluss auf das Verhalten der Drohne während der Präzisionsladung genommen werden kann, um ein Aufschwingen der Positionsregelung zu verhindern.
Für die Entwicklung hat sich die Simulation als äußerst hilfreich herausgestellt. Mit ihr kann schnell überprüft werden, ob der geschriebene Code prinzipiell funktionsfähig ist. Da bei der verwendeten Simulation die simulierte Drohne jedoch nicht angepasst werden kann, konnten keine Rückschlüsse über das dynamische Verhalten der Drohen in der Realität getroffen werden. Das dritte Verfahren, mit selbst implementiertem Regelkreis auf dem Companion Computer, wies eine hohe Übereinstimmung des simulierten Verhaltens mit dem realen Verhalten aus. Um ein stabiles Systemverhalten zu erreichen, mussten lediglich alle PID-Parameter leicht reduziert werden. Die Parametrierung der anderen beiden Verfahren muss zum Großteil über die Ardupilot Parameter erfolgen. Da bei den Flugtests während der Entwicklung aus kleinen Anflughöhen keine Probleme auftraten, wurde in diese Richtung auch weniger Zeit in die Optimierung investiert. Möglicherweise hätte die Verwendung einer Simulation, welche die eingesetzten realen Drohnen besser abbildet, dazu geführt, dass das reale Verhalten (Aufschwingen bei größeren Anflughöhen) schon vorher hätte erkannt werden können.
Durch die geringe Anzahl von Tests im Außenbereich, kann in dieser Arbeit keine Aussage über das Systemverhalten bei verschiedenen Lichtbedingungen getroffen werden. Hier muss

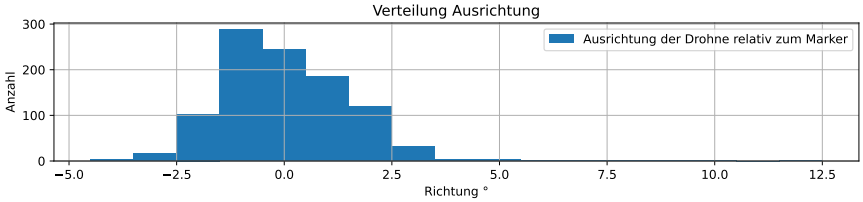
noch weiter untersucht werden, ob wechselnde Helligkeit, Reflexionen und Schattenwurf einen Einfluss auf die Erkennung der Marker haben.
Die in den Anforderungen vorgegebene Höhe von $25\,\mathrm{m}$, aus welcher der große Marker erkannt werden soll, wurde nicht erreicht. Bei machen Flügen wurde dieser erst ab einer Höhe von $14\,\mathrm{m}$ detektiert. Dies stellte aber in der Realität kein Problem dar. Die *Home* Position der Drohne über GNSS-Position war genau genug, um einen Anflug über dem Marker durchzuführen, bei dem die Marker zum Zeitpunkt der ersten Erkennung noch im Bild waren.

6.3. Ausblick

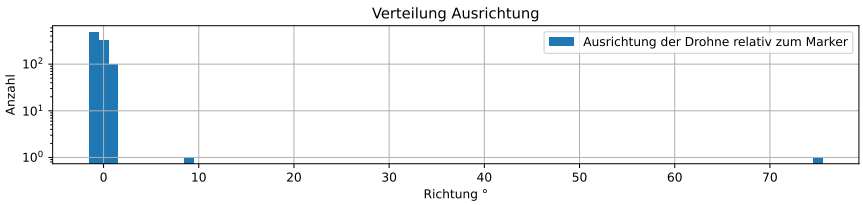
Bei der Verwendung optischer Positionierungssysteme mittels Marker zur Realisierung eines Präzisionslandesystems für UAVs besteht noch viel Optimierungs- und Erweiterungspotenzial. Diese Arbeit hat sich hauptsächlich mit der konkreten Steuerung und Regelung der Drohne beschäftigt. Ein Element um die Präzision dieser zu erhöhen, könnte eine Kompensation des Landefehlers aufgrund des Windes sein.
Ein Thema, welches darüber hinaus noch viel Potenzial bietet, ist die Erkennung der Marker. Diese zu verbessern und robuster gegen Umwelteinflüsse (Lichtbedingungen, teilweise Verschattung) oder zeitweise Verdeckung der Marker zu machen, könnte die Performance des Gesamtsystems erheblich verbessern. Dafür würde sich ein Vergleich verschiedener Fiducial Marker Typen anbieten sowie ein System, welches bei Verlust des Markers eine Prognose über seine aktuelle Position gibt. Ein solches System ließe sich mit einem Kalman Filter realisieren, wie von Wu 2019 [30] vorgestellt.
Ein weiterer interessanter Punkt, welcher für den realen Einsatz relevant ist, könnte die Entwicklung eines Systems sein, welches erkennt, ob sich die Landeplattform in einem für die Landung geeigneten Zustand befindet. Durch die Verwendung von neuronalen Netzen könnte überprüft werden, ob die Plattform frei ist und falls dies nicht der Fall sein sollte, ein geeigneter Ort in ihrem Umkreis für eine Notlandung gefunden werden. Mit dieser Technik wäre auch ein System ohne Fiducial Marker denkbar, bei dem die Positionierung rein über ein erlerntes Abbild der Landeplattform geschieht.
Auch die Erweiterung der Positionsbestimmung über optische Marker um ein weiteres, unabhängiges Positionierungssystem ist denkbar. Durch ein redundantes Positionierungssystem könnten Zuverlässigkeit und Robustheit des Systems erheblich gesteigert werden. Für diesen Zweck sollte die Nutzung eines Ortungssystem auf UWB (Ultra-Wideband, deutsch: Ultrabreitband) weiter erforscht werden.

