Swarm Intelligence

Studienarbeit

im Studiengang Luft- & Raumfahrttechnik/-elektronik

an der DHBW Ravensburg

Campus Friedrichshafen

von

Julian Klose,

Michael Eberhardt

und

Jan Krecke

08.07.2015

Bearbeitungszeitraum	01.10.2014 bis 19.06.2015
Matrikelnummer	3903860, 7698500 und 5275363
Betreuer	Prof. Dr. Thomas Mannchen

Erklärung

gemäß § 5 (3) der „Studien- und Prüfungsordnung DHBW Technik“ vom 22. September 2011.

Hiermit erkläre ich, dass ich die vorliegende Arbeit mit dem Titel

Swarm Intelligence

selbständig angefertigt, nicht anderweitig zu Prüfungszwecken vorgelegt, keine anderen als die angegebenen Hilfsmittel benutzt und wörtliche sowie sinngemäße Zitate als solche gekennzeichnet habe.

Manching, den 06.07.2015

Julian Klose Michael Eberhardt Jan Krecke

Kurzfassung

Die vorliegende Arbeit befasst sich mit der Konzeptierung eines Schwarms bestehend aus mehreren Multicoptern. In einem Schwarm ist das Verhalten durch die Interkation der Mitglieder geprägt. Die für das Bestehen des Schwarms nötige Intelligenz ist dabei nicht in den einzelnen Mitgliedern lokalisiert, sondern in dem Verbund. Deshalb nimmt die Ausarbeitung der Schwarmintelligenz, im Englischen „Swarm Intelligence“ genannt, die zentrale Rolle in dieser Studienarbeit ein.

Das Ziel der Implementierung einer Schwarmintelligenz bei Multicoptern ist der Gedanke eine Mission zu kommandieren, sodass diese völlig automatisch von dem Multicopterverbund durchgeführt wird. Zur Umsetzung dieses Ziels werden in dieser Arbeit mehrere Konzepte und Mechanismen erläutert.

Ein weiterer Bestandteil stellt die programmierte on-board Software der Multicopter in Matlab/Simulink dar, dessen Architektur und Funktionen detailliert erläutert werden. Die Software dient zur Flugbahnberechnung zu kommandierten Wegpunkten und zur Früherkennung von Kollisionen, sodass diese durch die Berechnung eines Ausweichpunktes verhindert werden können. Im Zuge der Verifikation werden unterschiedliche Testszenarien simuliert, die die Fehlerfreiheit der Software bestätigen, sodass diese zukünftig im Flugversuch erprobt werden kann.

Darüber hinaus soll auf den Multicoptern eine Kamera installiert sein, sodass zukünftig Ziele verfolgt und Hindernisse erkannt und umgangen werden können. Dazu wurde mit Matlab-Simulink ein Algorithmus entwickelt, der Objekte sowohl nach ihrer Farbe als auch Form analysiert. So kann aus den mit der Bordkamera erfassten Bildern die GPS-Koordinate des Zielobjektes bestimmt werden.

Abstract

This paper deals with the conception of a multi-drone swarm. A swarm's behavior is characterized by the interaction between its members. The intelligence that is necessary for the swarm's existence is not located in the single participants, but in the whole compound. This very intelligence, called *Swarm Intelligence,* plays therefore the central role in this paper.

The major goal of implementing a *Swarm Intelligence* algorithm is to enable a group of (in this case) multicopters to perform missions without any human interaction. This paper contains various concepts and mechanisms to fulfill this goal.

Furthermore, this paper addresses the topic of the drones' onboard software, realized with Matlab/Simulink. The software is used for calculating the trajectory to certain waypoints, which can be defined from the ground station. Additionally, potential collisions with other participants or obstacles will be sensed by the software so that evasive maneuvers can be conducted. Several test scenarios were created in order to verify the faultless functionality of the software.

In addition to these functions a camera will be installed on the drones with the purpose of detecting and tracking target objects. Matlab/Simulink was used to create an algorithm that detects objects both by its shape and color. That way, the target's GPS-coordinates can be determined.

Inhaltsverzeichnis

Erklärung ..... **II**

Kurzfassung ..... **III**

Abstract ..... **IV**

Inhaltsverzeichnis ..... **V**

Abbildungsverzeichnis ..... **VIII**

Tabellenverzeichnis ..... **X**

Abkürzungsverzeichnis ..... **XI**

1 Einleitung **1**

2 Zielsetzung und Aufgabenstellung **4**

3 Theoretische Grundlagen **5**

3.1 Schwarmverhalten nach Reynolds 5

3.1.1 Kohäsion *6*

3.1.2 Separation *6*

3.1.3 Ausrichtung *7*

3.2 Phasen der Schwarminteraktion 8

4 Softwarearchitektur des PCC-Modells **10**

4.1 Simulink Blockansatz 10

4.2 Matlab-Function Codeansatz 13

5 Implementierung des Modells in Matlab/Simulink **15**

5.1 POI_Entscheidung 15

5.1.1 Ein- und Ausgabeparameter *16*

5.1.2 Funktionsbeschreibung *17*

5.2 Kollisionslogik 17

5.2.1 Ein- und Ausgabeparameter *17*

5.2.2 Funktionsbeschreibung *19*

5.3 Berechne_Ausweichkoordinate ... 21
5.3.1 Ein- und Ausgabeparameter ... 21
5.3.2 Funktionsbeschreibung ... 22
5.4 Flugbahnberechnung ... 27
5.4.1 Ein- und Ausgabeparameter ... 28
5.4.2 Funktionsbeschreibung ... 28
5.5 NaechsterWegpunkt ... 30
5.5.1 Ein- und Ausgabeparameter ... 31
5.5.2 Funktionsbeschreibung ... 32
5.6 checkNaN ... 33
5.6.1 Ein- und Ausgabeparameter ... 33
5.6.2 Funktionsbeschreibung ... 33
5.7 checkSize ... 34
5.7.1 Ein- und Ausgabeparameter ... 34
5.7.2 Funktionsbeschreibung ... 34
6 Simulation ... 35
6.1 Simulationsumgebung und -parameter ... 35
6.2 APM-Modell ... 36
6.3 Simulationsdurchläufe ... 37
6.3.1 Wegpunktflug ... 38
6.3.2 Ausweichen eines statischen Hindernisses ... 39
6.3.3 Ausweichen von mehreren statischen Hindernissen ... 40
6.3.4 Co-Simulation: Ausweichen dynamischer Hindernissen ... 41
6.4 Portierung auf GPS-Koordinaten ... 43
6.5 Diskussion der Ergebnisse ... 46
7 Bilderkennung und -verarbeitung ... 48
7.1 Grundlagen der Bildverarbeitung ... 48
7.1.1 Das RGB-Farbmodell ... 49
7.1.2 Die Kreis – Hough - Transformation ... 50
7.1.3 Die Kamera ... 54

7.2 Die Bilderkennung mit Matlab/ Simulink 55
7.2.1 Der Matlab-Ansatz 55
7.2.2 Der Simulink-Ansatz 60
7.3 Berechnung der Zielkoordinate aus Bilderfassung 65
7.3.1 Vereinbarung der zwei Methoden zur Berechnung der Zielkoordinate 66
7.3.2 Berechnung der Distanz zum Zielobjekt 67
7.3.3 Berechnung der lateralen und longitudinalen Distanz zum Zielobjekt . 69
8 Zusammenfassung und Ausblick 71
Literaturverzeichnis 73

Abbildungsverzeichnis

Abbildung 3-1: Schwarmverhalten nach Craig Reynolds 6

Abbildung 3-2: Darstellung des Kohäsionsverhaltens innerhalb eines Schwarms 6

Abbildung 3-3: Darstellung des Separationsverhaltens innerhalb eines Schwarms ... 7

Abbildung 3-4: Darstellung des Ausrichtungsverhaltens innerhalb eines Schwarms .. 7

Abbildung 3-5: Angepasstes Phasenmodell nach Tuckman 8

Abbildung 4-1: Softwarearchitektur Basisansatz 10

Abbildung 4-2: Implementierungsbeispiel des Simulink Blockansatzes 11

Abbildung 4-3: Sequentielle Matlab-Function Softwarearchitektur 13

Abbildung 5-1: Darstellung des Inhalts der Flugmatrix 15

Abbildung 5-2: Konzept der Kollisionslogik 19

Abbildung 5-3: Kollisionsfälle 23

Abbildung 5-4: Winkelunterscheidung in der Kollisionslogik 23

Abbildung 5-5: Berechne_Ausweichkoordinate Fall St.1 24

Abbildung 5-6: Berechne_Ausweichkoordinate Fall St.2/St.3 25

Abbildung 5-7: Berechne_Ausweichkoordinate Fall Dy.1: Parameter 26

Abbildung 5-8: Berechne_Ausweichkoordinate Fall Dy.1: Flugbahnen 26

Abbildung 5-9: Berechne_Ausweichkoordinate Fall Dy.2 und Dy.3: Parameter 27

Abbildung 5-10: Berechne_Ausweichkoordinate Fall Dy.2 und Dy.3: Flugbahn 27

Abbildung 6-1: Systemdarstellung in Simulink 35

Abbildung 6-2: APM-Modell 37

Abbildung 6-3: Darstellung der Sollkoordinate und momentanen Position bei Wegpunktflug 38

Abbildung 6-4: Darstellung der Sollkoordinate und momentanen Position beim Ausweichen eines statischen Hindernisses 40

Abbildung 6-5: Darstellung der Sollkoordinate und momentanen Position beim Ausweichen mehrerer statischer Hindernisse 41

Abbildung 6-6: Umsetzung der Co-Simulation in Simulink 42

Abbildungsverzeichnis

Abbildung 6-7: Darstellung der momentanen Positionen zweier UAS beim Ausweichen eines dynamischen Hindernisses 43

Abbildung 6-8: Wegpunkte für die Verifizierung der GPS-Portierung 45

Abbildung 6-9: Ergebnis der GPS-Portierung 46

Abbildung 7-1: Der RGB-Farbwürfel 50

Abbildung 7-2: Zwei Kreise für die KHT 51

Abbildung 7-3: Anwendung der CED 52

Abbildung 7-4: Anwendung der Hough-Transformation auf einzelnen Kreis 52

Abbildung 7-5: Abbildung des Hough-Bereichs 53

Abbildung 7-6: Drei Punkte im Hough-Bereich 53

Abbildung 7-7: C930e von Logitech 54

Abbildung 7-8: Roter Ball auf grüner Wiese 56

Abbildung 7-9: Intensität der roten Komponente als Graustufenbild 57

Abbildung 7-10: Roter Ball nach Abzug der anderen Komponenten 57

Abbildung 7-11: Schwellenwertbetrachtung der Intensität 58

Abbildung 7-12: Blockschaltbild zur Bilderkennung in Simulink 61

Abbildung 7-13: Kameraparameter 61

Abbildung 7-14: Binärdarstellung nach Otsu 63

Abbildung 7-15: Zwei rote Bälle 64

Abbildung 7-16: Binärdarstellung mit fixem Schwellwert 64

Abbildung 7-17: Binärdarstellung durch Otsu-Methode 65

Abbildung 7-18: Blockschaltbild zur Berechnung von Zielkoordinate 66

Abbildung 7-19: Darstellung Drehung der Drohne um eine Achse 67

Abbildung 7-20: Von Kamera erzeugte Pixel 68

Abbildung 7-21: Darstellung eines einzelnen Pixels 68

Abbildung 7-22: Zusammenhang zwischen Distanzen 70

Tabellenverzeichnis

Tabelle 1: Startkoordinaten und Wegpunkte der jeweiligen RPAS 42

Tabelle 2: GPS-Koordinaten der Wegpunkte und Hindernisse 45

Abkürzungsverzeichnis

APM	Ardupilot Mega
CED	Canny Edge Detection
GPS	Global Positioning System
KHT	Kreis-Hough-Transformation
Lat	Latitude
Lon	Longitude
NaN	Not a Number
PCC	Payload Control Computer
POI	Point Of Interest
RGB	Rot Grün Blau
UAV	Unmanned Aerial Vehicle
UAS	Unmanned Aerial System

1 Einleitung

Wie bei vielen anderen wichtigen Erfindungen auch, liegen die Ursprünge der unbemannten Luftfahrt beim Militär. Bereits 1849 kamen österreichische Streitkräfte auf die Idee, Bomben per Ballon nach Venedig zu transportieren und abzuwerfen [1]. Diese zugegebenermaßen rudimentäre Art und Weise der Kriegsführung hat selbstverständlich wenig mit modernen Drohnen zu tun. Die ihr zugrunde liegende Problemstellung und die sich ergebenden Vorteile sind jedoch nicht allzu weit von denen der heutigen Einsatzgebiete entfernt: Unbemannte Flugobjekte sollen den Menschen entlasten oder Aufgaben durchführen, zu denen ein Mensch gar nicht fähig wäre. Ginge es den alpenländischen Invasoren im 19. Jahrhundert noch darum möglichst viele Leben der eigenen Soldaten zu schonen (ein Motiv, das sich auch heute noch wiederfindet, zum Beispiel in den von den US-amerikanischen Streitkräften geführten Drohnenkriegen im Nahen Osten), so lassen sich *Unmanned Aerial Vehicles* (UAVs) heutzutage für eine schier unbegrenzte Anzahl an Anwendungen einsetzen. Am *Massachussets Institute of Technology* beispielsweise, wurden Personen, die Gäste und Studenten über den Campus führen, durch Drohnen ersetzt, die sich vom Anwender per Smartphone-App rufen lassen, und dann per GPS und Bildverarbeitungssoftware zum Zielort navigieren [2]. Ähnlich futuristisch geht es vor den Küsten der Vereinigten Staaten zu, wo ferngesteuerte Drohnen zum Kampf gegen den Drogenschmuggel eingesetzt werden. Innerhalb der ersten zwei Wochen nach Einführung des Systems konnten über 600 kg Kokain sichergestellt werden [3]. Zu medialer Berühmtheit gelangte Amazons Vorhaben, Pakete innerhalb von maximal 30 Minuten seinen Kunden zuzustellen [4]. Dies soll ebenfalls durch den Einsatz von unbemannten und autonom fliegenden Drohnen erfolgen. Bis zur Markteinführung des Systems wird zwar noch etwas Zeit vergehen, aber der Anblick von selbständig agierenden Flugobjekten in unseren Städten wird uns in nicht allzu ferner Zukunft unter Umständen genau so vertraut sein wie der eines Lieferwagens.

Auch in den Bereichen Forschung, Aufklärung, Natur- und Tierschutz, Journalismus, Search and Rescue, Filmproduktion und Landwirtschaft können durch den Einsatz von UAVs Aufgaben bearbeitet werden, die vom Menschen nicht oder nur mit ungleich größerem Aufwand bewerkstelligt werden könnten.

Einleitung

Die unbemannte Luftfahrt ist also ein verheißungsvolles Gebiet, wenn es darum geht, zukünftige Märkte zu erschließen. Dementsprechend sinnvoll ist es, dass Hochschulen ihre Studierenden frühzeitig an das Thema heranführen. Im Wintersemester 2014/15 wurde deshalb am Campus Friedrichshafen der Dualen Hochschule Baden-Württemberg Ravensburg das studentische Projekt Locator ins Leben gerufen. 10 Studierende arbeiteten an dem Projekt und dokumentieren ihre Ergebnisse in dieser Reihe von Studienarbeiten. Die Bearbeitung fand im Wintersemester 2014/15 und im Sommersemester 2015 statt. Es waren die ersten sechs Monate des Projekts Locator, unterbrochen von drei Monaten Praxisphase. Dieser Zeitraum stellt lediglich den Anfang eines vermutlich mehrjährigen Projektes dar, das mehrere Generationen von Studenten beschäftigen wird. In dieser Anfangsphase wurde noch kein endgültiges Ziel bzw. Anwendungsgebiet definiert, vorstellbar wären aber Einsätze in der Landvermessung oder experimentelle Flüge zur Entwicklung von Schwarmkonzepten und Erforschung der dahinter stehenden Intelligenz.

Der Plan für die Anfangsphase des Projektes war ein grundsätzliches Verständnis für die unbemannte Luftfahrt zu erlangen. Dazu sollte eine Commercial-Off-The-Shelf-Drohne so modifiziert werden, dass sie einen farbigen Ball komplett autonom verfolgen kann. Da die Ausgangsdrohne für ein solches Unterfangen nicht ausgelegt ist, mussten Änderungen an der Software, Hardware und Struktur vorgenommen werden. Professor Mannchen, der leitende Dozent des Projektes, entschied, dass diese Maßnahmen am besten von einer Kombination von Studierenden aus den Luft- und Raumfahrtsystemen (TLS), der Luft- und Raumfahrtelektronik (TLE) und der Nachrichten- und Kommunikationstechnik (TEN) durchgeführt werden sollen. Von den zehn beteiligten Studierenden gehörten zwei zu TLS, sechs zu TLE und zwei zu TEN. Das Projekt wurde in vier Untergruppen aufgeteilt: Autonomous Flight, Swarm Intelligence, Sensor Integration und Structural Integration. Ursprünglich waren die Aufgaben wie folgt definiert: Autonomous Flight sollte Missionen und Anforderungen an das Flugobjekt definieren und die Systemarchitektur für die Drohnen entwerfen. Die Aufgabe von Swarm Intelligence war es, Konzepte für das Zusammenarbeiten mehrerer Drohnen innerhalb eines Schwarms zu entwickeln. Das Team Sensor Integration befasste sich mit der Integration von den Sensoren und dem Datenlink, sowie dem Batteriemanagement

und der Sensorfusion. Structural Integration, schließlich, befasste sich mit dem mechanischen Aufbau, der Konstruktion und Integration der Drohnen. Die Grenzen zwischen den Aufgaben verschwammen jedoch – speziell am Anfang des Projektes – zwischen den einzelnen Gruppen, da einige Aufgaben mit höherer Priorität bearbeitet werden mussten, oder, da die Bearbeitung bestimmter Aufgaben nicht ohne die Resultate anderer Arbeitspakete erledigt werden konnten, sodass durch die Verlagerung von „Manpower“ bestimmte Aufgaben beschleunigt wurden.

Die einzelnen Kapitel dieser Studienarbeit wurden dabei von den folgenden Personen geschrieben:

Michael Eberhardt:	Kap. 3, Kap. 4, Kap. 5 (5.3, 5.6, 5.7), Kap. 6 (6.2, 6.4)
Jan Krecke:	Kap. 1, Kap. 7, Kap. 8
Julian Klose:	Kap. 2, Kap. 5 (5.1, 5.2, 5.4, 5.5), Kap. 6 (6.1, 6.3, 6.4, 6.5), Kap. 8

2 Zielsetzung und Aufgabenstellung

Diese Studienarbeit befasst sich mit der Konzeptierung der Schwarmintelligenz von Multicoptern, im Englischen „Swarm Intelligence“ genannt. Das Ziel dieser Arbeit ist die Entwicklung eines Konzepts, das die zukünftige Interaktion mehrerer Multicopter innerhalb des Schwarms beschreiben soll. Dabei soll die Intelligenz dieses Schwarms nicht in den einzelnen Mitgliedern angesiedelt sein, sondern in dem Kollektiv, zu dem jeder Multicopter seinen Beitrag leistet.

Da die vollständige Umsetzung der umfangreichen Thematik der Schwarmintelligenz innerhalb einer Studienarbeitsphase nicht realisierbar ist, wurde die Aufgabenstellung auf einige zentrale und umsetzbare Punkte beschränkt. Zu diesen Punkten zählt zum einen die Definition eines Kommunikationsframeworks, das die Art und Weise der Kommunikation zwischen den Schwarmmitgliedern beschreibt. Zum anderen sollen Konzepte entwickelt werden, die zukünftige Einsatzszenarien eines Multicopter-Schwarms und die Umsetzung einer kommandierten Mission definieren.

Die Entwicklung der Systemarchitektur soll ebenfalls Thema dieser Studienarbeit sein, da diese als Grundlage für den System- und Softwareentwurf, sowie zur Verschaffung eines umfassenden Überblicks über die Hardware und die Signalverläufe des Multicopters dient.

Des Weiteren soll eine Simulation mit Hilfe von Matlab/Simulink programmiert werden, die den Mittelpunkt dieser Studienarbeit darstellt. Durch die Simulation soll die geschriebene Software, präziser ausgedrückt die on-board Intelligenz der Multicopter, verifiziert werden, sodass diese gefahrlos im realen Multicopter verwendet werden kann. Um die Funktionsfähigkeit beweisen zu können, müssen unter anderem die Dynamik des Multicopters, die Kommunikationsschnittstellen und die Software in eine gemeinsame Umgebung implementiert werden. Die Anforderungen an die Software bestehen aus dem Vorhandensein von Mechanismen zur Kollisionsvermeidung und das Zerteilen der Flugbahn zum Zielpunkt in mehrere Abschnitte. Dadurch soll der Einfluss von äußeren Effekten, wie Wind, auf dem gradlinigen Anflug zum Zielpunkt reduziert werden.

Zuletzt sollen ebenfalls Algorithmen entwickelt werden, die zur Erkennung von Zielen und Hindernissen dienen sollen.

3 Theoretische Grundlagen

Im Folgenden werden einige theoretische Grundlagen näher erläutert, welche Schwärme und damit verbundene Team- und Gruppenaufgaben näher definieren. Dazu gehört beispielsweise zu erörtern, nach welchen Regeln und mit welchen Verhaltensweisen sich Schwärme bilden und aus welchen Gründen dies sinnvoll erscheinen kann.

Grundlegend haben Schwärme ein regelbasiertes Verhalten, welches auf Wechselwirkung der einzelnen Schwarmmitglieder beruht. Durch die richtige Anwendung der Wechselwirkungen kann die Effizienz des Gesamtsystems im Vergleich zu der Leistung einzelner Teilnehmer stark ansteigen. [5]

Dieses regelbasierte Verhalten wird im Computermodel „Boids“ von Craig Reynolds näher erläutert, da es die Grundprinzipien von Schwärmen algorithmisch und bildlich darstellt. Weiterhin ist es neben den Regeln wichtig zu erkennen, welche Phasen ein Schwarm bei einer gestellten Aufgabe durchlaufen muss, um eine Problemlösung möglichst schnell und koordiniert zu veranlassen. Ein gutes Modell bietet hierfür das Phasenmodell nach Tuckman, welches in dieser Arbeit speziell auf den Einsatz für das Schwarmverhalten von UAS angepasst wird.

3.1 Schwarmverhalten nach Reynolds

Craig Reynolds ist eine Koryphäe auf dem Gebiet der Simulation von komplexen, in der Natur vorkommenden Phänomenen, wie beispielsweise dem Schwarmverhalten von Meerestieren und Vögeln. In seiner 1986 erschienenen Computersimulation „Boids“ beschreibt Reynolds mit drei einfachen Steuermechanismen das grundlegende Verhalten von Schwärmen. Diese drei Verhaltensweisen sind in Abbildung 3-1, sowie in den folgenden Unterkapiteln näher dargestellt. [6]

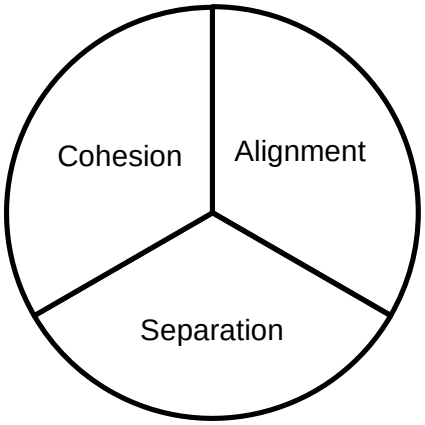

Abbildung 3-1: Schwarmverhalten nach Craig Reynolds

3.1.1 Kohäsion

Eine Grundlegende Eigenschaft von Schwärmen ist das gemeinsame Auftreten als Gruppe mehrerer Individuen. Um dies zu erreichen, müssen sich die Teilnehmer immer in einem gewissen Umkreis befinden, da sich sonst die Gruppenbeziehung auflösen würde. Ein Leitsatz, um diese Beziehung aufrecht zu erhalten, lautet: „Bewege dich in Richtung des Mittelpunkts derer, die du in deinem Umfeld siehst."

