Wilhelm Schickard-Institut für Informatik

Diplomarbeit

Verbesserte Eigenbewegungsschätzung für Flugroboter

von

Deniz Bahadir

Korrigierte Version: 1. April 2009

Betreuer:
Prof. Dr. rer. nat. Hanspeter A. Mallot
Lehrstuhl Kognitive Neurowissenschaft

Prof. Dr. rer. nat. Andreas Zell
Lehrstuhl Rechnerarchitektur

Tag der Anmeldung: 21. April 2008
Tag der Abgabe: 20. Januar 2009

Eberhard Karls
Universität
Tübingen

Ich erkläre hiermit, daß ich die vorliegende Arbeit selbständig verfaßt und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Tübingen, den 20. Januar 2009

Kurzfassung:

Autonomes Verhalten mobiler Roboter setzt die Kenntnis ihrer Bewegungsabläufe voraus. Für Flugroboter stellt sich deren Bestimmung als besonders schwierig dar. Die vorliegende Arbeit setzt sich mit der Möglichkeit auseinander, durch Entwicklung geeigneter Algorithmen eine verbesserte Abschätzung der Eigenbewegung und damit eine größere Autonomität von Flugrobotern zu erreichen.

Im ersten Teil dieser Arbeit werden der verwendete Flugroboter mit dem zugehörigen Computersystem vorgestellt, einige programmier-technische und biologische Grundlagen angesprochen sowie die verwendeten Algorithmen beschrieben und erläutert. Im zweiten Teil wird das in dieser Arbeit entwickelte und sehr modular gehaltene Softwaresystem zur Ansteuerung des verwendeten Flugroboters präsentiert und beschrieben. Weiterhin wird das zur Beschreibung des Flugverhaltens aufgestellte mathematische Modell vorgestellt und mit Hilfe von verschiedenen flugexperimentellen Versuchen verifiziert und getestet.

Es konnte gezeigt werden, daß das entwickelte Softwaresystem eine komfortable und effiziente Ansteuerung des Roboters ermöglicht und zugleich so modular gehalten ist, daß es sich leicht um weitere Funktionalitäten erweitern läßt. Darüber hinaus konnte gezeigt werden, daß das entwickelte Modell für das Flugverhalten eine gute Prädiktion der Eigenbewegung ermöglicht, mit deren Hilfe sich eine aus optischen Flußdaten berechnete Eigenbewegungsschätzung verbessern ließe.

Danksagung:

Diese Diplomarbeit entstand am Lehrstuhl Kognitive Neurowissenschaft der Eberhard Karls Universität Tübingen unter Leitung von Prof. Dr. Hanspeter A. Mallot. Ihm und Prof. Dr. Andreas Zell vom Lehrstuhl Rechnerarchitektur des Wilhelm-Schickard-Instituts für Informatik der Universität Tübingen möchte ich für die Vergabe und Betreuung der Diplomarbeit danken.
Danken möchte ich auch Fabian Recktenwald, der mir über die ganze Zeit mit Rat und Tat zu Seite stand und mir vor allem zum Ende der Arbeit hin eine sehr große Hilfe war.
Ein ganz besonderer Dank gilt Dr. Hansjürgen Dahmen für die vielen anregenden Diskussion und Vorschläge, sowie die Hilfe beim Bau der verwendeten Schaltungen. Ich habe ihn mit meinen Fragen so manches mal von seiner eigentlichen Arbeit abgehalten und bin ihm deshalb besonders dankbar, daß er sich dennoch immer diese Zeit für mich genommen hat.
Mein Dank gilt auch Prof. Dr.-Ing. Rolf Radespiel, Institut für Strömungsmechanik der Technischen Universität Braunschweig, sowie Prof. Dr.-Ing. Peter Vörsmann, Institut für Luft- undRaumfahrtsysteme der Technischen Universität Braunschweig, für wertvolle Tipps und Hinweise zu Fragen der Aerodynamik und Strömungsmechanik.
Außerdem gilt mein Dank Dominik Seffer für die freundliche und lustige Arbeitsatmosphäre. Er konnte mich so manches mal mit seinen Späßen aufbauen, wenn wieder mal etwas nicht funktionierte, wie geplant.
Meiner Freundin Berit danke ich für ihr Interesse und ihre Geduld beim Entstehen der Arbeit und auch dafür, daß sie mich über die ganze Zeit nie aufgegeben hat.
Nicht zuletzt möchte ich meinen Eltern dafür danken, daß sie mir durch ihre fortwährende Unterstützung das Studium und diese Arbeit ermöglicht haben. Ihnen habe ich am meisten zu danken.

Inhaltsverzeichnis

Abbildungsverzeichnis

Quellcodeverzeichnis

1. Einleitung

1.1 Einführung

Das Feld der Robotik hat mittlerweile Einzug in die unterschiedlichsten Bereiche des menschlichen Lebens gehalten. Roboter finden heutzutage Verwendung z. B. in der automatisierten Entwicklung und Produktion von Gütern, beim Erforschen fremder Planeten [1], können Landminen entschärfen, gehen Menschen im Haushalt als Service-Roboter zur Hand [2] und helfen in Form von künstlichen Prothesen und Exoskeletten [3] unsere Mobilität zu erhalten. Sie können sogar selbständig Fußball spielen [4, 5]!

Die aktuelle Forschung in der Robotik konzentriert sich aufgrund der sich damit neu eröffnenden Möglichkeiten vermehrt auf mobile Roboter [6]. Nachdem sich mobile Roboter zu Lande bereits etabliert haben, erobern sie seit einiger Zeit nunmehr auch die Luft. Flugfähige Roboter gibt es in verschiedenen Ausführungen: als mit Gas gefüllte Ballons oder Luftschiffe, als unbemannte Flugzeuge oder auch als mit Rotoren versehene Heli- bzw. Multikopter. Die Einsatzmöglichkeiten solcher *UAVs* (engl.: *Unmanned Aerial Vehicle*) sind zahlreich. Neben den häufig erwähnten militärischen Aufgaben sind diese vor allem in der Katastrophen- und Unfallrettung im Zusammenhang mit der Erkundung von unwirtlichem Terrain und Auffinden von Opfern zu finden. Die Fernerkundung z. B. zur Erhebung von Umweltdaten oder Kartographierung [7, 8] oder auch einfache Überwachungsaufgaben sind weitere Einsatzgebiete. Zur Zeit müssen viele Roboter bei der Bewältigung dieser Aufgaben noch von menschlicher Hand gesteuert werden. Die aktuelle Forschung geht jedoch dahin, ihnen mehr Selbständigkeit zu ermöglichen.

Verstärkte Autonomität bei mobilen Robotern bedeutet zumeist eine *selbständige Bewegung*. Allen mobilen Robotern ist jedoch gemein, daß sie als Bedingung hier-

für Kenntnis von ihrer Umgebung und ebenso ihrer ausgeführten Eigenbewegung erhalten. Dafür besitzen sie unterschiedliche Sensoren. Fahrbare Roboter können damit ihre Bewegung über sogenannte „dead-reckoning“-Methoden bestimmen, also der Berechnung der gefahrenen Trajektorie aus der Radstellung und -umdrehung. Kleinere Fehler können hierbei mit Hilfe von optischen Sensoren leicht ermittelt und kompensiert werden.
Für fliegende Roboter stellt sich dies jedoch als ungleich schwieriger dar. Zur Bewegungsmessung stehen ihnen häufig Beschleunigungssensoren zur Verfügung, die jedoch durch äußere Einflüsse wie Wind oder Drift fehlgeleitet werden können. Auch optische Sensoren können diese auftretenden Probleme nicht gänzlich kompensieren.

Der in dieser Arbeit verwendete Quadrokopter *AirRobot AR-100*, ein mit vier horizontal in einer Ebene angeordneten Rotoren, die es ihm ermöglichen, sich omnidirektional im dreidimensionalen Raum zu bewegen, gehört ebenfalls in diese Kategorie. Er ist mit unterschiedlichen Sensoren ausgestattet, wozu neben dem bereits erwähnten Beschleunigungssensor auch ein Barometer zur automatischen Höhenregelung und eine schwenkbare Kamera gehören. Über eine Funkverbindung werden die Sensordaten zur Bodenstation gesendet, von welcher der Roboter im Gegenzug seine Steuerkommandos erhält. Zum Zeitpunkt des Beginns dieser Arbeit war es noch erforderlich, daß die Bodenstation von einem Operator betrieben wurde, der auch die Steuerkommandos mit einer Fernbedienung aktiv erzeugen mußte.

1.2 Aufgabenstellung

Zur Bewältigung der oben beschriebenen Probleme von Flugrobotern und hin zu deren größerer Autonomität sollte als eines der Ziele der vorliegenden Diplomarbeit ein eigenes Softwaresystem zur generellen Ansteuerung entwickelt werden, das die proprietäre Software für die Bodenstation ersetzt. Dieses sollte so modular gehalten werden, daß es leicht um weitere Funktionalitäten erweitert werden kann. Damit sollte der Weg geebnet werden, um zukünftig Steuerkommandos auch allein vom Computer ohne die Hilfe eines Operators generieren zu lassen.

Als weiteres Ziel sollte ausgehend von der Auswertung des optischen Flusses eine vereinfachte Eigenbewegungsschätzung entwickelt werden, welche die aus der Neigung des Roboters resultierenden horizontalen Beschleunigungen berücksichtigt. Die bisher etablierten Verfahren zur Bestimmung der Eigenbewegung aus optischen Flußdaten liefern in der Praxis jedoch nur ungenaue Ergebnisse, so daß eine solche Schätzung mit geeigneten Filterungsmethoden verbessert werden sollte. Zu diesem Zweck sollte ein Modell für das Flugverhalten des Roboters aufgestellt und mit geeigneten Tests auf seine Gültigkeit untersucht werden.

1.3 Gliederung

Diese Arbeit gliedert sich in zwei Teile. Teil I beschreibt die Grundlagen und Voraussetzungen für die in Teil II vorgestellten und diskutierten Eigenentwicklungen.

Sie beschreibt zunächst im Kapitel 2 den verwendeten Flugroboter sowie die zugehörige Computer-Hardware. Kapitel 3 erläutert einige programmier-technische und biologische Grundlagen, wie das *Factory-Pattern* und den *optischen Fluß*, während Kapitel 4 verschiedene verwendete Algorithmen vorstellt.
Im Teil II, Kapitel 5 wird das im Rahmen dieser Arbeit entwickelte Softwaresystem vorgestellt. Kapitel 6 beschreibt das für die Eigenbewegung des Roboters entwickelte mathematische Modell, das in Kapitel 7 genauer untersucht und anhand realer flugexperimenteller Daten verifiziert wird. Den Abschluß bildet das Fazit, welches sich zusammen mit dem Ausblick im Kapitel 8 befindet.

Teil I

Hardware und Grundlagen

2. Verwendete Hardware

2.1 Der Flugroboter

Der in dieser Arbeit verwendete Flugroboter ist ein *AR100* der Firma *AirRobot*. Bei diesem handelt es sich um einen sogenannten Quadrokopter, ein mit vier Rotoren bestücktes Fluggerät (Abb. 2.1).

Ein Quadrokopter weist aufgrund seiner Funktionsweise eine gewisse Ähnlichkeit zu einem Helikopter auf, im Gegensatz zu diesem sind jedoch alle Rotoren waagrecht in einer Ebene angeordnet. Sie sorgen genau wie der große Rotor eines Helikopters für den Auftrieb, gleichen aber zusätzlich auch die Kreiselbewegung des Quadrokopters aus. Während dies bei einem herkömmlichen Helikopter durch den senkrecht am verlängerten Heck angebrachten Rotor geschieht, wird dies beim Quadrokopter hingegen durch die besondere Anordnung der vier Rotoren und ihrer Laufrichtung bewirkt.
Sie sind in einer gemeinsamen Ebene zu einem Quadrat um das Zentrum des Roboters angeordnet. Die jeweils diagonal gegenüberliegenden Rotoren drehen sich in die gleiche Richtung, während sich die beiden dazu um 90° versetzen Rotoren in die entgegengesetzte Richtung drehen. Aufgrund dieser gegenläufigen Rotationen heben sich die induzierten Drehmomente der beiden Rotorpaare bei gleicher Umdrehungszahl gegenseitig auf und der Quadrokopter bleibt bzgl. seiner Orientierung und Lage stabil. Jedoch ist dies ein sehr instabiler Zustand. Denn schon kleinste Abweichungen in den Drehzahlen der Rotoren können dieses Gleichgewicht im wörtlichen Sinne zum Kippen bringen, weshalb eine gute Regelung benötigt wird.

Die systembedingte Instabilität des Quadrokopters macht man sich jedoch auch für die Steuerung zunutze. Indem man die Geschwindigkeiten der einzelnen Rotoren variiert, lassen sich unterschiedliche Flugbewegungen vollführen.

Abbildung 2.1: Der verwendete Quadrokopter *AR100* (Firma *AirRobot*, Amsberg, Deutschland). Unterhalb des zentralen Körpers sitzt die Rotationstrommel mit der Kamera.

Wenn zwischen den beiden Rotorpaaren ein Geschwindigkeitsunterschied besteht, während die beiden Rotoren eines Rotorpaares jedoch mit gleicher Umdrehungszahl rotieren, vollführt der Quadrokopter eine Rotation um die Hochachse seines Zentrums. Er dreht sich dabei in Rotationsrichtung des sich schneller drehenden Rotorpaares.
Analog dazu läßt sich eine Translation des Quadrokopters dadurch hervorrufen, daß ein Geschwindigkeitsunterschied zwischen den einzelnen Rotoren eines in gleiche Richtung drehenden Rotorpaares auftritt. Er kippt dann in Richtung des langsamer drehenden Rotors ab und bewirkt aufgrund dieser Schiefstellung eine Translationsbewegung in diese Richtung. Dies hat also den gleichen Effekt wie die Schiefstellung des Rotors bei einem Helikopter, nur daß hier das gesamte Fluggerät schief gestellt wird.[1]

[1]Für größere Fluggeschwindigkeiten muß sich auch ein Helikopter neigen, da die Schiefstellung seines Rotors nur einen begrenzten Vortrieb erzeugen kann.

Schließlich wird eine Aufwärtsbewegung durch die gleichförmige Drehzahlerhöhung aller vier Rotoren erreicht, eine Abwärtsbewegung durch die gleichförmige Verringerung der Drehgeschwindigkeit aller Rotoren. Eine Kombination dieser verschiedenen Anpassungen der Rotorgeschwindigkeiten zusammen mit einer geeigneten Regelung läßt den Quadrokopter die unterschiedlichsten Flugmanöver durchführen. Weitere detailliertere Informationen dazu finden sich in [9, 10, 11].

Der verwendete *AR100* ist von Werk aus in der Lage, gesetzte Bewegungskommandos in korrekte Rotorgeschwindigkeiten umzusetzen, so daß die gewünschte Bewegung vollführt wird. Bei diesen Bewegungskommandos handelt es sich zum einen um *Nick*- und *Roll*-Winkel (engl.: *Pitch* and *Roll*), um die sich der Roboter neigen soll, was eine Vorwärts-/Rückwärts- bzw. Seitwärtsbewegung zur Folge hat. Zum anderen handelt es sich um die *Gier*-Winkelgeschwindigkeit (engl.: *Yaw*), mit welcher der Roboter um seine Hochachse rotieren, und den Schub bzw. die *Elevations*-Geschwindigkeit (engl.: *Thrust*), mit welcher der Roboter steigen bzw. sinken soll.
Diese Flugkommandos erhält der Roboter entweder von einer Funkfernbedienung, die von einem Piloten am Boden bedient wird, oder von einem Computer, der als Bodenstation dient und ebenfalls von einem Operator bedient werden muß. Der Roboter ist damit nicht vollständig autonom, da er nicht in der Lage ist, ohne Bodenstation bzw. Fernbedienung selbständig zu fliegen.[2]

„On-board“ besitzt der Roboter verschiedene Sensoren. Neben einem GPS-Sensor zur Positionsbestimmung besitzt er auch ein Barometer zur Höhenbestimmung. Diese Barometerdaten werden ebenfalls zur automatischen Höhenstabilisierung verwendet, welche die Drehgeschwindigkeiten der Rotoren so anpaßt, daß der Roboter ohne zusätzliches Schub-Kommando seine Höhe hält. Jedoch ist der Sensor anfällig für (durch den Roboter selbst erzeugte) Druckschwankungen, so daß der Roboter manches mal unvermittelt seine Höhe verändert.
Zusätzlich besitzt er auch einen Inertial-Sensor, der sich aus Beschleunigungsmesser und Gyroskop zusammensetzt. Mit Hilfe dieses Sensors wird analog zur automatischen Höhenstabilisierung eine automatische Positions- und Orientierungsstabilisierung durchgeführt. Allerdings scheint diese in der Praxis nicht immer einwandfrei zu funktionieren. Zumindest geringer Drift, wie er in Innenräumen durch die Turbulenzen der Rotoren auftritt, wird kaum ausgeglichen. Vermutlich sind die verwendeten Beschleunigungssensoren nicht hinreichend empfindlich. Im Freien läßt sich dieser Nachteil jedoch durch die zusätzliche Verwendung des GPS-Sensors beheben, indem die GPS-Positionsdaten zur automatischen Positionsstabilisierung herangezo-

[2] Allerdings kann der Roboter bei Abbruch der Funkverbindung zur Bodenstation selbständig landen. Jedoch verringert er dabei einfach nur schrittweise die Drehzahl der Rotoren, so daß er langsam nach unten sinkt. Weitere Schritte, wie z. B. eine geeignete Landeplatzwahl, kann er jedoch nicht ausführen.

gen werden. In der vorliegenden Diplomarbeit war die Verwendung des GPS-Moduls jedoch nicht vorgesehen, weshalb die automatische Positionsstabilisierung nur eingeschränkt funktioniert. Zudem läßt sie sich ebenso wenig abschalten, wie die automatische Höhenstabilisierung. Letztere ist zwar nicht unwichtig, funktioniert jedoch zumindest in Innenräumen nicht immer zuverlässig.[3]

Neben den bereits erwähnten Sensoren lassen sich auch die Rotationsgeschwindigkeiten der einzelnen Rotoren abfragen. Jedoch wird in dieser Arbeit allein der optische Sensor verwendet, eine digitale Farbkamera, welche in einer Rotationstrommel gelagert unterhalb des *AR100* angebracht ist (Abb. 2.1). Diese Kamera besitzt einen horizontalen Öffnungswinkel von ungefähr ±30 Grad[4] und ist im Stande, 25 Bilder pro Sekunde mit einer nativen Auflösung von 470 TV-Zeilen zu liefern. Zudem besitzt sie einen automatischen Weißabgleich, welcher sich ebenfalls nicht abschalten läßt.[5] Mit der Rotationstrommel ist es möglich, durch ein definiertes Steuersignal der Fernbedienung bzw. Bodenstation, die Ausrichtung der Kamera zu verändern. Jedoch läßt sie sich nur um eine Achse drehen, weshalb die Kamerablickrichtung auf einen Schwenkbereich von vorne nach hinten[6] von ungefähr 130° beschränkt ist. Seitliche Kameraschwenks sind jedoch nicht möglich.

Die Kamerabilder sowie die übrigen Sensordaten werden zusammen per Funk als analoges Videosignal an eine Empfangsantenne am Boden gesendet. Die übrigen Sensordaten sind dabei im Audiokanal codiert und müssen am Boden von einem Splitter erst wieder aus diesem extrahiert werden. Anschließend werden sie über einen Seriellanschluß-zu-USB-Konverter mit der Bodenstation verbunden, während das Kamerabild über einen Videograbber seinen Weg in die Bodenstation nimmt. Die Steuersignale, die von der Bodenstation wieder zum *AR100* gesendet werden, werden ebenfalls über einen Seriellanschluß-zu-USB-Konverter mit einer kleinen Sendeeinheit verbunden, die mit effektiv 9600 Baud funkt.

Der *AR100* ist ein auf geringes Eigengewicht optimiertes Fluggerät, weshalb er trotz seines Durchmessers von ca. 1 m weniger als 1 kg wiegt. Allerdings ist damit die Nutzlast, die es mitführen kann, auf nur ca. 200 g beschränkt. Von diesen 200 g Nutzlast muß zudem noch das Gewicht der Kamera von circa 75 g abgezogen werden, wodurch eine ungefähre zusätzliche Nutzlast von 125 g übrig bleibt.

[3] Wie bereits erwähnt, reagiert der verwendete Höhenmesser sehr empfindlich auf Druckschwankungen, welche in Innenräumen schon allein durch das Zuschlagen einer Tür entstehen können.

[4] Zwischenzeitlich wurde eine andere Linse eingesetzt, die einen horizontalen Öffnungswinkel von ±45 Grad besitzt.

[5] Für die Verwendung im Rahmen dieser Arbeit, speziell zur Bestimmung des optischen Flusses, hat sich dieser automatische Weißabgleich leider häufiger als sehr hinderlich erwiesen. Eine Ersatz-Kamera ohne bzw. mit deaktivierbaren Weißabgleich wäre für die vorgesehene Verwendung zweckmäßiger gewesen.

[6] Beim Blick nach hinten steht das Bild somit auf dem Kopf.

2.2 Das verwendete Computersystem

Als die im vorherigen Abschnitt 2.1 erwähnte Bodenstation kamen abwechselnd zwei unterschiedliche Computersysteme zum Einsatz. Zumeist war es ein *Intel Pentium 4* Desktop-Rechner mit 2,6 GHz Taktfrequenz und Hyperthreading-Technologie, ausgestattet mit 1 GB Arbeitsspeicher. Alternativ dazu wurde auch ein Laptop mit einem *Intel Core2 Duo T7500* Prozessor mit 2,2 GHz Taktfrequenz und 2 GB Arbeitsspeicher verwendet.
Beide Rechner liefen mit einem *Linux*-Betriebssystem (*OpenSUSE* 11) mit Kernel 2.6.25.5 und *KDE 3.5* als graphischer Benutzeroberfläche. Als Programmiersprache kam C++ zum Einsatz, welche mit dem *Gnu Compiler* in Version 4.3.1 übersetzt wurde. Zur Visualisierung der entwickelten graphischen Benutzeroberflächen wurde *Qt* in Version 3.3.8 verwendet.

Als Videograbber kam ein portables Modell der Firma *DIGITUS* zum Einsatz, das über USB 2.0 mit dem jeweiligen Computer verbunden wurde. Mit dem zugehörigen Linux-Treiber war es jedoch nur möglich, 25 Bilder[7] pro Sekunde mit einer festen Auflösung von 720 × 576 Pixel[8] im Bildformat *YUV 4:2:2*[9] bereitzustellen. D. h., die native Auflösung von nur 470 TV-Zeilen wurde vom Videograbber auf 576 Zeilen hochgerechnet. Alternativ konnte im Desktop-System aber auch die interne *Hauppauge* TV-Karte (mit Brooktree Bt878 Chipsatz) als Videograbber verwendet werden, welche die Bilder auch in anderen Auflösungen und Bildformaten liefern kann.
Zur Handsteuerung des Quadrokopters von der Bodenstation aus wurde ein handelsübliches Gamepad der Firma *Logitech*, das Modell *Dual Action*, verwendet.

[7] genauer: 50 Halbbilder, wobei jedoch zwei Halbbilder mit dem Zeilensprungverfahren zu einem gemeinsamen Bild „interlaced“ werden.
[8] 25 Bilder pro Sekunde mit einer Auflösung von 720×576 Pixel entspricht der PAL-Fernsehnorm.
[9] Auch als *YUYV* bekannt.

3. Begriffe, technische und biologische Grundlagen

3.1 Modularität und modulare Programmierung

In der Psycholinguistik wird der Begriff der *Modularität* wie folgt definiert [12]:

> Teilprozesse sind voneinander abgekapselt, sie arbeiten unabhängig voneinander, und sie sind lediglich durch definierte „Schnittstellen“ miteinander verbunden [...].

In der Software-Entwicklung läßt sich dieses Konzept ebenfalls sehr häufig finden [13, 14]. Dort spricht man vom Paradigma der *modularen Programmierung* und meint damit das Aufbrechen eines komplexen und komplizierten Softwaresystems in logische Teilblöcke, sogenannte *Module*, welche meist einfacher zu entwickeln und zu handhaben sind [15, 16].
Mit der Modularität einher geht auch die Möglichkeit einer vereinfachten Austauschbarkeit. So ist es bei geschickter modularer Programmierung möglich, einzelne Module durch andere zu ersetzen, um ein unterschiedliches Verhalten zu erzeugen, ohne dabei das gesamte Softwaresystem komplett überarbeiten zu müssen. Objektorientierte Programmiersprachen, wie die in dieser Arbeit verwendete Programmiersprache C++, erleichtern diese Vorgehensweise.

3.1.1 Vorteile von objektorientierten Programmiersprachen

Objektorientierte Programmiersprachen erlauben das Aufspalten in einzelne Module allein schon aufgrund ihrer Struktur und der Existenz von sogenannten „Klassen“

[15, 17]. Bei Klassen handelt es sich, vereinfacht ausgedrückt, um den Bauplan für einen bestimmten Typ von Objekten. Objekte sind somit Instanzen von Klassen und lassen sich bei der weiteren Programmierung bequem verwenden. Sie können verschiedenen Operationen[1] unterworfen werden, welche den Zustand eines solchen Objektes ggf. verändern.
Jeder Teilaufgabe werden nun eine oder mehrere definierte Klassen zugeordnet. Mit dem Konzept der „Vererbung“ lassen sich ähnliche Aufgabenbereiche dabei durch gemeinsame Basisklassen abdecken. Die speziellen Unterschiede der einzelnen Aufgabenbereiche lassen sich dann wiederum in verschiedenen, von dieser gemeinsamen Basisklasse abgeleiteten, Unterklassen implementieren.

Die Verwendung von Basisklassen hat neben der logischen Aufteilung einen weiteren praktischen Vorteil. Es lassen sich nämlich auch „abstrakte“ Basisklassen implementieren. Bei diesen handelt es sich um normale Basisklassen, welche unterschiedliche Klassenmethoden deklarieren, von denen aber einige (oder auch alle) nicht implementiert werden. Die Implementierung dieser „abstrakten“ Methoden muß dann in den abgeleiteten Unterklassen vorgenommen werden [18].
Der Vorteil besteht nun darin, daß ein Programmteil, welcher Instanzen verschiedener solcher abgeleiteter Unterklassen verwendet, nicht wissen muß, mit welcher der abgeleiteten Klassen er es gerade zu tun hat. Es reicht ihm, zu wissen, daß es eine Instanz *irgendeiner* abgeleiteten Unterklasse dieser Basisklasse ist.

Zur Verdeutlichung dieses Sachverhalts wird an dieser Stelle ein Beispiel vorgestellt. Nimmt man an, die Basisklasse hieße `GeomObj` und stelle ein geometrisches Objekt dar. Weiterhin nehme man an, diese Basisklasse besäße eine Klassenmethode mit der Bezeichnung `draw()`, welche nichts anderes täte, als das geometrische Objekt zu zeichnen. Bildet man nun drei verschiedene Unterklassen `Square`, `Circle` und `Triangle`, die allesamt von dieser gemeinsamen Basisklasse `GeomObj` abgeleitet sind, so muß folglich jede dieser Unterklassen die Klassenmethode `draw()` implementieren. Sinnvollerweise sollte die Implementation genau so erfolgen, daß der Aufruf von `Square.draw()` ein Quadrat, der Aufruf von `Circle.draw()` einen Kreis und der Aufruf von `Triangle.draw()` ein Dreieck zeichnet.
Interessant wird es allerdings erst dann, wenn der Programmteil, der mit solchen geometrischen Objekten umgehen muß, nicht mit den abgeleiteten Unterklassen direkt, sondern nur mit dem Basisklassen-Anteil dieser Objekte interagiert. Dann würde ein Aufruf von `GeomObj.draw()` z. B. ein Quadrat zeichnen, wenn es sich bei dem `GeomObj` in Wirklichkeit um ein `Square`, einen Kreis, wenn es sich um ein `Circle`, oder auch ein Dreieck, wenn es sich in Wirklichkeit um ein `Triangle` handelt. Quell. 3.1 verdeutlicht dieses Beispiel noch einmal.

[1] wie z. B. Addition, Multiplikation für numerische Objekte oder auch einfach nur für normale Status-Abfragen (`getValue`-Funktionen etc.)

An dieser Stelle sei noch erwähnt, daß eine Basisklasse nicht zwingend abstrakt sein muß. Sie kann sehr wohl eine Implementierung für die jeweilige Klassenmethode bereitstellen. Die abgeleiteten Unterklassen können diese dann überschreiben und ihre angepaßte Implementation verwenden.

3.1.2 Das Factory-Pattern

Aus den einzelnen Eigenschaften von objektorientierten Programmiersprachen lassen sich unterschiedliche *Design-Patterns* ableiten, die verschiedene Vorteile bei der Programmierung oder auch bei der Verwendung von fertigen Programmen bieten. Bereits 1995 haben Gamma et al., die sogenannte *Gang of Four*, verschiedene dieser Design-Patterns vorgestellt [19]. Ein Design-Pattern wurde bei der Erstellung dieser Arbeit besonders häufig verwendet, das sogenannte *Factory-Pattern*.

Das *Factory-Pattern* leitet sich direkt aus dem im Abschnitt 3.1.1 beschriebenen Konzept der Vererbung bei Programmiersprachen her. Während, wie im Beispiel in Quell. 3.1 demonstriert, die ausschließliche Verwendung der Basisklassen-Anteile zwar eine gewisse Unabhängigkeit von den einzelnen abgeleiteten Unterklassen erlaubt, ist eine vollständige Unabhängigkeit von diesen standardmäßig jedoch nicht gegeben. Denn zumindest beim Konstruktor-Aufruf, der eine Instanz einer bestimmten Klasse generiert, muß der genaue Klassentyp angegeben werden.
An dieser Stelle setzt das Factory-Pattern an. Der Konstruktor-Aufruf wird in eine (statische) *Creator*-Funktion verschoben, welche Anhand von übergebenen Parametern erkennt, von welcher abgeleiteten Unterklasse die zurückzugebende Instanz sein soll. Der jeweilige Konstruktor wird dann innerhalb dieser Funktion aufgerufen und das generierte Objekt zurückgegeben. Der diese *Creator*-Funktion aufrufende Programmteil behandelt das zurückgelieferte Objekt einfach als Instanz der Basisklasse, unabhängig davon, von welcher Unterklasse es nun wirklich gebildet ist.

Da die übergebenen Parameter nun die Informationen über den gewünschten Objekttyp beinhalten, wird es deutlich, daß man nichts gewonnen hätte, wenn man diese Parameter direkt im Programm „hart“ einprogrammieren würde. Stattdessen sollten diese Parameter zur Laufzeit des Programms vom Benutzer abgefragt werden. Dies kann entweder durch direkte Aufforderung zur Eingabe eines solchen Parameters erfolgen oder als Kommandozeilenparameter beim Starten des Programms direkt übergeben werden. Alternativ kann das Programm diese Parameter auch aus einer Datei einlesen. Letzteres ist übrigens die Vorgehensweise, welche bei dem im Kapitel 5 beschriebenen Software-Framework Verwendung findet.
Quell. 3.2 verdeutlicht das Factory-Pattern noch einmal anhand der Fortführung des Beispiels aus dem vorherigen Abschnitt 3.1.1.

```
1  // Deklaration der abstrakten Basisklasse "GeomObj".
2  class GeomObj {
3    public:
4    virtual void draw() = 0; // Deklaration der Klassenmethode "draw".
5  };
6  // Deklaration der von "GeomObj" abgeleiteten Unterklasse "Square".
7  class Square : public GeomObj {
8    public:
9    void draw() { ... }; // Implementation der Klassenmethode "draw".
10 };
11 // Deklaration der von "GeomObj" abgeleiteten Unterklasse "Circle".
12 class Circle : public GeomObj {
13   public:
14   void draw() { ... }; // Implementation der Klassenmethode "draw".
15 };
16 // Deklaration der von "GeomObj" abgeleiteten Unterklasse "Triangle".
17 class Triangle : public GeomObj {
18   public:
19   void draw() { ... }; // Implementation der Klassenmethode "draw".
20 };
21 // Deklaration einer Funktion, welche einen Zeiger auf ein "GeomObj"
       als Parameter nimmt und die zugehörige "draw"-Methode aufruft.
22 void drawGeomObj(GeomObj* param) { param->draw(); };
23
24
25 // Hauptprogramm-Routine
26 int main()
27 {
28   GeomObj* obj;    // 'obj' ist ein Zeiger auf ein "GeomObj" Objekt.
29
30   obj = new Square();    // 'obj' zeigt nun auf ein "Square" Objekt.
31   drawGeomObj(obj);      // Zeichnet ein Quadrat.
32
33   obj = new Circle();    // 'obj' zeigt nun auf ein "Circle" Objekt.
34   drawGeomObj(obj);      // Zeichnet einen Kreis.
35
36   obj = new Triangle(); // 'obj' zeigt nun auf ein "Triangle" Objekt.
37   drawGeomObj(obj);      // Zeichnet ein Dreieck.
38
39   return 0; // Programm-Ende
40 }
```

Quellcode 3.1: C++-Code zur Verdeutlichung der Vorteile von *Vererbung* in Objektorientierten Programmiersprachen.

```
1  // Implementation der Creator-Funktion, die anhand des übergebenen
       Parameters erkennt, welchen Konstruktor sie aufrufen soll. Sie
       gibt einen Zeiger auf dieses neu erstellte Objekt zurück.
2  GeomObj* createInstance(string param) {
3    if      (param == "Quadrat") return new Square();
4    else if (param == "Kreis")   return new Circle();
5    else if (param == "Dreieck") return new Triangle();
6    else abort(); // Unbekannter Klassentyp, Abbruch!
7  };
8
9
10 // Hauptprogramm-Routine
11 int main()
12 {
13   ...
14   string className; // Variable für Name des gewünschten Klassentyps.
15   cout << "Bitte gewünschten Klassentyp eingeben: " << endl;
16   cin >> className; // Eingabe einlesen, welche den Klassentyp
         beschreibt, und in Variable 'className' speichern.
17
18   GeomObj* obj = createInstance(className); // 'obj' zeigt nun auf
         ein Objekt des gewünschten Klassentyps.
19   drawGeomObj(obj); // Zeichnet die korrekte Form des Objekts.
20
21   return 0; // Programm-Ende
22 }
```

Quellcode 3.2: C++-Code zur Verdeutlichung des *Factory-Patterns* anhand der Fortführung des Beispiels aus Quell. 3.1.

3.2 Der optische Fluß

Die vorliegende Arbeit basiert zu großen Teilen auf der Auswertung des *optischen Flusses*. Dabei handelt es sich vereinfacht ausgedrückt, um die Bewegung, welche auf der Netzhaut des Betrachters[2] entsteht, wenn sich entweder der Betrachter selber oder die betrachtete Umwelt bewegt. Z. B. wandert der Punkt auf der Netzhaut eines Betrachters, auf welchen ein Gegenstand der Umwelt projiziert wird, wenn sich der Betrachtungswinkel zwischen dem Betrachter und dem Gegenstand ändert, sei es durch die Bewegung des Betrachters selst oder die Bewegung des Gegenstands. Dies führt zu einer Bildverschiebung, dem optischen Fluß.

[2] *bzw. allg.:* auf der Bildebene einer Kamera

3.2.1 Das zweidimensionale Bewegungsfeld

Ausgangspunkt für den optischen Fluß ist das dreidimensionale Bewegungsfeld der Umwelt, welches durch die Eigenbewegung des Betrachters bzw. durch unabhängige Bewegung in der Umgebung entsteht. Dieses Bewegungsfeld entspricht somit der Relativbewegung zwischen dem Betrachter (Kameraknotenpunkt) und der Umwelt. Dieses wird im allgemeinen mit $\vec{\mathbf{w}}(\vec{\mathbf{p}}) \in \mathbb{R}^3$ bezeichnet, wobei $\vec{\mathbf{p}} = (p_x, p_y, p_c)^\mathsf{T} \in \mathbb{R}^3$ ein beliebiger Punkt im Raum sein kann.
Durch die Projektion des dreidimensionalen Bewegungsfeldes auf die Bildebene des Betrachters entsteht ein zweidimensionales Bewegungsfeld $\vec{\mathbf{v}}(\vec{\mathbf{p}}') \in \mathbb{R}^2$ (mit $\vec{\mathbf{p}}' = (p'_x, p'_y)^\mathsf{T} \in \mathbb{R}^2$), der optische Fluß. Dabei werden nur die Bewegungsvektoren, welche an sichtbaren Oberflächen der Umwelt ansetzen, auf die Bildebene projiziert. Dieses Projektionsschema ist in Abb. 3.1 dargestellt.