A. Anhang

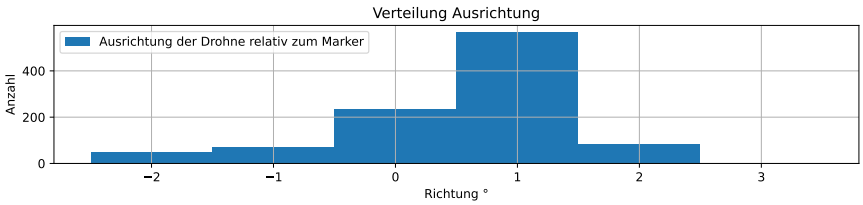
A.1. Abbildungen



(a)

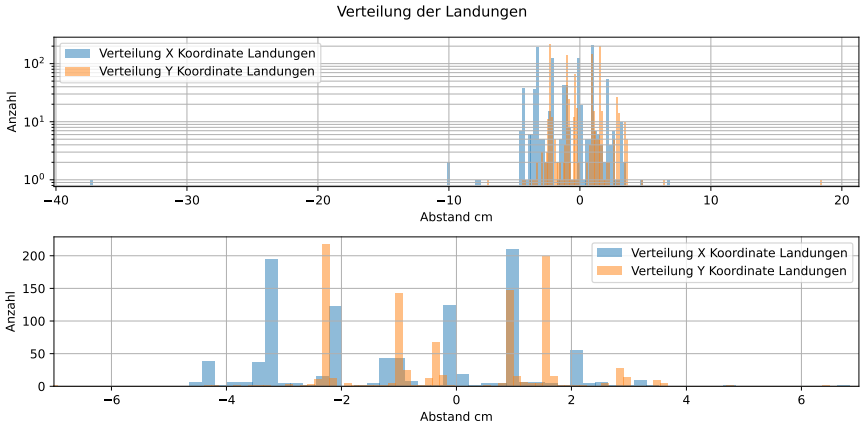


(b)

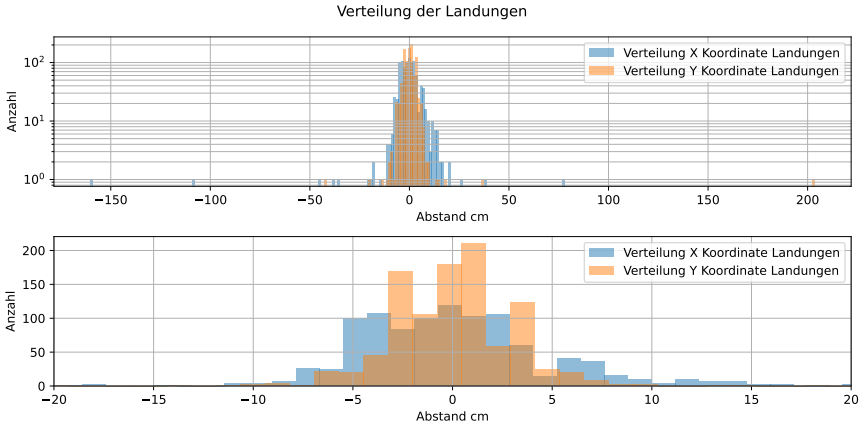


(c)