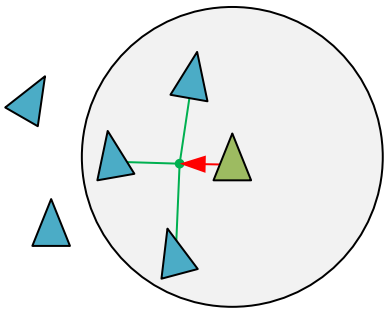

Abbildung 3-2: Darstellung des Kohäsionsverhaltens innerhalb eines Schwarms

3.1.2 Separation

Im gleichen Zuge dürfen sich die Teilnehmer jedoch auch nicht zu nahe kommen, daher gilt gleichzeitig zum Kohäsionsverhalten der Separationssatz: „Bewege dich weg, sobald dir jemand zu nahe kommt." – Mit Hilfe dieses Verhaltens soll sichergestellt

werden, dass es zu keiner Kollision der Schwarmteilnehmer kommt. Hierfür muss jedem Mitglied sein nahes Umfeld bekannt sein, sei es durch Kommunikation oder aktive Wahrnehmung.

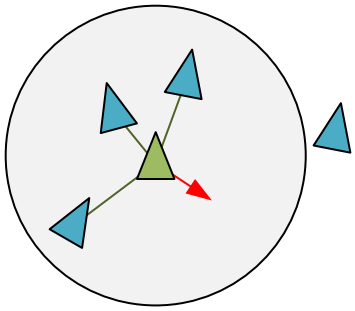

Abbildung 3-3: Darstellung des Separationsverhaltens innerhalb eines Schwarms

3.1.3 Ausrichtung

Die dritte Regel des von Reynolds aufgestellten Modells ist das Ausrichtungsverhalten der Schwarmteilnehmer. Nach dem Motto „Bewege dich in etwa in dieselbe Richtung wie deine Nachbarn“ wird verhindert, dass die Kohäsion der Teilnehmer unterbrochen wird, da sich alle Mitglieder trotz Bewegung grob in die gleiche Richtung bewegen müssen.

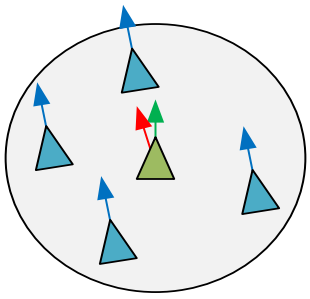

Abbildung 3-4: Darstellung des Ausrichtungsverhaltens innerhalb eines Schwarms

Diese Regeln alleine schaffen jedoch keine Effizienzsteigerung des Schwarmes, da so nur ein Zusammenhalt, beziehungsweise eine Kollisionsfreiheit aufrechterhalten wird. Um das Kollektiv sinnvoll zu nutzen müssen anstehende „Schwarmtätigkeiten“ korrekt und für alle Mitglieder erkenntlich kommuniziert und aufgeteilt werden. Ein Modell für diesen Planungsablauf bietet das Phasenmodell nach Tuckman.

3.2 Phasen der Schwarminteraktion

Das Phasenmodell nach Tuckman beschreibt eine abstrakte Möglichkeit einer Teambildungsmaßnahme, um in arbeitsteiliger Verantwortung ein gemeinsames Ziel zu erreichen. Hierfür wird der Arbeitsablauf, wie in Abbildung 3-5 dargestellt, in fünf Phasen eingeteilt. Zu beachten ist, dass das hier beschriebene Modell bereits direkt an die Schwarmintelligenz von UAS adressiert ist und damit vom grundlegenden Phasenmodell nach Tuckman abweicht.

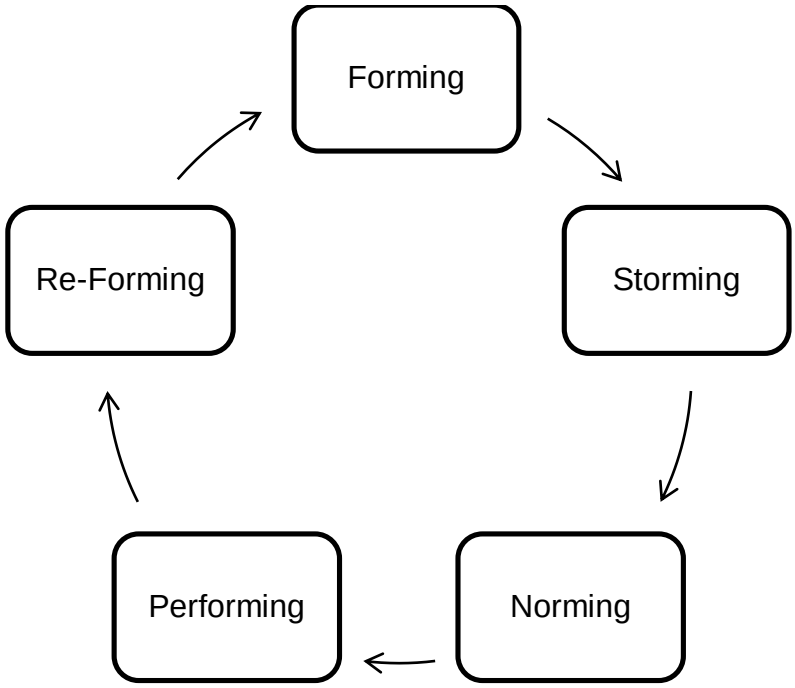

Abbildung 3-5: Angepasstes Phasenmodell nach Tuckman

In der Regel beginnen die einzelnen Einheiten in der *Forming*-Phase, in der das Kollektiv gebildet wird. Hier wird der momentane Zustand der einzelnen Teilnehmer kommuniziert. Alle UAS geben ihre derzeitige Konfiguration, als auch ihre Position bekannt, wodurch jedes Mitglied des Schwarmes über das momentane Umfeld im Klaren ist.

Die *Storming*-Phase dient der Ressourcen- als auch der Ablaufplanung. Da die Konfiguration jedes Teilnehmers bekannt ist, können anhand der Daten die jeweils besten Akteure für die geforderte Aufgabe gesucht und etwaige Signallaufpläne, wie es beispielweise bei Signalketten der Fall sein könnte, erstellt werden. Auch spezifische Flugrouten, sofern denn explizit gefordert, können hier definiert werden.

In der anschließenden *Norming*-Phase werden die einzelnen Aufgaben fest auf die Mitglieder des Schwarmes verteilt.

Das eigentliche Ausführen der Aufgabe wird in der *Performing*-Phase vollbracht. Dabei übt jedes UAS nur die Tätigkeit aus, in die es vom Kollektiv eingeteilt wurde, bis die Aktion beendet ist oder eine Änderung innerhalb des Schwarmes auftritt.

Abschließend wird die Gruppenbeziehung der UAS in der *Re-Forming*-Phase aufgelöst, womit sich der Schwarm wieder in seine Mitglieder aufteilt und auf weitere Anweisungen wartet.

Durch dieses definierte Vorgehen im Schwarm wird sichergestellt, dass das Kollektiv von den Stärken eines einzelnen profitiert, ohne dabei die Gruppenbeziehung untereinander zu definieren.

4 Softwarearchitektur des PCC-Modells

Das Kapitel *Softwarearchitektur* soll das Design der zu implementierenden Software des Payload Control Computers widergeben. Hierbei soll auf zwei Architekturansätze der Designfindungsphase eingegangen, deren Vor- und Nachteile genannt und schließlich die finale Designauswahl bewertet werden.

4.1 Simulink Blockansatz

Der Basisansatz des Softwaredesigns geht von zwei Stufen der Softwareroutine aus. Hierbei wird angenommen, dass zuerst ein Point Of Interest (POI) anhand einer definierten Logik ausgewählt und anschließend, in Teilstrecken zerlegt, in den Flugbahnspeicher abgelegt wird (vgl. Abbildung 4-1, weiß markiert).

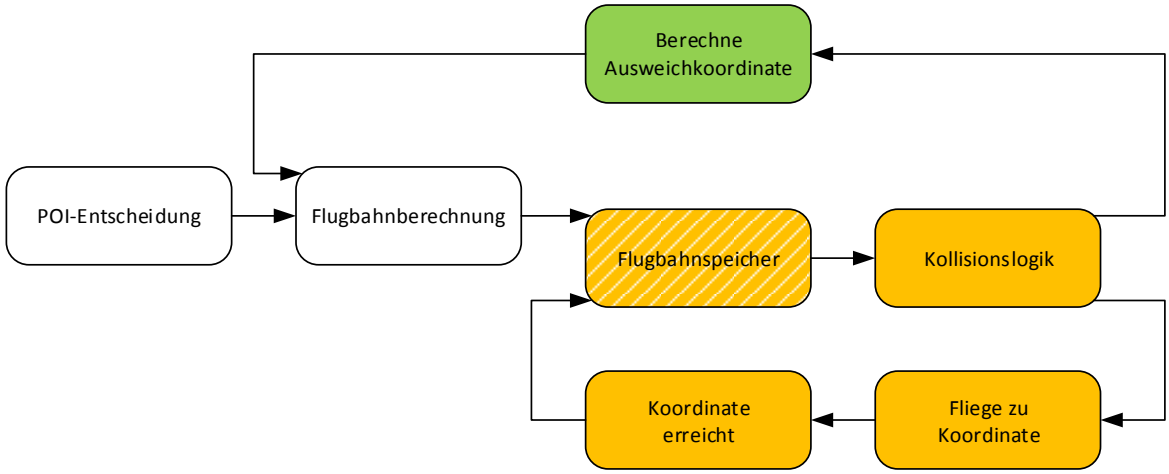

Abbildung 4-1: Softwarearchitektur Basisansatz

Von diesem Block aus wird die zweite Stufe der Architektur (in der Abbildung orange eingezeichnet) abgearbeitet. Ein Punkt des Flugbahnspeichers wird entnommen, woraufhin über einen Kollisionsblock der Flugpfad auf eine mögliche Kollision mit einem Hindernis überprüft werden soll. Sofern keine Kollisionsgefahr für das UAS besteht, wird der Zielpunkt angeflogen. Bei Erreichen der vorgegebenen Koordinate wird der Zeiger, der auf dem momentanen Punkt des Flugbahnspeichers steht um eine Zelle erhöht, wodurch sich der orangene Ablauf wiederholt, bis der definierte POI erreicht ist.

Unterbrochen werden kann dieser Vorgang durch die Kollisionslogik, welche über einen zweiten Pfad eine Neuberechnung der Flugbahn veranlassen kann (grün markiert). Durch das Aufrufen des Ausweich-Algorithmus wird mit Hilfe der bekannten Positions- und Geschwindigkeitsdaten des fremden Objektes eine Ausweichkoordinate berechnet, welche vor dem eigentlichen POI angeflogen werden soll und somit ein sicheres Fliegen ermöglicht.

Eine vorzeitige, konzeptionelle Implementierungsstufe in Matlab/Simulink ist in Abbildung 4-2 dargestellt.

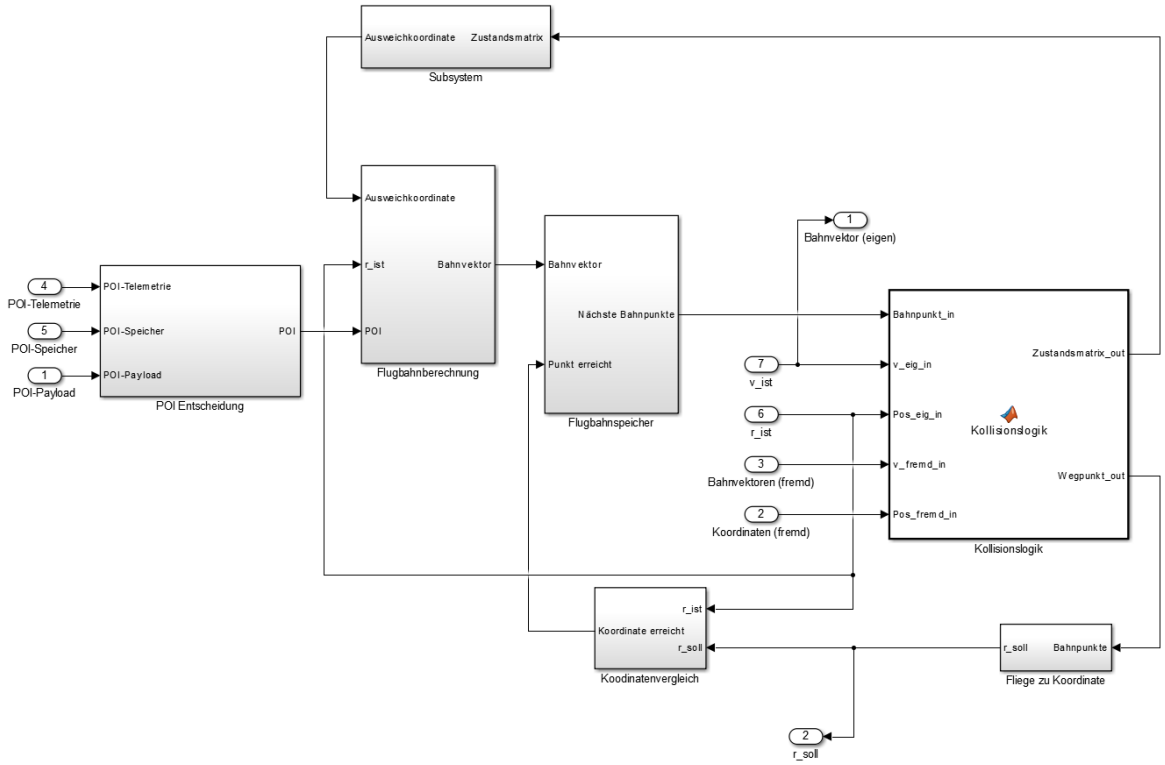

Abbildung 4-2: Implementierungsbeispiel des Simulink Blockansatzes

Hierbei wird der Modellansatz sehr gut umgesetzt, sodass sich zwischen Ansatz und Implementierung auf hoher Designebene nur wenige Unterschiede ergeben. Entscheidende Nachteile ergeben sich jedoch bei der Implementierung in der Low-Level Designebene. Im Folgenden sollen einige dieser Mängel aufgeführt werden.

1. Startwertproblem beim parallelen Programmablauf

   Ein grundlegendes Problem bei der Verwendung von vielen Subsystemen in Matlab/Simulink ist die Definition des Startwerts: Da Simulink versucht, in einem Simulationszeitschritt die Ausgabewerte aller Blöcke zu berechnen, benötigen diese

auch valide Eingabewerte im entsprechenden Zeitschritt. Jedoch sind diese durch den im Designansatz definierten, sequentiellen Programmablauf in Simulink nicht immer sofort bei Simulationsstart verfügbar, sofern sich die Simulation ohne Startwerte überhaupt starten ließe. Somit müsste man von vornherein feste, den fehlerfreien sequentiellen Ablauf nicht störende Startwerte definieren, oder eine zeitlich verspätetes Triggern (sprich Aktivieren) der einzelnen Blöcke in Betracht ziehen.

2. Gezieltes Triggern der Simulinkblöcke

   Damit ergibt sich das zweite Problem dieses Designansatzes: Nicht jeder Funktionsblock des Simulinkmodells wird zu jeder Zeit benötigt, ein Paradebeispiel ist hierfür der *Berechne_Ausweichkoordinate*-Block. Dieser wird nur benötigt, wenn sich ein Hindernis auf dem Flugpfad befindet und eine Kollision bevor steht. Eine andauernde Berechnung von Ausweichkoordinaten würde somit unnötig Rechenleistung des PCCs verbrauchen. Ein gezieltes Triggern wird hier von Nöten, wobei im parallelen Simulationsablauf ständig Daten an den Eingängen der Funktionsnachfolger liegen müssen. Somit ergibt sich hier ein ähnlicher Konflikt wie bei Nachteil Nummer eins, es müssten fest definierte Alternativwerte zur Verfügung stehen, wenn ein Block nicht aktiv ist und keine Ausgangswerte zur Verfügung stellt.

3. Große Anzahl persistenter Variablen

   Durch das prinzipiell unregelmäßige, zufällige triggern einzelner Funktionsblöcke müssen nachfolgende Subsysteme immer den momentanen Eingabewert mit dem Wert des vorhergehenden Zeitschrittes vergleichen, um auf neue Eingangsdaten zu reagieren. Es muss also immer ein großer Datensatz zwischengespeichert werden, da ansonsten, wie es im Beispiel bei der Interaktion des POI-Entscheidungs- mit dem Flugbahnberechnungs-Block der Fall ist, ständig eine komplett neue Flugroute und damit auch ein komplett neuer Datensatz berechnet werden würde.

Aufgrund dieser Nachteile wird im Folgenden der Ansatz, die gesamte PCC-Logik als reinen Matlab Code zu implementieren, welcher aber in Simulink funktions- und autocodingfähig ist, näher erläutert.

4.2 Matlab-Function Codeansatz

Viele der in 4.1 genannten Probleme liegen dem Ansatz zugrunde, dass die Funktionalitäten des Modells mehr sequentiell als parallel ablaufen. Gerade diese Mängel lassen sich dadurch beheben, dass die Systemarchitektur nicht auf Blockbasis aufbaut, sondern einen Codeansatz wählt wird. Der große Vorteil liegt hierbei, dass Simulink pro Zeitschritt den gesamten Codeblock sequentiell durchrechnet und dadurch Parallelitäten vermieden werden.

In der folgenden Abbildung 4-3 ist der sequentielle Ablauf der Software dargestellt. Im Vergleich zur Darstellung im vorhergehenden Kapitel ist dieser Designansatz bereits viel ausgereifter und tiefergehend entwickelt, wodurch sich ein Plus an Funktionseinheiten ergibt. Die grundlegenden Funktionen finden sich aber auch hier wieder.

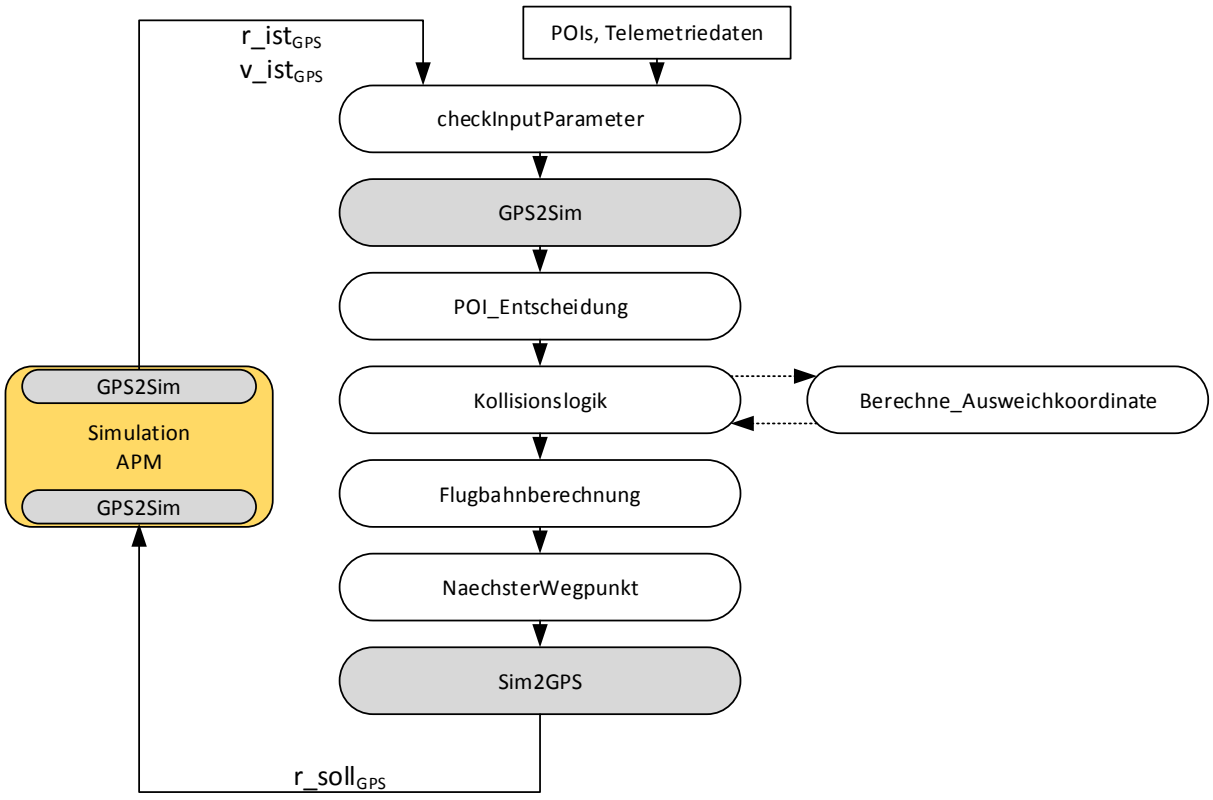

Abbildung 4-3: Sequentielle Matlab-Function Softwarearchitektur

Der innerste Kern des sequentiellen Programmablaufs befindet sich auf vertikaler Ebene zwischen den weißen Elementen *POI_Entscheidung* und *NaechsterWegpunkt*. Die Funktionen der einzelnen Stufen sind bereits aus 4.1 bekannt, mit Ausnahme von

NaechsterWegpunkt. Dieser Block gibt dem APM (in Blockschaubild orange dargestellt) eine neue Zielkoordinate vor. Auf die Ausweichfunktion kann durch eine einfache *wenn/dann*-Abfrage zugegriffen werden, wodurch auch hier eine dauerhafte Konsistenz der benötigten Startwerte gegeben ist. Die grauen Elemente dienen bereits der Portierung auf GPS-Koordinaten und können für das Top-Level Softwaredesign vorerst vernachlässigt werden.

Die in Abbildung 4-3 beschriebenen Sequenzen können nun folglich als Matlab-Code in einem gemeinsamen Block implementiert werden, wie es in Kapitel 5 beschrieben wird.

5 Implementierung des Modells in Matlab/Simulink

In diesem Kapitel werden die implementierten Funktionen der Flight Intelligence detailiert erläutert, um die Funktionsweise der Algorithmik nachzuvollziehen.

Vorab ist zu erwähnen, dass hinter der Flight Intelligence eine *Main*-Funktion steckt, die die im Folgenden beschriebenen Funktionen nacheinander aufruft. Des Weiteren werden dort persistente Variablen definiert, sodass in einem Durchlauf bestimmte Parameter aus dem vorherigen Druchlauf zu Vergleichszwecken zur Verfügung stehen. Zu diesen Variablen zählen zum einen die *Flugmatrix*, deren Inhalt in Abbildung 5-1 zu sehen ist und der sogenannte *Counter*, der angibt welche Koordinate, bestimmt durch die Zeile der *Flugmatrix, momentan angeflogen wird*.

$$Flugmatrix = \begin{bmatrix} Ausweichkoordinate \\ Zielkoordinate \\ Position\ des\ fremden\ Objekts \\ Geschwindigkeitsvektor\ des\ fremden\ Objekts \\ Zwischenziel\ 1 \\ Zwischenziel\ 2 \\ \dots \\ Zwischenziel\ 25 \end{bmatrix} \left.\begin{matrix} \\ \\ \\ \\ \\ \\ \\ \\ \end{matrix}\right\} \text{Davon max. 5 Zwischenziele zur Ausweichkoordinate}$$

Abbildung 5-1: Darstellung des Inhalts der Flugmatrix

In dieser *Flugmatrix* sind alle Informationen der momentan verfolgten Flugbahn enthalten, wie die Koordinate eines möglichen Ausweichpunktes, des Zielpunktes und der Zwischenziele zum Ziel. Des Weitern sind die Koordinate und der Geschwindigkeitsvektor des kritischsten Objekts beinhaltet. Die Bestimmung dieses kritischsten Objekts wird in Abschnitt 5.2 beschrieben.

5.1 POI_Entscheidung

Die Funktion *POI_Entscheidung* ermittelt aufgrund ihrer Eingabeparameter, welche Zielkoordinate vom UAS angeflogen werden soll. Dazu werden diese einer Priorisierung unterworfen.

