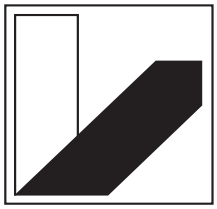

UNIVERSITÄT
BAYREUTH

Algorithmen für die Regelung eines Quadcopters mit Kameraunterstützung

Masterarbeit

von

Matthias Höger

FAKULTÄT FÜR MATHEMATIK, PHYSIK UND INFORMATIK
MATHEMATISCHES INSTITUT

Datum: 30. Juni 2016

Gutachter:
Prof. Dr. L. Grüne
Prof. Dr. K. Chudej

Inhaltsverzeichnis

Symbolverzeichnis

Modellkonstanten

b	Konstante, die vom Auftriebskoeffizienten des Quadcopters abhängt
c	Konstante, die von den Eigenschaften des Motors abhängt
d	Abstand der Rotoren zum Masseschwerpunkt
g	Erdbeschleunigung
I_r	Trägheitsmoment der Rotoren
I_x, I_y, I_z	Trägheitsmomente des Quadcopters
k	Konstante, die vom Strömungswiderstandskoeffizienten der Rotoren abhängt
m	Masse des Quadcopters

Zustände und Kontrolle

$\xi = \begin{pmatrix} x & y & z \end{pmatrix}^T$	Position im Inertialsystem
$v = \begin{pmatrix} v_x & v_y & v_z \end{pmatrix}^T$	Geschwindigkeit im Inertialsystem
$\eta = \begin{pmatrix} \phi & \theta & \psi \end{pmatrix}^T$	Euler-Winkel Roll, Pitch und Yaw
$\nu = \begin{pmatrix} p & q & r \end{pmatrix}^T$	Winkelgeschwindigkeiten im Hauptachsensystem
$\omega = \begin{pmatrix} \omega_1 & \omega_2 & \omega_3 & \omega_4 \end{pmatrix}^T$	Kontrolle interpretiert als Winkelgeschwindigkeiten
$u = \begin{pmatrix} u_1 & u_2 & u_3 & u_4 \end{pmatrix}^T$	Kontrolle interpretiert als Raten

Sonstiges

$\Lambda(A)$	Menge der Eigenwerte der Matrix A
$\text{diag}(x_1, \cdots, x_n)$	$n \times n$ Diagonalmatrix mit den Einträgen $x_1, \cdots, x_n$ auf der Diagonalen
$E(\cdot)$	Erwarungswert

Kapitel 1

Einleitung

1.1 Problemstellung

In dieser Arbeit soll eine kameraunterstützte Positionsregelung eines Quadcopters umgesetzt werden. Dazu muss erst ein mal geklärt werden, wie sich der Systemzustand eines Quadcopters bei dieser Problemstellung zusammensetzt. Um den Zustand zu beschreiben, werden zwei rechtshändige Koordinatensysteme definiert. Zum einen wird ein beliebiges, aber festes Weltkoordinatensystem gewählt, das auch Inertialsystem genannt wird. Zum anderen hat man mit dem sogenannten Hauptachsensystem ein Koordinatensystem, dessen Ursprung im Masseschwerpunkt des Quadcopters liegt und dessen positive x- und y-Achse durch den Mittelpunkt zweier benachbarter Rotoren verläuft. Ist der Quadcopter also in Bewegung, so ändert sich dementsprechend auch die Position und Orientierung des Hauptachsensystems bezüglich des Inertialsystems. In Abbildung 1.1 ist das Inertialsystem blau und das Hauptachsensystem rot dargestellt.

Der 12-dimensionale Systemzustand des Quadcopters wird nun durch die Position $\xi \in \mathbb{R}^3$ und der Geschwindigkeit $v \in \mathbb{R}^3$ seines Masseschwerpunktes im Inertialsystem, den Euler-Winkeln $\eta = (\phi, \theta, \psi)^T \in \mathbb{R}^3$ und den Winkelgeschwindigkeiten $\nu = (p, q, r)^T \in \mathbb{R}^3$ beschrieben. Mit den Euler-Winkeln wird die Orientierung des Hauptachsensystems und damit des Quadcopters bezüglich dem Inertialsystem ausgedrückt. ϕ wird dabei als Roll-Winkel, θ als Pitch-Winkel und ψ als Yaw-Winkel bezeichnet. Ist also ein beliebiger Punkt A im Hauptachsensystem durch die Koordinaten A_x, A_y und A_z gegeben, so können die Koordinaten A_X, A_Y und A_Z von A im Inertialsystem über die Rotationsmatrix R_η, die von den Euler-Winkel definiert wird, bestimmt werden:

$$\begin{pmatrix} A_X \\ A_Y \\ A_Z \end{pmatrix} = R_\eta \cdot \begin{pmatrix} A_x \\ A_y \\ A_z \end{pmatrix} + \xi.$$

Die Winkelgeschwindigkeiten ν geben an, wie schnell sich der Quadcopter um die drei Ach-

sen des Hauptachsensystems dreht.

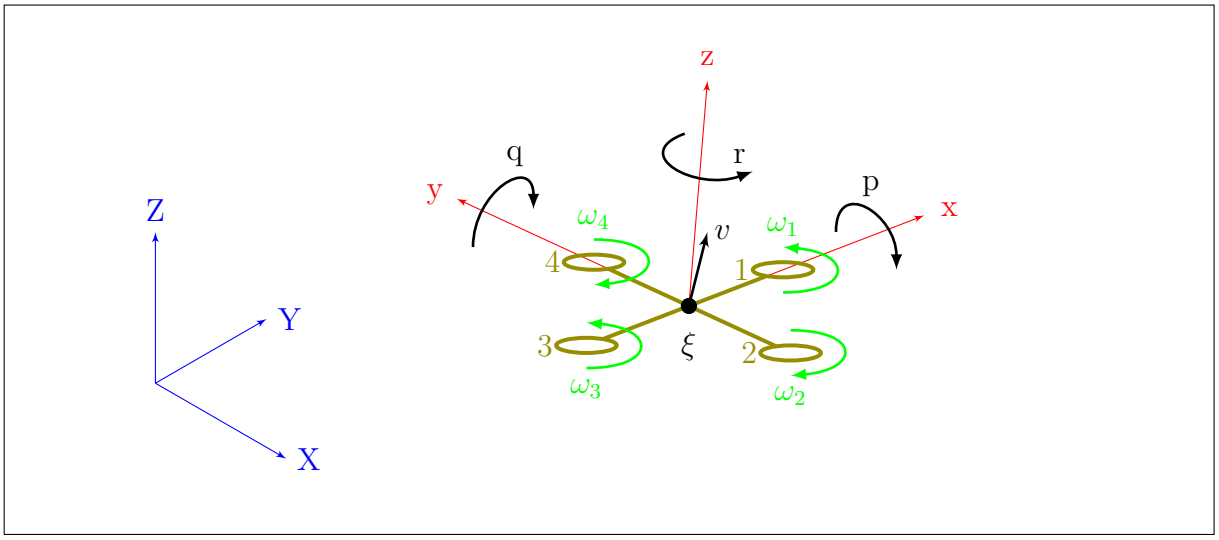

Abbildung 1.1: Der Zustand (schwarz) des Quadcopters (oliv) wird mit Hilfe des Inertialsystems (blau) und dem Hauptachsensystem (rot) beschrieben. Die Euler-Winkel sind hier nicht direkt eingezeichnet, ergeben sich aber aus der Orientierung des Inertial- und des Hauptachsensysems. Rotor i dreht sich mit einer Winkelgeschwindigkeit von $\omega_i \geq 0$ (grün) und erzeugt so eine Schubkraft in Richtung der positiven z-Achse.

Über die Winkelgeschwindigkeiten ω_i der vier Rotoren kann der Systemzustand $x \in \mathbb{R}^{12}$ beeinflusst werden. Mit einer passenden (nichtlinearen) Funktion $f : \mathbb{R}^{12} \times \mathbb{R}^4 \to \mathbb{R}^{12}$ können wir also ein Kontrollsystem angeben, das die Systemdynamik eines Quadcopters modelliert:

$$\frac{d}{dt}x(t) = f(x(t), \omega(t)).$$

Die Frage lautet nun, wie wir die Winkelgeschwindigkeiten $\omega(t)$ zum Zeitpunkt t wählen müssen, damit sich das System um einen gewünschten Zustand stabilisiert. Die Kontrolltheorie liefert zum Beispiel mit dem linear-quadratischen Regler ein Verfahren, das das obige Kontrollsystem zumindest lokal um einen Gleichgewichtspunkt (x^*, ω^*) stabilisiert. Dabei wird eine Feedbackmatrix $K \in \mathbb{R}^{4\times 12}$ berechnet, mit der man ausgehend vom aktuellen Zustand $x(t)$ die Winkelgeschwindigkeiten der Rotoren wie folgt berechnen kann:

$$\omega(t) = K \cdot (x(t) - x^*) + \omega^*.$$

Damit man diese Regelung also anwenden kann, muss der komplette Systemzustand $x(t)$ zum Zeitpunkt t bekannt sein. Der Ausgangspunkt dieser Arbeit ist aber, dass wir lediglich die Euler-Winkel η und die Winkelgeschwindigkeiten ν des Quadcopters messen können. Um

die Position ξ zu bestimmen, soll eine Stereo-Kamera eingesetzt werden. Dabei filmen zwei stationäre Kameras den Quadcopter. In dem entstehenden Bildmaterial muss dann nach dem Flugobjekt gesucht werden. Ist die Suche (oder auch das Tracken) erfolgreich, so kann mit dem Ergebnis der Ort ξ rekonstruiert werden, falls die Position der Kameras im Inertialsystem bekannt ist.

Um die Geschwindigkeit v zu bestimmen, soll das Erweiterte Kalman Filter eingesetzt werden. Das Prinzip des Kalman Filters ist recht intuitiv: Man geht davon aus, dass man den aktuellen Systemzustand $x(t)$ oder zumindest eine gute Schätzung davon kennt. Mit dem Kontrollsystem f weiß man, wie sich der Zustand nominell verhalten wird. Man erhält somit einen Vorhersagewert $x(t^+)$ zum Zeitpunkt $t^+ > t$. Erfolgt nun zu diesem Zeitpunkt eine Messung, so werden diese Messwerte mit den entsprechenden Komponenten von $x(t^+)$ verglichen. Abhängig von diesem Ergebnis wird der Vorhersagewert $x(t^+)$ mehr oder weniger stark korrigiert. Über Gewichtungsmatrizen kann der Anwender dabei angeben, ob eher dem Vorhersagewert oder den Messwerten vertraut werden soll.

1.2 Verwendete Hardware

Bei dem hier eingesetzten Quadcopter handelt es sich um den *Crazyflie Nano Quadcopter 1.0* von *Bitcraze*. Dieser wurde zusätzlich mit einer LED versehen, um das Tracken des Crazyflies zu erleichtern. Von einem Rechner aus kann man über die USB-Antenne von *Bitcraze* mit dem Quadcopter kommunizieren.

Um ein Stereo-Kamera-System zu bilden, werden zwei *HD PRO WEBCAM C920* USB-Kameras von *Logitech* verwendet. Diese Kameras unterstützen mehrere Auflösungen und Bildkodierungen. Für diese Arbeit wurde eine Auflösung von 1280×720 Pixel ausgewählt. Die Kameras liefern dabei 30 Bilder pro Sekunde.

Die in dieser Arbeit angegebenen Laufzeiten von Berechnungen beziehen sich immer auf den verwendeten Testrechner:

- Prozessor: Intel Core i7-4720HQ, 2.60 GHz $\times$ 8
- Speicher: 8 GiB
- Betriebssystem: Ubuntu 15.10, 64 Bit

1.3 Aufbau

Zunächst wird auf die Software des *Crazyflies* und vor allem auf die Unterschiede zur in [14] verwendeten Software eingegangen. Im dritten Kapitel wird zunächst auf das verwendete Ka-

meramodell eingegangen um die Bedeutung der Modellparameter zu verstehen. Anschließend wird bei der Kalibrierung der (Stereo-) Kamera gezeigt, wie man diese Parameter bestimmt. Im letzten Abschnitt wird auf die Verarbeitung von Bildern eingegangen. Kapitel vier führt zunächst die Theorie des Linear-Quadratischen Reglers ein. Anschließend wird gezeigt, wie man mit der Matrix Signumsfunktion das linear-quadratische Problem lösen kann. Im fünften Kapitel wird das Erweiterte Kalman Filter hergeleitet. In Kapitel sechs wird schließlich der linear-quadratische Regler auf den Crazyflie zur Positionregelung angewendet. Abschließen wird die Arbeit im letzten Kapitel mit einer Zusammenfassung sowie einem Ausblick.

Kapitel 2

Crazyflie

2.1 Informationen über den Crazyflie

Wie der Name *Crazyflie Nano Quadcopter* schon andeutet, handelt es sich dabei um einen relativ kleinen Quadcopter. Er wiegt lediglich um die 20 Gramm und besitzt eine Spannweite von etwa 9 Zentimetern. Eine 170 mAh Lithium-Polymer-Batterie soll eine Flugzeit von um die 7 Minuten ermöglichen. Auf dem Crazyflie ist außerdem eine inertiale Messeinheit (engl. inertial measurement unit, kurz: IMU) verbaut. Diese verfügt über einen Beschleunigungssensor und einem gyroskopischen Sensor, mit denen die Beschleunigungen bzw. die Winkelgeschwindigkeiten des Quadcopters im drei-dimensionalen Raum gemessen werden. Diese Daten werden auch verwendet, um mit dem IMU und AHRS (Attitude Heading Reference System) Algorithmus von Madgwick die Euler-Winkel zu schätzen. Für weitergehende Informationen zu diesem Verfahren sei hier auf [22] bzw. dessen Quellen verwiesen.

Von einem Rechner aus kann man über eine USB-Antenne mit dem Crazyflie kommunizieren. So kann zum Beispiel der Anwender über einen USB-Controller den Schub sowie die Euler-Winkel vorgeben und zum Quadcopter senden. Dort findet dann in der ursprünglichen Version eine Lageregelung mittels einem PID-Regler statt. Da es sich bei der Software des Crazyflies um Open-Source handelt, ist dieser Quadcopter vor allem für Entwickler interessant, weil man so zum Beispiel eigene Reglertypen entwerfen und auch in der Praxis testen kann.

Die Hersteller haben mit dem Projekt *Crazyflie* aber noch nicht abgeschlossen und entwickeln sowohl an der Hardware als auch an der Software noch weiter. So gibt es inzwischen auch schon eine zweite Version des Crazyflies. Aktuell arbeiten die Hersteller unter anderem auch an einer Positionsregelung. Die Lokalisierung soll dabei aber nicht wie in dieser Arbeit über eine Stereo-Kamera erfolgen, sondern über sechs im Raum verteilte Transceiver [1]. Der Crazyflie kommuniziert mit diesen Transceiver und kann seinen Abstand zu diesen abfragen [2]. Die Genauigkeit dieser Anfragen soll laut dem Hersteller bis zu 10 Zentimeter betragen

[9]. Ist also die Position der Transceiver bekannt, so kann der Crazyflie durch die ermittelten Abstände seine eigene Position schätzen. Seit Anfang 2016 wurden drei Demonstrations-Videos veröffentlicht [4, 5, 6]. Den letzten beiden Videos kann man entnehmen, dass sie schon deutliche Fortschritte gemacht haben, auch wenn die Positionsregelung noch Instabilitäten aufweist. Wie man der zweiten Demonstration entnehmen kann, vermuten die Entwickler, dass diese unter anderem deswegen entstehen, weil die Kontrollen momentan noch auf dem Rechner berechnet und dann erst zum Crazyflie gesendet werden.

2.2 Software

Die Software des Crazyflies ist in dieser Arbeit in fünf Teile aufgeteilt:

- Crazyradio-Firmware
- C-Bootloader
- C-Client
- Crazyflie-Firmware
- Python-Client

Die Crazyradio-Firmware ist das Programm, das sich auf der USB-Antenne befindet und für dessen richtige Funktionsweise sorgt. Auf dem Crazyflie befindet sich zum einen der C-Bootloader und zum anderen die Crazyflie-Firmware. Die Crazyflie-Firmware ist für den „normalen“ Betrieb des Quadcopters zuständig. Das heißt, dass unter anderem die Daten der inertialen Messeinheit ausgewertet und die aktuellen Kontrollwerte für die Rotoren angewendet werden. Über die USB-Antenne kann man auch eine neue Crazyflie-Firmware auf den Crazyflie aufspielen. Der C-Bootloader empfängt dabei die neue Firmware und speichert sie entsprechend ab. Der Client ist das Programm, das auf dem Rechner läuft und unter anderem als Schnittstelle zum Anwender dient. Hier wurde hauptsächlich der C-Client verwendet, der auf den in C/C++ geschriebene Client von [21] basiert. Da dieser Client aber das Aufspielen einer neuen Crazyflie-Firmware nicht unterstützt, wurde für diesen Zweck auch der offizielle Python-Client von *Bitcraze* verwendet.

Der C-Bootloader und die Crazyradio-Firmware wurden nur der Vollständigkeit halber aufgelistet. Da wir an dieser Software keine Änderungen vornehmen, ist es auch nicht notwendig, dass sie im Source-Code vorliegen. Ist man mit der Crazyflie-Firmware, wie sie sich momentan auf dem Crazyflie befindet, zufrieden, so können weiterführende Projekte auch auf die Crazyflie-Firmware und den Python-Client verzichten. Das heißt, dass in diesem Fall lediglich der C-Client und die dafür benötigte Software vorhanden sein muss (siehe dazu Abschnitt 2.3).

An dieser Stelle wollen wir kurz etwas näher auf die Crazyflie-Firmware und den C-Client eingehen:

Crazyflie-Firmware

In der ursprünglichen Version von Bitcraze findet eine Lageregelung mit einem PID-Regler statt. Der Anwender gibt zum Beispiel über einen Controller die Euler-Winkel und den Schub vor und schickt sie über den Client zum Crazyflie. Der PID-Regler berechnet dort auf Grund der aktuellen Daten der inertialen Messeinheit die entsprechenden Kontrollwerte. In einer vorangegangenen Seminararbeit zur Lageregelung des Crazyflies [15] wurde die Firmware um einen LQ-Regler erweitert. Da sich auf dem Crazyflie aber keine Software zum Berechnen der LQ-Feedbackmatrix befindet, muss die entsprechende Matrix auf einem Rechner bestimmt und dann zum Quadcopter gesendet werden. Beim Verbindungsaufbau mit dem Crazyflie wurde dann schließlich festgelegt, welcher Regler verwendet werden sollte. Ein „online-Wechsel“ des Reglertypes war dabei aber nicht möglich.

Diese Firmware wurde in dieser Arbeit noch einmal erweitert. Nun ist es möglich gleich die Kontrollwerte für die Rotoren zum Crazyflie zu senden. Die Berechnung der Kontrollwerte kann also komplett auf den Client verschoben werden. Dies birgt den Vorteil, dass nun theoretisch ein beliebiger Reglertyp eingesetzt werden kann und man nicht darauf angewiesen ist, dass die dafür notwendige Software auch auf dem Crazyflie läuft. „Theoretisch“ deswegen, weil natürlich eine gewisse Rate eingehalten werden muss, mit der die Kontrollwerte aktualisiert werden. Ist ein Reglertyp zu langsam, so wird sich der Crazyflie damit auch nicht stabilisieren lassen.

Der Nachteil bei dieser Vorgehensweise ist, dass die gemessenen Euler-Winkel und Gyrodaten nun zum Client gesendet werden müssen. In der [14] beschriebenen Methode zum Loggen von Daten (mit dem Python-Client) bzw. in der entsprechenden Version des C-Client konnten die Daten nicht zuverlässig in einer hohen Rate geloggt werden. Dabei wurde im Client festgelegt, welche Daten in welcher Rate geloggt werden sollen. Auf dem Crazyflie wurden die Daten dann in den entsprechenden Raten in einer FIFO-Liste abgespeichert, die dann vom Echtzeitbetriebssystem abgearbeitet werden sollte. Es gab immer wieder Phasen von bis zu 20 Millisekunden, in der keine neuen Daten beim Client eingetroffen sind. Warum dies so ist, konnte nicht festgestellt werden. Eventuell liegt es am Echtzeitbetriebssystem des Crazyflies bzw. an dessen Scheduler. Vielleicht liegen phasenweise immer relativ viele Aufgaben mit hoher Priorität vor, so dass die Log-Daten nicht losgeschickt werden. Eine erhöhte Priorität für die Log-Daten hat aber auch keine Abhilfe geschaffen.

Daher wurde dann auf das bisherige Logging-System verzichtet und ein eigenes verwendet. Nun wird für jedes Kontrollpaket das beim Crazyflie ankommt, sofort ein Logging-Paket mit den Euler-Winkeln und den Gyro-Daten zurückgeschickt. Auf diese Weise ist es nun möglich, die Daten in etwa alle zwei bis drei Millisekunden zu loggen. So hat man zwar auch Einbußen beim Komfort, da die bisherigen Logging-Routinen nicht mehr verwendet werden sollten

und man damit nicht mehr vom Client aus entscheiden kann, welche Daten geloggt werden sollen, aber weil wir sowieso nur an den Euler-Winkel und Gyro-Daten interessiert sind, ist dies für uns vertretbar.

Bei der jetzt vorliegenden Firmware ist es außerdem möglich, während dem Flug zwischen verschiedenen Reglertypen zu wechseln. Dazu wurden die Kontrollpakete, die der Client versendet, mit einer ID versehen. Empfängt nun der Crazyflie ein Kontrollpaket, so wird zuerst diese ID ausgelesen. Anhand dieser ID weiß der Crazyflie, um was für ein Paket es sich handelt und wie dieses aufgebaut ist. Gegebenfalls wird dabei auch der eingesetzte Reglertyp gewechselt.

C-Client

Der C-Client gliedert sich zunächst in fünf Teile auf:

- Aruco
- CClient
- Controller
- Modules
- Vision

Aruco ist ein Zusatzpaket der *OpenCV*-Bibliothek. Der Kern dieses Programmes ist der *CClient*. Dieser Ordner enthält den (angepassten) Client von [21] und kommuniziert mit dem Crzayflie. Hier werden auch die entworfenen LQ-Regler ausgeführt. *Controller* enthält eine Joystick-Klasse, mit der Signale von einem USB-Controller ausgelesen und verarbeitet werden können. Je nach Typ des verwendeten Controllers müssen die Tastenbelegungen aber eventuell neu angepasst werden. Die Klasse basiert dabei auf [18]. *Vision* enthält die Klassen, die für die Verwendung der Stereo-Kamera notwendig sind. *Modules* beinhaltet die Klassen für die mathematischen Berechnungen, wie zum Beispiel das mathematische Modell eines Quadcopters und das Erweiterte Kalman Filter.

2.3 Vorarbeiten der Entwicklungsumgebung

In dieser Arbeit kommen noch einige Zusatzprogramme zum Einsatz. In diesem Abschnitt wird eine kurze Anleitung angegeben, wie man diese Software unter Ubuntu 15.10 installiert und was dabei zu beachten ist.

Für die Kalibrierung der Kameras (vgl. Abschnitt 3.1) wurde eine graphische Benutzeroberfläche implementiert. Dabei kommt die Bibliothek *Gtk 3.0* zum Einsatz. Das Programm

v4l-utils wird benutzt um den Autofokus der vorhandenen Kameras zu deaktivieren. Aus der *Eigen 3.0* Bibliothek werden die Matrixoperationen und Löser von linearen Gleichungssystemen verwendet. Um auf die USB-Geräte zuzugreifen, wird *libusb 1.0* verwendet. All diese Programme sind in den offiziellen Paketquellen enthalten und können somit über den Softwaremanager installiert werden:

```
$ sudo apt-get install v4l-utils
$ sudo apt-get install libusb-1.0-0-dev
$ sudo apt-get install libeigen3-dev
$ sudo apt-get install libgtkmm-3.0-dev
```

Um auf die Kameras zuzugreifen und das entstehende Bildmaterial weiter zu verarbeiten, wird *OpenCV 3.1* verwendet. Für die Installation kann man der Anleitung in [23] folgen. Bei der cmake-Konfiguration müssen allerdings noch die folgenden Optionen angefügt werden: -WITH_GTK=ON und -WITH_GTK3=ON. Ansonsten wird zum Anzeigen von Bildern *Gtk 2* verwendet. Diese Version ist aber mit *Gtk 3*, das für unsere graphische Oberfläche eingestzt wird, nicht kompatibel. Wird während der Laufzeit festgestellt, dass beide Versionen verwendet werden, so bricht das Programm ab.