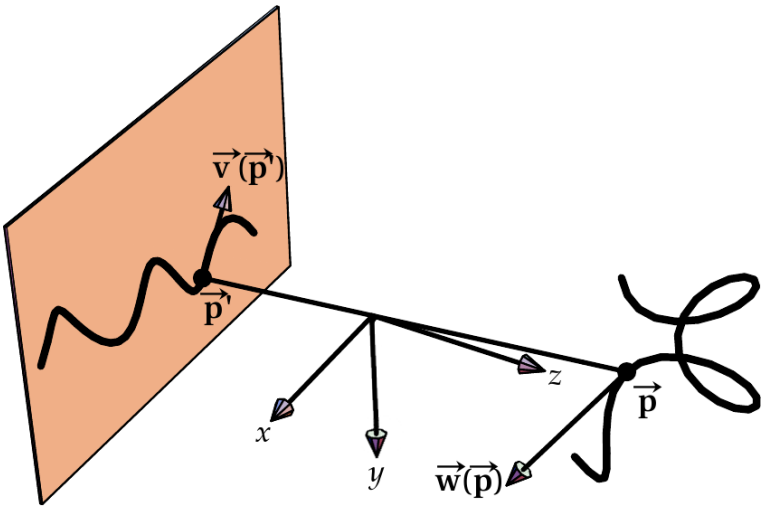

Abbildung 3.1: Schema der Projektion von Geschwindigkeiten. Ein Punkt $\vec{\mathbf{p}}$, welcher sich im Raum entlang der dreidimensionalen Kurve rechts bewegt, wird im Bild auf den Punkt $\vec{\mathbf{p}}'$ abgebildet, welcher sich entlang der ebenen Kurve links bewegt. (Verändert nach [20])

Für die Projektion geht man von einer Lochkamera mit Brennweite der Länge 1 aus, so daß sich die Projektion von $\vec{\mathbf{p}}$ zu $\vec{\mathbf{p}}'$ durch folgende Gleichung ergibt:

$$\vec{\mathbf{p}}' = -\frac{1}{p_z}\begin{pmatrix} p_x \\ p_y \end{pmatrix} \quad . \tag{3.1}$$

Die z-Komponente des Ortspunktes $\vec{\mathbf{p}}$ beschreibt dabei die Entfernung vom Kameraknotenpunkt, während x- und y-Komponenten der Breite und der Höhe bezogen auf diesen entsprechen.

Die Bahnkurve eines dreidimensionalen Punktes $\vec{\mathbf{p}}$ im Raum läßt sich in Abhängigkeit von der Zeit t als vektorwertige Funktion $\vec{\mathbf{p}}\,(t) = (p_x(t), p_y(t), p_z(t))^\mathsf{T}$ relativ zum Kamerakoordinatensystem beschreiben. Die momentane Geschwindigkeit zum Zeitpunkt t entspricht dann genau dem Wert des dreidimensionalen Geschwindigkeitsfeldes $\vec{\mathbf{w}}$:

$$\frac{d}{dt}\vec{\mathbf{p}}\,(t) = \vec{\mathbf{w}}\,(\vec{\mathbf{p}}\,(t)) \quad . \tag{3.2}$$

Die Projektion der Bahnkurve auf die Bildebene ergibt sich zu

$$\vec{\mathbf{p}}'\,(t) = -\frac{1}{p_z(t)} \begin{pmatrix} p_x(t) \\ p_y(t) \end{pmatrix} \quad , \tag{3.3}$$

was einer Parametrisierung nach der Zeit von Gl. 3.1 entspricht. Durch Differenzieren von Gl. 3.3 nach der Zeit t erhält man somit den zugehörigen zweidimensionalen Bewegungsvektor $\vec{\mathbf{v}}$ eines dreidimensionalen Bewegungsvektors $\vec{\mathbf{w}}$, welcher zu einem Punkt $\vec{\mathbf{p}}$ im Raum gehört.

Für die x-Komponente ergibt sich somit mit Hilfe der Quotientenregel:

$$\frac{d}{dt}p'_x\,(t) = \frac{d}{dt}\left(-\frac{p_x(t)}{p_z(t)}\right) = -\frac{1}{p_z(t)}\left(\frac{d}{dt}p_x(t) + p'_x(t)\,\frac{d}{dt}p_z(t)\right) \quad . \tag{3.4}$$

Analog ergibt sich für die y-Komponente:

$$\frac{d}{dt}p'_y\,(t) = -\frac{1}{p_z(t)}\left(\frac{d}{dt}p_y(t) + p'_y(t)\,\frac{d}{dt}p_z(t)\right) \quad . \tag{3.5}$$

Wenn man nun beachtet, daß $\frac{d}{dt}\vec{\mathbf{p}}\,(t) = \vec{\mathbf{w}}\,(\vec{\mathbf{p}}\,(t))$ gilt, so erhält man

$$\left(\frac{d}{dt}p'_x(t),\ \frac{d}{dt}p'_y(t)\right)^\mathsf{T} = -\frac{1}{p_z(t)}\left(\begin{pmatrix} w_x\,(p_x(t)) \\ w_y\,(p_y(t)) \end{pmatrix} + w_z\,(p_z(t)) \begin{pmatrix} p'_x(t) \\ p'_y(t) \end{pmatrix}\right) \quad , \tag{3.6}$$

bzw. einfacher ausgedrückt

$$\vec{\mathbf{v}}\ (\vec{\mathbf{p}}') := -\frac{1}{p_z}\left(\begin{pmatrix} w_x \\ w_y \end{pmatrix} + w_z \begin{pmatrix} p'_x \\ p'_y \end{pmatrix}\right) \quad . \tag{3.7}$$

Dies ist die allgemeine Formel für die Projektion auf das zweidimensionale Bewegungsfeld $\vec{\mathbf{v}}$, den optischen Fluß.

Aus Gl. 3.7 erkennt man, daß, da durch die z-Komponente des zugehörigen Ortspunktes $\vec{\mathbf{p}}$ dividiert wird, eine Bewegung in größerem Abstand zu einer kleineren Bildbewegung führt als dieselbe Bewegung in geringerem Abstand. Analog läßt sich damit auch folgern, daß Bewegungen in der Umwelt immer nur bis auf einen konstanten Faktor genau angegeben werden können. Das bedeutet also, daß ein sich mit einer bestimmten Geschwindigkeit bewegendes Objekt den gleichen optischen Fluß

erzeugt, wie ein n mal schnelleres Objekt, das sich in einer n mal größeren Entfernung bewegt.
Geht das dreidimensionale Bewegungsfeld $\vec{\mathbf{w}}$ auf eine Bewegung des Betrachters zurück, so lassen sich die Distanzen von Objekten immer nur in Vielfachen der in einer bestimmten Zeiteinheit zurückgelegten Strecke angeben. Auf diese Problematik wird in späteren Abschnitten noch näher eingegangen.

3.2.2 Der meßbare Bereich des Bewegungsfeldes

Wenn man in der Praxis vom *optischem Fluß* spricht, so meint man meist nicht den im vorherigen Abschnitt beschriebenen (idealen) optischen Fluß $\vec{\mathbf{v}}$, sondern den meßbaren Bildfluß $\vec{\mathbf{v}}^*$. Auch in den späteren Abschnitten dieser Arbeit ist mit „optischer Fluß“ i.d.R. der meßbare Bildfluß $\vec{\mathbf{v}}^*$ und nicht der reale optische Fluß $\vec{\mathbf{v}}$ gemeint.
Die Unterscheidung ist dennoch wichtig, denn im allgemeinen sind $\vec{\mathbf{v}}$ und $\vec{\mathbf{v}}^*$ nicht identisch. Der Bildfluß hängt nämlich in hohem Maße von den Kontrastverhältnissen sowie dem verwendeten Bewegungsdetektor ab und berücksichtigt dementsprechend nur die meßbaren Anteile des optischen Flusses.

Geringer Kontrast, wie z. B. von einem weißen Blatt Papier, das vor einer genauso weißen Oberfläche verschoben wird, bewirkt keinerlei oder im günstigsten Fall nur einen geringen Bildfluß. Hingegen können sich verändernde Lichtverhältnisse oder einfaches Bildrauschen ebenfalls einen Bildfluß erzeugen und somit eine real nicht existierende Bewegung vortäuschen. (Abb. 3.2 illustriert diese Problematik.)

Die daraus entstehenden Probleme sind nicht unbedeutend. Bei der Erstellung dieser Arbeit haben sich diese Effekte als schwerwiegende Probleme dargestellt, welche in späteren Abschnitten noch näher beleuchtet werden.

3.2.3 Der optische Fluß bei Eigenbewegungen des Betrachters

Die dreidimensionale Bewegung eines Betrachters im Raum hat sechs Freiheitsgrade. Davon bestimmen drei die jeweilige Position und die übrigen drei die Orientierung bzw. Blickrichtung im Raum. Eine Bewegung läßt sich somit immer vollständig in Translation und Rotation zerlegen.
Ist $\vec{\mathbf{u}}$ die Translationsrichtung[3] des Betrachters und v die zugehörige Translationsgeschwindigkeit, $\vec{\mathbf{r}}$ die Rotationsachse im Raum[4] und ω die zugehörige Winkelgeschwindigkeit, mit der sich der Betrachter um diese Achse dreht, dann ergibt sich das dreidimensionale Bewegungsfeld $\vec{\mathbf{w}}$ für statische Umgebungen zu:

$$\vec{\mathbf{w}}\,(\vec{\mathbf{p}}) = -v\vec{\mathbf{u}} - \omega\vec{\mathbf{r}} \times \vec{\mathbf{p}} \ . \tag{3.8}$$

[3] ein Einheitsvektor
[4] ebenfalls ein Einheitsvektor

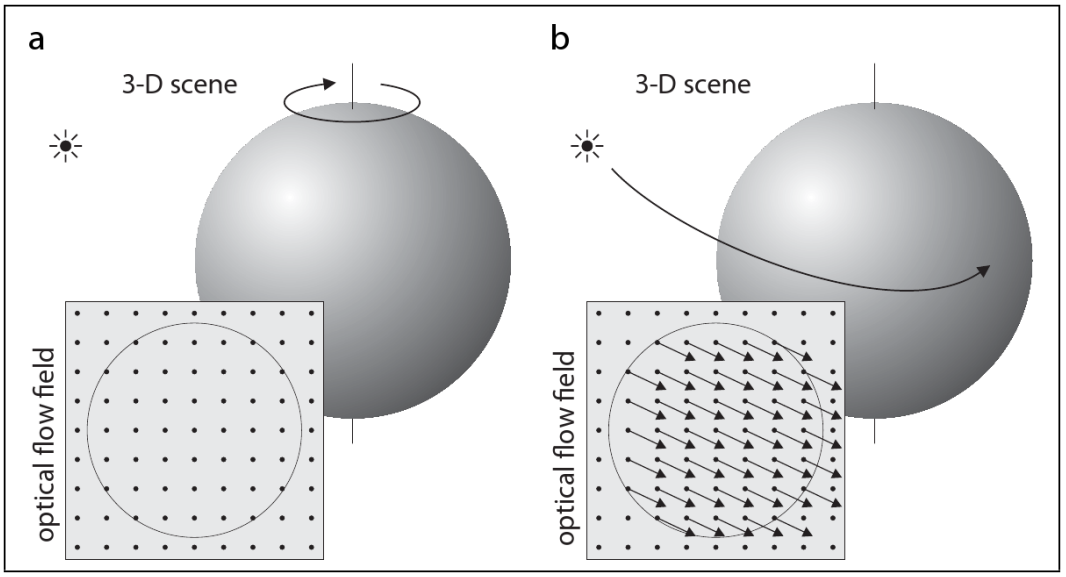

Abbildung 3.2: Problematik des Messens von optischem Fluß. In **(a)** führt eine rotierende ideale (und damit kontrastarme) Kugel bei statischer Lichtquelle fälschlicherweiße zu keinerlei optischem Fluß, wohingegen in **(b)** dieselbe, dieses Mal stillstehende, Kugel bei sich bewegender Lichtquelle irrtümlich zu optischem Fluß führt. (Übernommen aus [21])

Man sieht, daß der Translationsanteil des dreidimensionalen Bewegungsfeldes unabhängig von der Position des jeweiligen Punktes $\vec{\mathbf{p}}$ in der Umwelt ist, dieser also für alle Punkte konstant ist. Hingegen ist der Rotationsteil des Bewegungsfeldes charakterisiert durch das Kreuzprodukt der Rotationsachse mit dem jeweiligen Raumpunkt $\vec{\mathbf{p}}$ und damit abhängig von der Lage des Raumpunktes und seiner Entfernung zum Betrachter.

Die aus diesen beiden Bewegungsarten resultierenden optischen Flußfelder werden in den folgenden Abschnitten genauer betrachtet. (Für weitere Betrachtungen sei auf [20] verwiesen.)

3.2.3.1 Optischer Fluß bei Rotation

Wenn der Betrachter um eine Achse $\vec{\mathbf{r}}$ durch den Kameraknotenpunkt mit der Winkelgeschwindigkeit ω rotiert, so ergibt sich die Bewegung eines Punktes $\vec{\mathbf{p}}$ in der Umwelt laut Gl. 3.8 zu:

$$\vec{\mathbf{w}}\left(\vec{\mathbf{p}}\right) = -\omega\vec{\mathbf{r}} \times \vec{\mathbf{p}} = -\omega \begin{pmatrix} r_y p_z - r_z p_y \\ r_z p_x - r_x p_z \\ r_x p_y - r_y p_x \end{pmatrix} \quad . \tag{3.9}$$

Wird dieser Vektor nach Gl. 3.7 auf die Bildebene projiziert, so ergibt sich folgender optische Fluß:

$$\vec{\mathbf{v}}\left(\vec{\mathbf{p}}'\right) = \frac{\omega}{p_z}\left(\begin{pmatrix} r_y p_z - r_z p_y \\ r_z p_x - r_x p_z \end{pmatrix} + \left(r_x p_y - r_y p_x\right)\begin{pmatrix} p'_x \\ p'_y \end{pmatrix}\right) \quad . \tag{3.10}$$

Man erkennt leicht, daß die Raumkoordinaten (p_x, p_y, p_z) hier nur als Quotienten $\frac{p_x}{p_z}$ und $\frac{p_y}{p_z}$ eingehen. Das heißt aber wiederum, daß alle Raumpunkte, die auf einem gemeinsamen Sehstrahl des Betrachters liegen, den gleichen optischen Fluß erzeugen. Somit läßt sich aus einfacher Rotation keine Tiefeninformationen über die Umwelt entnehmen. Daher kann man nun von den dreidimensionalen Koordinaten unter Verwendung von Gl. 3.1 vollständig zu Bildkoordinaten (p'_x, p'_y) übergehen und Gl. 3.10 läßt sich umformen zu:

$$\vec{\mathbf{v}}\left(\vec{\mathbf{p}}'\right) = \omega\left(\begin{pmatrix} r_y + r_z p'_y \\ -r_z p'_x - r_x \end{pmatrix} + \left(-r_x p'_y + r_y p'_x\right)\begin{pmatrix} p'_x \\ p'_y \end{pmatrix}\right) \tag{3.11}$$

$$= \omega\left(r_x\begin{pmatrix} p'_x p'_y \\ 1 + p'^2_y \end{pmatrix} + r_y\begin{pmatrix} 1 + p'^2_x \\ p'_x p'_y \end{pmatrix} + r_z\begin{pmatrix} p'_y \\ p'_x \end{pmatrix}\right) \quad . \tag{3.12}$$

Zwei Spezialfälle von Rotationen zusammen mit ihren resultierenden Flußfeldern sind in Abb. 3.3 dargestellt: Zum einen (a) das resultierende Flußfeld bei Rotation um die Blickachse, und zum anderen (b) das Flußfeld, welches aus einer Rotation um die Hochachse des Betrachters entsteht.

3.2.3.2 Optischer Fluß bei Translation

Analog zum Fall der reinen Rotation berechnet sich das Bewegungsfeld bei reiner Translation mit Geschwindigkeit v entlang der Achse $\vec{\mathbf{u}}$ nach Gl. 3.8 zu:

$$\vec{\mathbf{w}}\left(\vec{\mathbf{p}}\right) = -v\vec{\mathbf{u}} = -v\begin{pmatrix} u_x \\ u_y \\ u_z \end{pmatrix} \quad . \tag{3.13}$$

Der optische Fluß ergibt sich nach Gl. 3.7 zu:

$$\vec{\mathbf{v}}\left(\vec{\mathbf{p}}'\right) = \frac{\omega}{p_z}\left(\begin{pmatrix} u_x \\ u_y \end{pmatrix} + u_z\begin{pmatrix} p'_x \\ p'_y \end{pmatrix}\right) \quad . \tag{3.14}$$

Anders als bei reiner Rotation sieht man hier, daß das Translationsflußfeld sehr wohl vom jeweiligen Punkt $\vec{\mathbf{p}}$ in der Umwelt abhängt, da die einzelnen Koordinaten nicht nur als Quotienten eingehen. Da der Kehrwert der z-Komponente des Raumpunktes $\vec{\mathbf{p}}$ in die Projektion eingeht, ist der resultierende Fluß um so größer, je dichter sich der Raumpunkt am Betrachter befindet.

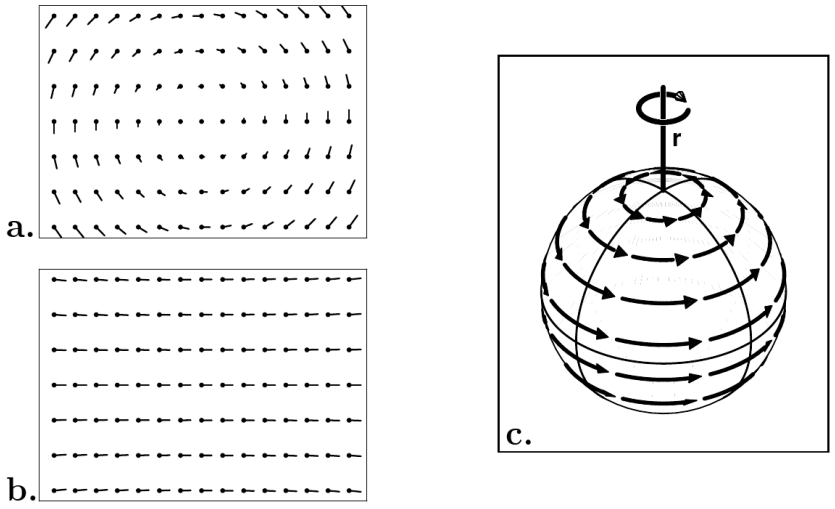

Abbildung 3.3: Flußfelder, welche durch reine Rotation um eine Achse $\vec{\mathbf{r}}$ durch den Kameraknotenpunkt entstehen. Die einzelnen Flüsse verlaufen in Richtung der Punkte. **(a)** Ausschnitt des resultierenden Flußfeldes bei Rotation des Betrachters (entgegen dem Uhrzeigersinn) um seine Blickachse. **(b)** Ausschnitt des resultierenden Flußfeldes bei Rotation (im Uhrzeigersinn) um die Hochachse des Betrachters. **(c)** Projektion der Bewegung auf eine Kugeloberfläche. Es bilden sich Breitenkreise, anhand derer sich der optische Fluß für jede beliebige Blickrichtung berechnen läßt. (Verändert nach [20])

Somit lassen sich also Tiefeninformationen nur aus dem Translationsanteil des optischen Flusses entnehmen. Im Alltag macht sich das dadurch bemerkbar, daß bei Translationen nahe Objekte schneller am Betrachter vorbeiziehen, als weiter entfernte Objekte.

Eine zusätzliche Besonderheit bei Translationen liegt darin, daß es für $u_z \neq 0$ stets einen Punkt im Bild gibt, in dem das projizierte Vektorfeld auftaucht bzw. verschwindet. Mathematisch läßt sich das wie folgt zeigen.

$$\vec{\mathbf{v}}\ (\vec{\mathbf{p}}'_0) = 0 \quad \Rightarrow \quad u_x + u_z\, p'_{0x} = 0\ ,\ u_y + u_z\, p'_{0y} = 0 \tag{3.15}$$

$$\Rightarrow \quad \begin{pmatrix} p'_{0x} \\ p'_{0y} \end{pmatrix} = -\frac{1}{u_z} \begin{pmatrix} u_x \\ u_y \end{pmatrix} =: \vec{\mathbf{F}}\ . \tag{3.16}$$

Man bezeichnet $\vec{\mathbf{F}}$ als *Expansionspunkt* (engl.: *focus of expansion*, *f.o.e.*) bzw. bei negativem Vorzeichen von u_z als Kontraktionspunkt (engl.: *focus of contraction*, *f.o.c.*). $\vec{\mathbf{F}}$ ist der Fluchtpunkt, der zur durch $\vec{\mathbf{u}}$ festgelegten Raumrichtung gehört, also der Punkt auf der Bildebene, auf den man sich zu- (*f.o.e.*) bzw. von dem man sich wegbewegt (*f.o.c.*). Demzufolge laufen alle Flußvektoren vom Expansionpunkt weg bzw. auf den Kontraktionspunkt zu, sind also Linien durch diese. Abb. 3.4 zeigt einige

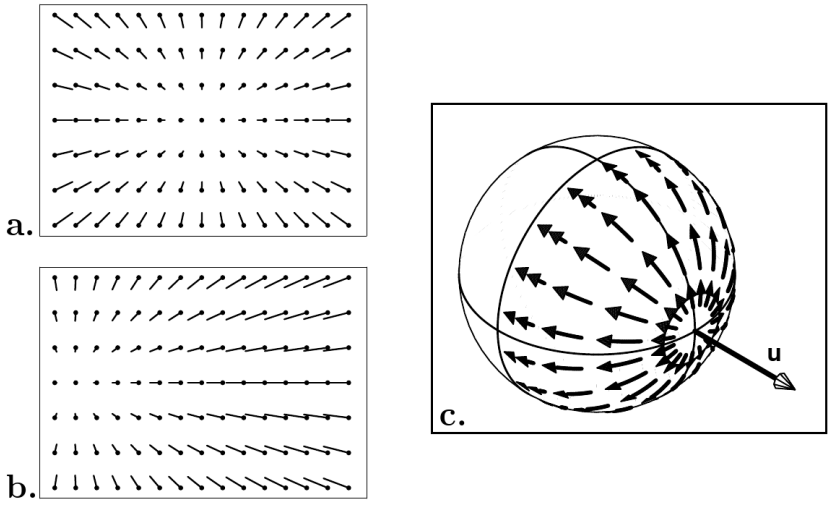

Abbildung 3.4: Flußfelder, welche durch reine Translation entlang der Achse $\vec{\mathbf{u}}$ entstehen. Ausschnitt des resultierenden Flußfeldes bei Translation des Betrachters senkrecht auf eine Wand **(a)** in Blickrichtung und **(b)** schräg zur Blickrichtung. In beiden Fällen zeigt sich der Expansionspunkt deutlich. **(c)** Projektion der Bewegung auf eine Kugeloberfläche. Die Flußlinien bilden Großkreise durch den Expansions- und Kontraktionspunkt. (Aus [20])

Translationsflußfelder mit Expansionspunkt. Die Länge der Flußvektoren hängt vom realen Abstand der abgebildeten Raumpunkte zum Betrachter ab, zusätzlich jedoch auch vom Abstand in der Bildebene zum Expansions- bzw. Kontraktionspunkt.

4. Verwendete Algorithmen

Im folgenden werden einige bei der Erstellung dieser Arbeit verwendete Algorithmen beschrieben. Das Hauptaugenmerk richtet sich hierbei auf die Beschreibung des *Erweiterten Kalman-Filters* in Abschnitt 4.3. Dieser wird nämlich später im Kapitel 6.1 implementiert.

4.1 Algorithmen zur Berechnung des optischen Flusses

Zur Berechnung des optischen Flusses gibt es verschiedene mehr oder weniger effektive und effiziente Verfahren. Dabei erfreut sich die Verwendung von differentiellen Methoden zur Berechnung großer Beliebtheit. Diese lassen sich grundsätzlich in zwei Klassen unterteilen: in lokale und globale Methoden. Lokale Methoden haben den Vorteil, daß sie im allgemeinen gegenüber Rauschen robuster sind, wohingegen globale Methoden ein dichteres Flußfeld liefern. Ein bekannter Vertreter für solche globalen Methoden ist z. B. der von Horn und Schunk 1981 [22] vorgestellte Algorithmus. Zu den globalen Methoden gehört ebenfalls das von Lucas und Kanade 1981 [23] vorgestellte Verfahren zur optischen Flußbestimmung. Dieses wird zur optischen Flußberechnung im Rahmen dieser Arbeit verwendet und deshalb im folgenden näher erläutert. Darüber hinaus gibt es aber auch weitere Verfahren, welche versuchen, globale und lokale Methoden zu kombinieren [24], die in dieser Arbeit jedoch nicht eingesetzt wurden.

4.1.1 Lucas-Kanade-Algorithmus

Eine Grundannahme des von Lucas und Kanade beschriebenen Verfahrens zur optischen Flußbestimmung geht davon aus, daß Helligkeitsänderungen im Bild nur auf

Verschiebungen des ursprünglichen Bildes zurückzuführen sind und somit auf lokale Bereiche beschränkt bleiben. Demzufolge nutzt das Verfahren den räumlichen Helligkeitsgradienten, um die Suche nach dem optischen Fluß in die richtige Richtung zu leiten und damit zu verkürzen.

Im folgenden seien A und B zwei 2D Grauwertbilder[1], wobei mit $A(\vec{x}) = A(x, y)$ und $B(\vec{\mathbf{x}}) = B(x, y)$ die Helligkeitswerte an der Pixelposition $\vec{\mathbf{x}} = (x, y)^\mathsf{T}$ beschrieben werden. A bezeichnet dabei das Ausgangsbild und B das aus einer Verschiebung[2] hervorgehende Bild.
Das Verfahren versucht nun für einen Bildpunkt $\vec{\mathbf{u}} = (u_x, u_y)$ den optischen Fluß, also den lokalen Geschwindigkeitsvektor $\vec{\mathbf{d}} = (d_x, d_y)$, zu bestimmen, so daß gilt: $A(\vec{\mathbf{u}}) \approx B(\vec{\mathbf{u}} + \vec{\mathbf{d}})$.[3] Dies wird erreicht, indem der Algorithmus die Summe der lokal gewichteten Fehlerquadrate in der Nachbarschaft U des Bildpunktes $\vec{\mathbf{u}}$ minimiert, innerhalb derer von einem gleichförmigen Fluß ausgegangen wird. Die zugehörige Fehlerfunktion ε lautet damit:

$$\varepsilon(\vec{\mathbf{d}}) = \sum_{x=u_x-w_x}^{u_x+w_x} \sum_{y=u_y-w_y}^{u_y+w_y} \omega(x-w_x\,,\, y-w_y) \cdot (A(x,y) - B(x+d_x\,,\, y+d_y))^2 \quad . \tag{4.1}$$

$\omega(x, y)$ beschreibt dabei eine Gewichtungsfunktion, die über die $(2w_x{+}1) \times (2w_y{+}1)$ Pixel große, um $\vec{\mathbf{u}}$ vorhandene Nachbarschaftsumgebung U gelegt wird. Im einfachsten Fall gewichtet sie alle Nachbarn gleichermäßig, jedoch werden meist andere Filterfunktionen für ω verwendet, wie zum Beispiel ein Gauß-Filter. Mit diesem läßt sich z. B. hochfrequentes Bildrauschen unterdrücken.[4] Da die Gewichtung mit ω für alle Bildpunkte $\vec{\mathbf{u}}$ der beiden Bilder A und B meist auch im Vorfeld vorgenommen werden kann, wird sie im folgenden weggelassen, womit sich Gl. 4.1 vereinfacht zu:

$$\varepsilon(\vec{\mathbf{d}}) = \varepsilon(d_x, d_y) = \sum_{x=u_x-w_x}^{u_x+w_x} \sum_{y=u_y-w_y}^{u_y+w_y} (A(x,y) - B(x+d_x\,,\, y+d_y))^2 \quad . \tag{4.2}$$

Im optimalen Fall wird die erste Ableitung von ε nach $\vec{\mathbf{d}}$ zu Null:

$$0 = \frac{\partial\, \varepsilon(\vec{\mathbf{d}})}{\partial\, \vec{\mathbf{d}}} = -2 \cdot \sum_{x=u_x-w_x}^{u_x+w_x} \sum_{y=u_y-w_y}^{u_y+w_y} (A(x,y) - B(x+d_x\,,\, y+d_y)) \cdot \left(\frac{\partial B}{\partial x}, \frac{\partial B}{\partial y}\right) \quad . \tag{4.3}$$

[1] Da nur die Helligkeitsänderung Verwendung findet, kann bei Farbbildern auf den Farbanteil verzichtet werden.

[2] Diese muß jedoch nicht notwendigerweise durch eine reine Translation begründet, sondern kann auch von einer Scherung hervorgerufen worden sein, die auf eine Rotation der im Bild abgebildeten 3D-Umwelt zurückgeht.

[3] Im besten Fall sind $A(\vec{\mathbf{u}})$ und $B(\vec{\mathbf{u}} + \vec{\mathbf{d}})$ identisch, jedoch kann aufgrund des sogenannten *Apertur Problems* nicht davon ausgegangen werden, daß damit auch wirklich die reale Verschiebung angegeben ist. Auf das Apertur Problem wird hier jedoch nicht weiter eingegangen.

[4] Allerdings auch kleinste Bewegungen, sofern sie nicht sowieso von Rauschen überlagert sind.

Da in erster Näherung von kleinen Bildverschiebungen ausgegangen wird, kann die Helligkeit in Bild B lokal durch eine Linearisierung an der Stelle $(x+d_x\,,\,y+d_y)$ mit Hilfe der Taylor-Entwicklung erster Ordnung an dieser Stelle angenähert werden:

$$B\,(x+d_x\,,\,y+d_y) \approx B\,(x,y) + \left(\tfrac{\partial B}{\partial x}, \tfrac{\partial B}{\partial x}\right) \cdot \vec{\mathbf{d}}\ . \tag{4.4}$$

Einsetzen in Gl. 4.3 liefert somit:

$$0 = \frac{\partial\,\varepsilon\,(\vec{\mathbf{d}})}{\partial\,\vec{\mathbf{d}}} \approx -2 \cdot \sum_{x=u_x-w_x}^{u_x+w_x} \sum_{y=u_y-w_y}^{u_y+w_y} \left(A\,(x,y) - B\,(x,y) - \left(\tfrac{\partial B}{\partial x}\,,\,\tfrac{\partial B}{\partial y}\right)\vec{\mathbf{d}}\right) \cdot \left(\tfrac{\partial B}{\partial x}\,,\,\tfrac{\partial B}{\partial y}\right)\ . \tag{4.5}$$

Die Differenz $A\,(x,y) - B\,(x,y)$ beschreibt genau besehen die zeitliche Ableitung an der Stelle (x,y), kann somit also vereinfacht ausgedrückt werden als

$$\delta I\,(x,y) \doteq A\,(x,y) - B\,(x,y)\ . \tag{4.6}$$

Die Matrix $\left(\tfrac{\partial B}{\partial x}\,,\,\tfrac{\partial B}{\partial y}\right)$ wiederum ist nichts anderes als der räumliche Helligkeitsgradient und läßt sich wie folgt verkürzt darstellen:

$$\nabla I = \begin{pmatrix} I_x \\ I_y \end{pmatrix} \doteq \left(\tfrac{\partial B}{\partial x}\,,\,\tfrac{\partial B}{\partial y}\right)^{\mathsf{T}}\ . \tag{4.7}$$

Damit läßt sich Gl. 4.5 folgendermaßen umformulieren:

$$0 = \frac{1}{2}\frac{\partial\,\varepsilon\,(\vec{\mathbf{d}})}{\partial\,\vec{\mathbf{d}}} \approx \sum_{x=u_x-w_x}^{u_x+w_x} \sum_{y=u_y-w_y}^{u_y+w_y} \left(\nabla I^{\mathsf{T}} \cdot \vec{\mathbf{d}} - \delta I\right) \cdot \nabla I^{\mathsf{T}}\ . \tag{4.8}$$

Ausklammern und transponieren liefert:

$$0 = \frac{1}{2}\left(\frac{\partial\,\varepsilon\,(\vec{\mathbf{d}})}{\partial\,\vec{\mathbf{d}}}\right)^{\mathsf{T}} \approx \sum_{x=u_x-w_x}^{u_x+w_x} \sum_{y=u_y-w_y}^{u_y+w_y} \left(\begin{pmatrix} I_x^2 & I_xI_y \\ I_xI_y & I_y^2 \end{pmatrix} \cdot \vec{\mathbf{d}} - \begin{pmatrix} \delta I \cdot I_x \\ \delta I \cdot I_y \end{pmatrix}\right)\ . \tag{4.9}$$

Definiert man nun noch

$$\mathbf{G} \doteq \sum_{x=u_x-w_x}^{u_x+w_x} \sum_{y=u_y-w_y}^{u_y+w_y} \begin{pmatrix} I_x^2 & I_xI_y \\ I_xI_y & I_y^2 \end{pmatrix} \tag{4.10}$$

und

$$\vec{\mathbf{b}} \doteq \sum_{x=u_x-w_x}^{u_x+w_x} \sum_{y=u_y-w_y}^{u_y+w_y} \begin{pmatrix} \delta I \cdot I_x \\ \delta I \cdot I_y \end{pmatrix}\ , \tag{4.11}$$

so kann man Gl. 4.9 schreiben als:

$$0 = \frac{1}{2}\left(\frac{\partial\,\varepsilon\,(\vec{\mathbf{d}})}{\partial\,\vec{\mathbf{d}}}\right)^{\mathsf{T}} \approx \mathbf{G}\,\vec{\mathbf{d}} - \vec{\mathbf{b}}\ . \tag{4.12}$$

Für die Berechnung des optischen Flußvektors ergibt sich somit:

$$\vec{\mathbf{d}} \approx \mathbf{G}^{-1}\,\vec{\mathbf{b}} \ . \tag{4.13}$$

Ein optischer Flußvektor $\vec{\mathbf{d}}$ läßt sich auf diese Weise also nur berechnen, wenn die Matrix $\mathbf{G}$ invertierbar ist. Dies ist gleichbedeutend damit, daß innerhalb der Umgebung U um den betrachteten Bildpunkt herum Gradienten-Information in beiden Richtungen x und y vorliegen muß, was wiederum eine zuvor durchgeführte gute Merkmalsauswahl (engl.: *Feature Selection*) des zu untersuchenden Bildes voraussetzt.

Die Voraussetzung für die Linearisierung mit Hilfe der Taylor-Entwicklung besagt, daß kleine Bildverschiebungen vorliegen müssen. Für Bildverschiebungen, welche größer als die betrachtete Nachbarschaftumgebung U sind, läßt sich damit jedoch kein korrektes Ergebnis finden. Dies läßt sich entweder durch die Vergrößerung der verwendeten Umgebung U beheben, womit sich die Berechnungsdauer jedoch stark erhöhen würde, oder besser durch ein iteratives *Newton-Raphson* Verfahren. Bei diesem werden die oben beschrieben Berechnungsschritte mehrfach hintereinander ausgeführt, wobei das Ergebnis aus dem vorangegangenen Iterationsschritt als Startschätzung für den nachfolgenden Iterationsschritt gewählt wird. Erkauft wird diese höhere Genauigkeit jedoch ebenfalls mit einer höheren Rechenzeit.

4.1.2 Pyramidaler Lucas-Kanade-Algorithmus

Um die durch das Newton-Raphson-Verfahren erhöhte Berechnungsdauer des *Lucas-Kanade-Algorithmus* zu vermeiden, aber dennoch auch größere Bildverschiebungen annähernd korrekt bestimmen zu können, bietet sich die Verwendung einer *grob-zu-fein* Strategie an.
Eine solche Strategie stellt der *Pyramidale Lucas-Kanade-Algorithmus* [25] dar. Dieser erstellt vorab für jedes der beiden Ausgangsbilder A und B verschieden stark gerasterte Versionen und führt die (iterative) Berechnung des optischen Flusses nach *Lucas-Kanade* beginnend bei den am gröbsten gerasterten Bildern bis zur feinsten Stufe durch. Die auf jeder Stufe erhaltene (grobe) Schätzung des optischen Flusses wird dabei als Eingangschätzung für die nächst feiner Stufe genommen. Dies wird so oft durchgeführt, bis auf der feinsten Stufe, also den originalen Bildern, die endgültige optische Flußschätzung bestimmt wurde.
Die Laufzeit im Gegensatz zum normalen iterativen *Lucas-Kanade-Algorithmus* wird auf diese Weise stark verringert.

4.2 Algorithmen zur Eigenbewegungsbestimmung aus dem optischen Fluß

Zur Berechnung der Eigenbewegung aus optischen Flußdaten gibt es verschiedene Algorithmen [26, 27, 28, 29, 30], die bisher in der Literatur nebeneinander zum Einsatz kommen. In dieser Arbeit wurde der im folgenden beschriebene *Kanatani*-Algorithmus zur Eigenbewegungsbestimmung verwendet.