5.1.1 Ein- und Ausgabeparameter

Eingabeparameter

- $Telemetrie\ (Form: 1x3)$: $[Longitude \quad Latitude \quad Height]$

  Der Eingabeparameter $Telemetrie$ beinhaltet die Zielkoordinate, die von der Bodenstation über die Datenverbindung kommandiert wird.

- $Payload\ (Form: 1x3)$: $[Longitude \quad Latitude \quad Height]$

  Der Eingabeparameter $Payload$ beinhaltet die Zielkoordinate, die von der Payload (Kamera) vorgegeben wird, um ein Objekt zu verfolgen.

- $Missionsspeicher\ (Form: nx3)$: $\begin{bmatrix} Longitude_1 & Latitude_1 & Height_1 \\ & \dots & \\ Longitude_n & Latitude_n & Height_n \end{bmatrix}$

  Der Eingabeparameter $Missionsspeicher$ beinhaltet eine Anzahl von *n* Koordinaten, die das UAS anfliegen soll und die vor dem Flug auf den Speicher geschrieben worden sind. Dabei steht in der ersten Zeile die „Home-Koordinate“ und in den restlichen die verschiedenen Zielkoordinaten.

- $Counter\ (Form: 1x1)$

  Der Eingangsparameter $Counter$ beschreibt die Zeile des Missionsspeichers, die momentan als POI gesetzt ist.

- $Position_{eigen}\ (Form: 1x3)$: $[Longitude \quad Latitude \quad Height]$

  Der Eingabeparameter $Position_{eigen}$ beinhaltet die momentane GPS-Position des eigenen UAS.

Ausgabeparameter

- $Zielkoordinate\ (Form: 1x3)$: $[Longitude \quad Latitude \quad Height]$

  Der Ausgabeparameter $Zielkoordinate$ beinhaltet die Zielkoordinate, die als nächstes angeflogen werden soll.

- $Counter\ (Form: 1x1)$

  Der Eingangsparameter $Counter$ beschreibt die Zeile des Missionsspeichers, die momentan als POI gesetzt ist.

5.1.2 Funktionsbeschreibung

Die Entscheidung welcher Zielpunkt (im folgenden POI genannt) angeflogen wird, unterliegt der Priorisierung bestimmter Eingabeparameter. Umgesetzt wurde dies im Matlabcode mittels einer *if-elseif* - Abfrage.

1. Wenn von der Bodenstation eine Koordinate kommandiert wurde, soll diese als POI gelten.
2. Wenn ein POI von der Payload generiert wurde, soll dieser angeflogen werden.
3. Andernfalls werden die POIs angeflogen, die im Missionsspeicher hinterlegt sind. Um festzustellen welcher der POIs aus dem Speicher momentan als Zielkoordinate aktiv ist, wird der Counter herangezogen. Anschließend wird der Abstand zwischen der Zielkoordinate und der momentanen Position des UAS berechnet. Wenn dieser kleiner als 2m ist, wird die Speicherzelle des Missionsspeichers, sprich der Counter, inkrementiert und eine neue Zielkoordinate auf den Ausgang gegeben. Falls dem nicht so ist, wird der momentane POI beibehalten.

5.2 Kollisionslogik

Die Funktion *Kollisionslogik* entscheidet über einen Algorithmus, der in Abschnitt 5.2.2 ausführlich beschrieben wird, ob sich ein Objekt auf Kollisionskurs befindet. Wenn dies der Fall ist, wird eine Funktion aufgerufen, die eine Ausweichkoordinate berechnet, sodass die prognostizierte Kollision verhindert werden kann.

5.2.1 Ein- und Ausgabeparameter

<u>Eingabeparameter:</u>

- $Vektor_{eigen}\ (Form: 1x3)$: $[Longitude \quad Latitude \quad Height]$

Der Eingabeparameter $Vektor_{eigen}$ ist der momentane Geschwindigkeitsvektor des eigenen UAS, der vom APM generiert wird.

- $Position_{eigen}\ (Form: 1x3)$: $[Longitude \quad Latitude \quad Height]$

  Der Eingabeparameter $Position_{eigen}$ beinhaltet die momentane GPS-Position des eigenen UAS

- $Vektor_{fremd}\ (Form: nx3)$: $\begin{bmatrix} Longitude_1 & Latitude_1 & Height_1 \\ & \dots & \\ Longitude_n & Latitude_n & Height_n \end{bmatrix}$

  Der Eingabeparameter $Vektor_{fremd}$ beinhaltet die Geschwindigkeitsvektoren aller registrierten fremden Objekte, zu denen statische und dynamische Hindernisse zählen.

- $Position_{fremd}\ (Form: nx3)$: $\begin{bmatrix} Longitude_1 & Latitude_1 & Height_1 \\ & \dots & \\ Longitude_n & Latitude_n & Height_n \end{bmatrix}$

  Der Eingabeparameter $Position_{fremd}$ beinhaltet die Positionen aller eben erwähnten fremden Objekte.

- $Flugmatrix\ (Form: 29x3)$: $\begin{bmatrix} Longitude_1 & Latitude_1 & Height_1 \\ & \dots & \\ Longitude_{29} & Latitude_{29} & Height_{29} \end{bmatrix}$

  Der Eingabeparameter $Flugmatrix$, wurde im vorherigen Durchlauf definiert und gespeichert. Er enthält die Informationen über die momentan verfolgte Flugbahn, die durch die in Abbildung 5-1 zu sehenden Koordinaten definiert ist.

<u>Ausgabeparameter</u>

- $Ausweichkoordinate\ (Form: 1x3)$: $[Longitude \quad Latitude \quad Height]$

  Der Ausgabeparameter $Ausweichkoordinate$ enthält die Positions- und Höhenangabe des berechneten Ausweichpunkts.

- $Data_{fremd}\ (Form: 2x3)$: $\begin{bmatrix} Longitude_1 & Latitude_1 & Height_1 \\ Longitude_2 & Latitude_2 & Height_2 \end{bmatrix}$

  Der Ausgabeparameter $Data_{fremd}$ beinhaltet die Geschwindigkeits- und Positionsinformation über das kritischste fremde Objekt.

  1. Zeile: Fremder Geschwindigkeitsvektor
  2. Zeile: Fremde Position

5.2.2 Funktionsbeschreibung

Das Konzept oder die Idee, die hinter der Kollisionslogik steckt, ist in Abbildung 5-2 dargestellt.

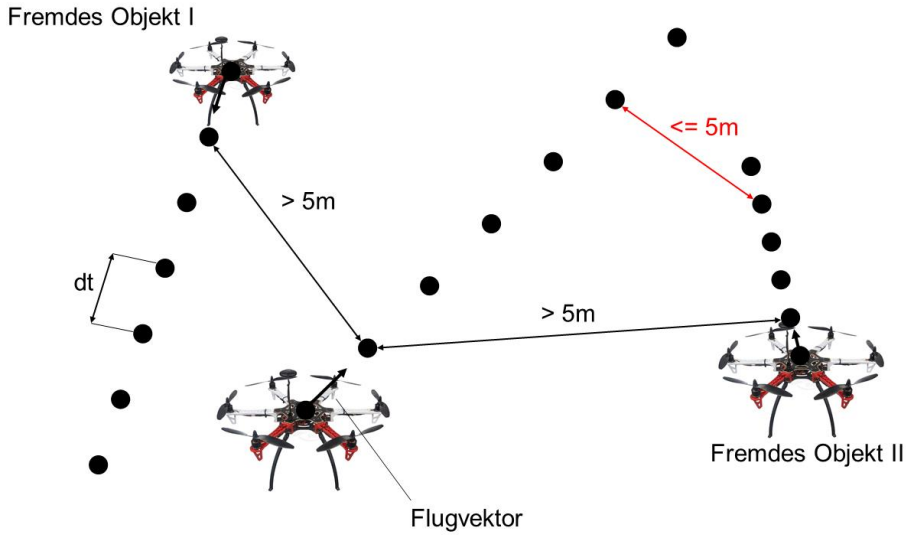

Abbildung 5-2: Konzept der Kollisionslogik

In Abhängigkeit der momentanen Position und des momentanen Geschwindigkeitsvektors des eigenen UAS und der fremden Objekte, werden die zukünftigen Positionen berechnet. Die Distanz zwischen zwei Punkten eines Objekts ist direkt von der Geschwindigkeit abhängig, sodass eine höhere Geschwindigkeit zu einer größeren Distanz zwischen zwei Punkten führt. Anschließend werden die Abstände der jeweils ersten, zweiten, dritten und darauf folgenden Punkte des eigenen UAS zu den fremden Objekten bestimmt. Wenn der Abstand zwischen zwei Punkten den Mindestabstand von 5m unterschreitet, wird eine Kollisionswarnung erzeugt, die zur Berechnung einer Ausweichkoordinate führt.

Im Matlabcode ist dieses Konzept folgendermaßen umgesetzt worden: Um eine zukünftige Position bestimmen zu können, muss zu der momentanen Position ein Abstandswert dazu addiert werden, der abhängig von der Geschwindigkeit und einem definierten Zeitraum ist, da der physikalische Zusammenhang

$$Strecke = Geschwindigkeit \cdot Zeit \qquad (5.1)$$

gilt. Als Zeitschritt dt zwischen zwei Punkten wurden 0,25s gewählt und es sollen die 20 nächsten Bahnpunkte (zukünftige Positionen) berechnet werden, sodass der Weg,

der in den nächsten 5 Sekunden zurück gelegt wird, auf Hindernisfreiheit überprüft werden kann. Zur Berechnung der nächsten Flugbahnpunkte wurde eine *for*-Schleife implementiert, die 20-mal durchlaufen werden soll. In jedem Durchlauf wird ein zukünftiger Bahnpunkt über die Formel (5.2) berechnet.

$$Bahnpunkte(n) = Position + (n-1) \cdot dt \cdot Vektor \qquad (5.2)$$

Dabei ist n der Schleifenparameter, $Position$ die momentane Position, dt der angesprochene Zeitschritt von 0,25s und $Vektor$ der Geschwindigkeitsvektor. In dieser *for*-Schleife werden die Bahnpunkte für alle Objekte (eigenes UAS und fremde Objekte) bestimmt, sodass für jedes Objekt ein Bahnvektor der Größe 20x1 erzeugt wird.

Anschließend werden die Flugbahnen mittels einer *While*-Schleife auf eine mögliche Kollision hin untersucht. In dieser Schleife werden die Abstände zwischen eigenem UAS und den fremden Objekten beim ersten, zweiten, dritten und den darauffolgenden Punkten über die Formel (5.3) bestimmt.

$$Abstand = \left| Bahnpunkte_{eigen}(n) - Bahnpunkte_{fremd}(n) \right| \qquad (5.3)$$

Wenn der Abstand zwischen zwei Punkten kleiner als fünf Meter sein sollte, führt dies zu einer Kollisionswarnung. Die Schleife wird maximal 20-mal durchlaufen, sodass jeder Punkt der Flugbahn auf Kollisionsgefahr untersucht wird. Kommt es jedoch beispielsweise beim fünften Punkt zu einer Unterschreitung des Mindestabstands von fünf Metern, wird die Schleife über eine Abbruchbedingung verlassen und eine Kollisionswarnung ausgelöst. Außerdem wird durch diese Schleife das kritischste fremde Objekt bestimmt, das die geringste Distanz zum eigenen UAS aufweist.

Im Anschluss an diesen Vorgang befinden sich mehrere *if*-Abfragen, um das weitere Vorgehen zu bestimmen. Zuallererst wird abgefragt, ob es sich bei dem ermittelten kritischsten Objekt um das gleiche wie beim vorherigen Durchlauf der *Main*-Funktion handelt. Diese Information wird aus der dritten und vierten Zeile des Eingangsparameters *Flugmatrix* generiert.

1. Verschiedene Objekte:

   Es wird eine *Zustandsmatrix* generiert, die die Position und den Geschwindigkeitsvektor des eigenen UAS und des kritischsten Objektes enthält. Dieser Parameter wird an die Funktion *Berechne_Ausweichkoordiante* übergeben, die

eine *Ausweichkoordinate* berechnet. Außerdem wird der zweite Ausgabeparameter $Data_{fremd}$ mit den Informationen (Position und Geschwindigkeit) über das kritischste fremde Objekt erzeugt.

2. Identische Objekte:

   Falls es sich um dasselbe Objekt handelt, wird geprüft, ob in der momentanen *Flugmatrix* bereits eine Ausweichkoordinate hinterlegt ist (erste Zeile) und ob eine Kollisionswarnung in der *While*-Schleife erzeugt wurde.

   1. Keine Ausweichkoordinate & Kollisionswarnung

      Die *Zustandsmatrix* wird generiert und von *Berechne_Ausweichkoordiante* eine Ausweichkoordinate berechnet. $Data_{fremd}$ wird mit den Informationen über das kritischste fremde Objekt gefüllt.

   2. Alle anderen Fälle

      Ausweichkoordinate und $Data_{fremd}$ werden mit NaN gefüllt, sodass in später ausgeführten Funktionen keine neue *Flugmatrix* berechnet werden muss.

5.3 Berechne_Ausweichkoordinate

Die Funktion *Berechne_Ausweichkoordinate* wird im Falle einer Kollisionswarnung aufgerufen, um eine von dem fremden und vom eigenen Objekt abhängige Ausweichkoordinate zu berechnen, sodass eine Kollision vermieden wird.

5.3.1 Ein- und Ausgabeparameter

Eingabeparameter:

- $Zustandsmatrix\ (Form: 4x3):\begin{bmatrix} Vektor_{eigen,Lat} & Vektor_{eigen,Lon} & Vektor_{eigen,Alt} \\ Position_{eigen,Lat} & Position_{eigen,Lon} & Alt_{eigen} \\ Vektor_{fremd,Lat} & Vektor_{fremd,Lon} & Vektor_{fremd,Alt} \\ Position_{fremd,Lat} & Position_{fremd,Lon} & Alt_{fremd} \end{bmatrix}$

Der Eingabeparameter $Zustandsmatrix$ enthält die für die Berechnung der Ausweichkoordinate relevanten Flugdaten über das eigene UAS, als auch die Daten für das von

der Kollisionslogik erkannte Hindernis. Die Informationen sind als Zeilenvektoren in der Matrix enthalten. Die

1. Zeile enthält den momentanen Flugvektor sowie die
2. Zeile die momentane Position des ausführenden UAS beinhaltet. Die
3. und 4. Zeile enthalten die oben genannten Flugdaten, jedoch für das fremde Objekt.

Ausgabeparameter

- $Ausweichkoordinate\ (Form: 1x3)$: $[Longitude \quad Latitude \quad Altitude]$

Der Ausgabeparameter *Ausweichkoordinate* enthält die Positions- und Höhenangabe des berechneten Ausweichpunkts. Falls die Ausweichlogik keine Ausweichkoordinate errechnet, da beispielsweise der Abstand zum Hindernis größer als der definierte Minimalabstand ist, so wird der Vektor *Ausweichkoordinate* mit „Not a Number“ befüllt.

5.3.2 Funktionsbeschreibung

Um eine sichere Ausweichkoordinate zu berechnen muss zuerst die genaue Lage des Hindernisses relativ zur eigenen Position erkannt werden. Dafür ist es hilfreich zu wissen, in welchem Winkel sich das fremde Objekt zum momentan eingehaltenen Flugvektor befindet. Erreicht wird dies durch Formel (5.4), welche den Winkel bereits in Grad [°] umgerechnet widerspiegelt.

$$winkel_{2D} = \frac{180}{\pi} \cdot \arccos\left(complex\left(\frac{Vekt_{eig2D} \circ Vekt_{zuHindernis2D}}{|Vekt_{eig2D}| \cdot |Vekt_{zuHindernis2D}|} \right) \right) \qquad (5.4)$$

Zu beachten ist hierbei, dass diese Formel den Winkel ohne genauen Lagebezug widergibt. Das Objekt könnte sich also sowohl rechts als auch links vom eigenen Flugvektor befinden. Um dieses Lageproblem zu lösen wird das Kreuzprodukt zwischen dem Flugvektor und dem Vektor zum Hindernis berechnet (vgl. (5.5)).

$$Lage_{2D} = Vekt_{eig} \times Vekt_{zuHindernis} \qquad (5.5)$$

Das Kreuzprodukt erzeugt einen Vektor, welcher senkrecht auf den beiden Eingangsvektoren steht und außerdem die Eigenschaft besitzt, mit diesen ein „Rechtshandsystem“ zu bilden. Ist die dritte Komponente des resultierenden Vektors $Lage_{2D}$

kleiner Null, so kann davon ausgegangen werden, dass sich das Objekt rechts der eigenen Flugbahn befindet. Diese Information wird auf die Variable $winkel_{2D}$ durch Änderung des Vorzeichens übertragen, positive Gradzahlen zeigen in die geodätisch positive Drehrichtung.

Anhand der so gewonnen Informationen kann man anschließend sechs mögliche Kollisionsfälle unterscheiden:

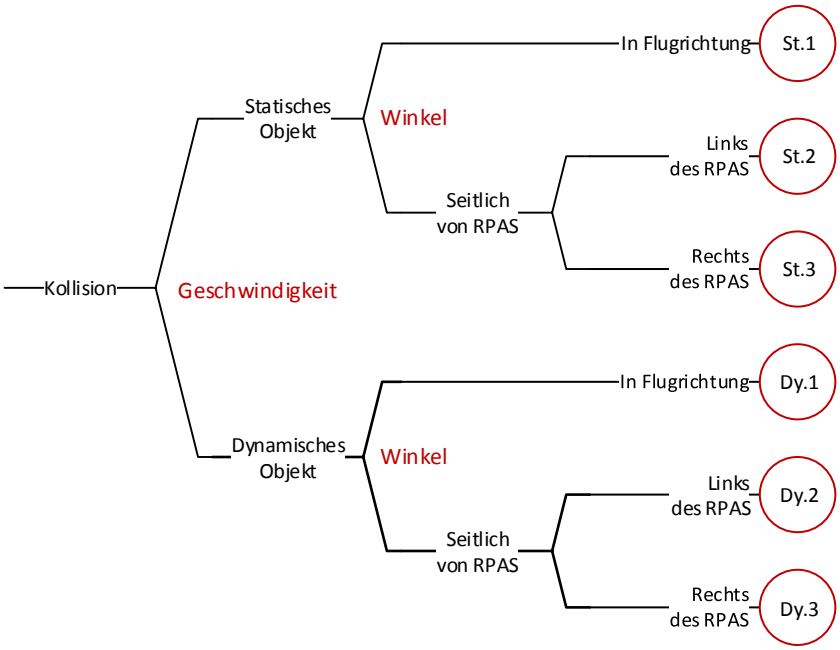

Abbildung 5-3: Kollisionsfälle

Dabei wird als grundlegendes Unterscheidungsmerkmal der Betrag des Flugvektors des fremden Objektes herangezogen. Überschreitet die Geschwindigkeit des Hindernisses einen definierten Wert, so wird dieses als dynamisch angesehen, im anderen Fall als statisch. Die zweite Charakterisierung ist durch den Winkel, dessen Generierung bereits ausführlich erklärt wurde, gegeben. Hierbei wird zwischen φ_{vorne} und $\varphi_{seitlich}$ unterschieden (vgl. Abbildung 5-4).

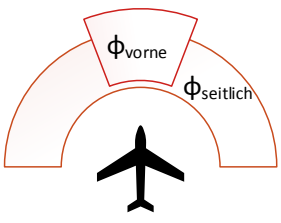

Abbildung 5-4: Winkelunterscheidung in der Kollisionslogik

Statische Fälle

Ist das Hindernis als statisch erkannt worden, wird als erstes Kriterium die relative Lage zur eigenen Position ausgewertet. Etwaige Höhendifferenzen werden von der gesamten Logik nicht beachtet.

Ist der Betrag des Winkels zwischen eigener Flugbahn und fremden Objekt weniger als φ_{vorne}, so wird eine Ausweichkoordinate anhand der Formel (5.6) generiert. Dies entspricht Fall **St.1**.

$$Ausweichkoord. = Pos_{fremd} + (1{,}35 \cdot abstand_{min}) \cdot \frac{[Vekt_{eig,Long} \quad -Vekt_{eig,Lat} \quad Alt]}{|[Vekt_{eig,Long} \quad Vekt_{eig,Lat} \quad Alt]|} \qquad (5.6)$$

Die Ausweichkoordinate liegt dabei, wie in Abbildung 5-5 zu sehen ist, immer rechts vom Hindernis und steht senkrecht auf dem Flugvektor.

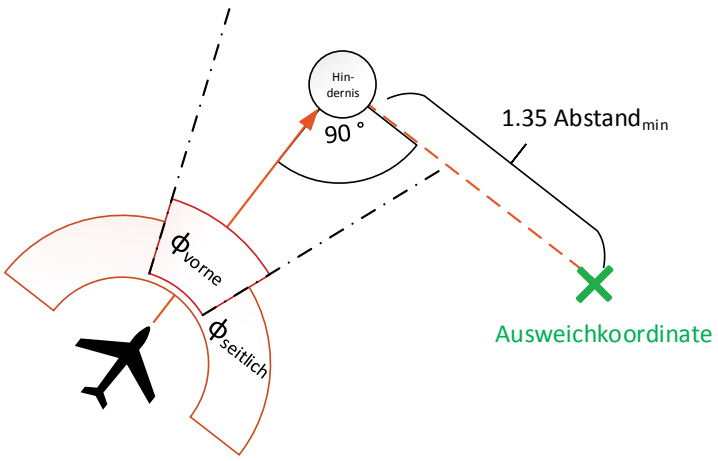

Abbildung 5-5: Berechne_Ausweichkoordinate Fall St.1

Der Abstand der Ausweichkoordinate zum Hindernis ist bei diesem Fall mit 1.35 * minimalem Abstand definiert. Die angenommene Trägheit des Multikopters liegt diesem Faktor zugrunde. Je weiter die Ausweichkoordinate der derzeitigen Flugbahn entfernt ist, umso stärker reagiert das UAS im Ausweichmanöver.

Hat der Winkel zum Hindernis einen größeren Betrag als φ_{vorne}, jedoch kleiner als $\varphi_{seitlich}$, so entspricht dies Fall **St.2** oder **St.3**, je nach Lage des Hindernisses (vgl. Abbildung 5-6).

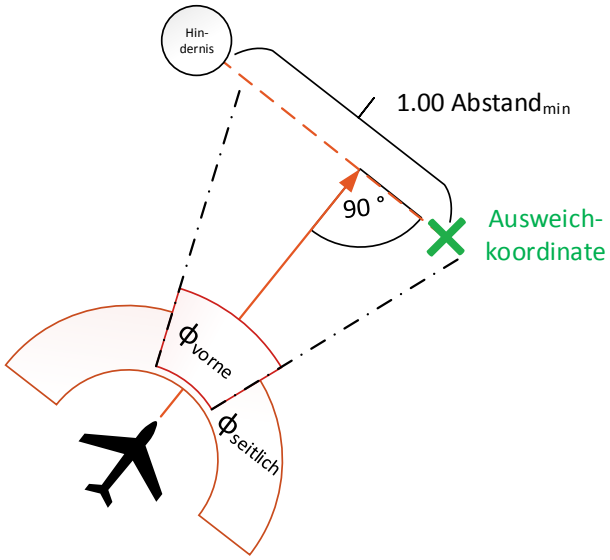

Abbildung 5-6: Berechne_Ausweichkoordinate Fall St.2/St.3

Die Berechnungsformel ist der der frontalen Konfrontation ähnlich, jedoch wird hier der Mindestabstand auf den einfachen minimalen Abstand belassen:

$$Ausweichkoordinate = Pos_{fremd} + abstand_{min} \cdot \frac{[Vekt_{eig,Long} \quad -Vekt_{eig,Lat} \quad Alt]}{|[Vekt_{eig,Long} \quad Vekt_{eig,Lat} \quad Alt]|} \qquad (5.7)$$

Sobald der errechnete Winkel zwischen Flugbahn und Hindernis den Wert $\varphi_{seitlich}$ übersteigt, werden keine weiteren Ausweichmaßnahmen vorgenommen.