Außerdem muss die Option -DWITH_IPP=ON angefügt werden, damit die Datei *libippicv.a* erstellt wird. Nach dem Bauen muss die Datei aber gegebenenfalls trotzdem noch in den richtigen Ordner kopiert werden:

```
$ sudo cp /usr/local/share/OpenCV/3rdparty/lib/libippicv.a \
> /usr/local/lib
```

Will man auch die Crazyflie-Firmware ändern und neu auf den Crazyflie aufspielen, so müssen weitere zusätzliche Pakete installiert werden. Dabei kann man der Anleitung in [14] folgen.

Kapitel 3

Bildverarbeitung

3.1 OpenCV

Zur Positionsbestimmung eines Objekts wird in dieser Arbeit ein Stereo-Kamera-System verwendet. Dabei werden zwei Kameras so im Raum aufgestellt, dass sich deren Sichtfeld zum Teil überdecken. Dieses Sichtfeld wird Arbeitsbereich genannt. Eine einzelne Kamera besitzt zwar keine Tiefeninformation, aber mit Hilfe einer Stereo-Kamera lässt sich diese erzeugen und damit auch eine 3D-Rekonstruktion bewerkstelligen. Für eine erfolgreiche 3D-Rekonstruktion müssen aber die Kameras kalibriert werden. Bei der Kalibrierung werden die sogenannten intrinsische und extrinsische Parameter bestimmt. Die intrinsischen Parameter hängen ausschließlich von der Kamera selbst ab. Zu ihnen zählt zum Beispiel die Brennweite. Die extrinsischen Parameter geben die Position und Orientierung der Kamera im Raum an. Die Bestimmung dieser Parameter wird Kalibrierung bzw. Stereokalibrierung genannt.
In dieser Arbeit wird die *OpenCV*-Bibliothek mit dem Zusatzpaket *Aruco* verwendet um die Bildinformationen der Kameras auszulesen und weiter zu verarbeiten.

3.1.1 Kameramodell

Die Kameras werden in *OpenCV* als Lochkamera modelliert. Die Kamera besteht dabei aus einer Bildebene und einer dazu parallelen Lochebene, in der sich ein infinitesimal kleines Loch befindet, das wir Zentralprojektionspunkt nennen. Das Lot auf die Lochebene durch den Zentralprojektionspunkt wird optische Achse genannt. Der Schnittpunkt der optischen Achse mit der Bildebene wird Hauptpunkt genannt.

Wird mit der Kamera eine Szene aufgenommen, so dringt von jedem beliebigen Punkt der Szene nur genau ein Lichtstrahl durch das Loch und trifft auf die Bildebene. So entsteht ein umgedrehtes Abbild der aufgenommenen Szene auf der Bildebene. Abbildung 3.1 verdeutlicht diesen Vorgang. Die Größe des Abbildes hängt vom Abstand f der Bildebene zur Lochebene ab und wird Brennweite genannt. Wird ein Raumpunkt P aufgenommen, der den Abstand

$Z > 0$ zur Lochebene besitzt, so lässt sich der auf die Bildebene projezierte Punkt p mit Hilfe des Strahlensatzes berechnen:

$$\overrightarrow{cp} = -f\frac{\overrightarrow{P_0P}}{Z}$$
$$p = c + \overrightarrow{cp},$$

wobei c der Hauptpunkt der Kamera und P_0 die orthogonale Projektion von P auf die optische Achse ist.

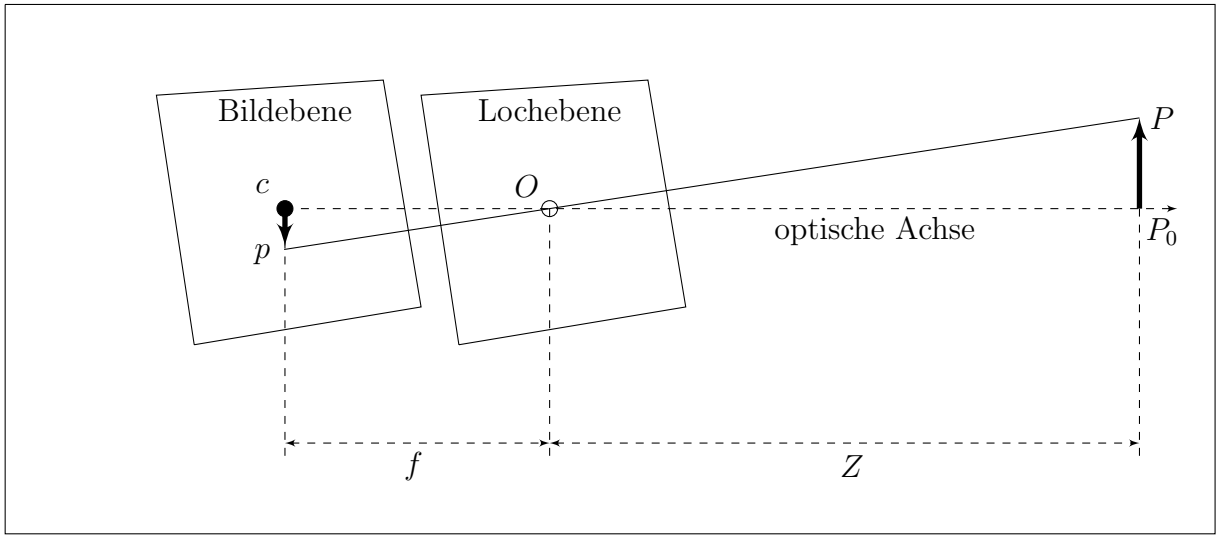

Abbildung 3.1: Lochkamera

Damit die Abbildungen auf der Bildebene nicht umgedreht dargestellt werden, wird im Folgenden ein angepasstes Modell betrachtet. Dazu wird die Bildebene um den Vektor $2 \cdot \overrightarrow{cO}$ verschoben, wobei O der Zentralprojektionspunkt der Kamera ist. Damit befindet sich die Bildebene nun im Abstand f „vor“ der Lochebene[1]. In diesem Fall gilt für das obige Szenario:

$$\overrightarrow{cp} = f\frac{\overrightarrow{P_0P}}{Z}$$
$$p = c + \overrightarrow{cp} \tag{3.1}$$

In der Realität ist die Bildebene natürlich begrenzt. Wir nehmen an, dass diese begrenzte Ebene rechteckig ist und nennen sie Bildfläche. Diese Bildfläche ist in jeweils gleich große, rechteckige Pixel, aufgeteilt[2]. Die Breite eines Pixel sei durch l_x^{-1}, die Höhe mit l_y^{-1} gegeben.

[1] In Wirklichkeit lässt sich solch eine Kamera natürlich nicht realisieren.
[2] Vor allem bei preisgünstigen Kameras kommt es vor, dass die Pixel nicht quadratisch sind [7, S. 373].

$l_x > 0$ und $l_y > 0$ sind also Größen mit der Einheit „Pixel pro Längenheinheit“. Für diese Bildfläche legen wir ein (kontinuierliches) $\bar{x}$-$\bar{y}$-Pixelkoordinatensystem (PKS) fest, dessen Ursprung aus Sicht des Zentralprojektionspunktes O in der linken, oberen Ecke der Bildfläche liegt. Die $\bar{x}$-Achse zeigt aus Sicht des Zentralprojektionspunktes nach rechts, während die $\bar{y}$-Achse nach unten zeigt. Das x-y-z-Kamerakoordinatensystem (KKS) wird so definiert, dass der Ursprung im Zentralprojektionspunkt O liegt, die z-Achse der optischen Achse entspricht und die x- bzw. y-Achse in die gleiche Richtung zeigen wie die entsprechenden Achsen des Pixelkoordinatensystems. Dies ist in Abbildung 3.2 veranschaulicht.

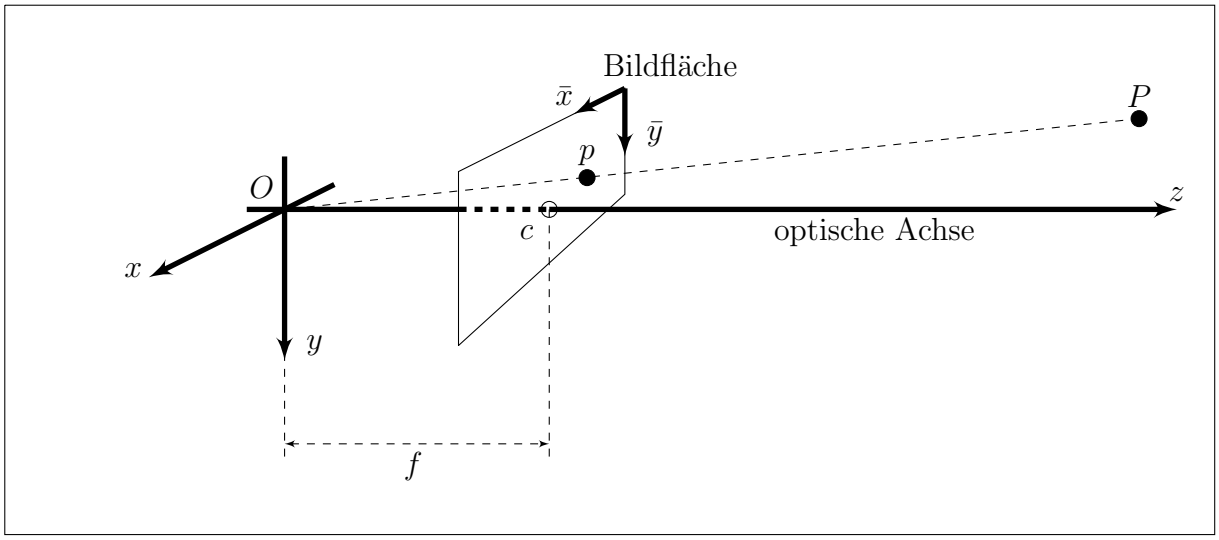

Abbildung 3.2: neues Modell

Ist nun ein Punkt $P = (P_x, P_y, P_z)^T$ mit $P_z \neq 0$ im KKS gegeben[3], der auf den Punkt $p = (p_x, p_y, f)^T$ gegeben im KKS projeziert wird, so gilt nach Gleichung (3.1):

$$p_x = f\frac{P_x}{P_z}, \qquad p_y = f\frac{P_y}{P_z}$$

Ausgedrückt in Pixelkoordinaten gilt damit für $p = (p_{\bar{x}}, p_{\bar{y}})^T$ mit dem Hauptpunkt $c =$

[3]Wir können für unsere Zwecke ohne Einschränkung annehmen, dass $P_z \neq 0$ gilt. Wäre $P_z = 0$ so könnte die Kamera den Punkt nur wahrnemen, wenn P gleich dem Zentralprojektionspunkt wäre. Für den Zentralprojektionspunkt als betrachteten Raumpunkt interessieren wir uns aber nicht und schließen diesen Fall aus. Somit gilt im Folgenden immer $P_z \neq 0$.

$(c_{\bar{x}}, c_{\bar{y}})^T$ im PKS:

$$\begin{aligned} p_{\bar{x}} &= p_x l_x + c_{\bar{x}} \\ &= f l_x \frac{P_x}{P_z} + c_{\bar{x}} \\ &=: f_x \frac{P_x}{P_z} + c_{\bar{x}} \qquad (3.2) \\ p_{\bar{y}} &= p_y l_y + c_{\bar{y}} \\ &= f l_y \frac{P_y}{P_z} + c_{\bar{y}} \\ &=: f_y \frac{P_y}{P_z} + c_{\bar{y}}. \qquad (3.3) \end{aligned}$$

f_x und f_y kann man als neue Brennweiten der Kamera interpretieren.

Zusätzlich zum PKS und KKS führen wir noch ein drei-dimensionales, karthesisches X-Y-Z-Weltkoordinatensystem ein (WKS). Die Rotationsmatrix $R \in \mathbb{R}^{3\times3}$ und der Translationsvektor $t \in \mathbb{R}^3$ bilden das WKS auf das KKS ab. Ein beliebiger Punkt $P = (P_X, P_Y, P_Z)^T$ im WKS wird also im KKS durch $P = (P_x, P_y, P_z)^T$ mit

$$\begin{pmatrix} P_x \\ P_y \\ P_z \end{pmatrix} = R \begin{pmatrix} P_X \\ P_Y \\ P_Z \end{pmatrix} + t \qquad (3.4)$$

dargestellt.

Die Matrix $\begin{pmatrix} f_x & 0 & c_{\bar{x}} \\ 0 & f_y & c_{\bar{y}} \\ 0 & 0 & 1 \end{pmatrix}$ wird in *OpenCV* Kameramatrix genannt [17, S. 393]. Diese Matrix ergibt sich, wenn man die Gleichungen (3.2) bis (3.4) kombiniert und in homogene Koordinaten übergeht. In diesem Fall lässt sich die Abbildung von Punkten im WKS auf die Bildfläche auch wie folgt beschreiben[4]:

$$s \begin{pmatrix} p_{\bar{x}} \\ p_{\bar{y}} \\ 1 \end{pmatrix} = \begin{pmatrix} f_x & 0 & c_{\bar{x}} \\ 0 & f_y & c_{\bar{y}} \\ 0 & 0 & 1 \end{pmatrix} (R|t) \begin{pmatrix} P_X \\ P_Y \\ P_Z \\ 1 \end{pmatrix} \qquad (3.5)$$

Üblicherweise kommen heutzutage aber keine Lochkameras, sondern Linsenkameras zum Einsatz. Die Linse verzerrt jedoch das projezierte Bild. Über zusätzliche Parameter wird in

[4]In dieser Darstellung sind auch Punkte zulässig, die im KKS in z-Richtung den Wert 0 besitzen.

OpenCV diese Verzerrung modelliert. Ein Punkt $P = (P_X, P_Y, P_Z)^T$ im WKS wird demnach wie folgt auf die Bildfäche abgebildet[5]:

$$\begin{pmatrix} P_x \\ P_y \\ P_z \end{pmatrix} = R \begin{pmatrix} P_X \\ P_Y \\ P_Z \end{pmatrix} + t$$

$$x' = P_x/P_z$$
$$y' = P_y/P_z$$
$$r^2 = x'^2 + y'^2$$
$$x'' = x' \frac{1 + k_1 r^2 + k_2 r^4 + k_3 r^6}{1 + k_4 r^2 + k_5 r^4 + k_6 r^6} + 2p_1 x' y' + p_2(r^2 + 2x'^2) + s_1 r^2 + s_2 r^4$$
$$y'' = y' \frac{1 + k_1 r^2 + k_2 r^4 + k_3 r^6}{1 + k_4 r^2 + k_5 r^4 + k_6 r^6} + p_1(r^2 + 2y'^2) + 2p_2 x' y' + s_3 r^2 + s_4 r^4$$
$$p_{\bar{x}} = f_x x'' + c_{\bar{x}}$$
$$p_{\bar{y}} = f_y y'' + c_{\bar{y}}$$

$k_1, \cdots, k_6$ entsprechen dabei radiale Verzerrungskoeffizienten, p_1 und p_2 tangentiale Verzerrungskoeffizienten und $s_1, \cdots, s_4$ Verzerrungskoeffizienten einer dünnen Linse. Dabei ist zu beachten, dass das Modell laut der *OpenCV*-Dokumentation in der Gleichung für y'' s_1 und s_2 verwendet anstatt s_3 bzw. s_4 [17, 394]. Ein Blick in den Source Code der *OpenCV*-Bibliothek zeigt aber, dass das oben angegebene Modell implementiert ist. Dem Code kann man auch entnehmen, dass P_z auf 1 gesetzt wird, falls P_z eigentlich den Wert 0 hat [16, Z. 857-868 und 764].

Für die Ursache der Verzerrungen sei auf [7, S. 375ff.] und der darin angegebenen Literatur verwiesen.

3.1.2 Kalibrierung

Die Bestimmung der in Abschnitt 3.1.1 erwähnten Verzerrungskoeffizienten wird Kalibrierung genannt. Die Kalibrierung kann zum Beispiel mit Hilfe eines Schachbrettmusters durchgeführt werden. Dabei geht man folgendermaßen vor:

1. Nehme mit der zu kalibrierenden Kamera ein Bild des Schachbrettmusters auf.

2. Bestimme die Koordinaten aller Eckpunkte des Schachbrettmusters in einem beliebigen drei-dimensionalen, karthesischen Koordinatensystems. In der Regel wird der Ursprung dieses Koordinatensystems in eine der vier Randeckpunkten gelegt. Außerdem wird das System so orientiert, dass das Schachbrettmuster im ersten Quadranten liegt. In diesem Fall lassen sich die Eckpunkte e_{ij} leicht bestimmen: $e_{ij} = (il, jl, 0)^T$, wobei l

[5] Wir gehen hier wieder davon aus, dass $P_z \neq 0$ gilt.

der Seitenlänge der einzelnen Quadrate des Schachbrettmusters entspricht[6]. In *OpenCV* werden diese Punkte *objectPoints* genannt.

3. Bestimme in dem Bild die entsprechenden zwei-dimensionalen Pixelkoordinaten der Eckpunkte. In *OpenCV* werden diese Punkte *imagePoints* genannt und können mit der Methode *cv::findChessboardCorners* berechnet werden.

4. Wiederhole Schritt 1 bis 3, so dass das Schachbrettmuster aus verschiedenen Positionen und Neigungswinkeln aufgenommen wurde. Bei einem 7×8 Schachbrettmuster sollten es mindestens zehn solcher Aufnahmen sein um ausreichend Informationen für die Kalibrierung zu bekommen [7, S. 388].

Bemerkung: Wir sprechen von einem Eckpunkt, wenn sich in diesem Punkt vier Quadrate berühren. Außerdem bezieht sich die Angabe von $n \times m$ bei einem $n \times m$ Schachbrettmuster auf diese Eckpunkte. Das Schachbrett besteht in diesem Fall also aus $(n+1) \times (m+1)$ Quadraten.

Die Methode *cv::calibrateCamera* nutzt nun den Levenberg-Marquardt-Algorithmus um die Parameter zu bestimmen. Dazu werden die *obejectPoints* mit Hilfe der aktuellen Parameterwerte auf die Bildebene der Kamera projeziert. Als Kostenfunktion wird die Summe der quadrierten Abstände dieser Punkte zu den entsprechenden *imagePoints* gewählt [17, S. 397]. In [17, S. 397] wird lediglich erwähnt, dass es sich beim Rückgabewert der Methode um den Reprojektionsfehler handelt. Ein Blick in den Source-Code zeigt, dass es sich bei dem Reprojektionsfehler um das Quadratische Mittel der (nicht quadrierten) Abstände handelt [16, Z. 1547-1580]. Der Wert ist also ein Indikator für die Güte der Kalibrierung. Ist der Reprojektionsfehler kleiner als 1, so liegen die projezierten Punkte im (quadratischen) Mittel um weniger als einen Pixel von den eigentlichen *imagePoints* entfernt und spricht dafür, dass die gefundenen Parameter eine gute Schätzung für die tatsächlichen Parameter sind.

Hinweise:

- Bei der Herleitung des Kameramodells haben wir gesehen, dass f_x und f_y von der Brennweite f der Kamera abhängen (vgl. Gleichungen (3.2) und (3.3)). Verwendet man also eine Kamera mit integriertem Auto-Fokus, so sollte man diesen deaktivieren[7].

- Da $c_{\bar{x}}$ und $c_{\bar{y}}$ in Pixelkoordinaten angegeben sind, hängen diese von der ausgewählten Auflösung des Bildes ab, wie man sich an Hand der Abbildung 3.2 überlegen kann. Auch l_x und l_y sind von der Auflösung abhängig und damit gilt das gleiche für die Brennweiten f_x und f_y. Es reicht aber aus, diese Koeffizienten nur einmal zu bestimmen. Ändert man die Auflösung um einen bestimmten Faktor in x- bzw. y-Richtung,

[6] Die Seitenlänge sollte man dabei in der Längeneinheit angeben, in der man später die Ergebnisse der 3D-Rekonstruktion haben möchte. In unserem Fall wird es also Meter sein.

[7] In der Implementierung zu dieser Arbeit wird dafür das Programm *v4l2* verwendet.

so passt man f_x und $c_{\bar{x}}$ bzw. f_y und $c_{\bar{y}}$ mit dem gleichen Faktor an, da eine direkte Proportionalität zwischen diesen Größen besteht.

- Die Verzerrungskoeffizienten sind unabhängig von der gewählten Auflösung [17, S. 394].
- Die Parameter der Kameramatrix und die Verzerrungskoeffizienten hängen nur von der Kamera ab. Daher werden sie intrinsische Parameter genannt.
- *cv::calibrateCamera* schätzt nicht nur die intrinsischen Parameter, sondern bestimmt für jedes Schachbrettmuster aus Schritt 2 auch dessen Lage und Orientierung im KKS. Dies wird über einen Rotationsvektor[8] und einem Translationsvektor beschrieben, die unabhängig von den kameraspezifischen Eigenschaften sind. Daher gehören beide Vektoren zu den extrinsischen Parametern.

Hat man die Verzerrungskoeffizienten bestimmt, so kann man in *OpenCV* mit den Methoden *cv::remap* oder *cv::undistort* verzerrte Bilder wieder entzerren. Alternativ kann man auch die Methode *cv::undistortPoints* verwenden um nur einzelne Pixel zu korrigieren. Wir werden diese Funktion nutzen, da wir nur an der Korrektur einzelner Pixel interessiert sind (die Pixel, die die Position des Crazyflies wiedergeben). Würden wir das komplette Bild entzerren, so würde das nur unnötig die Laufzeit erhöhen.

Ziel der Kalibrierung einer Stereo-Kamera ist es, die Positionierung der beiden Kameras zueinander herauszufinden. Es wird also versucht eine Rotationsmatrix R und einen Translationsvektor t zu bestimmen, mit deren Hilfe man das KKS der einen Kamera auf das KKS der anderen Kamera abbilden kann. R und t sind unabhängig von den kameraspezifischen Eigenschaften und gehören damit zu den extrinsischen Parametern einer Stereo-Kamera.

Bei der Stereokalibrierung geht man im Prinzip genauso vor wie bei der Kalibrierung einer einzelnen Kamera:

1. Nehme mit beiden Kameras der Stereo-Kamera gleichzeitig ein Bild des Schachbrettmusters auf.
2. Bestimme die Koordinaten aller Eckpunkte des Schachbrettmusters in einem beliebigen drei-dimensionalen, karthesischen Koordinatensystems. In der Regel wird der Ursprung dieses Koordinatensystems in eine der vier Randeckpunkten gelegt. Außerdem wird das System so orientiert, dass das Schachbrettmuster im ersten Quadranten liegt. In diesem Fall lassen sich die Eckpunkte e_{ij} leicht bestimmen: $e_{ij} = (il, jl, 0)^T$, wobei l der Seitenlänge der einzelnen Quadrate des Schachbrettmusters entspricht[9]. In *OpenCV* werden diese Punkte *objectPoints* genannt.

[8] Im $\mathbb{R}^3$ lässt sich mit Hilfe der Rodrigues Transformation eine 3×3 Rotationsmatrix durch einen drei-dimensionalen Rotationsvektor darstellen [7, S. 401f.].

[9] Die Seitenlänge sollte man dabei in der Längeneinheit angeben, in der man später die Ergebnisse der 3D-Rekonstruktion haben möchte. In unserem Fall wird es also Meter sein.

3. Bestimme in beiden Bildern die entsprechenden zwei-dimensionalen Pixelkoordinaten der Eckpunkte. In *OpenCV* werden diese Punkte für die erste Kamera *imagePoints1* und für die zweite Kamera *imagePoints2* genannt und können mit der Methode *cv::findChessboardCorners* berechnet werden.

4. Wiederhole Schritt 1 bis 3, so dass das Schachbrettmuster aus verschiedenen Positionen und Neigungswinkeln aufgenommen wurde.

Die Methode *cv::stereoCalibrate* nutzt nun den Levenberg-Marquardt-Algorithmus um die Rotationsmatrix R und den Translationvektor t zu bestimmen. Dabei wird versucht den Reprojektionsfehler für beide Kameras zu minimieren. Wie bei der Kalibrierung einer einzelnen Kamera wird für jedes Schachbrettmuster dessen Lage und Orientierung im jeweiligen KKS bestimmt. Für das i-te Schachbrettmuster sei diese Beziehung durch die Rotationsmatrix R_1^i und dem Translationsvektor t_1^i für die erste Kamera gegeben und durch R_2^i sowie t_2^i für die zweite Kamera. Daraus lässt sich natürlich bereits R und t bestimmen. [7, S. 428].

Genauso wie bei der Kalibrierung einer einzelnen Kamera wird bei der Stereo-Kalibrierung der Reprojektionsfehler zurückgegeben. Dem Source-Code kann man entnehmen, wie sich dieser berechnet [16, Zeilen 1939-2116]:

Zunächst werden die *objectPoints* in das KKS der ersten Kamera abgebildet. Anschließend werden diese Punkte auf die Bildfälche der ersten Kamera projeziert und der Abstand zu den gemessenen *imagePoints1* berechnet. Für die zweite Kamera werden die *objectPoints* auch in das KKS der ersten Kamera abgebildet. Diese Punkte werden dann über die gefundenen Parameter R und t in das KKS der zweiten Kamera überführt. Nach der Projektion auf die Bildfläche der zweiten Kamera wird der Abstannd zu den gemessenen *imagePoitns2* berechnet. Der Reprojektionsfehler wird als das quadratischen Mittel dieser Abstände definiert.