Kanatani-Algorithmus

Im von Kanatani 1993 [30] vorgestellten Verfahren zur Eigenbewegungsbestimmung aus optischen Flußdaten wird anhand eines Paares $(\mathbf{K}, \boldsymbol{\nu})$ aus sogenannten *essentiellen Parametern* (engl.: *essential parameters*) die Epipolargleichung

$$\dot{\mathbf{m}}^* \cdot \boldsymbol{\nu} + \mathbf{m} \cdot \mathbf{K}\,\mathbf{m} = 0 \tag{4.14}$$

gelöst, welche das Problem der Eigenbewegungsbestimmung beschreibt. $\mathbf{m}$ ist dabei ein zu untersuchender Bildpunkt, während $\dot{\mathbf{m}}^*$ den dazugehörigen optischen Fluß in einer speziellen Form, als orthogonal auf dem Vektor $\mathbf{m}$ stehenden *verdrehten Fluß* (engl.: *twisted flow*), darstellt. Die für die detaillierte Darstellung des Algorithmus' benötigten Ausführungen sind sehr umfangreich, weshalb auf eine detaillierte Darstellung an dieser Stelle verzichtet wird. Zudem wurde der *Kanatani*-Algorithmus im Rahmen einer am Lehrstuhl Kognitive Neurowissenschaften der Universität Tübingen derzeit durchgeführten Doktorarbeit [31] bereits implementiert und stand für die Verwendung in der vorliegenden Arbeit zur Verfügung. Im folgenden wird deshalb nur das diesem Algorithmus zugrunde liegende Konzept in Grundzügen vorgestellt.

Der grundsätzliche Ansatz zum Lösen von Gl. 4.14 beruht auf einer Fehlerminimierung mit dem Verfahren der kleinsten Fehlerquadrate. Die sich damit ergebende Schätzung für die Translation ist jedoch mit einem systematischen Fehler behaftet. Dieser wurde von Kanatani mit einem einfachen Gauß'schen Rauschmodell analysiert, womit er in der Lage war, ein Verfahren zu entwickeln, welches diesen Fehler automatisch kompensiert. Darin bedient er sich der Tatsache, daß der normalisierte Eigenvektor des kleinsten Eigenwertes einer hier nicht näher beschriebenen Matrix $\mathbf{A}$ ein guter Schätzer ohne systematischen Fehler für die Translationsgeschwindigkeit $\boldsymbol{\nu}$ ist. Dieses *Renormalisierungs*-Verfahren (engl.: *renormalization method*) ist Bestandteil seines Algorithmus' zur Eigenbewegungsbestimmung.

Wie Kanatani in seiner dazu veröffentlichten Arbeit anmerkt, hat dieses Verfahren jedoch nur theoretischen Wert, sofern nicht relativ präzise optische Flußdaten vorliegen.

4.3 Kalman-Filter

Ein Kalman-Filter ist ein stochastischer Zustandsschätzer für dynamische Systeme, der von seinem Namensgeber Rudolph Kálmán im Jahre 1960 entwickelt wurde [32]. In seiner ursprünglichen Form wurde er für zeitdiskrete lineare Systeme entwickelt. Sein Zweck liegt in der Schätzung von Zuständen bzw. Parametern des Systems aufgrund von teils redundanten Messungen, die wiederum von Rauschen überlagert sein können. Der Kalman-Filter verfolgt dabei den Ansatz der Minimierung des mittleren quadratischen Fehlers.

Das grundsätzliche Vorgehen des Kalman-Filters besteht darin, daß er in einem ersten Schritt, dem *Zeit-Update* Schritt, eine Abschätzung für den Zustand macht, welche in einem zweiten Schritt, dem *Messung-Update* Schritt, unter Zuhilfenahme einer oder mehrerer verrauschter Messungen verbessert wird. (Siehe Abb. 4.1.)

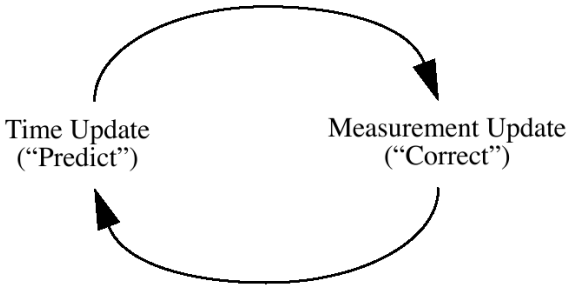

Abbildung 4.1: Der Kreislauf eines Kalman-Filters. Zuerst wird im Zeit-Update Schritt der betrachtete Zustand in der Zeit vorwärts prädiziert und somit eine neue Abschätzung getroffen. Danach wird im Messung-Update Schritt diese Abschätzung noch mit einer (verrauschten) Messung korrigiert. Der nächste Zeitschritt beginnt dann wieder beim Zeit-Update Schritt. (Entnommen aus [33].)

Im folgenden wird zunächst der ursprüngliche *Diskrete Kalman-Filter* (DKF) beschrieben, gefolgt vom davon abgeleiteten *Erweiterten Kalman-Filter* (EKF), der auch auf nicht-lineare Systeme anwendbar ist.

4.3.1 Diskreter Kalman-Filter

4.3.1.1 Der zu schätzende lineare Prozeß

Der DKF versucht einen Zustand $x \in \mathbb{R}^n$ eines diskret-Zeit kontrollierten Prozesses zu schätzen, der durch die folgende lineare stochastische Gleichung beschrieben wird:

$$x_k = \mathbf{A}\, x_{k-1} + \mathbf{B}\, u_{k-1} + w_{k-1} \ . \tag{4.15}$$

Eine zugehörige Messung $z \in \mathbb{R}^m$ ist wie folgt definiert:

$$z_k = \mathbf{H}\, x_k + v_k \ . \tag{4.16}$$

Die Variablen w_{k-1} und v_k sind die sogenannten Prozeß- bzw. Meßrauschen. Sie werden als voneinander *unabhängig*, *weiß*[5] und *standard-normal* verteilt angenommen:

$$p\,(w) \sim N\,(0, \mathbf{Q}) \ , \tag{4.17}$$

$$p\,(v) \ \sim N\,(0, \mathbf{R}) \ . \tag{4.18}$$

Die Kovarianzmatrizen des Prozeßrauschens $\mathbf{Q}$ und des Meßrauschens $\mathbf{R}$ können in der Praxis zu jedem Zeitpunkt verschieden sein, doch werden sie hier zur Vereinfachung als konstant angenommen.

Die $n \times n$ Matrix $\mathbf{A}$ aus Gl. 4.15 setzt den vorherigen Zustand aus Zeitschritt k–1 zum aktuellen Zustand im Zeitschritt k in Beziehung, die $n \times l$ Matrix $\mathbf{B}$ hingegen ordnet den optionalen Kontroll-Input $u \in \mathbb{R}^l$ dem Zustand x zu. Die $m \times n$ Matrix $\mathbf{H}$ aus Gl. 4.16 setzt wiederum den aktuellen Zustand x_k zur Messung z_k in Beziehung. Die Matrizen $\mathbf{A}$ und $\mathbf{H}$ können sich in jedem Zeitschritt ändern, werden hier allerdings ebenfalls als konstant angenommen.

4.3.1.2 Einige Definitionen

Für die weitere Betrachtung werden an dieser Stelle einige Definitionen eingeführt.

Die Zustandsschätzung[6] wird mit $\hat{x} \in \mathbb{R}^n$ bezeichnet, wobei noch genauer differenziert wird: Mit $\hat{x}_k^- \in \mathbb{R}^n$ wird die *a priori* Schätzung zum Zeitpunkt k bezeichnet, also der Schätzwert mit Kenntnis des gesamten Prozesses bis zum aktuellen Zeitpunkt k, wohingegen $\hat{x}_k \in \mathbb{R}^n$ die *a posteriori* Schätzung bezeichnet, die zusätzlich noch Kenntnis von der Messung z_k zum aktuellen Zeitpunkt k besitzt.

Desweiteren lassen sich daraus die *a priori* und *a posteriori* Fehler bestimmen:

$$e_k^- = x_k - \hat{x}_k^- \ , \tag{4.19}$$

$$e_k \ = x_k - \hat{x}_k \ . \tag{4.20}$$

Sie sind definiert als die Differenz aus wahrem Zustand x_k und der jeweiligen Zustandsschätzung $\hat{x}_k^-$ bzw. $\hat{x}_k$.

[5] Unter einem *weißen Rauschen* versteht man ein theoretisches Modell für einen stochastischen Prozeß von unkorrelierten Zufallsvariablen mit Erwartungswert Null. Es ist, einfacher ausgedrückt, ein *stationäres Rauschen*.

[6] also der Erwartungswert von Zustand x

Die *a priori* Schätzfehler-Kovarianzmatrix läßt sich dann mit dem Erwartungswert aus dem dyadischen Produkt des *a priori* Fehlers mit sich selber (in transponierter Form) erhalten:

$$\mathbf{P}_k^- = E(e_k^- \cdot e_k^{-\mathsf{T}}) \ . \tag{4.21}$$

Analog dazu erhält man die Kovarianz des *a posteriori* Fehlers:

$$\mathbf{P}_k = E(e_k \cdot e_k^{\mathsf{T}}) \ . \tag{4.22}$$

4.3.1.3 Der *Zeit-Update* Schritt

Der Zeit-Update Schritt des DKF-Kreislaufs (siehe Abb. 4.1) beschreibt den zeitlichen Fortschritt vom vorherigen Zeitschritt k–1 zum aktuellen Zeitschritt k. Dementsprechend wird von der *a posteriori* Schätzung des vorherigen Zeitschritts $\hat{x}_{k-1}$ auf die *a priori* Schätzung des aktuellen Zeitschritts $\hat{x}_k^-$ geschlossen:

$$\hat{x}_k^- = \mathbf{A}\,\hat{x}_{k-1} + \mathbf{B}\,u_{k-1} \ . \tag{4.23}$$

Gl. 4.23 weist große Ähnlichkeit zur ursprünglichen Prozeßgleichung 4.15 auf. Die Zustandsschätzung aus dem vorherigen Schritt wurde genauso berücksichtigt, wie der (optionale) Kontroll-Input. Nur die Ungenauigkeit w des Prozesses wurde bisher nicht betrachtet. Da diese jedoch nicht einfach vernachlässigt werden darf, wird sie mit folgender Gleichung berücksichtigt:

$$\mathbf{P}_k^- = \mathbf{A} \cdot \mathbf{P}_{k-1} \cdot \mathbf{A}^{\mathsf{T}} + \mathbf{Q} \ . \tag{4.24}$$

Hierbei wird die *a posteriori* Fehler-Kovarianzmatrix $\mathbf{P}_{k-1}$ des Prozesses aus dem vorherigen Zeitschritt k–1 um den Wert der Kovarianzmatrix des Prozeßrauschens erhöht und damit zur aktuellen *a priori* Fehler-Kovarianzmatrix $\mathbf{P}_k^-$ ergänzt.

4.3.1.4 Der *Messung-Update* Schritt

Der zweite Schritt des Kreislaufes in Abb. 4.1 kombiniert nun die *a priori* Zustandsschätzung $\hat{x}_k^-$ mit der Messung z_k zur neuen *a posteriori* Schätzung $\hat{x}_k$. Dabei werden abhängig von den Fehlern der Messung und der *a priori* Schätzung die einzelnen Werte unterschiedlich gewichtet:

$$\hat{x}_k = \hat{x}_k^- + \mathbf{K} \cdot \left(z_k - \mathbf{H}\,\hat{x}_k^-\right) \ . \tag{4.25}$$

Der Klammerausdruck, der den Unterschied zwischen Messung und vorheriger Schätzung des Zustandes beschreibt, wird als *Residuum* bezeichnet. Ein Residuum von Null bedeutet schlicht, daß die *a priori* Schätzung (nach Transformierung durch $\mathbf{H}$)

mit der Messung übereinstimmt, so daß diese auch gleich zur *a posteriori* Schätzung wird.

Der Gewichtungsfaktor des Residuums, die $n \times m$ Matrix $\mathbf{K}$, ist das sogenannte *Kalman Gain*, welches die *a posteriori* Fehler-Kovarianz aus Gl. 4.22 verkleinert. Eine mögliche Darstellungsform für $\mathbf{K}$ ist in folgender Gleichung 4.26 gegeben. Die Umformung obiger Gleichungen zu Gl. 4.26 ist nicht sonderlich schwierig, wird hier jedoch weggelassen.[7]

$$\mathbf{K}_k = \mathbf{P}_k^- \, \mathbf{H}^\mathsf{T} \cdot \left(\mathbf{H} \, \mathbf{P}_k^- \, \mathbf{H}^\mathsf{T} + \mathbf{R} \right)^{-1} \quad . \tag{4.26}$$

Bei näherer Betrachtung wird man feststellen, daß das *Kalman Gain* das Residuum stärker gewichtet, wenn die Meßfehler-Kovarianzmatrix $\mathbf{R}$ gegen Null geht:

$$\lim_{\mathbf{R}_k \to 0} \mathbf{K}_k = \mathbf{H}^{-1} \quad . \tag{4.27}$$

Im Umgekehrten Fall wird das Residuum weniger gewichtet, wenn die *a priori* Fehler-Kovarianzmatrix $\mathbf{P}_k^-$ gegen Null strebt:

$$\lim_{\mathbf{P}_k^- \to 0} \mathbf{K}_k = 0 \quad . \tag{4.28}$$

Anders ausgedrückt, wird also in Gl. 4.25 der Messung mehr Glauben geschenkt, wenn das Meßrauschen kleiner wird, womit auch die Meßfehler-Kovarianzmatrix $\mathbf{R}$ gegen Null geht, wohingegen dem vorhergesagten Meßwert $\mathbf{H} \, \hat{x}_k^-$ nur wenig Bedeutung beigemessen wird. Genau umgekehrt verhält es sich für den Fall, daß die *a priori* Fehler-Kovarianzmatrix $\mathbf{P}_k^-$ gegen Null strebt, das Prozeßrauschen damit abnimmt, wobei der wirklichen Messung z_k viel weniger Bedeutung beigemessen wird als der prädizierten Messung $\mathbf{H} \, \hat{x}_k^-$.

Was noch fehlt, ist der neue Wert für die *a posteriori* Fehler-Kovarianz $\mathbf{P}_k$, der im nächsten Zeitschritt in Gl. 4.24 wieder benötigt wird:

$$\mathbf{P}_k = (\mathbf{I} - \mathbf{K}_k \, \mathbf{H}) \, \mathbf{P}_k^- \quad . \tag{4.29}$$

4.3.1.5 Der gesamte Update-Kreislauf

Die Gleichungen von Zeit-Update und Messung-Update werden in Abb. 4.2 zusammenfassend dargestellt. Zuletzt müssen noch initiale Werte für $\hat{x}_{k-1}$ und $\mathbf{P}_{k-1}$ gewählt werden, damit der DKF seine Arbeit aufnehmen kann.

[7] Der interessierte Leser wird hierfür und für weitere detaillierte Ausführungen auf [33, 34, 35, 36] verwiesen.

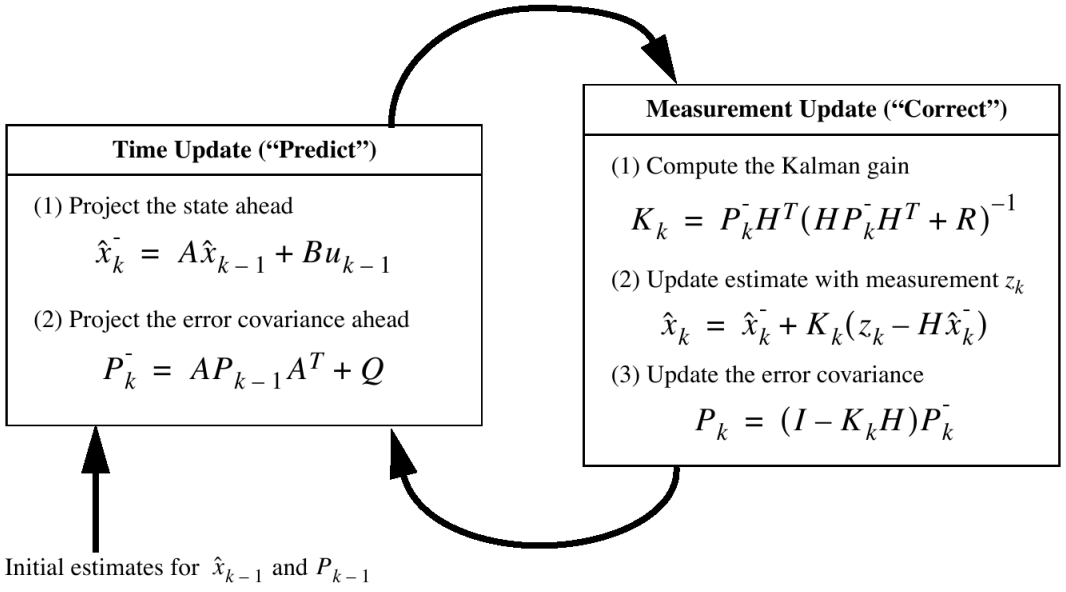

Abbildung 4.2: Der vollständige Kreislauf des *Diskreten Kalman-Filters*. (Entnommen aus [33])

4.3.1.6 Parameter-Optimierung

Je nach Implementierung kann der DKF unterschiedlich effizient sein. Ein möglicher Weg ihn zu beschleunigen, ist die geeignete Vorausberechnung einiger seiner Parameter.

Zum einen läßt sich die Meßfehler-Kovarianzmatrix $\mathbf{R}$ häufig im voraus bestimmen. Da für die Verwendung des DKF der zu bestimmende Prozeß ohnehin meßbar sein muß, sollte auch der Meßfehler und damit dessen Kovarianzmatrix $\mathbf{R}$ bestimmbar sein. Eine einfache und relativ schnelle Methode hierfür ist, eine Versuchsreihe durchzuführen und bei deren Auswertung den Fehler zu bestimmen. Damit kann man $\mathbf{R}$ relativ leicht vorab bestimmen.
Analog kann man versuchen, die Kovarianzmatrix $\mathbf{Q}$ des Prozeßrauschens zu bestimmen. Dies gestaltet sich allerdings als schwierig, da der zu untersuchende Prozeß meist nicht direkt observierbar ist. Allerdings sollte man beachten, daß ein relativ einfaches Prozeßmodell dennoch ein gutes Resultat liefern kann, wenn nur genug „Unsicherheit" durch geeignete Wahl von $\mathbf{Q}$ eingespeist wird. Dies setzt jedoch voraus, daß die Messungen z_k zuverlässig sind.[8]

Sollten sich die Kovarianzmatrizen $\mathbf{Q}$ und $\mathbf{R}$ als konstant herausstellen, so werden sich die Schätzfehler-Kovarianz $\mathbf{P}_k$ und das *Kalman Gain* $\mathbf{K}_k$ relativ schnell stabilisieren. Diese können dann ebenfalls vor Verwendung des DKF „offline" bestimmt

[8]Eine Annahme, die nicht auf die vom *Kanatani*-Algorithmus gelieferte Schätzung anwendbar ist, die als Messung verwendet wird.

werden und brauchen somit auch nicht mehr für jeden Zyklus neu berechnet zu werden.

Ein gerne verwendetes Hilfsmittel zur *offline*-Optimierung ist häufig ein weiterer Kalman-Filter, dessen Parameter ebenfalls noch optimiert werden können.

4.3.2 Erweiterter Kalman-Filter

Den Nachteil des DKF, nur auf *lineare* dynamische Systeme anwendbar zu sein, behebt der *Erweiterte Kalman-Filter* (*kurz:* EKF). Dieser kann auch auf *nicht-lineare* dynamische Systeme angewandt werden, indem er über arithmetisches Mittel und Kovarianz linearisiert.

4.3.2.1 Der zu schätzende nicht-lineare Prozeß

Der EKF versucht analog zum DKF einen Zustand $x \in \mathbb{R}^n$ eines diskret-Zeit kontrollierten Prozesses zu schätzen, der durch die folgende, nicht-lineare, stochastische Gleichung beschrieben ist:

$$x_k = f\left(x_{k-1}, u_{k-1}, w_{k-1}\right) \quad . \tag{4.30}$$

Eine zugehörige Messung $z \in \mathbb{R}^m$ ist wie folgt definiert:

$$z_k = h\left(x_k, v_k\right) \quad . \tag{4.31}$$

Die Variablen w_{k-1} und v_k repräsentieren wie beim DKF das Prozeß- bzw. Meßrauschen und sind genau wie in Gl. 4.17 bzw. 4.18 verteilt.

Die *nicht-lineare* Funktion f ordnet den vorherigen Zustand x_{k-1}, den optionalen Kontroll-Input u_{k-1} und das Prozeßrauschen w_{k-1} dem aktuellen Zustand x_k zu. Analog setzt die ebenfalls *nicht-lineare* Funktion h den Zustand x_k mitsamt Meßrauschen v_k zur Messung z_k in Beziehung.

In der Praxis kennt man die individuellen Werte für w_{k-1} und v_k häufig nicht. Doch kann man den Zustandsvektor sowie den Meßvektor approximieren, so daß man

$$\tilde{x}_k = f\left(\hat{x}_{k-1}, u_{k-1}, 0\right) \tag{4.32}$$

und

$$\tilde{z}_k = h\left(\hat{x}_k, 0\right) \tag{4.33}$$

erhält, wobei $\hat{x}_{k-1}$ eine nicht näher beschriebene *a posteriori* Schätzung des Zustands aus dem vorherigen Zeitschritt k–1 ist, $\tilde{x}_k$ die approximierte *a priori* Schätzung und $\tilde{z}_k$ die approximierte Messung.

Ein wesentlicher Nachteil des EKF besteht darin, daß die Verteilung (bzw. deren Dichte) der verschiedenen Zufallsvariablen durch die nicht-linearen Transformationen so verändert wird, daß sie nicht länger als standard-normal verteilt angenommen werden können.

4.3.2.2 Linearisierte Form des nicht-linearen Prozesses

Um nun einen, wie im vorangegangenen Abschnitt beschriebenen, nicht-linearen Prozeß zu schätzen, wird eine Linearisierung, ähnlich einer Taylor-Entwicklung, folgendermaßen vorgenommen:

$$x_k \approx \tilde{x}_k + \mathbf{A}\,(x_{k-1} - \hat{x}_{k-1}) + \mathbf{W}\,w_{k-1} \quad , \tag{4.34}$$

$$z_k \approx \tilde{z}_k + \mathbf{H}\,(x_k - \tilde{x}_k) + \mathbf{V}\,v_k \quad . \tag{4.35}$$

Hierbei sind x_k und z_k die wahren Zustands- und Meßvektoren, $\tilde{x}_k$ und $\tilde{z}_k$ die approximierten Zustands- und Meßvektoren aus Gl. 4.32 und 4.33 sowie $\hat{x}_{k-1}$ eine *a posteriori* Schätzung des Zustandsvektors. Wie in Gl. 4.17 und 4.18 angegeben, repräsentieren die Zufallsvariablen w_{k-1} und v_k wieder das Prozeß- bzw. Meßrauschen.

Bei den vier Matrizen $\mathbf{A}$, $\mathbf{W}$, $\mathbf{H}$ und $\mathbf{V}$ handelt es sich um die Jacobi-Matrizen mit den partiellen Ableitungen der Funktionen f und h. Diese sind im folgenden verkürzt wiedergegeben:[9]

$$\mathbf{A}_{[i,j]} = \frac{\partial f_{[i]}}{\partial x_{[j]}}\,(\hat{x}_{k-1}, u_{k-1}, 0) \quad , \tag{4.36}$$

$$\mathbf{W}_{[i,j]} = \frac{\partial f_{[i]}}{\partial w_{[j]}}\,(\hat{x}_{k-1}, u_{k-1}, 0) \quad , \tag{4.37}$$

$$\mathbf{H}_{[i,j]} = \frac{\partial h_{[i]}}{\partial x_{[j]}}\,(\tilde{x}_k, 0) \quad , \tag{4.38}$$

$$\mathbf{V}_{[i,j]} = \frac{\partial h_{[i]}}{\partial v_{[j]}}\,(\tilde{x}_k, 0) \quad . \tag{4.39}$$

4.3.2.3 Der Update-Kreislauf

Nach einigen Umformungen, ähnlich wie beim DKF, erhält man aus Gl. 4.34 und 4.35 die einzelnen Gleichungen des EKF für den Zeit-Update und Messung-Update Schritt. Diese sind in Abb. 4.3 mitsamt des zugehörigen Kreislaufs dargestellt. Eine detailliertere Betrachtung dieser Umformungen und der resultierenden Gleichungen wird an dieser Stelle weggelassen.[10]

Es bleibt noch zu erwähnen, daß ein wichtiges Charakteristikum des EKF darin besteht, daß die Jacobi-Matrix $\mathbf{H}_k$ in der Gleichung für das *Kalman Gain* $\mathbf{K}_k$ dabei hilft, nur die relevante Komponente der gemessenen Information zu propagieren.

[9] Für bessere Lesbarkeit wurde der Index für den jeweiligen Zeitschritt k weggelassen, obwohl die Jacobi-Matrizen in jedem Zeitschritt sehr wohl unterschiedlich sein können.

[10] Detailliertere Informationen dazu finden sich im Abschnitt „The Computational Origin of the Filter“ in [33].

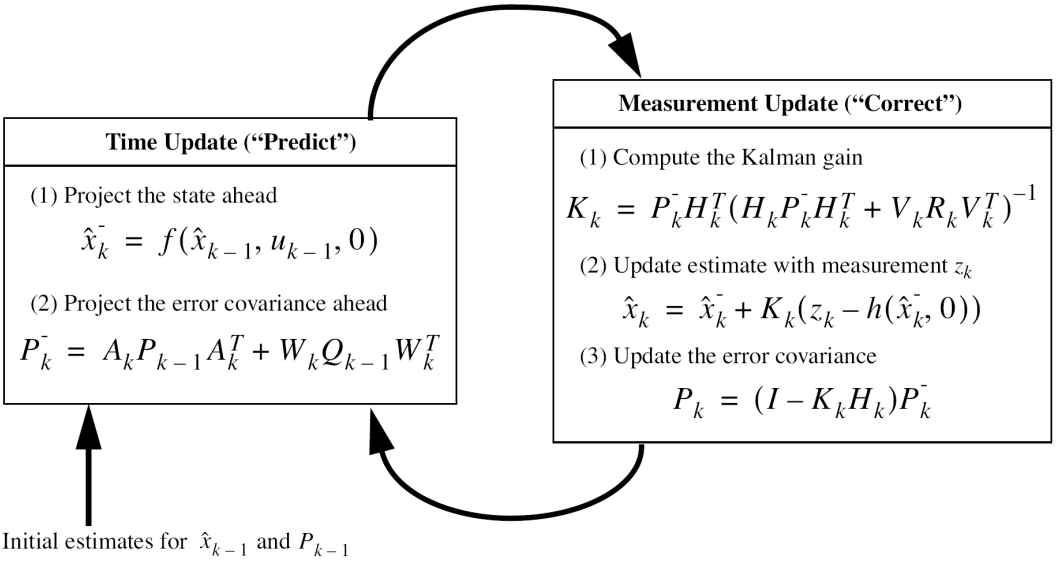

Abbildung 4.3: Der vollständige Kreislauf des *Erweiterten Kalman-Filters*. (Entnommen aus [33])

Falls z.B. keine eindeutige eins-zu-eins Abbildung zwischen Messung z_k und dem Zustand x_k via h möglich ist[11], so beeinflußt $\mathbf{H}_k$ das *Kalman Gain* derart, daß es nur den Teil des Residuums $z_k - h(\hat{x}^-, 0)$ näher beleuchtet, welches auch den Zustand x_k beeinflußt. Falls über alle Messungen keine einzige eins-zu-eins Abbildung zwischen Messung z_k und Zustand via h existiert, so kann der Filter für diese fehlenden Komponenten relativ schnell divergieren. Der Prozeß ist damit in seiner Gesamtheit *nicht observierbar* [33].

[11] So wie im späteren Abschnitt 6.1.3.

Teil II

Ergebnisse und Diskussion

5. Softwaresystem zur Ansteuerung des Flugroboters

Bevor in den folgenden Abschnitten eine Eigenbewegungsbestimmung des Flugroboters entwickelt werden kann, muß zuvor ein Softwaresystem entwickelt werden, welches das Grundgerüst (im folgenden *Framework* genannt) zur Ansteuerung des Flugroboters darstellt. Dazu gehören neben der eigentlichen Steuerung des Roboters auch das Abrufen, Verarbeiten und Visualisieren von Sensordaten.

5.1 Modularität des Softwaresystems

Bei der Entwicklung dieses Frameworks wurde viel Wert auf Modularität und damit leichte Austauschbarkeit gelegt. Deshalb wurde unter anderem die objektorientierte Programmiersprache C++ verwendet. Diese hat zudem den Vorteil, daß sie sich in sehr effizienten Maschinencode übersetzen läßt.
Um ein hohes Maß an Modularität und Austauschbarkeit zu erreichen, wurden die einzelnen Teilaufgaben in separate Module aufgeteilt, die sogar als einzelne Prozesse ausgelagert werden können, welche über einen gemeinsamen Programmspeicher (*Shared Memory*) miteinander kommunizieren und Daten austauschen. Zum anderen wurden die einzelnen Module so gestaltet, daß sie sich selbst wieder leicht durch andere Module ersetzen lassen. Diese Ersatz-Module könnten dann die jeweilige Aufgabe auf andere Weise ausführen oder sogar nur als Dummy-Module fungieren und sie gänzlich unbearbeitet lassen. Letzteres ist natürlich nur während der Entwicklungs- und Testphase sinnvoll.

5.1.1 Inter-Prozeß-Kommunikation

Bereits mit Hilfe der besonderen Eigenschaften von objektorientierten Programmiersprachen, welche in Kapitel 3.1 angesprochen wurden, läßt sich eine Modularisierung weitgehend bewerkstelligen. Eine weitere Separierung in einzelne Module erfordert eine Aufspaltung dieser Module in einzelne Prozesse. Dies stellt grundsätzlich kein großes Problem dar, jedoch müssen diese einzelnen Prozesse in der Lage sein, miteinander zu kommunizieren und Daten auszutauschen.
Dies läßt sich z. B. über den Weg gemeinsamer Dateien erreichen, die von einem Prozeß geschrieben und vom anderen gelesen werden. Jedoch ist dies eine sehr ineffiziente Art, Daten auszutauschen. Der Flaschenhals ist dabei der I/O-Zugriff auf die Festplatte, welcher um einiges langsamer ist, als der Zugriff auf den Programmspeicher. Ein weit besserer und vor allem effizienterer Weg ist die Kommunikation über gemeinsamen Programmspeicher, sogenanntes „*Shared Memory*“.

Dabei wird von einem Prozeß ein Shared Memory Segment im Programmspeicher angelegt, zu dem sich andere Prozesse verbinden können. Um den Zugriff darauf zu synchronisieren, damit nicht etwa ein Prozeß gerade dann versucht, Daten zu überschreiben, während ein anderer Prozeß noch am Lesen der vorherigen Daten ist, wird ein System aus sogenannten *Semaphoren* verwendet, die einer Art Zähler entsprechen, der angibt, wieviele Prozesse gerade Lese- bzw. Schreibzugriff auf das jeweilige Shared Memory Segment haben. Daten dürfen somit nur überschrieben werden, wenn außer dem schreibenden Prozeß kein weiterer Prozeß Lese- oder Schreibzugriff auf das Shared Memory Segment ausführt. Die genaue Funktionsweise dieser wird hier nicht beschrieben. Stattdessen wird auf die Arbeit von Edsger Dijkstra verwiesen, in der grundlegende Überlegungen zu diesem Thema ausgeführt werden [37, 38][1].

Um zu Verhindern, daß der schreibende Prozeß immer solange warten muß, bis der lesende Prozeß fertig ist, wurde bei dem in dieser Arbeit entwickelten Software-Framework in jedem Shared Memory Segment eine Ringpuffer-Struktur angelegt. Hierbei wird jede einzelne Zelle des Ringpuffers mit einer eigenen Semaphore synchronisiert. Der schreibende Prozeß sucht sich dann die nächste Pufferzelle, von der kein Prozeß liest, und schreibt seine Daten dort hinein. Damit der lesende Prozeß weiß, wo die neuen Daten liegen, wird am Anfang des Shared Memory Segments vor dem Ringpuffer noch in einen kleinen Bereich geschrieben, in welcher Zelle des Ringpuffers die neuesten Daten zu finden sind und wann sie dort abgelegt wurden.

Um die Abstraktion der Modularisierung auch über Prozeßgrenzen hinweg aufrecht zu erhalten, ist der Zugriff auf diese im Shared Memory befindlichen Ringpuffer

[1] Bei [37] handelt es sich um die von Dijkstra auf niederländisch verfaßte Originalarbeit. Für eine englische Erläuterung wird auf [38] verwiesen.

durch zwei verschieden bezeichnete Modul-Implementationen geregelt. Ein das Shared Memory Segment beschreibende Modul verwendet seine sogenannte *ShmServer*-Implementation, ein weiteres, dieses lesende Modul hingegen seine *ShmClient*-Implementation. Für die verschiedenen Module sind die jeweiligen ShmServer/Client-Implementationen zwar ähnlich, aber dennoch insoweit verschieden, als daß jedes Modul seine eigene spezielle ShmServer- bzw. ShmClient-Implementation erhalten muß.

5.2 Die einzelnen Software-Module

Im folgenden wird näher auf die einzelnen Software-Module eingegangen. Dafür wurde in Abb. 5.1 das Software-Framework mit den wesentlichen Modulen bzw. Prozessen dargestellt.[2]

Fett schwarz umrandet sind die einzelnen Prozesse dargestellt, dünn grau umrandet die unterschiedlichen Implementierungen für die sie umschließenden Module, welche über einen Eintrag in der zugehörigen Konfigurationsdatei ausgewählt werden. Wie man sieht, sind einige der Module sowohl als einzelne, über Shared Memory kommunizierende Prozesse als auch als Untermodule eines anderen Prozesses (hier des *RobotControl* Prozesses) vorhanden. Dies beruht auf der Tatsache, daß einzelne Module sowohl als Teil eines anderen Prozeß vorkommen, als auch als eigene Prozesse ausgelagert werden können.
Falls ein Modul als eigener Prozeß ausgelagert wird, greift ein anderer Prozeß über dessen *ShmClient*- bzw. *ShmServer*-Implementierung auf das Modul (bzw. dessen Daten) zu. Für diesen anderen Prozeß ist es dann so, als wäre das gesamte Modul nur ein Untermodul im gleichen Prozeß.

Z. B. ist es für den *RobotControl* Prozeß unerheblich, ob der *ImageGrabber* Prozeß ausgelagert wird oder nicht; die Art und Weise, wie er auf Daten von diesem zugreift, ist genau gleich: Er ruft die zugehörige `grab`-Funktion auf.
Wenn nun also der *ImageGrabber* als eigener Prozeß ausgelagert wurde, dann wurde somit das entsprechende *ImageGrabber*-Untermodul im *RobotControl* Prozeß durch den zugehörigen *ShmClient* implementiert. Dessen `grab`-Funktion greift dann auf das entsprechende Shared Memory Segment zu und stellt die dort vorhandenen Bilddaten dem *RobotControl* Prozeß zur Verfügung. Der ausgelagerte *ImageGrabber* Prozeß hingegen sorgt dafür, daß sich im Shared Memory Segment immer die aktuellsten Bilddaten befinden. Analog gehen die übrigen Module vor.

[2] Das gesamte Software-Framework ist durchaus größer und komplexer, jedoch ist der in Abb. 5.1 dargestellte Ausschnitt ausreichend, um die wesentliche Struktur nachzuvollziehen. Das gesamte Framework ist einheitlich aufgebaut und läßt sich beim Nachvollziehen dieses Ausschnitts leicht aus dem Quellcode erschließen.