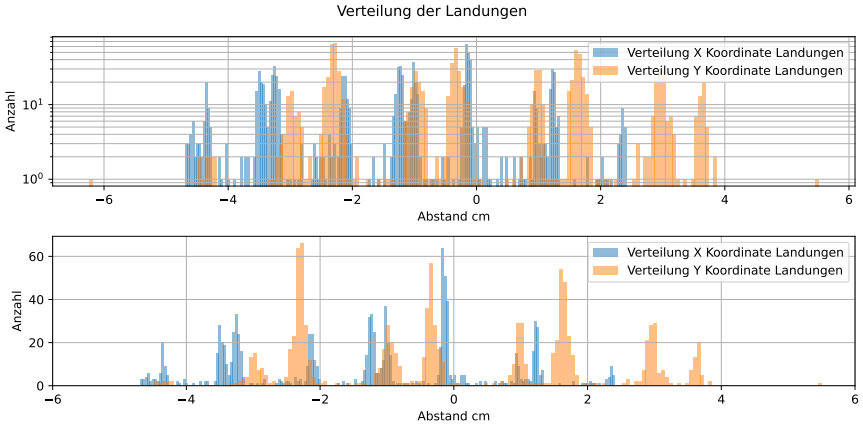
Abbildung A.1.: Verteilung der Landeausrichtung für die Evaluation mittels SITL Simulation (a) positionsbasiertes Verfahren, (b) Ardupilot Precision Landing Verfahren, (c) geschwindigkeitsbasiertes Verfahren.



(a)



(b)



(c)

Abbildung A.2.: Verteilung der Landungen der simulierten Flüge, jeweils gesamt (oben) und Ausschnitt um den Marker (unten)
(a) positionsbasiert, (b) Ardupilot Precision Landing, (c) geschwindigkeitsbasiert

A.2. Tabellen

Field Name	Type	Units	Values	Description
time_boot_ms	uint32_t	ms		Timestamp (time since system boot).
target_system	uint8_t			System ID
target_component	uint8_t			Component ID
coordinate_frame	uint8_t		MAV_FRAME	Valid options are: MAV_FRAME_LOCAL_NED = 1, MAV_FRAME_LOCAL_OFFSET_NED = 7, MAV_FRAME_BODY_NED = 8, MAV_FRAME_BODY_OFFSET_NED = 9
type_mask	uint16_t		POSITION_TARGET _TYPEMASK	Bitmap to indicate which dimensions should be ignored by the vehicle.
x	float	m		X Position in NED frame
y	float	m		Y Position in NED frame
z	float	m		Z Position in NED frame (note, altitude is negative in NED)
vx	float	m/s		X velocity in NED frame
vy	float	m/s		Y velocity in NED frame
vz	float	m/s		Z velocity in NED frame
afx	float	m/s/s		X acceleration or force (if bit 10 of type_mask is set) in NED frame in meter / s^2 or N
afy	float	m/s/s		Y acceleration or force (if bit 10 of type_mask is set) in NED frame in meter / s^2 or N
afz	float	m/s/s		Z acceleration or force (if bit 10 of type_mask is set) in NED frame in meter / s^2 or N
yaw	float	rad		Yaw setpoint
yaw_rate	float	rad/s		Yaw rate setpoint

Tabelle A.1.: Vollständige Definition der MAVLink Message SET_POSITION_TARGET_LOCAL_NED [18]

Param (:Label)	Description	Values	Units
Angle	target angle, 0 is north		deg
Angular Speed	angular speed		deg/s
Direction	direction: -1: counter clockwise, 1: clockwise	min: -1 max:1 increment:2	
Relative	0: absolute angle, 1: relative offset	min:0 max:1 increment:1	

Tabelle A.2.: Auszug aus der Definition der MAVLink Message MAV_CMD_CONDITION_YAW [18]

Literaturverzeichnis

[1] Richard Hartley and Andrew Zisserman. *Multiple View Geometry in Computer Vision.* Cambridge University Press, New York, NY, USA, 2 edition, 2003.

[2] David A. Forsyth and Jean Ponce. *Computer Vision - A Modern Approach, Second Edition.* Pitman, 2012.

[3] Prof. Dr. Thomas Frischgesell. Vorlesungsunterlagen Robotertechnik Abschnitt 2. 2022.

[4] Sajjad Mohammadian, Jantina Fokkema, and Alexandra V. Agronskaia et. al. High accuracy, fiducial marker-based image registration of correlative microscopy images. *Sci Rep*, (9), 2019. `https://doi.org/10.1038/s41598-019-40098-4`.

[5] Ashley R. Carter, Gavin M. King, Theresa A. Ulrich, Wayne Halsey, David Alchenberger, and Thomas T. Perkins. Stabilization of an optical microscope to 0.1 nm in three dimensions. *Appl. Opt.*, 46(3):421–427, Jan 2007.

[6] Surface Mount Land Patterns Subcommittee (1-13) of the Printed Board Design Committee (1-10) of IPC. IPC-7351A - Generic Requirements for Surface Mount Design and Land Pattern Standard, 2005.

[7] Edwin Olson. AprilTag: A robust and flexible visual fiducial system. In *Proceedings - IEEE International Conference on Robotics and Automation*, pages 3400 – 3407, 06 2011.