Hinweise:

- In Schritt 1 wird erwähnt, dass die Bilder gleichzeitig aufgenommen werden sollten. Das Bild der ersten Kamera sollte also möglichst zeitgleich zum Bild der zweiten Kamera enstehen. Im Optimalfall ist kein Zeitversatz vorhanden. Um den Zeitversatz zu reduzieren, ruft man in *OpenCV* für jede Kamera zunächst nur die Rohdaten mit der Methode *cv::grab* ab. Erst anschließend werden diese Daten mit der zeitintensiveren Methode *cv::retrieve* decodiert.

- An dieser Stelle sei schon darauf hingewiesen, dass die beiden Kameras so aufgestellt werden sollten, dass ihre Bildebenen möglichst parallel und nur horizontal aber nicht vertikal versetzt sind. Auf den Grund dafür werden wir jetzt im nächsten Abschnitt etwas näher eingehen.

3.1.3 3D-Rekonstruktion

In diesem Abschnitt zeigen wir, wie in *OpenCV* die drei-dimensionalen Weltkoordinaten eines realen Punktes mit Hilfe der Bilder einer Stereo-Kamera rekonstruiert werden können. Das verwendete Verfahren setzt dabei voraus, dass die Stereo-Kamera frontal-parallel ist, das heißt:

- Beide Kameras besitzen die gleiche Brennweite f: $f^l = f^r = f$.
- Der Hauptpunkt c^l der linken Kamera und der Hauptpunkt c^r der rechten Kamera gegeben im PKS der linken bzw. rechten Kamera besitzen die gleichen Koordinaten: $c^l = c^r$.
- Die Bildflächen beider Kameras sind koplanar, gleich orientiert und befinden sich auf gleicher Höhe, das heißt: Wird ein realer Punkt P auf den Bildpunkt p^l in der linken Bildfläche abgebildet und auf den Bildpunkt p^r in der rechten Bildfläche, so gilt: $p^l_{\bar{y}} = p^r_{\bar{y}}$. Wird ein zweiter Punkt Q auf die linke Bildfläche mit den Pixelkoordinaten q^l abgebildet so dass $q^l_{\bar{x}} > p^l_{\bar{x}}$ gilt, so muss für dessen Abbildung auf den Punkt q^r in der rechte Bildfläche die Folgerung $q^r_{\bar{x}} > p^r_{\bar{x}}$ erfüllt sein.
  In diesem Fall nennen wir die Bildebenen zeilenweise ausgerichtet.

Abbildung 3.3 zeigt eine frontal-parallele Stereo-Kamera.

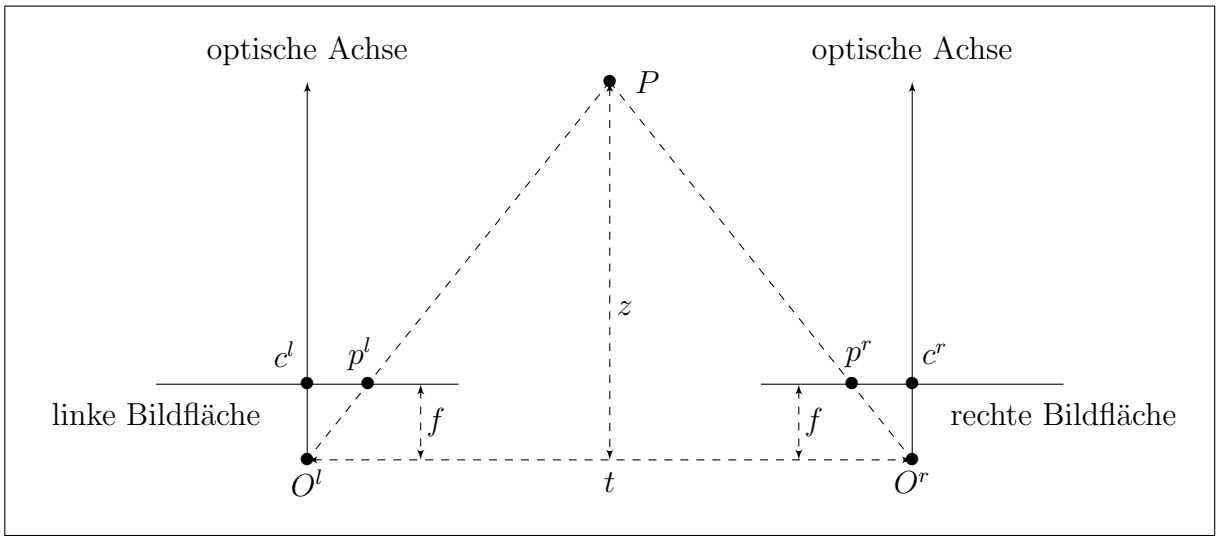

Abbildung 3.3: Idealisierte Stereo-Kamera

In der Realität wird eine Stereo-Kamera diese Anforderungen aber nicht erfüllen. Daher muss eine frontal-parallele Stereo-Kamera simuliert werden, indem Projektions- und Verzerrungsabbildungen auf die linke und rechte Bildebene angewendet werden. Diese Abbildungen

werden mit Hilfe mathematischer Verfahren bestimmt. Damit diese Verfahren gute Ergebnisse liefern, sollten die beiden Kameras bereits möglichst frontal-parallel aufgestellt werden. Wenn dem nicht so ist, können die gefundenen Verzerrungsabbildungen extrem starke Verzerrungen hervorrufen, so dass die beiden Kameras der simulierten Stereo-Kamera kaum noch oder überhaupt keinen gemeinsamen Sichtbereich mehr haben. [7, S. 418]

Mit der Rotationsmatrix R und dem Translationsvektor t aus der Stereokalibrierung haben wir bereits eine Abbildung um die linke und rechte Bildebene in koplanare Bildebenen überzuführen. Um diese Bildebenen nun auch zeilenweise auszurichten, wird mit der *OpenCV* Methode *stereoRectify* eine Rektifizierung durchgeführt. Dabei werden mit den intrinsischen Parametern beider Kameras und den extrinsischen Parametern R und t unter anderem für beide Kameras die Rotationsmatrizen $R1$ bzw. $R2 \in \mathbb{R}^{3\times3}$ und die Projektionsmatrizen $P1$ bzw. $P2 \in \mathbb{R}^{3\times4}$ ermittelt. In [7, S. 419-435] wird näher darauf eingegangen, wie *OpenCV* diese Matrizen berechnet.

Für unsere Zwecke reicht es zu wissen, dass $R1$ und $R2$ die Bildflächen so rotiert, dass sie zeilenweise ausgerichtet sind. Mit den Matrizen $P1$ und $P2$ können reale drei-dimensionale Punkte gegeben in homogenen Koordinaten in das rektifizierte Koordinatensystem der ersten bzw. zweiten Kamera abgebildet werden. Das Ergebnis dieser Abbildung sind die entsprechenden Pixelkoordinaten in homogener Darstellung.

In der Theorie könnte man nun einen Punkt P mit der idealisierten Stereo-Kamera rekonstruieren, indem man den Schnittpunkt der Geraden $O^l p^l$ und $O^r p^r$ bestimmt. Aber durch Messungenauigkeiten und alleine schon weil die Bildpunkte p^l und p^r von P nur in diskreten Pixelkoordinaten vorliegen, werden diese Geraden in der Regel windschief sein. Daher kann der Punkt P nur geschätzt werden. In *OpenCV* kann man dazu die Methode *cv::triangulatePoints* einsetzen, die die Projektionsmatrizen $P1$ und $P2$ sowie die Bildpunkte von P gegeben durch p^l und p^r im PKS der linken bzw. rechten Kamera verwendet. Anschaulich wird der geschätzte Raumpunkt $\hat{P}$ gegeben im KKS der ersten Kamera so gewählt, dass dessen mittels den Matrizen $P1$ und $P2$ projezierten Bildpunkten $\hat{p}^l$ und $\hat{p}^r$ zu den tatsächlich gemessenen Bildpunkten p^l bzw. p^r einen möglichst kleinen Abstand besitzen. Die Theorie der hier verwendeten Methode und weitere alternative Ansätze zur Rekonstruktion von Punkten ist in [12, K. 12] gegeben.

Hinweise:

- Wie schon bei der Stereokalibrierung sollte man auch bei der 3D-Rekonstruktion darauf achten, dass die Bilder der ersten und zweiten Kamera möglichst zeitgleich aufgenommen werden.
- Die Methode heißt *cv::triangulatePoints*, weil man ihr auch mehrere Bildpunktpaare übergeben kann und somit auch mehrere drei-dimensionale Punkte mit einem Aufruf rekonstruieren kann.

- Die Argumente der Methode *cv::triangulatePoints* müssen vom Typ Float sein, damit der verwendete Algorithmus richtige Ergebnisse liefert [17, S. 424].

Wir wollen aber den Punkt P bzw. den geschätzten Punkt $\hat{P}$ nicht im KKS der ersten Kamera haben, sondern in einem von uns angegebenen Weltkoordinatensystem. Im vorherigen Abschnitt wurde erwähnt, dass *OpenCV* bei der Kalibrierung die Position und Orientierung des Schachbrettmusters bezüglich der Kamera feststellen kann. Dabei wird die Methode *cv::solvePnP* verwendet ([16, Z. 1546] und [17, S. 404]), der man die *objectPoints* und die entsprechenden *imagePoints* übergibt. Die Methode berechnet dann den dazugehörigen Rotationsvektor und Translationsvektor.

Der Versuchsaufbau, der in dieser Arbeit verwendet wurde, ist in Abbildung 3.4 illustriert. Die beiden Kameras werden auf einem Schrank positioniert mit Blickrichtung „vorne-unten“. Wir wollen ein Weltkoordinatensystem, bei dem die x-y-Ebene parallel zum Boden ist und die z-Achse zur Decke zeigt. Dazu legen wir das Schachbrettmuster flach auf den Boden. In diesem Aufbau konnte die Methode *cv::detectChessboardcorners* aber das verwendete Schachbrettmuster nicht erkennen und dementsprechend konnten die entsprechenden *imagePoints* nicht bestimmt werden. Dies ist in der Regel dann der Fall, wenn das Muster von der Kamera zu weit entfernt ist[10], oder wenn der Winkel zwischen der optische Achse der Kamera und der Ebene, in der das Schachbrettmuster liegt, zu klein ist. In unserem Fall also, wenn der Winkel α (vergleiche Abbildung 3.4) zwischen der optischen Achse und dem Boden zu klein ist.

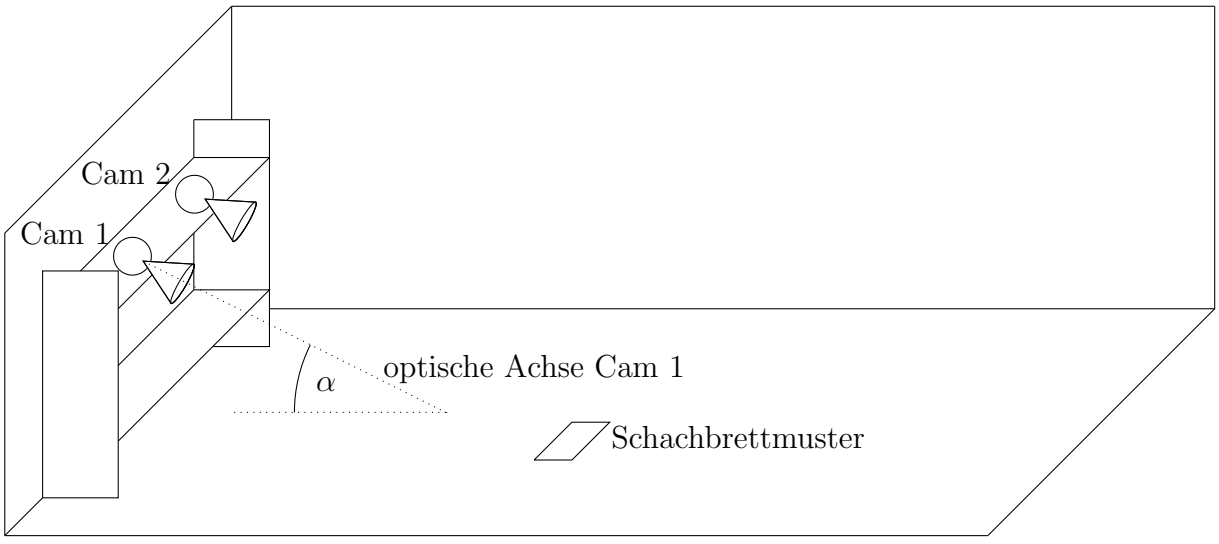

Abbildung 3.4: Versuchsaufbau

[10] bzw. wenn das Muster zu klein ist

Es wurde dann statt dem Schachbrettmuster ein sogenannter AR-Marker[11] verwendet. In Abbildung 3.5 ist der in dieser Arbeit verwendete Marker abgebildet. *OpenCV* stellt mit *Aruco* ein Zusatzpaket zur Verfügung, das den Einsatz von AR-Marker ermöglicht. Mit der Methode *cv::aruco::detectMarkers* lassen sich die Eckpunkte eines AR-Marker in einem Bild bestimmen. Die Methode *cv::aruco::estimatePoseSingleMarkers* stellt dann die Position und Orientierung der gefundenen Marker fest. In der Praxis hat sich gezeigt, dass diese Methode robuster ist, als die Erkennung des Schachbrettmusters. Wird also in Abbildung 3.4 das Schachbrettmuster durch einen AR-Marker ersetzt, so wird dieser Marker erkannt und man kann dessen Lage bestimmen. Auf diese Weise erhält man eine Rotationsmatrix R (eigentlich einen Rotationsvektor) und einen Translationsvektor t um das Weltkoordinatensystem in das KKS der ersten Kamera zu überführen. Um also einen Punkt im KKS in das WKS zu transformieren, wird die Rotationsmatrix $R^{-1} = R^T$ und der Translationsvektor $-R^T t$ verwendet.

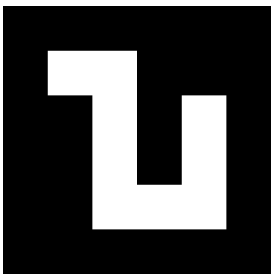

Abbildung 3.5: Beispiel eines AR-Marker

3.1.4 Ergebnisse

Bei der Kalibrierung der Kameras kann man festlegen, welches Kameramodell benutzt werden soll. Man hat dabei unter anderem folgende Optionen:

- Bestimme k_1, k_2, p_1, p_2 und k_3. Die restlichen Verzerrungskoeffizienten werden auf 0 gesetzt.
- Bestimme k_1, k_2, p_1, p_2 und k_3 bis k_6. Die restlichen Verzerrungskoeffizienten werden auf 0 gesetzt.
- Alle Verzerrungskoeffizienten werden bestimmt.

Standardmäßig werden nur die ersten drei radialen Verzerrungskoeffizienten k_1 bis k_3 sowie die beiden tangentialen Verzerrungskoeffizienten p_1 und p_2 verwendet. Tabelle 3.1 vergleicht

[11] Diese Marker werden oft in Augmented-Reality Anwendungen eingesetzt. Daher auch der Name AR-Marker.

die Ergebnisse der Kalibrierung unter Verwendung von 5 bzw. 8 Verzerrungskoeffizienten. Beide Varianten lieferten mit den verwendeten Kameras also ähnliche Ergebnisse. Deshalb bleiben wir bei dem vorgeschlagenen Standardmodell und verwenden nur die fünf Parameter k_1, k_2, k_3, p_1 und p_2.

Anzahl Verzerrungskoeffizienten	5	8
Reprojektionsfehler Kamera 1	0.269306	0.269155
Reprojektionsfehler Kamera 2	0.271555	0.270094
Reprojektionsfehler Stereokamera	0.273156	0.274578

Tabelle 3.1: Ergebnis der (Stereo-) Kalibrierung unter Verwendung von 5 bzw. 8 Verzerrungskoeffizienten.

Die Kalibrierung mit 12 Verzerrungskoeffizienten konnte nicht durchgeführt werden, da die vorliegende *OpenCV* Version anscheinend einen Bug aufweist. In [16, Z. 3401] wird für die Verzerrungskoeffizienten unabhängig von den gesetzten Optionen ein Vektor der Länge 14 angelegt[12]. Dieser Vektor wird in [16, Z. 3452 und 3455] an die Methode übergeben, die die eigentliche Kalibrierung durchführt. Dort wird aber in [16, Z. 1332-1334] geprüft, ob der Vektor die Länge 12 hat. Nachdem dies nicht der Fall ist, wird an dieser Stelle eine Exception geworfen und das Programm bricht ab.

3.2 Tracking

Wollen wir also die Position eines Objektes im Weltkoordinatensystem bestimmen, so benötigen wir die entsprechenden Pixelkoordinaten p^l und p^r, die die Position des Objektes im Bild der ersten bzw. zweiten Kamera angeben. Diese Werte sind aber zunächst natürlich unbekannt und müssen erst bestimmt werden. Im Folgenden werden zwei einfache Verfahren vorgestellt, mit denen man die Position des Crazyflies in den Bildern bestimmen kann. Beide Verfahren nutzen dabei aus, dass der Crazyflie mit einer LED ausgestattet worden ist.

3.2.1 Hellster Punkt

Das erste Verfahren nimmt an, dass die hellsten Pixel im Bild von der LED des Crazyflies erzeugt werden. Um also die Bildkoordinaten des Crazyflies zu bestimmen, wird nach dem hellsten Pixel gesucht. *OpenCV* stellt dazu die Methode *cv::minMaxLoc* bereit, der man ein Graubild übergibt. Die Pixel des Bildes werden nun reihenweise und von links nach rechts

[12] Offensichtlich können also auch noch ein 13-ter und 14-ter Verzerrungskoeffizient angegeben werden. Diese werden in [17, Seite 396] aber noch nicht erwähnt. Da damit auch nicht klar ist, wie diese Parameter in das Kameramodell einfließen, wurde die Kalibierung dieser 14 Parameter in dieser Arbeit auch nicht getestet.

durchgegangen um so den minimalen und maximalen Pixelwert sowie die entsprechenden Pixelkoordinaten zu bestimmen. Je heller ein Pixel ist, desto größer ist dessen Pixelwert. Wir nehmen also an, dass die Koordinaten des Pixels mit dem größten Wert die Position des Crazyflies im Bild darstellt.

Ist der Arbeitsraum ausreichend abgedunkelt, so wird diese Vorgehensweise robust genug sein. Ist dies nicht der Fall, so kann dieses Verfahren schnell scheitern und falsche Ergebnisse liefern. Im Praxistest wurde zum Beispiel ab und zu eine Reflektion einer anderen Lichtquelle (die sich nicht im Arbeitsbereich befand) als hellster Punkt erkannt. Siehe dazu auch Abbildung 3.6. In diesem Fall hat es sich nur um kleine Störpixel gehandelt. Diese kann man beseitigen, indem man über das Bild einen Filter laufen lässt. Dabei wird der Wert jedes Pixels mit den Werten der Pixel in dessen Umgebung verglichen und angepasst. *OpenCV* stellt mit *cv::GaussianBlur* einen Gauß-Filter zur Verfügung. Wendet man diesen Filter auf das Bild an und sucht dann nach dem hellsten Pixel, so hat sich das Verfahren als robuster herausgestellt. Dies geht allerdings zu Lasten der Laufzeit. Im Algorithmus 1 ist dieses einfache Verfahren dargestellt.

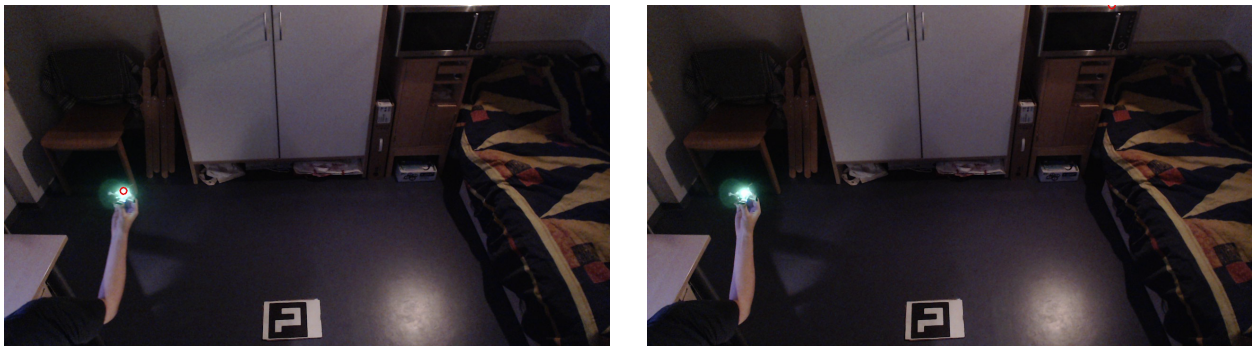

Abbildung 3.6: Suche nach dem hellsten Pixel ohne Anwendung des Gauß-Filters. Im linken Bild wurde der Crazyflie richtig erkannt (roter Kreis), im rechten Bild wurde aber ein Punkt am oberen, rechten Bildrand als hellster Pixel bestimmt.

Algorithmus 1 Suche nach dem hellsten Pixel.

```
1: procedure BRIGHTESTPIXEL(img)
2:     img ← gaussianBlur(img)
3:     (minVal, maxVal, minLoc, maxLoc) ← minMaxLoc(img)
4:     return maxLoc
```

3.2.2 Hellster Bereich

Das zweite Verfahren geht auch davon aus, dass die hellsten Pixel im Bild von der LED des Crazyflies erzeugt werden. Anstatt nun aber nur nach einem Pixel zu suchen, wird die

größte Ansammlung von hellen Pixel gesucht. Zunächst wird wieder der Wert des hellsten Pixels bestimmt. Mit der Methode *cv::threshold* erstellen wir dann ein Schwarz-Weiß-Bild. Dabei übergeben wir neben einem Graubild auch einen Schwellwert. Für jeden Pixel des Bildes wird nun dessen Wert mit dem Schwellwert verglichen. Ist der Wert größer als der Schwellwert, so wird der entsprechende Pixel im Schwarz-Weiß-Bild auf weiß gesetzt, ansonsten auf schwarz. Als Schwellwert benutzen wir den Wert des hellsten Pixels, der gefunden wurde. Um die Robustheit zu erhöhen, kann man diesen Wert auch etwas verringern. In dieser Arbeit wurde der Wert $maxVal - 25$ verwendet. Mit der Methode *cv::findContours* werden nun die Konturen des Bildes bestimmt. Eine zusammenhängende Fläche von weißen Pixeln entspricht dabei einer Kontur. Mit der Methode *cv::contourArea* kann man die Größe dieser Kontur berechnen. Wir nehmen nun an, dass die größte Kontur von der LED des Crazyflies erzeugt wird. Um diese Kontur legen wir mit *cv::boundingRect* ein Rechteck. Dessen Mittelpunkt interpretieren wir dann als Pixelkoordinate des Crazyflies im Bild.

Dieses Verfahren ist schneller als die Suche nach dem hellsten Pixel, wenn dort der Gauß-Filter angewendet wurde und ist zudem relativ robust. Daher wurde schließlich dieses Verfahren zum Tracken des Crazyflies verwendet. Algorithmus 2 zeigt den Pseudo-Code dieses Verfahrens.

Algorithmus 2 Suche nach dem hellsten Bereich.

```
1:  procedure BRIGHTESTSPOT(img, r)
2:      currentAera ← 0
3:      maxAera ← 0
4:      (minVal, maxVal, minLoc, maxLoc) ← minMaxLoc(img)
5:      img ← threshold(img, maxVal − r, 255)
6:      contours ← findContours(img)
7:      for each currentContour in contours do
8:          currentAera ← contourAera(currentContour)
9:          if currentAera > maxAera then
10:             bestContour ← currentContour
11:             maxAera ← currentAera
12:     if maxAera = 0 then
13:         return (false, 0)
14:     else
15:         rectangle ← boundingRect(bestContour)
16:         loc ← center(rectangle)
17:         return (true, loc)
```

3.3 Ergebnisse der 3D-Rekonstruktion

Um erste Erkenntnisse über die Genauigkeit der 3D-Rekonstruktion zu erhalten, wurde der Crazyflie an (per Hand) ausgemessene Positionen platziert. Mit der Stereo-Kamera und dem in Abschnitt 3.2.2 vorgestellten Trackingverfahren wurde dann die Position des Crazyflies rekonstruiert und mit den tatsächlichen Werten verglichen. Das Ergebnis dieser Testreihe ist in den folgenden vier Abbildungen dargestellt. In grün ist jeweils die ausgemessene Position in x-, y- bzw. z- Richtung eingetragen. Die rote Linie resultiert aus der 3D-Rekonstruktion. Bei jedem Test wurden 100 Aufnahmen gemacht. Die Abweichung betrug bis auf eine Ausnahme maximal 10 Zentimeter. Wie sich das im Praxistest bei der Positionsregelung niederschlägt, wird sich später in Kapitel 6 zeigen.