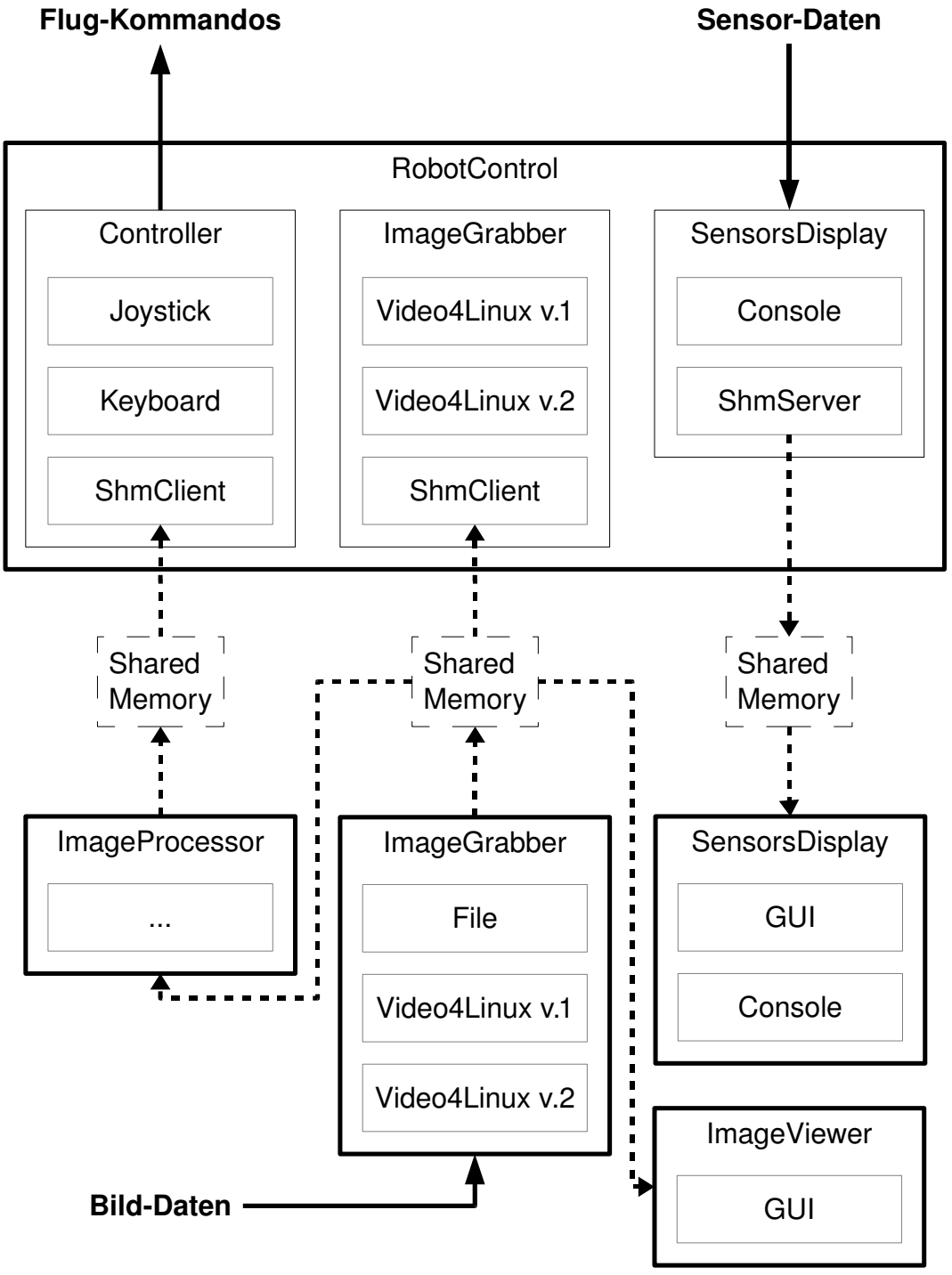

Abbildung 5.1: Repräsentativer Ausschnitt des Software-Frameworks mit den einzelnen Modulen bzw. Prozessen. Fett schwarz umrandet sind einzelne Prozesse und dünn grau umrandet hingegen mögliche Implementierungen für die jeweiligen Module dargestellt.

Im folgenden werden die einzelnen Module näher erläutert. (Hinweis: Alle Module verfügen grundsätzlich über eine *Dummy*-Implementation, welche keine Funktionen besitzt und nur für Testzwecke zum Einsatz kommt.)

5.2.1 Das *ImageGrabber* Modul

Der *ImageGrabber* ist das Software-Modul, welches für das Bereitstellen der Kamerabilder im gewünschten Bildformat und mit der gewünschten Auflösung zuständig ist. Der ImageGrabber besitzt vier verschiedene Implementationen. Zwei davon stellen direkt die von der Kamera zum Videograbber gesendeten Bilder zur Verfügung, während eine weitere eine vormals aufgenommene und in Dateien gespeicherte Bildersequenz wieder bereitstellt. Die vierte Implementation hingegen ist die im vorherigen Abschnitt schon erwähnte *ShmClient*-Implementation, welche anderen Prozessen ermöglicht, auf die von dem eigentlichen ImageGrabber-Prozeß bereitgestellten Bilddaten über Shared Memory zuzugreifen.

Die erste Implementation verwendet die sogenannte *Video4Linux Version 1* API des Linux-Betriebsystems. Diese ermöglicht es, über einige einfache Funktionsaufrufe den Videograbber zu initialisieren, so daß dieser die von ihm digitalisierten Kamerabilder im gewünschten Bildformat und mit der gewünschten Auflösung liefert. Das Abrufen dieser digitalisierten Bilder geschieht ebenfalls über solche simplen „high-level“ Funktionen.
Als Implementation wurde ein schon vorhandenes einfaches Programm zum Abrufen von Bildern über diese Video4Linux-Schnittstelle so modifiziert, daß es sich als Modul im ImageGrabber einbinden läßt. Jedoch wird die Version 1 der Video4Linux API seit einiger Zeit nicht mehr weiterentwickelt, weshalb die Linux-Treiber neuerer Videograbber diese nicht mehr zu 100% unterstützen.[3]

Allerdings hat die Linux-Community diese API nicht einfach nur fallen lassen, sondern durch eine leistungsfähigere *Video4Linux Version 2* API ersetzt. Die zweite, gänzlich neu geschriebene Implementation für den ImageGrabber verwendet diese neuere Schnittstelle. Damit bot sich auch der Vorteil, in die Implementation zugleich eine größere Funktionalität einzubauen. Im Vergleich zur zuvor beschriebenen Implementierung lassen sich die Bilder nun in weiteren unterschiedlichen Bildformaten abrufen. Auch läßt sich die zu verwendende Fernsehnorm (PAL, NTSC) direkt angeben sowie die Verwendung des Zeilensprungverfahrens deaktivieren, um sich z. B. nur eines der beiden Halbbilder ausgeben zu lassen.[4] Daneben ist es jetzt auch

[3] Unter anderem fällt der bei dieser Arbeit verwendete mobile Videograbber in diese Kategorie, weshalb dessen Verwendung mit der erwähnten ImageGrabber-Implementation nicht möglich war.

[4] Das Zeilensprungverfahren, ein Relikt aus der Zeit des Analogfernsehens, setzt ein Bild aus zwei nacheinander aufgenommene Halbbilder zusammen, welche zeilenversetzt dargestellt werden. Damit kann es bei schneller Bildbewegung zu störenden Streifeneffekten kommen.

möglich, einen Bildausschnitt auszuwählen, der anstelle des vollständigen Bildes zurückgegeben wird. Ein weiterer großer Vorteil besteht darin, daß die Bilder nun mit einem realen Zeitstempel (*Timestamp*), also dem Zeitpunkt, zu dem sie gemacht wurden (bzw. zu dem die ersten Bilddaten im Videograbber ankamen), versehen werden und eine fortlaufende Nummer erhalten. Somit läßt sich später leicht zurückverfolgen, ob man zwischenzeitlich ein oder mehrere Bilder ausgelassen hat und wie alt die verwendeten Bilddaten sind.[5]

Beiden ImageGrabber-Implementationen ist gemein, daß die Verwendung der Video4Linux Schnittstelle eine effiziente Einbindung der vom Videograbber digitalisierten Bilder in das eigene Programm ermöglicht. Denn anstatt gezwungen zu sein, die Bilder über den langsamen *read/write*-Mechanismus des Linux-Betriebsystems in den Adreßspeicher des eigenen Programms zu kopieren, läßt sich der Speicher des Videograbbers, in dem das Bild residiert, in den Adreßspeicher des Programms „mappen“. Somit wird ein Zugriff auf diesen Adreßspeicher im Programm automatisch auf den Speicher des Videograbbers umgeleitet, in dem sich das Bild befindet.[6]

Falls der ImageGrabber als eigener Prozeß gestartet wird, besteht die Möglichkeit, über einen zusätzlichen Kommandozeilen-Parameter den ImageGrabber anzuweisen, die bereitgestellten Bilder automatisch als eigene Dateien auf der Festplatte abzulegen. Dabei werden die reinen Datenbytes jedes Bildes binär in eine Datei geschrieben, wobei am Anfang der Datei noch ein Block mit Meta-Informationen geschrieben wird. Dieser enthält alle wesentlichen Informationen wie Bildgröße, Bildformat, verwendetes Zeilendarstellungsverfahren, aber auch Timestamp und Nummer.[7]

Um diese gespeicherten Bilder nun wieder bereitstellen zu können, wurde eine dritte Implementation für den ImageGrabber entwickelt, welche ein wenig anders aufgebaut ist als die ersten beiden. Anstatt, wie es die anderen beiden Implementationen tun, beim Starten alle Bildeinstellungen aus einer Konfigurationsdatei zu übernehmen, öffnet sich hier ein graphischer Benutzerdialog, der es erlaubt auszuwählen, welche Bilddateien man laden und bereitstellen möchte. Die Bilder werden dabei in alphanumerischer Reihenfolge abgearbeitet. Abb. 5.2 stellt die Oberfläche dieser „ImageFileGrabber“ Implementation dar.

[5] Mit dieser ImageGrabber-Implementation konnte nunmehr auch der mobile Videograbber verwendet werden. Allerdings stellte sich heraus, daß dessen Linux-Teiber sehr unflexibel programmiert wurde, so daß alle Bilder nur im Zeilensprungverfahren mit einer festen Auflösung von 720×576 Pixel im *YUV422* Bildformat abgerufen werden. Der zweite, alternativ verwendete, stationäre Videograbber hingegen übernahm alle vorgenommenen Einstellungen ohne Probleme.

[6] Dies entspricht somit einer Art Shared Memory.

[7] Es wurde davon abgesehen, ein existierendes Bildformat, wie z.B. JPEG oder PNG, zu verwenden, weil damit eine (verlustbehaftete) Konvertierung der Daten und ein größerer Rechenaufwand verbunden wäre. Der dafür in Kauf genommene Nachteil besteht allerdings in einem erhöhten Speicherplatzbedarf.

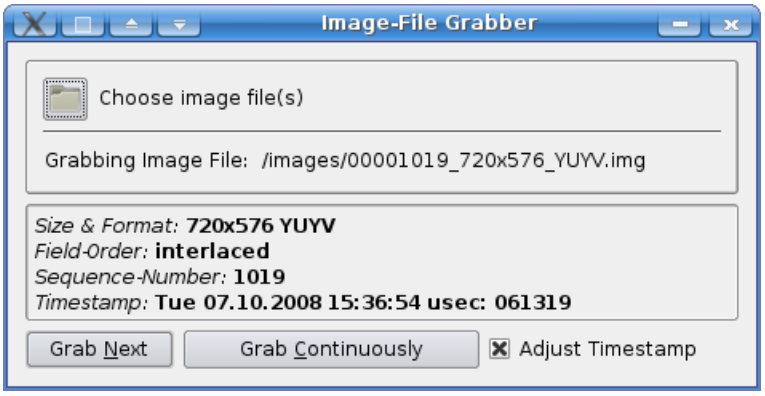

Abbildung 5.2: Graphische Benutzeroberfläche der *File*-Implementation des *Image-Grabbers*. Diese Implementation lädt zuvor gespeicherte Bildersequenzen und stellt sie wieder bereit.

Man kann durch Klicken des Buttons „Grab Next“ das nächste Bild einzeln laden, wobei die in der jeweiligen Bilddatei gespeicherten Meta-Informationen übernommen werden. Das Bild wird dann so lange bereitgestellt, bis der nächste Klick auf einen weiteren Button erfolgt.
Falls bei „Adjust Timestamp“ kein Häkchen gesetzt wurde, wird die Timestamp ebenso wie die übrigen Meta-Informationen des gespeicherten Bildes unverändert übernommen, bei gesetztem Häkchen wird sie stattdessen durch die aktuelle Zeit ersetzt. Die Notwendigkeit zum Anpassen der Timestamp ist dadurch gegeben, daß andere, auf diesen Bildern operierende Module häufiger Zeitvergleiche anstellen, um das Alter der Aufnahme und damit die seitdem vergangene Zeit festzustellen. Somit ist die originale Timestamp nur für Test- und Debugging-Zwecke sinnvoll.
Anstatt auf „Grab Next“ zu klicken, kann man auch alternativ auf „Grab Continuously“ klicken, wodurch die Bilder kontinuierlich hintereinander geladen und bereitgestellt werden. Auch hierbei wird die Timestamp der Bilder bei gesetztem Häkchen durch die aktuelle Zeit ersetzt. Der Unterschied besteht darin, daß für alle weiteren Bilder der zeitliche Abstand der ursprünglichen Timestamps Beachtung findet. D. h., der ImageGrabber berechnet aus den ursprünglichen Timestamps des gerade dargestellten Bildes und des Folgebildes die Zeitdauer, die er warten muß, bis er das Folgebild (mit der dann aktuellen Zeit bei gesetztem Häkchen bzw. der originalen Zeit bei nicht gesetztem Häkchen) bereitstellen darf. Somit wird erreicht, daß die Bilder in der genauen zeitlichen Abfolge abgespielt werden, in der sie aufgenommen wurden, womit sich wiederum für ein anderes Programm-Modul, welches auf Bildern des ImageGrabbers operiert, kein Unterschied zwischen einer direkt von der Kamera gelieferten Sequenz und einer zuvor aufgezeichneten und nun wieder bereitgestellten Bildersequenz feststellen läßt.

Unabhängig davon, welche der drei ImageGrabber-Implementation verwendet wird oder ob er als eigener Prozeß oder als Untermodul eines anderen Prozesses betrieben wird, ist es möglich, die gelieferten Bilder (zusätzlich) in einem Shared Memory Ringpuffer abzulegen. Somit ist es auch anderen Prozessen möglich, auf diese Bilder zuzugreifen. Dafür müssen diese nur die vierte Implementation, die schon erwähnte *ShmClient*-Implementation des ImageGrabbers, als Untermodul einbinden. Diese verbindet sich zu dem (in der Konfigurationsdatei angegebenen) Shared Memory Ringpuffer und stellt die dort abgelegten Bilder dem jeweiligen Prozeß zur Verfügung.

5.2.2 Das *ImageViewer* Modul

Um die vom ImageGrabber gelieferten Bilder auch betrachten zu können, wurde das *ImageViewer* Modul entwickelt. Dieses ist so konzipiert, daß es nur als eigener Prozeß betrieben werden kann.[8] Der ImageViewer kann den ImageGrabber entweder direkt als eigenes Untermodul im gleichen Prozeß starten, um so unmittelbar auf dessen gelieferte Bilder zuzugreifen und sie darzustellen, oder er startet nur dessen *ShmClient*-Implementation als Untermodul und greift somit über Shared Memory auf die Bilddaten eines weiteren, in einem anderen Prozeß betriebenen, ImageGrabbers zu. Letzteres ist zu bevorzugen, denn damit wird dem Benutzer die Möglichkeit eröffnet, den ImageViewer jederzeit (nachträglich) hinzu zu schalten und eine Kontrolle der Bilddaten durchzuführen.

Der ImageViewer besitzt eine graphische Benutzeroberfläche, welche in Abb. 5.3 dargestellt ist. Wird er gestartet, so greift er auf die Bilder des ImageGrabbers zu, stellt sie aber nur dar, sofern der Benutzer dies durch einen Klick auf einen der beiden Buttons signalisiert.
Der Benutzer kann, analog zur *File*-Implementation des ImageGrabbers, entweder nur das nächste (bzw. aktuellste) Bild, welches vom ImageGrabber geliefert wird, darstellen lassen, indem er auf „Grab Next“ klickt, oder durch Klicken von „Grab Continuously“ kontinuierlich das jeweils aktuellste Bild und somit die aktuelle Bildersequenz. Zusätzlich werden auch die übrigen Meta-Informationen des jeweiligen Bildes (wie Auflösung, Bildformat, Timestamp und Nummer) angezeigt.

Die Darstellung ist leider aufgrund von Beschränkungen der verwendeten GUI-Entwicklungsbibliotheken etwas ineffizient, denn alle Bilddaten müssen zur Darstellung in das RGB32-Format[9] umgewandelt werden.

[8] Prinzipiell ist es möglich, den ImageViewer auch so abzuwandeln, daß er als Untermodul in einem anderen Prozeß betrieben werden kann, aber in der Praxis macht dies aus Gründen der Effizienz keinen Sinn.

[9] RGB32 besitzt vier Kanäle mit jeweils acht Bit, wobei drei Kanäle für die Grundfarben rot, grün und blau verwendet werden und der vierte Kanal als Alpha-Kanal Verwendung findet.

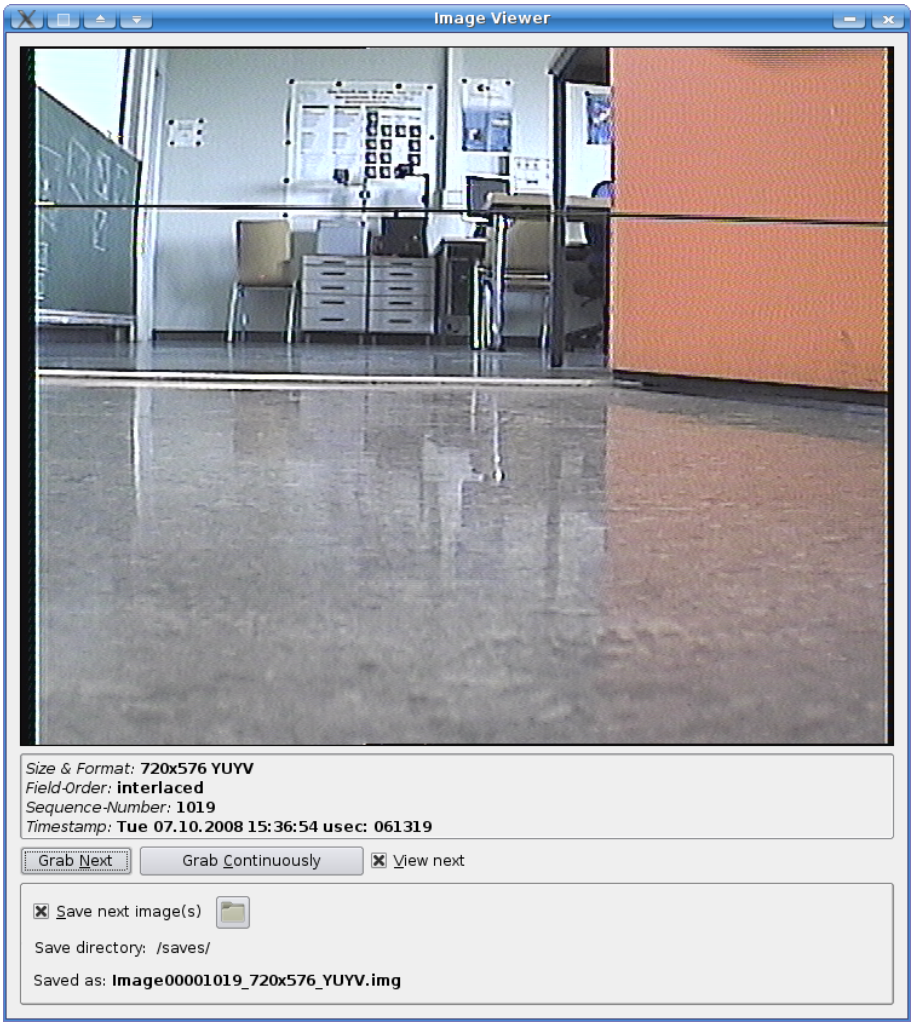

Abbildung 5.3: Graphische Benutzeroberfläche des *ImageViewer*s. Im oberen Bereich wird das Bild dargestellt, darunter dessen Meta-Informationen. Ganz unten läßt sich ein Ort zum Speichern der noch folgenden Bilder angeben.

Neben der Option zum Darstellen von Bildern lassen sich auch Bilder bzw. Bildersequenzen in einem ausgewählten Ordner speichern. Sie werden dabei im gleichen Format gespeichert, wie im vorherigen Abschnitt über den ImageGrabber beschrieben. Im Gegensatz zum ImageGrabber, der alle Bilder speichert, können mit dem ImageViewer auch nur die ausgewählten Bilder bzw. Bildersequenzen gespeichert werden. Um die Effizienz beim Speichern einer Bildersequenz mit dem ImageViewer zu erhöhen, läßt sich das Häkchen bei „View Next“ entfernen, wodurch die zu speichernden Bilder nicht mehr in das RGB32-Format umgewandelt, allerdings damit auch nicht mehr dargestellt werden.

5.2.3 Das *SensorsDisplay* Modul

Das *SensorsDisplay* ist, wie der Name bereits besagt, das Modul, welches die von den Sensoren (mit Ausnahme der Kamera) gelieferten Daten sowie die zum Roboter geschickten Flugkommandos darstellt. Es kann ebenfalls entweder als Untermodul des RobotControl-Prozesses oder als eigener Prozeß gestartet werden. Sofern es als Untermodul gestartet wird, lassen sich die Sensordaten (und Flugkommandos) mit der *Console*-Implementation darstellen. Diese verwendet die Linux-Bibliothek „Ncurses“, welche es erlaubt, auf der Ausgabe-Konsole an definierte Positionen zu schreiben, anstatt immer nur an deren Ende. Damit läßt sich ein definierter Ausgabe-Bereich erstellen, in dem die darzustellenden Daten an die jeweils zugehörigen Positionen geschrieben werden. Abb. 5.4 zeigt beispielhaft die Console-Implementation des SensorsDisplays mitsamt einiger Daten.

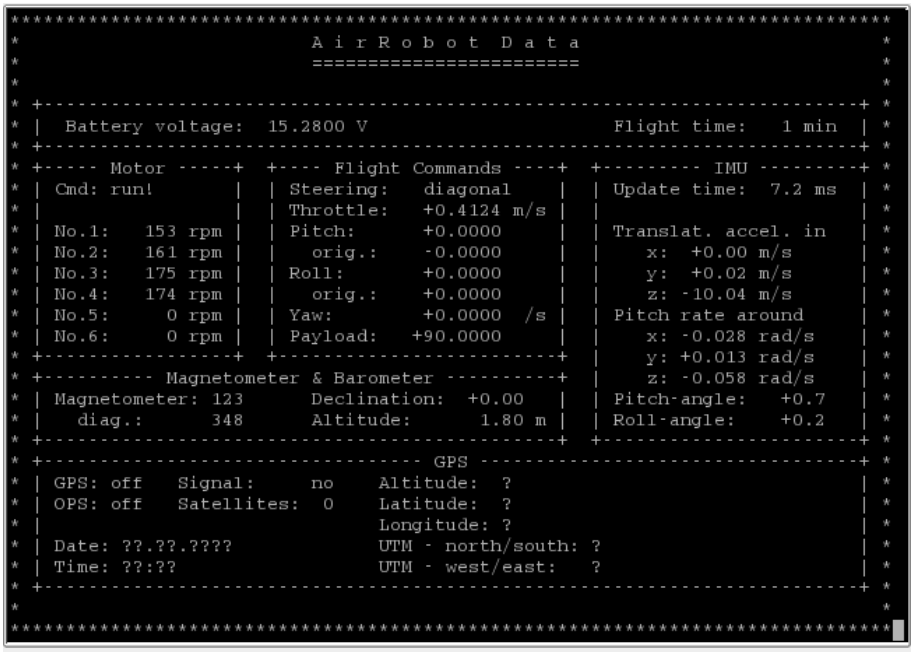

Abbildung 5.4: Die *Console*-Implementation des *SensorsDisplay*s. Es stellt die Daten aller Sensoren (mit Ausnahme der Kamera) in der Ausgabe-Konsole und zusätzlich die aktuellen Flugkommandos dar, welche zum Roboter geschickt werden.

Wenn das SensorsDisplay als eigener Prozeß gestartet wird, muß dieses über Shared Memory auf die darzustellenden Daten zugreifen. Somit muß der die Sensordaten (und Flugkommandos) bereitstellende Prozeß (im allgemeinen der RobotControl-Prozeß) intern die *ShmServer*-Implementation des SensorsDisplays verwenden, welche die Daten in den zugehörigen Shared Memory Ringpuffer schreibt, anstatt sie anzuzeigen. Von dort werden die Daten später vom SensorsDisplay-Prozeß ausgelesen und dargestellt.

Das Starten des SensorsDisplays als eigenen Prozeß hat, neben der nachträglichen Zuschaltmöglichkeit durch den Benutzer (analog zum ImageViewer), den Nebeneffekt, daß die Daten auch in einem Fenster mit graphischer Oberfläche angezeigt werden können. Dafür wird die *GUI*-Implementation des SensorsDisplays verwendet. Diese ist in Abb. 5.5 gezeigt.

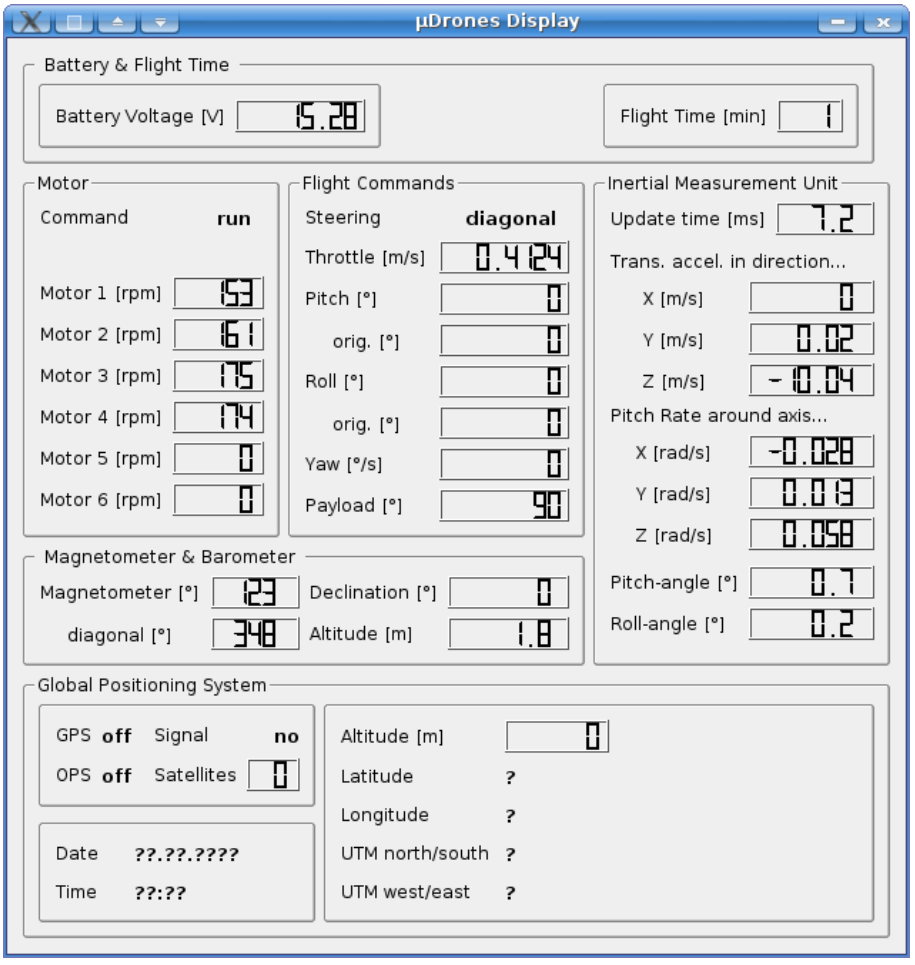

Abbildung 5.5: Die *GUI*-Implementation des *SensorsDisplay*s. Analog zur *Console*-Implementation stellt es die Daten aller Sensoren (mit Ausnahme der Kamera) dar, zusammen mit den aktuellen Flugkommandos, welche zum Roboter geschickt werden.

Um zu verhindern, daß Prozesse, welche das SensorsDisplay als Untermodul einbinden, gezwungen sind, alle zur graphischen Darstellung benötigten GUI-Bibliotheken hinzuzulinken, ist diese *GUI*-Implementation nur dann verfügbar, wenn das SensorsDisplay als ein eigener Prozeß gestartet wird. Bis auf die bessere Übersichtlichkeit bietet es gegenüber der *Console*-Implementation keine weiteren Vorteile.

5.2.4 Das *RobotControl* Modul

Das *RobotControl* Modul ist die wesentliche Verbindungsschnittstelle zwischen Roboter und Bodenstation und für die direkte Kommunikation mit dem Roboter zuständig. Es ruft Sensordaten von ihm ab und sendet im Gegenzug die Flugkommandos zu ihm zurück. Das RobotControl Modul kann nur als ein eigener Prozeß gestartet und nicht als Untermodul in andere Prozesse eingebunden werden. Stattdessen bindet es selber einige der anderen bisher vorgestellten Module und noch einige weitere als Untermodule ein. Zur besseren Übersicht wird im folgenden der Programmcode der Hauptschleife des RobotControl-Prozesses explizit dargestellt (siehe Quell. 5.1).

```
1  // cycle until CTRL-C was pressed
2  while (signal_Quit == false) {
3    grabber->release(); // release formerly obtained image
4    // get next available image (which might be the same as last time)
5    if (grabber->grab(image, GRAB_LATEST, timeout)) {
6      ... // some operations on the image
7    }
8
9    // update commands from controller(s)
10   controller->update(flightCmd);
11
12   // save flight commands to file?
13   if (save) saveCommand(flightCmd, image.sequenceNum, saveDir);
14   // load and replay flight command from file?
15   if (replay) replayCommand(flightCmd, image.sequenceNum, replayDir);
16
17   // send commands
18   sender->send(flightCmd);
19
20   // update data
21   receiver->updateData();
22   dataRobot = receiver->get_AirRobot_data();
23   dataGPS   = receiver->get_GPS_data();
24   dataIMU   = receiver->get_IMU_data();
25   // display data
26   display->update(dataRobot);
27   display->update(dataGPS);
28   display->update(dataIMU);
29   display->update(flightCmd);
30   display->show(); // display now!
31 } // while
```

Quellcode 5.1: C++-Code, welcher die Hauptschleife des *RobotControl*-Prozesses darstellt und die einzelnen intern verwendeten Module aufzeigt.

Als erstes ruft der RobotControl-Prozeß intern über die `grab`-Funktion des *ImageGrabber*-Moduls (siehe Zeile 5 in Quell. 5.1) das neueste Kamerabild ab, um es weiteren Funktionen zur Verfügung zu stellen. (Dies ist in Zeile 6 von Quell. 5.1 angedeutet.) Die genaue Funktion des ImageGrabbers wurde bereits in Abschnitt 5.2.1 beschrieben. Vornehmlich sollte der ImageGrabber hier mit seiner *ShmClient*-Implementation eingebunden werden, womit die Bilder nur aus dem Shared Memory abgerufen werden müssen. Der externe ImageGrabber-Prozeß stellt sicher, daß sich in dem Shared Memory Segment immer die aktuellsten Bilder befinden.

5.2.4.1 Das *Controller* Modul

Als nächstes werden vom *Controller*-Modul (siehe Zeile 10 in Quell. 5.1) die Flugkommandos abgerufen, welche zum Roboter geschickt werden sollen. Der Controller besitzt ebenfalls verschiedene Implementierungen: eine *Keyboard*-Implementation, welche die Steuerkommandos für den Flugroboter von der Tastatur entgegen nimmt, eine *Joystick*-Implementation, welche die Kommandos von einem Joystick bzw. Gamepad abruft, sowie ebenfalls eine *ShmClient*-Implementation, welche die Steuerkommandos von einem anderen Prozeß erhält. Anstatt nur eine einzige Implementation gleichzeitig verwenden zu können (wie z. B. beim ImageGrabber), gibt es auch eine *Handler*-Implementation, welche mehrere unterschiedliche Controller-Implementationen intern verwalten und nacheinander abrufen kann.[10]
Damit ist es möglich, Steuerkommandos aus verschiedenen Quellen (also Controller-Implementationen) abzurufen, wobei die später abgefragten Quellen Vorrang besitzen und die früheren zum Teil oder auch ganz überschrieben werden. Dies ist vor allem dann wichtig, wenn ggf. automatisch generierte Flugkommandos über die ShmClient-Implementation abgerufen werden. Damit ist es dem Benutzer möglich, falls erforderlich z. B. mit dem Joystick einzugreifen und potentiell gefährliche Flugkommandos zu überschreiben.

5.2.4.2 Das *CmdSender* Modul

In das Shared Memory Segment, aus dem der ShmClient-Controller seine Flugkommandos bezieht, sollte (aus einem anderen Prozeß heraus) mit der *ShmServer*-Implementation des *CmdSender*-Moduls geschrieben werden. In Zeile 18 in Quell. 5.1 macht die Verwendung dieser Implementation für den CmdSender jedoch nur Sinn, um die aktuellen Flugkommandos an einen anderen Prozeß weiterzugeben (der diese wiederum mit der ShmClient-Controller Implementation abrufen kann).
Auch wenn Controller und CmdSender dadurch ziemlich eng miteinander verknüpft

[10] Diese, wie auch alle anderen *Handler*-Implementationen werden automatisch aktiviert, wenn in der Konfigurationsdatei die gleichzeitige Verwendung von mehr als einer Implementation für das jeweilige Modul gewünscht wird.

sind, ist es die vornehmliche Aufgabe des CmdSenders, die Flugkommandos an den Roboter zu senden. Dies wird mit der *Device*-Implementation vorgenommen. Diese bereitet die Flugkommandos dem Kommunikationsprotokoll[11] in erforderlicher Weise auf und schreibt sie auf das jeweilige Blockdevice, welches die Verbindungsschnittstelle zur Sender-Hardware darstellt.[12] Zu Debugging-Zwecken kann dieses Blockdevice auch anstelle der seriellen Schnittstelle auf eine Datei oder Socket oder ähnliches verweisen. Dies läßt sich wiederum sehr komfortabel über einen Eintrag in der Konfigurationsdatei des RobotControl-Prozesses vornehmen. Für die Device-Implementation des CmdSenders ändert sich dadurch jedoch nichts, da auch hier die in Kapitel 3.1 vorgestellte Modularisierung, wie auch in sämtlichen bisher erwähnten und noch folgenden Modulen, vorgenommen wird.
Um beide Implementationen für das CmdSender-Modul gleichzeitig verwenden zu können, gibt es, wie beim Controller-Modul, eine *Handler*-Implementation, wobei bei dieser jedoch keinerlei Daten überschrieben, sondern nur alle Implementationen der Reihe nach abgearbeitet werden.

Nach Start des RobotControl-Prozesses mit dem passenden Kommandozeilenparameter lassen sich die Flugkommandos auch auf die Festplatte speichern (Zeile 13 in Quell. 5.1) oder von der Festplatte wieder laden (Zeile 15 in Quell. 5.1). Damit ist es also möglich, von einer Flugsequenz die gesetzten Flugkommandos zu speichern und später wieder zu laden, was allerdings nur in Verbindung mit der Speicherung bzw. Wiedergabe der zugehörigen Bildsequenz sinnvoll ist (welche somit auch gespeichert und wiedergegeben werden muß). Die einzelnen Flugkommandos werden nämlich anhand der Nummer des zugehörigen Bildes identifiziert.[13]

5.2.4.3 Das *DataReceiver* Modul

Nach der Bearbeitung der Flugkommandos, werden die Sensordaten vom *DataReceiver*-Modul abgerufen (Zeilen 21 bis 24 in Quell. 5.1). Es ist analog zum CmdSender aufgebaut und besitzt ebenso wie dieser eine *Handler*-Implementation zur gleichzeitigen Verwendung von mehr als einer DataReceiver-Implementation. Es greift genau

[11] Dabei handelt es sich um ein von der Herstellerfirma des Flugroboters entwickeltes eigenes Format.

[12] Aufgrund von gesetzlichen Bestimmungen können die Daten nur mit geringerer Geschwindigkeit von der Sender-Hardware zum Roboter gefunkt werden, als sie vom Programm zu ihr geschickt werden können. Um einen möglichen Pufferüberlauf zu verhindern, werden die Flugkommandos deshalb intern an einen eigenen Thread übergeben, welcher die Daten nur circa alle 25 ms auf das Blockdevice schreibt.

[13] An dieser Stelle ist noch Raum gelassen für Verbesserungen. Man könnte z. B. die gespeicherten Flugkommandos zusätzlich mit einer Timestamp versehen, welche die genaue zeitliche Abfolge angibt.

wie der CmdSender mit seiner *Device*-Implementation auf ein Blockdevice zu, welches die Verbindung zur Empfänger-Hardware darstellt. Auch dieses Blockdevice kann je nach Eintrag in der Konfigurationsdatei anstelle der seriellen Schnittstelle auf eine Datei oder Socket oder ähnlichem verweisen. Damit ist es z. B. möglich, dem Programm künstlich erzeugte Sensordaten zu übergeben.[14]
Außer diesen beiden[15] gibt es bisher noch keine weiteren Implementationen für den DataReceiver. Jedoch ist mit der Handler-Implementation die Möglichkeit für die Verwendung weiterer zusätzlicher DataReceiver-Implementationen gegeben.

Zum Abschluß der Hauptschleife des RobotControl-Prozesses werden (in Zeile 30 in Quell. 5.1) die aktualisierten Sensordaten noch an das zuvor schon in Abschnitt 5.2.3 beschriebene *SensorsDisplay*-Modul übergeben, welches die Daten darstellt. Empfohlen wird hier die Verwendung der *ShmServer*-Implementation für das SensorsDisplay, um eine zusätzlich zuschaltbare Darstellung zu erhalten, ohne gezwungen zu sein, die Sensordaten ständig darstellen zu müssen.

5.2.5 Das *ImageProcessor* Modul

Neben dem RobotControl-Modul ist das *ImageProcessor*-Modul das andere große Modul. In ihm befindet sich die „Intelligenz“, denn es ist vor allem für die Auswertung der Bilddaten zuständig. Grundsätzlich ist es möglich, das ImageProcessor-Modul innerhalb des RobotControl-Prozesses anzusiedeln. Schließlich paßt es wunderbar in den in Quell. 5.1 in Zeile 6 nur angedeuteten Bildverarbeitungsbereich. Jedoch wurde aus Effizienzgründen davon abgesehen, da damit die Zeit eines Durchlaufs der Hauptschleife im RobotControl-Prozeß deutlich erhöht würde. Zur Steuerung des Roboters wird jedoch eine geringe Zykluszeit benötigt, damit der Kontakt zum Flugroboter nicht abbricht und eine robuste Steuerung des Roboter weiterhin möglich ist.[16]

Für den besseren Überblick werden der Programmcode der Hauptschleife des ImageProcessor-Prozesses (siehe Quell. 5.2) und die einzelnen Module dargestellt, allerdings nicht im Detail besprochen.