Dynamische Fälle

Wenn das fremde Objekt eine Geschwindigkeit besitzt, die größer als der vordefinierte Unterscheidungswert ist, wird ein Fall des dynamischen Astes (**Dy.1** bis **Dy.3**) aktiviert. Beachtet werden muss hierbei, dass es sich bei den dynamischen Fällen um *experimentelle* Ausweichmanöver handelt, welche keinesfalls alle möglichen Kollisions-Variationen beinhaltet. Vielmehr stellt diese Funktion einen Ansatz der Kollisionsvermeidung für zukünftige, mit multiplen UAS ausgeführte Missionen dar, welche eine stetige Kommunikation untereinander aufweisen.

Die Parameter φ_{vorne} und $\varphi_{seitlich}$ sind auch in diesem Ast der Logik wieder vorhanden. Fliegen zwei UAS aufeinander zu (Winkel < φ_{vorne}), so wird in beiden der gleiche Fall **Dy.1** getriggert und beide weichen, wie in Abbildung 5-7 zu sehen ist, aus.

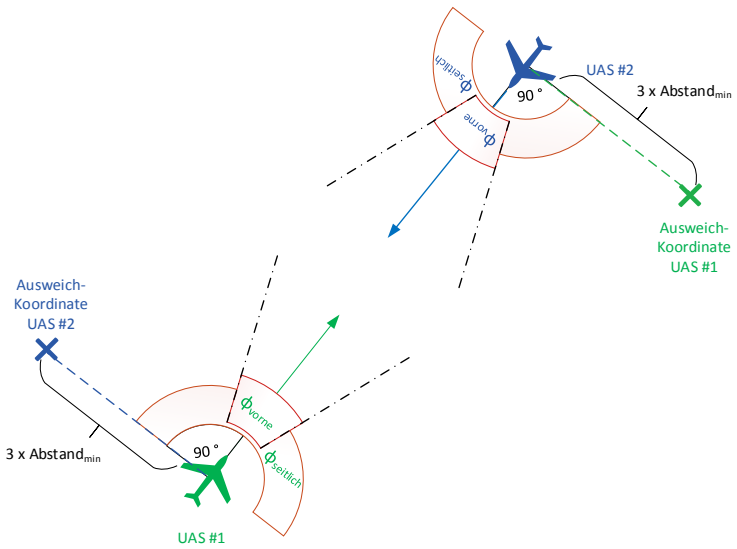

Abbildung 5-7: Berechne_Ausweichkoordinate Fall Dy.1: Parameter

Auch hier wird die Ausweichkoordinate wie in den statischen Fällen generiert, aufgrund der angenommenen Trägheit jedoch mit dem dreifachen Minimalabstand. Die beiden UAS fliegen dann wie in Abbildung 5-8 dargestellt aneinander vorbei, wodurch eine Kollision beider Teilnehmer vermieden wird.

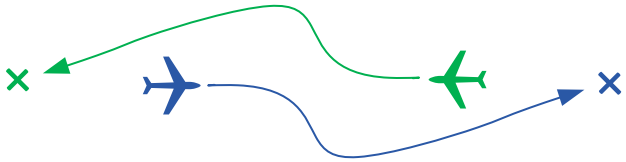

Abbildung 5-8: Berechne_Ausweichkoordinate Fall Dy.1: Flugbahnen

Als grundlegendes Modell für den seitlichen Fall ($\varphi_{vorne} < \varphi_{berechnet} < \varphi_{seitlich}$) wird das aus dem Straßenverkehr bekannte „rechts vor links“ genutzt, welches in ähnlicher Weise auch in der bemannten Luftfahrt eingesetzt wird [7]. Ein UAS, dessen Flugbahn von einem von rechts kommenden Hindernis geschnitten wird, muss dem Hindernis „Vorfahrt“ gewähren und daher selbstständig ausweichen. Dies bedeutet im gleichen Moment aber auch, dass sich das herannahende Hindernis (sofern es denn die gleiche Kollisionslogik besitzt) auf das Ausweichmanöver des von links kommenden UAS verlassen können muss. Abgebildet ist dieses Verhalten in Abbildung 5-9, als Mindestabstand wird der einfache Minimalabstand gewählt.

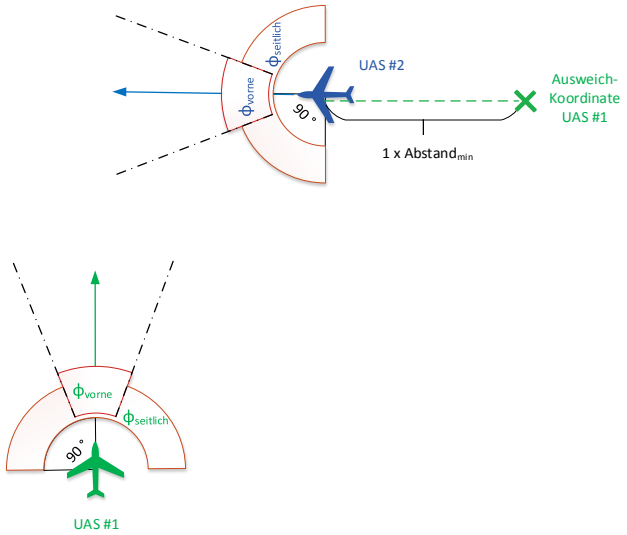

Abbildung 5-9: Berechne_Ausweichkoordinate Fall Dy.2 und Dy.3: Parameter

Die von beiden Teilnehmern geflogene Flugbahn ist in Abbildung 5-10 dargestellt.

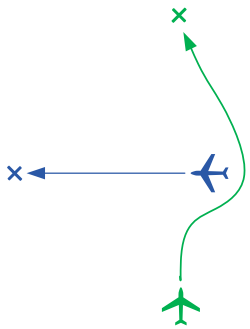

Abbildung 5-10: Berechne_Ausweichkoordinate Fall Dy.2 und Dy.3: Flugbahn

5.4 Flugbahnberechnung

Diese Funktion soll in Abhängigkeit der Eingabeparameter Zwischenziele auf dem Weg zum Zielpunkt berechnen, sodass das RPAS auf direktem Weg zum Ziel fliegen kann und nicht durch möglichen Wind von der Flugbahn abkommt.

5.4.1 Ein- und Ausgabeparameter

Eingabeparameter

- $Position_{eigen}\ (Form: 1x3)$: $[Longitude \quad Latitude \quad Height]$

  Der Eingabeparameter $Position_{eigen}$ beinhaltet die momentane GPS-Position des eigenen RPAS.
- $Zielkoordinate\ (Form: 1x3)$: $[Longitude \quad Latitude \quad Height]$

  Der Eingabeparameter $Zielkoordinate$ ist die Koordinate, die vom UAS angeflogen werden soll.
- $Ausweichkoordinate\ (Form: 1x3)$: $[Longitude \quad Latitude \quad Height]$

  Der Eingabeparameter $Ausweichkoordinate$ beinhaltet die Koordinate des Ausweichpunktes, wenn eine Kollisionsgefahr vorliegt.
- $Data_{fremd}\ (Form: 2x3)$: $\begin{bmatrix} Longitude_1 & Latitude_1 & Height_1 \\ Longitude_2 & Latitude_2 & Height_2 \end{bmatrix}$

  Der Ausgabeparameter $Data_{fremd}$ beinhaltet die Geschwindigkeits- und Positionsinformationen über das kritischste fremde Objekt.

  1. Zeile: Fremder Geschwindigkeitsvektor
  2. Zeile: Fremde Position
- $POI_{Flugmatrix}\ (Form: 1x3)$: $[Longitude \quad Latitude \quad Height]$

  Der Eingabeparameter $POI_{Flugmatrix}$ beinhaltet die Zielkoordinate (POI), die zurzeit angeflogen wird.

Ausgabeparameter

- $Flugmatrix_{Vergleich}\ (Form: 29x3)$: $\begin{bmatrix} Longitude_1 & Latitude_1 & Height_1 \\ & \dots & \\ Longitude_{29} & Latitude_{29} & Height_{29} \end{bmatrix}$

  Der Ausgabeparameter $Flugmatrix_{Vergleich}$ ist die Flugmatrix, die in *Naechster-Wegpunkt* zu Vergleichszwecken dient.

5.4.2 Funktionsbeschreibung

Wenn die Matrix $Data_{fremd}$ nur *NaN*-Werte aufweist, soll die aktuelle Flugmatrix beibehalten werden, weshalb der Ausgabeparameter $Flugmatrix_{Vergleich}$ ebenfalls mit

NaN gefüllt wird, wodurch die Funktion *NaechsterWegpunkt* registriert, dass die aktuelle Flugmatrix beibehalten werden soll.

Andernfalls wird eine neue Flugmatrix berechnet. Bevor die Zeilen der Flugmatrix mit Werten gefüllt werden, wird ihre Größe vorab auf 29x3 definiert und alle Matrixelemente mit *NaN* belegt, damit spätere Algorithmen leere Zeilen anhand non *NaN*-Werten erkennen, wenn nicht alle Zeilen mit Koordinaten gefüllt sein sollten.

Zwischenziele zur Ausweichkoordinate (optional)

Der erste Teil der Flugbahnberechnung ist optional. Dieser wird nur aktiv, wenn eine Ausweichkoordinate vorhanden ist. Die maximal ersten fünf Koordinaten sollen mit Zwischenzielen zum Ausweichpunkt belegt sein und die restlichen mit denen zur Zielkoordinate (insgesamt maximal 25 Zwischenziele). Falls eine Ausweichkoordinate vorliegt, werden zuerst die Distanz und der Richtungsvektor zwischen der momentanen Position und der Koordinate des Ausweichpunktes über die folgenden Formeln bestimmt:

$$Abstand = \left|Ausweichkoordinate - Position_{eigen}\right| \qquad (5.8)$$

$$Richtung = Ausweichkoordinate - Position_{eigen} \qquad (5.9)$$

Die Anzahl der Zwischenpunkte zur Ausweichkoordinate wird anschließend in Abhängigkeit der Distanz mittels Formel (5.10) bestimmt.

$$Anzahl\ der\ Zwichenziele = \sqrt{Distanz} \qquad (5.10)$$

Dabei wird das Ergebnis aufgerundet und bei Überschreitung der Anzahl von fünf Zwischenzielen wird die Anzahl auf fünf begrenzt. Zur Berechnung der Zwischenziele wird eine *for*-Schleife mit dem Schleifenparameter i sooft durchlaufen, bis i gleich der Anzahl an Zwischenzielen ist. In jedem Schleifendurchlauf wird ein Zwischenziel über Formel (5.11) berechnet.

$$Zwischenziele(i,:) = Position_{eigen} + \frac{i \cdot Flugrichtung}{Anzahl\ der\ Zwischenziele} \qquad (5.11)$$

Zwischenziele zur Zielkoordinate

Anschließend wird die eben beschriebene Prozedur für die Zielkoordinate durchgeführt, wobei in Formel (5.11) nicht die $Position_{eigen}$ als Startkoordinate sondern die Koordinate des Ausweichpunktes gewählt wird, falls eine Ausweichkoordinate vorliegt. Da die Zwischenziele zum POI ebenfalls in die $Zwischenziele$-Matrix geschrieben werden, muss darauf geachtet werden, dass die neuen Koordinaten nicht die alten überschreiben. Deshalb wird erst ab einer bestimmten Zeile die $Zwischenziele$-Matrix beschrieben, die durch die Anzahl der Zwischenziele zur Ausweichkoordinate +1 definiert ist. Die maximale Anzahl an Zwischenzielen zum POI ist über Formel (5.12) definiert, da die $Zwischenziele$-Matrix maximal 25 Zwischenziele aufweisen soll.

$$Anzahl\ Zwischenziele\ zum\ POI = 25 - Anzahl\ Zwischenziele\ zur\ Ausweichkoordinate \quad (5.12)$$

Nachdem die Flugbahn vollständig berechnet wurde, wird der Ausgabeparameter $Flugmatrix_{Vergleich}$ mit den in Abbildung 5-1 zu sehenden Informationen gefüllt.

1. Zeile: Ausweichkoordinate
2. Zeile: Zielkoordinate
3. Zeile: Position des kritischsten Objekts
4. Zeile: Geschwindigkeitsvektor des kritischsten Objekts
5. – 29. Zeile: Zwischenziele (davon maximal 5 Zwischenziele zur Ausweichkoordinate, unbeschriebene Zeilen mit *NaN* gefüllt)

5.5 NaechsterWegpunkt

Die Funktion *NaechsterWegpunkt* ermittelt in Abhängigkeit der Eingabeparameter welche Koordinate als nächstes vom UAS angeflogen werden soll.

5.5.1 Ein- und Ausgabeparameter

Eingabeparameter

- $Flugmatrix_{alt}\ (Form: 29x3)$: $\begin{bmatrix} Longitude_1 & Latitude_1 & Height_1 \\ & \dots & \\ Longitude_{29} & Latitude_{29} & Height_{29} \end{bmatrix}$

  Der Eingabeparameter $Flugmatrix_{alt}$, wurde im vorherigen Durchlauf definiert und gespeichert. Er enthält die Informationen über die momentan verfolgte Flugbahn, die durch die in Abbildung 5-1 zu sehenden Koordinaten definiert ist.

- $Flugmatrix_{Vergleich}\ (Form: 29x3)$: $\begin{bmatrix} Longitude_1 & Latitude_1 & Height_1 \\ & \dots & \\ Longitude_{29} & Latitude_{29} & Height_{29} \end{bmatrix}$

  Der Eingabeparameter $Flugmatrix_{Vergleich}$ ist die Flugmatrix, die in *Flugbahnberechnung* erzeugt worden ist und zu vergleichszwecken dient.

- $Position_{eigen}\ (Form: 1x3)$: $[Longitude \quad Latitude \quad Height]$

  Der Eingabeparameter $Position_{eigen}$ beinhaltet die momentane GPS-Position des eigenen UAS.

- $Counter\ (Form: 1x1)$

  Der Eingabeparameter $Counter$ wurde im vorherigen Durchlauf generiert und gespeichert, damit der Funktion bekannt ist, welche Zeile der *Flugmatrix* momentan als *Sollkoordinate* aktiv ist.

Ausgabeparameter

- $Sollkoordinate\ (Form: 1x3)$: $[Longitude \quad Latitude \quad Height]$

  Der Ausgabeparameter $Sollkoordinate$ ist die Koordinate, die als nächsten von dem UAS auf dem Weg zur Zielkoordinate angeflogen werden soll.

- $Flugmatrix\ (Form: 29x3)$: $\begin{bmatrix} Longitude_1 & Latitude_1 & Height_1 \\ & \dots & \\ Longitude_{29} & Latitude_{29} & Height_{29} \end{bmatrix}$

  Der Ausgabeparameter $Flugmatrix$ beinhaltet alle relevanten Daten der momentanen Flugbahn, die gespeichert wird, sodass sie im nächsten Durchlauf zur Verfügung steht.

- $Counter$ $(Form: 1x1)$
  Der Ausgabeparameter $Counter$ beinhaltet die Nummer der aktuell als Sollkoordinate verwendeten Zeile der Flugmatrix.

5.5.2 Funktionsbeschreibung

Der allgemeine Ablauf dieser Funktion besteht aus mehreren *if-else*-Abfragen, über die der Inhalt der Ausgabeparameter gesteuert wird. In der Funktion *Flugbahnberechnung*, die zuvor den Eingabeparameter $Flugmatrix_{Vergleich}$ generiert hat, wurde überprüft, ob es sich im letzten Durchlauf um dasselbe kritischste Objekt handelt und ob die Zielkoordinate dieselbe geblieben ist. Wenn dem so ist, wurde der Parameter $Flugmatrix_{Vergleich}$ mit *NaN* gefüllt.

Aufgrund dessen besteht die erste Abfrage darin, ob eine mit Werten oder mit *NaN* gefüllte Matrix vorliegt.

1. $Flugmatrix_{Vergleich}$ mit *NaN* gefüllt:
   Da die Matrix mit *NaN* gefüllt ist, handelt es sich um dasselbe kritischste Objekt, sodass die aus dem vorherigen Durchlauf übergebene $Flugmatrix_{alt}$ beibehalten werden kann. Der Ausgabeparameter $Flugmatrix$ wird mit den Werten der $Flugmatrix_{alt}$ belegt. Außerdem wird über den $Counter$ ermittelt, welche Zeile der Flugmatrix momentan als $Sollkoordinate$ aktiv ist. Nachdem diese ermittelt ist, wird der Abstand zwischen der momentanen Position $Position_{eigen}$ und der $Sollkoordinate$ berechnet. Wenn der Abstand kleiner als 2m sein sollte, wird der an den Ausgang gegebene $Counter$ inkrementiert und die darauf folgende Zeile der Flugmatrix wird als $Sollkoordinate$ verwendet, sofern diese Zeile auch mit Werten gefüllt ist.
2. $Flugmatrix_{Vergleich}$ mit Werten gefüllt:

   In diesem Fall handelt es sich um ein anderes Objekt als im letzten Durchlauf.

   1. Wenn dieses Objekt keine Kollisionswarnung ausgelöst hat und somit keine Ausweichkoordinate berechnet wurde und die Zielkoordinaten der $Flugmatrix_{alt}$ und der $Flugmatrix_{Vergleich}$ identisch sind, wird der Ausgabeparameter $Flugmatrix$ mit $Flugmatrix_{alt}$ beschrieben. Anschließend

wird wieder der Abstand zwischen $Position_{eigen}$ und der momentanen $Sollkoordinate$ berechnet und bei Unterschreitung von 2m der $Counter$ inkrementiert und die $Sollkoordinate$ neu beschrieben.

2. Andernfalls wird der $Counter$ auf 5 zurückgesetzt (die erste Zeile in der eine Zwischenzielkoordinate beinhaltet ist), $Flugmatrix$ mit $Flugmatrix_{Vergleich}$ gefüllt und die $Sollkoordinate$ durch den $Counter$ ermittelt.

5.6 checkNaN

Die Funktion *checkNaN* überprüft, ob die übergebene Matrix aus *NaN*-Elementen besteht und gibt davon abhängig einen bool'schen Parameter zurück.

5.6.1 Ein- und Ausgabeparameter

Eingabeparameter

- $variable_{in}\ (Form: nxn): \begin{bmatrix} i_{11} & \cdots & i_{1n} \\ \vdots & \ddots & \vdots \\ i_{n1} & \cdots & i_{nn} \end{bmatrix}$

  Der Eingabeparameter $variable_{in}$ ist eine Matrix beliebiger Größe, welche auf den Inhalt NaN (= Not A Number) überprüft werden soll.

Ausgabeparameter

- $boolean\ (Form: Skalar): boolean$

  Der Ausgabeparameter $boolean$ enthält das Ergebnis der Auswertung. Ist diese Variable 1, so handelt es sich beim Input um eine Matrix, welche mit NaN gefüllt ist. Der Ausgabewert 0 signalisiert, dass mindestens eine Komponente der Matrix nicht mit NaN gefüllt ist.

5.6.2 Funktionsbeschreibung

Zu Beginn der Funktion wird aus dem Input die Größeninformation extrahiert, woraufhin eine Vergleichsmatrix gleicher Größe mit Einsen gefüllt wird. Anschließend wird auf den Input die Matlabfunktion *isnan()* ausgeführt, welche als Ausgabe eine Matrix gleicher Größe errechnet. Diese Funktion iteriert durch jede Komponente des Inputs

und setzt in der Ausgabematrix den Wert „1“, falls es sich bei der Komponente um „NaN“ handelt. Andernfalls wird die Komponente auf „0“ gesetzt.

Die Ausgabematrix wird nach Abschluss mit der Standardfunktion *compare()* mit der Vergleichsmatrix verglichen, wodurch der Ausgabeparameter $boolean$ definiert wird.

5.7 checkSize

Die Funktion *checkSize* überprüft den übetragenen Vektor, ob er in der für die Simulation benötigten Form vorliegt.

5.7.1 Ein- und Ausgabeparameter

Eingabeparameter

- $variable_{in}$ $(Form: 1x3\ oder\ 3x1)$: $[x_1 \quad x_2 \quad x_3]\ oder\ \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}$

  Der Eingabeparameter $variable_{in}$ ist ein beliebiger Vektor der Länge drei, welcher als Zeilen- oder Spaltenvektor vorhanden ist.

Ausgabeparameter

- $variable_{out}$ $(Form: 1x3)$: $[x_1 \quad x_2 \quad x_3]$

  Der Ausgabeparameter $variable_{out}$ enthält die Daten des Eingabeparameters $variable_{in}$, ist jedoch garantiert als Zeilenvektor dargestellt.

5.7.2 Funktionsbeschreibung

Auf die Inputvariable $variable_{in}$ wird die Matlabfunktion *size($variable_{in}$, 2)* angewandt, welche die Anzahl der Spalten des Vektors als Ergebnis liefert. Beträgt der Ausgabewert der Funktion den Wert drei, so liegt der Vektor als Zeilenvektor vor und muss daher nicht transponiert werden. Damit ist der Rückgabewert der Gesamtfunktion gleich der Eingangsvariable.

Im Falle eines anderen Wertes der Funktion *size()* wird der Input transponiert und der so umgeformte Vektor zurückgegeben.

6 Simulation

In diesem Kapitel soll die korrekte Funktionsweise der programmierten Algorithmen anhand unterschiedlicher Szenarien untersucht werden. Dazu wird zuerst aufgezeigt in welcher Simulationsumgebung die *Flight Intelligence* eingebettet ist, sodass ein Überblick über die Vielzahl an Variationsmöglichkeiten der Simulationsparameter gegeben wird. Diese werden anschließend näher beschrieben. Zuletzt werden unterschiedliche Szenarien simuliert und ausgewertet, sodass eine Aussage über die Korrektheit der implementierten Funktion getroffen werden kann.

6.1 Simulationsumgebung und -parameter

In diesem Abschnitt werden die Simulationsumgebung der Flight Intelligence und die veränderbaren Simulationsparameter betrachtet. In Abbildung 6-1 ist die Simulationsumgebung in Form der Systemdarstellung in Simulink zu sehen.

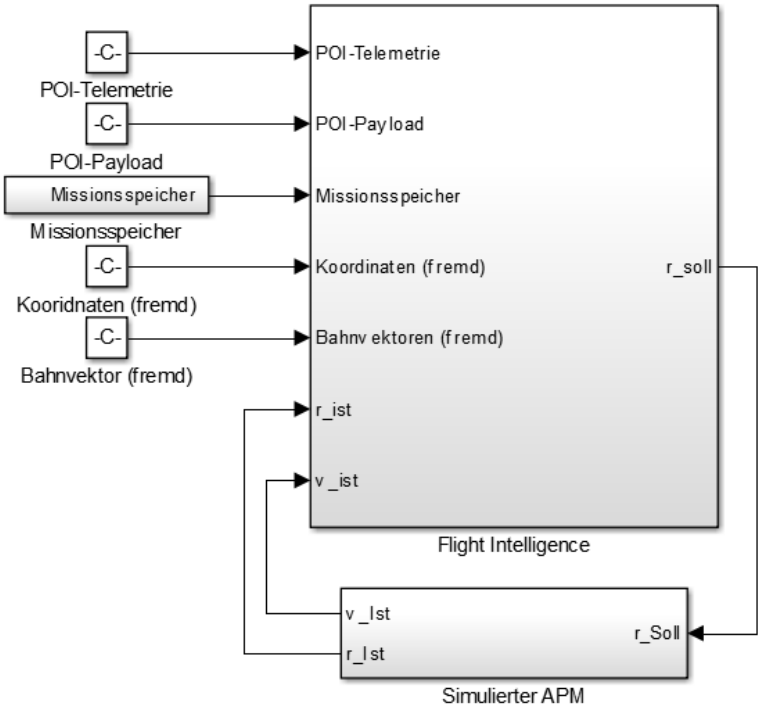

Abbildung 6-1: Systemdarstellung in Simulink

Fünf der acht Eingänge müssen vom Anwender definiert werden, damit die Simulation durchlaufen werden kann. Zu diesen zählen der POI, der zum einen von der Telemet-

rieschnittstelle kommandiert worden ist und zum anderen der von der Payload vorgegeben wird. Des Weiteren besteht ein Eingang aus dem Missionsspeicher, in welchem die erste Zeile mit der „Home“-Koordinate belegt ist und in weiteren Zeilen die anzufliegenden POIs hinterlegt sind. Die letzten beiden Eingänge, die vom Benutzer gesteuert werden können, sind die Koordinate eines statischen Objekts, dessen Geschwindigkeitsvektor in diesem Fall auf [0 0 0] gesetzt werden muss. Bei dynamischen Hindernissen, sprich einem weiteren UAS, werden die Eingänge anders verschaltet. Hierbei sei jedoch auf Abschnitt 6.3.4 verwiesen. Wenn einer der zu sehenden Blöcke keinen Einfluss auf die Simulation haben soll, muss der jeweilige Block mit einem 1x3 *NaN*-Vektor gefüllt werden. Wie bereits beschrieben, erkennen die implementierten Funktionen nicht zu verwendende Eingänge anhand dieses *NaN*-Vektors.