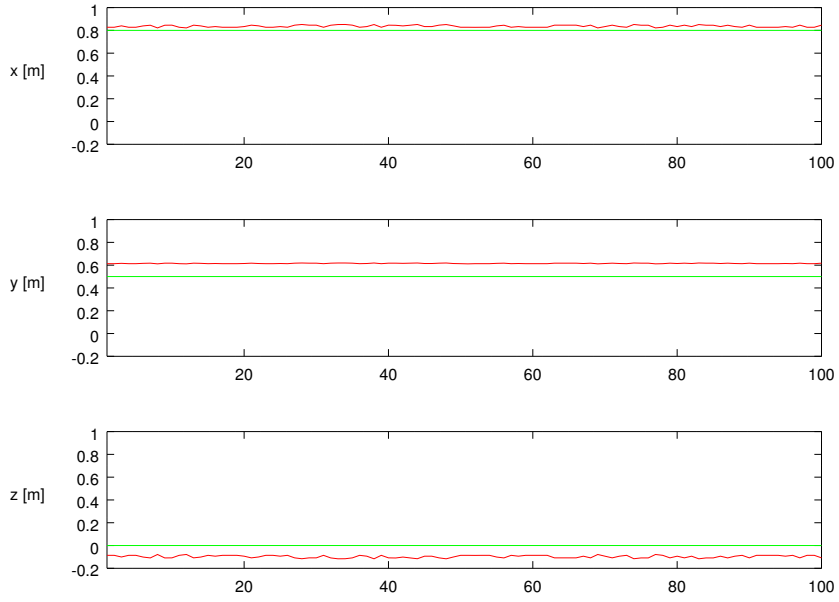

Abbildung 3.7: grün: per Hand gemessene Position, rot: 3D-rekonstruierte Position

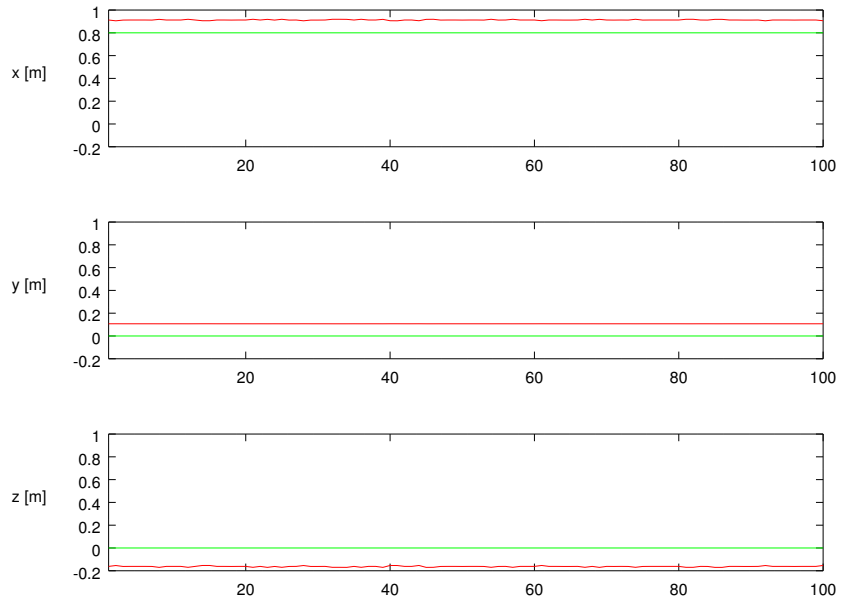

Abbildung 3.8: grün: per Hand gemessene Position, rot: 3D-rekonstruierte Position

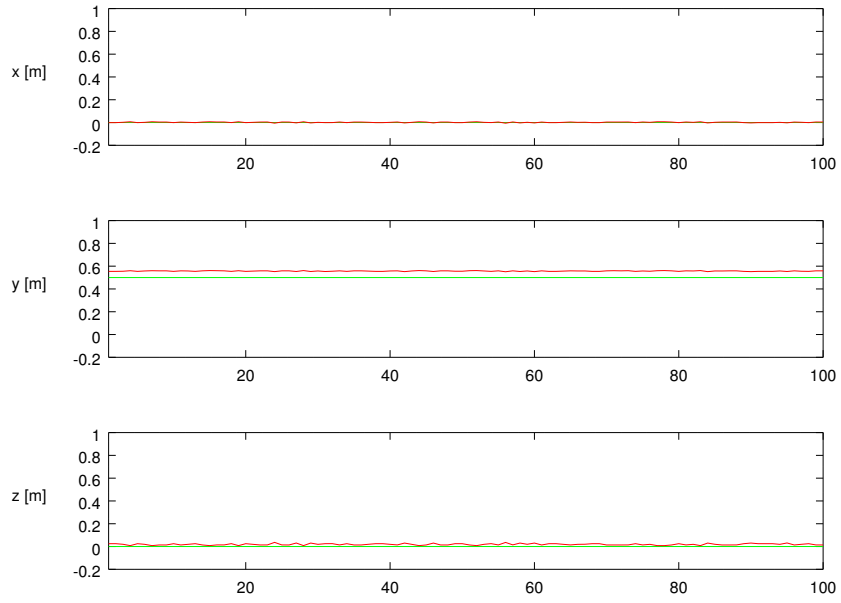

Abbildung 3.9: grün: per Hand gemessene Position, rot: 3D-rekonstruierte Position

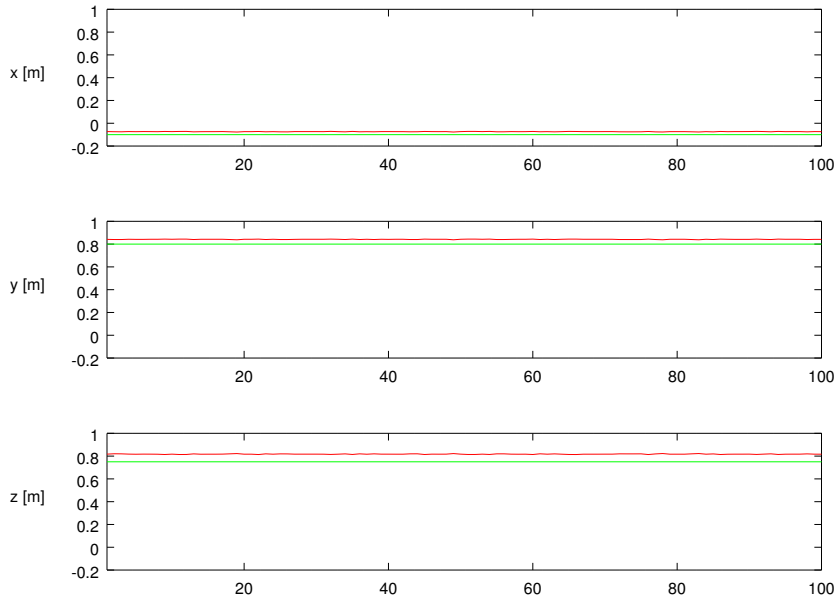

Abbildung 3.10: grün: per Hand gemessene Position, rot: 3D-rekonstruierte Position

Kapitel 4

Linear-quadratischer Regeler

4.1 Theorie

In diesem Abschnitt gehen wir auf für unser Anliegen wichtige Resultate aus der Kontrolltheorie ein. Dabei werden wir auf die Beweise der Sätze verzichten. Die einzelnen Ergebnisse stammen aus [11].

Definition 4.1 Ein lineares zeitinvariantes Kontrollsystem ist gegeben durch die Differentialgleichung

$$\dot{x}(t) = Ax(t) + Bu(t)$$

mit $A \in \mathbb{R}^{n\times n}$ und $B \in \mathbb{R}^{n\times m}$

Definition 4.2 Eine quadratische Kostenfunktion $g : \mathbb{R}^n \times \mathbb{R}^m \to \mathbb{R}_0^+$ ist gegeben durch

$$g(x,u) = \begin{pmatrix} x^T & u^T \end{pmatrix} \begin{pmatrix} M & R \\ R^T & N \end{pmatrix} \begin{pmatrix} x \\ u \end{pmatrix}$$

mit $M \in \mathbb{R}^{n\times n}$, $R \in \mathbb{R}^{n\times m}$ und $N \in \mathbb{R}^{m\times m}$, so dass $G := \begin{pmatrix} M & R \\ R^T & N \end{pmatrix}$ symmetrisch und positiv definit ist.

Definition 4.3 Sei $x(t, x_0, u)$ die Lösung eines linearen Kontrollsystems mit Anfangsbedingung $x(0, x_0, u) = x_0$. Sei zudem g eine quadratische Kostenfunktion. Dann ist das linear-quadratische optimale Steuerungsproblem gegeben durch das Optimierungsproblem:

$$\text{Minimiere } J(x_0, u) := \int_0^\infty g(x(t, x_0, u), u(t)dt$$
$$\text{über } u \in U \text{ für jedes } x_0 \in \mathbb{R}^n$$

Die Funktion

$$V(x_0) := \inf_{u\in U} J(x_0, u)$$

wird als optimale Wertefunktion dieses optimalen Steuerungsproblems bezeichnet.

Ein Paar $(x^*, u^*) \in \mathbb{R}^n \times U$ mit $J(x^*, u^*) = V(x^*)$ wird als optimales Paar bezeichnet.

Lemma 4.4 Betrachte das linear-quadratische optimale Steuerungsproblem. Wenn die Matrix $Q \in \mathbb{R}^{n\times n}$ eine symmetrische und positiv definite Lösung der algebraischen Riccati-Gleichung

$$QA + A^TQ + M - (QB + R)N^{-1}(B^TQ + R^T) = 0 \tag{4.1}$$

ist, so ist die optimale Wertefunktion des Problems gegeben durch $V(x) = x^TQx$.

Lemma 4.5 Falls das linear-quadratische optimale Steuerungsproblem eine optimale Wertefunktion der Form $V(x) = x^TQx$ besitzt, so sind die optimalen Paare von der Form (x^*, u^*) mit

$$u^*(t) = Fx(t, x^*, F)$$

und $F \in \mathbb{R}^{m\times n}$ gegeben durch

$$F = -N^{-1}(B^TQ + R^T),$$

wobei $x(t, x^*, F)$ die Lösung des mittels F geregelten Systems

$$\dot{x}(t) = (A + BF)x(t)$$

mit Anfangsbedingung $x(0, x^*, F) = x^*$ bezeichnet.

Darüber hinaus ist das mittels F geregelte System exponentiell stabil.

Satz 4.6 Gegeben sei ein nichtlineares Kontrollsystem $\dot{x}(t) = f(x(t), u(t))$ mit $f(0, 0) = 0$ und Linearisierung

$$\begin{aligned}\dot{x}(t) &= Ax(t) + Bu(t), \text{ wobei}\\ A &= \frac{\partial f}{\partial x}(0, 0)\\ B &= \frac{\partial f}{\partial u}(0, 0)\end{aligned}$$

Dann gilt:

Ein lineares Feedback $F \in \mathbb{R}^{m\times n}$ stabilisiert den Nullpunkt $x^* = 0$ des nichtlinearen Kontrollsystems lokal exponentiell genau dann, wenn F die Linearisierung global exponentiell stabilisiert.

Bemerkung 4.7 Soll ein nichtlineares Kontrollsystem $f(x, u)$ um einen Gleichgewichtspunkt $(x^*, u^*) \neq (0, 0)$ stabilisiert werden, so kann Satz 4.6 trotzdem angewendet werden. Man muss dazu aber ein transformiertes Kontrollsystem $\tilde{f}(x, u)$ einführen:

$$\tilde{f}(x, u) = f(x + x^*, u + u^*)$$

Stabilisiert ein lineares Feedback $\tilde{F}$ die Linearisierung des transformierten Systems $\tilde{f}$ global exponentiell, so stabilisiert es den Nullpunkt des nichtlinearen Systems $\tilde{f}$ lokal exponentiell. Bezeichnet $\hat{x}$ den gemessenen Systemzustand des Systems f, so stabilisiert

$$u = \tilde{F}(\hat{x} - x^*) + u^*$$

dieses System lokal exponentiell um (x^*, u^*), da mit $\hat{x} = x + x^*$ folgendes gilt:

$$\begin{aligned}\tilde{f}(x, \tilde{F}x) &= f(x + x^*, \tilde{F}x + u^*) \\ &= f[\hat{x}, \tilde{F}(\hat{x} - x^*) + u^*].\end{aligned}$$

4.2 Matrix Signumsfunktion

Kennen wir also die symmetrische und positiv definitie Lösung der algebraischen Riccati Gleichung (4.2), so kann daraus mit Lemma 4.5 ein stabilisierendes Feedback konstruiert werden. In diesem Abschnitt werden wir die Matrix Signumsfunktion vorstellen und näher betrachten. Wir beziehen uns dabei auf [19, K. 4] und [13, K. 5.3]. Mit deren Hilfe können wir dann im nächsten Abschnitt die algebraische Riccati Gleichung lösen.

Definition 4.8 Seien $p \in \mathbb{N}$ und $G \in \mathbb{C}^{p\times p}$, wobei G nur Eigenwerte mit Realteil ungleich 0 besitzt. Sei $K = V^{-1}GV$ die jordansche Normalform von G. Die Matrix Signumsfunktion von G ist dann definiert durch:

$$\operatorname{sign}(G) := VSV^{-1}$$

wobei $S = \operatorname{diag}(s_1, \cdots, s_p)$ und für $i = 1, \cdots, p$ gilt:

$$s_i = \begin{cases} +1, & \text{falls } \operatorname{Re}(K_{ii}) > 0 \\ -1, & \text{falls } \operatorname{Re}(K_{ii}) < 0. \end{cases}$$

Ohne Einschränkung kann angenommen werden, dass die Jordanblöcke so angeordnet sind, dass $\operatorname{Re}(K_{11}), \cdots, \operatorname{Re}(K_{NN}) < 0$ und $\operatorname{Re}(K_{(N+1)(N+1)}), \cdots, \operatorname{Re}(K_{pp}) > 0$ gilt. Sei V die entsprechende Transformationmatrix, so gilt mit $P = p - N$:

$$\operatorname{sign}(G) = V \begin{pmatrix} -I_N & 0 \\ 0 & I_P \end{pmatrix} V^{-1}$$

Bemerkung 4.9 Die Matrix Signumsfunktion von G ist nicht definiert, wenn G einen Eigenwert mit Realteil gleich 0 besitzt.

Satz 4.10 Sei $G \in \mathbb{C}^{p\times p}$ und die Matrix Signumsfunktion von G existiere. Seien $\lambda_1, \cdots, \lambda_p$ die Eigenwerte von G. Dann sind die Eigenwerte von $G \pm \operatorname{sign}(G)$ gegeben durch:

$$\mu_i = \lambda_i \pm \operatorname{sign}(\lambda_i), \text{ für } i = 1, \cdots, p.$$

Beweis Sei $K = V^{-1}GV$ die jordansche Normalform von G mit zugehöriger Transformationsmatrix V. Es gilt:

$$\begin{aligned} G \pm \operatorname{sign}(G) &= VKV^{-1} \pm V \begin{pmatrix} -I_N & 0 \\ 0 & I_P \end{pmatrix} V^{-1} \\ &= V(K \pm \begin{pmatrix} -I_N & 0 \\ 0 & I_P \end{pmatrix})V^{-1} \end{aligned}$$

Somit sind die Eigenwerte von $G \pm \operatorname{sign}(G)$ durch die Diagonalelemente der oberen Dreiecksmatrix $J := K \pm \begin{pmatrix} -I_N & 0 \\ 0 & I_P \end{pmatrix}$ gegeben. Da K die jordansche Normalform von G ist, sind die Diagonalelemente K_{ii} die Eigenwerte von G. K ist so konstruiert, dass der Realteil der ersten N Diagonalelemente negativ ist und der restlichen Diagonalelemente positiv. Somit folgt mit $\lambda_i = K_{ii}$ und $\mu_i = J_{ii}$ die Behauptung.

Folgerung 4.11 Für den Realteil der Eigenwerte $\mu_i = \lambda_i + \operatorname{sign}(\lambda_i)$ von $G + \operatorname{sign}(G)$ gilt also:

$$\operatorname{Re}(\lambda_i + \operatorname{sign}(\lambda_i)) = \begin{cases} \operatorname{Re}(\lambda_i) + 1, & \text{falls } \operatorname{Re}(\lambda_i) > 0 \\ \operatorname{Re}(\lambda_i) - 1, & \text{falls } \operatorname{Re}(\lambda_i) < 0 \end{cases}$$

Der Realteil aller Eigenwerte von $G + \operatorname{sign}(G)$ ist somit ungleich 0. Es folgt, dass die Matrix $G + \operatorname{sign}(G)$ invertierbar ist.

Satz 4.12 Seien $p \in \mathbb{N}$ und $G \in \mathbb{C}^{p \times p}$, wobei G nur Eigenwerte mit Realteil ungleich 0 besitzt. Dann gilt:

$$G \operatorname{sign}(G) = \operatorname{sign}(G)G$$

Beweis Es gilt:

$$\begin{aligned} G \operatorname{sign}(G) &= VKV^{-1}V \begin{pmatrix} -I_N & 0 \\ 0 & I_P \end{pmatrix} V^{-1} \\ &= VK \begin{pmatrix} -I_N & 0 \\ 0 & I_P \end{pmatrix} V^{-1} \end{aligned}$$

und

$$\begin{aligned} \operatorname{sign}(G)G &= V \begin{pmatrix} -I_N & 0 \\ 0 & I_P \end{pmatrix} V^{-1}VKV^{-1} \\ &= V \begin{pmatrix} -I_N & 0 \\ 0 & I_P \end{pmatrix} KV^{-1} \end{aligned}$$

Es muss also gezeigt werden, dass K und $I^{\mp} := \begin{pmatrix} -I_N & 0 \\ 0 & I_P \end{pmatrix}$ kommutieren. Allgemein gilt für die Multiplikation von rechts bzw. von links einer Matrix A mit einer Diagonalmatrix D

passender Dimension:

$$(AD)_{ij} = A_{ij}D_{jj}$$
$$(DA)_{ij} = D_{ii}A_{ij}$$

K liegt in jordanscher Normalform vor. Somit gilt $K_{i(i+1)} \in \{0,1\}$ für $i = 1, \cdots, p-1$, $K_{ii} \in \mathbb{C}$ für $i = 1, \cdots, p$ und die restlichen Einträge sind gleich 0. Für die Hauptdiagonaleinträge von $KI^{\mp}$ bzw. $I^{\mp}K$ gilt damit:

$$(KI^{\mp})_{ii} = K_{ii}I^{\mp}_{ii} = I^{\mp}_{ii}K_{ii} = (I^{\mp}K)_{ii} \text{ , für } i = 1, \cdots, p$$

Für die Einträge der unteren Nebendiagonalen und der k-ten oberen Nebendiagonalen mit $k > 1$ gilt:

$$(KI^{\mp})_{ij} = 0I^{\mp}_{ij} = 0 = I^{\mp}_{ii}0 = (I^{\mp}K)_{ij}$$

Für die Kommutativität von K und $I^{\mp}$ muss also nur noch gezeigt werden, dass die Einträge der ersten oberen Nebendiagonale übereinstimmen. Es gilt:

$$(KI^{\mp})_{i(i+1)} = K_{i(i+1)}I^{\mp}_{(i+1)(i+1)}$$
$$(I^{\mp}K)_{i(i+1)} = I^{\mp}_{ii}K_{i(i+1)}$$

Ist $K_{i(i+1)} = 0$, so gilt offensichtlich $(KI^{\mp})_{i(i+1)} = 0 = (I^{\mp}K)_{i(i+1)}$.
Ist $K_{i(i+1)} = 1$, so bedeutet das, dass K_{ii} und $K_{(i+1)(i+1)}$ zum gleichen Jordanblock gehören. Damit gilt insbesondere $K_{ii} = K_{(i+1)(i+1)}$. Nach der Definition der Matrix Signumsfunktion gilt $I^{\mp}_{ii} = \text{sign}(K_{ii})$ für $i = 1, \cdots, p$. Somit gilt $I^{\mp}_{(i+1)(i+1)} = \text{sign}(K_{(i+1)(i+1)}) = \text{sign}(K_{ii}) = I^{\mp}_{ii}$. Damit stimmen $KI^{\mp}$ und $I^{\mp}K$ überein und die Kommutativität von G und $\text{sign}(G)$ ist gezeigt.

Satz 4.13 Seien $p \in \mathbb{N}$ und $G \in \mathbb{C}^{p\times p}$, wobei G nur Eigenwerte mit Realteil ungleich 0 besitzt. Dann sind die Eigenwerte der Matrix $X := (G - \text{sign}(G))(G + \text{sign}(G))^{-1}$ gegeben durch:

$$\mu_i = \frac{\lambda_i - \text{sign}(\lambda_i)}{\lambda_i + \text{sign}(\lambda_i)} \text{ , mit } \lambda_i \in \Lambda(G)$$

Beweis Folgerung 4.11 liefert die Existenz der Matrix X. Es gilt:

$$\begin{aligned}
X &= (G - \text{sign}(G))(G + \text{sign}(G))^{-1} \\
&= (VKV^{-1} - V\begin{pmatrix} -I_N & 0 \\ 0 & I_P \end{pmatrix}V^{-1})(VKV^{-1} + V\begin{pmatrix} -I_N & 0 \\ 0 & I_P \end{pmatrix}V^{-1})^{-1} \\
&= V(K - \begin{pmatrix} -I_N & 0 \\ 0 & I_P \end{pmatrix})V^{-1}(V(K + \begin{pmatrix} -I_N & 0 \\ 0 & I_P \end{pmatrix})V^{-1})^{-1} \\
&= VK_-V^{-1}VK_+^{-1}V^{-1} \\
&= VK_-K_+^{-1}V^{-1}
\end{aligned}$$

mit

$$K_- := K - \begin{pmatrix} -I_N & 0 \\ 0 & I_P \end{pmatrix}$$
$$K_+ := K + \begin{pmatrix} -I_N & 0 \\ 0 & I_P \end{pmatrix}$$

K_+ ist eine obere Dreiecksmatrix mit Diagonaleinträgen $\lambda_i + \text{sign}(\lambda_i)$. Die Inverse ist damit eine obere Dreiecksmatrix mit Diagonaleinträgen $(\lambda_i + \text{sign}(\lambda_i))^{-1}$. $K_- K_+^{-1}$ ist als Produkt zweier oberer Dreiecksmatrizen wieder eine Dreiecksmatrix mit den Diagonalelmenten $(\lambda_i - \text{sign}(\lambda_i))(\lambda_i + \text{sign}(\lambda_i))^{-1}$. Die Diagonalelemente entsprechen den Eigenwerten von X und damit folgt die Behauptung.

Lemma 4.14 Seien $p \in \mathbb{N}$, $G \in \mathbb{C}^{p\times p}$ und sign(G) existiere. Dann gilt:
Das Newton Verfahren zur Berechnung der Quadratwurzel von I_p mit der Iterationsvorschrift $W_1 = G$ und $W_{k+1} = (W_k + W_k^{-1})/2$ konvergiert gegen sign(G).

Beweis Mit $\text{sign}(G)\text{sign}(G) = V \begin{pmatrix} -I_N & 0 \\ 0 & I_P \end{pmatrix} V^{-1} V \begin{pmatrix} -I_N & 0 \\ 0 & I_P \end{pmatrix} V^{-1} = V I_p V^{-1} = I_p$ folgt sofort, dass sign(G) eine Quadratwurzel von I_p ist. Es bleibt zu zeigen, dass das Newton Verfahren mit der gegebenen Iterationsvorschrift tatsächlich gegen sign(G) konvergiert.

Zunächst wollen wir per Induktion die Existenz von W_k^{-1} zeigen.

- Induktionsanfang $k = 1$:
  Nach Voraussetzung existiert die Matrix Signumsfunktion von G und damit besitzt G keine Eigenwerte mit Realteil gleich 0. Es folgt, dass G und somit auch $W_1 = G$ invertierbar sind.