[14] Für eine zukünftige Verwendung mit einem Simulator ist dies eine interessante Option.

[15] Wenn man von den obligatorischen (für alle Module existierenden) *Dummy*-Implementationen absieht, welche keine weiteren Funktionen übernehmen.

[16] Empirische Messungen der Sendeleistung ergaben, daß die beste Zykluszeit zum Senden der Daten bei circa 25 ms liegt. (Siehe auch Fußnote 12 in Abschnitt 5.2.4.2.) Das Erzeugen von neuen Steuerkommandos (ob automatisch oder von der Handsteuerung) mit einer kürzeren Zykluszeit macht insofern keinen wirklichen Unterschied. Jedoch werden alle Flugkommandos, die mit einer höheren Zykluszeit vom Controller abgefragt werden, somit verspätet zum Roboter gesendet, was wiederum mit einer verzögerten Reaktion und damit einer ungenauen Steuerung des Roboters einhergeht.

```
1  // cycle until CTRL-C was pressed
2  while (signal_Quit == false) {
3    // fetch the next image
4    images[index] = preprocessor->fetch();
5
6    // calculate optical flow
7    flowCalculator->calculate(images[1-index], images[index]);
8
9    // get formerly set flight commands and forward to flow-evaluator
10   controller->update(flightCmd);
11   flowEvaluator->setFlightCommands(flightCmd);
12   // query and evaluate flow data
13   flowEvaluator->evaluate(flowCalculator->get_opticalFlow());
14   // get suggested flight commands and write it to shared memory
15   flowEvaluator->getFlightCommands(flightCmd);
16   sender->send(flightCmd);
17
18   // output optical flow image to shared memory for debugging?
19   if (output_opticalFlowImage) {
20     // show vectors in newer of the two images
21     flowEvaluator->outputFlowImage(images[index]);
22   }
23   // point to older image (which will be overwritten next cycle)
24   index = 1 - index;
25   preprocessor->release();
26 } // while
```

Quellcode 5.2: C++-Code, der die Hauptschleife des *ImageProcessor*-Prozesses darstellt und die einzelnen intern verwendeten Module aufzeigt.

In Zeile 4 wird ein neues Bild vom ImageGrabber abgerufen. Allerdings erfolgt dies implizit durch das *ImagePreprocessor* Modul, welches das abgerufene Bild noch in eine „Container"-Struktur einbindet. Dieser Container ist nur ein „Wrapper" um das Bild, welcher diverse Bildbearbeitungs- und Filterungsmethoden zur Verfügung stellt.[17]

Anschließend wird in Zeile 7 aus dem aktuellen und dem zuvor abgerufenen Bild der optische Fluß berechnet. Je nachdem, welche Implementation des *OpticalFlow-Calculator*-Moduls verwendet wird, wird der optische Fluß dabei unterschiedlich berechnet. Als mögliche Berechnungsalgorithmen für die einzelnen Implementationen kommen dabei die in Kapitel 4.1 erwähnten Algorithmen zum Einsatz.

[17] Z. B. wird für die Berechnung des optischen Flusses nur der Grauwertanteil des Bildes benötigt, und der Container stellt sicher, daß die zugehörigen Algorithmen (wie der Gauß'sche Weichzeichnungsfilter etc.) nur auf diesen Bilddaten arbeiten.

Nach der Berechnung des optischen Flusses werden in Zeile 10 die zuletzt zum Roboter geschickten Flugkommandos (über die *ShmClient*-Implementation des *Controller*-Moduls) abgerufen und in den folgenden Zeilen zusammen mit den optischen Flußdaten vom *ImageFlowEvaluator*-Modul ausgewertet. Die aus der Auswertung resultierenden neuen Flugkommandos werden dann in Zeile 16 zum Roboter geschickt (bzw. richtigerweise mit der *ShmServer*-Implementation des *CmdSender*-Moduls via Shared Memory an den RobotControl Prozeß übergeben, welcher sie dann an den Roboter weitersendet).
Zum gegenwärtigen Zeitpunkt ist der ImageFlowEvaluator noch nicht in der Lage, sinnvolle Flugkommandos zu generieren, weshalb die von ihm zurückgeschickten Kommandos bisher noch den unmodifizierten, zuvor vom Controller an den ImageFlowEvaluator übergebenen, Flugkommandos entsprechen. Der in Abschnitt 4.2 beschriebene *Kanatani*-Algorithmus zur Eigenbewegungsbestimmung aus optischen Flußdaten sowie das in den folgenden Abschnitten beschriebene Verfahren zur Verbesserung dieser Eigenbewegungsschätzung findet ebenfalls innerhalb des ImageFlowEvaluator-Moduls Anwendung. Allerdings wird an dieser Stelle nicht weiter darauf eingegangen.

Im Anschluß an die Auswertung der optischen Flußdaten kann für Debugging-Zwecke (bei Start des ImageProcessors mit dem entsprechenden Parameter) noch das aktuelle Bild zusammen mit den eingezeichneten Flußvektoren ausgegeben werden (Zeile 21 in Quell. 5.2). Dafür wird das Bild in ein Shared Memory Segment geschrieben (analog zu der Vorgehensweise des ImageGrabbers) und kann von einem weiteren ImageViewer abgerufen und angezeigt[18] werden.

Zum Ende der Hauptschleife des ImageProcessors wird noch der Index des Arrays, in dem die Zeiger auf die beiden, zur optischen Flußberechnung verwendeten Bilder enthalten sind, verschoben, so daß er nun auf das Feld mit dem Zeiger für das ältere der beiden Bilder verweist (Zeile 24 in Quell. 5.2). Danach wird (in Zeile 25) das ältere Bild zusammen mit der sie umschließenden Container-Struktur wieder freigegeben, so daß es im nächsten Schleifendurchlauf mit dem dann neuen Bild überschrieben werden kann.

[18]Natürlich muß dafür dessen zugehörige Konfigurationsdatei entsprechend angepaßt werden, damit sich der ImageViewer auch zum richtigen Shared Memory Segment verbindet.

6. Bestimmung der Eigenbewegung des Flugroboters

Um die Eigenbewegung des Flugroboters bestimmen zu können, wird der in Kapitel 4.2 beschriebene *Kanatani*-Algorithmus verwendet. Dieser berechnet die Bewegung der Kamera (und damit des Roboters) aus den optischen Flußdaten. Es hat sich im Laufe dieser Arbeit jedoch gezeigt, daß diese berechnete Bewegung in der Praxis, vor allem mit aus realen Bildern berechneten optischen Flüssen, nicht immer ganz zuverlässig ist.
Insbesondere die berechnete Translationsrichtung schwankt von einem zum nächsten Berechnungsschritt sehr stark und zudem auch nicht nur um die korrekte Translationsrichtung, sondern weist des öfteren auch in eine gänzlich falsche Richtung. Dieser Effekt hängt in großem Maße vom Öffnungswinkel der verwendeten Kamera ab und tritt verstärkt dann auf, wenn sie nur einen geringen Öffnungswinkel hat. Die Kamera des bei dieser Arbeit verwendeten Flugroboters fällt ebenfalls in diese Kategorie.

Ein erster Schritt zur Verbesserung der Bewegungschätzung wäre, die Berechnung des optischen Flusses zu verbessern und fehlerhafte Flüsse zu eliminieren. Dies ist ein grundsätzlich vernünftiger Weg, wird damit die Fehlerwahrscheinlichkeit des *Kanatani*-Algorithmus doch von vornherein herabgesetzt. Während der Erstellung der vorliegenden Arbeit wurde dies auch gemacht und die Ergebnisse daraus flossen zum Teil schon in diese Arbeit mit ein. Jedoch schwankt die vom *Kanatani*-Algorithmus berechnete Translationsrichtung noch immer, und am Problem des zu geringen Kameraöffnungswinkels konnte es auch nichts ändern. Somit kann eine inkorrekte Be-

wegungsbestimmung damit allein nicht ganz ausgeschlossen werden, weshalb in der vorliegenden Arbeit ein anderer Ansatz verfolgt wurde.

Da der Einsatz einer Rundumsichtkamera nicht möglich war und zudem eine allgemeinere Lösung gesucht wird, welche auch für Kameras mit geringerem Öffnungswinkel funktioniert, wurde stattdessen versucht, die vom *Kanatani*-Algorithmus berechnete Bewegung *nachträglich* zu verbessern, indem die zuletzt zum Roboter gesandten Flugkommandos zur Korrektur herangezogen wurden. Um dieses Vorhaben etwas zu vereinfachen, beschränkt sich die Korrektur auf den Fall einer Translationsbewegung bei konstanter Höhe.

Im folgenden wird nun dargestellt, wie die vom *Kanatani*-Algorithmus gelieferte Schätzung für die Translation in einer Ebene unter Zuhilfenahme eines *Erweiterten Kalman-Filters*, welcher die Flugkommandos zur Berechnung intern verwendet, verbessert und stabilisiert werden kann.

Hinweis: Es sei an dieser Stelle darauf hingewiesen, daß die im folgenden verwendeten Koordinatensysteme (für eine bessere Übereinstimmung mit dem im *Kanatani*-Algorithmus verwendeten Koordinatensystem) von der in der Mathematik üblichen Konvention abweichen. Bei diesen handelt es sich um normale rechtshändige Koordinatensysteme, wobei jedoch die x-Achse nach rechts, die z-Achse nach vorne und die y-Achse nach unten weist. Somit bilden x- und z-Achse eine horizontale Ebene, während die y-Achse die (umgekehrte) Hochachse darstellt.
Auch die in der Aeronautik üblichen Konventionen für Rollen, Nicken und Gieren wurden damit notwendigerweise abgewandelt. Der Roll-Winkel Φ läuft somit um die z-Achse, der Nick-Winkel Θ um die x-Achse und der Gier-Winkel Ψ um die y-Achse. (Siehe auch Abb. 6.1.)

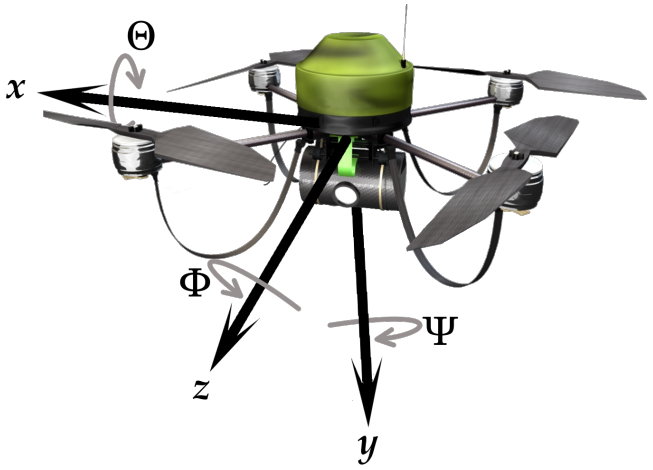

Abbildung 6.1: Die im weiteren verwendete Konvention für die Koordinatenachsen und die zugehörigen Steuerwinkel beim Flugroboter. (Verändert nach [39])

6.1 Stabilisierung der Eigenbewegungsschätzung mit einem EKF

Zur Verbesserung und Stabilisierung der Eigenbewegungsschätzung wird vom vereinfachten Fall der Bewegung des Flugroboters in einer Ebene ausgegangen. Der Roboter verändert nach dieser Annahme seine Flughöhe also nicht. Alle aktiven Änderungen der Bewegung beruhen somit nur auf den drei Winkeln zum Rollen (Φ), Nicken (Θ) und Gieren (Ψ), während der zusätzliche Schub (Λ), der die Flughöhe des Roboters ändert, als Null angenommen wird.

6.1.1 Zustandsmodell

Das folgende Zustandsmodell bestimmt die Translation des Roboters. Diese Translation wird mit dem Geschwindigkeitsvektor $\vec{\mathbf{v}} \in \mathbb{R}^3$ beschrieben, der sich auf ein im Zentrum des Roboters festgemachtes Lage-invariantes Koordinatensystem bezieht, welches des weiteren als „*Roboter-Referenz*“-Koordinatensystem oder kurz als *RR-KS* bezeichnet wird. Es entspricht dem eigentlichen Roboter-Koordinatensystem im Ruhezustand, also für den Fall, daß $\Phi = 0$ und $\Theta = 0$ gilt. Ψ ist für die Neigung des Roboters unerheblich. Demzufolge ist das *RR-KS* so definiert, daß es sich mit dem Roboter beim Gieren mitdreht. Damit sind auch bei $\Psi \neq 0$ dessen z-Achse immer nach vorne, y-Achse immer nach unten und x-Achse immer nach rechts gerichtet (jeweils bezogen auf das Roboter-Koordinatensystem im Ruhezustand, Abb. 6.2).

Das Zustandsmodell läßt sich aus der Kraftgleichung für die Bewegung herleiten. Grundsätzlich gilt, daß sich die Gesamtkraft F_{net} aus einer Vortriebskraft F_{thrust} und einer (entgegengesetzten) Luftwiderstandskraft F_{drag} zusammensetzt:

$$F_{net} = F_{thrust} + F_{drag} \quad . \tag{6.1}$$

Mit Hilfe des zweiten Newton'schen Axioms, welches besagt, daß die Kraft gleich der Impulsänderung pro Zeit ist, also $F = m\,\dot{p}$, der Impuls wiederum als Masse mal Geschwindigkeit, $p = m\,\mathrm{v}$, definiert ist, womit sich die Kraft also zu Masse mal Beschleunigung, $F = m\,\dot{\mathrm{v}}$, ergibt[1], läßt sich nun Gl. 6.1 in eine Gleichung der Beschleunigung umformulieren, indem man durch die Masse des Flugroboters dividiert:

$$\dot{\mathrm{v}}_{net} = \dot{\mathrm{v}}_{thrust} + \dot{\mathrm{v}}_{drag} \quad . \tag{6.2}$$

Um nun aus Gl. 6.2 eine Geschwindigkeit zu erhalten, müssen beide Seiten der Gleichung mit dem betrachteten Zeitintervall multipliziert werden. Die resultierende Geschwindigkeit ergibt jedoch nur dann die gültige Gesamtgeschwindigkeit, wenn

[1] Genauer folgt für die Kraft mittels der Produktregel $F = m\,\dot{\mathrm{v}} + \dot{m}\,\mathrm{v}$. Jedoch vereinfacht sich dies für Systeme mit konstanter Masse, wie z. B. im Fall des verwendeten Flugroboters, zu $F = m\,\dot{\mathrm{v}}$.

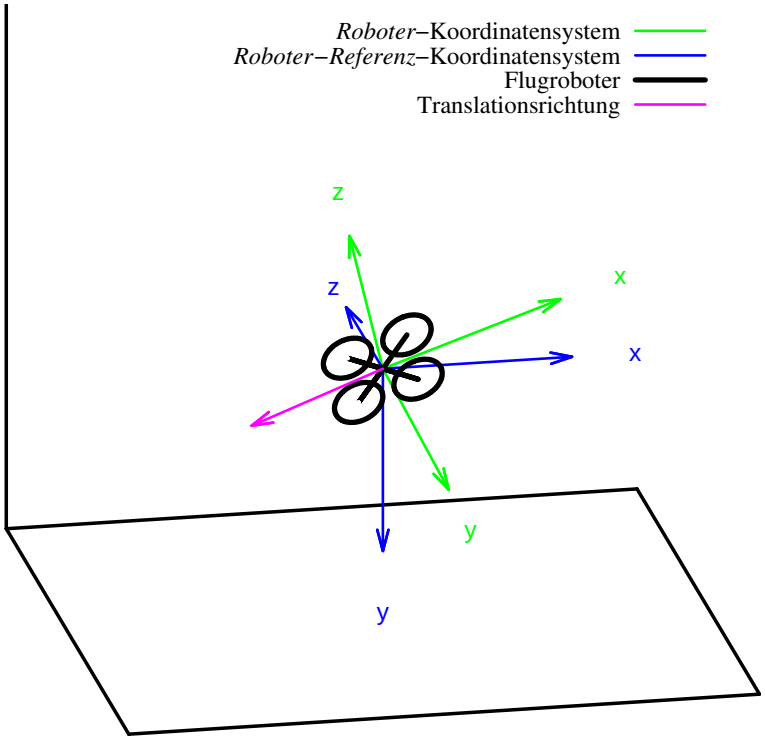

Abbildung 6.2: Schema des Roboters in geneigter Stellung zusammen mit den zugehörigen Koordinatensystemen und der aus der Neigung resultierenden Translationsrichtung. Bei Neigung mit Roll- und Nick-Winkel vollführt der Roboter eine Translation (pink), die entweder in Koordinaten des mitgeneigten Roboter-Koordinatensystems (grün) oder in Koordinaten des Lage-invarianten *Roboter-Referenz*-Koordinatensystems (blau) angegeben werden kann. Die gezeigte Neigung resultiert aus einem Nick-Winkel von 25° und einem Roll-Winkel von -25°.

beide Beschleunigungen $\dot{\mathrm{v}}_{thrust}$ und $\dot{\mathrm{v}}_{drag}$ über die gesamte Zeitdauer konstant gehalten werden.
Für die Vortriebsbeschleunigung ist dies mit Einschränkungen möglich. Man kann jedoch nicht davon ausgehen, daß die Luftwiderstandskraft F_{drag} bei unterschiedlichen Geschwindigkeiten konstant bleibt. Deshalb muß das betrachtete Zeitintervall infinitesimal klein sein, womit die resultierende Geschwindigkeit somit nur den Geschwindigkeitszuwachs bzw. -abfall v_{change} beschreibt. In der Praxis ist die Messung eines solch kleinen Zeitintervalls natürlich nicht möglich, weshalb es nur näherungsweise mit Δt angegeben werden kann. Die innerhalb eines solchen Zeitintervalls auftretende Änderungsgeschwindigkeit lautet somit:

$$\mathrm{v}_{change} = \dot{\mathrm{v}}_{net} \cdot \Delta t = \dot{\mathrm{v}}_{thrust} \cdot \Delta t + \dot{\mathrm{v}}_{drag} \cdot \Delta t = \mathrm{v}_{thrust} + \mathrm{v}_{drag} \ . \tag{6.3}$$

Um schließlich die Gesamtgeschwindigkeit zu erhalten, muß die alte Geschwindigkeit, die sich bis direkt vor den betrachteten Zeitpunkt (bzw. vor das betrachtete Zeitintervall) ergeben hat, noch hinzuaddiert werden:

$$\mathrm{v}_{total} = \mathrm{v}_{old} + \mathrm{v}_{change} = \mathrm{v}_{old} + \mathrm{v}_{thrust} + \mathrm{v}_{drag} \quad . \tag{6.4}$$

Für den Fall, daß der Luftwiderstand F_{drag} betraglich gleich der Vortriebskraft F_{thrust} ist, jedoch mit umgekehrtem Vorzeichen, heben sich diese beiden Kräfte und damit auch die daraus resultierenden Geschwindigkeiten v_{thrust} und v_{drag} gegenseitig auf. Damit bleibt v_{total} konstant und entspricht somit der Maximalgeschwindigkeit, die sich aus gegebener Vortriebskraft F_{thrust} ergibt.

Für das Zustandsmodell wird davon ausgegangen, daß sich der Geschwindigkeitsvektor $\vec{\mathbf{v}}_k$ zum Zeitpunkt k aus drei Teilen additiv wie folgt zusammensetzt:

$$\vec{\mathbf{v}}_k = \vec{\mathbf{v}}_{k-1\,,\,k} + \vec{\mathbf{v}}_{drag\,,\,k} + \vec{\mathbf{v}}_{thrust\,,\,k} \quad . \tag{6.5}$$

$\vec{\mathbf{v}}_{k-1}$ bezeichnet logischerweise die alte Geschwindigkeit des Roboters zum vorherigen Zeitpunkt k–1. Mit $\vec{\mathbf{v}}_{drag}$ hingegen ist die Geschwindigkeitsänderung gemeint, welche sich aufgrund des durch die Bewegung mit der alten Geschwindigkeit $\vec{\mathbf{v}}_{k-1}$ erzeugten Gegenwinds ergibt. $\vec{\mathbf{v}}_{thrust}$ ist der zusätzliche Geschwindigkeitsvortrieb des Roboters vom Zeitschritt k–1 zum Zeitschritt k. Der zusätzliche zweite Index k bedeutet, daß die jeweilige Geschwindigkeit in Koordinaten des *RR-KS* des aktuellen Zeitschritts k ausgedrückt ist.

6.1.1.1 Koordinatentransformation von $\vec{\mathbf{v}}_{k-1}$

Da das *RR-KS* zu jedem Zeitschritt zusammen mit dem Roboter giert, muß die im vorherigen Zeitschritt berechnete Geschwindigkeit zuerst noch in Koordinaten des aktuellen *RR-KS* transformiert werden. In Gl. 6.5 wurde dies durch den zweiten zusätzlichen Index angedeutet.
Somit wird die alte Geschwindigkeit $\vec{\mathbf{v}}_{k-1}$, welche in Koordinaten des *RR-KS* des vorherigen Zeitschritts k–1 formuliert ist, folgendermaßen in Koordinaten des *RR-KS* des aktuellen Zeitschritts k umgerechnet:

$$\vec{\mathbf{v}}_{k-1\,,\,k} = \mathcal{R}_y\,(\Psi) \cdot \vec{\mathbf{v}}_{k-1} \tag{6.6}$$

$$= \begin{pmatrix} \cos(\Psi) & 0 & -\sin(\Psi) \\ 0 & 1 & 0 \\ \sin(\Psi) & 0 & \cos(\Psi) \end{pmatrix} \vec{\mathbf{v}}_{k-1} \tag{6.7}$$

$$= \begin{pmatrix} (\mathrm{v}_x)_{k-1}\,\cos(\Psi) - (\mathrm{v}_z)_{k-1}\,\sin(\Psi) \\ (\mathrm{v}_y)_{k-1} \\ (\mathrm{v}_x)_{k-1}\,\sin(\Psi) + (\mathrm{v}_z)_{k-1}\,\cos(\Psi) \end{pmatrix} \quad . \tag{6.8}$$

Die Rotationsmatrix $\mathcal{R}_y(\alpha)$ dreht ein Koordinatensystem mit dem Winkel α um seine y-Achse im Raum und transformiert so jeden Vektor, der von rechts daran multipliziert wird ebenfalls in Koordinaten des neuen Koordinatensystems. Da sich der Flugroboter und damit sein *RR-KS* vom vorherigen zum aktuellen Zeitschritt mit dem Winkel Ψ um seine y-Achse gedreht hat, wird die Geschwindigkeit $\vec{\mathbf{v}}_{k-1}$ durch Multiplikation mit $\mathcal{R}_y(\Psi)$ somit in Koordinaten des aktuellen *RR-KS* im Zeitschritt k ausgedrückt, also zu $\vec{\mathbf{v}}_{k-1\,,k}$ transformiert.

6.1.1.2 Modell für die Luftwiderstandsgeschwindigkeit $\vec{\mathbf{v}}_{drag}$

Physikalisch ist der Luftwiderstand als Kraft mit folgender Formel definiert [40]:

$$F_{drag} = c_{drag} \cdot A \cdot \frac{\rho}{2} \cdot \mathrm{v}^2 \ . \tag{6.9}$$

Dabei ist A die quer zur Bewegungsrichtung projizierte Fläche (in m^2) des durch die Luft bewegten Gegenstandes, auf die die (stehende) Luft wirkt, v die Geschwindigkeit relativ zur Luft[2] (in m/s), mit der sich dieser Gegenstand durch die Luft bewegt, c_{drag} ein dimensionsloser Luftwiderstandsbeiwert, der je nach Form und Oberflächenbeschaffenheit des Gegenstands unterschiedlich sein kann, und ρ die Dichte der Luft (in $\mathrm{kg/m}^3$), die je nach Luftdruck und Temperatur ebenfalls variiert.
Die Kraft F_{drag} besitzt die Einheit Newton ($\mathrm{kg \cdot m/s}^2$) und kann auch als Produkt aus Masse und Beschleunigung dargestellt werden.[3] Um aus F_{drag} eine Geschwindigkeit zu erhalten, muß sie also noch durch die Masse des durch die Luft bewegten Gegenstands geteilt und mit dem betrachteten Zeitintervall multipliziert werden. Um schließlich eine entgegen der Bewegung des Gegenstandes gerichtete Geschwindigkeit zu erhalten, muß zudem noch der mit negativem Vorzeichen versehene Einheitsvektor $\frac{\vec{\mathbf{v}}}{\mathrm{v}}$ des zu v gehörenden Geschwindigkeitsvektors $\vec{\mathbf{v}}$ dazu multipliziert werden. Somit ergibt sich Gl. 6.9 zu:

$$\vec{\mathbf{v}}_{drag} = -c_{drag} \cdot A \cdot \frac{\rho}{2m} \cdot \mathrm{v} \cdot \vec{\mathbf{v}} \cdot \Delta t \ . \tag{6.10}$$

Leider läßt sich für den Flugroboter der Luftwiderstandsbeiwert c_{drag} und ebenso die Projektionsfläche A nicht ohne weiteres bestimmen. A ist in Wirklichkeit eine Funktion, die von Φ und Θ abhängt[4], denn je nach Neigung des Roboters verändert

[2] Die Geschwindigkeit v ist also relativ zur eigentlichen Windgeschwindigkeit zu sehen. Sollte Wind wehen, dann müßte die ursprünglich gesetzte Geschwindigkeit entsprechend angepaßt werden, um v zu erhalten: vergrößert bei Rückenwind, verkleinert bei Gegenwind.

[3] Siehe dazu die Ausführungen im vorherigen Abschnitt 6.1.1 über den Zusammenhang von Kraft und Impulsänderung.

[4] Zusätzlich hängt A sogar noch vom Schub Λ ab, welcher die Höhe des Roboters ändert. Da hier aber von einer Bewegung des Roboters in einer Ebene ausgegangen wird, braucht Λ nicht weiter berücksichtigt zu werden.

sich die Projektionsfläche quer zur Flugrichtung. c_{drag} hängt von der Form der Projektionsfläche ab (somit also auch von Φ und Θ) und kann ohne die Verwendung eines Windkanals nicht exakt ermittelt werden.
Die genaue Luftdichte ρ ließe sich vor jedem Flug ermitteln, jedoch ist dies für den praktischen Einsatz kaum praktikabel. Nur die Masse des Flugroboters kann vorab bestimmt und mit 0,903 kg direkt eingegeben werden.

Da die Bestimmung der erwähnten Unbekannten also schwierig ist, können sie zunächst zusammengefaßt und durch eine (von den Neigungswinkeln abhängende) Konstante ersetzt werden, deren Wert in einem späteren Abschnitt dieser Arbeit näherungsweise bestimmt wird. Zusätzlich wird der (allerdings bekannte) Faktor $\frac{1}{2\,m}$ aus Gl. 6.10 in diese Konstante hineingezogen. Für diese Konstante ξ gilt dann:

$$\xi = \frac{1}{2\,m} \cdot c_{drag} \cdot A \cdot \rho \ . \tag{6.11}$$

Für den Luftwiderstand aus Gl. 6.5 ergibt sich hieraus zusammen mit Gl. 6.10:

$$\vec{\mathbf{v}}_{drag\,,k} = -\xi \cdot \Delta t \cdot \mathrm{v}_{k-1} \cdot \vec{\mathbf{v}}_{k-1\,,k} \ . \tag{6.12}$$

6.1.1.3 Modell für die Vortriebsgeschwindigkeit $\vec{\mathbf{v}}_{thrust}$

Die zusätzliche Vortriebsgeschwindigkeit $\vec{\mathbf{v}}_{thrust}$ ist unabhängig von der ursprünglichen Geschwindigkeit und setzt sich aus der Beschleunigung $\vec{\mathbf{a}}_{thrust}$ und dem verstrichenen Zeitintervall Δt zusammen:

$$\vec{\mathbf{v}}_{thrust} = \vec{\mathbf{a}}_{thrust} \cdot \Delta t \ . \tag{6.13}$$

Diese Beschleunigung $\vec{\mathbf{a}}_{thrust}$ wiederum beruht gänzlich auf den Neigungswinkeln Φ und Θ.

Die von den vier Rotoren des Roboters erzeugte Auftriebskraft muß bei (angenommenen) $\Lambda = 0$ die Höhe halten, also die Erdanziehung kompensieren.[5] Wenn nun entweder Φ oder Θ oder beide ungleich Null sind, so heißt das, daß die erzeugte Beschleunigung nicht mehr senkrecht nach oben, sondern schräg nach oben gerichtet ist. Damit strömt die durch die Rotoren gesaugte Luft nicht nur nach unten, sondern auch nach hinten und bewirkt eine Vortriebsbeschleunigung. Gleichzeit kann der erzeugte Auftrieb der Rotoren die Erdanziehung nicht mehr gänzlich kompensieren und der Flugroboter beginnt zu sinken. Der bei dieser Arbeit verwendete Flugroboter ist allerdings so geregelt, daß er automatisch die Drehzahl der Rotoren anpaßt, so daß der Roboter trotz des Vortriebs die gleiche Höhe beibehält.[6]

[5] Der Fall $\Lambda \neq 0$ ist zur Berechnung um einiges komplizierter, da nicht genug Informationen über die interne Regelungstechnik des verwendeten Flugroboters bekannt sind.

[6] Das fehlende Wissen über die genauen Vorgänge dieser Regelung ist übrigens auch der Grund, weshalb sich eine Geschwindigkeitschätzung in drei Dimensionen als sehr schwierig herausstellt.

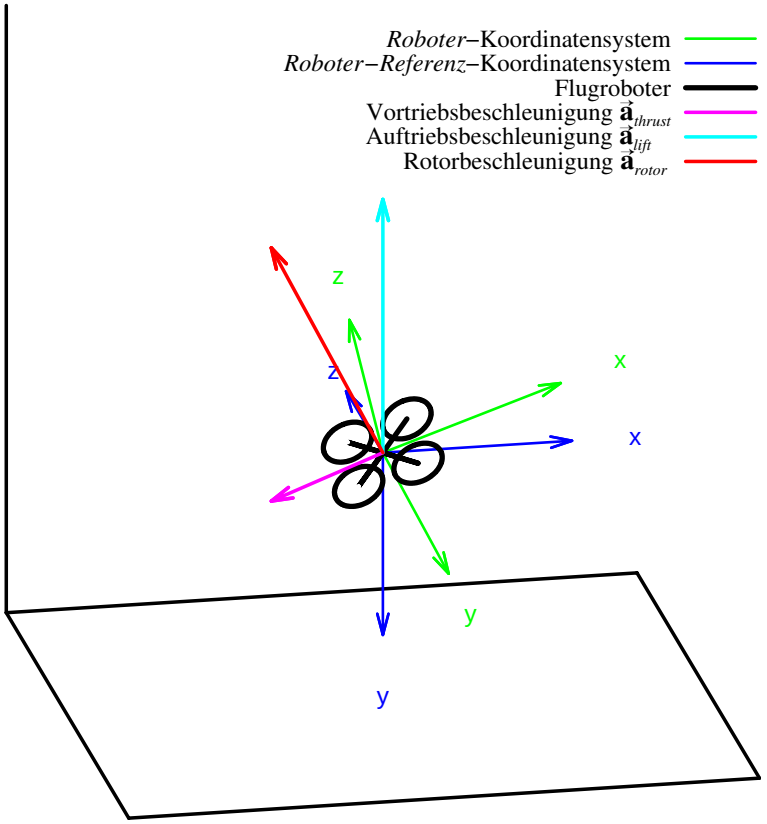

Abbildung 6.3: Die gleiche Darstellung des Roboters wie in Abb. 6.2 zusätzlich mit den auftretenden Beschleunigungen. Die von den Rotoren erzeugte Beschleunigung $\vec{\mathbf{a}}_{rotor}$ (rot) ist senkrecht zur Rotorebene angeordnet. Sie setzt sich zusammen aus der Auftriebsbeschleunigung $\vec{\mathbf{a}}_{lift}$ (hellblau) senkrecht zum Boden und der Vortriebs- bzw. Translationsbeschleunigung $\vec{\mathbf{a}}_{thrust}$ (pink) parallel zum Boden.

D. h. also, die zur Rotorebene des Roboters senkrecht nach oben (also in Roboter-Koordinaten in Richtung des Einheitsvektors $(0,-1,0)^\mathsf{T}$) verlaufende Rotorbeschleunigung $\vec{\mathbf{a}}_{rotor}$ spaltet sich auf in eine Auftriebsbeschleunigung $\vec{\mathbf{a}}_{lift}$ und die gesuchte Vortriebs- bzw. Translationsbeschleunigung $\vec{\mathbf{a}}_{thrust}$ (siehe Abb. 6.3).

Da der Roboter seine Höhe hält, gilt für die Auftriebsbeschleunigung in Koordinaten des *RR-KS*

$$\vec{\mathbf{a}}_{lift} = \mathrm{g} \cdot \begin{pmatrix} 0 \\ -1 \\ 0 \end{pmatrix} \quad , \tag{6.14}$$

wobei $\mathrm{g} \approx 9,81\ \mathrm{m/s^2}$ die Erdbeschleunigung darstellt. Die Richtung der Rotorbeschleunigung $\vec{\mathbf{a}}_{rotor}$ in *Roboter-Referenz*-Koordinaten läßt sich aus dem Einheitsvektor $(0,-1,0)^\mathsf{T}$ in Roboter-Koordinaten berechnen. Dazu muß dieser nur den Rota-

tionen, welche durch Φ und Θ hervorgerufen werden, in umgekehrter Reihenfolge[7] und Richtung unterzogen werden:

$$\frac{\vec{\mathbf{a}}_{rotor}}{\|\vec{\mathbf{a}}_{rotor}\|} = \mathcal{R}_x(-\Theta) \cdot \mathcal{R}_z(-\Phi) \cdot \begin{pmatrix} 0 \\ -1 \\ 0 \end{pmatrix} \tag{6.15}$$

$$= \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos(-\Theta) & \sin(-\Theta) \\ 0 & -\sin(-\Theta) & \cos(-\Theta) \end{pmatrix} \cdot \begin{pmatrix} \cos(-\Phi) & \sin(-\Phi) & 0 \\ -\sin(-\Phi) & \cos(-\Phi) & 0 \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 0 \\ -1 \\ 0 \end{pmatrix}$$

$$= \begin{pmatrix} -\sin(-\Phi) \\ -\cos(-\Theta)\cos(-\Phi) \\ \sin(-\Theta)\cos(-\Phi) \end{pmatrix} = \begin{pmatrix} \sin(\Phi) \\ -\cos(\Theta)\cos(\Phi) \\ -\sin(\Theta)\cos(\Phi) \end{pmatrix} \quad . \tag{6.16}$$

Der letzte Umformungsschritt bedient sich der Tatsache, daß $\sin(-\alpha) = -\sin(\alpha)$ und $\cos(-\alpha) = \cos(\alpha)$ sind.

Die Rotorbeschleunigung $\vec{\mathbf{a}}_{rotor}$ läßt sich nun mit ein paar Überlegungen bestimmen:

- Die y-Komponente von $\vec{\mathbf{a}}_{rotor}$ muß der y-Komponente von $\vec{\mathbf{a}}_{lift}$ entsprechen:

$$(\vec{\mathbf{a}}_{rotor})_y = (\vec{\mathbf{a}}_{lift})_y = -\mathrm{g} \quad . \tag{6.17}$$

- Mit dem Strahlensatz läßt sich folgern, daß das Verhältnis der x-Komponente von $\vec{\mathbf{a}}_{rotor}$ zur x-Komponente des zugehörigen Einheitsvektors $\frac{\vec{\mathbf{a}}_{rotor}}{\|\vec{\mathbf{a}}_{rotor}\|}$ gleich dem Verhältnis der y-Komponenten dieser beiden Vektoren sein muß (siehe auch Abb. 6.4):

$$\frac{(\vec{\mathbf{a}}_{rotor})_x}{\left(\frac{\vec{\mathbf{a}}_{rotor}}{\|\vec{\mathbf{a}}_{rotor}\|}\right)_x} = \frac{(\vec{\mathbf{a}}_{rotor})_y}{\left(\frac{\vec{\mathbf{a}}_{rotor}}{\|\vec{\mathbf{a}}_{rotor}\|}\right)_y} \quad . \tag{6.18}$$

  Damit ergibt sich die x-Komponente von $\vec{\mathbf{a}}_{rotor}$ zu:

$$(\vec{\mathbf{a}}_{rotor})_x = \frac{(\vec{\mathbf{a}}_{rotor})_y \cdot \left(\frac{\vec{\mathbf{a}}_{rotor}}{\|\vec{\mathbf{a}}_{rotor}\|}\right)_x}{\left(\frac{\vec{\mathbf{a}}_{rotor}}{\|\vec{\mathbf{a}}_{rotor}\|}\right)_y} = \mathrm{g} \cdot \frac{\tan(\Phi)}{\cos(\Theta)} \quad . \tag{6.19}$$

- Analog zum vorherigen Punkt läßt sich die z-Komponente von $\vec{\mathbf{a}}_{rotor}$ mit dem Strahlensatz bestimmen:

$$(\vec{\mathbf{a}}_{rotor})_z = \frac{(\vec{\mathbf{a}}_{rotor})_y \cdot \left(\frac{\vec{\mathbf{a}}_{rotor}}{\|\vec{\mathbf{a}}_{rotor}\|}\right)_z}{\left(\frac{\vec{\mathbf{a}}_{rotor}}{\|\vec{\mathbf{a}}_{rotor}\|}\right)_y} = -\mathrm{g} \cdot \tan(\Theta) \quad . \tag{6.20}$$

[7] Die allgemeine Konvention für die drei Orientierungswinkel besagt, daß zur mathematischen Berechnung als erstes das Gieren, dann das Nicken und zuletzt das Rollen durchgeführt wird, also ein Vektor $\vec{\mathbf{x}}$ wie folgt gedreht wird: $\mathcal{R}_z(\Phi) \cdot \mathcal{R}_x(\Theta) \cdot \mathcal{R}_y(\Psi) \cdot \vec{\mathbf{x}}$.