Des Weiteren weist der Flight Intelligence Block die Eingänge *r_Ist* (momentane Position), *v_Ist* (momentaner Geschwindigkeitsvektor) auf, die vom APM-Modell generiert werden, welches im folgenden Abschnitt beschrieben wird. Der einzige Ausgabeparameter, den die Flight Intelligence liefert, *r_soll*, ist zugleich der einzige Eingangsparameter, den das APM-Modell zur Berechnung benötigt.

6.2 APM-Modell

Das APM-Modell bildet den physikalischen Grundbaustein der Simulation. Dieser Block wird im Flugversuch auf dem Payload Control Computer nicht mitfliegen, da er nur zum Schließen der Simulationsschleife ohne den echten Flugregelungscomputer (APM) benötigt wird. In diesem Subsystem wird die Reaktion des Multikopters auf Eingabewerte (Fordern einer neuen Zielkoordinate) auf einem niedrigen Level simuliert. Hierbei werden physikalische Begebenheiten wie beispielsweise Schwerkraft, Luftwiderstand oder Momente durch Rotoren gänzlich vernachlässigt. Stattdessen liegt das Hauptaugenmerk des Systems im geregelten Erreichen des vorgegebenen Zielpunktes unter Berücksichtigung maximaler Geschwindigkeiten und Beschleunigungen. Als grundsätzliche Annahmewerte dienen hierzu eine maximale Geschwindigkeit von ±10 m/s als auch eine definierte Maximalbeschleunigung von ±3 m/s², kombiniert in allen drei Achsrichtungen.

Wie in Abbildung 6-2 zu sehen ist, wird zuerst die momentane Position r_{ist} von r_{soll} abgezogen, um eine Regelabweichung zu erhalten. Aus dieser Regelabweichung und mit Hilfe deren Betrages kann ein Einheitsvektor in Richtung der Zielkoordinate erstellt werden, wobei hier auf eine Division durch Null überprüft werden muss.

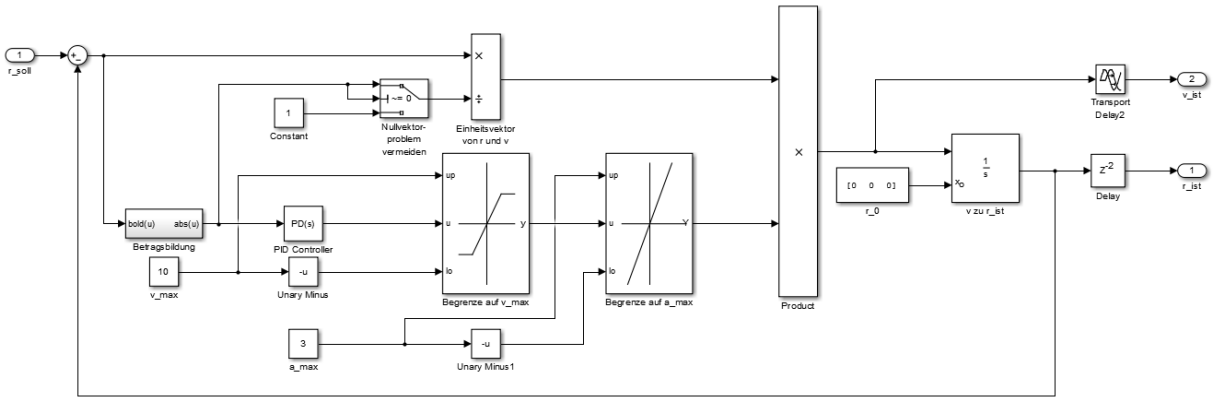

Abbildung 6-2: APM-Modell

Ein PD-Regler mit den Werten P = 0,5 und D = 0,6 überführt die in der Simulation geforderte Position auf den Ausgang, wobei die Geschwindigkeit durch einen Saturation-Block sowie auch die Beschleunigung durch einen Rate-Limiter-Block begrenzt werden.

Anschließend wird der so erhaltene Wert mit dem Einheitsvektor multipliziert, wodurch sich die momentane Geschwindigkeit am Ausgang v_{ist} ergibt. Durch einmalige Integration der Geschwindigkeit steht die neue Position am Ausgang r_{ist} des Subsystems.

6.3 Simulationsdurchläufe

Dieser Abschnitt dient zur Darstellung der möglichen Hindernisszenarien, die ein UAS bewältigen muss. Die Komplexität der Szenarien steigert sich von Schritt zu Schritt. Dazu wird zuerst eine Flugstrecke mit mehreren Wegpunkten simuliert. Anschließend werden erst ein und danach mehrere statische Hindernisse in die Flugbahn des UAS gesetzt, sodass die Algorithmen des Kollisionsvermeidung getestet werden. Zuletzt sollen diese Algorithmen ebenfalls bei dynamischen Hindernissen verifiziert werden, sodass bei einer möglichen Mission zwei verschiedene UAS nicht miteinander kollidieren.

6.3.1 Wegpunktflug

An dieser Stelle soll die Umsetzung des Konzepts der Flugmatrix dargestellt werden, indem das UAS zwei verschiedene Ziele anfliegt. In der Flugmatrix sind abhängig von der Distanz zu einem Zielpunkt, eine unterschiedliche Anzahl von Zwischenpunkten zum Ziel hinterlegt. Dadurch wird der geradlinige Anflug des Ziels nicht durch Wind beeinflusst, da das UAS jedes Zwischenziel direkt anfliegt.

Um dieses Szenario abbilden zu können, wurden die Eingangsblöcke von *POI-Payload, Koordinaten fremd, Bahnvektoren fremd* und *POI-Telemetrie* mit *NaN*-Werten gefüllt, sodass diese keinen Einfluss auf die Simulation haben. Des Weiteren wurde der Missionsspeicher mit der „Home“-Koordinate [0 0 0] und den Wegpunkten [0 50 100] und [100 0 0] gefüllt und der Startwert im simulierten APM auf [0 0 0] festgelegt.

Nachdem die Simulation gestartet wurde, resultieren für die Sollkoordinate und die momentane Position die in Abbildung 6-3 dargestellten Kurvenverläufe. Auf der x-Achse ist die Zeit in Sekunden und auf der y-Achse die Distanz in Metern abgetragen. Jeweils die blaue Kurve beschreibt die Koordinate in x, die rote in y und die grüne in z-Richtung.

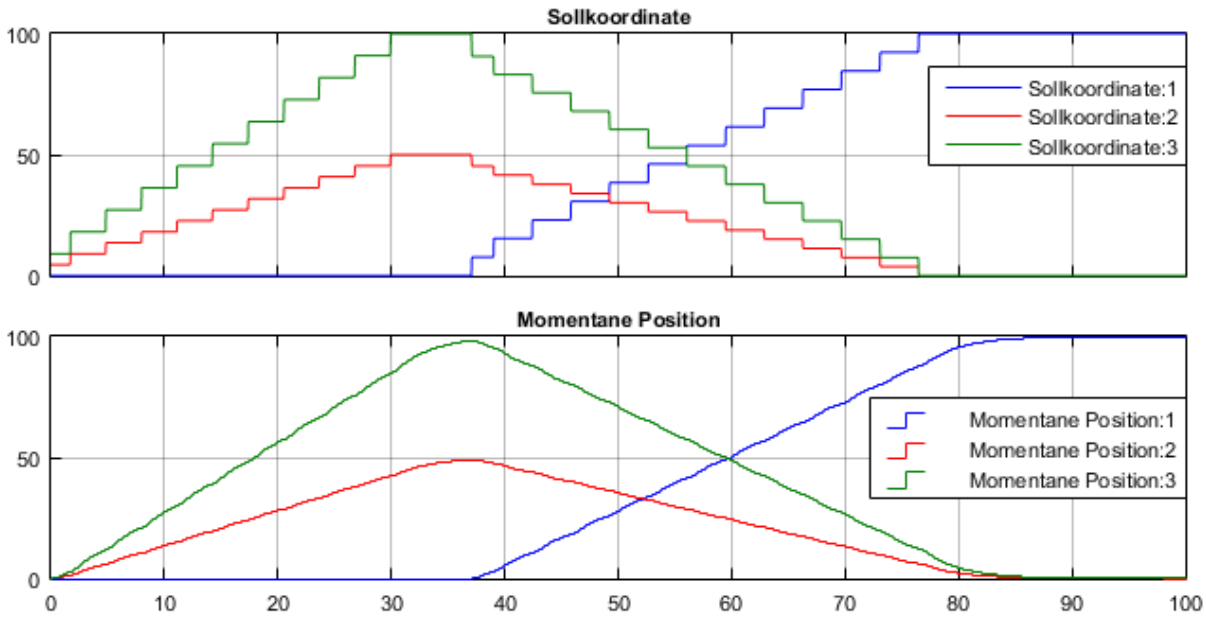

Abbildung 6-3: Darstellung der Sollkoordinate und momentanen Position bei Wegpunktflug

Anhand dieser Abbildung wird das Anfliegen jedes Zwischenziels zum Zielpunkt, die in der Flugmatrix abgelegt sind, sehr gut deutlich, da die Sollkoordinate einen stufenweisen Verlauf aufweist. Dies ist darin begründet, dass ein Zwischenziel so lange aktiv

ist, bis die Differenz zwischen einem Zwischenziel und der momentanen Position kleiner als 2m ist. Anschließend wird die Flugmatrixzeile inkrementiert, sodass eine neue Sollkoordinate erzeugt wird. Des Weiteren soll nach Unterschreitung der Distanz von 2m zwischen dem aktuellen POI und momentaner Position die POI-Zeile aus dem Missionsspeicher inkrementiert werden. Dadurch ist der minimale Abstand im Diagramm der momentanen Position zwischen dem Maximum der grünen Kurve und der 100m Linie begründet.

6.3.2 Ausweichen eines statischen Hindernisses

Der zweite Simulationsdurchlauf beinhaltet den Test inwieweit das UAS einem statischen Hindernis ausweichen kann, das auf der Flugbahn zur Zielkoordinate liegt. Dazu wurden in den Missionsspeicher die POIs [0 100 0] und als zweites die „Home“-Koordinate [0 0 0] geschrieben. In dem Block *Koordinate fremd* wurde die Position des Hindernisses mit [0 50 0] festgelegt. Da es sich hier um ein statisches Hindernis handelt, besteht der Geschwindigkeitsvektor aus [0 0 0].

Die resultierenden Kurvenverläufe der Sollkoordinate und der momentanen Position sind in Abbildung 6-4 dargestellt. Dabei ist auf der x-Achse wieder die Zeit in Sekunden und auf der y-Achse die Distanz in Metern abgetragen. Wie zuvor beschreibt die blaue Kurve die Koordinate in x, die rote in y und die grüne in z-Richtung.

Es ist deutlich zu sehen, dass das UAS auf dem Weg zur Zielkoordinate nicht den direkten Weg nimmt, sondern dass aufgrund des Hindernisses ein Ausweichmanöver durchgeführt wird. Das Hindernis wird nach 10s Flugzeit erkannt und das Ausweichmanöver eingeleitet, da an dieser Stelle der Verlauf der Sollkoordinate eine Veränderung aufweist. Das UAS weicht dabei nach rechts aus, wie es in Abschnitt 5.3 definiert wurde. Nachdem das Hindernis überwunden wurde, fliegt das RPAS wieder auf direktem Weg zum Zielpunkt. Nachdem es diesen erreicht hat, quert es zur „Home“-Koordinate zurück. Dabei muss wiederum ein Ausweichmanöver vollzogen werden. Da das UAS wiederum rechts ausweicht, ändert sich das Vorzeichen der Ausweichkoordinate in x-Richtung im Vergleich zum Hinflug. Wie zuvor bewegt sich das UAS nach Überwindung des Hindernisses auf direktem Weg zur Zielkoordinate.

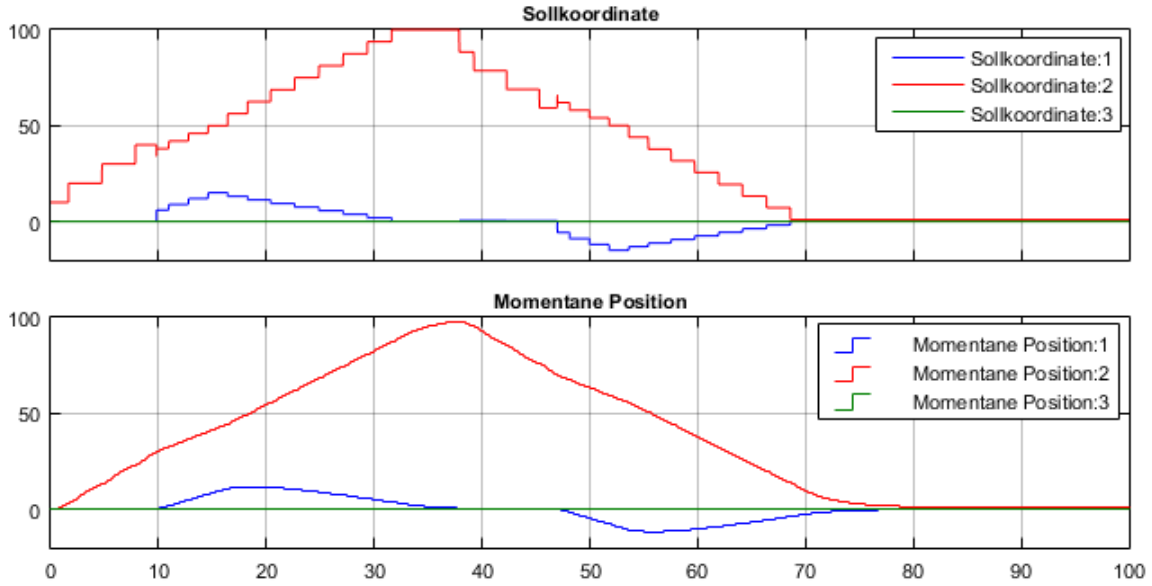

Abbildung 6-4: Darstellung der Sollkoordinate und momentanen Position beim Ausweichen eines statischen Hindernisses

6.3.3 Ausweichen von mehreren statischen Hindernissen

Um das vorherige Szenario komplexer zu gestalten, werden in diesem mehrere Hindernisse in die Flugbahn des UAS gesetzt. Die Flugbahn verläuft wie zuvor auch zum Punkt [0 100 0] und wieder zurück zur „Home“-Koordinate [0 0 0]. Die statischen Hindernisse befinden sich an den Punkten [0 20 0] (Hindernis 1), [10 50 0] (Hindernis 2) und [15 80 0] (Hindernis 3). Da es sich hier ausschließlich um statische Hindernisse handelt, werden die Geschwindigkeitsvektoren mit [0 0 0] belegt.

Die resultierenden Kurvenverläufe der Sollkoordinate und der momentanen Position sind in Abbildung 6-5 dargestellt. Dabei ist wiederum auf der x-Achse die Zeit in Sekunden und auf der y-Achse die Distanz in Metern abgetragen. Wie zuvor auch, beschreibt die blaue Kurve die Koordinate in x, die rote in y und die grüne in z-Richtung.

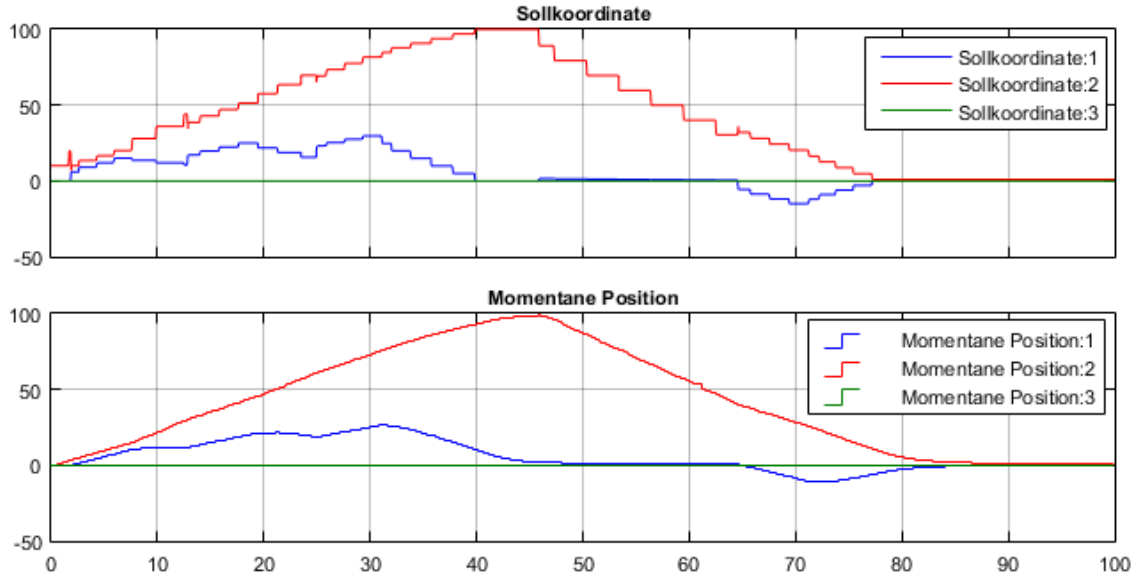

Abbildung 6-5: Darstellung der Sollkoordinate und momentanen Position beim Ausweichen mehrerer statischer Hindernisse

Aus der Abbildung wird deutlich, dass das UAS, das für den Moment kritischste Objekt erkennt und aufgrund dessen ein Ausweichmanöver einleitet. Dies lässt sich daran erkennen, dass das UAS auf dem Hinflug den Hindernissen 1, 2 und 3 nacheinander ausweicht. Auf dem Rückflug wird lediglich Hindernis 1 ausgewichen, da dieses das einzige Objekt ist, das eine Kollisionswarnung ausgelöst hat. Die anderen Hindernisse sind nur beim Hinflug relevant, da sich das UAS aufgrund des Ausweichmanövers von Hindernis 1 in Richtung der positiven x-Achse bewegt. Deshalb führen auch die Hindernisse 2 und 3 zu Kollisionswarnungen, da diese eine positive Verschiebung von der y-Achse in x-Richtung aufweisen und somit von der Kollisionslogik der Flight Intelligence erfasst werden. Die Zeitpunkte an denen das kritischste Objekt neu bestimmt wird und woraufhin eine neue Flugbahn berechnet wird, lässt sich anhand der Diskontinuitäten in dem stufenförmiger Verlauf der Sollkoordinate erkennen.

6.3.4 Co-Simulation: Ausweichen dynamischer Hindernissen

Der letzte Schritt, der zur Verifikation der Funktionsweise der Flight Intelligence dient, ist das Ausweichen eines dynamischen, sich bewegenden Hindernisses. Dazu wurde das in Abbildung 6-1 zu sehende Simulink-Blockbild in ein System integriert.

Dieses System weist die Eingänge *Bahnvektor fremd* und *Koordinate fremd* und die Ausgänge *r_ist* und *v_ist* auf. Anschließend wurde dieses System dupliziert und jeweils die Ausgänge des einen mit den Eingängen des anderen Systems zusammengeschlossen, sodass das Simulink-Blockbild, abgebildet in Abbildung 6-6, entsteht. Jedes System soll ein eigenständiges UAS darstellen.

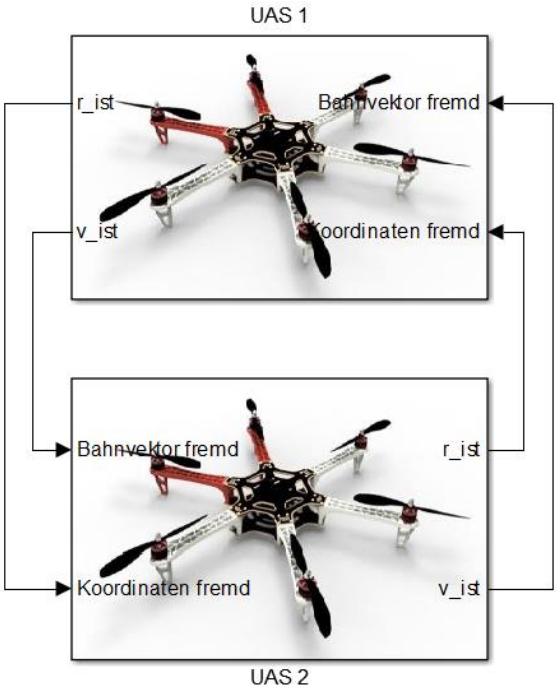

Abbildung 6-6: Umsetzung der Co-Simulation in Simulink

Anschließend werden die Wegpunkte und Startkoordinaten definiert, die in Tabelle 1 aufgelistet sind. Die Startkoordinate wird wie bereits erwähnt im Block des APM-Modells und die Wegpunkte im Missionsspeicher definiert.

Tabelle 1: Startkoordinaten und Wegpunkte der jeweiligen RPAS

Art	*RPAS 1*	*RPAS 2*
Startkoordinate	[0 0 0]	[0 100 0]
Wegpunkt 1	[0 100 0]	[0 0 0]
Wegpunkt 2	[0 0 0]	[0 100 0]

Diese Konstellation ermöglicht ein Szenario, indem sich die UAS auf den Hin- und Rückweg zu ihrem ersten Wegpunkt und zurück zur Startkoordinate (Wegpunkt 2) begegnen, sodass Ausweichmanöver eingeleitet werden müssen.

Die resultierenden Kurvenverläufe der momentanen Positionen der UAS sind in Abbildung 6-7 dargestellt. Dabei ist wiederum auf der x-Achse die Zeit in Sekunden und auf

der y-Achse die Distanz in Metern abgetragen. Wie zuvor auch, beschreibt die blaue Kurve die Koordinate in x, die rote in y und die grüne in z-Richtung.

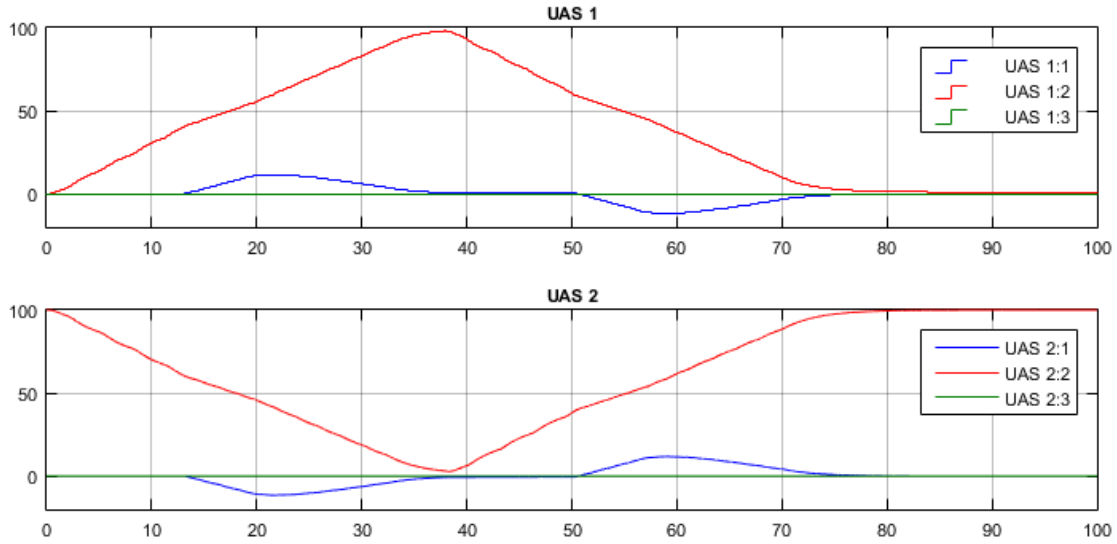

Abbildung 6-7: Darstellung der momentanen Positionen zweier UAS beim Ausweichen eines dynamischen Hindernisses

Aus den Diagrammen wird deutlich, dass beim Anflug der jeweiligen Wegpunkte beide UAS Ausweichmanöver ausgeführt haben, um eine Kollision zu vermeiden. Dabei weichen die UAS in Flugrichtung immer nach rechts aus, wie es in der Funktion *Berechne_Ausweichkoordinate* implementiert worden ist. Des Weiteren weisen sie bei den Ausweichmanövern einen großen Abstand von mehr als 15 m zueinander auf. Dadurch wird eine hohe Sicherheitsmargin erzeugt, sodass unterschiedliche Störungsszenarien auf die Flugbahn nicht zu einer Gefährdung der UAS führen.