- Induktionsschritt $k \to k + 1$:
  Sei λ_k beliebiger Eigenwert von W_k. Dann gilt mit einem zugehörigen Eigenvektor v_k:

$$\begin{aligned} W_{k+1} v_k &= \frac{1}{2}(W_k + W_k^{-1}) v_k \\ &= \frac{1}{2}(W_k v_k + W_k^{-1} v_k) \\ &= \frac{1}{2}(\lambda_k v_k + \lambda_k^{-1} v_k) \\ &= \frac{1}{2}(\lambda_k + \lambda_k^{-1}) v_k \end{aligned}$$

  Die Eigenwerte von W_{k+1} sind also von der Form $\lambda_{k+1} = \frac{1}{2}(\lambda_k + \lambda_k^{-1})$ mit λ_k Eigenwert von W_k. Sei $re^{\mathrm{i}\theta}$ die Polardarstellung von $\lambda_k = r_k e^{\mathrm{i}\theta_k} = r_k \cos(\theta_k) + \mathrm{i} r_k \sin(\theta_k)$. Damit

gilt für die Eigenwerte von W_{k+1}:

$$\begin{aligned}\frac{1}{2}(\lambda_k + \lambda_k^{-1}) &= \frac{1}{2}(r_k e^{\mathrm{i}\theta_k} + r_k^{-1} e^{-\mathrm{i}\theta_k}) \\ &= \frac{1}{2}(r_k + r_k^{-1})\cos(\theta_k) + \mathrm{i}\frac{1}{2}(r_k - r_k^{-1})\sin(\theta_k)\end{aligned}$$

Nach Induktionsvoraussetzung ist der Realteil von λ_k ungleich 0. Es folgt, dass, sowohl r_k als auch $\cos(\theta_k)$ ungleich 0 sind und damit gilt auch $\mathrm{Re}(\lambda_{k+1}) = \frac{1}{2}(r_k + r_k^{-1})\cos(\theta_k) \neq 0$.

- Induktionsschluss:
  Da alle Eigenwerte von W_{k+1} einen Realteil ungleich 0 besitzen, ist W_{k+1} invertierbar. Insbesondere gilt auch: $\mathrm{sign}(W_{k+1}) = \mathrm{sign}(W_k) = \cdots \mathrm{sign}(W_0) = \mathrm{sign}(G) = S$.

Nach Satz 4.12 kommutiert $S = \mathrm{sign}(W_k)$ mit W_k und somit gelten die folgenden Gleichungen:

$$\begin{aligned}W_{k+1} \pm S &= \frac{1}{2}(W_k + W_k^{-1} \pm 2S) \\ &= \frac{1}{2}W_k^{-1}(W_k^2 \pm 2W_k S + I_p) \\ &= \frac{1}{2}W_k^{-1}(W_k \pm S)^2.\end{aligned}$$

Folgerung 4.11 garantiert mit $S = \mathrm{sign}(W_k) = \mathrm{sign}(W_{k+1})$ die Existenz folgender Matrix:

$$\begin{aligned}(W_{k+1} - S)(W_{k+1} + S)^{-1} &= \frac{1}{2}W_k^{-1}(W_k - S)^2 2(W_k + S)^{-2} W_k \\ &= (W_k - S)^2 W_k^{-1} W_k (W_k + S)^{-2} \\ &= ((W_k - S)(W_k + S)^{-1})^2\end{aligned}$$

Setze nun $X_k := (W_k - S)(W_k + S)^{-1}$. Offensichtlich gilt $X_{k+1} = X_k^2 = \cdots = X_1^{2^k}$. Für die Eigenwerte μ_i von $X_1 = (G - S)(G + S)^{-1}$ gilt nach Satz 4.13

$$\mu_i = \frac{\lambda_i - \mathrm{sign}(\lambda_i)}{\lambda_i + \mathrm{sign}(\lambda_i)}$$

für $\lambda_i = a_i + \mathrm{i}b_i$ Eigenwert von G. Mit

$$\begin{aligned}|\mu_i|^2 &= \frac{|\lambda_i - \mathrm{sign}(\lambda_i)|^2}{|\lambda_i + \mathrm{sign}(\lambda_i)|^2} \\ &= \frac{(a_i - \mathrm{sign}(a_i))^2 + b_i^2}{(a_i + \mathrm{sign}(a_i))^2 + b_i^2} \\ &= \frac{a_i^2 - 2a_i\mathrm{sign}(a_i) + 1 + b_i^2}{a_i^2 + 2a_i\mathrm{sign}(a_i) + 1 + b_i^2} \\ &= \frac{a_i^2 - 2|a_i| + 1 + b_i^2}{a_i^2 + 2|a_i| + 1 + b_i^2} \\ &< 1\end{aligned}$$

folgt für den Spektralradius von X_1: $\rho(X_1) < 1$.
Es gilt also:

$$X_k = X_1^{2^k} \longrightarrow 0_p \text{ für } k \longrightarrow \infty$$

und damit konvergiert das Newton Verfahren zur Berechung der Quadratwurzel von I_p unter der gegebenen Iterationsvorschrift gegen sign(G):

$$W_k = (I_p - X_k)^{-1}(I_p + X_k)S \longrightarrow S = \text{sign}(G) \text{ für } k \longrightarrow \infty,$$

wobei die erste Gleichheit aus der Konstruktion von X_k folgt.

Numerische Betrachtungen

Das Verfahren konvergiert letztendlich zwar quadratisch, aber die Anfangsgeschwindigkeit kann relativ langsam sein. Man kann diese aber durch einen passenden Skalierungsfaktor $\alpha_k > 0$ in der Newton Iterationsvorschrift beschleunigen:

$$W_{k+1} = \frac{1}{2}(\alpha_k W_k + \alpha_k^{-1} W_k^{-1})$$

[19, S. 284] schlägt als Skalierung im k-ten Schritt den Faktor $\alpha_k := |\det(W_k)|^{-1/p}$ vor. Hintergrund dieser Wahl ist, dass die Eigenwerte der skalierten Matrix möglichst nahe dem Einheitskreis sein sollen. Es gilt:

$$|\det(W_k)|^{-1/p} = \underset{c>0}{\text{argmin}} \sum_{\lambda \in \Lambda(cW_k)} (\ln|\lambda|)^2$$

In [13, S. 119] werden auch noch alternative Skalierungsfaktoren vorgestellt, die vom Spektralradius bzw. von der Norm der Matrix W_k abhängen:

$$\alpha_k = \sqrt{\rho(W_k^{-1})/\rho(W_k)}$$
$$\alpha_k = \sqrt{||W_k^{-1}||/||W_k||}$$

In dieser Arbeit wurde der erste Skalierungsfaktor verwendet, der über die Determinante bestimmt wird.

Laut [19, S. 283] ist es aus numerischer Sicht günstiger, wenn man in 4.14 die Iterationsvorschrift

$$W_{k+1} = (W_k + W_k^{-1})/2$$

durch die mathematisch äquivalente Form

$$W_{k+1} = W_k - (W_k - W_k^{-1})/2$$

ersetzt.

Diese Ergebnisse resultieren im Algorithmus 3, mit dem man die Matrix Signumsfunktion passender Matrizen G berechnen kann. Da es sich hier um ein iteratives Verfahren handelt, müssen auch geeignete Abbruchbedingungen eingeführt werden. Die erste Bedingung in Zeile 10 sorgt dafür, dass das Verfahren abbricht, wenn sich W durch die Iteration kaum noch ändert. ϵ_M bezeichnet dabei die relative Maschinengenauigkeit. Die dritte Bedingung in Zeile 10 bricht die Iteration ab, wenn die Rundungsfehler beim Invertieren der Matrix Z zu groß werden. Nach [19, S. 300f.] tritt die dritte Bedingung in der Praxis aber manchmal zu früh ein. Mit der Forderung $k \geq \tau$ kann man diesen Fall künstlich hinauszögern. Sollte keine dieser Bedingungen zum Abbruch der Iteration führen, so wird das Verfahren nach der k_{max}-ten Iteration beendet und die Nullmatrix zurückgegeben.

Algorithmus 3 Skaliertes Newton-Verfahren für die Matrix Signumsfunktion

```
1:  procedure SIGNFUNCTION(G,k_max)
2:      n ← rows(G)
3:      τ ← 8
4:      W ← G
5:      for k ← 1 to k_max do
6:          d ← |det(W)|^(1/(2n))
7:          Z ← W/d
8:          κ ← cond(W)
9:          W ← Z − (Z − Z^(−1))/2
10:         s ← ||Z − Z^(−1)||
11:         if s/2 ≤ ε_M||Z|| or k ≥ τ and s ≤ 2nε_Mκ||Z^(−1)|| then
12:             return (W, k)                          ▷ Algorithm converged
13:     return (0_n, −1)                               ▷ No solution
```

4.3 Verfahren zum Lösen der algebraischen Riccati Gleichung

In diesem Abschnitt wird ein Verfahren vorgestellt, mit dem man die algebraische Riccati Gleichung (4.1) mit $R = 0$:

$$QA + A^TQ + M - QBN^{-1}B^TQ = 0 \tag{4.2}$$

lösen kann. Das Verfahren verwendet dabei die Matrix Signumsfunktion aus Abschnitt 4.2. In diesem Abschnitt beziehen wir uns wieder auf [19, K. 4].

Satz 4.15 Sei $Q \in \mathbb{R}^{n\times n}$ die symmetrisch, positiv semidefinite Lösung der algebraischen Riccati Gleichung (4.2). Seien $F := BN^{-1}B^T$ und $\tilde{G} := \begin{pmatrix} A-FQ & -F \\ 0_n & -(A-FQ)^T \end{pmatrix}$. Dann gilt:

$$\begin{pmatrix} A & -F \\ -M & -A^T \end{pmatrix} = \begin{pmatrix} I_n & 0_n \\ Q & I_n \end{pmatrix} \tilde{G} \begin{pmatrix} I_n & 0_n \\ -Q & I_n \end{pmatrix}.$$

Beweis Da die Matrizen Q und F symmetrisch sind, gilt:

$$\begin{aligned}
\begin{pmatrix} I_n & 0_n \\ Q & I_n \end{pmatrix} \tilde{G} \begin{pmatrix} I_n & 0_n \\ -Q & I_n \end{pmatrix} &= \begin{pmatrix} I_n & 0_n \\ Q & I_n \end{pmatrix} \begin{pmatrix} A-FQ & -F \\ 0_n & -(A-FQ)^T \end{pmatrix} \begin{pmatrix} I_n & 0_n \\ -Q & I_n \end{pmatrix} \\
&= \begin{pmatrix} I_n & 0_n \\ Q & I_n \end{pmatrix} \begin{pmatrix} A & -F \\ (A-FQ)^TQ & -(A-FQ)^T \end{pmatrix} \\
&= \begin{pmatrix} A & -F \\ QA+(A-FQ)^TQ & -QF-(A-FQ)^T \end{pmatrix} \\
&= \begin{pmatrix} A & -F \\ QA+A^TQ-QFQ & -A^T \end{pmatrix} \\
&= \begin{pmatrix} A & -F \\ -M & -A^T \end{pmatrix}
\end{aligned}$$

Lemma 4.16 Seien Q die symmetrisch, positiv semidefinite Lösung der algebraischen Riccati Gleichung (4.2), $G := \begin{pmatrix} A & -F \\ -M & -A^T \end{pmatrix}$ mit F symmetrisch und $\operatorname{sign}(G) =: \begin{pmatrix} W_{11} & W_{12} \\ W_{21} & W_{22} \end{pmatrix}$ mit $W_{11}, W_{12}, W_{21}, W_{22} \in \mathbb{R}^{n\times n}$.

Dann gilt:

$$\begin{pmatrix} W_{12} \\ W_{22}+I_n \end{pmatrix} Q = -\begin{pmatrix} W_{11}+I_n \\ W_{21} \end{pmatrix}$$

und $\begin{pmatrix} W_{12} \\ W_{22}+I_n \end{pmatrix}$ besitzt vollen Rang.

Beweis Nach Satz 4.15 gilt:

$$G = \begin{pmatrix} I_n & 0_n \\ Q & I_n \end{pmatrix} \begin{pmatrix} A-FQ & -F \\ 0_n & -(A-FQ)^T \end{pmatrix} \begin{pmatrix} I_n & 0_n \\ -Q & I_n \end{pmatrix}.$$

Als symmetrisch, positive semidefinite Lösung der algebraischen Riccati Gleichung stabilisiert Q das System $A - BN^{-1}B^TQ = A - FQ$ und damit ist der Realteil der Eigenwerte

jeweils kleiner 0. Entsprechend ist der Realteil der Eigenwerte von $-(A-FQ)^T$ jeweils größer 0. Unter Ausnutzung der Darstellung von G gilt für die Matrix Signumsfunktion von G:

$$\begin{aligned}\operatorname{sign}(G) = \begin{pmatrix} W_{11} & W_{12} \\ W_{21} & W_{22} \end{pmatrix} &= \begin{pmatrix} I_n & 0_n \\ Q & I_n \end{pmatrix} \begin{pmatrix} -I_n & Z \\ 0_n & I_n \end{pmatrix} \begin{pmatrix} I_n & 0_n \\ -Q & I_n \end{pmatrix} \\ &= \begin{pmatrix} -I_n & Z \\ -Q & QZ + I_n \end{pmatrix} \begin{pmatrix} I_n & 0_n \\ -Q & I_n \end{pmatrix}\end{aligned}$$

mit $\begin{pmatrix} I_n & 0_n \\ -Q & I_n \end{pmatrix}^{-1} = \begin{pmatrix} I_n & 0_n \\ Q & I_n \end{pmatrix}$. Daraus folgt die äquivalente Darstellung:

$$\begin{pmatrix} W_{11} + W_{12}Q & W_{12} \\ W_{21} + W_{22}Q & W_{22} \end{pmatrix} = \begin{pmatrix} -I_n & Z \\ -Q & QZ + I_n \end{pmatrix}. \tag{4.3}$$

Auf die gleiche Seite gebracht:

$$\begin{pmatrix} 0_n & 0_n \\ 0_n & 0_n \end{pmatrix} = \begin{pmatrix} W_{11} + I_n + W_{12}Q & W_{12} - Z \\ W_{21} + (W_{22} + I_n)Q & W_{22} - QZ - I_n \end{pmatrix}$$

Betrachtet man die ersten n Spalten, so gilt also:

$$\begin{pmatrix} W_{12} \\ W_{22} + I_n \end{pmatrix} Q = - \begin{pmatrix} W_{11} + I_n \\ W_{21} \end{pmatrix}.$$

Damit ist der erste Teil der Behauptung gezeigt. Um zu zeigen, dass $\begin{pmatrix} W_{12} \\ W_{22} + I_n \end{pmatrix}$ vollen Rang hat, betrachte folgendes Hilfssystem:

$$\begin{aligned}(-Q \quad I_n) \begin{pmatrix} W_{12} \\ W_{22} + I_n \end{pmatrix} &= (-Q \quad I_n) \begin{pmatrix} Z \\ QZ + 2I_n \end{pmatrix} \\ &= 2I_n\end{aligned}$$

Die erste Gleichheit gilt aufgrund der Gleichung (4.3). Es gilt also:
$n = \operatorname{rank}(2I_n) \leq \operatorname{rank} \begin{pmatrix} W_{12} \\ W_{22} + I_n \end{pmatrix} \leq n$ und damit hat $\begin{pmatrix} W_{12} \\ W_{22} + I_n \end{pmatrix}$ vollen Rang.

Folgerung 4.17 Existiert die symmetrische, positiv semidefinite Lösung Q der algebraischen Riccati Gleichung (4.2), so kann man Q durch das Lösen der folgenden linearen Gleichungssysteme berechnen:

$$\begin{pmatrix} W_{12} \\ W_{22} + I_n \end{pmatrix} x_i = - \begin{pmatrix} W_{11} + I_n \\ W_{21} \end{pmatrix}_{*i} \qquad\qquad i = 1, \cdots, n$$

wobei $x_i \in \mathbb{R}^n$ und $(\cdot)_{*i}$ die i-te Spalte einer Matrix kennzeichnet. Nach Lemma 4.16 existieren für diese n Gleichungssysteme Lösungen und da $\begin{pmatrix} W_{12} \\ W_{22} + I_n \end{pmatrix}$ vollen Rang besitzt, der Rang

also somit gleich der Anzahl der Komponenten von x_i ist, sind diese Lösungen auch eindeutig. Die symmetrische, positiv semidefinite Lösung Q der algebraischen Riccati Gleichung ist also damit gegeben durch:

$$Q = (x_1| \cdots |x_n).$$

Im Algorithmus 4 ist dieses Verfahren zum Berechnen der Lösung der algebraischen Riccati Gleichung als Pseudo-Code angegeben. Um das lineare Gleichungssystem

$$\begin{pmatrix} W_{12} \\ W_{22} + I_n \end{pmatrix} x_i = - \begin{pmatrix} W_{11} + I_n \\ W_{21} \end{pmatrix}_{*i} \qquad i = 1, \cdots, n$$

zu lösen, wird in dieser Arbeit die LU-Zerlegung mit vollständiger Pivotisierung der *Eigen*-Bibliothek verwendet. In der Theorie sollte das Ergebnis symmetrisch sein, da die Matrix die symmetrisch, positiv semidefinite Lösung der algebraischen Riccati Gleichung darstellt. Durch Rundungsfehler und weil es sich bei der Berechnung der Signumsfunktion um ein iteratives Verfahren handelt, muss dies aber nicht unbedingt erfüllt sein. Daher wird in Zeile 11 die Matrix Q anschließend noch symmetrisiert, wie es auch in [19, S. 301] empfohlen wird.

Algorithmus 4 Matrix Signumsfunktions-Methode

1: **procedure** SIGNM(A,B,M,N,k_{max})
2: $F \leftarrow BN^{-1}B^T$
3: $W \leftarrow \begin{pmatrix} A & -F \\ -M & -A^T \end{pmatrix}$
4: $(W, k) \leftarrow \text{signfunction}(W, k_{max})$
5: **if** $k = -1$ **then**
6: **return** $(0_n, false)$ ▷ No solution
7: **else**
8: $L \leftarrow \begin{pmatrix} W_{12} \\ W_{22} + I_n \end{pmatrix}$
9: $R \leftarrow - \begin{pmatrix} W_{11} + I_n \\ W_{21} \end{pmatrix}$
10: $Q \leftarrow \text{solve}(L, R)$
11: $Q \leftarrow (Q + Q^T)/2$
12: **return** $(Q, true)$ ▷ Solution found

Kapitel 5

Kalman Filter

Für die Regelung von Systemen muss der aktuelle Zustand bekannt sein. Mit Hilfe von Sensoren müssen also die einzelnen Zustandskomponenten gemessen werden. Diese Messungen sind aber in der Regel verrauscht. Außerdem kommt es auch oft vor, dass bestimmte Zustandskomponenten gar nicht direkt gemessen werden können. Mit einem Beobachter ist es aber unter gewissen Voraussetzungen dennoch möglich, eine gute Schätzung für den aktuellen Zustand zu erhalten. Ein solcher Beobachter ist zum Beispiel das Kalman Filter, das ein optimaler Beobachter für lineare Systeme ist. Das Erweiterte Kalman Filter stellt eine Verallgemeinerung des Kalman Filters da und lässt sich auch auf nicht-lineare Systeme anwenden.

Das Kalman Filter und dessen nicht-lineare Version werden oft entweder durch ausschließlich kontinuierliche oder ausschließlich diskrete Systeme modelliert. Viele reale Systemdynamiken sind aber durch kontinuierliche Systeme gegeben, während die Messungen der Sensoren zu diskreten Zeitpunkten erfolgt. Es liegt zunächst also eine Mischung von kontinuierlichen und diskreten Systemen vor. Nun kann man zum Beispiel die Systemdynamik diskretisieren um ein einheitliches System zu erhalten. Eine andere Möglichkeit ist der Einsatz des Hybriden Erweiterten Kalman Filters (HEKF). Dieses wollen wir in diesem Kapitel herleiten. Wir beziehen uns dabei auf [20, K. 3.3, 5.1, 8.2, 13.1 und 13.2].

In diesem Kapitel bezeichne $w(t) \sim (0, Q)$ ein zeitkontinuierliches, weißes Rauschen mit Kovarianzmatrix Q. $v_k \sim (0, R_k)$ bezeiche ein zeitdiskretes, weißes Rauschen mit Kovarianzmatrix R_k. δ_{ij} bezeichne das Kronecker-Delta und $\delta(t)$ den Dirac-Impuls.
$w(t) \sim (0, Q)$ ist dabei nach [20, S. 231] äquivalent zu: $E[w(t)w^T(\tau)] = Q\delta(t - \tau)$.

5.1 Rekursive Methode der kleinsten Quadrate

In diesem Abschnitt wollen wir einen konstanten aber unbekannten Vektor $x \in \mathbb{R}^n$ mit Hilfe von verrauschten, linearen Messdaten $y_i \in \mathbb{R}^m$ in einem gewissen Sinn optimal schätzen. Mit $\hat{x}_{k-1} \in \mathbb{R}^n$ bezeichnen wir den optimalen Schätzwert von x, in dessen Berechnung die ersten

$k-1$ Messungen eingeflossen sind. Liegt nun ein neuer Messwert y_k vor, so soll damit $\hat{x}_{k-1}$ zu einer neuen Schätzung $\hat{x}_k$ überführt werden.

Definition 5.1 Sei $x \in \mathbb{R}^n$ konstanter, aber unbekannter Zufallsvektor. $y_k \in \mathbb{R}^m$ bezeichne eine Messung zum Zeitpunkt k. $H_k \in \mathbb{R}^{m \times n}$ und $K_k \in \mathbb{R}^{n \times m}$ sind Matrizen und v_k ein Zufallsvektor zum Zeitpunkt k.
Ein linear rekursiver Schätzer von x ist gegeben durch:

$$\begin{aligned} y_k &= H_k x + v_k \\ \hat{x}_k &= \hat{x}_{k-1} + K_k(y_k - H_k \hat{x}_{k-1}) \end{aligned}$$

mit einer gegebenen Startschätzung $\hat{x}_0$ von x.
$\epsilon_{x,k} := x - \hat{x}_k$ wird Schätzfehler zum Zeitpunkt $k \geq 0$ genannt.

Satz 5.2 Sei ein linear rekursiver Schätzer von x nach Definition 5.1 gegeben. Es verschwinde sowohl der Erwartungswert des Schätzfehlers $\epsilon_{x,k-1}$ zum Zeitpunkt $k-1$ als auch der Erwartungswert des Messrauschens v_k zum Zeitpunkt k. Dann gilt:

$$E(\epsilon_{x,k}) = 0$$

Beweis Nach Voraussetzung gilt $E(\epsilon_{x,k-1}) = 0$ und $E(v_k) = 0$. Damit folgt:

$$\begin{aligned} E(\epsilon_{x,k}) &= E(x - \hat{x}_k) \\ &= E[x - \hat{x}_{k-1} - K_k(y_k - H_k \hat{x}_{k-1})] \\ &= E[\epsilon_{x,k-1} - K_k(H_k x + v_k - H_k \hat{x}_{k-1})] \\ &= E[\epsilon_{x,k-1} - K_k H_k(x - \hat{x}_{k-1}) - K_k v_k] \\ &= (I - K_k H_k) E(\epsilon_{x,k-1}) - K_k E(v_k) \\ &= 0 \end{aligned}$$

Bemerkung 5.3 Satz 5.2 gilt unabhängig von der Wahl von H_k und K_k. Während H_k durch die jeweilige Anwendung vorgegeben ist, kann K_k frei gewählt werden. Insbesondere kann K_k also dazu verwendet werden, dass der linear rekusrive Schätzer weitere Optimalitätskriterien erfüllt. Der folgenden Satz gibt an, wie man K_k wählen muss, damit die Summe der Varianzen des Schätzfehlers zum Zeitpunkt k minimiert wird.

Satz 5.4 Sei ein linear rekursiver Schätzer von x nach Definition 5.1 gegeben. P_{k-1} bezeichne die Kovarianzmatrix des Schätzfehlers $\epsilon_{x,k-1}$ und R_k die Kovarianzmatrix von v_k. Dann gilt:

$$K_k := P_{k-1} H_k^T (H_k P_{k-1} H_k^T + R_k)^{-1}$$

minimiert die Kostenfunktion

$$J_k := E[(x_1 - \hat{x}_{1,k})^2] + \cdots + E[(x_n - \hat{x}_{n,k})^2]$$

wobei $\hat{x}_{i,k}$ die i-te Komponente von $\hat{x}_k$ bezeichnet.