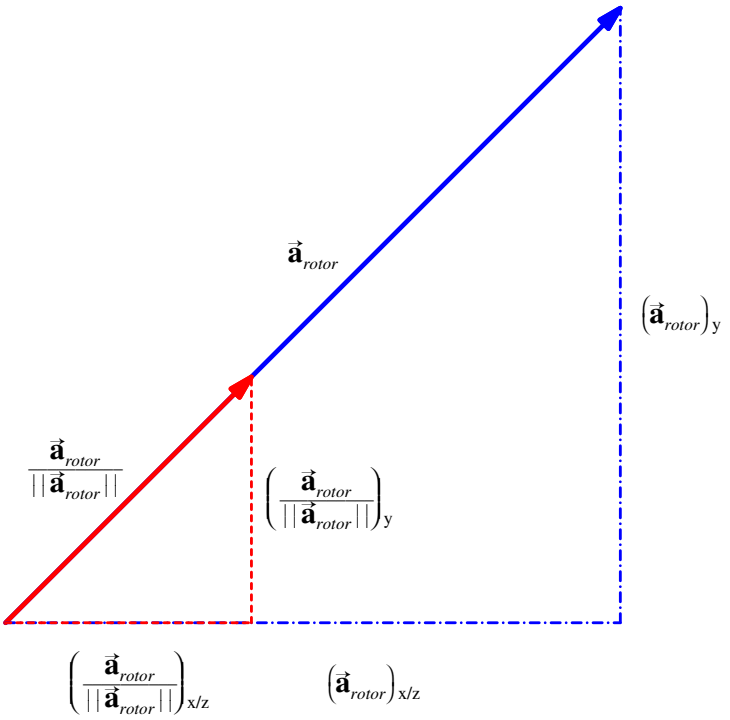

Abbildung 6.4: Die x- und z-Komponente der Rotorbeschleunigung $\vec{\mathbf{a}}_{rotor}$ läßt sich bei Kenntnis der y-Komponente und des zugehörigen Einheitsvektors $\frac{\vec{\mathbf{a}}_{rotor}}{\|\vec{\mathbf{a}}_{rotor}\|}$ mit Hilfe des Strahlensatzes direkt bestimmen. Denn es gilt: $\frac{(\vec{\mathbf{a}}_{rotor})_{x/z}}{\left(\frac{\vec{\mathbf{a}}_{rotor}}{\|\vec{\mathbf{a}}_{rotor}\|}\right)_{x/z}} = \frac{(\vec{\mathbf{a}}_{rotor})_{y}}{\left(\frac{\vec{\mathbf{a}}_{rotor}}{\|\vec{\mathbf{a}}_{rotor}\|}\right)_{y}}$.

Aus der Rotorbeschleunigung $\vec{\mathbf{a}}_{rotor}$ läßt sich nun aber direkt die gesuchte Vortriebsbeschleunigung $\vec{\mathbf{a}}_{thrust}$ ablesen:

$$\vec{\mathbf{a}}_{thrust} = \begin{pmatrix} (\vec{\mathbf{a}}_{rotor})_x \\ 0 \\ (\vec{\mathbf{a}}_{rotor})_z \end{pmatrix} = \mathrm{g} \cdot \begin{pmatrix} \frac{\tan(\Phi)}{\cos(\Theta)} \\ 0 \\ -\tan(\Theta) \end{pmatrix} \quad . \tag{6.21}$$

Es sollte an dieser Stelle erwähnt werden, daß Gl. 6.21 natürlich nur für Neigungen und damit zugehörige Neigungswinkel Φ und Θ definiert ist, welche der Flugroboter auch tatsächlich in der Lage ist, einzunehmen. Denn rein theoretisch müßte nach dieser Gleichung der Roboter für z. B. $\Theta = 90\,^\circ$ eine nahezu unendliche Vortriebsgeschwindigkeit erhalten, was sich natürlich nicht mit dem realen Fall deckt. In der Realität würde der Roboter stattdessen abstürzen, da ihm der nötige Auftrieb fehlte.

Mit den drei Geschwindigkeitsanteilen aus Gl. 6.8, 6.12 und 6.21 ergibt sich für Gl. 6.5 nunmehr die neue Geschwindigkeit zum Zeitschritt k zu:

$$\vec{\mathbf{v}}_k = \vec{\mathbf{v}}_{k-1\,,k} + \vec{\mathbf{v}}_{drag\,,k} + \vec{\mathbf{v}}_{thrust\,,k} \tag{6.22}$$

$$= \begin{pmatrix} (\mathrm{v}_x)_{k-1}\,\cos(\Psi) - (\mathrm{v}_z)_{k-1}\,\sin(\Psi) \\ (\mathrm{v}_y)_{k-1} \\ (\mathrm{v}_x)_{k-1}\,\sin(\Psi) + (\mathrm{v}_z)_{k-1}\,\cos(\Psi) \end{pmatrix} \cdot (1 - \xi\,\Delta t \cdot \|\vec{\mathbf{v}}_{k-1}\|)$$

$$+ \begin{pmatrix} \mathrm{g} \cdot \frac{\tan(\Phi)}{\cos(\Theta)} \\ 0 \\ -\mathrm{g} \cdot \tan(\Theta) \end{pmatrix} \cdot \Delta t \quad . \tag{6.23}$$

Als Hinweis sei an dieser Stelle noch erwähnt, daß $(\mathrm{v}_y)_{k-1}$ in diesem Modell durchgängig als Null angenommen wird.

6.1.2 Zustandsgleichungen des EKF

Die allgemeine Zustandsgleichung des Erweiterten Kalman-Filters lautet:

$$\vec{\mathbf{x}}_k = f\,(\vec{\mathbf{x}}_{k-1}, \vec{\mathbf{u}}_{k-1}, \vec{w}_{k-1}) \quad . \tag{6.24}$$

Der zu schätzende Zustand $\vec{\mathbf{x}}_k \in \mathbb{R}^3$ entspricht dem Geschwindigkeitsvektor $\vec{\mathbf{v}}_k \in \mathbb{R}^3$ aus dem vorherigen Abschnitt,

$$\vec{\mathbf{x}}_k = \vec{\mathbf{v}}_k = \begin{pmatrix} \mathrm{v}_x \\ \mathrm{v}_y \\ \mathrm{v}_z \end{pmatrix} \quad , \tag{6.25}$$

der Kommandovektor $\vec{\mathbf{u}} \in \mathbb{R}^3$ beinhaltet die drei Steuerwinkel,[8]

$$\vec{\mathbf{u}}_{k-1} = \begin{pmatrix} \Phi_{k-1} \\ \Theta_{k-1} \\ \Psi_{k-1} \end{pmatrix} \quad , \tag{6.26}$$

[8]Korrekterweise enthält das Flugkommando anstelle des Gier-Winkels Ψ dessen Winkelgeschwindigkeit $\dot{\Psi}$. Jedoch läßt sich aus dieser mit dem betrachteten und bekannten Zeitintervall Δt der Winkel Ψ herleiten. In den weiteren Abschnitten wird deshalb immer direkt mit Ψ anstelle von $\dot{\Psi}$ gerechnet.

und die zugehörige nicht-lineare Funktion $f\left(\vec{\mathbf{x}}_{k-1}, \vec{\mathbf{u}}_{k-1}, \vec{w}_{k-1}\right)$ entspricht wiederum dem Zustandsmodell aus Gl. 6.23, wobei jedoch noch der Fehlervektor $\vec{w} \in \mathbb{R}^3$ additiv hinzugerechnet wird:

$$\vec{\mathbf{v}}_k = f\left(\vec{\mathbf{v}}_{k-1}, \vec{\mathbf{u}}_{k-1}, \vec{w}_{k-1}\right) \tag{6.27}$$

$$= \begin{pmatrix} (\mathrm{v}_x)_{k-1} \cos(\Psi_{k-1}) - (\mathrm{v}_z)_{k-1} \sin(\Psi_{k-1}) \\ (\mathrm{v}_y)_{k-1} \\ (\mathrm{v}_x)_{k-1} \sin(\Psi_{k-1}) + (\mathrm{v}_z)_{k-1} \cos(\Psi_{k-1}) \end{pmatrix} \cdot (1 - \xi\, \Delta t \cdot \|\vec{\mathbf{v}}_{k-1}\|)$$

$$+ \begin{pmatrix} \mathrm{g} \cdot \frac{\tan(\Phi_{k-1})}{\cos(\Theta_{k-1})} \\ 0 \\ -\mathrm{g} \cdot \tan(\Theta_{k-1}) \end{pmatrix} \cdot \Delta t + \begin{pmatrix} (w_x)_{k-1} \\ (w_y)_{k-1} \\ (w_z)_{k-1} \end{pmatrix} \quad . \tag{6.28}$$

Für den Korrekturschritt des Kalman-Filters wird die Translationsschätzung des *Kanatani*-Algorithmus als Messung $\vec{\mathbf{z}}_k$ herangezogen. Sie gibt jedoch nur die Richtung der Translation an, nicht jedoch ihren Betrag. Somit ist es nicht möglich, eine Korrektur der betraglichen Geschwindigkeit der Zustandsschätzung vorzunehmen.[9] Die Richtung der Zustandsschätzung läßt sich hingegen damit korrigieren. Die Translationsrichtung des *Kanatani*-Algorithmus ist in Kamera-Koordinaten angegeben, während die Schätzung $\vec{\mathbf{v}}_k$ des Translationsmodells in Koordinaten des *RR-KS* angegeben ist. Für den Korrekturschritt des EKF muß diese (bzw. ihre Richtung $\frac{\vec{\mathbf{v}}_k}{\mathrm{v}_k}$) nun in Kamera-Koordinaten überführt werden.
Dafür transformiert man die Geschwindigkeit $\vec{\mathbf{v}}$ zuerst in Roboter-Koordinaten, indem sie der Rotation durch Nicken und Rollen unterzogen wird, und im Anschluß daran weiter in Kamera-Koordinaten, indem man sie einer weiteren Drehung um die x-Achse des Roboters mit einem Winkel α unterzieht. Damit wird der Verdrehung der Kamerablickrichtung zur Geradeausrichtung des Roboters um den Winkel α Rechnung getragen.[10] Je nach Kamera-Stellung muß α automatisch angepaßt werden, wird im folgenden jedoch als konstant angenommen. Man darf nicht vergessen, das Ergebnis noch durch den Betrag v_k der Geschwindigkeit zu dividieren, um einen Einheitsvektor, also eine reine Translationsrichtung, zu erhalten.

[9] Um eine Korrektur der betraglichen Geschwindigkeit vornehmen zu können, müßten Messungen dieser Geschwindigkeit vorliegen, z. B. durch einen Fahrtmesser oder Windsensor. Der verwendete Flugroboter besitzt jedoch keinerlei solcher Sensoren, weshalb eine Korrektur der betraglichen Geschwindigkeit nicht durchführbar ist.

[10] Die Drehachse der Kamera liegt in Wahrheit nicht in der Rotorebene, sondern darunter, so daß für die Transformation von $\vec{\mathbf{v}}_k$ in Kamera-Koordinaten ein zusätzlicher (translatorischer) Versatz hinzukommen müßte. Dieser wird hier jedoch weggelassen, da er vernachlässigbar klein ist. Zudem ist nicht sicher, ob aufgrund des Roboter-Gewichtes die Drehachsen des Roboters nicht vielleicht unterhalb der Rotorebene liegen und ihr Ursprung damit annähernd mit dem der Kamera-Drehachse zusammenfällt.

Für die Funktion $h\left(\vec{\mathbf{x}}_k, \vec{v}\right)$, welche die Geschwindigkeitsschätzung $\vec{\mathbf{x}}_k (= \vec{\mathbf{v}}_k)$ zusammen mit einem Fehlerterm $\vec{v}$ zur Messung $\vec{\mathbf{z}}_k$, also der Bewegungsschätzung des *Kanatani*-Algorithmus, in Beziehung setzt, ergibt sich dann:

$$\vec{\mathbf{z}}_k = h\left(\vec{\mathbf{v}}_k, \vec{v}_k\right) \tag{6.29}$$

$$= \mathcal{R}_x\left(\alpha\right) \cdot \mathcal{R}_z\left(\Phi_{k-1}\right) \cdot \mathcal{R}_x\left(\Theta_{k-1}\right) \cdot \frac{\vec{\mathbf{v}}_k + \vec{v}_k}{\|\vec{\mathbf{v}}_k + \vec{v}_k\|} \tag{6.30}$$

$$= \frac{\begin{pmatrix} 1 & 0 & 0 \\ 0 & \mathrm{c}_\alpha & \mathrm{s}_\alpha \\ 0 & -\mathrm{s}_\alpha & \mathrm{c}_\alpha \end{pmatrix} \cdot \begin{pmatrix} \mathrm{c}_\Phi & \mathrm{s}_\Phi & 0 \\ -\mathrm{s}_\Phi & \mathrm{c}_\Phi & 0 \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 & 0 \\ 0 & \mathrm{c}_\Theta & \mathrm{s}_\Theta \\ 0 & -\mathrm{s}_\Theta & \mathrm{s}_\Theta \end{pmatrix} \cdot \begin{pmatrix} \left(\mathrm{v}_x\right)_k + \left(v_x\right)_k \\ \left(\mathrm{v}_y\right)_k + \left(v_y\right)_k \\ \left(\mathrm{v}_z\right)_k + \left(v_z\right)_k \end{pmatrix}}{\sqrt{\left(\left(\mathrm{v}_x\right)_k + \left(v_x\right)_k\right)^2 + \left(\left(\mathrm{v}_y\right)_k + \left(v_y\right)_k\right)^2 + \left(\left(\mathrm{v}_z\right)_k + \left(v_z\right)_k\right)^2}}$$

$$= \frac{\begin{pmatrix} \mathrm{c}_\Phi & \mathrm{s}_\Phi\mathrm{c}_\Theta & \mathrm{s}_\Phi\mathrm{s}_\Theta \\ -\mathrm{c}_\alpha\mathrm{s}_\Phi & \mathrm{c}_\alpha\mathrm{c}_\Phi\mathrm{c}_\Theta - \mathrm{s}_\alpha\mathrm{s}_\Theta & \mathrm{c}_\alpha\mathrm{c}_\Phi\mathrm{s}_\Theta + \mathrm{s}_\alpha\mathrm{c}_\Theta \\ \mathrm{s}_\alpha\mathrm{s}_\Phi & -\mathrm{s}_\alpha\mathrm{c}_\Phi\mathrm{c}_\Theta - \mathrm{c}_\alpha\mathrm{s}_\Theta & -\mathrm{s}_\alpha\mathrm{c}_\Phi\mathrm{s}_\Theta + \mathrm{c}_\alpha\mathrm{c}_\Theta \end{pmatrix} \cdot \begin{pmatrix} \left(\mathrm{v}_x\right)_k + \left(v_x\right)_k \\ \left(\mathrm{v}_y\right)_k + \left(v_y\right)_k \\ \left(\mathrm{v}_z\right)_k + \left(v_z\right)_k \end{pmatrix}}{\sqrt{\left(\left(\mathrm{v}_x\right)_k + \left(v_x\right)_k\right)^2 + \left(\left(\mathrm{v}_y\right)_k + \left(v_y\right)_k\right)^2 + \left(\left(\mathrm{v}_z\right)_k + \left(v_z\right)_k\right)^2}} \quad . \tag{6.31}$$

Hinweis: Zur besseren Darstellung wurde $\cos(\gamma)$ durch c_γ und $\sin(\gamma)$ durch s_γ ersetzt, sowie der Index k–1 für Φ und Θ fallengelassen.

6.1.3 Jacobi-Matrizen des EKF

Die einzelnen, im Erweiterten Kalman-Filter verwendeten Jacobi-Matrizen sind im folgenden angegeben. Sie lassen sich durch Berechnung der partiellen Ableitungen erstellen.

Für die Jacobi-Matrix $\mathbf{A}_k$, welche die partiellen Ableitungen der Funktion f nach der Geschwindigkeit $\vec{\mathbf{v}}_{k-1}$ enthält, ergibt sich eine nicht ganz einfache Matrix, weshalb hier jede partielle Ableitung einzeln aufgeführt ist. Zur besseren Übersicht wurde der Index k weggelassen und wie bereits bei Gl. 6.31 cos und sin verkürzt dargestellt.

$$A_{1,1} = \mathrm{c}_\Psi - \xi\,\Delta t \cdot \left(\frac{\mathrm{v}_x \cdot \left(\mathrm{v}_x\,\mathrm{c}_\Psi - \mathrm{v}_z\,\mathrm{s}_\Psi\right)}{\sqrt{\mathrm{v}_x^2 + \mathrm{v}_y^2 + \mathrm{v}_z^2}} + \mathrm{c}_\Psi \sqrt{\mathrm{v}_x^2 + \mathrm{v}_y^2 + \mathrm{v}_z^2} \right) \quad , \tag{6.32}$$

$$A_{1,2} = -\xi\,\Delta t \cdot \frac{\mathrm{v}_y \cdot \left(\mathrm{v}_x\,\mathrm{c}_\Psi - \mathrm{v}_z\,\mathrm{s}_\Psi\right)}{\sqrt{\mathrm{v}_x^2 + \mathrm{v}_y^2 + \mathrm{v}_z^2}} \quad , \tag{6.33}$$

$$A_{1,3} = -\mathrm{s}_\Psi - \xi\,\Delta t \cdot \left(\frac{\mathrm{v}_z \cdot \left(\mathrm{v}_x\,\mathrm{c}_\Psi - \mathrm{v}_z\,\mathrm{s}_\Psi\right)}{\sqrt{\mathrm{v}_x^2 + \mathrm{v}_y^2 + \mathrm{v}_z^2}} - \mathrm{s}_\Psi \sqrt{\mathrm{v}_x^2 + \mathrm{v}_y^2 + \mathrm{v}_z^2} \right) \quad , \tag{6.34}$$

$$A_{2,1} = -\xi\,\Delta t \cdot \frac{\mathrm{v}_x \cdot \mathrm{v}_y}{\sqrt{\mathrm{v}_x^2 + \mathrm{v}_y^2 + \mathrm{v}_z^2}} \quad , \tag{6.35}$$

$$A_{2,2} = 1 \; -\xi\,\Delta t \cdot \left(\frac{\mathrm{v}_y \cdot \mathrm{v}_y}{\sqrt{\mathrm{v}_x^2 + \mathrm{v}_y^2 + \mathrm{v}_z^2}} + \sqrt{\mathrm{v}_x^2 + \mathrm{v}_y^2 + \mathrm{v}_z^2} \right) \quad , \tag{6.36}$$

$$A_{2,3} = -\xi\,\Delta t \cdot \frac{\mathrm{v}_z \cdot \mathrm{v}_y}{\sqrt{\mathrm{v}_x^2 + \mathrm{v}_y^2 + \mathrm{v}_z^2}} \quad , \tag{6.37}$$

$$A_{3,1} = \mathrm{s}_\Psi \; -\xi\,\Delta t \cdot \left(\frac{\mathrm{v}_x \cdot \left(\mathrm{v}_x\,\mathrm{c}_\Psi + \mathrm{v}_z\,\mathrm{s}_\Psi\right)}{\sqrt{\mathrm{v}_x^2 + \mathrm{v}_y^2 + \mathrm{v}_z^2}} + \mathrm{s}_\Psi \sqrt{\mathrm{v}_x^2 + \mathrm{v}_y^2 + \mathrm{v}_z^2} \right) \quad , \tag{6.38}$$

$$A_{3,2} = -\xi\,\Delta t \cdot \frac{\mathrm{v}_y \cdot \left(\mathrm{v}_x\,\mathrm{c}_\Psi + \mathrm{v}_z\,\mathrm{s}_\Psi\right)}{\sqrt{\mathrm{v}_x^2 + \mathrm{v}_y^2 + \mathrm{v}_z^2}} \quad , \tag{6.39}$$

$$A_{3,3} = \mathrm{c}_\Psi \; -\xi\,\Delta t \cdot \left(\frac{\mathrm{v}_z \cdot \left(\mathrm{v}_x\,\mathrm{c}_\Psi + \mathrm{v}_z\,\mathrm{s}_\Psi\right)}{\sqrt{\mathrm{v}_x^2 + \mathrm{v}_y^2 + \mathrm{v}_z^2}} + \mathrm{c}_\Psi \sqrt{\mathrm{v}_x^2 + \mathrm{v}_y^2 + \mathrm{v}_z^2} \right) \quad . \tag{6.40}$$

Für den speziellen Fall, daß der Betrag der Geschwindigkeit aus dem vorherigen Zeitschritt k–1 Null beträgt, vereinfacht sich $\mathbf{A}_k$ zur Rotationsmatrix $\mathcal{R}_y(\Psi_{k-1})$:

$$\lim_{\|\vec{\mathbf{v}}_{k-1}\| \to 0} \mathbf{A}_k = \begin{pmatrix} \cos(\Psi_{k-1}) & 0 & -\sin(\Psi_{k-1}) \\ 0 & 1 & 0 \\ \sin(\Psi_{k-1}) & 0 & \cos(\Psi_{k-1}) \end{pmatrix} \quad . \tag{6.41}$$

Die Matrix $\mathbf{W}_k$, mit den partiellen Ableitungen der Funktion f nach dem Fehler $\vec{w}_{k-1}$, entspricht der Identität:

$$\mathbf{W}_k = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \quad . \tag{6.42}$$

Die Jacobi-Matrix $\mathbf{H}_k$ der partiellen Ableitungen von Funktion h nach der (*a priori*) Geschwindigkeitsschätzung $\vec{\mathbf{v}}_k$ sieht ebenfalls um einiges komplizierter aus, weshalb jede partielle Ableitung wie schon für $\mathbf{A}_k$ einzeln und verkürzt dargestellt ist. Zusätz-

lich ist mit $[\,\mathcal{R}_{\alpha,\Phi,\Theta}\cdot\vec{\mathbf{v}}\,]_n$ die n-te Zeile der Rotationsmatrix aus Gl. 6.31 multipliziert mit der (*a priori*) Geschwindigkeitsschätzung $\vec{\mathbf{v}}_k$ gemeint.

$$H_{1,1} = \frac{\mathrm{c}_\Phi}{\sqrt{\mathrm{v}_x^2+\mathrm{v}_y^2+\mathrm{v}_z^2}} - \mathrm{v}_x \frac{[\,\mathcal{R}_{\alpha,\Phi,\Theta}\cdot\vec{\mathbf{v}}\,]_1}{\left(\sqrt{\mathrm{v}_x^2+\mathrm{v}_y^2+\mathrm{v}_z^2}\right)^3} \quad , \tag{6.43}$$

$$H_{1,2} = \frac{\mathrm{s}_\Phi \mathrm{c}_\Theta}{\sqrt{\mathrm{v}_x^2+\mathrm{v}_y^2+\mathrm{v}_z^2}} - \mathrm{v}_y \frac{[\,\mathcal{R}_{\alpha,\Phi,\Theta}\cdot\vec{\mathbf{v}}\,]_1}{\left(\sqrt{\mathrm{v}_x^2+\mathrm{v}_y^2+\mathrm{v}_z^2}\right)^3} \quad , \tag{6.44}$$

$$H_{1,3} = \frac{\mathrm{s}_\Phi \mathrm{s}_\Theta}{\sqrt{\mathrm{v}_x^2+\mathrm{v}_y^2+\mathrm{v}_z^2}} - \mathrm{v}_z \frac{[\,\mathcal{R}_{\alpha,\Phi,\Theta}\cdot\vec{\mathbf{v}}\,]_1}{\left(\sqrt{\mathrm{v}_x^2+\mathrm{v}_y^2+\mathrm{v}_z^2}\right)^3} \quad , \tag{6.45}$$

$$H_{2,1} = \frac{-\mathrm{c}_\alpha \mathrm{s}_\Phi}{\sqrt{\mathrm{v}_x^2+\mathrm{v}_y^2+\mathrm{v}_z^2}} - \mathrm{v}_x \frac{[\,\mathcal{R}_{\alpha,\Phi,\Theta}\cdot\vec{\mathbf{v}}\,]_2}{\left(\sqrt{\mathrm{v}_x^2+\mathrm{v}_y^2+\mathrm{v}_z^2}\right)^3} \quad , \tag{6.46}$$

$$H_{2,2} = \frac{\mathrm{c}_\alpha \mathrm{c}_\Phi \mathrm{c}_\Theta - \mathrm{s}_\alpha \mathrm{s}_\Theta}{\sqrt{\mathrm{v}_x^2+\mathrm{v}_y^2+\mathrm{v}_z^2}} - \mathrm{v}_y \frac{[\,\mathcal{R}_{\alpha,\Phi,\Theta}\cdot\vec{\mathbf{v}}\,]_2}{\left(\sqrt{\mathrm{v}_x^2+\mathrm{v}_y^2+\mathrm{v}_z^2}\right)^3} \quad , \tag{6.47}$$

$$H_{2,3} = \frac{\mathrm{c}_\alpha \mathrm{c}_\Phi \mathrm{s}_\Theta + \mathrm{s}_\alpha \mathrm{c}_\Theta}{\sqrt{\mathrm{v}_x^2+\mathrm{v}_y^2+\mathrm{v}_z^2}} - \mathrm{v}_z \frac{[\,\mathcal{R}_{\alpha,\Phi,\Theta}\cdot\vec{\mathbf{v}}\,]_2}{\left(\sqrt{\mathrm{v}_x^2+\mathrm{v}_y^2+\mathrm{v}_z^2}\right)^3} \quad , \tag{6.48}$$

$$H_{3,1} = \frac{\mathrm{s}_\alpha \mathrm{s}_\Phi}{\sqrt{\mathrm{v}_x^2+\mathrm{v}_y^2+\mathrm{v}_z^2}} - \mathrm{v}_x \frac{[\,\mathcal{R}_{\alpha,\Phi,\Theta}\cdot\vec{\mathbf{v}}\,]_3}{\left(\sqrt{\mathrm{v}_x^2+\mathrm{v}_y^2+\mathrm{v}_z^2}\right)^3} \quad , \tag{6.49}$$

$$H_{3,2} = \frac{-\mathrm{s}_\alpha \mathrm{c}_\Phi \mathrm{c}_\Theta - \mathrm{c}_\alpha \mathrm{s}_\Theta}{\sqrt{\mathrm{v}_x^2+\mathrm{v}_y^2+\mathrm{v}_z^2}} - \mathrm{v}_y \frac{[\,\mathcal{R}_{\alpha,\Phi,\Theta}\cdot\vec{\mathbf{v}}\,]_3}{\left(\sqrt{\mathrm{v}_x^2+\mathrm{v}_y^2+\mathrm{v}_z^2}\right)^3} \quad , \tag{6.50}$$

$$H_{3,2} = \frac{-\mathrm{s}_\alpha \mathrm{c}_\Phi \mathrm{s}_\Theta + \mathrm{c}_\alpha \mathrm{c}_\Theta}{\sqrt{\mathrm{v}_x^2+\mathrm{v}_y^2+\mathrm{v}_z^2}} - \mathrm{v}_z \frac{[\,\mathcal{R}_{\alpha,\Phi,\Theta}\cdot\vec{\mathbf{v}}\,]_3}{\left(\sqrt{\mathrm{v}_x^2+\mathrm{v}_y^2+\mathrm{v}_z^2}\right)^3} \quad . \tag{6.51}$$

Für den speziellen Fall, daß der Betrag der (*a priori*) Geschwindigkeit v_k sich zu Null ergibt, wird $\mathbf{H}_k$ zur Nullmatrix:

$$\lim_{\|\vec{\mathbf{v}}_{k-1}\|\to 0} \mathbf{H}_k = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} \quad . \tag{6.52}$$

Für die Matrix $\mathbf{V}_k$, mit den partiellen Ableitungen der Funktion h nach dem Fehler $\vec{v}_{k-1}$ mit anschließender Gleichsetzung des Fehlers zu Null, ergibt sich das gleiche Ergebnis wie für Matrix $\mathbf{H}_k$:

$$\mathbf{V}_k = \mathbf{H}_k \quad . \tag{6.53}$$

Die zusätzliche Grenzwertbetrachtung gilt gleichermaßen.

6.1.4 Fehler-Kovarianzmatrizen des EKF

Für die Bildung der Kovarianzmatrix $\mathbf{R}_k$ des vom *Kanatani*-Algorithmus eingebrachten Meßfehlers v_k wurde eine künstliche Bildfolge in einem Simulator erzeugt. Dadurch lagen exakte Positions- und damit auch Bewegungsdaten des simulierten Fluges vor. Die erhaltene Bildersequenz wurde zur Berechnung des optischen Flusses verwendet, welcher wiederum dem *Kanatani*-Algorithmus zur Berechnung der simulierten Kamerabewegung vorgelegt wurde. Aus der Abweichung zwischen der exakten Bewegungsrichtung und der vom *Kanatani*-Algorithmus berechneten Bewegungrichtung für jede der drei Komponenten des Richtungsvektors läßt sich dann eine konstante Kovarianzmatrix $\mathbf{R}'$ wie folgt bilden:

$$\mathbf{R}' = \begin{pmatrix} \mathrm{Var}\,(v_x) & \mathrm{Cov}\,(v_x, v_y) & \mathrm{Cov}\,(v_x, v_z) \\ \mathrm{Cov}\,(v_y, v_x) & \mathrm{Var}\,(v_y) & \mathrm{Cov}\,(v_y, v_z) \\ \mathrm{Cov}\,(v_z, v_y) & \mathrm{Cov}\,(v_z, v_y) & \mathrm{Var}\,(v_z) \end{pmatrix} \quad . \tag{6.54}$$

Dabei bezeichnen

$$\mathrm{Cov}\,(a, b) = \frac{1}{n-1} \cdot \sum_{i=1}^{n} (a_i - \bar{a}_i) \cdot \left(b_i - \bar{b}_i\right) \quad , \tag{6.55}$$

$$\mathrm{Var}\,(a) = \mathrm{Cov}\,(a, a) = \frac{1}{n-1} \cdot \sum_{i=1}^{n} (a_i - \bar{a}_i)^2 \tag{6.56}$$

die Stichprobenkovarianz bzw. -varianz für die Komponenten a und b des Bewegungsvektors über alle n Bewegungsschätzungen der Bildfolge, wobei mit $\bar{a}_i$ und $\bar{b}_i$ die exakte Bewegungskomponente der jeweiligen Bildverschiebung i gemeint ist.

[11] Da die simulierte Bildfolge jedoch nur einen Flug in Kamerablickrichtung darstellt, kann auch nur für einen solchen Fall die Matrix $\mathbf{R}'$ direkt als Fehler-Kovarianzmatrix $\mathbf{R}$ verwendet werden. Andernfalls käme die gemessene Fehlerellipse falsch zum liegen. Für den allgemeinen Fall muß die Matrix $\mathbf{R}'$ also noch um den jeweiligen Winkel γ gedreht werden, der die horizontale Verdrehung zwischen dem im vorherigen Zeitschritt k–1 berechneten Bewegungsvektor $\vec{\mathbf{v}}_{k-1,k}$ und dem *a priori* Bewegungsvektor $\vec{\mathbf{v}}_k$ beschreibt. Damit ergibt sich für $\mathbf{R}_k$:

$$\mathbf{R}_k = \mathcal{R}_y(\gamma_k) \cdot \mathbf{R}' = \begin{pmatrix} \cos(\gamma_k) & 0 & -\sin(\gamma_k) \\ 0 & 1 & 0 \\ \sin(\gamma_k) & 0 & \cos(\gamma_k) \end{pmatrix} \cdot \mathbf{R}' \quad . \tag{6.57}$$

γ_k wiederum läßt sich unter Verwendung der Definition des Skalarprodukts leicht bestimmen:

$$\gamma_k = \arccos\left(\frac{\vec{\mathbf{v}}_k \quad \cdot \quad \vec{\mathbf{v}}_{k-1,k}}{\|\vec{\mathbf{v}}_k\| \cdot \|\vec{\mathbf{v}}_{k-1,k}\|} \right) \quad . \tag{6.58}$$

[11] Der folgende Abschnitt wurde in der Errata korrigiert, die sich am Ende der Arbeit befindet.

Die Kovarianzmatrix $\mathbf{Q}_k$ des Prozeßfehlers w_k aufzustellen, ist hingegen nicht so einfach möglich. Da das Modell bisher davon ausgeht, daß die per Flugkommando gesetzten Neigungen vom Roboter instantan eingenommen werden, was natürlich nicht der Wirklichkeit entsprechen kann, wird der Fehler bei Änderung des Flugkommandos höchstwahrscheinlich größer ausfallen. Somit müßte die Zeit, die der Roboter braucht, um die gewünschten Neigung aus der vorherigen einzunehmen, vorab gemessen und eine zugehörige Funktionsgleichung aufgestellt werden. Für den Fall einer erneuten Änderung des Flugkommandos innerhalb dieses Zeitraums verkompliziert sich diese Berechnung zusätzlich. Da dies jedoch den Rahmen der vorliegenden Arbeit sprengen würde, wurde $\mathbf{Q}_k$ deshalb für alle Zeitschritte k in erster Näherung als konstant angenommen. Als zu verwendender Fehler w wurde dafür der auf Erfahrungswerten beruhende maximale horizontale Drift des Roboters in Ruhelage von ungefähr 10 cm pro Sekunde gewählt.

$$\mathbf{Q}_k = \mathbf{Q} = \begin{pmatrix} \mathrm{Var}\,(w_x) & 0 & 0 \\ 0 & \mathrm{Var}\,(w_y) & 0 \\ 0 & 0 & \mathrm{Var}\,(w_z) \end{pmatrix} \quad . \tag{6.59}$$

7. Flugexperimente und Ergebnisse

7.1 Bestimmung der Funktionskonstante ξ

Zur Bestimmung der Funktionskonstante ξ, die in den Luftwiderstandsterm des Geschwindigkeitsmodells aus Gl. 6.5 einfließt, wurden Messungen am realen Flugroboter durchgeführt.
Die grundsätzliche Idee besteht hierbei darin, daß der Roboter eine gerade Flugstrecke zurücklegt, währenddessen seine Position zusammen mit der vergangenen Zeit aufgezeichnet wird. Die erhaltenen Daten müssen dann mit einer geeigneten Funktion „gefittet“ werden, welche die gesuchte Konstante ξ bzw. die in ihr enthaltenen Variablen c_{drag}, A und ρ als Fit-Parameter enthält.

7.1.1 Anordnung zum Versuchsaufbau

Um äußere Einflüsse wie Wind auszuschließen, durfte die Testrecke nicht im Freien liegen, weshalb die Länge der schließlich gewählten Teststrecke auf ca. 18 m begrenzt war,[1] von denen jedoch aus Sicherheitsgründen[2] für die eigentliche Messung nur 10 m zu Verfügung standen. In Ermangelung einer geeigneten (transportablen) Positionstracking-Hardware[3] wurde auf folgendes System zur Berechnung der Position zurückgegriffen.

[1] Die Teststrecke befand sich im Foyer der Mensa Morgenstelle Tübingen des Studentenwerks Tübingen-Hohenheim A.d.ö.R.

[2] Abbremsweg und Sicherheitsabstand von Wänden

[3] GPS-Daten sind zum einen nur im Freien verfügbar und zum anderen für den gewünschten Zweck zu ungenau.