6.4 Portierung auf GPS-Koordinaten

Da die Simulation bis zu diesem Punkt nur mit einem kartesischen Simulationskoordinatensystem gearbeitet hat, dessen Einheit auf den drei Achsen Metern ist, muss sie zur Anwendung auf den Multicoptern auf GPS-Koordinaten portiert werden. Damit nicht alle mathematischen Funktionen der Simulation umgeschrieben werden müssen, wird lediglich eine Transformation der Koordinatensysteme an den Ein- und Ausgänge vorgenommen. Bei den Eingängen geschieht diese Transformation für die momentane Position, die fremden Koordinaten und den Missionsspeicher. Bei den Ausgängen ist lediglich die Sollkoordinate betroffen. Zur Umwandlung einer GPS-Koordinate in das

Simulationskoordinatensystem, wurde die Funktion *GPS2Sim* implementiert. Dabei ist vorwegzunehmen, dass die GPS-Koordinate, an der der Multicopter startet, als Startkoordinate festgelegt wird. Dieser Punkt ist im Simulationskoordinatensystem als [0 0 0] definiert. Zur Bestimmung des Abstandes in lateraler und longitudinaler Richtung (entspricht der x und y-Koordinate) werden die folgenden Formel verwendet.

$$Distanz_{lat} = |GPS_{lat} - Start_{lat}| \quad (6.1)$$

$$Distanz_{lon} = |GPS_{lon} - Start_{lon}| \quad (6.2)$$

$$Distanz_{Höhe} = GPS_{Höhe} - Start_{Höhe} \quad (6.3)$$

Es ist anzumerken, dass die Betragsbildung in einer gesonderten Funktion stattfindet, da diese die Position auf der Erde berücksichtigt. Diese Information wird zur Streckenbestimmung benötigt, da der Abstand zwischen zwei Längengeraden vom Breitengrad abhängig und somit auf der ganzen Welt unterschiedlich ist. Anschließend wird das Vorzeichen dieser Distanzen über einen Vergleich der GPS- und Startgröße festgelegt.

Zur Umrechnung der Simulationskoordinaten am Ausgang in GPS-Koordinaten wurde die Funktion *Sim2GPS* implementiert, die die folgenden Formeln beinhaltet.

$$Latitude = Start_{lat} + \frac{180}{\pi} \cdot \frac{Distanz_{lat}}{Erdradius} \quad (6.4)$$

$$Longitude = Start_{lon} + \frac{180}{\pi} \cdot \frac{Distanz_{lon}}{Erdradius \cdot cos\left(Start_{lat} \cdot \frac{\pi}{180}\right)} \quad (6.5)$$

$$Höhe = Start_{Höhe} + Distanz_{Höhe} \quad (6.6)$$

Aufgrund der eben beschriebenen Funktionen *GPS2Sim* und *Sim2GPS* ist es möglich die Simulation auf GPS-Koordinaten zu portieren, sodass der Funktionsfähigkeit auf dem Multicopter gewährleistet ist.

Folgend soll nun die GPS-Portierung verifiziert werden. Mit Hilfe von Google Maps werden GPS-Koordinaten von markanten Punkten des Sportplatzes in Friedrichshafen-Ost (vgl. Abbildung 6-8) extrahiert und als Wegpunkte in die Simulation eingeführt. Außerdem wird auf den Koordinaten des Mittelpunktes des Sportplatzes ein statisches Hindernis platziert.

Abbildung 6-8: Wegpunkte für die Verifizierung der GPS-Portierung

In Tabelle 2 sind nochmals die Werte der GPS-Koordinaten ersichtlich:

Tabelle 2: GPS-Koordinaten der Wegpunkte und Hindernisse

	Latitude [°]	*Longitude [°]*	*Altitude [m]*
Startpunkt	47.653503	9.498422	0
WP1	47.654363	9.498996	0
WP2	47.654114	9.499823	0
WP3	47.653252	9.499250	0
WP4	47.653379	9.498835	0
Zielpunkt	47.654239	9.499408	0
Hindernis	47.653808	9.499122	0

Führt man diese Simulation mit den definierten Koordinaten nun aus, so ergibt sich folgendes Bild (vgl. Abbildung 6-9):

Abbildung 6-9: Ergebnis der GPS-Portierung

Wie sehr gut erkenntlich ist, fliegt das UAS in der Simulation die Wegpunkte an, bis die minimale Distanz zwischen Position und Wegpunkt erreicht ist. Anschließend verfolgt das UAS den nächsten Wegpunkt, bis es schließlich an der Zielkoordinate angelangt ist. Auch das Ausweichen um das Hindernis verläuft, wie bereits im Simulationskoordinatensystem dargestellt, ohne Probleme. Somit ist die Portierung der Simulation auf GPS-Koordinaten verifiziert und kann verwendet werden.

6.5 Diskussion der Ergebnisse

In den letzten beiden Abschnitten dieses Kapitels wurden verschiedene Testszenarien simuliert, deren Komplexität von Schritt zu Schritt erhöht wurde. Dabei wurden zuerst lediglich verschiedene Wegpunkte über den Missionsspeicher kommandiert, um die Berechnung der Flugbahn zum Zielpunkt, die durch mehrere Zwischenziele definiert wird, zu verifizieren. Die korrekte Funktionsweise wurde durch den stufenweisen Verlauf der Sollkoordinate bestätigt, da nach jedem erreichten Zwischenziel das nächste Zwischenziel kommandiert wurde, sodass der eben angesprochene Verlauf resultiert. Anschließend wurden erst ein statisches Hindernis und danach mehrere statische Hindernisse in die Flugbahn des UAS gesetzt, wodurch die Algorithmen zur Kollisionsfrüherkennung und Kollisionsvermeidung überprüft wurden. Die Ergebnisse der Simulationen zeigten, dass das UAS einem Hindernis immer nach rechts ausweicht, so wie

es im Kollisionskonzept geplant war. Da durch diese Simulationen die einwandfreie Funktionalität der Flight Intelligence nachgewiesen werden konnte, wurde ein Szenario erstellt, dass die größt mögliche Komplexität aufweist. Dieses Szenario beinhaltet das simultane Ausweichen zweier dynamischer Hindernisse, sprich das gegenseitige Ausweichen von zwei UAS. Auch diese Simulation lieferte positive Ergebnisse, da die beiden UAS nicht nur ein Ausweichmanöver, sondern dies auch mit einem großen Abstand zueinander durchgeführt haben. Aufgrund dessen können unterschiedliche Beeinflussungen der Flugbahn nicht zur Gefährdung der UAS führen, da die Ausweichdistanz große Sicherheitsreserven beinhaltet.

Wie bereits beschrieben wurde, sind die vorherigen Simulationen in einem kartesischen Simulationskoordinatensystem durchgeführt worden. Um die Software jedoch auf einem realen Hexacopter laufen zu lassen, muss die Software auf GPS-Koordinaten portiert werden. Um eine komplette Abänderung der Funktionen zu verhindern, werden lediglich die Ein- und Ausgangsparameter zwischen den GPS und Simulationskoordinaten transformiert. Zur Verifizierung dieser Transformationen wurde ein Szenario erstellt, in dem reale GPS-Koordinaten im Missionsspeicher hinterlegt waren. Es zeigte sich, dass auch die GPS-Portierung positive Ergebnisse lieferte, da durch das Abtragen der simulierten Flugspur auf eine Karte zu sehen ist, dass die kommandierten Wegpunkte, definiert durch GPS-Koordinaten, angeflogen worden sind.

Zusammenfassend wurde die Flight Intelligence Software durch eine Vielzahl von unterschiedlich komplexen Simulationen auf ihre korrekte Funktionsweise geprüft. Da diese Simulationen allesamt positive Ergebnisse lieferten und auch die Portierung auf GPS-Koordinaten ermöglicht wurde, kann die Software nun am realen Hexacopter anhand eines Flugtests verifiziert werden.

7 Bilderkennung und -verarbeitung

Ein weiterer Bestandteil des Projektes Locator war das Vorhaben, farbige Objekte verfolgen zu können. Diese sollten per Kamera detektiert werden. Aus den Kamerabildern und der aktuellen GPS-Position der Drohne können die Position des Balles und damit eine Flugroute bestimmt werden.

Im Folgenden wird zunächst auf die Grundlagen der Bildverarbeitung eingegangen. Anschließend wird geschildert wie Bilderkennung und -verarbeitung für den Multicopter mit Matlab/Simulink realisiert wurden. Daraufhin wird auf die Berechnung der Koordinate des Balles aus dem Bildmaterial eingegangen.

7.1 Grundlagen der Bildverarbeitung

Unter Bildverarbeitung versteht man das Bearbeiten eines Bildes in einer Art und Weise, die es erlaubt das Bild zu verändern oder Informationen aus ihm zu gewinnen [8]. Es handelt sich dabei um eine Art der Signalverarbeitung. Das Eingangssignal stellt ein unbearbeitetes Bild und daraus gewonnene Informationen oder ein bearbeitetes Bild das Ausgangssignal dar. Ein Bildverarbeitungssystem behandelt ein Bild typischerweise als zweidimensionales Signal und wendet vordefinierte Algorithmen und Methoden darauf an. Bildverarbeitung ist ein schnellwachsender Zweig der Signalverarbeitung und gewinnt immer mehr Bedeutung für Wissenschaft und Wirtschaft.

Die meisten Bildverarbeitungsalgorithmen bestehen aus drei Schritten:

1. Importieren des Bildes ins Verarbeitungssystem durch optische Scanner (für analoge, in Hardcopy-Form vorliegende, Bilder) oder digitale Kameras, die direkt ans System angeschlossen sind.
2. Analysieren und Manipulieren des Bildes, d.h. Anwenden von Algorithmen zur Kompression, Verbesserung, Mustererkennung oder Ähnliches.
3. Ausgabe des manipulierten Bildes oder von Informationen, die daraus generiert wurden.

Die für das Bildverarbeitungssystem der Locator-Drohnen relevanten Methoden und Grundlagen werden im Folgenden erklärt. Dabei wird insbesondere auf das RGB-Modell und die Hough-Kreiserkennung eingegangen, da die zwei Methode grundlegend zur Objekterkennung sind.

7.1.1 Das RGB-Farbmodell

Bei dem RGB-Modell handelt sich um ein additives Farbmodell, in dem Farben als Zahlentupel dargestellt werden. RGB steht für die drei verwendeten Primärfarben rot, grün und blau. Die Hauptanwendung des RGB-Modells liegt in der Bilddarstellung in elektronischen Systemen wie Computern oder Fernsehgeräten. Bei dem Modell handelt es sich um ein Gerät-abhängiges Modell. Das heißt, dass unterschiedliche Systeme RGB-Werte nicht exakt gleich erkennen und darstellen, was auf beispielsweise unterschiedliche Verwendung von Leuchtstoffen zurückgeführt werden kann. Das führt dazu, dass ein RGB-Wert nicht plattformübergreifend als Definition einer Farbe verwendet werden kann.

Das Modell ist aus der Farbwahrnehmung von trichromatischen Organismen (z.B. das menschliche Auge) abgeleitet. Denn Farben eignen sich dann besonders gut, wenn sie möglichst unterschiedliche Reaktionen in den Fotorezeptoren - genauer gesagt den Zapfen - auf der Retina des menschlichen Auges auslösen [9]. Da der Mensch Rezeptoren für eben jene Farben rot, grün und blau verfügt, bietet sich die Wahl dieser Farben an. Die drei Primärfarben werden mit unterschiedlichen Intensitäten überlagert, sodass eine neue Farbe entsteht. Die Überlagerung aller drei Farben mit gleicher Intensität ergibt beispielsweise weiß. Diese Art der Farberzeugung wird additiv genannt. Sie steht im Kontrast zur subtraktiven Farberzeugung, bei der Farben von einem sogenannten Key abgezogen werden. Als Beispiel für ein solches Modell ist das sogenannte CMYK-Modell, das unter anderem in Druckern zum Einsatz kommt [10].

Die Darstellung der einzelnen Primärfarben erfolgt in der Regel mit acht Bit (bezeichnet als Farbtiefe), sodass sich pro Komponente 255 verschiedene Töne darstellen lassen. Bei drei Komponenten ergibt sich daraus eine Vielfalt von 16.777.216 darstellbaren Farben. Da die Unterschiede zwischen den einzelnen Farbtönen kleiner als die vom

menschlichen Auge wahrnehmbaren Unterschiede sind, spricht man von „Wahrhaftigen Farben“.

Die folgende Grafik [11] gibt einen Überblick über die mit RGB darstellbaren Farben. Auf der Diagonalen zwischen schwarz und weiß befinden sich alle darstellbaren Graustufen.

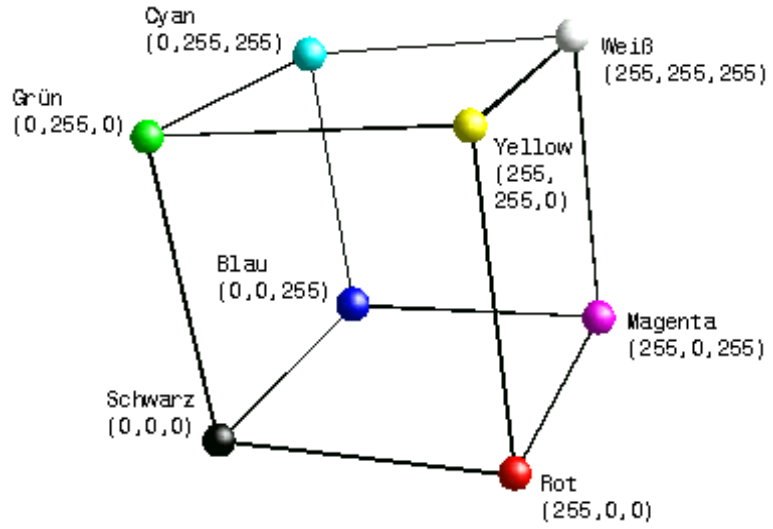

Abbildung 7-1: Der RGB-Farbwürfel

7.1.2 Die Kreis – Hough - Transformation

Die Kreis-Hough-Transformation (KHT) ist eine Abwandlung der allgemeinen Hough-Transformation. Diese wurde 1972 von Richard Duda und Peter Hart auf der Grundlage eines Patentes von Paul Hough aus dem Jahre 1962 erfunden [12]. Bei der allgemeinen Hough Transformation handelt es sich um eine Methode Geraden in Bildern aufzuspüren. Wie der Name nahe legt dient die KHT zum Detektieren von Kreisen. Im Folgenden wird die Funktionsweise der KHT beschrieben.

Die KHT benötigt ein binäres Bild, also ein Bild, das nur aus weißen und schwarzen Pixeln besteht. Da eine Mustererkennung (Kreis) durchgeführt werden soll, ist es sinnvoll, wenn dieses binäre Bild die Kanten aller Objekte im Bild enthielte. Dazu kann auf das Ausgangsbild ein Kantendetektionsalgorithmus angewendet werden, zum Beispiel die von John F. Canny entwickelte *Canny Edge Detection* [13].

Ein Kreis lässt sich über folgende Gleichung beschreiben:

$$(x-a)^2 + (y-b)^2 = R^2 \qquad (7.1)$$

Es werden also offensichtlich drei Parameter benötigt um einen Kreis zu beschreiben: Sein Mittelpunkt (a,b) und sein Radius R. Da ein dreidimensionaler Parameterraum viel Arbeitsspeicher und Rechenzeit beanspruchen würde, wird durch die Festlegung des Radius auf zwei Dimensionen beschränkt. Jeder Bildpunkt wird nun in den sogenannten Hough-Raum transformiert: Dies geschieht schlicht dadurch, dass jeder Punkt, der auf einer Kante liegt, also im binären Bild einem Pixel entspricht, das eine andere Farbe als der Hintergrund hat, zum Mittelpunkt eines Kreises mit dem Radius R wird. Diese Kreise werden in die sogenannte Akkumulatormatrix geladen. Punkte, die zu besonders vielen Kreisen gehören, sind mit erhöhter Wahrscheinlichkeit der Mittelpunkt eines Kreises. Die folgenden Graphiken sollen die Logik verdeutlichen: In Abbildung 7-2 [16] sind zwei sich überlappende Kreise zu sehen. Dass es sich schon um eine binäre Abbildung handelt, ist Zufall.

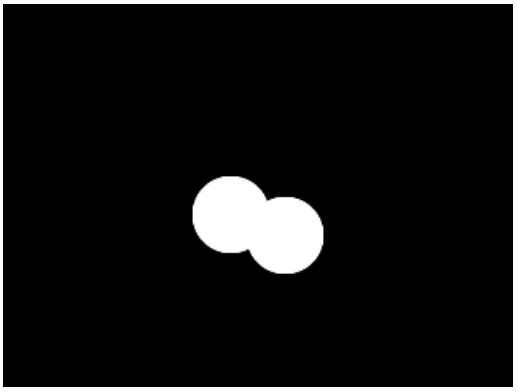

Abbildung 7-2: Zwei Kreise für die KHT

Mittels der *Canny Edge Detection* werden aus Grafik nun die Umrisse der Kreise ermittelt (siehe Abbildung 7-3 [16]).

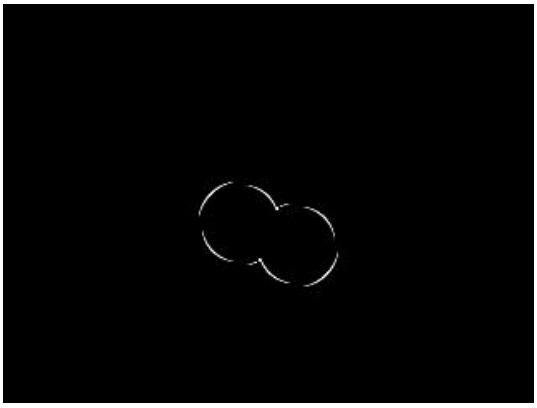

Abbildung 7-3: Anwendung der CED

Nun sind von den ausgefüllten Kreisen nur noch die Umrisse zu erkennen. Jeder weiße Pixel wird in den Hough-Bereich transformiert, also zu dem Mittelpunkt eines Kreises mit dem Radius R gemacht. Dies ist in Abbildung 7-4 [16] veranschaulicht

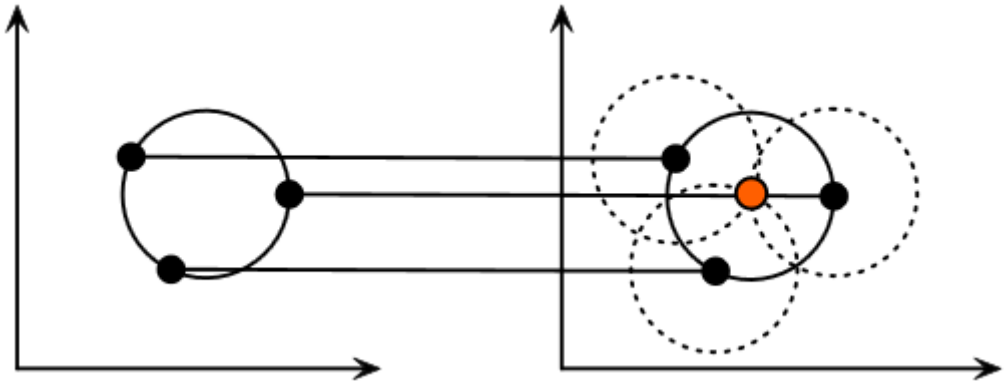

Abbildung 7-4: Anwendung der Hough-Transformation auf einzelnen Kreis

Jeder im Hough-Bereich erzeugte Kreis schneidet den Mittelpunkt des Ursprungskreises. Werden alle „belegten“ Punkte des Hough-Bereichs in einer Grafik dargestellt, ergibt sich das in Abbildung 7-5 [16] zu sehende Bild (bezogen auf das vorangegangene Beispiel):

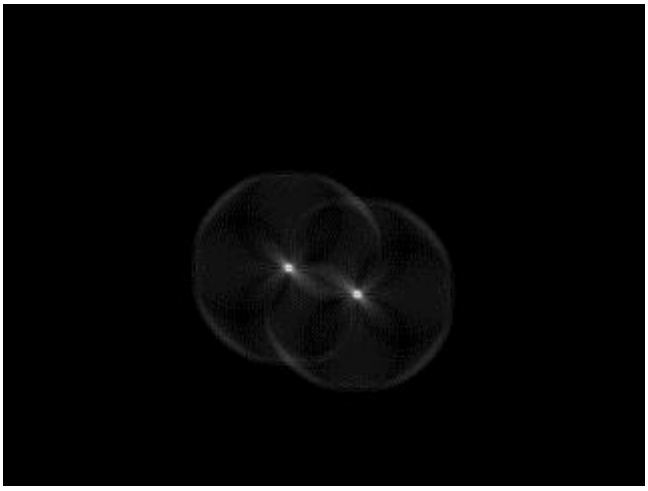

Abbildung 7-5: Abbildung des Hough-Bereichs

Je heller ein Punkt im Hough-Bereich ist, desto mehr Kreise schneiden sich in ihm. Es ist zu sehen, dass die Mittelpunkte der ursprünglichen Kreise sehr hell sind. Das lässt sich darauf zurückführen, dass sich in ihnen alle im Hough-Bereich erzeugten Kreise schneiden. In der folgenden Graphik (Abbildung 7-6 [16]) wurden zufällige drei Punkte ausgewählt und die von ihnen erzeugten Hough-Kreise eingezeichnet.

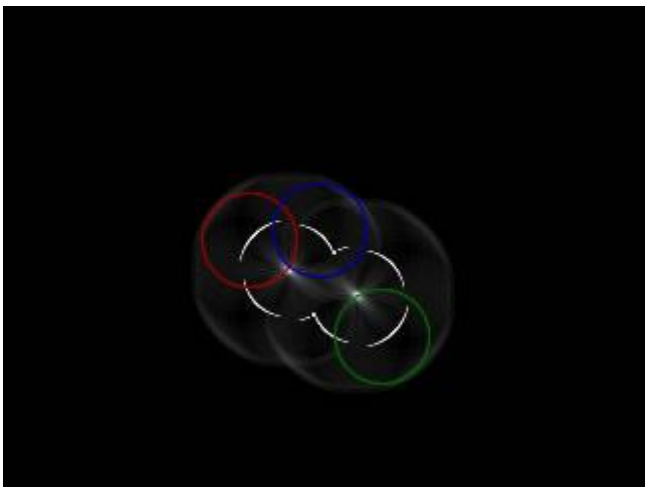

Abbildung 7-6: Drei Punkte im Hough-Bereich

Für jeden Pixel im Hough-Bereich wird mittels der Akkumulationsmatrix bestimmt, wie viele Kreise ihn schneiden. So lassen sich die restlichen Parameter a und b der Gleichung (7.1) bestimmen.