Beweis Im Beweis von Satz 5.2 wird gezeigt, dass für den Schätzfehler $\epsilon_{x,k}$ die Gleichung:

$$\epsilon_{x,k} = (I - K_k H_k)\epsilon_{x,k-1} - K_k v_k$$

gilt. Für die Kovarianzmatrix P_k des Schätzfehlers $\epsilon_{x,k}$ gilt damit:

$$\begin{aligned}
P_k &= E(\epsilon_{x,k}\epsilon_{x,k}^T) \\
&= E\{[(I - K_k H_k)\epsilon_{x,k-1} - K_k v_k][(I - K_k H_k)\epsilon_{x,k-1} - K_k v_k]^T\} \\
&= E\{[(I - K_k H_k)\epsilon_{x,k-1} - K_k v_k][\epsilon_{x,k-1}^T(I - K_k H_k)^T - v_k^T K_k^T]\} \\
&= (I - K_k H_k)E(\epsilon_{x,k-1}\epsilon_{x,k-1}^T)(I - K_k H_k)^T - \\
&\quad (I - K_k H_k)E(\epsilon_{x,k-1}v_k^T)K_k^T - \\
&\quad K_k E(v_k\epsilon_{x,k-1}^T)(I - K_k H_k)^T + \\
&\quad K_k E(v_k v_k^T)K_k^T
\end{aligned}$$

Der Schätzfehler $\epsilon_{x,k-1}$ zum Zeitpunkt $k-1$ ist unabhängig vom Messrauschen v_k zum Zeitpunkt k. Somit gilt:

$$\begin{aligned}
E(\epsilon_{x,k-1}v_k^T) &= E(v_k\epsilon_{x,k-1}^T) \\
&= E(v_k)E(\epsilon_{x,k-1}) \\
&= 0
\end{aligned}$$

Für die Kovarianzmatrix P_k gilt also:

$$P_k = (I - K_k H_k)P_{k-1}(I - K_k H_k)^T + K_k R_k K_k^T$$

wobei $R_k = E(v_k v_k^T)$ die Kovarianzmatrix des Messrauschens zum Zeitpunkt k beschreibt.

Für die Kostenfunktion J_k gilt:

$$\begin{aligned}
J_k &= E[(x_1 - \hat{x}_{1,k})^2] + \cdots + E[(x_n - \hat{x}_{n,k})^2] \\
&= E(\epsilon_{x_1,k}^2 + \cdots + \epsilon_{x_n,k}^2) \\
&= E(\epsilon_{x,k}^T\epsilon_{x,k}) \\
&= E[Tr(\epsilon_{x,k}\epsilon_{x,k}^T)] \\
&= Tr[E(\epsilon_{x,k}\epsilon_{x,k}^T)] \\
&= Tr(P_k) \\
&= Tr[(I - K_k H_k)P_{k-1}(I - K_k H_k)^T + K_k R_k K_k^T]
\end{aligned}$$

Für die erste und zweite Ableitung von J_k bezüglich K_k gilt [20, S. 17]

$$\begin{aligned}
\frac{\partial J_k}{\partial K_k} &= 2(I - K_k H_k)P_{k-1}(-H_k^T) + 2K_k R_k \\
\frac{\partial^2 J_k}{\partial K_k \partial K_k} &= 2H_k P_{k-1} H_k^T + 2R_k
\end{aligned}$$

Mit K_k wie im Satz angegeben gilt also:

$$\frac{\partial J_k}{\partial K_k} = 0$$

und $\frac{\partial^2 J_k}{\partial K_k \partial K_k}$ positiv semidefinit. K_k ist also ein Minimierer der Kostenfunktion J_k.

Bemerkung 5.5 Nach [20, S. 88] gilt:

$$\begin{aligned} K_k &= P_{k-1}H_k^T(H_kP_{k-1}H_k^T + R_k)^{-1} \\ &= P_kH_k^TR_k^{-1} \\ P_k &= (I - K_kH_k)P_{k-1}(I - K_kH_k)^T + K_kR_kK_k^T \\ &= (P_{k-1}^{-1} + H_k^TR_k^{-1}H_k)^{-1} \\ &= (I - K_kH_k)P_{k-1} \end{aligned}$$

Im mathematischen Sinne sind dies also äquivalente Updateformeln für K_k bzw. P_k. Aus numerischer Sicht unterscheiden sie sich aber im Berechnungsaufwand und Robustheit. So ist $P_k = (I - K_kH_k)P_{k-1}$ zum Beispiel die einfachste Updateformel für P_k, aber es kann passieren, dass P_k nach der numerischen Berechnung nicht mehr positiv definit ist [20, S. 87]. Letztendlich hängt es von der Anwendung ab, welche Formeln man verwenden sollte.

5.2 Zeitdiskretes Kalman Filter

Mit der rekursiven Methode der kleinsten Quadrate kann man das Kalman Filter für lineare, zeitdiskrete Kontrollsysteme herleiten.

Gegeben sei folgendes lineare, zeitdiskrete Kontrollsystem:

$$\begin{aligned} x_k &= F_{k-1}x_{k-1} + G_{k-1}u_{k-1} + w_{k-1} \\ y_k &= H_kx_k + v_k \end{aligned}$$

mit $x_k \in \mathbb{R}^n$, $y_k \in \mathbb{R}^m$, F_k, G_k und H_k reelle Matrizen passender Dimension und Zufallsprozesse $\{w_k\}$ und $\{v_k\}$ so dass gilt:

$$\begin{aligned} w_k &\sim (0, Q_k) \\ v_k &\sim (0, R_k) \\ E(w_kw_j^T) &= Q_k\delta_{kj} \\ E(v_kv_j^T) &= R_k\delta_{kj} \\ E(v_kw_j^T) &= 0. \end{aligned}$$

w_k stellt Systemrauschen und v_k Messrauschen dar.

Ziel des Kalman Filters ist es, den Zustand des Systems zum Zeitpunkt k zu schätzen. Dabei führt man zwei Phasen aus. In der Prädiktionsphase wird der neue Zustand nur auf Grund der Systemdynamik geschätzt. Diesen Zustand bezeichnen wir mit $\hat{x}_k^- = E(x_k|y_1, \cdots, y_{k-1})$ und wird a priori Schätzung genannt. Er hängt nur von der Systemdynamik und den ersten $k-1$ Messwerten ab. In der Korrekturphase fließt nun auch der k-te Messwert ein und führt die a priori Schätzung in die a posteriori Schätzung $\hat{x}_k^+ = E(x_k|y_1, \cdots, y_k)$ über. Sowohl $\hat{x}_k^-$ als auch $\hat{x}_k^+$ stellen also eine Schätzung für den Zustand x_k zum Zeitpunkt k dar.

Die Kovarianzmatrizen der Schätzfehler von $\hat{x}_k^-$ und $\hat{x}_k^+$ bezeichnen wir mit $P_k^- = E[(x_k - \hat{x}_k^-)(x_k - \hat{x}_k^-)^T]$ bzw. $P_k^+ = E[(x_k - \hat{x}_k^+)(x_k - \hat{x}_k^+)^T]$.

Zu Beginn wird die a posteriori Schätzung $\hat{x}_0^+$ und die zugehörige Kovarianzmatrix P_0^+ zum Zeitpunkt $k = 0$ vom Anwender initialisiert. In jedem folgenden Zeitschritt $k > 0$ wird dann die Prädiktionsphase gefolgt von der Korrekturphase durchgeführt. Die beiden Phasen lassen sich wie folgt beschreiben:

In der Prädiktionsphase wird angenommen, dass $\hat{x}_{k-1}^+$ und P_{k-1}^+ bekannt sind. Betrachtet man nun das Kontrollsystem und bezieht keine neuen Messdaten mit ein, so gilt für den Erwartungswert von x_k zum Zeitpunkt k:

$$\begin{aligned}\hat{x}_k^- &= E(x_k) \\ &= E(F_{k-1}x_{k-1} + G_{k-1}u_{k-1}) \\ &= F_{k-1}E(x_{k-1}) + G_{k-1}u_{k-1} \\ &= F_{k-1}\hat{x}_{k-1}^+ + G_{k-1}u_{k-1}\end{aligned}$$

Damit gilt für die Kovarianzmatrix P_k^-:

$$\begin{aligned}P_k^- &= E[(x_k - \hat{x}_k^-)(x_k - \hat{x}_k^-)^T] \\ &= E\{[F_{k-1}(x_{k-1} - \hat{x}_{k-1}^+) + w_{k-1}][F_{k-1}(x_{k-1} - \hat{x}_{k-1}^+) + w_{k-1}]^T\} \\ &= F_{k-1}E[(x_{k-1} - \hat{x}_{k-1}^+)(x_{k-1} - \hat{x}_{k-1}^+)^T]F_{k-1}^T + \\ &\quad F_{k-1}E[(x_{k-1} - \hat{x}_{k-1}^+)w_{k-1}^T] + \\ &\quad E[w_{k-1}(x_{k-1} - \hat{x}_{k-1}^+)^T]F_{k-1}^T + \\ &\quad E(w_{k-1}w_{k-1}^T) \\ &= F_{k-1}P_{k-1}^+F_{k-1}^T + Q_{k-1}\end{aligned}$$

Nun folgt die Korrekturphase. $\hat{x}_k^{-1}$ und P_k^- sind also bekannt. Außerdem liegt ein Messwert y_k zum Zeitpunkt k vor. Mit dem linear rekursiven Schätzer aus Abschnitt 5.1 (mit angepasster Notation) kann nun $\hat{x}_k^+$ und P_k^+ bestimmt werden:

$$K_k = P_k^- H_k^T (H_k P_k^{-1} H_k^T + R_k)^{-1} \tag{5.1}$$

$$\hat{x}_k^+ = \hat{x}_k^- + K_k(y_k - H_k\hat{x}_k^-) \tag{5.2}$$

$$P_k^+ = (I - K_k H_k)P_k^-(I - K_k H_k)^T + K_k R_k K_k^T \tag{5.3}$$

Eine Voraussetzung für den linear rekursiven Schätzer ist, dass ein konstanter Zustandsvektor geschätzt wird. Dies ist hier der Fall, da sowohl $\hat{x}_k^-$ als auch $\hat{x}_k^+$ den Zustand x_k zum Zeitpunkt k schätzen.

Im Algorithmus 5 ist das zeitdiskrete Kalman Filter noch mal zusammengefasst. In den Zeilen 2 und 3 wird das Kalman Filter initialisiert. Die Zeilen 5 und 6 entsprechen der Prädiktionsphase, während die Zeilen 8 und 9 die Korrekturphase darstellen.

Algorithmus 5 zeitdiskretes Kalman Filter

1: **procedure** DKF()
2: $\hat{x}_0^+ \leftarrow E(x_0)$
3: $P_0^+ \leftarrow E[(x_0 - \hat{x}_0^+)(x_0 - \hat{x}_0^+)^T]$
4: **for** $k \leftarrow 1, 2, \cdots$ **do**
5: $P_k^- \leftarrow F_{k-1} P_{k-1}^+ F_{k-1}^T + Q_{k-1}$
6: $\hat{x}_k^- \leftarrow F_{k-1} \hat{x}_{k-1}^+ + G_{k-1} u_{k-1}$
7: $K_k \leftarrow P_k^- H_k^T (H_k P_k^- H_k^T + R_k)^{-1}$
8: $\hat{x}_k^+ \leftarrow \hat{x}_k^- + K_k (y_k - H_k \hat{x}_k^-)$
9: $P_k^+ \leftarrow (I - K_k H_k) P_k^- (I - K_k H_k)^T + K_k R_k K_k^T$

5.3 Zeitkontinuierliches Kalman Filter

In diesem Abschnitt leiten wir die zeitkontinuierliche Version des Kalman Filters her. Wir betrachten also ein System gegeben durch:

$$\begin{aligned} \dot{x}(t) &= Ax(t) + Bu(t) + w(t) \\ y(t) &= Cx(t) + v(t) \\ w(t) &\sim (0, Q) \\ v(t) &\sim (0, R). \end{aligned}$$

Diskretisieren wir dieses System mit dem Euler-Verfahren und einer Schrittweite T, so gilt [20, S. 234]

$$\begin{aligned} x_k &= (I + AT)x_{k-1} + BTu_{k-1} + w_k \\ y_k &= Cx_k + v_k \\ w_k &\sim (0, QT) \\ v_k &\sim (0, R/T) \end{aligned}$$

Das System liegt damit also gerade in der Form wie in Abschnitt 5.2 vor. Für die Kovarianzmatrizen P_k^- und P_k^+ gilt somit (unter Beachtung der alternativen Updateformeln aus Bemerkung 5.5):

$$
\begin{aligned}
P_k^- &= (I + AT)P_{k-1}^+(I + AT)^T + QT \\
K_k &= P_k^- C^T (CP_k^- C^T + R/T)^{-1} \\
P_k^+ &= (I - K_k C)P_k^-
\end{aligned}
$$

Damit gilt:

$$
\begin{aligned}
\lim_{T\to 0} \frac{1}{T} K_k &= \lim_{T\to 0} P_k^- C^T (CP_k^- C^T T + R)^{-1} \\
&= P_k^- C^T R^{-1}
\end{aligned}
$$

Insbesondere gilt also auch $\lim_{T\to 0} K_k = 0$.
Es gilt:

$$
\begin{aligned}
\frac{1}{T}(P_{k+1}^- - P_k^-) &= \frac{1}{T}[(I + AT)P_k^+(I + AT)^T + QT - P_k^-] \\
&= \frac{1}{T}[(I + AT)(I - K_k C)P_k^-(I + AT)^T + QT - P_k^-] \\
&= \frac{1}{T}[(I - K_k C + AT - AK_k CT)(P_k^- + P_k^- A^T T) + QT - P_k^-] \\
&= \frac{1}{T}[P_k^- - K_k CP_k^- + AP_k^- T - AK_k CP_k^- T \\
&\qquad + P_k^- A^T T - K_k CP_k^- A^T T + AP_k^- A^T T^2 - AK_k CP_k^- A^T T^2 \\
&\qquad + QT - P_k^-] \\
&= -\frac{1}{T} K_k CP_k^- + AP_k^- - AK_k CP_k^- + P_k^- A^T - K_k CP_k^- A^T + Q \\
&\quad + AP_k^- A^T T - AK_k CP_k^- A^T T
\end{aligned}
$$

Es folgt:

$$
\lim_{T\to 0} \frac{1}{T}(P_{k+1}^- - P_k^-) = P_k^- C^T R^{-1} CP_k^- + AP_k^- + P_k^- A^T + Q
$$

Für $T \to 0$ gilt $P(kT) = P_k^-$ und damit:

$$
\dot{P}(t) = AP(t) + P(t)A^T + Q - P(t)C^T R^{-1} CP_k^-
$$

Der Zustand des diskretisierten Systems wird entsprechend Gleichung (5.2) aktualisiert:

$$
\hat{x} = (I + AT)\hat{x}_k + BTu_k + K_{k+1}[y_{k+1} - H(I + AT)\hat{x}_k - HBTu_k]
$$

Mit $\hat{x}(kT) = \hat{x}_k$ für $T \to 0$ gilt also:

$$\begin{aligned}\dot{\hat{x}}(t) &= \lim_{T\to 0} \frac{\hat{x}_{k+1} - \hat{x}_k}{T} \\ &= A\hat{x}_k + Bu_k + \frac{K_{k+1}}{T}[z_{k+1} - H\hat{x}_k - H(A\hat{x}_k + Bu_k)T] \\ &= A\hat{x}(t) + Bu(t) + P(t)H^T R^{-1}[y(t) - H\hat{x}(t)]\end{aligned}$$

Definiert man die Kalmanmatrix K wie folgt:

$$K(kT) = \frac{K_k}{T}$$

so gilt für $T \to 0$:

$$K(t) = P(t)H^T R^{-1}$$

und damit:

$$\dot{\hat{x}}(t) = A\hat{x}(t) + Bu(t) + K(t)[y(t) - H\hat{x}(t)]$$

5.4 Linearisiertes Kalman Filter

In diesem Abschnitt wollen wir das Kalman Filter auf eine zeitkontinuierliche, nicht-lineare Systemdynamik und Ausgangsfunktion anwenden. Es ist also folgendes System gegeben:

$$\begin{aligned}\dot{x}(t) &= f[t, x(t), u(t), w(t)] \\ y(t) &= h[x(t), v(t)] \\ w(t) &\sim (0, Q) \\ v(t) &\sim (0, R).\end{aligned}$$

Zusätzlich führen wir Referenztrajektorien ein, die durch die nominelle Systemdynamik bzw. des nominellen Ausgangs gegeben sind:

$$\begin{aligned}\dot{x}_{ref}(t) &= f[t, x_{ref}(t), u(t), 0] \\ y_{ref}(t) &= h[x_{ref}(t), 0]\end{aligned}$$

Die Änderung des Fehlers zwischen dem realen und nominellen System approximieren wir

nun durch Linearisierung:

$$\begin{aligned}
\Delta x(t) &= x(t) - x_{ref}(t) \\
\Delta \dot{x}(t) &= \dot{x}(t) - \dot{x}_{ref}(t) \\
&= f[t, x(t), u(t), w(t)] - f[t, x_{ref}(t), u(t), 0] \\
&\approx f[t, x_{ref}(t), u(t), 0] + \left.\frac{\partial f(\bar{t}, \bar{x}, \bar{u}, \bar{w})}{\partial \bar{t}}\right|_{(\cdot)_{ref}} (t - t) + \\
&\quad \left.\frac{\partial f(\bar{t}, \bar{x}, \bar{u}, \bar{w})}{\partial \bar{x}}\right|_{(\cdot)_{ref}} [x(t) - x_{ref}(t)] + \left.\frac{\partial f(\bar{t}, \bar{x}, \bar{u}, \bar{w})}{\partial \bar{u}}\right|_{(\cdot)_{ref}} [u(t) - u(t)] + \\
&\quad \left.\frac{\partial f(\bar{t}, \bar{x}, \bar{u}, \bar{w})}{\partial \bar{w}}\right|_{(\cdot)_{ref}} [w(t) - 0] - f[t, x_{ref}(t), u(t), 0] \\
&=: \tilde{A}(t)\Delta x(t) + \tilde{L}(t)w(t)
\end{aligned}$$

wobei $(\cdot)_{ref}$ bedeutet, dass die partiellen Ableitungen im Punkt $(\bar{t}, \bar{x}, \bar{u}, \bar{w}) = (t, x_{ref}(t), u(t), 0)$ ausgewerten werden.

Für den Fehler zwischen dem realen und nominellen Ausgang gilt:

$$\begin{aligned}
\Delta y(t) &= y(t) - y_{ref}(t) \\
&= h[x(t), v(t)] - h[x_{ref}(t), 0] \\
&\approx h[x_{ref}(t), 0] + \left.\frac{\partial h(\bar{x}, \bar{v})}{\partial \bar{x}}\right|_{(\cdot)_{ref}} [x(t) - x_{ref}(t)] + \\
&\quad \left.\frac{\partial h(\bar{x}, \bar{v})}{\partial \bar{v}}\right|_{(\cdot)_{ref}} [v(t) - 0] - h[x_{ref}(t), 0] \\
&=: \tilde{C}(t)\Delta x(t) + \tilde{M}(t)v(t)
\end{aligned}$$

wobei hier $(\cdot)_{ref}$ für $(\bar{x}, \bar{v}) = (x_{ref}(t), 0)$ steht.

Für ein keines $\delta t > 0$ können $\tilde{A}(t)$, $\tilde{L}(t)$, $\tilde{C}(t)$ und $\tilde{M}(t)$ für alle $t \in [t_0, t_0 + \delta t]$ wie folgt approximiert werden:

$$\begin{aligned}
\tilde{A}(t) &\approx \tilde{A}(t_0) =: A \\
\tilde{L}(t) &\approx \tilde{L}(t_0) =: L \\
\tilde{C}(t) &\approx \tilde{C}(t_0) =: C \\
\tilde{M}(t) &\approx \tilde{M}(t_0) =: M.
\end{aligned}$$

Damit folgt [20, S. 398]:

$$\begin{aligned}
\Delta \dot{x}(t) &= A\Delta x(t) + Lw(t) \\
\Delta y(t) &= C\Delta x(t) + Mv(t) \\
Lw(t) &\sim (0, LQL^T) \\
Mv(t) &\sim (0, MRM^T).
\end{aligned}$$

Auf dieses System kann man also das zeitkontinuierliche (lineare) Kalman Filter aus Abschnitt 5.3 anwenden, wenn man $\Delta x(t)$ als Zustand und $\Delta y(t)$ als Messung interpretiert. Mit $\Delta\hat{x}(t) = \hat{x}(t) - x_{ref}(t)$ können wir also für $t \in [t_0, t_0 + \delta t]$ das linearisierte Kalman Filter für das ursprüngliche System wie folgt angeben:

$$\begin{aligned}
\Delta\hat{x}(t_0) &= 0 \\
P(t_0) &= E\{[\Delta x(t_0) - \Delta\hat{x}(t_0)][\Delta x(t_0) - \Delta\hat{x}(t_0)]^T\} \\
\Delta\dot{\hat{x}}(t) &= A\Delta\hat{x}(t) + K[\Delta y(t) - C\Delta\hat{x}(t)] \\
K &= P(t)C^T(MRM^T)^{-1} \\
\dot{P}(t) &= AP(t) + P(t)A^T + LQL^T - P(t)C^T(MRM^T)^{-1}CP(t) \\
\hat{x}(t) &= x_{ref}(t) + \Delta x(t).
\end{aligned}$$

Bemerkung 5.6 Durch die Linearisierung und den Abschätzungen um die Matrizen A, L, C und M zu bestimmen, wird P in Wirklichkeit nicht mehr exakt der Kovarianzmatrix des Schätzfehlers entsprechen [20, S. 399].

5.5 Hybrides Erweitertes Kalman Filter

Mit diesen Ergebnissen können wir das Hybride Erweiterte Kalman Filter herleiten, das für zeitkontinuierliche (nicht-lineare) Systemdynamiken und zeitdiskreten (nicht-lineare) Messungen gedacht ist. Wir betrachten also ein System gegeben durch:

$$\begin{aligned}
\dot{x}(t) &= f[t, x(t), u(t), w(t)] \\
y_k &= h_k[x(t_k), v_k] \\
w(t) &\sim (0, Q) \\
v_k &\sim (0, R_k)
\end{aligned}$$

Zwischen zwei Messzeitpunkten kann der geschätzte Zustand $\hat{x}$ mit der nominellen Systemdynamik vorausgesagt werden:

$$\dot{\hat{x}}(t) = f[t, \hat{x}(t), u(t), 0].$$

Im vorherigen Abschnitt haben wir gesehen, dass sich die Kovarianzmatrix des Fehlerschätzers bei einem kontinuierlichen Ausgang wie folgt ergibt:

$$\dot{P}(t) = AP(t) + P(t)A^T + LQL^T - P(t)C^T(MRM^T)^{-1}CP(t).$$

In unserem Fall haben wir aber nur zu diskreten Zeitpunkten Zugang zu Messungen. Insbesondere liegen zwischen zwei Messpunkten auch keine Messungen vor und man sollte beim Integrieren von P den Summanden, der R enthält, ignorieren. [20, S. 404] rechtfertigt das damit, dass die Kovaranzmatrix R in diesem Fall unbegrenzt ist und R^{-1} somit gegen 0

strebt.

Somit erhalten wir für den Prädiktionsschritt um $\hat{x}$ von $\hat{x}_{k-1}^+$ auf $\hat{x}_k^-$ und P von P_{k-1}^+ auf P_k^- zu überführen:

$$\begin{aligned}
\dot{\hat{x}}(t) &= f[t, \hat{x}(t), u(t), 0] \\
\dot{P}(t) &= AP(t) + P(t)A^T + LQL^T \\
&\text{mit} \\
A &= \left.\frac{\partial f}{\partial x}\right|_{(t_{k-1}, \hat{x}_{k-1}^+, u_{k-1}, 0)} \\
L &= \left.\frac{\partial f}{\partial w}\right|_{(t_{k-1}, \hat{x}_{k-1}^+, u_{k-1}, 0)}
\end{aligned}$$

Liegt eine neue Messung vor, so können wir den Zustand und die Kovrianzmatrix wie im zeitdiskreten (linearen) Kalman Filter aktualisieren:

$$\begin{aligned}
K_k &= P_k^- H_k^T (H_k P_k^{-1} H_k^T + M_k R_k M_k^T)^{-1} \\
\hat{x}_k^+ &= \hat{x}_k^- + K_k[y_k - h_k(\hat{x}_k^-, 0)] \\
P_k^+ &= (I - K_k H_k) P_k^- (I - K_k H_k)^T + K_k M_k R_k M_k^T K_k^T \\
&\text{mit} \\
H_k &= \left.\frac{\partial h}{\partial x}\right|_{(\hat{x}_k^-, 0)} \\
M_k &= \left.\frac{\partial h}{\partial v}\right|_{(\hat{x}_k^-, 0)}.
\end{aligned}$$

Wie auch im linearen Fall können K_k und P_k^+ durch die äquivalenten Formulierungen nach Bemerkung 5.5 ersetzt werden.

In Algorithmus 6 ist das Hybride Erweiterte Kalman Filter noch ein mal zusammengefasst. Zum Lösen der Differentialgleichungen wurde in dieser Arbeit das Runge Kutta 5(4)-Verfahren, wie es in [10, S. 56] beschrieben ist, verwendet.