Der Roboter flog oberhalb einer genau definierten Linie, während von der Seite eine Kamera die Flugbewegung aufzeichnete und zu jedem Videobild die vergangene Zeit speicherte. Kleine Positionsmarken in Abständen von einem Meter, die an der Linie angebracht waren, ermöglichten die anschließende Auswertung dieser Videobilder. Um zusätzlich den Start und das Ende einer Messung im Bild anzuzeigen, befand sich im Sichtbereich der Kamera eine Leuchtdiode, die automatisch von der Bodenstation angeschaltet wurde, sobald das erste für die Messung relevante Flugkommando zum Roboter geschickt wurde. Analog dazu wurde sie ausgeschaltet, wenn das nächste davon abweichende Flugkommando zum Roboter gesendet wurde. Bei der Auswertung wurden vorab einmalig die Koordinaten im Bild per Mausklick bestimmt, an denen die verschiedenen Positionsmarken sichtbar waren, und daraus der horizontale Streckenverlauf rekonstruiert. Mit einem eigens zu diesem Zweck entwickelten Auswertungsprogramm ließ sich die Videosequenz nun Bild für Bild durchgehen und, durch Klicken mit der Maus auf das Zentrum des Roboters im Bild, dessen horizontale Position bestimmen. Diese wurde automatisch[4] von Bildkoordinaten in reale Distanzen umgerechnet und zusammen mit der aus dem Bild ausgelesenen Zeitinformation in einer Datei gespeichert. Abb. 7.1 zeigt exemplarisch den Meßaufbau aus Sicht der verwendeten Kamera.

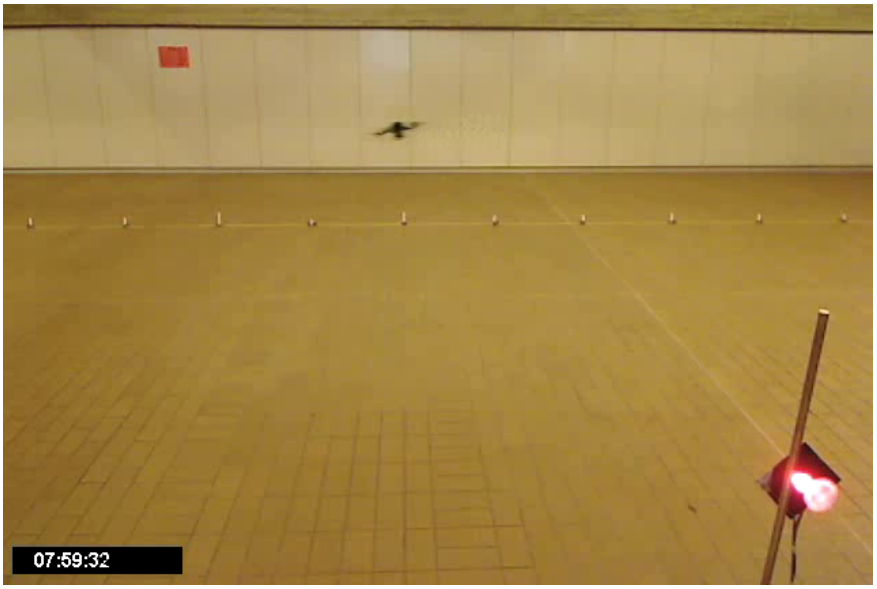

Abbildung 7.1: Meßaufbau aus Sicht der verwendeten Kamera. Der Roboter fliegt oberhalb der weißen Meßlinie, auf der im Abstand von einem Meter Positionsmarken angebracht sind, von rechts nach links. Rechts im Vordergrund befindet sich die Leuchtdiode, die signalisiert, daß eine auszuwertende Messung gerade aufgezeichnet wird. Links unten im Bild ist der zugehörige Zeitstempel eingeblendet.

[4] aufgrund der vorhergegangenen Streckenverlaufsberechnung

Aufgrund der fehlenden Tiefeninformationen war es zwingend erforderlich, daß der Roboter exakt oberhalb der definierten Meßlinie flog. Denn flöge er schräg davor, also näher an der Kamera, so legte er im Kamerabild die gleiche Strecke schneller zurück und die Auswertung ginge fälschlicherweise davon aus, daß sich der Roboter, anders als in Wirklichkeit, ebenfalls mit höherer Geschwindigkeit bewegte. Analog verhielte es sich mit dem umgekehrten Fall, wobei die Auswertung dann eine zu geringe Geschwindigkeit ergäbe.
Dieser Fehler ließ sich anhand des Größenunterschieds im Bild nur schlecht feststellen, weshalb der Roboter zur nachträglichen Kontrolle der Flugbahn den Flug mit seiner eigenen Kamera zusätzlich aufzeichnete. Dabei war die Kamera senkrecht nach unten gerichtet, so daß man erkennen konnte, ob der Roboter oberhalb der Meßlinie flog oder daneben. Je nachdem wurde eine Messung zur weiteren Auswertung verwendet oder verworfen.

7.1.2 Beschleunigungsversuche

Um ξ für unterschiedliche Neigungswinkel des Roboters zu bestimmen, wurden in den durchgeführten Versuchen die Beschleunigung des Roboters aus der Ruhelage untersucht. Dafür wurde der bereits im Schwebeflug befindliche Roboter von Hand oberhalb des Startpunktes ausgerichtet und dann durch ein über die gesamte Messung konstant gehaltenes Flugkommando in Bewegung gesetzt. Als Flugkommando wurde bei diesen Versuchen ausschließlich der Nick-Winkel Θ gesetzt und zwischen den einzelnen Meßreihen variiert, während Roll- und Gier-Winkel sowie der zusätzliche Schub konstant bei Null belassen wurden.

7.1.2.1 Aufstellung der Fit-Gleichung

Zum Fitten der erhaltenen Daten wurde eine Funktion $s(t)$ verwendet, die die Entwicklung des Ortes (bzw. der zurückgelegten Distanz) über die Zeit beschreibt. Die Herleitung ist im folgenden beschrieben:

Ausgehend von der Kraftgleichung aus Gl. 6.1, wobei jedoch die Luftwiderstandskraft gleich mit negativen Vorzeichen versehen wird, und der zugehörigen Formel für den Luftwiderstand aus Gl. 6.9 erhält man:

$$F_{net} = F_{thrust} - F_{drag} \tag{7.1}$$

$$\Leftrightarrow \quad m \cdot \dot{\mathrm{v}}_{net} = m \cdot \dot{\mathrm{v}}_{thrust} - c_{drag} \cdot A \cdot \frac{\rho}{2} \cdot \mathrm{v}^2 \tag{7.2}$$

$$\Leftrightarrow \quad \dot{\mathrm{v}}_{net} = \dot{\mathrm{v}}_{thrust} - c_{drag} \cdot A \cdot \frac{\rho}{2\,m} \cdot \mathrm{v}^2 \tag{7.3}$$

$$\Leftrightarrow \quad \frac{d\mathrm{v}}{dt} = \dot{\mathrm{v}}_{thrust} - c_{drag} \cdot A \cdot \frac{\rho}{2\,m} \cdot \mathrm{v}^2 \tag{7.4}$$

$$\Leftrightarrow \quad dt = \frac{d\mathrm{v}}{\dot{\mathrm{v}}_{thrust} - \frac{c_{drag}\,A\,\rho}{2\,m} \cdot \mathrm{v}^2} \quad . \tag{7.5}$$

Integration beider Seiten der Gleichung liefert:

$$\Leftrightarrow \quad \int_0^t dt' = \frac{1}{\dot{\mathrm{v}}_{thrust}} \cdot \int_0^{\mathrm{v}} \frac{d\mathrm{v}'}{1 - \frac{c_{drag}\, A\, \rho}{2\, m \cdot \dot{\mathrm{v}}_{thrust}} \cdot (\mathrm{v}')^2} \tag{7.6}$$

$$\Leftrightarrow \quad \int_0^t dt' = \frac{1}{\dot{\mathrm{v}}_{thrust}} \cdot \int_0^{\mathrm{v}} \frac{d\mathrm{v}'}{1 - k^2 \cdot (\mathrm{v}')^2} \quad \text{mit} \quad k = \sqrt{\frac{c_{drag}\, A\, \rho}{2\, m \cdot \dot{\mathrm{v}}_{thrust}}} \tag{7.7}$$

$$\Leftrightarrow \quad \int_0^t dt' = \frac{1}{\dot{\mathrm{v}}_{thrust}} \cdot \int_0^{\mathrm{v}} \frac{d\mathrm{v}'}{(1 + k\, \mathrm{v}') \cdot (1 - k\, \mathrm{v}')} \ . \tag{7.8}$$

Mit Hilfe der Partialbruchzerlegung erhält man:

$$\Leftrightarrow \quad \int_0^t dt' = \frac{1}{\dot{\mathrm{v}}_{thrust}} \cdot \left[\int_0^{\mathrm{v}} \frac{1/2}{(1 + k\, \mathrm{v}')} d\mathrm{v}' + \int_0^{\mathrm{v}} \frac{1/2}{(1 - k\, \mathrm{v}')} d\mathrm{v}' \right] \tag{7.9}$$

$$\Leftrightarrow \quad t = \frac{1}{\dot{\mathrm{v}}_{thrust}} \cdot \left[\frac{\ln(1 + k\, \mathrm{v}')}{2\, k} - \frac{\ln(1 - k\, \mathrm{v}')}{2\, k} + C \right]_{\mathrm{v}'=0}^{\mathrm{v}'=\mathrm{v}} \tag{7.10}$$

$$\Leftrightarrow \quad t = \frac{1}{2\, k \cdot \dot{\mathrm{v}}_{thrust}} \cdot \ln\left(\frac{1 + k\, \mathrm{v}}{1 - k\, \mathrm{v}} \right) \ . \tag{7.11}$$

Wie in vielen Formelsammlungen (z. B. [41]) zu finden, ist die *Area*-Funktion des *Tangens Hyperbolicus* definiert als: $\operatorname{artanh}(x) = \frac{1}{2} \ln\left(\frac{1+x}{1-x}\right)$. Damit folgt:

$$\Leftrightarrow \quad t = \frac{\operatorname{artanh}(k\, \mathrm{v})}{k \cdot \dot{\mathrm{v}}_{thrust}} \tag{7.12}$$

$$\Leftrightarrow \quad \mathrm{v} = \frac{1}{k} \cdot \tanh(t \cdot k\, \dot{\mathrm{v}}_{thrust}) \tag{7.13}$$

$$\Rightarrow \quad \mathrm{v}(t) = \sqrt{\frac{2\, m \cdot \dot{\mathrm{v}}_{thrust}}{c_{drag}\, A\, \rho}} \cdot \tanh\left(t \cdot \sqrt{\frac{\dot{\mathrm{v}}_{thrust} \cdot c_{drag}\, A\, \rho}{2\, m}} \right) \ . \tag{7.14}$$

Gl. 7.14 ist nun die Funktion, die den Geschwindigkeitsverlauf des Flugroboters beschreibt. Für die Beschleunigung $\dot{\mathrm{v}}_{thrust}$ wird der Betrag des Vektors $\vec{\mathbf{a}}_{thrust}$ aus Gl. 6.21 eingesetzt. Da bei den folgenden Messungen nur der Nick-Winkel Θ variiert und die übrigen Winkel gleich Null gehalten werden, ergibt sich damit für die Beschleunigung $\dot{\mathrm{v}}_{thrust}$:

$$\dot{\mathrm{v}}_{thrust} = \mathrm{g} \cdot \tan(\Theta) \ . \tag{7.15}$$

Analog folgt für den Fall reiner Roll-Neigung $\dot{\mathrm{v}}_{thrust} = \mathrm{g} \cdot \tan(\Phi)$. Allgemein ergibt sich beim Setzen von nur einem Neigungswinkel ungleich Null für $\dot{\mathrm{v}}_{thrust}$ immer das Produkt aus Erdbeschleunigung g und *Tangens* des Neigungswinkels.[5]
Für t gegen unendlich strebt der *Tangens hyperbolicus* aus Gl. 7.14 gegen eins, weshalb der davorstehende Wurzelterm somit die Maximalgeschwindigkeit darstellt.

[5] Im übrigen ist Gl. 7.14 für $\dot{\mathrm{v}}_{thrust} = \mathrm{g}$ die Gleichung für die Geschwindigkeitsentwicklung beim freien Fall. (Siehe [42]; mit analoger Herleitung der Gleichung.)

Um nun die gemessenen Daten fitten zu können, muß die Geschwindigkeitsgleichung noch integriert werden. Ausgehend von Gl. 7.13 folgt:

$$\frac{ds}{dt} = \frac{1}{k} \cdot \tanh\left(t \cdot k\, \dot{\mathrm{v}}_{thrust}\right) \tag{7.16}$$

$$\Leftrightarrow \quad \int_0^s ds' = \frac{1}{k} \cdot \int_0^t \tanh\left(t' \cdot k\, \dot{\mathrm{v}}_{thrust}\right) dt' \ . \tag{7.17}$$

Mit dem Wissen über $\int \tanh(a\,x)\,dx = \frac{1}{a}\,\ln\left(\cosh(a\,x)\right) + C$ ergibt sich:

$$\Leftrightarrow \qquad s = \frac{1}{k} \cdot \left[\frac{\ln\left(\cosh\left(t' \cdot k\, \dot{\mathrm{v}}_{thrust}\right)\right)}{k \cdot \dot{\mathrm{v}}_{thrust}} + C\right]_{t'=0}^{t'=t} \tag{7.18}$$

$$\Leftrightarrow \qquad s = \frac{1}{k^2 \cdot \dot{\mathrm{v}}_{thrust}} \cdot \ln\left(\cosh\left(t \cdot k\, \dot{\mathrm{v}}_{thrust}\right)\right) \tag{7.19}$$

$$\Leftrightarrow \qquad s = \frac{2\,m}{c_{drag}\,A\,\rho} \cdot \ln\left(\cosh\left(t \cdot \sqrt{\frac{\dot{\mathrm{v}}_{thrust} \cdot c_{drag}\,A\,\rho}{2\,m}}\right)\right) \ . \tag{7.20}$$

Da es keine Hinweise auf die genauen Werte von c_{drag}, A und ρ gibt, wird das gemeinsame Produkt von ihnen durch eine Konstante κ ersetzt:[6]

$$\Rightarrow \qquad s\left(t\right) = \frac{2\,m}{\kappa} \cdot \ln\left(\cosh\left(t \cdot \sqrt{\frac{\dot{\mathrm{v}}_{thrust} \cdot \kappa}{2\,m}}\right)\right) \ . \tag{7.21}$$

7.1.2.2 Ergebnisse

Die gemessenen Daten wurden mit dem Programm *Gnuplot* gefittet, das zur Fehlerminimierung das Verfahren der *kleinsten Fehlerquadrate* verwendet. Zur Darstellung als Graphen in den folgenden Abbildungen wurde ebenfalls Gnuplot verwendet. Zuvor mußten die Daten jedoch noch normalisiert werden, indem die Startzeit und Startdistanz auf Null gesetzt und die übrigen Meßwerte um diese Differenz verschoben wurden.

In Abb. 7.2 ist exemplarisch ein Fit für 5 ° Neigung in Flugrichtung (also $\Theta = -5\,°$) dargestellt. Die Meßdaten liegen in Abständen von ungefähr 60 ms vor, wobei jedoch nur jede dritte Messung in Abb. 7.2 eingezeichnet ist.[7] Es läßt sich anhand des angegebenen Fehlers und aus dem vergrößerten Ausschnitt gut erkennen, daß der Fit sehr gut paßt, die Messung somit mit hoher Wahrscheinlichkeit sehr exakt erfolgt war. Der sich aus diesem Fit ergebende Wert für κ liegt bei $0,5697\,\mathrm{kg/m}$.

[6] Der ungefähre Wert der Luftdichte ρ ist sehr wohl bekannt. Er beträgt unter Normalbedingungen $1,204\,\frac{\mathrm{kg}}{\mathrm{m}^3}$ (20 °C und 1013,25 hPa). Allerdings kann er für die eigenen Experimente nicht ganz exakt angegeben werden, weshalb ρ hier nicht im Einzelnen betrachtet wurde.

[7] Bei einigen anderen Messungen lagen die Meßdaten in 30 ms Abständen vor. Dieser Unterschied ist auf unterschiedliche Lichteinstellungen der verwendeten Kamera zurückzuführen.

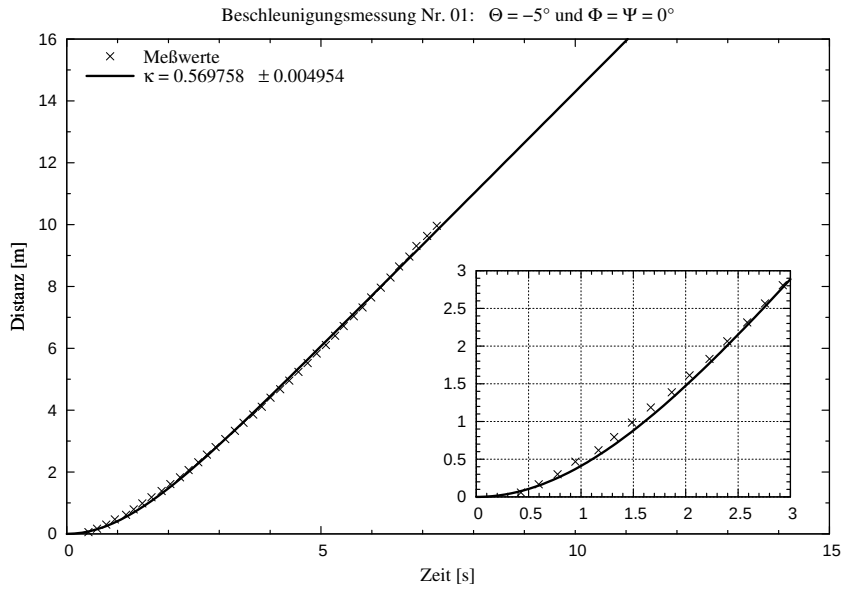

Abbildung 7.2: Darstellung der Daten eines Meßdurchlaufs bei 5 ° Neigung zusammen mit dem Graphen der an sie gefitteten Funktion. Im vergrößerten Ausschnitt läßt sich die nur geringe Abweichung der Meßwerte vom Funktions-Fit erkennen. Der gesuchte Fit-Parameter κ ist ebenfalls angegeben.

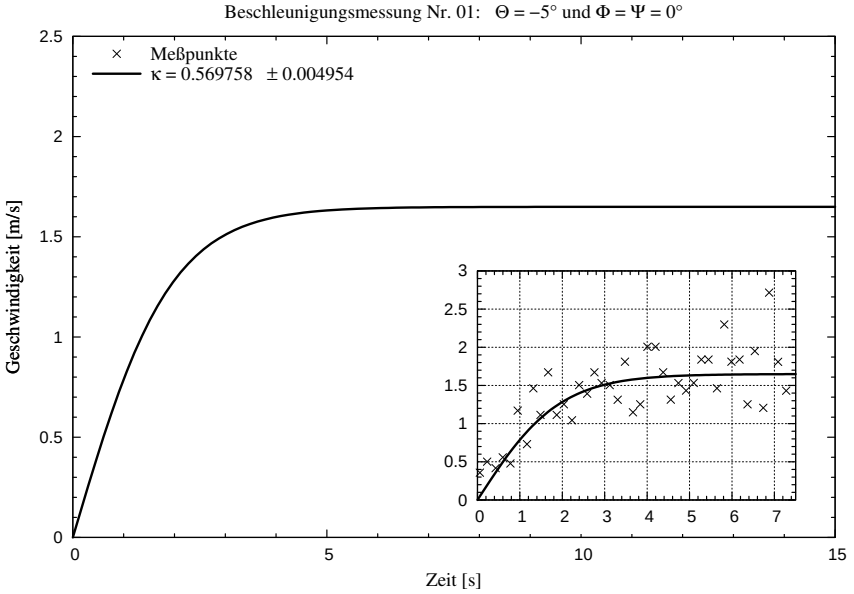

Abbildung 7.3: Darstellung des aus dem Fit in Abb. 7.2 resultierenden Geschwindigkeitsverlaufs bei 5 ° Neigung. Die Maximalgeschwindigkeit liegt bei ungefähr 1,65 m/s. Im vergrößerten Ausschnitt sind zusätzlich die aus den Meßwerten gebildeten Momentangeschwindigkeiten vermerkt. (Man erkennt leicht, daß aufgrund der Differentiation der Meßwerte die Genauigkeit der Daten abgenommen hat, weshalb ein Fit nur mit den Originaldaten an die Distanz-Gleichung 7.21 sinnvoll ist.)

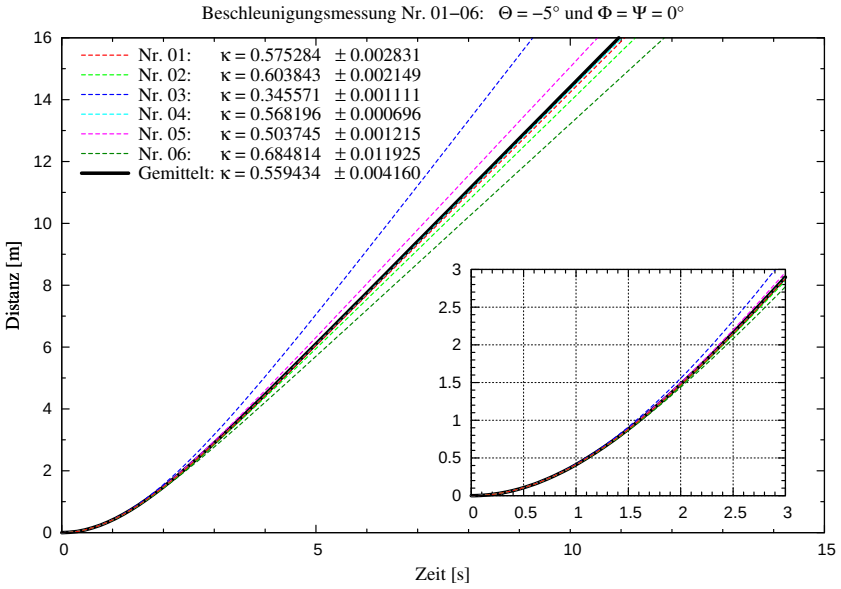

Abbildung 7.4: Die Graphen aller gefitteten Durchläufe bei 5 ° Neigung zusammen mit dem mittlerem Fit über die gesamten Daten. Für den mittleren Fit ergibt sich für κ ein ungefährer Wert von $0,5594\,\mathrm{kg/m}$.

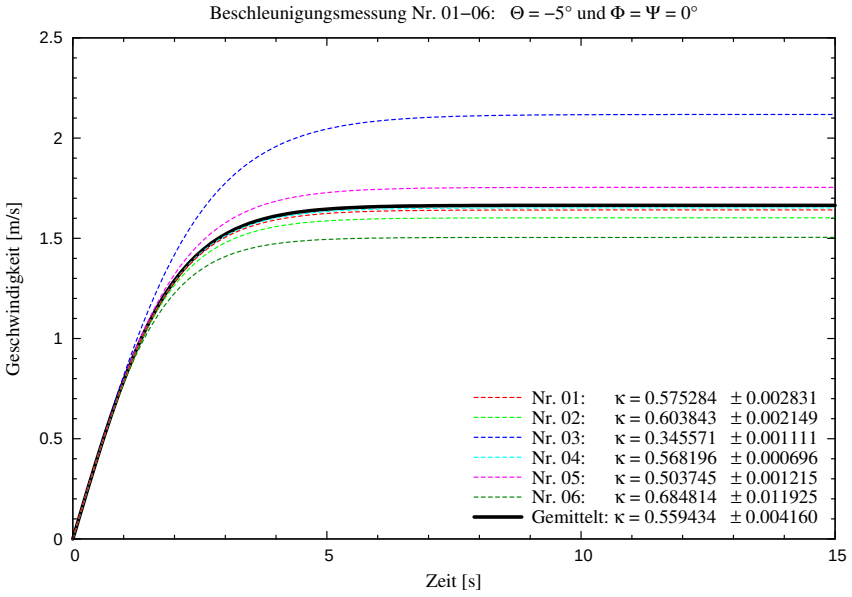

Abbildung 7.5: Die Geschwindigkeitsgraphen aller Durchläufe bei 5 ° Neigung zusammen mit dem Geschwindigkeitsgraphen für den mittleren Fit über die gesamten Daten. Die Maximalgeschwindigkeit des mittleren Fits liegt bei ungefähr $1{,}66\,\mathrm{m/s}$.

Die zugehörige Geschwindigkeitskurve ist in Abb. 7.3 dargestellt. Aus ihr läßt sich erkennen, daß die Maximalgeschwindigkeit bei ungefähr 1,65 m/s liegt und nach etwa sieben Sekunden erreicht wird.

Die aus allen ausgewerteten Messungen für einen Neigungwinkel von 5 ° resultierenden Distanz-Kurven, sowie die mittlere, über alle Meßdaten gefittete Distanz-Kurve sind in Abb. 7.4 aufgetragen. Man erkennt, daß die Messung Nr. 03 stärker von den übrigen Messungen abweicht. Wie im vergrößerten Ausschnitt gut zu erkennen, stimmt ihre Kurve in den ersten 1,5 Sekunden noch gut mit den übrigen Kurven überein, steigt dann jedoch steiler an. Die nachträgliche Kontrolle mit den von der Roboterkamera aufgezeichneten Daten ergab, daß der Roboter in dieser Messung ein wenig neben der eigentlichen Fluglinie, nämlich dichter an der Kamera, geflogen war. Diesen Abstand hielt er jedoch über die gesamte Flugstrecke exakt ein, weshalb diese Messung nicht verworfen wurde.
Messung Nr. 05 hingegen stimmt ziemlich gut mit dem gemittelten Fit überein, obwohl der Startpunkt des Roboters leicht außerhalb des aufgezeichneten Bildbereichs lag. Dieser konnte jedoch näherungsweise nachträglich ermittelt werden. Die Startzeit wiederum war aufgrund der Signal-Leuchtdiode genau bekannt.
Die zu den Messungen gehörenden Geschwindigkeitsverläufe sind in Abb. 7.5 dargestellt. Daraus läßt sich ablesen, daß die Maximalgeschwindigkeit des mittleren Fits bei 1,66 m/s liegt, was ziemlich genau dem zuvor ermittelten Wert für Messung Nr. 01 entspricht.

Die mittleren Geschwindigkeitsverläufe für verschiedene untersuchte Neigungswinkel sind in Abb. 7.6 dargestellt. Es wird deutlich, daß mit einem ansteigenden Neigungswinkel die Maximalgeschwindigkeit schneller erreicht wird. Allerdings wird bei einer Verdopplung des Neigungswinkels die resultierende Maximalgeschwindigkeit nicht ebenfalls verdoppelt, sondern diese verändert sich um einen Faktor kleiner als zwei. Dies deckt sich mit den Erwartungen aus der Theorie, die davon ausgeht, daß der Luftwiderstand proportional zum Quadrat der Geschwindigkeit des Roboters wächst.

Die Auswertung der Meßdaten für Neigungswinkel kleiner als 4 ° stellt sich als schwierig heraus, da hierbei der Drift des Roboters seine eigentliche Bewegung häufig überlagerte. Jedoch existierten ausreichende Meßdaten, um dennoch akzeptable Kurven zu erhalten.
Hingegen sind die Kurven für 15 ° und 25 ° mit Einschränkungen behaftet. Die verwendete Teststrecke war nicht lang genug, um eine ausreichende Anzahl von Daten aufzuzeichnen. Zudem lagen für diese hohen Neigungswinkel generell nur wenige Messungen zur Auswertung vor. Es ist auch zu berücksichtigen, daß 25 ° den maximalen Neigungswinkel darstellt, den der Roboter einnehmen kann, was die Ursache für mögliche Meßfehler sein könnte.

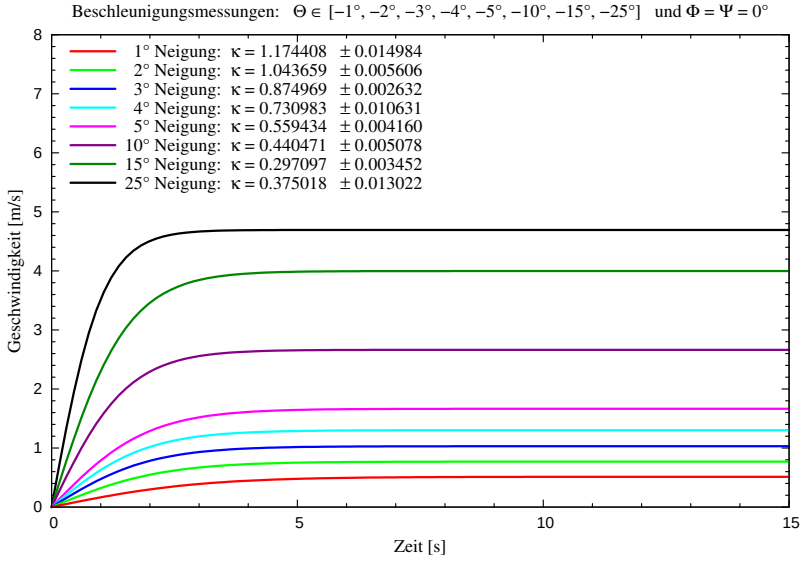

Abbildung 7.6: Die gemessenen mittleren Geschwindigkeitsverläufe für unterschiedliche Neigungswinkel. Die Geschwindigkeiten bei Neigungen kleiner als 4 ° waren schwer zu bestimmen, da der Drift des Roboters dort eine nicht unerhebliche Rolle spielt.

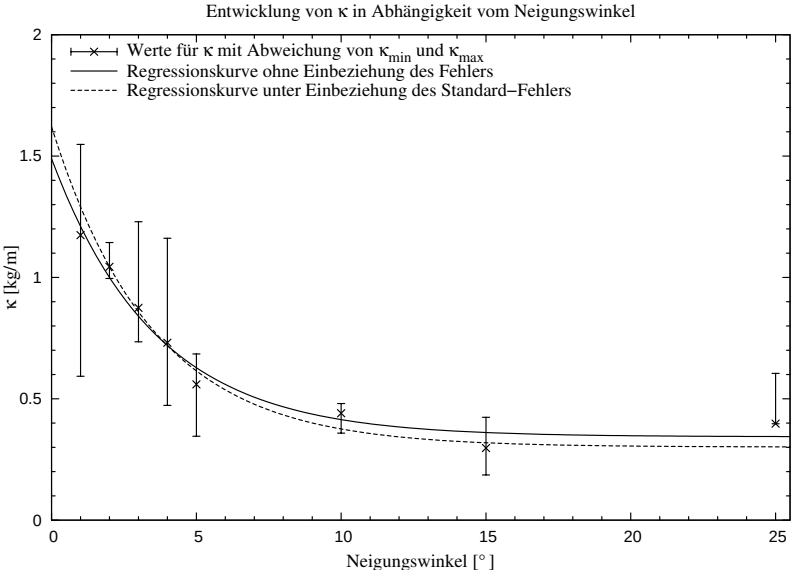

Abbildung 7.7: Der Verlauf der Variablen κ in Abhängigkeit von der Neigung. Der Wert für κ nimmt bei stärkerer Neigung ab. Die Fehlerbalken bezeichnen die maximalen und minimalen Werte von κ, die bei den Messungen für den jeweiligen Winkel auftreten. Die beiden Regressionskurven wurden mit der Funktion $f(x) = a \cdot \exp\left(-\left(b \cdot x + c\right)\right) + d$ gebildet, wobei zur Erstellung der gestrichelten Kurve die einzelnen Werte von κ mit ihren Standard-Fehlern gewichtet wurden, während für die durchgezogene Kurve κ gleichmäßig gewichtet wurde.

Es fällt auf, daß die Werte für κ bei stärkerer Neigung abnehmen, was besonders gut in Abb. 7.7 zu erkennen ist. Nur bei 25 ° steigt der erhaltene Wert im Vergleich zum vorherigen bei 15 ° wieder etwas an, was jedoch auf Meßungenauigkeiten beruhen könnte. Diese Abnahme von κ in Abhängigkeit von der Neigung verwundert, denn man würde zunächst vermuten, daß sich bei stärkerer Neigung die Oberfläche quer zur Flugrichtung aufgrund der Schiefstellung des Roboters vergrößern und der Luftwiderstand damit zunehmen würde. Das scheint jedoch nicht der Fall zu sein, was darauf schließen läßt, daß die sich drehenden Rotoren nicht als eine luftundurchlässige Scheibe angenommen werden dürfen. Vielmehr erscheint es so, daß die Rotoren, welche die Luft schließlich von oben nach unten durch ihre Rotationsebene saugen und dabei beschleunigen (*Strahltheorie*; [43]), bei größerer Schiefstellung mehr Luft aus Flugrichtung einsaugen. Damit würden diese Luftteilchen weniger stark bremsen und der Flugroboter dadurch schneller beschleunigen. Sehr wahrscheinlich jedoch ist der maximale Luftdurchsatz durch die Rotoren beschränkt. Somit können die Rotoren die Luft nur bis zu einem gewissen Grad beschleunigen, so daß der Luftwiderstand niemals Null wird. Sobald eine optimale Geschwindigkeit überschritten wird, wird der Luftwiderstand wieder stärker zunehmen, womit κ ebenfalls ansteigen sollte. Abb. 7.7 läßt die Vermutung zu, daß diese Geschwindigkeit bei einer Neigung von ungefähr 15 ° erreicht wird und deshalb der Luftwiderstand danach wieder stärker zunimmt. Allerdings kann dies aufgrund der bereits beschriebenen Meßunsicherheiten nicht ohne größeren experimentellen Aufwand (Windkanal) verifiziert werden.
Als weitere mögliche Erklärung für die Abnahme von κ bei höheren Neigungwinkeln könnte die Tatsache dienen, daß die Oberseite des Zentralkörpers des verwendeten Flugroboters abgerundet ist und damit eine bessere Aerodynamik aufweist. Während bei einem Flug mit geringer Neigung die Luft seitlich gegen den Zentralkörper strömt und es speziell an der Rotationstrommel der Kamera zu größeren, bremsend wirkenden Verwirbelungen kommen dürfte, strömt sie bei starken Neigungen hingegen vermehrt gegen die abgerundete Oberseite des Zentralkörpers. Diese lenkt die Luft seitlich um den Zentralkörper und vermindert damit deren Bremswirkung. Zusätzlich verschwindet die Kamera-Rotationstrommel bei größerem Neigungswinkel zunehmend hinter dem Zentralkörper und bietet damit weniger Angriffsfläche für die Luft. Ob sich jedoch an den Übergangskanten nicht dennoch bremsende Verwirbelungen ausbilden, kann nicht mit Gewißheit gesagt werden. Zusätzliche Flugmessungen ohne Kamera wurden nicht durchgeführt, so daß keine Aussage über deren Einfluß auf den Luftwiderstand gemacht werden kann. Grundsätzlich führen diese Überlegungen jedoch dazu, daß sich der in κ enthaltene Luftwiderstandsbeiwert c_{drag} bei größeren Neigungen vermindern müßte. Genauere Untersuchungen dazu können wiederum nur im Windkanal durchgeführt werden.

Die in Abb. 7.7 durch die Werte für κ gelegten Ausgleichskurven wurden mit einer Funktion

$$f(x) = a \cdot \exp\left(-\left(b \cdot x + c\right)\right) + d \tag{7.22}$$

gebildet. Für die darin enthaltenen Koeffizienten a bis d ergaben sich bei der durchgezogen gezeichneten Ausgleichskurve folgende Werte:

$$a = 0,965934 \quad b = 0,278448 \quad c = -0,170570 \quad d = 0,343490 \ . \tag{7.23}$$

Bei der gestrichelt gezeichneten Ausgleichskurve, die aus mit ihren Standard-Fehlern unterschiedlich gewichteten Werten von κ gebildet wurde, resultierten hingegen folgende Werte für die Koeffizienten:

$$a = 1,039424 \quad b = 0,287179 \quad c = -0,238395 \quad d = 0,301074 \ . \tag{7.24}$$

Ob der Verlauf für κ durch eine dieser Kurven korrekt beschrieben wird, kann jedoch nicht mit Sicherheit gesagt werden. Die zugehörige Funktionsgleichung Gl. 7.22 leitet sich aus keiner bekannten physikalischen Bedingung her, sondern wurde lediglich aufgrund ihrer ähnlichen Anpassung gewählt. Ob mit dieser Kurve auch die Entwicklung von κ für Neigungen entgegen der Bewegungsrichtung, also in Abb. 7.7 für negative Neigungswinkel, tatsächlich korrekt beschrieben wird, kann ebenfalls nicht mit Sicherheit gesagt werden. Dazu wurden keine Versuche durchgeführt. Jedoch ist zu vermuten, daß sich der Wert für κ erhöhen dürfte, zum einen aufgrund der vermehrten Anströmung der Luft auf die wenig aerodynamisch geformte Unterseite des Roboter-Zentralkörpers mit der Kamera-Rotationstrommel und den sich damit vermutlich bildenden bremsenden Verwirbelungen, zum anderen aufgrund der gegensätzlichen Beschleunigung der Luft durch die Rotoren. Normale Freifeldflüge haben zumindest gezeigt, daß sich der Roboter bei starker Gegensteuerung in kürzester Zeit auf Null abbremsen läßt. Deshalb wurde im weiteren für negative Neigungswinkel der fortgesetzte Verlauf dieser Kurve als korrekt angenommen.

7.1.3 Gesamtgleichung für ξ

Wie im vorherigen Abschnitt bereits erwähnt, beschreibt Gl. 7.22 nur die Entwicklung der Luftwiderstandskomponente κ für Neigungen in Richtung der jeweils aktuellen Bewegung, also in Richtung des im vorherigen Zeitschritt berechneten Geschwindigkeitsvektors $\vec{\mathbf{v}}_{k-1,k}$ aus Gl. 6.5. Für Neigungen exakt entgegengesetzt zu dieser Bewegungsrichtung liefert sie ebenfalls Werte für κ, die, obwohl sie nicht experimentell verifiziert werden konnten, im Folgenden als gültig angesehen und verwendet werden. Außer für diese beiden Richtungen macht Gl. 7.22 jedoch direkt keinerlei Aussage.