Es ist zu beachten, dass lediglich Kreise mit einem bestimmten Radius auf diese Art und Weise detektiert werden können. Um Kreise mit variablem Radius zu finden, muss

einfach das oben beschriebene Verfahren nach der Trial-and-Error-Methode mehrmals ausgeführt werden. Das kann unter Umständen viel Rechenzeit in Anspruch nehmen, ist jedoch schneller als ein Verfahren für den dreidimensionalen Parameterraum zu entwickeln.

7.1.3 Die Kamera

Als Kamera wurde das Modell C930e von Logitech verwendet [14]:

Abbildung 7-7: C930e von Logitech

Sie weist folgende Eigenschaften auf [14]:

Tabelle 7.1: Eigenschaften von Logitech C930e

Eigenschaft	Wert
Öffnungswinkel (diagonal)	90°
Max. Auflösung	1080p (1920x1080)
Datenübertragung	USB 2.0 oder 3.0

Die sehr hohe Auflösung wird zur Bildverarbeitung nicht benötigt. Um Rechenzeit einzusparen, wird stattdessen mit einer Auflösung von 320x240 Pixeln gearbeitet. Die 1080p-Auflösung kann aber zum Beispiel zur Übertragung eines Livefeeds benutzt werden.

7.2 Die Bilderkennung mit Matlab/ Simulink

Zur Bildverarbeitung eignen sich grundsätzlich sowohl Matlab als auch Simulink. Es wurden beide Ansätze ausprobiert und mit beiden prinzipiell passable Ergebnisse erzielt. Der Matlab-Ansatz lässt sich aufgrund von einigen, nicht Autocoding-fähigen Funktionen allerdings nicht auf dem Beaglebone Black verwenden. Deshalb wurde im Nachgang noch der Simulink-Ansatz erstellt.

7.2.1 Der Matlab-Ansatz

Die Grundidee ist, die Bilder der Kamera nach einem Objekt mit zwei Eigenschaften zu durchsuchen: Es muss rund sein und die richtige Farbe besitzen. Dazu sind in Matlab bereits zwei Funktionen implementiert: *imfindcircles* und die Fähigkeit Bilddateien als Matrizen zu behandeln. Die einzelnen Bildpunkte werden dabei Zahlentupel aus den Werten für rot, grün und blau dargestellt. In Matlab wird ein Bild also als dreidimensionales Array mit den Dimensionen M (horizontale Bildpunkte), N (vertikale Bildpunkte) und 3 (für die RGB-Farben) dargestellt[1].

Farberkennung

Zur Farberkennung in Matlab wird das Bild zunächst in seine drei Farbkomponenten zerlegt. Dies ist mit einer auf Matrizenrechnung spezialisierten Software wie Matlab relativ einfach zu bewerkstelligen. Wenn *img* das zu untersuchende Bild ist, erfolgt die Extrahierung der Farbkomponenten mit den folgenden drei Befehlen:

- r = img(: , : ,1);
- g = img(: , : ,2);
- b = img(: , : ,3);

Aus *img* werden also alle vertikalen und horizontalen Pixel übernommen, aber jeweils nur eine Farbkomponente. Bei r, g und b handelt es sich um Graustufenbilder, die die Intensität der jeweiligen Farbe angeben. Durch Subtraktion der nicht gewünschten

[1] Zum Matlab-Ansatz gehörende Dateien befinden sich auf der beigefügten CD: Image Processing/ Matlab

Farbkomponenten mit einer bestimmten Gewichtung von der gewünschten Komponente (in unserem Fall rot) lässt sich ein Bild erzeugen, dass die Pixel mit besonders hohen roten Anteilen betont. Das kann wie folgt aussehen:

- red_intensity = r - g/2 - b/2;

Nach dieser Operation erscheinen Pixel, in denen rot gar nicht oder nur mit geringerer Intensität Auftritt als schwarz Entsprechend der Intensität des Rotanteils steigt die Helligkeit eines Pixels. Um zu bewerten, ob ein Pixel „rot genug“ ist, wird ein binäres Bild aus der Intensitätsdarstellung gewonnen. Dazu muss ein bestimmter Schwellenwert an Intensität überschritten werden. Nachfolgend wird der Vorgang anhand eines Beispielbildes (siehe Abbildung 7-8) erläutert:

Abbildung 7-8: Roter Ball auf grüner Wiese

Als Beispiel dient dieser rote Ball auf einer grünen Wiese. Erfolgt die Extrahierung der drei Farbkomponenten und Darstellung in Graustufenbildern ergibt sich – am Beispiel Rot – die in Abbildung 7-9 zu sehende Grafik:

Abbildung 7-9: Intensität der roten Komponente als Graustufenbild

In dieser Darstellung steht weiß für die höchstmögliche Intensität und schwarz für ein nicht-Vorhandensein der Farbe Rot. Der Abbildung 7-9 lässt sich entnehmen, dass der Ball in der Mitte des Bildes eine große Menge rotes Licht abstrahlt. Um herauszufinden, wie groß der Rotanteil im Vergleich zu den anderen Primärfarben ist, erfolgt nun die gewichtete Subtraktion der anderen Komponenten (wie oben beschrieben):

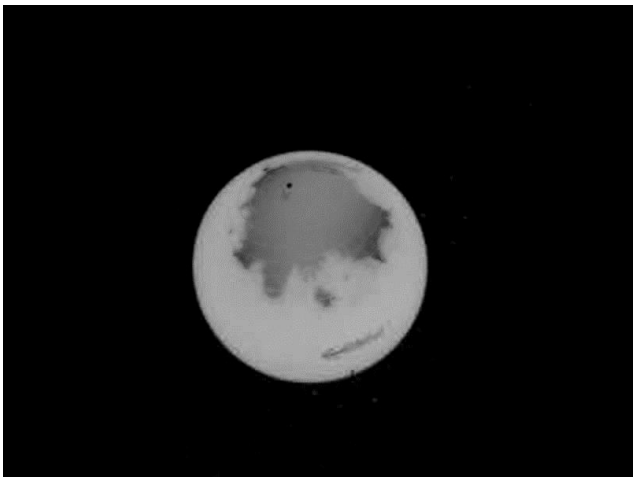

Abbildung 7-10: Roter Ball nach Abzug der anderen Komponenten

In Abbildung 7-10 steht weiß wieder für eine besonders hohe Intensität. Der Ball wird nun erneut mit überwiegend weißen Pixeln dargestellt. Der dunkle Fleck auf seiner Oberseite rund um das Ventil entsteht dadurch, dass hier die Sonne besonders stark reflektiert wird. Das reflektierte Licht ist weiß und enthält damit alle Farben. Neben einem hohen Rotanteil treten hier also auch grün und blau mit hoher Intensität auf. Werden diese Komponenten mit einer Gewichtung von 0,5 subtrahiert, erscheint der Ort, an dem alle Komponenten mit hoher Intensität auftreten dunkler. Nun erfolgt noch

eine Untersuchung welche Pixel in der gewichteten Intensitätsdarstellung einen bestimmten Schwellwert überschreiten:

- bin = red_only > tresh;

Das Ergebnis ist ein binäres Bild, dessen Pixel den Wert 0 (schwarz) und 1 (weiß) annehmen können. Und zwar sind alle Pixel, die in *red_only* einen bestimmten Intensitätswert überschreiten, weiß. Die Intensität wird in Graustufen von 0 bis 255 gemessen. Bei einem Schwellenwert von *tresh* = 50 ergibt sich die in Abbildung 7-11 zu sehende Darstellung:

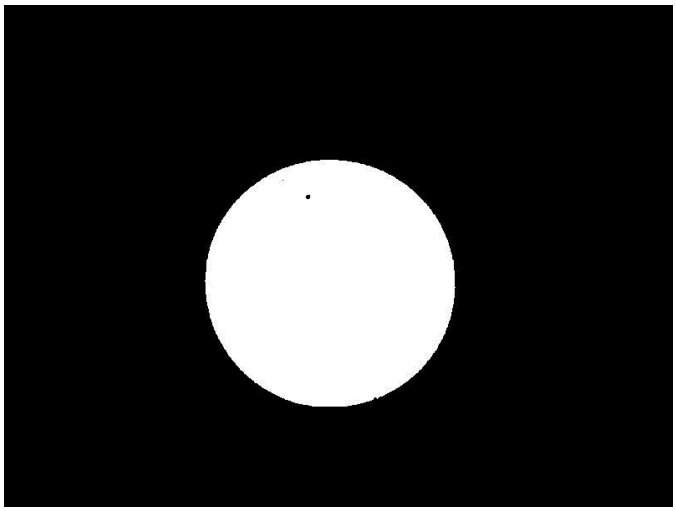

Abbildung 7-11: Schwellenwertbetrachtung der Intensität

Offensichtlich überschreiten alle zum Ball gehörenden Pixel den Schwellenwert (bis auf das Ventil). Eine Erkennung eines roten Objektes unter diesen Bedingungen ist also relativ gut realisierbar. Dies lässt sich unter anderem darauf zurückführen, dass es sich sowohl bei der Farbe des Balles, als auch der Farbe des Hintergrundes um Komponenten des RGB-Modells handelt.

Die Position des Balles im Bild muss allerdings noch bestimmt werden. Dazu wird am besten sein Mittelpunkt als Referenzpunkt verwendet. Die Funktion *find* findet alle besetzten Elemente einer Matrix. Wird diese Funktion also auf das binäre Bild *bin* angewendet, werden zwei Vektoren mit den Positionen aller weißen Pixel erstellt:

- [pos_x, pos_y] = find(bin);

Durch die Funktionen *mean* und *round* wird für beide Dimensionen (horizontal und vertikal) der Mittelwert gebildet und auf den nächsten ganzzahligen Wert gerundet:

- x_m = round(mean(pos_x));
- y_m = round(mean(pos_y));

Diese Werte geben die Koordinate des Mittelpunktes des Balles an.

Kreiserkennung

Mit der Funktion *imfindcircles* können Kreise in einem Bild detektiert werden. Die Funktion wendet die Hough-Transformation an, wobei die Kantendetektion nach Canny bereits implementiert ist. Ein allgemeiner Aufruf dieser Funktion sieht wie folgt aus:

- [centers, radii] = imfindcircles(img, radius_range, 'ObjectPolarity', 'Sensitivity');

Die Funktion übergibt die Koordinaten der Mittelpunkte aller gefundenen Kreise und deren Radien. Dazu benötigt sie folgende Eingangsparameter: Ein Bild (img), ein Bereich von zu untersuchenden Radien[2] (radius_range), die Polarität der zu findenden Kreise (sollen Kreise gefunden werden, die hell oder dunkel im Vergleich zum Hintergrund sind?) und die Sensitivität des Algorithmus: Dieser Wert hängt direkt mit der Hough-Transformation zusammen. Er bewegt sich zwischen 0 und 1 und ist ein Maß für die Menge an Kreisen im Hough-Bereich, die einen Punkt schneiden müssen. Damit wird dieser als Mittelpunkt eines Ausgangskreises erkannt. Je größer der Sensitivitätswert ist, desto schneller wird ein Kreis erkannt.

Die Funktion *imfindcircles* verwendet Subfunktionen die von der für das Projekt erforderlichen Autocoding-Funktion nicht unterstützt werden. Das ist der Grund für das Scheitern des Matlab-Ansatzes, allerdings wurden trotzdem wertvolle Erfahrungen im Bereich der Bildverarbeitung mit Matlab gesammelt.

Da die Kreiserkennung unter Umständen mehr Kreise erkennt, als die Farberkennung rote Objekte, ist eine Logik notwendig, die dem roten Objekt (sofern vorhanden) den richtigen Kreis zuordnet. Dazu wird die Distanz (in Pixeln) zwischen den Mittelpunkten der Kreise zum Mittelpunkt des farbigen Objekts berechnet. Wenn die kleinste Distanz einen bestimmten Wert (zehn Pixel hat sich als passable Größe erwiesen) unterschreitet, dann wird der dazugehörige Kreis als roter Ball akzeptiert.

[2] Wir erinnern uns: die Hough Transformation funktioniert im Prinzip nur mit einem festen Radius, für variable Radien muss der Algorithmus mehrmals durchlaufen werden

Da *imfindcircles* auch die Radien der detektierten Kreise liefert, lässt sich die Entfernung zum Ball und damit über die Lagewinkel der Drohne auch die Flughöhe bestimmen. r_m sei der Radius der Balls in Metern, r_p der von *imfindcircles* gefundene Radius in Pixeln. Aus dem Verhältnis $\frac{r_p}{r_m}$ die von der Kamera abgedeckte Entfernung in horizontaler und vertikaler Richtung in Metern berechnen (aus Sicht der Kamera, nicht in Bezug auf den Horizont), und damit über die Öffnungswinkel der Kamera deren Höhe über dem Ball. P_h und P_v seien jeweils die Anzahl der Pixel des Bildes in beiden Dimensionen (also die Auflösung der Kamera). Dann gilt (für die horizontale Bilddimension):

$$\frac{r_p}{r_m} = \frac{P_h}{d_h} \qquad (7.2)$$

d_h ist dabei die von der Kamera abgedeckte Strecke. Für die Höhe der Kamera gilt damit:

$$h = \frac{d_h}{\tan\frac{\alpha_h}{2}} = \frac{P_h \cdot r_m}{r_p \cdot \tan\frac{\alpha_h}{2}} \qquad (7.3)$$

α_h ist der Öffnungswinkel der Kamera in horizontaler Richtung. Die Berechnung der Flughöhe per Bilderkennung ist insofern sinnvoll, da die GPS-Höhe sehr ungenau ist. So könnte ein Abgleich durchgeführt werden, mit dem die Integrität der GPS-Höhe überprüft werden kann.

7.2.2 Der Simulink-Ansatz

Ähnlich wie mit Matlab soll das Bild mit Simulink sowohl nach Farbe als auch Muster durchsucht werden. Dazu wird das in Abbildung 7-12 dargestellte Blockschaltbild verwendet[3]:

[3] Zugehörige Dateien auf CD unter Image Processing/ Simulink

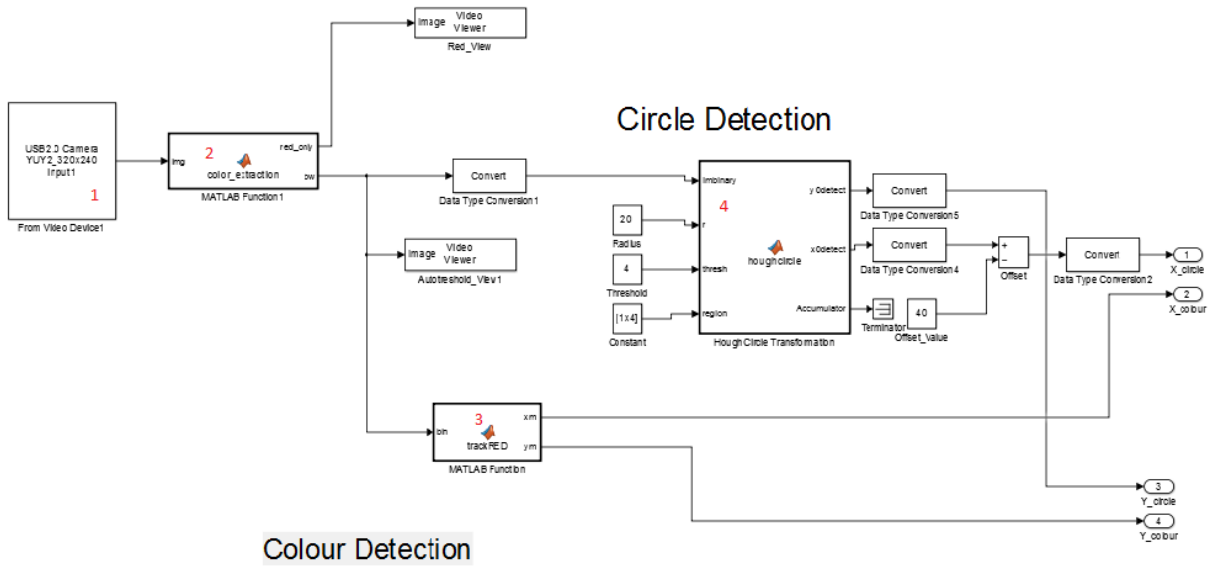

Abbildung 7-12: Blockschaltbild zur Bilderkennung in Simulink

Allgemeines

Block 1 ist der Kamerablock. Mit ihm werden die einzelnen Frames aus der angeschlossen Kamera ausgewählt. In einem Menü lassen sich diverse Parameter zur Bilderfassung einstellen (siehe Abbildung 7-13):

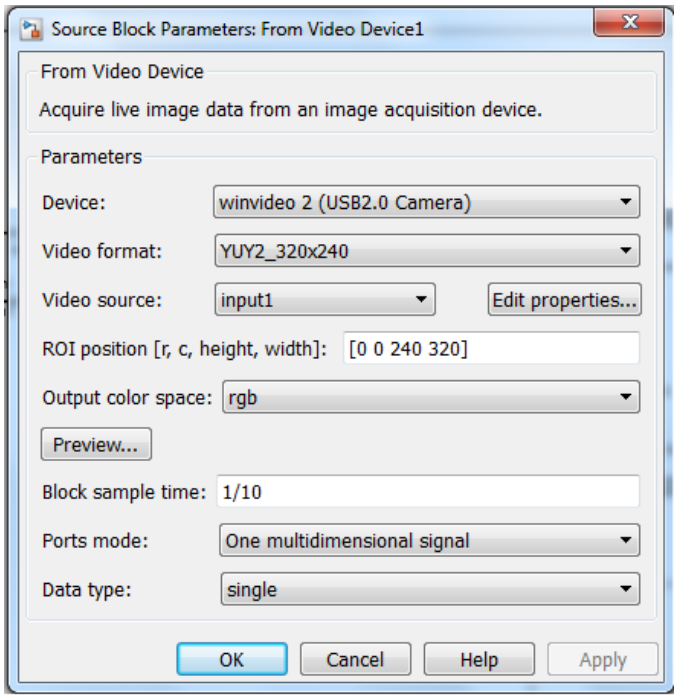

Abbildung 7-13: Kameraparameter

Unter *Device* werden alle angeschlossen Kameras angezeigt; dort kann das gewünschte Gerät ausgewählt werden. Unter *Video format* kann die gewünschte Bildauflösung eingestellt werden. Für unsere Zwecke ist eine Auflösung von 320 x 240- Pixeln ausreichend. Da es nur um Objekterkennung geht und nicht um eine für das menschliche Auge schöne Darstellung, ist keine höhere Auflösung notwendig. Für die vorliegende Anwendung wurde selbstverständlich die Darstellung im RGB-Farbraum gewählt; ansonsten sind noch Darstellungen in einem Graustufenmodell und im YCbCr-Raum auswählbar. Unter *Ports mode* lässt sich einstellen, über wie viele Ports das Bildsignal ausgegeben wird. Es können zum Beispiel alle RGB-Komponenten als separate Intensitätsdarstellungen ausgegeben werden. Da in Block 2 aber die bereits von Matlab-Ansatz bekannte Aufteilung der Komponenten stattfindet, wird das Kamerasignal als einfaches Signal übergeben, um die Anzahl der Eingangsparameter des nachfolgenden Blockes gering zu halten.

Im Block 3 wird nach der gleichen Methode wie beim Matlab-Ansatz (Kapitel 7.2.1) die Koordinate des Mittelpunktes des Balles aus der Farberkennung berechnet. Das in Block 2 erstellte Binärbild dient dazu als Grundlage. Dieses dient außerdem als Eingangssignal für die Hough-Transformation (s. Kapitel 7.1.2). Bei Hough-Transformation im Simulink-Ansatz handelt es sich um eine andere Implementierung als im Matlab-Ansatz. Mit dieser können lediglich die Koordinaten von detektierten Kreisen erfasst werden, nicht aber deren Radien.

Berechnung des Schwellwertes zur Binärdarstellung

In Block 2 wird die auf die gleiche Art und Weise wie bei dem Matlab-Ansatz ein Binärbild erzeugt. Dazu wird – wie beim Matlab-Ansatz – ein fixer Schwellwert verwendet. In Simulink selbst ist die Möglichkeit einen fixen Schwellwert zu implementieren nicht gegeben. Stattdessen kann mit Simulink mit der sogenannten Otsu-Methode nur ein automatischer Schwellwert berechnet werden [15]. Die Otsu-Methode basiert auf der Verteilungsfunktion der Pixel des Graustufenbildes (*red_only*). Nach dieser Methode wird der Schwellwert so berechnet, dass die Summe der Varianzen der Verteilungsfunktionen der Vorder- und Hintergrundpixel so gering wie möglich wird. Der Einsatz dieses Algorithmus ergibt durchaus Sinn, wenn es darum geht ein Graustufenbild so Originalgetreu wie möglich in den Binärbereich abzubilden. Im Idealfall sieht Kamera der Locator-Drohne nur einen roten Ball, der auf einer grünen Wiese liegt. In diesem

Fall ergibt sich kein großer Unterschied zwischen der Otsu-Methode und einem fixen Schwellwert. Die folgende Graphik (Abbildung 7-14) soll dies veranschaulichen:

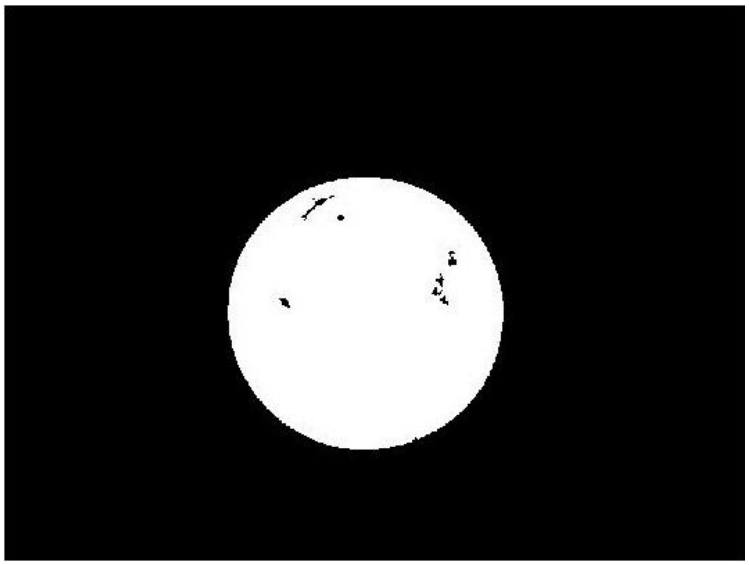

Abbildung 7-14: Binärdarstellung nach Otsu

Die Darstellung basiert genau wie Abbildung 7-10 auf Abbildung 7-7. Nun werden einige Stellen des Balles nicht als rot „genug“ erkannt. Für die Farberkennung, die auch beim Simulink-Ansatz zum Einsatz kommt, spielt das nur eine kleine Rolle, da die gesamte Masse an weißen Pixeln davon kaum beeinflusst wird. Auch auf die Kreiserkennung nach Hough wirken sich diese kleinen Unterschiede nicht aus, da die Kreisform nicht beeinflusst wird.