Algorithmus 6 Hybrides Erweitertes Kalman Filter

1: **procedure** HEKF()
2: $\hat{x}_0^+ \leftarrow E(x_0)$
3: $P_0^+ \leftarrow E[(x_0 - \hat{x}_0^+)(x_0 - \hat{x}_0^+)^T]$
4: **for** $k \leftarrow 1, 2, \cdots$ **do**
5: Solve the initial value problem given by:

$$\dot{\hat{x}}(t) = f(t, \hat{x}(t), u(t), 0)$$
$$\dot{P}(t) = AP(t) + P(t)A^T + LQL^T$$

and initial value:

$$\hat{x}(t_{k-1}) = \hat{x}_{k-1}^+$$
$$P(t_{k-1}) = P_{k-1}^+$$

where

$$A = \left.\frac{\partial f}{\partial x}\right|_{(t_{k-1}, \hat{x}_{k-1}^+, u_{k-1}, 0)}$$
$$L = \left.\frac{\partial f}{\partial w}\right|_{(t_{k-1}, \hat{x}_{k-1}^+, u_{k-1}, 0)}$$

6: $\hat{x}_k^- \leftarrow \hat{x}(t_k)$
7: $P_k^- \leftarrow P(t_k)$
8: $H_k \leftarrow \frac{\partial h}{\partial x}|_{(\hat{x}_k^-, 0)}$
9: $M_k \leftarrow \frac{\partial h}{\partial v}|_{(\hat{x}_k^-, 0)}$
10: $K_k \leftarrow P_k^- H_k^T (H_k P_k^- H_k^T + M_k R_k M_k^T)^{-1}$
11: $P_k^+ \leftarrow (I - K_k H_k) P_k^- (I - K_k H_k)^T + K_k M_k R_k M_k^T K_k^T$

Kapitel 6

Realisierung der Positionsregelung

6.1 Modell des Crazyflies

In [14] wird bereits ein mathematisches Modell inklusive den Modellkonstanten für den Crazyflie vorgestellt. Der 12-dimensionale Systemzustand $x = \begin{pmatrix}\xi^T & v^T & \eta^T & \nu^T\end{pmatrix}^T \in \mathbb{R}^{12}$ des Quadcopters ist dabei gegeben durch die Position $\xi \in \mathbb{R}^3$, die Geschwindigkeit $v \in \mathbb{R}^3$, den Euler-Winkeln $\eta \in \mathbb{R}^3$ in der zyx-Konvention und den Winkelgeschwindigkeiten $\nu \in \mathbb{R}^3$. ξ und v werden dabei im Inertialsystem angegeben, während sich ν auf das Hauptachsensystem bezieht. Mit der Kontrolle $u \in [0, 60000]^4$ kann man Einfluss auf die vier Rotoren nehmen. Das Kontrollsystem ist schließlich gegeben durch:

$$\dot{x} = f(x,u) = \begin{pmatrix} v \\ \frac{1}{m}(G + R_\eta T) \\ W_\eta^{-1}\nu \\ I^{-1}(-\nu \times I \cdot \nu - \Gamma + \tau) \end{pmatrix} \tag{6.1}$$

mit

- der Gewichtskraft $G = mg \begin{pmatrix} 0 \\ 0 \\ -1 \end{pmatrix}$
- der Rotationsmatrix $R_\eta = \begin{pmatrix} C_\psi C_\theta & C_\psi S_\theta S_\phi - S_\psi C_\theta & C_\psi S_\theta C_\phi + S_\psi S_\phi \\ S_\psi C_\theta & S_\psi S_\theta S_\phi + C_\theta C_\phi & S_\psi S_\theta C_\phi - C_\psi S_\phi \\ -S_\theta & C_\theta S_\phi & C_\theta C_\phi \end{pmatrix}$
  wobei $S_\alpha := \sin(\alpha)$ und $C_\alpha := \cos(\alpha)$
- der Schubkraft $T = \begin{pmatrix} 0 \\ 0 \\ b_c(u_1^2 + u_2^2 + u_3^2 + u_4^2) \end{pmatrix}$

- der Rotationsmatrix $W_\eta^{-1} = \begin{pmatrix} 1 & \sin(\phi)\tan(\theta) & \cos(\phi)\tan(\theta) \\ 0 & \cos(\phi) & -\sin(\phi) \\ 0 & \frac{\sin(\phi)}{\cos(\theta)} & \frac{\cos(\phi)}{\cos(\theta)} \end{pmatrix}$

- dem Moment aufgrund der Zentripetalkräfte $\nu \times I \cdot \nu = \begin{pmatrix} (I_z - I_y)qr \\ (I_x - I_z)pr \\ (I_y - I_x)pq \end{pmatrix}$

- dem Kreiselmoment $\Gamma = I_{r,c}(u_1 - u_2 + u_3 - u_4) \begin{pmatrix} q \\ -p \\ 0 \end{pmatrix}$

- den externen Drehmomenten $\tau = \begin{pmatrix} 0 & -db_c & 0 & db_c \\ -db_c & 0 & db_c & 0 \\ -k_c & k_c & -k_c & k_c \end{pmatrix} \begin{pmatrix} u_1^2 \\ u_2^2 \\ u_3^2 \\ u_4^2 \end{pmatrix}$

- und den Modellkonstanten $g, m, I_{r,c}, d, b_c, k_c$ sowie $I = \begin{pmatrix} I_x & 0 & 0 \\ 0 & I_y & 0 \\ 0 & 0 & I_z \end{pmatrix}$

Üblicherweise werden die Kontrollen eines Quadcopters über die Winkelgeschwindigkeiten ω_i der vier Rotoren modelliert. Beim Crzayflie kann man die Winkelgeschwindigkeit aber nicht direkt vorgeben. Stattdessen kann man nur einen Wert u_i im Intervall $[0, 60000]$ angeben. Dieser Wert bestimmt dann, wie groß die am jeweiligen Motor anliegende Spannung sein soll und resultiert somit in unterschiedliche Winkelgeschwindigkeiten des entsprechenden Rotors. In [14] wird für jeden Rotor der gleiche, lineare Zusammenhang zwischen der Winkelgeschwindigkeit und dem eigentlichen Kontrollwert angenommen: $\omega_i = cu_i$ für $i = 1, \cdots, 4$. Da c aber nicht direkt bestimmt werden konnte, musste c als Produkt mit anderen Modellkonstanten modelliert werden. Der Index c bei einigen der oben auftauchenden Modellkonstanten signalisiert diese Abhängigkeit.

Tabelle 6.1 können die Einheiten und die Bedeutung der einzelnen Modellkonstanten entnommen werden.

Konstante	**Einheit**	**Bedeutung**
g	$\mathrm{m{\cdot}s^{-2}}$	Erdbeschleunigung
m	kg	Masse des Quadrocopters
d	m	Abstand Rotor - Masseschwerpunkt
I_x, I_y, I_z	$\mathrm{kg{\cdot}m^2}$	Trägheitsmomente des Quadrocopters
I_r	$\mathrm{kg{\cdot}m^2}$	Trägheitsmoment der Rotoren
b	kg·m	Konstante, die vom Auftriebskoeffizienten des Quadrocopters abhängt
k	$\mathrm{kg{\cdot}m^2}$	Konstante, die vom Strömungswiderstandskoeffizienten der Rotoren abhängt
c	-	Skalierungsfaktor
$I_{r,c} := I_r c$	$\mathrm{kg{\cdot}m^2}$	-
$b_c := bc^2$	kg·m	-
$k_c := kc^2$	$\mathrm{kg{\cdot}m^2}$	-

Tabelle 6.1: Modellkonstanten und ihre Bedeutung

Die Euler-Winkel η und die Winkelgeschwindigkeiten ν lassen sich direkt auf dem Crazyflie messen. Mit den in Abschnitt 3.2 vorgestellten Trackingverfahren und einer kalibrierten Stereo-Kamera kann die Position ξ des Crazyflies bestimmt werden. Die Geschwindigkeit v können wir aber nicht direkt feststellen. Stattdessen werden wir das Hybride Erweitere Kalman Filter einsetzen, um den vollen Systemzustand zu schätzen.

In dieser Anwendung ist aber zu beachten, dass die Euler-Winkel und die Winkelgeschwindigkeiten zwar in etwa alle zwei Millisekunden gemessen werden können, aber neue Positionsdaten liegen höchstens alle 1000/30 Millisekunden vor. Das liegt daran, dass die verwendeten Kameras nur 30 Bilder pro Sekunde liefern können. In der Praxis werden die Positionsdaten aber noch später vorliegen, da das Bildmaterial auch erst noch ausgewertet werden muss. Dies hat aber zur Folge, dass die Messdaten in nicht geordneter Reihenfolge eintreffen können. Abbildung 6.1 verdeutlicht diesen Vorgang. Die durchgezogenen Pfeile zwischen den beiden Zeitleisten entsprechen den Messungen der Euler-Winkel und Winkelgeschwindigkeiten. Hier beseteht ein Zeitversatz zwischen Messzeitpunkt und Zeitpunkt der Verfügbarkeit, weil die Daten erst per Funk zu dem Rechner geschickt werden müssen. In den Praxistests hat sich ergeben, dass dieser Datentransfer etwa zwei bis drei Millisekunden in Anspruch nimmt. Der gestrichelte Pfeil entspricht einer Positionsmessung. In diesem Fall benötigt die Bildverarbeitung acht Millisekunden. In der Praxis hat sich gezeigt, dass die Positionsbestimmung inklusive der notwendigen Bildverarbeitung für beide Kamerabilder auf dem Testrechner etwa 9 Millisekunden dauert (unter Einsatz des Trackingverfahrens 3.2.2).

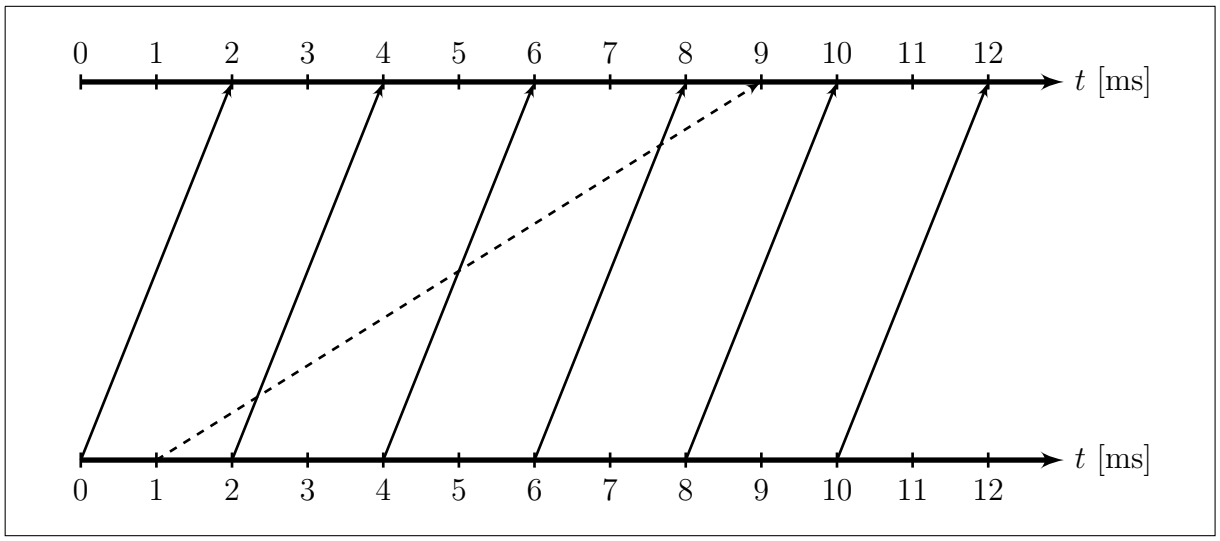

Abbildung 6.1: Ein Szenario von Messdaten, die nicht in geordneter Reihenfolge eintreffen. Die untere Zeitleiste gibt an, auf welchen Zeitpunkt sich eine Messung bezieht. Die obere Zeitleiste gibt an, zu welchem Zeitpunkt das Messergebnis dem Nutzer zur Verfügung steht.

Das in Kapitel 5 vorgestellte Kalman Filter geht aber davon aus, dass die Messdaten in geordneter Reihenfolge eintreffen. Würden wir nur die Positionsmessung betrachten, oder nur die Messung der Euler-Winkel und Winkelgeschwindigkeiten, so wäre diese Voraussetzung erfüllt. In unserer Anwendung können wir die Messungen separat betrachten, wenn wir das System (6.1) in zwei Teilsysteme aufteilen. Das eine System enthält dabei die Dynamik der Position und der Geschwindigkeit:

$$\dot{x} = f^{\psi}_{\xi,v}(x,u) = \begin{pmatrix} v \\ \frac{1}{m}(G + R_\eta T_{\xi,v}) \end{pmatrix} \tag{6.2}$$

mit der Position und Geschwindigkeit als Zustand $x = \begin{pmatrix} \xi^T & v^T \end{pmatrix}^T$, einer drei-dimensionalen Kontrolle $\tilde{u}$, sowie:

$$\eta = \begin{pmatrix} \tilde{u}_1 \\ \tilde{u}_2 \\ \psi \end{pmatrix}$$

$$T_{\xi,v} = \begin{pmatrix} 0 \\ 0 \\ b_c \tilde{u}_3 \end{pmatrix}$$

In diesem System dienen also die beiden ersten Euler-Winkel ϕ und θ als Kontrollwerte. Außerdem gibt es einen dritten Kontrollwert, der die Rolle von $u_1^2 + u_2^2 + u_3^2 + u_4^2$ aus Gleichung (6.1) übernimmt. Der Yaw-Winkel ψ kann bei der Positionsregelung vom Anwender

vorgegeben werden. Da dieser Winkel aber nicht Bestandteil des Systemzustandes in (6.2) ist, wird er als zusätzlicher Parameter interpretiert.

Das zweite System modelliert den Zustand $x = \begin{pmatrix}\eta^T & \nu^T\end{pmatrix}^T$ der Euler-Winkel und Winkelgeschwindigkeiten. Die Kontrolle u entspricht der Kontrolle aus System (6.1):

$$\dot{x} = f_{\eta,\nu}(x,u) = \begin{pmatrix} W_\eta^{-1}\nu \\ I^{-1}(-\nu \times I \cdot \nu - \Gamma + \tau) \end{pmatrix} \tag{6.3}$$

Indem wir für die beiden neuen Teilsysteme jeweils einen Regler entwerfen, können wir für das ursprüngliche System (6.1) einen Kaskadenregler angeben, wie es in Abbildung 6.2 illustriert ist.

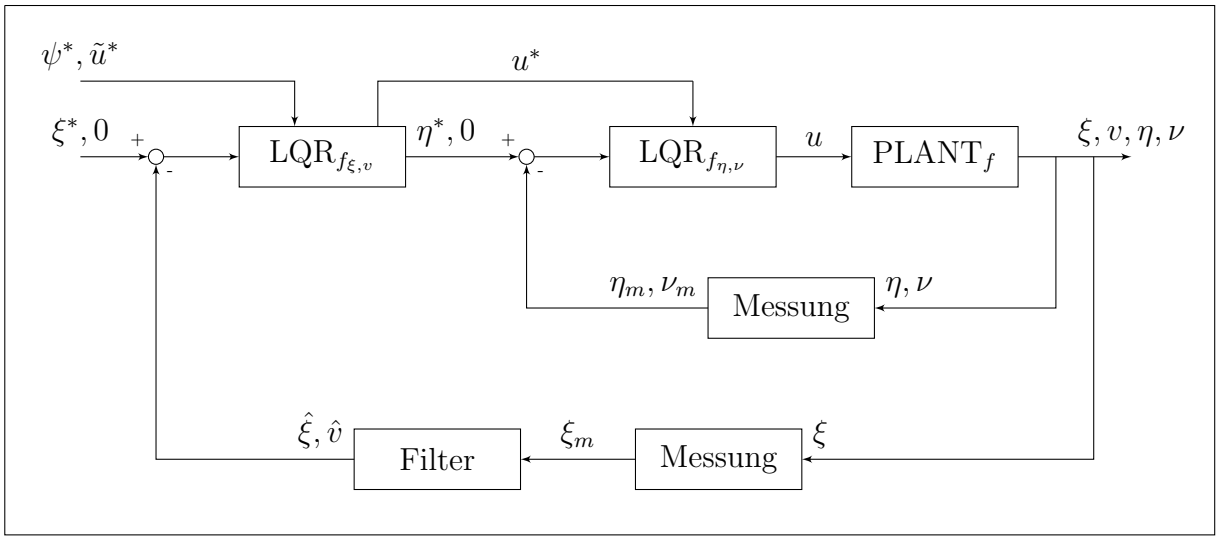

Abbildung 6.2: Kaskadenregelung der Positionsregelung

Das äußere System ist durch Gleichung (6.2) gegeben. Die aktuelle Position ξ wird von der Stereo-Kamera gemessen und liefert einen Messwert ξ_m. Mit Hilfe eines HEKF wird der komplette Systemzustand bestehend aus der Position $\hat{\xi}$ und der Geschwindigkeit $\hat{v}$ geschätzt. Dieser wird neben dem gewünschten Systemzustand, dem gewünschtem Yaw-Winkel ψ^* und der Sollkontrolle $\tilde{u}^*$ dem äußeren Regler übergeben. Aus den resultierenden Kontrollwert $\tilde{u}$ wird der neue Input für den inneren Regelkreis, der für das System (6.3) entworfen ist, konstruiert. Die gewünschten Euler-Winkel werden aus den ersten beiden Komponenten der Kontrolle $\tilde{u}$ und dem dem Yaw-Winkel ψ^* gebildet: $\eta^* := (\tilde{u}_1, \tilde{u}_2, \psi^*)^T$. Mit der dritten Komponente von $\tilde{u}$ wird der Sollschub u^* berechnet: $u^* := 0.5 \cdot \sqrt{\tilde{u}_3} \cdot (1,1,1,1)^T$. Zusätzlich werden die Messdaten η_m und ν_m des aktuellen Systemzustandes (der inneren Dynamik) gegeben durch η und ν an den inneren Regler übergeben. Die resultierende Kontrolle wird schließlich

auf dem Crazyflie angewendet. Entsprechend der Dynamik (6.1) ändert sich der Systemzustand durch diese Kontrollen, werden aber auch durch äußere Störgrößen beeinflusst. Dies wird durch den Block $PLANT_f$ symbolisiert.

6.2 Modellkonstanten

Wir wollen also einen LQ-Regler sowohl für das System (6.2) als auch für das System (6.3) entwerfen. Die Modellkonstanten können wir dabei [14] entnehmen und sind gegeben durch:

Konstante	**Wert**
g	9.81
m	0.02
d	0.045
I_x	$6.702 \cdot 10^{-4}$
I_y	$3.383 \cdot 10^{-4}$
I_z	$7.689 \cdot 10^{-4}$
$I_{r,c}$	$2.893 \cdot 10^{-6}$
b_c	$2.524 \cdot 10^{-11}$
k_c	$1.103 \cdot 10^{-12}$

Tabelle 6.2: Wert der Modellkonstanten, wie sie in [14] bestimmt worden sind.

Die ersten Entwürfe eines LQ-Reglers waren schon vielversprechend, haben jedoch eines gezeigt: Je nach Ladezustand der Batterie des Crazyflies wirken sich die berechneten Kontrollwerte unterschiedlich stark auf die resultierenden Winkelgeschwindigkeiten der Rotoren und damit auf die Auftriebskraft aus. In Abbildung 6.3 ist die z-Position eines etwa drei Minuten langen Fluges angetragen. Der Crazyflie sollte einen Meter über dem Boden fliegen, verlor mit der Zeit aber deutlich an Höhe.

Bei der Positionsregelung muss also auf den aktuellen Batteriestand eingegangen werden. Dies ist den Herstellern des Crazyflies offensichtlich auch aufgefallen, da sie inzwischen eine Funktionalität zur Kompensation des Batteriestandes implementiert haben [3, Z. 219-231]. Diese Funktionalität mit den entsprechenden Parametern wurde zwar für den Crazyflie 2.0 entworfen, wird von uns aber dennoch unverändert übernommen. Es hat sich herausgestellt, dass sich damit der Crazyflie 1.0 auch über einen längeren Zeitraum auf einer konstanten Höhe halten lässt (vgl. Abbildung 6.4).

Da bei der Parametrisierung in [14] der Batteriestand nicht beachtet wurde, ist davon auszugehen, dass die gefundenen Parameter nicht unbedingt den tatsächlichen Werten ent-

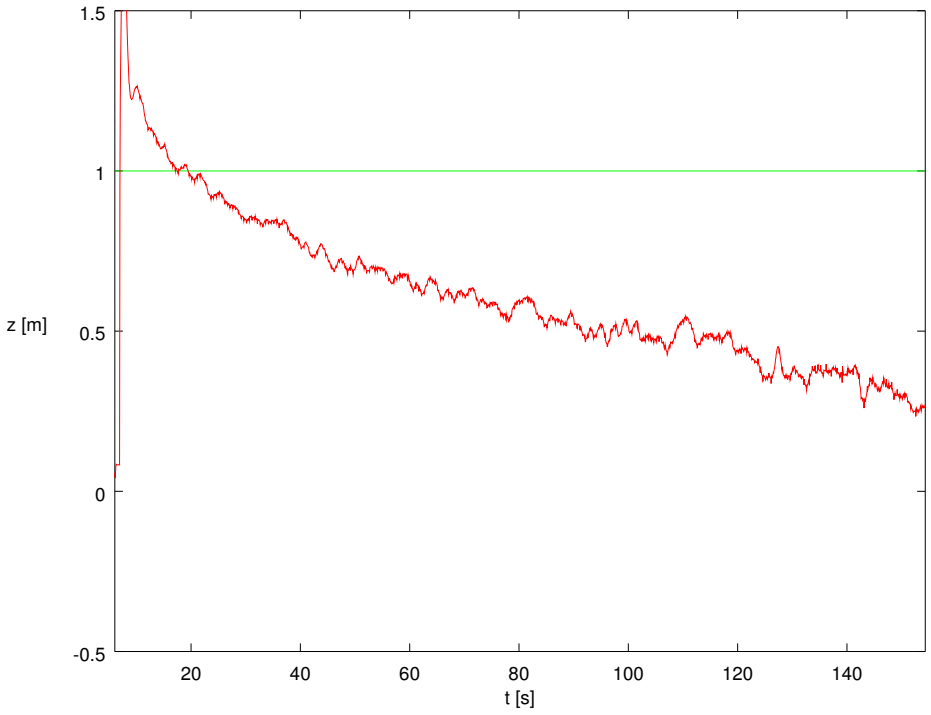

Abbildung 6.3: Der Crazyflie soll in einem Meter Höhe schweben (grüne Linie). Mit der Zeit, also mit abnehmendem Batteriestand, verlor er aber immer weiter an Höhe (rote Linie).

sprechen. Durch Tests wurden daher die Modellkonstanten neu geschätzt. Die besten Testergebnisse konnte mit den Werten aus Tabelle 6.3 erzielt werden.

Konstante	**alter Wert**	**neuer Wert**
g	9.81	9.81
m	0.02	0.02
d	0.045	0.045
I_x	$6.702 \cdot 10^{-4}$	$8 \cdot 10^{-5}$
I_y	$3.383 \cdot 10^{-4}$	$8 \cdot 10^{-5}$
I_z	$7.689 \cdot 10^{-4}$	$8 \cdot 10^{-3}$
$I_{r,c}$	$2.893 \cdot 10^{-6}$	$3.5 \cdot 10^{-6}$
b_c	$2.524 \cdot 10^{-11}$	$3.105 \cdot 10^{-11}$
k_c	$1.103 \cdot 10^{-12}$	$1.3 \cdot 10^{-11}$

Tabelle 6.3: Alter und neu bestimmter Wert der Modellkonstanten.

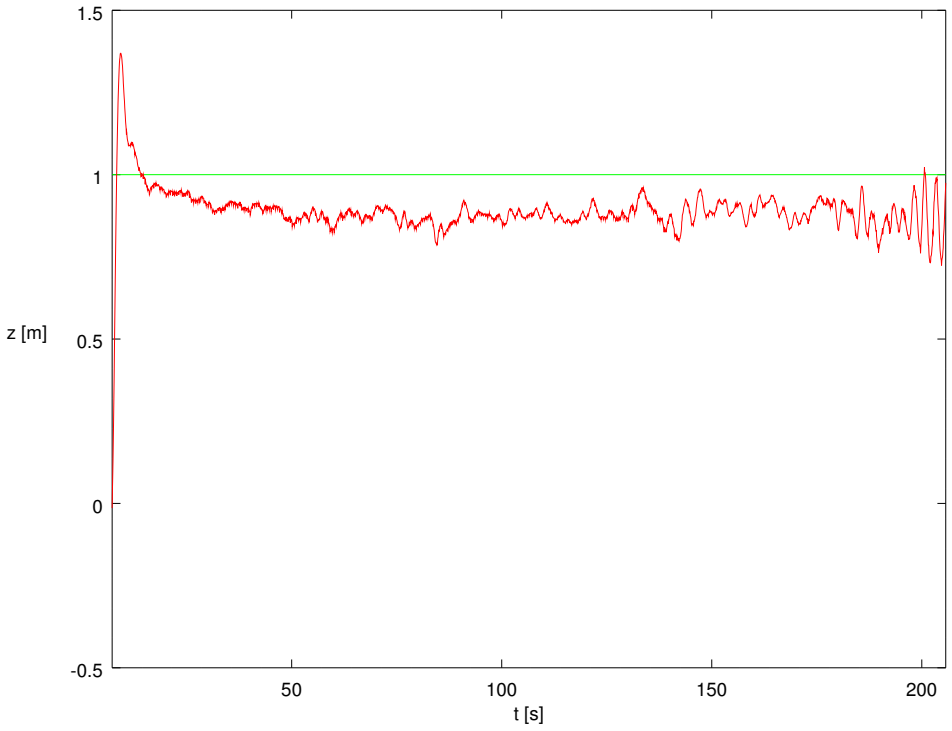

Abbildung 6.4: Der Crazyflie soll in einem Meter Höhe schweben (grüne Linie). Da nun der Batteriestand in die Berechnungen mit einfließt, kann sich der Crazyflie auch über längere Zeit auf der gleichen Höhe halten (rote Linie).