Der Fall für Neigungen quer zur Bewegungsrichtung wurde nicht untersucht. Jedoch wird in erster Näherung davon ausgegangen, daß sich der Wert für κ bei Neigungen exakt orthogonal zur Bewegungsrichtung, also $\pm 90\,^\circ$ um die Hochachse rotiert, dem Wert für $0\,^\circ$ Neigung aus Gl. 7.22 annähert. Dies wäre bei Verwendung der Koeffizienten aus Gl. 7.23 für die in Abb. 7.7 durchgezogen gezeichnete Kurve somit $\kappa = 1,4891\,\frac{kg}{m}$.
Für Neigungen, die weder orthogonal noch direkt in oder entgegengesetzt zur Flugrichtung verlaufen, lassen sich nun Werte für κ mit Hilfe des Winkels zwischen der geneigten Hochachse des Roboters und der Bewegungsrichtung interpolieren. Um den zugehörigen Neigungswinkel β zu erhalten muß dieser Winkel nur noch von $90\,^\circ$ abgezogen werden und das erhaltene Ergebnis kann in Gl. 7.22 eingesetzt werden.

Die geneigte Hochachse des Roboters entspricht in Koordinaten des verwendeten *Roboter-Referenz*-Koordinatensystems dem Einheitsvektor der Rotorbeschleunigung a_{rotor}. Dieser kann direkt aus Gl. 6.16 übernommen werden. Der schon im korrekten Koordinatensystem definierte Geschwindigkeitsvektor $\vec{\mathbf{v}}_{k-1,k}$ muß schließlich noch normiert werden, womit sich für den Winkel β mit der Definition des Skalarprodukts ergibt:

$$\beta = 90^\circ - \arccos\left(\frac{\vec{\mathbf{a}}_{rotor}}{\|\vec{\mathbf{a}}_{rotor}\|} \cdot \frac{\vec{\mathbf{v}}_{k-1,k}}{\|\vec{\mathbf{v}}_{k-1,k}\|}\right) \tag{7.25}$$

$$= 90^\circ - \arccos\left(\begin{pmatrix} \sin(\Phi_{k-1}) \\ -\cos(\Theta_{k-1})\cos(\Phi_{k-1}) \\ -\sin(\Theta_{k-1})\cos(\Phi_{k-1}) \end{pmatrix} \cdot \frac{1}{\|\vec{\mathbf{v}}_{k-1,k}\|} \begin{pmatrix} \mathrm{v}_x \\ \mathrm{v}_y \\ \mathrm{v}_z \end{pmatrix}_{k-1,k}\right)$$

$$= 90^\circ - \arccos\left(\frac{\mathrm{s}_\Phi \cdot (\mathrm{v}_x)_{k-1,k} - \mathrm{c}_\Theta\,\mathrm{c}_\Phi \cdot (\mathrm{v}_y)_{k-1,k} - \mathrm{s}_\Theta\,\mathrm{c}_\Phi \cdot (\mathrm{v}_z)_{k-1,k}}{\sqrt{(\mathrm{v}_x)^2_{k-1,k} + (\mathrm{v}_y)^2_{k-1,k} + (\mathrm{v}_z)^2_{k-1,k}}}\right) \quad . \tag{7.26}$$

Hinweis: Im letzten Umformungsschritt wurde die gleiche verkürzte Schreibweise für cos und sin verwendet, wie schon zuvor in Abschnitt 6.1.

Somit kann Gl. 6.11 nun abgewandelt werden und die Funktionskonstante ξ ergibt sich zu:

$$\xi(\Phi, \Theta, \vec{\mathbf{v}}) = \frac{\kappa}{2\,m} = \frac{1}{2\,m} \cdot (a \cdot \exp(-(b \cdot \beta + c)) + d) \tag{7.27}$$

$$= \frac{a \cdot \exp\left(-\left(b \cdot \left(90^\circ - \arccos\left(\frac{\mathrm{s}_\Phi \cdot \mathrm{v}_x - \mathrm{c}_\Theta\,\mathrm{c}_\Phi \cdot \mathrm{v}_y - \mathrm{s}_\Theta\,\mathrm{c}_\Phi \cdot \mathrm{v}_z}{\sqrt{\mathrm{v}_x^2 + \mathrm{v}_y^2 + \mathrm{v}_z^2}}\right)\right) + c\right)\right) + d}{2\,m} \quad , \tag{7.28}$$

mit den Koeffizienten a bis d so gewählt, wie in Gl. 7.23 angegeben.

7.2 Test des Geschwindigkeitsmodells

Zum Testen des nun vollständig aufgestellten Modells wurden die Geschwindigkeitsentwicklungen für einige Flugkommandos untersucht, indem simulierte Testläufe durchgeführt wurden. Zu dem Zweck wurden feste Neigungswinkel vorgegeben, aus denen das Modell mit einem fest gewählten Δt von 20 ms die simulierten Geschwindigkeitsverläufe berechnete.

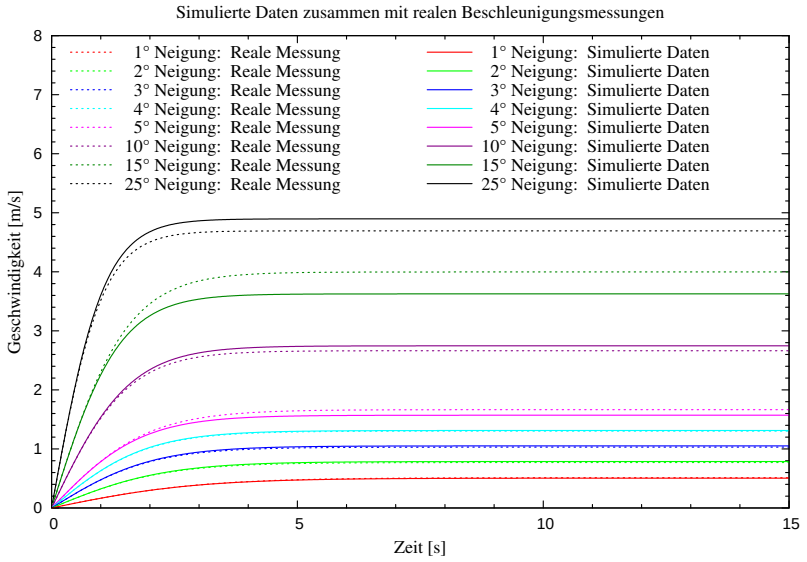

Abbildung 7.8: Vergleich realer und simulierter Geschwindigkeitsverläufe. Es sind erneut die real gemessenen Geschwindigkeitsverläufe aus Abb. 7.6 für unterschiedliche Neigungswinkel in Bewegungsrichtung dargestellt, diesmal jedoch zusammen mit den des entwickelten Modells simulierten Daten. Diese wurden mit einer Samplingrate von 50 Hz, also einem Δt von 20 ms, berechnet.

Abb. 7.8 stellt die mit dem entwickelten Modell simulierten Daten den real gemessenen Geschwindigkeitsverläufen aus Abb. 7.6 gegenüber. Man erkennt, daß der real gemessene Geschwindigkeitsverlauf vom Modell sehr gut wiedergegeben wird. Es gibt jedoch einen Versatz in der Maximalgeschwindigkeit, der vor allem für höhere Neigungswinkel größer ausfällt. Dies ist darauf zurückzuführen, daß die aus den Beschleunigungsversuchen gewonnene und in Abb. 7.6 dargestellte Verlaufskurve für κ bei niedrigeren Neigungswinkeln besser mit den gemittelten Meßwerten übereinstimmte als bei höheren Neigungswinkeln. Dennoch zeigt der grundsätzlich gleichförmige Verlauf der simulierten Werte in Abb. 7.8 eindeutig, daß das Modell geeignet ist, die reale Geschwindigkeitsentwicklung wiederzugeben.
Die simulierten Daten wurden für die entsprechend angegebenen Nick-Winkel erhalten, wobei Roll- und Gier-Winkel bei Null belassen wurden. Die mit ihnen dargestellten Geschwindigkeitsverläufe lassen sich jedoch ebenso gut für mit negativen Vorzeichen versehene Nick-Winkel oder auch für auf Null gesetzte Nick- und Gier-Winkel

und dafür entsprechend variierte Roll-Winkel erhalten. Analog ergeben sie sich für jede andere Kombination von Nick- und Roll-Winkel bei einem Gier-Winkel von Null, sofern die aus ihnen gesamt-resultierenden Winkel den dargestellten Neigungswinkel (oder ihren negativen Pendants) entsprechen. Der einzige Unterschied besteht im Winkel zwischen der Orientierung des Roboters und seiner Bewegungsrichtung.

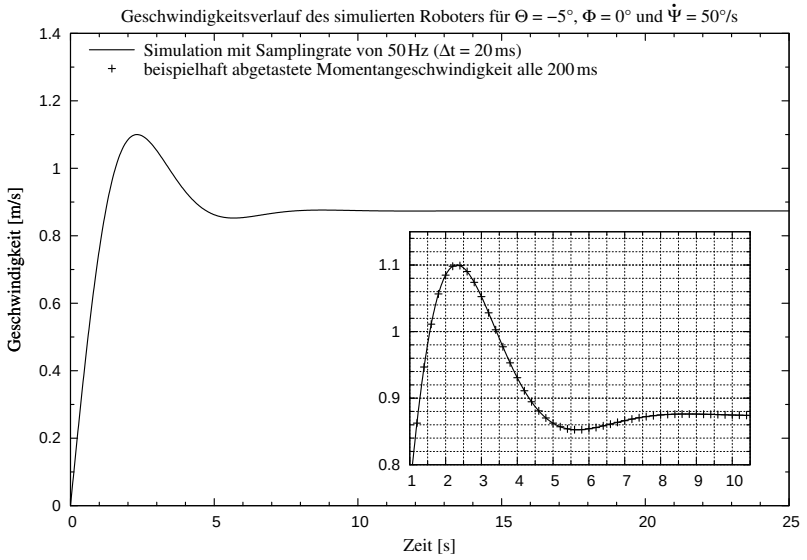

Abbildung 7.9: Der Geschwindigkeitsverlauf bei einer simulierten Kreisflugbewegung. Diese wurde mit einer Ausgangsgeschwindigkeit von 0 m/s mit Neigungswinkel von $\Theta = -5\,^\circ$ und $\Phi = 0\,^\circ$ sowie einer Gier-Winkelgeschwindigkeit von $\dot{\Psi} = 50\,^\circ/s$ erzeugt, was bei der verwendeten Samplingrate von 50 Hz ($\Delta t = 20$ ms) einem Gierwinkel von $\Psi = 1\,^\circ$ pro Zeitschritt entspricht. Auffällig sind das lokale Maximum bei 2,32 s sowie das lokale Minimum bei 5,68 s. Diese sind im vergrößerten Ausschnitt gut zu erkennen und wurden zusammen mit exemplarisch alle 200 ms abgetasteten Momentangeschwindigkeiten dargestellt.

Interessant ist nun der Fall für Gier-Winkel $\Psi \neq 0\,^\circ$. Abb. 7.9 stellt den simulierten Geschwindigkeitsverlauf für eine Bewegung mit Neigungen von $\Theta = -5\,^\circ$ und $\Phi = 0\,^\circ$ sowie einer Gier-Winkelgeschwindigkeit von $\dot{\Psi} = 50\,^\circ/\text{s}$ dar. Die zur Simulation verwendete Zykluszeit betrug ebenfalls 20 ms, womit sich die Winkelgeschwindigkeit $\dot{\Psi}$ zu einem Gier-Winkel von $\Psi = 1\,^\circ$ pro Zeitschritt ergab.
Auffällig sind das lokale Maximum bei 2,32 s sowie das folgende lokale Minimum bei 5,68 s, wobei auch noch weitere lokale Extrema auftraten, deren Unterschied jedoch nur sehr gering waren. Laut den ermittelten Daten schwingt die Geschwindigkeit bis zum endgültigen Wert von 0,873690 m, welcher nach 23,16 s erreicht wird, noch drei weitere Male hin und her. Allerdings beschränkt sich schon die insgesamt dritte Änderung nur noch auf die 4 Stelle hinter dem Komma.

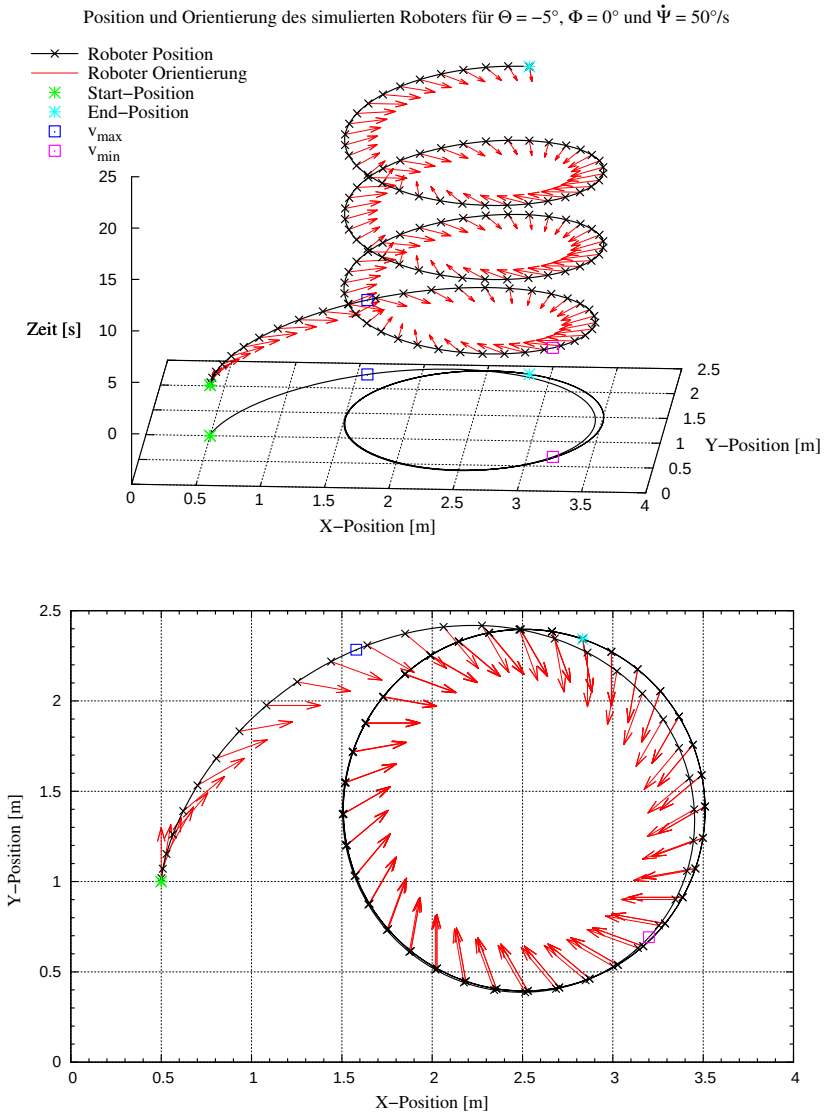

Abbildung 7.10: Rekonstruktion des zum Geschwindigkeitsverlauf in Abb. 7.9 gehörenden simulierten Kreisflugs. Dargestellt sind jeweils die ersten 25 Sekunden, zum einen als reine 2D-Ansicht, zum anderen als 3D-Ansicht, wobei die Höhe den zeitlichen Verlauf angibt. Zur besseren Darstellung ist nur alle 200 ms die Position und Orientierung des simulierten Roboters eingetragen. Zusätzlich sind die Positionen der beiden in Abb. 7.9 erkennbaren Maximal- und Minimalgeschwindigkeiten markiert. Der Kreisradius von ca. 1 m entspricht im übrigen ziemlich genau dem aus Erfahrungswerten im praktischen Einsatz zu erwartenden Kreisradius.

Die aus diesen simulierte Geschwindigkeitsdaten rekonstruierte Bewegung ist in Abb. 7.10 dargestellt. Aus ihr wird ersichtlich, daß der Roboter mit dem gesetzten Kommando eine Kreisbewegung vollführt hat, was zu erwarten war. Die beiden gut sichtbaren Extrema der ersten Schwingung sind ebenfalls eingezeichnet. Man erkennt auch, daß der Roboter etwas mehr als eine ganze (schwankende) Kreisbewegung benötigt, bis er sich auf der endgültigen Kreisbahn befindet.

Man kann in der 2D-Darstellung in Abb. 7.10 aus dem Kurvenverlauf und der eingezeichneten Orientierung gut erkennen, daß der Roboter bei v_{max} noch nicht die endgültige horizontale Verdrehung zur Geradeausrichtung erreicht hat. Sie liegt für v_{max} bei 49,9011 °, während die endgültige Verdrehung, die ungefähr nach 18,74 Sekunden erreicht ist, bei 61,6610 ° liegt. Vom Start bis zur Position von v_{max} hat der effektive Neigungswinkel β aus Gl. 7.26 von seinem ursprünglichen Wert von 5 ° Neigung auf 3,1718 ° abgenommen. Damit einher geht eine Erhöhung der Luftwiderstandskonstanten κ bzw. ξ.
Ca. 300 ms bevor die Position von v_{max} erreicht ist, hat die horizontale Verdrehung zwischen der Orientierung des Roboters und seiner Translationsrichtung den Wert von 45 ° erreicht. Ab diesem Punkt ist der zusätzliche Vortrieb also stärker quer zur jeweils aktuellen Translationsrichtung gerichtet als parallel dazu. Es wäre zu vermuten gewesen, daß sich aufgrund dessen die Maximalgeschwindigkeit an diesem Punkt ergeben müßte. Wahrscheinlich spielt aber der simulierte Luftwiderstand noch eine Rolle, die nicht so leicht ersichtlich ist, weshalb v_{max} erst 300 ms später erreicht wird. Wieso genau es zu diesem anfänglichen Schwingverhalten kommt, ist nicht ganz klar ersichtlich, allerdings konnte dies beim realen Roboter ebenfalls beobachtet werden.

Leider standen keine geeigneten Meßinstrumente zur Verfügung, um diese Kreisbewegung beim realen Flugroboter untersuchen zu können. Ebenso war es im Rahmen dieser Arbeit nicht möglich, das Modell unter Verwendung des Kalman-Filters und des *Kanatani*-Algorithmus' in detaillierter Weise zu testen. Rein optisch durchgeführte Ad-hoc-Versuche legten jedoch die Vermutung nahe, daß das entwickelte Modell auch in Zusammenarbeit mit dem beschriebenen Kalman-Filter im Stande ist, eine Eigenbewegungsschätzung zu verbessern. Die gesamte Berechnungsdauer eines solchen Kalman-Filter-Durchlaufs betrug dabei im Mittel nur rund eine halbe Millisekunde.[8]

[8] Zur Implementation des Kalman-Filters wurde eine OpenSource Bibliothek [44] verwendet, die dessen interne Berechnungen mit optimierten Algorithmen durchführt. Kompiliert wurde dieser, ebenso wie auch das gesamte restliche in Kapitel 5 vorgestellte Software-Framework, mit aktivierten Optimierungen des verwendeten GNU Compilers. Die Ad-hoc-Versuche wurden auf dem in Kapitel 2.2 beschriebenen Desktop-Computer mit einer vorab gespeicherten Flugsequenz (mitsamt zugehöriger Flugkommandos) durchgeführt.

8. Schlußfolgerung und Ausblick

Es konnte gezeigt werden, daß das in dieser Arbeit entwickelte Softwaresystem zur Ansteuerung des Roboters einwandfrei funktioniert. Denn seine reibungslose Funktionsweise war die Grundvoraussetzung für die ebenfalls in dieser Arbeit durchgeführten flugexperimentellen Messungen zur Geschwindigkeitsentwicklung. Die angestrebte hohe Modularität wurde ebenfalls erreicht. Es war ohne größeren Aufwand möglich, für die durchgeführten Beschleunigungsversuche ein weiteres (*Controller*-) Modul zu entwickeln und in das bestehende Softwaresystem einzubauen, welches auf Tastendruck vorab definierte Flugkommandos an den Roboter sendet und dabei über den Parallelport des als Bodenstation verwendeten Computers eine Leuchtdiode schaltet. Einige weitere, in dieser Arbeit nicht näher erwähnten Module, die z. B. die Verwendung anderer Methoden zur optischen Fluß Detektion ermöglichten, ließen sich ebenfalls ohne großen Aufwand in das entwickelte Softwaresystem einbauen und verwenden.

Die grundsätzliche Gültigkeit des in dieser Arbeit vorgestellten vereinfachten Modells zur Bestimmung der Flugbewegung des Roboters konnte für einfache Flugmanöver anhand von flugexperimentellen Untersuchungen mit anschließender Simulation der Bewegung durch eben dieses Modell gezeigt werden. Weitergehende Untersuchungen, inwieweit auch kompliziertere Flugmanöver korrekt prädiziert werden, konnten aufgrund des Fehlens adäquater Meßtechnik leider nicht durchgeführt werden und stehen somit noch aus. Demzufolge konnten auch Messungen, welche das Modell in Verwendung mit dem beschriebenen *Erweiterten Kalman-Filter* zur Verbesserung der vom *Kanatani*-Algorithmus gelieferten Eigenbewegungsschätzung untersuchen, nicht durchgeführt werden. Optisch durchgeführte Ad-hoc-Versuche stützen jedoch die These, daß das entwickelte Modell auch im Zusammenhang mit dem beschriebenen Kalman-Filter geeignet ist, eine aus dem optischen Fluß gewonnene Eigenbewe-

gungsschätzung zu verbessern. Weiterer Forschungsarbeit zu diesem Punkt ist somit noch Raum gelassen.

Eine Einschränkung des aufgestellten Modell des Flugverhaltens betrifft jedoch die Annahme, daß die durch das jeweilige Flugkommando gesetzten Neigungen augenblicklich erreicht werden. Speziell bei größeren Änderungen der Neigung wird dies einen nicht ganz unerheblichen negativen Einfluß auf die Genauigkeit der Prädiktion haben. Die Latenz zwischen Absenden des Flugkommandos zum Roboter und dessen erster sichtbarer Reaktion konnte bei den durchgeführten Beschleunigungsversuchen zwar als Nebenprodukt gewonnen werden,[1] doch wären genauere Untersuchungen dazu erforderlich. Zur Zeitdauer, die der Roboter zur Einnahme einer neuen Neigung braucht, ließe sich wahrscheinlich über den Winkel zwischen der Ausgangsneigung und der Zielneigung, ähnlich zu der in Kapitel 7.1.3 beschriebenen Vorgehensweise, eine Aussage treffen. Dennoch sind genauere meßtechnische Untersuchungen dazu unabdingbar.
Natürlich könnte man auch stattdessen die an das Modell gegebenen Neigungswinkel vorab insoweit modifizieren, als daß sie nicht die zum Flugroboter gesendeten Kommandos, sondern die realerweise existierende momentane Fluglage des Roboters repräsentieren. Damit verlagert man das Problem jedoch nur aus dem Modell heraus auf eine andere Ebene. Denn nun gilt es, die aktuelle Lage genau bestimmen zu können. Dafür sind zwar auch in dem in dieser Arbeit verwendeten Roboter Sensoren eingebaut, jedoch wurde nicht getestet, wie zuverlässig sie arbeiten. Grundsätzlich wäre dies jedoch eine interessante und wichtige Aufgabe, die sicherlich zu einer Verbesserung der Bewegungsprädiktion des hier beschriebenen Modells führt.

Neben diesen Einschränkungen gibt es noch eine weitere, da das Modell nur eine Bewegung in der Ebene beschreibt, der Flugroboter aber sehr wohl in der Lage ist, seine Höhe zu verändern und dies auch tut. Ebenso können äußere Einflüsse wie Wind die Prädiktion entscheidend beeinflussen. Demzufolge ist das Modell leider mit Einschränkungen behaftet und stellt Anforderungen an die Umwelt, die unter Freilandbedingungen nur bedingt erfüllt werden.
Die Ergebnisse der vorliegenden Arbeit stellen jedoch eine gute Grundlage für die weitere Entwicklung dieses Flugroboters im Hinblick auf die Hard- als auch die Software dar. In Verbindung mit einem Zugang zu Details der Steuerung und der Implementation weiterer Sensoren zur Messung und Korrektur der Flugbewegungen können in nachfolgenden Arbeiten die hier erfolgreich begonnenen Entwicklungen zu Ende geführt werden.

[1] Sie schwankten zwischen ungefähr 150 und 200 ms.

Literaturverzeichnis

[1] Volpe, R.: Rover functional autonomy development for the mars mobile science laboratory. In: *Aerospace Conference, 2003. Proceedings. 2003 IEEE* Jet Propulsion Laboratory, California Institute of Technology, 2005, S. 2643–2652

[2] Schraft, Rolf D. ; Schmierer, Gernot: *Serviceroboter – Produkte, Szenarien, Visionen.* Springer, 1998

[3] Kiguchi, Kazuo ; Esaki, Ryo ; Fukuda, Toshio: Development of a wearable exoskeleton for daily forearm motion assist. In: *Advanced Robotics* 19 (2005), Nr. 7, S. 751–771

[4] Kitano, H. ; Asada, M. ; Kuniyoshi, Y. ; Noda, I. ; Osawa, E. ; Matsubara, H.: RoboCup: A challenge problem of AI. In: *AI Magazine* 18 (1997)

[5] Heinemann, Patrick: *Cooperative Multi-Robot Soccer in a Higly Dynamic Environment*, Universität Tübingen, Diss., 2007

[6] Parker, Lynne E.: Current state of the art in distributed autonomous mobile robotics. In: *Distributed Autonomous Robotic Systems*, Springer, 2000, S. 3–12

[7] Eugster, Hannes ; Nebiker, Stephan: UAV-based Augmented Monitoring – Real-time Georeferencing and Integration of Video Imagery with Virtual Globes. In: *The XXI ISPRS Congress* Bd. XXXVII. Beijing, China, Juni 2008. – ISSN 1682–1750, 1229ff

[8] Gerke, Markus: Dense Image Matching in Airborne Video Sequences. In: *The XXI ISPRS Congress* Bd. XXXVII. Beijing, China, Juni 2008. – ISSN 1682–1750, 639ff

[9] Adigbli, Patrick ; Grand, Christophe ; Mouret, Jean-Baptiste ; Doncieux, Stéphane: Nonlinear Attitude and Position Control of a Micro Quadrotor using Sliding Mode and Backstepping Techniques. In: *7th European Micro Air Vehicle Conference (MAV07).* Toulouse, 2007

[10] Hoffmann, Gabe ; Rajnarayan, Dev G. ; Wasl, Steven L. u. a.: The Stanford Testbed of Autonomous Rotorcraft for Multi Agent Control (STARMAC). In: *In Proceedings of the 23rd Digital Avionics Systems Conference* Bd. 2, 2004, S. 121–10

[11] Pounds, Paul ; Mahony, Rob ; Hynes, Peter ; Roberts, Jonathan: Design of a Four-Rotor Aerial Robot. In: *Proceedings of Australian Conference on Robotics and Automation (ACRA).* Auckland, New Zealand, November 2002, S. 145–150

[12] *Kapitel* 15. In: Rickheit, Gert (Hrsg.) ; Deutsch, Werner (Hrsg.) ; Hermann, Theo (Hrsg.): *Psycholinguistik / Psycholinguistics.* Berlin : Mouton de Gruyter, 2003 (Handbücher zur Sprach-und Kommunikationswissenschaft / Handbooks of Linguistics and Communication Science 24), S. 218

[13] In: Aitken, Peter ; Jones, Bradley: *C in 21 Tagen: Schritt für Schritt zum Profi.* Pearson Education, 2007, S. 652ff

[14] Wippermann, Hans-Wilm (Hrsg.): *Software-Architektur und modulare Programmierung: Tagung des German Chapter of the ACM (1986).* Stuttgart : Teubner, 1986

[15] *Kapitel* 23. In: Oualline, Steve: *Praktische C++-Programmierung: [mit Kontrollfragen und Übungen].* O'Reilly Germany, 2003, S. 415ff

[16] *Kapitel* 5.1.8. In: Gumm, Heinz-Peter ; Sommer, Manfred: *Einführung in die Informatik.* 7. Oldenbourg Wissenschaftsverlag, 2006, S. 404

[17] *Kapitel* 10. In: Stroustrup, Bjarne: *Die C++-Programmiersprache.* 4. Programmer's Choice, 2000, S. 237ff

[18] *Kapitel* 12. In: Stroustrup, Bjarne: *Die C++-Programmiersprache.* 4. Programmer's Choice, 2000, S. 331ff

[19] Gamma, Erich ; Helm, Richard ; Johnson, Ralph ; Vlissides, John: *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley Professional, 1995

[20] *Kapitel* 10. In: Mallot, Hanspeter A. ; Allen, John S.: *Computational Vision: Information Processing in Perception and Visual Behavior.* MIT Press, 2000, S. 200ff

[21] In: Jähne, Bernd ; Haussecker, Horst ; Geissler, Peter: *Handbook of Computer Vision and Applications.* Bd. 2: *Signal Processing and Pattern Recognition.* Academic Press, 1999, S. 311

[22] Horn, Berthold K. P. ; Schunk, Brian G.: Determining optical flow. In: *Artificial Intelligence* 17 (1981), S. 185–203

[23] Lucas, Bruce D. ; Takeo, Kanade: An iterative image registration technique with an application to stereo vision. In: *Proc. Seventh International Joint Conference on Artificial Intelligence*. Vancouver, Canada, 1981, S. 674–679

[24] Bruhn, Andrés ; Weickert, Joachim ; Schnörr, Christoph: Lucas/Kanade meets Horn/Schunck: combining local and global optic flow methods. In: *International Journal of Computer Vision* 61 (2005), Nr. 3, S. 211–231

[25] Bouguet, Jean-Yves: *Pyramidal Implementation of the Lucas Kanade Feature Tracker: Description of the algorithm*. 2002

[26] Bruss, Anna R. ; Horn, Berthold K. P.: Passive Navigation. In: *Computer Vision, Graphics, and Image Processing* 21 (1983), S. 3–20

[27] Jepson, Allan D. ; Heeger, David J.: Linear Subspace Methods for Recovering Translational Direction. In: *Spatial Vision in Humans and Robots*, Cambridge University Press, 1993, S. 39–62

[28] Tomasi, Carlo ; Shi, Jianbo: Direction of Heading from Image Deformations. In: *In Proceedings of IEEE Conference on Computer Vision and Pattern Recognition*, 1993, S. 422–427

[29] Prazdny, K.: Egomotion and Relative Depth Map from Optical Flow. In: *Biological Cybernetics* 36 (1981), S. 87–102

[30] Kanatani, Kenichi: 3-D interpretation of optical flow by renormalization. In: *International Journal of Computer Vision* 11 (1993), Nr. 3, S. 267–282

[31] Recktenwald, Fabian: *Dissertation in Bearbeitung*. Lehrstuhl Kognitive Neurowissenschaften, Universität Tübingen, 2008. – unveröffentlicht

[32] Kalman, Rudolph E.: A New Approach to Linear Filtering and Prediction Problems. In: *Transactions of the ASME–Journal of Basic Engineering* 82 (1960), Nr. D, S. 35–45

[33] Welch, Greg ; Bishop, Gary: An Introduction to the Kalman Filter. 1995. – Forschungsbericht

[34] Maybeck, Peter S.: *Mathematics in Science and Engineering*. Bd. 141: *Stochastic models, estimation, and control - Volume 1*. Academic Press, Inc., 1979

[35] Brown, Robert G. ; Hwang, Patrick Y. C.: *Introduction to Random Signals and Applied Kalman Filterung*. 2. John Wiley and Sons, Inc., 1992

[36] Jacobs, Oliver Louis R.: *Introduction to Control Theory*. 2. Oxford University Press, 1993

[37] Dijkstra, Edsger W.: *Over seinpalen (EWD74)*. http://www.cs.utexas.edu/users/EWD/ewd00xx/EWD74.PDF. Version: zwischen 1962–1964. – private Korrespondenzen

[38] *Kapitel* 7.4. In: Silberschatz, Abraham ; Galvin, Peter B. ; Gagne, Greg: *Operating System Concepts*. 6. John Wiley & Sons, Inc., 2003, S. 201ff

[39] Axolair.de: *Wallpaper2*. http://axolair.de/wallpapers/axolair2_1280x960.jpg. Version: Januar 2009

[40] *Kapitel* 1.4.3. In: Brüning, Gerhard ; Hafer, Xaver: *Flugleistungen*. Berlin, Heidelberg, New York : Springer, 1978, S. 37

[41] Merziger, Gerhard ; Wirth, Thomas: *Repetitorium der höheren Mathematik*. 4. Binomi, 1991

[42] *Kapitel* 4.3.4. In: Richard, Hans A. ; Sander, Manuela: *Technische Mechanik*. Bd. 3: *Dynamik*. Vieweg, 2007, S. 66–68

[43] *Kapitel* 4.1. In: Bittner, Walter: *Flugmechanik der Hubschrauber – Technologie, das flugdynamische System Hubschrauber, Flugstabilitäten, Steuerbarkeit*. 2. Springer, 2005, S. 47ff

[44] Zalzal, Vincent: *The KFilter Project: A Variable Dimension Extended Kalman Filter Library*. http://kalman.sourceforge.net/doc/index.html. Version: Januar 2009

Errata *(Stand: April 2009)*

In der vorliegenden Version dieser Arbeit wurden einige wenige Rechtschreib- und Zeichensetzungsfehler korrigiert. Zudem sind noch folgende Korrekturen anzuführen:

Abschnitt 6.1.4, Seite 74

Der folgende Abschnitt wird gestrichen:

> Da die simulierte Bildfolge jedoch nur einen Flug in Kamerablickrichtung darstellt, kann auch nur für einen solchen Fall die Matrix $\mathbf{R}'$ direkt als Fehler-Kovarianzmatrix $\mathbf{R}$ verwendet werden. Andernfalls käme die gemessene Fehlerellipse falsch zum liegen. Für den allgemeinen Fall muß die Matrix $\mathbf{R}'$ also noch um den jeweiligen Winkel γ gedreht werden, der die horizontale Verdrehung zwischen dem im vorherigen Zeitschritt k–1 berechneten Bewegungsvektor $\vec{\mathbf{v}}_{k-1,k}$ und dem *a priori* Bewegungsvektor $\vec{\mathbf{v}}_k$ beschreibt. Damit ergibt sich für $\mathbf{R}_k$:
>
> $$\mathbf{R}_k = \mathcal{R}_y(\gamma_k) \cdot \mathbf{R}' = \begin{pmatrix} \cos(\gamma_k) & 0 & -\sin(\gamma_k) \\ 0 & 1 & 0 \\ \sin(\gamma_k) & 0 & \cos(\gamma_k) \end{pmatrix} \cdot \mathbf{R}' \quad . \tag{6.57}$$
>
> γ_k wiederum läßt sich unter Verwendung der Definition des Skalarprodukts leicht bestimmen:
>
> $$\gamma_k = \arccos\left(\frac{\vec{\mathbf{v}}_k \quad \cdot \quad \vec{\mathbf{v}}_{k-1,k}}{\|\vec{\mathbf{v}}_k\| \cdot \|\vec{\mathbf{v}}_{k-1,k}\|} \right) \quad . \tag{6.58}$$

Stattdessen wird der folgende Abschnitt eingefügt:

> Da die simulierte Bildfolge jedoch nur einen Flug in Kamerablickrichtung darstellt, kann auch nur für einen solchen Fall die Matrix $\mathbf{R}'$ direkt als Fehler-Kovarianzmatrix $\mathbf{R}$ verwendet werden. Andernfalls käme die gemessene Fehlerellipse falsch zum liegen. Für den allgemeinen Fall muß die Matrix $\mathbf{R}'$ also noch um den jeweiligen Winkel γ gedreht werden,

der die horizontale Verdrehung zwischen der Kamerablickrichtung, also der z-Richtung, und dem *a priori* Bewegungsvektor $\vec{\mathbf{v}}_k$ beschreibt. Damit ergibt sich für $\mathbf{R}_k$:

$$\mathbf{R}_k = \mathcal{R}_y(\gamma_k) \cdot \mathbf{R}' = \begin{pmatrix} \cos(\gamma_k) & 0 & -\sin(\gamma_k) \\ 0 & 1 & 0 \\ \sin(\gamma_k) & 0 & \cos(\gamma_k) \end{pmatrix} \cdot \mathbf{R}' \quad . \tag{6.57}$$

γ_k wiederum läßt sich unter Verwendung der Definition des Skalarprodukts leicht bestimmen:

$$\gamma_k = \arccos\left(\frac{\vec{\mathbf{v}}_k}{\|\vec{\mathbf{v}}_k\|} \cdot \frac{(0,0,1)^\mathsf{T}}{1}\right) = \arccos\left(\frac{(\vec{\mathbf{v}}_k)_z}{\|\vec{\mathbf{v}}_k\|}\right) \quad . \tag{6.58}$$