[8] S. Garrido-Jurado, R. Muñoz-Salinas, F.J. Madrid-Cuevas, and M.J. Marín-Jiménez. Automatic generation and detection of highly reliable fiducial markers under occlusion. *Pattern Recognition*, 47(6):2280–2292, 2014.

[9] M. Fiala. ARTag, a fiducial marker system using digital techniques. In *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, volume 2, pages 590 – 596 vol. 2, 07 2005.

[10] Michail Kalaitzakis, Sabrina Carroll, Anand Ambrosi, Camden Whitehead, and Nikolaos Vitzilaios. Experimental Comparison of Fiducial Markers for Pose Estimation. In *2020 International Conference on Unmanned Aircraft Systems (ICUAS)*, pages 781–789, 2020.

[11] D. Jurado-Rodriguez, R. Muñoz-Salinas, and S. et al Garrido-Jurado. Planar fiducial markers: a comparative study. *Virtual Reality*. 27, 1733-1749 (2023). https://doi.org/10.1007/s10055-023-00772-5.

[12] John Canny. A computational approach to edge detection. *IEEE Transactions on pattern analysis and machine intelligence*, (6):679–698, 1986.

[13] ArduPilot Dev Team. Ardupilot Developer Documentation. `https://ardupilot.org/dev/`. Abgerufen am 17.10.2023.

[14] Jan Lunze. *Regelungstechnik 2 : Mehrgrößensysteme, Digitale Regelung*. Springer Berlin Heidelberg, 2016.

[15] Patrick Vogl. Design und Implementierungverschiedener Regelungskonzepte zur Lageregelung eines Quadrokopters. Masterarbeit, HAW Hamburg, 2014.

[16] Ardupilot.org - Flight Modes. `ttps://ardupilot.org/copter/docs/flight-modes.html`. Abgerufen am 02.11.2023.

[17] MAVLink Developer Guide. `https://mavlink.io/en/`. Abgerufen am 13.09.2023.

[18] MAVLINK Common Message Set. `https://mavlink.io/en/messages/common.html`. Abgerufen am 13.09.2023.

[19] Mavlink - Landing Target Protocol. `https://mavlink.io/en/services/landingtarget.html`. Abgerufen am 13.09.2023.

[20] Mavlink Documentation - Using Pymavlink Libraries (mavgen). `https://mavlink.io/en/mavgenpython`. Abgerufen am 01.11.2023.

[21] DroneKit-Python documentation. `https://dronekit-python.readthedocs.io`. Abgerufen am 01.11.2023.

[22] Joshua Springer. Autonomous Landing of a Multicopter Using Computer Vision. Master's thesis, Reykjavík University;, 2020.

[23] Matthew Dupree and Yingchao Zhu. Optically-Guided Multirotor Autonomous Descent and Landing on a Moving Target. *International Telemetering Conference Proceedings, 56*, 2021.

[24] Nguyen Xuan-Mung, Sung Kyung Hong, Ngoc Phi Nguyen, Le Nhu Ngoc Thanh Ha, and Tien-Loc Le. Autonomous Quadcopter Precision Landing Onto a Heaving Platform: New Method and Experiment. *IEEE Access*, 8:167192–167202, 2020.

[25] Tiziano Fiorenzani. Github tizianofiorenzani - how_do_drones_work. `https://github.com/tizianofiorenzani/howdodroneswork`. Abgerufen am 14.09.2023.

[26] Ardupilot Complete Parameter List - Full Parameter List of Copter stable V4.4.1. `https://ardupilot.org/copter/docs/parameters.html`. Abgerufen am 21.10.2023.

[27] ArduPilot Dev Team. Github - Ardupilot Code - AC_PrecLand.cpp. `https://github.com/ArduPilot/ardupilot/blob/master/libraries/ACPrecLand/ACPrecLand.cpp`. Abgerufen am 27.10.2023.

[28] Andrew Banks, Ed Briggs, Ken Borgendale, and Rahul Gupta. OASIS MQTT Version 5.0. `https://docs.oasis-open.org/mqtt/mqtt/v5.0/mqtt-v5.0.pdf`. Abgerufen am 17.10.2023.

[29] ArduPilot Dev Team. Ardupilot Documentation - Barometer Temperature Compensation. `https://ardupilot.org/copter/docs/common-baro-temp-comp.html`. Abgerufen am 27.10.2023.

[30] Yibin Wu, Xiaoji Niu, Junwei Du, Le Chang, Hailiang Tang, and Hongping and Zhang. Artificial Marker and MEMS IMU-Based Pose Estimation Method to Meet Multirotor UAV Landing Requirements. *Sensors (Basel).* 2019 Dec 9;19(24):5428. doi: 10.3390/s19245428.

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.

Hamburg, 13. November 2023 Julian Wollenberg