Zu größeren Problemen kommt es unter Umständen, wenn sich mehrere rote Objekte im Erfassungsbereich der Kamera befinden. Dazu wird folgende Fotografie (Abbildung 7-15) betrachtet:

Abbildung 7-15: Zwei rote Bälle

Der kleinere Ball an der Seite stellt nun ein nicht gewolltes, dem Zielobjekt aber ähnliches Störobjekt dar. Wird aus diesem Kamerabild nun über die zwei besprochenen Methoden ein Binärbild erzeugt, ergeben sich die folgenden Darstellungen (siehe Abbildung 7-16):

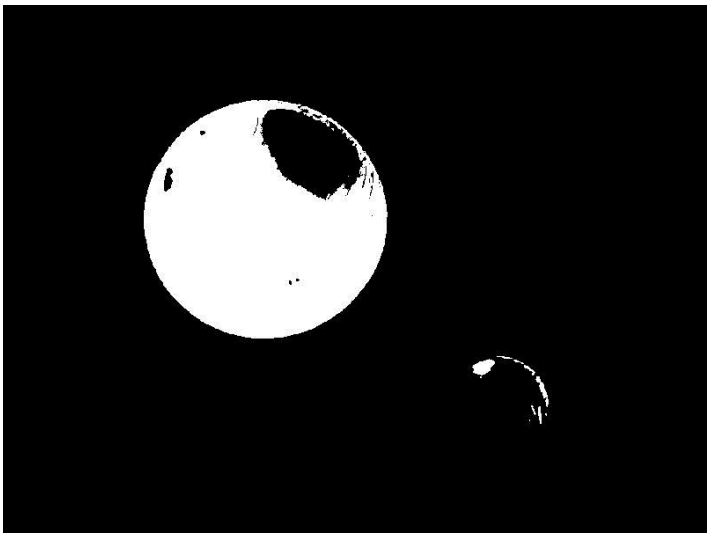

Abbildung 7-16: Binärdarstellung mit fixem Schwellwert

Das Störobjekt wird in dieser Darstellung relativ gut unterdrückt. Dafür wird ein Teil des Zielobjektes ebenfalls unterdrückt, was sich auf die Genauigkeit der Berechnung der Zielkoordinate durch die Farberkennung auswirkt. Nach der Otsu-Methode kommt folgende Darstellung (Abbildung 7-17) zustande:

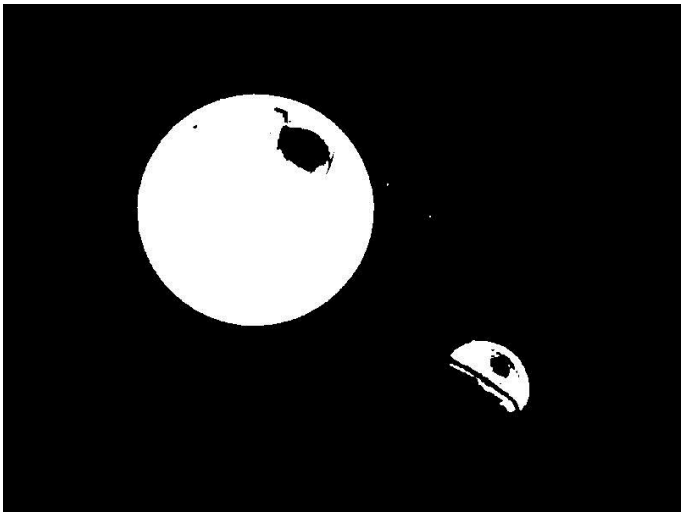

Abbildung 7-17: Binärdarstellung durch Otsu-Methode

Hier wird das Störobjekt nicht so gut unterdrückt. Die Genauigkeit der Berechnung der Zielkoordinate wird nun ebenfalls beeinflusst, diesmal vom nicht unterdrückten Störobjekt. Als zusätzlicher Störfaktor kommt hinzu, dass die Kreiserkennung auf das störende Objekte anspringen kann. Je weiter sich die Kamera von den beiden Objekten entfernt, desto näher erscheinen sie für sie. Eine eindeutige Zuteilung der „Farbkoordinate“ an eine „Kreiskoordinate“ wird dadurch schwerer.

Beide Methoden verfügen über Vor- und Nachteile. Die Otsu-Methode berechnet den Schwellwert ständig selbst, weswegen kein menschliches Eingreifen nötig ist. Dafür werden Störobjekte in der Regel nicht so gut unterdrückt. Die Einstellung eines fixen Schwellwertes dreht diese Erscheinung genau um. Das Eingreifen eines Bedieners muss aber nicht als Problem, sondern als Chance begriffen werden. Da die Drohne sowieso über einen Echtzeit Datenlink verfügt, kann der Schwellwertparameter jederzeit den sich wechselnden Umweltbedingungen angepasst werden, denn seiner optimaler Wert hängt sowohl von den Lichtbedingungen ab, als auch von potentiellen Störobjekten (gibt es z.B. rote Büsche auf dem überflogenen Gebiet?). Diese Einflüsse kann ein erfahrener Bediener bei der Wahl des Parameters beachten, wobei die automatische Schwellwertberechnung keine Berücksichtigung darauf nimmt (was daran liegt, dass sie für andere Anwendungen konzipiert wurde).

7.3 Berechnung der Zielkoordinate aus Bilderfassung

Die Berechnung der Zielkoordinate erfolgt ebenfalls mit Simulink. Dazu wird das in Abbildung 7-18 zu sehende Blockschaltbild verwendet:

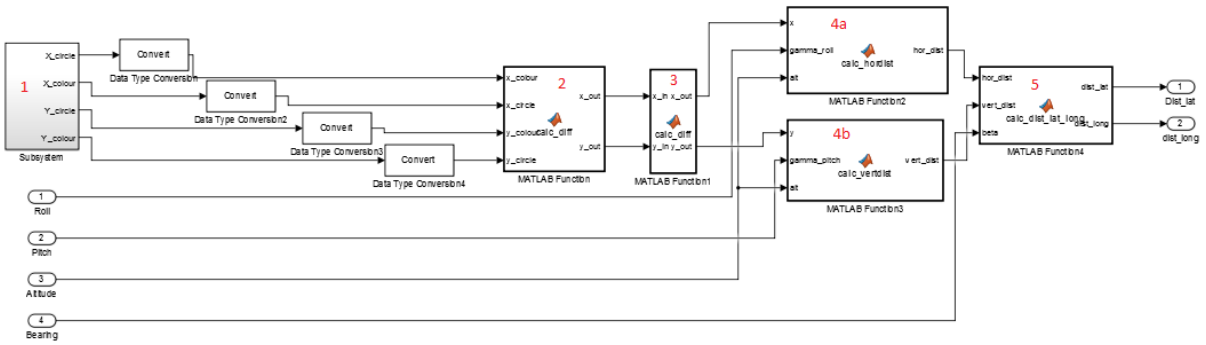

Abbildung 7-18: Blockschaltbild zur Berechnung von Zielkoordinate

In Block 1 wird der Ball per Bilderkennung detektiert seine Position im Bild nach zwei Berechnungsmethoden ausgegeben (siehe Kapitel 7.2.2). Die Blöcke zwei bis 5 werden in diesem Teilkapitel erklärt.

7.3.1 Vereinbarung der zwei Methoden zur Berechnung der Zielkoordinate

Block 1 gibt zwei Signale aus: Die Koordinaten des Zielobjektes sowohl nach Farb- als auch Kreiserkennung. Diese werden in Block 2 zu einer einzigen Koordinate verrechnet:

$$X_{out} = \frac{1}{2}(X_{Colour} + X_{Circle}) \qquad (7.4)$$

$$Y_{out} = \frac{1}{2}(Y_{Colour} + Y_{Circle}) \qquad (7.5)$$

Dazu wird einfach aus den zwei Koordinaten der Mittelwert gebildet. Diese Berechnung wird allerdings nur dann durchgeführt, wenn sowohl die x- als auch die y-Koordinaten weniger als 50 Pixel auseinander liegen. Im Mittel liegen die beiden Koordinaten bei detektiertem Ziel weniger als diese Distanz auseinander. Ist für eine der beiden Dimensionen der Unterschied größer als dieser Wert, dann wird für beide Komponenten der Koordinate ein Wert von 400 angenommen. So wird signalisiert, dass kein Objekt detektiert wurde.

Für die in den Blöcken 4 und 5 erfolgenden Berechnungen muss der Mittelpunkt des bildeigenen Koordinatensystems in die Bildmitte verschoben werden, da Matlab/Simulink ihn in die obere linke Ecke des Bildes legt. Bei einer Auflösung von 320 x 240 Pixeln müssen dazu folgende Berechnungen angewendet werden:

$$X_{out} = X_{in} - 160 \qquad (7.6)$$

$$Y_{out} = 120 - Y_{in} \qquad (7.7)$$

Die x-Koordinate wird um 160 Pixel nach links geschoben, die y-Koordinate um 120 nach oben.

7.3.2 Berechnung der Distanz zum Zielobjekt

Dieses Kapitel befasst sich mit der Berechnung der Entfernungen zum Zielobjekt quer und parallel zur Rollachse der Drohne. Dabei ist nur die Entfernung auf dem Boden von Bedeutung, nicht die tatsächliche Entfernung zum Ziel, da aus den Ergebnissen anschließend die laterale und longitudinale Entfernung bestimmt wird. Von Interesse ist also nur, wie weit das Ziel in Nord/ Süd- und Ost/ West-Richtung entfernt ist. Berücksichtigt werden dabei auch Roll- und Pitch- Winkel. Zu beachten ist, dass lediglich Roll ODER Pitch eintreten dürfen, da keine bessere Berechnungsmethode gefunden wurde.

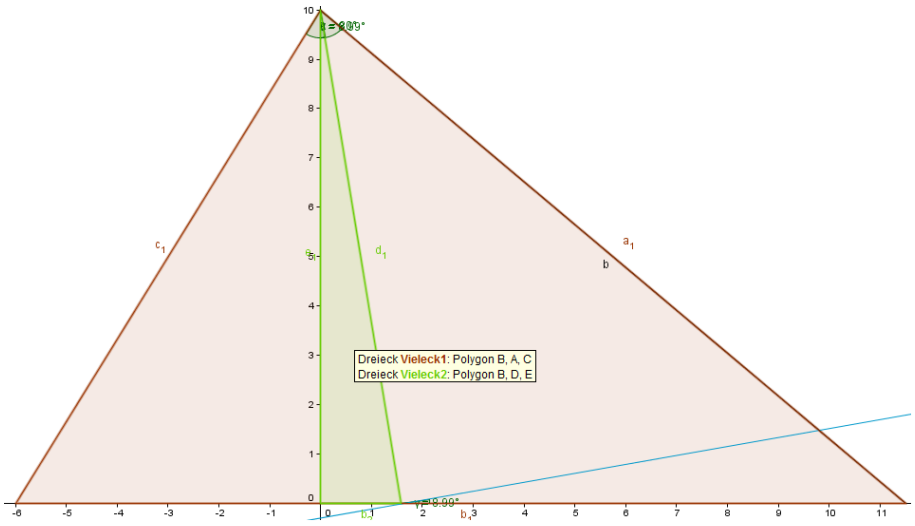

Abbildung 7-19: Darstellung Drehung der Drohne um eine Achse

Abbildung 7-19[4] symbolisiert die Drehung der Drohne um eine Achse (Roll oder Pitch). Das grüne Dreieck zeigt die Abweichung der Drohnennormalen vom Nullwinkel, also vom Parallelflug zur Erdoberfläche. In diesem Beispiel beträgt die Abweichung β =

[4] Abbildungen 7-18 bis 7-20 wurden mit der Geometrie-Software „Geogebra“ erstellt

8,99°. Das rote Dreieck gibt den Öffnungswinkel der Kamera an (hier: α = 80°). Es ist ebenfalls um β gekippt. Die blaue Gerade steht für die „fiktive Erdoberfläche“, die durch die Rotation entsteht. Sie wird im Folgenden als *g* bezeichnet.

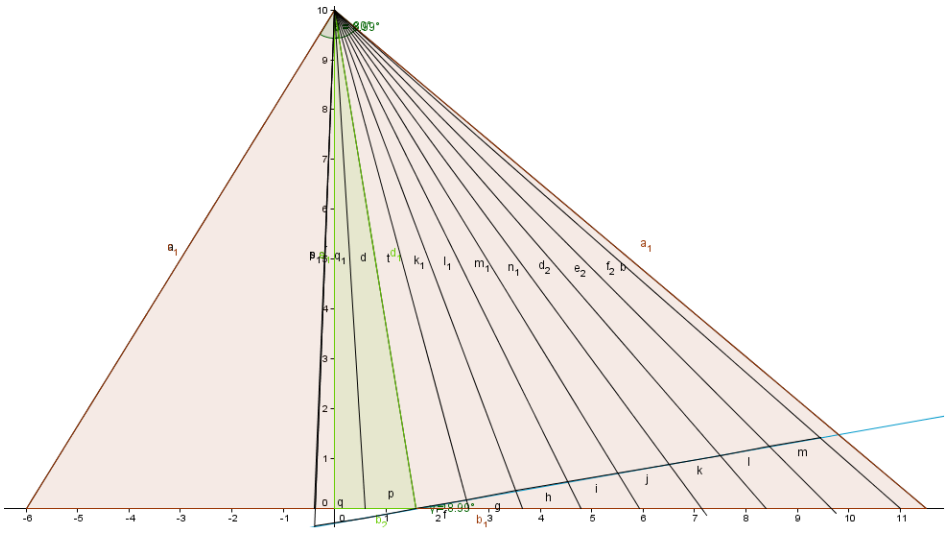

Abbildung 7-20: Von Kamera erzeugte Pixel

Abbildung 7-20 symbolisiert die Aufteilung des Erdbodens in gleiche Strecke durch die Pixel der Kamera. Die Gerade g wird dabei in N gleich große Strecken der Länge d zerlegt, wobei N die Anzahl der verfügbaren Pixel der jeweiligen Dimension angeben (x oder y). Das Ziel wird nun an einem bestimmten Pixel detektiert. Ziel ist es herauszufinden, welche Distanz dieser Pixel zwischen Zielobjekt und Drohne repräsentiert. Abbildung 7-21 soll dies veranschaulichen:

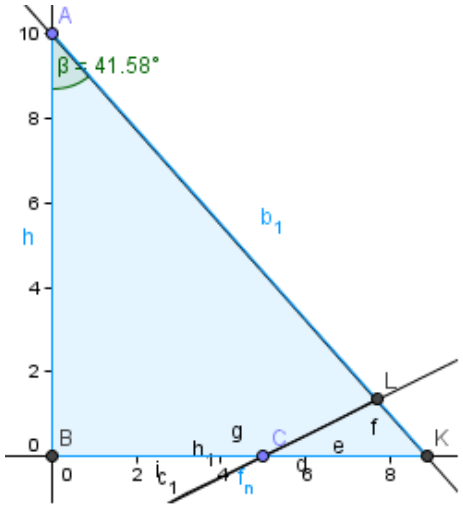

Abbildung 7-21: Darstellung eines einzelnen Pixels

Sei f_n die gesuchte Distanz zum Zielobjekt, dann gilt:

$$f_n = h \cdot \tan(\gamma + \delta_n) \qquad (7.8)$$

δ_n ist dabei der zusätzliche Winkel zwischen der gekippten Drohnennormalen und dem Strahl des „Zielpixels“ n und γ gibt den Drehwinkel der Drohne an. *h* ist die Flughöhe der Drohne. Für δ_n gilt:

$$\tan \delta_n = \frac{n \cdot d}{a} \qquad (7.9)$$

a ist dabei die Länge der gekippten Drohnennormalen. Sie berechnet sich zu

$$a = \sqrt{h^2 + e^2} = h \cdot \sqrt{1 + \tan \gamma^2} \qquad (7.10)$$

e ist die Entfernung zwischen den Auftreffpunkten auf der Erdoberfläche der ursprünglichen Drohnennormalen h und der gekippten a. Die Strecke d lässt sich über den Öffnungswinkel der Kamera α und die Flughöhe h bestimmen:

$$d = \frac{2h \cdot \tan \frac{\alpha}{2}}{N_T} \qquad (7.11)$$

Werden die Gleichungen (7.9), (7.10) und (7.11) in (7.8) eingesetzt, ergibt sich folgender Term:

$$f_n = h \cdot \tan(\gamma + \tan^{-1} \frac{2n \cdot \tan \frac{\alpha}{2}}{N_T \cdot \sqrt{1 + \tan(\gamma)^2}}) \qquad (7.12)$$

Diese Gleichung existiert sowohl für Roll- als auch Pitchrotation bzw. für die Entfernung quer und längs zur Rollachse der Drohne. Sie berücksichtigt also jeweils nur einen Winkel. Gleichzeitiges Rollen und Pitchen sind mit dieser Berechnung zu vermeiden.

7.3.3 Berechnung der lateralen und longitudinalen Distanz zum Zielobjekt

Folgende Abbildung 7-22 illustriert den Zusammenhang zwischen Quer- und Längsentfernungen und Lateral- und Longitudinalentfernungen:

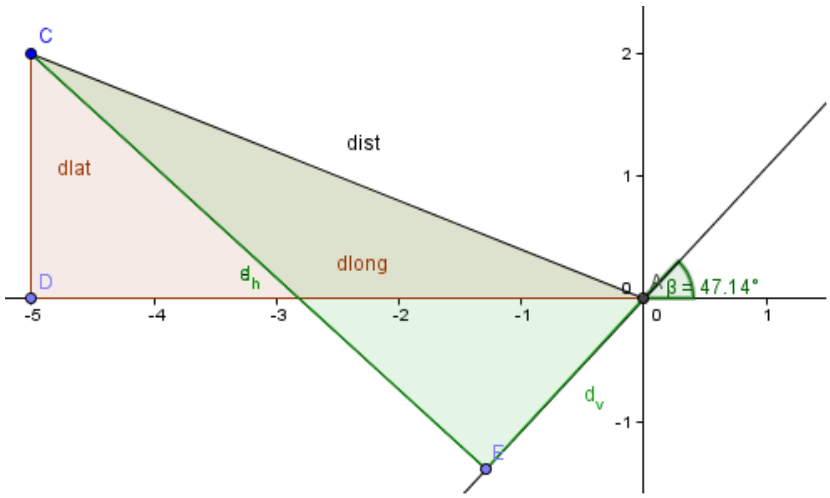

Abbildung 7-22: Zusammenhang zwischen Distanzen

Zur Berechnung der longitudinalen und lateralen Distanz müssen die jeweiligen Komponenten der horizontalen und vertikalen Distanz zum Objekt berechnet und addiert bzw. subtrahiert werden. Abhängig vom Azimuthwinkel ϕ (90°-β) ergibt sich:

$$d_{long} = d_v \cdot \sin\phi + d_h \cdot \cos\phi \qquad (7.13)$$

$$d_{lat} = d_v \cdot \cos\phi - d_h \cdot \sin\phi \qquad (7.14)$$

Die berechneten Entfernungen werden benutzt, um die GPS-Koordinate des Zielobjekts zu bestimmen. Diese werden dem Bordcomputer übergeben und können so angesteuert werden.

8 Zusammenfassung und Ausblick

Die Thematik der Schwarmintelligenz beinhaltet die Interaktion der Schwarmmitglieder. Schwerpunkte stellen hierbei die im Kollektiv lokalisierte Intelligenz, die Kommunikation zwischen den einzelnen Multicoptern und die durch den Schwarm ausgeführten Missionen dar.

In dieser Studienarbeit werden im Bereich der Swarm Intelligence die Kernthemen Schwarmkonzepte, Programmierung der on-board Flugsoftware und die Verifikation durch verschiedene Simulationen abgehandelt.

Unter dem Punkt Schwarmkonzepte werden mögliche Einsatzszenarien dargestellt, die ein Schwarm von Multicoptern zukünftig ausführen könnte. Darüber hinaus wird ein Schema vorgestellt, durch das die optimale Konfiguration der einzelnen Schwarmmitglieder für eine Mission ermittelt wird, sodass diese mit optimalen Ressourcenverbrauch ausgeführt werden kann.

Das Hauptaugenmerk liegt in dieser Studienarbeit auf der Implementierung einer Flugsoftware, sodass statischen und dynamischen Hindernissen ausgewichen werden kann, damit keine Kollisionen mit einem Multicopter entstehen. Die programmierte Software wird detailliert erläutert und durch eine Vielzahl von Simulationen in Matlab/Simulink auf ihre Funktionsweise hin getestet. Die Verifikationsschritte liefern durchgängig positive Ergebnisse, weshalb die Software zukünftig zu Testzwecken in einem Flugversuch verwendet werden kann.

Eine weiterführende Studienarbeit im Bereich der Schwarmintelligenz sollte zur Verwirklichung eines Schwarms aus Multicoptern die Kommunikationsstruktur zwischen den einzelnen Mitgliedern herausarbeiten. Erst dadurch können die vorgestellten Schwarmkonzepte an realen Multicoptern verwirklicht werden. Des Weiteren sollten, wie bereits angesprochen, die programmierten Algorithmen zur Flugbahnberechnung und -steuerung durch einen Flugversuch verifiziert und bedarfsweise ver- und ausgebessert werden.

Um mit der Bilderkennung in Zukunft nicht nur die aktuelle Position des Balles bestimmen zu können, sondern auch Aussagen über dessen voraussichtliche zukünftige Position treffen zu können, wäre es wünschenswert, wenn eine Art Interpolation der Position einzurichten. So könnte die Verfolgung beschleunigt werden. Des Weiteren ist

zu beachten ist, dass durch die Linsenverzerrung der Kamera unter Umständen Berechnungsfehler entstehen können, speziell in den Bildecken. Durch einen entsprechenden Anti-Distortion-Algorithmus kann diese ausgeglichen werden. Auf diesem Gebiet wurde bereits viel entwickelt, sodass eventuell ein bereits fertiger Algorithmus in modifizierter Form verwendet werden kann.

Ebenfalls problematisch ist die aktuelle, starre Kameraaufhängung. Hier wäre eine flexible Lösung wünschenswert, um die durch die Drohne verursachten Vibrationen auszugleichen. Wie störend diese sind kann aufgrund mangelnder Testerfahrung momentan nur gemutmaßt werden, es ist aber davon auszugehen, dass die Bilderfassung ohne Vibrationen wesentlich besser funktioniert.

Graphische Hinderniserkennung wurde bisher noch gar nicht betrachtet, auch hier ergeben sich potentielle Arbeitspakete.

Literaturverzeichnis

[1] „Finding Dulcinea - Librarian of the Internet“, http://www.findingdulcinea.com/news/on-this-day/July-August-08/On-this-Day--Austria-Rains-Balloon-Bombs-on-Venice.html, Einsichtnahme: 28.06.2015

[2] „Vision Systems“, http://www.vision-systems.com/articles/2013/10/uavs-guide-students-around-mit-campus.html, Einsichtnahme: 28.06.2015

[3] „Vision Systems“, http://www.vision-systems.com/articles/2013/07/u-s-coast-guard-makes-first-drug-bust-using-uavs.html, Einsichtnahme: 28.06.2015

[4] Amazon : „Amazon“, http://www.amazon.com/b?ie=UTF8&node=8037720011, Einsichtnahme: 29.06.2015

[5] B. Alavi, F. Bachmann und F. Holfert: „Aktorik, Lernen, Motion Tracking, diverse Roboter: Schwärme - steering behaviors“, http://www2.informatik.huberlin.de/ki/lehre/ws0304/kogrob/Schwarmverhalten_ohnevideos.ppt, Einsichnahme: 03.07.2015

[6] C. Reynolds: „Boids“, http://www.red3d.com/cwr/boids/, 01.09.2001, Einsichtnahme: 24.03.2015

[7] Civil Aviation Authority: „Annex F: Standardised European Rules Of The Air (SERA) & 'Rules Of the Air Regulation' 2014“, 2014

[8] „Engineers Garage“, http://www.engineersgarage.com/articles/image-processing-tutorial-applications, Einsichtnahme: 29.06.2015

[9] Wikipedia: „RGB color model “, https://en.wikipedia.org/wiki/RGB_color_model, Einsichtnahme: 29.06.2015

[10] Wikipedia: „CMYK Color Model“, https://en.wikipedia.org/wiki/CMYK_color_model, Einsichtnahme: 29.06.2015

[11] „Farbe Computer“, http://farbe-computer.de/kapitel22.html Einsichtnahme: 28.06.2015

[12] „Cite Seer X“, http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.324.4122&rep=rep1&type=pdf, Einsichtnahme: 30.06.2015

[13] Wikipedia: „Canny Edge Detector“, https://en.wikipedia.org/wiki/Canny_edge_detector, Einsichtnahme: 30.06.2015

[14] Logitech: „Logitech.com“, http://www.logitech.com/assets/47868/logitech-webcam-c930e-data-sheet.pdf, Einsichtnahme: 03.07.2015

[15] Mathworks: „Otsu's Method“, http://de.mathworks.com/help/vision/ref/autothreshold.html, Einsichtnahme: 02.07.2015

[16] U. Sinha: „Aishack“, http://www.aishack.in/tutorials/circle-hough-transform/, Einsichtnahme: 30.06.2015

[17] C. F. Carlson: „Center for Imaging Science“, https://www.cis.rit.edu/class/simg782/lectures/lecture_10/lec782_05_10.pdf, Einsichtnahme: 30.06.2015