6.3 Realisierung der Positionsregelung

Die Kaskadrenregelung, wie sie in Abbildung 6.2 illustriert ist, kann auf zwei Arten umgesetzt werden. Bei beiden Arten wird der äußere LQ-Regler inklusive der Positionsmessung und dem Hybriden Kalman Filter auf dem Testrechner ausgeführt. Die Feedbackmatrix des inneren LQ-Regler muss auch auf dem Testrechner berechnet werden, da auf dem Crazyflie dafür die benötigte Software fehlt. Nun gibt es zwei Möglichkeiten:

Entweder man sendet diese Feedbackmatrix inklusive den gewünschten Euler-Winkel und dem gewünschten Schub zum Crazyflie. Dort findet auch die Messung der aktuellen Euler-Winkel und der Gyrodaten statt und kann dann schließlich sofort auf die Feedbackmatrix angewendet werden. Diese Variante werden wir im Folgenden mit LQ-LQ-Regler bezeichnen.

Man kann aber auch die auf dem Crazyflie gemessenen Euler-Winkel und Gyrodaten zum Testrechner schicken. Dann wird dort mit dem inneren LQ-Regler die vier Motorkontrollen berechnet und nur diese vier Werte zum Crzayflie gesendet. Diese Variante nennen wir SM-LQ-LQ-Regler (SM für *send motorratios*).

Da auf dem Crazyflie vom Hersteller aber auch ein PID-Regler zur Lageregelung imple-

mentiert ist, haben wir auch eine dritte Möglichkeit für eine Positionsregelung. Wir ersetzen dabei den inneren LQ-Regler aus Abbildung 6.2 durch den PID-Regler. In diesem Fall bestimmt also wieder der äußere LQ-Regler die gewünschten Euler-Winkel und Schub, die dann zum Crazyflie gesendet werden. Dort können dann neue Messdaten sofort angewendet werden. Diese Variante bezeichnen wir mit LQ-PID-Regler.

Die LQ-Regler und das Hybride Erweiterte Kalman Filter erfordern vom Anwender passende Gewichtungs- bzw. Kovarianzmatrizen. Der innere bzw. der äußere LQ-Regler wurde in dieser Arbeit für alle drei Regelvarianten gleich entworfen. In der Praxis konnten die besten Ergebnisse erzielt werden, wenn die Regler und der Filter wie folgt entworfen wurden:

äußerer LQ-Regler

Die Gleichgewichtspunkte (x^*, u^*) der äußeren Systemdynamik (6.2) sind gegeben durch:

$$x^* = \begin{pmatrix} \xi^* \\ 0 \\ 0 \\ 0 \end{pmatrix} \text{ mit } \xi^* \in \mathbb{R}^3,$$

$$u^* = \begin{pmatrix} 0 \\ 0 \\ mg/b_c \end{pmatrix}.$$

Gibt der Anwender nun noch einen Soll-Yaw-Winkel $\psi^* \in \mathbb{R}$ vor, so gilt mit Satz 4.6 und Bemerkung 4.7 für die Matrizen A und B aus der algebraischen Riccati Gleichung:

$$A = \frac{\partial f_{\xi,v}^{\psi^*}(x + x^*, u + u^*)}{\partial x}(0,0) = \begin{pmatrix} 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

$$B = \frac{\partial f_{\xi,v}^{\psi^*}(x + x^*, u + u^*)}{\partial u}(0,0) = \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ \sin(\psi^*)g & \cos(\psi^*)g & 0 & 0 \\ -\cos(\psi^*)g & \sin(\psi^*)g & 0 & 0 \\ 0 & 0 & 0 & b_c/m \end{pmatrix}$$

Da A und B unabhängig von x^* und u^* ist, gilt das gleiche auch für die Lösung der algebraischen Riccati Gleichung und damit für die Feedbackmatrix des LQ-Reglers. Man

muss also die Riccati Gleichung nur erneut lösen, wenn der Anwender einen neuen Soll-Yaw-Winkel ψ^* vorgibt. Für die Matrizen M und N aus Gleichung 4.2 wurde folgende Wahl getroffen:

$$M = \text{diag}(10^2, 10^2, 10^2, 1, 1, 1)$$
$$N = \text{diag}(1.62 \cdot 10^3, 1.62 \cdot 10^3, 1.45 \cdot 10^{-17}).$$

innerer LQ-Regler

Die Gleichgewichtspunkte (x^*, u^*) der inneren Systemdynamik (6.3) sind gegeben durch:

$$x^* = \begin{pmatrix} \phi^* \\ \theta^* \\ \psi^* \\ 0 \\ 0 \\ 0 \end{pmatrix} \text{ mit } \phi^*, \theta^*, \psi^* \in \mathbb{R} \text{ und } \cos(\theta^*) \neq 0,$$

$$u^* = \begin{pmatrix} \lambda^* \\ \lambda^* \\ \lambda^* \\ \lambda^* \end{pmatrix} \text{ mit } \lambda^* \in \mathbb{R}.$$

Für die Matrizen A und B aus der algebraischen Riccati Gleichung gilt mit Satz 4.6 und Bemerkung 4.7:

$$A = \frac{\partial f_{\eta,\nu}(x + x^*, u + u^*)}{\partial x}(0,0) = \begin{pmatrix} 0 & 0 & 0 & 1 & \sin(\phi^*)\tan(\theta^*) & \cos(\phi^*)\tan(\theta^*) \\ 0 & 0 & 0 & 0 & \cos(\phi^*) & -\sin(\phi^*) \\ 0 & 0 & 0 & 0 & \sin(\phi^*)/\cos(\theta^*) & \cos(\phi^*)/\cos(\theta^*) \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

$$B = \frac{\partial f_{\eta,\nu}(x + x^*, u + u^*)}{\partial u}(0,0) = \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & -2db_c\lambda^*/I_x & 0 & 2db_c\lambda^*/I_x \\ -2db_c\lambda^*/I_y & 0 & 2db_c\lambda^*/I_y & 0 \\ -2k_c\lambda^*/I_z & 2k_c\lambda^*/I_z & -2k_c\lambda^*/I_z & 2k_c\lambda^*/I_z \end{pmatrix}$$

Für die Matrizen M und N aus Gleichung 4.2 wurde folgende Wahl getroffen:

$$M = \text{diag}(5 \cdot 10^3, 5 \cdot 10^3, 5 \cdot 10^3, 4, 4, 4)$$
$$N = \text{diag}(4 \cdot 10^{-5}, 4 \cdot 10^{-5}, 4 \cdot 10^{-5}, 4 \cdot 10^{-5}).$$

Hybrides Erweiterte Kalman Filter

Für die Kovarianzmatrizen des Hybriden Erweiterten Kalman Filters wurde folgende Wahl getroffen:

$$Q(t) = \text{diag}(1, 1, 1, 10^3, 10^3, 10^3)$$
$$R_k = \text{diag}(10^{-1}, 10^{-1}, 10^{-1})$$

Die Kovarianzmatrizen wurden also konstant gewählt. Außerdem wurde angenommen, dass die Ableitung der Systemdynamik nach dem Prozessrauschen bzw. die Ableitung der Ausgangsfunktion nach dem Messrauschen jeweils die Einheitsmatrix bilden.

6.4 Ergebnisse

Die Abbildungen 6.5, 6.6 und 6.7 zeigen Testflüge mit den entsprechenden Regelvarianten. Insgesamt sind hier also 9 Testflüge abgebildet, wobei die Flüge pro Abbildung immer direkt nacheinander stattgefunden haben.

In grün ist die gewünschte Position dargestellt. Der Crazyflie sollte jeweils an der Position $(0, 0, 1)^T$ schweben (in Metern). Es ist festzustellen, dass sich die verschiedenen Regelmechanismen in der resultierenden Höhe kaum unterscheiden und ähnlich gute Ergebnisse liefern. Betrachtet man aber die Position in x- und y-Richtung, so sind deutliche Unterschiede zu erkennen. Der LQ-PID-Regler schneidet auch hier mit Abweichungen um die 10 Zentimeter relativ gut ab. Der LQ-LQ- und auch der SM-LQ-LQ-Regler haben diesbezüglich oft unterschiedlich gute Leistungen gezeigt. Typisch war aber ein Versatz von 10 bis 30 Zentimetern entlang der positiven x- bzw. y-Achse. In Abbildung 6.6 war der Batteriestand des Crazyflies relativ niedrig. Dies ist wahrscheinlich der Grund für die relativ großen Abweichungen bzgl. der gewünschten Position in y-Richtung. Hier hat sich dann auch der LQ-PID-Regler gegen Ende des Fluges weiter als sonst von der Zielposition entfernt. Interessant dabei ist auch der ähnliche Verlauf der y-Komponente beim LQ-LQ- und SM-LQ-LQ-Regler. Es handelt sich hier um zwei verschiedene Testflüge, aber trotzdem sind beide zum gleichen Zeitpunkt (gemessen vom jeweiligen Startzeitpunkt) ungewöhnlich weit und im gleichen Ausmaße von der Zielposition abgewichen.

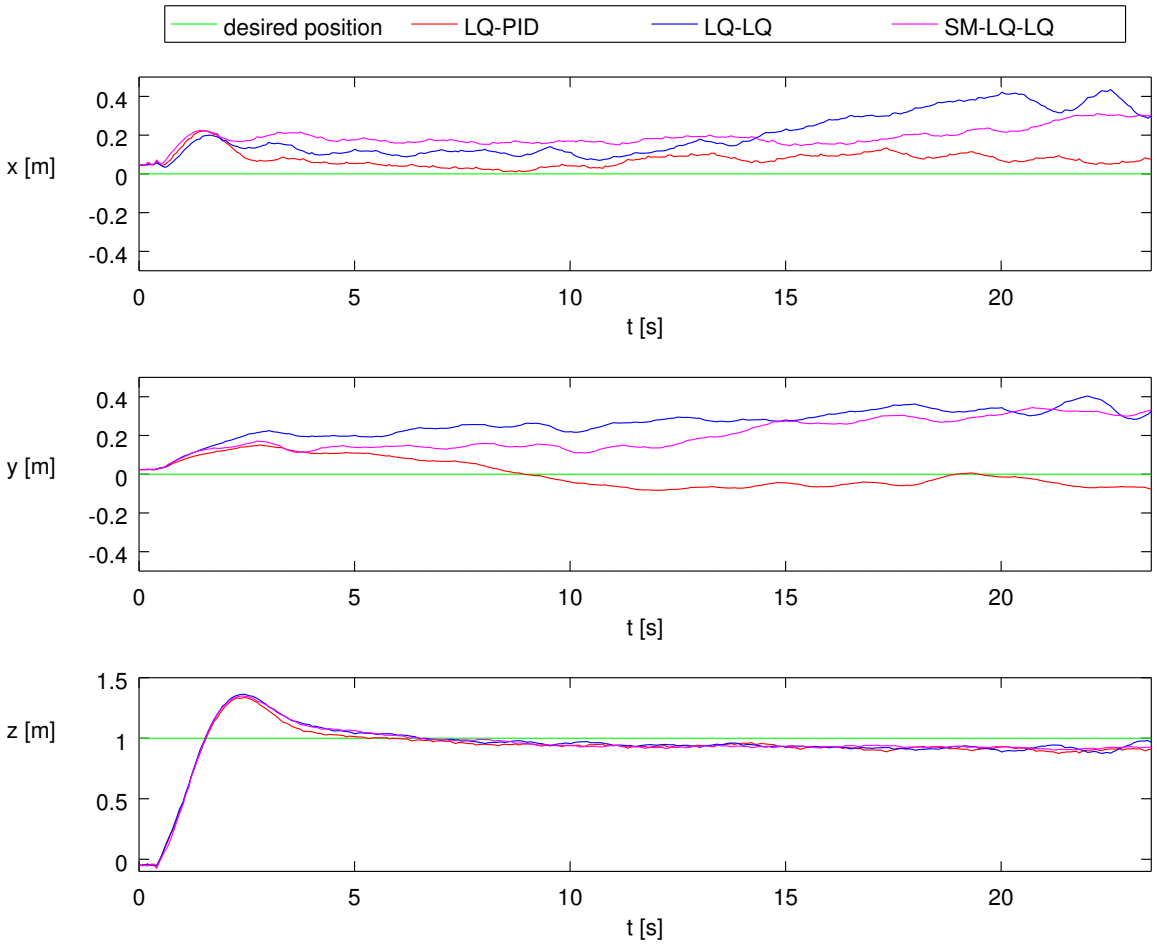

Abbildung 6.5: 3 Testflüge mit den verschiedenen Regelmechanismen.

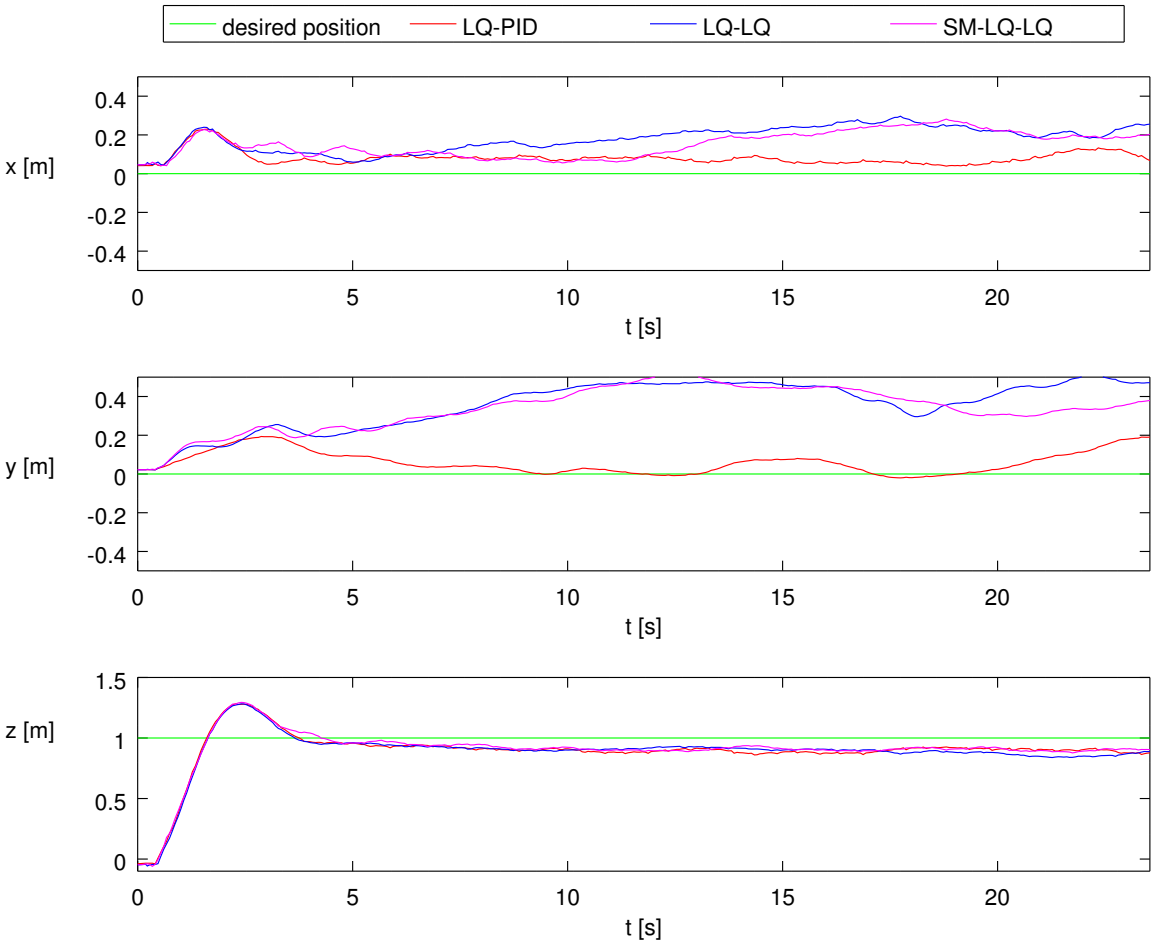

Abbildung 6.6: 3 weitere Testflüge mit den verschiedenen Regelmechanismen, allerdings mit einem relativ geringen Ladezustand der Batterie.

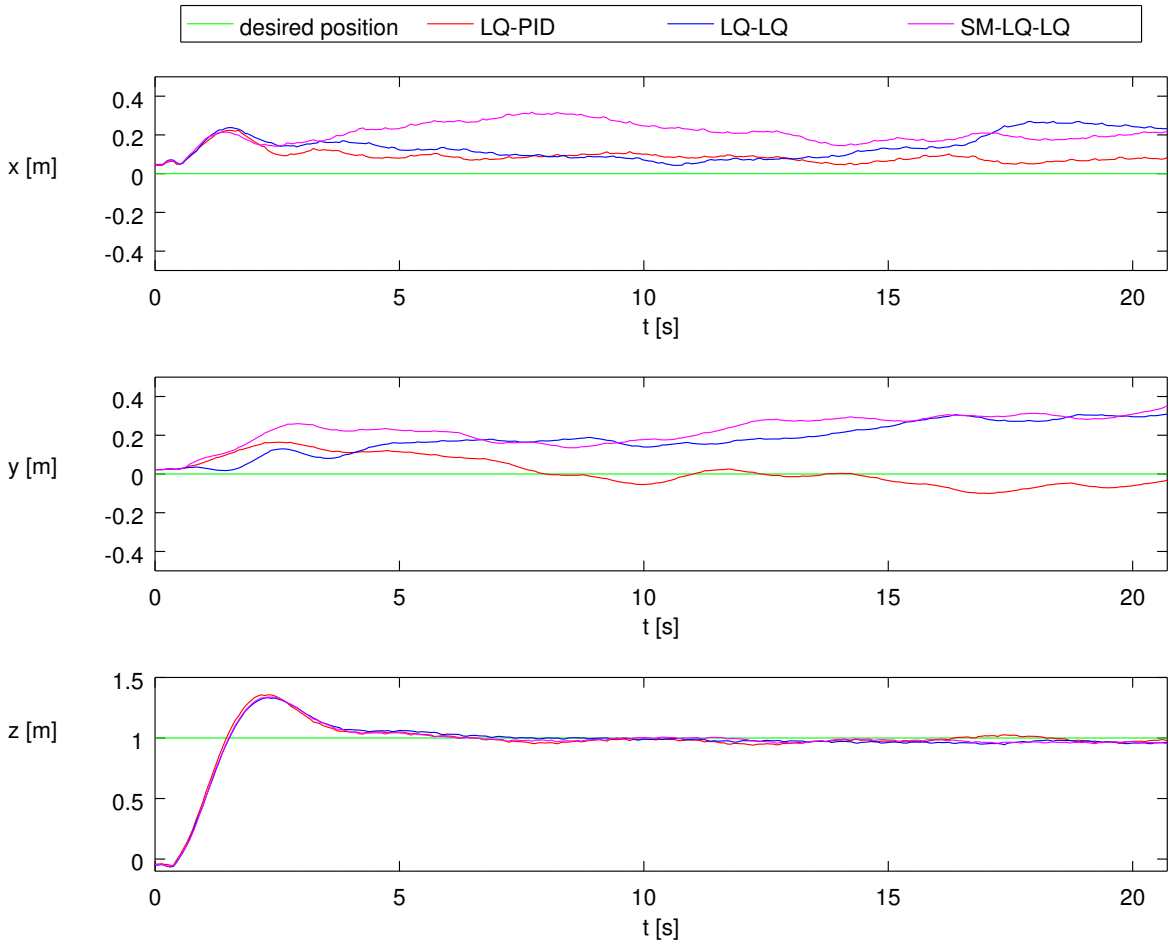

Abbildung 6.7: 3 weitere Testflüge mit den verschiedenen Regelmechanismen. Die Batterie war hier wieder aufgeladen.

Um den LQ-LQ- und den SM-LQ-LQ-Regler zu verbessern, müssen die Modellkonstanten wohl noch weiter optimiert werden. Mit den hier vorgestellten Parametern konnte gegenüber den in [14] genannten Konstanten schon Verbesserungen erreicht werden. Dort traten sogar Abweichungen um die 50 Zentimeter auf.

Es könnte aber auch sein, dass der Quadcopter nicht genau genug modelliert wurde. Es wurde nämlich angenommen, dass gleiche Kontrollwerte $u_i = u_j$ mit $0 < i < j < 5$ in gleiche Winkelgeschwindigkeiten für Rotor i und j resultieren. Subjektiv hat der Autor aber den Eindruck, dass sich Rotor 2 und 3 leichter drehen lassen als Rotor 1 und 4. Dies würde auch die Ergebnisse der obigen Testflüge erklären. Drehen sich die Rotoren 1 und 4 langsamer als sie sollten, so wird die tatsächliche Position entlang der x- und y-Achse größer sein, als gewünscht. Mit anderen Worten: Der innere LQ-Regler wird den berechneten Roll- und Pitch-Winkel, die der äußere Regler vorgibt, nicht einhalten können. Der Roll-Winkel wird zu gering sein, während der Pitch-Winkel in Wirklichkeit zu groß sein wird. Der LQ-PID-Regler könnte diesem Phänomen gegenüber robuster sein, da der I-Teil über den Fehler bezüglich dem Roll- bzw. Pitch-Winkel integriert. Weichen diese Winkel über eine längere Zeit vom Soll-Winkel ab, so wirkt sich das auch auf die berechneten Kontrollwerte aus.

Anstatt nun das mathematische Modell zu verfeinern, kann man unter der Voraussetzung, dass der tatsächliche Roll-Winkel um einen konstanten Wert $\Delta\phi > 0$ zu gering und der tatsächliche Pitch-Winkel um einen konstanten Wert $\Delta\theta > 0$ zu groß ist, den LQ-LQ und SM-LQ-LQ-Regler wie folgt anpassen:

Der äußere LQ-Regler berechnet die Soll-Winkel ϕ^* und θ^*. Anstatt nun diese Werte an den inneren LQ-Regler weiterzugeben, werden die Winkel $\phi^* + \Delta\phi$ und $\theta^* - \Delta\theta$ verwendet. In dieser Arbeit wurde angenommen, dass $\Delta\phi = 2\text{rad}$ und $\Delta\theta = 1\text{rad}$ gilt. In Abbildung 6.8 ist das Ergebnis dieser Modifikation zu sehen. Die drei Arten der Regelungen liefern nun keine bemerkenswerten Unterschiede mehr. $\Delta\theta$ könnte noch etwas erhöht werden um die x-Komponente noch ein wenig zu verringern. Aber auch so liefern alle drei Regelungen nun relativ gute Ergebnisse.

In Abbildung 6.9 sollte der Crazyflie einen Pfad (grün) mit dem LQ-PID-Regler nachfliegen. Dabei wurden ausgewählte Punkte des Pfades vorgegeben, die dann als Zielpunkte eingesetzt wurden (grüne Kreuze). Hat sich der Crazyflie in einer Umgebung um den aktuellen Zielpunkt befunden (hier in einer Entfernung von 15 Zentimetern), so wurde der nächste Punkt des Pfades als neues Ziel gewählt. In rot ist schließlich die Flugbahn des Crazyflies eingezeichnet. Die blauen Kreuze signalisieren, wann der Crazyflie in die oben erwähnte Umgebung der aktuellen Zielposition eingedrungen ist und das Ziel dementsprechend aktualisiert worden ist.

Startpunkt des Crazyflies war in etwa im Nullpunkt. Der erste Zielpunkt war in $(0, 0, 0.5)^T$. Abgesehen von dem starken Überschwingen der Höhe beim Start, wird der Pfad relativ gut nachgeflogen. Dass meistens zu früh ein neuer Zielpunkt gesetzt wird, liegt daran, dass ge-

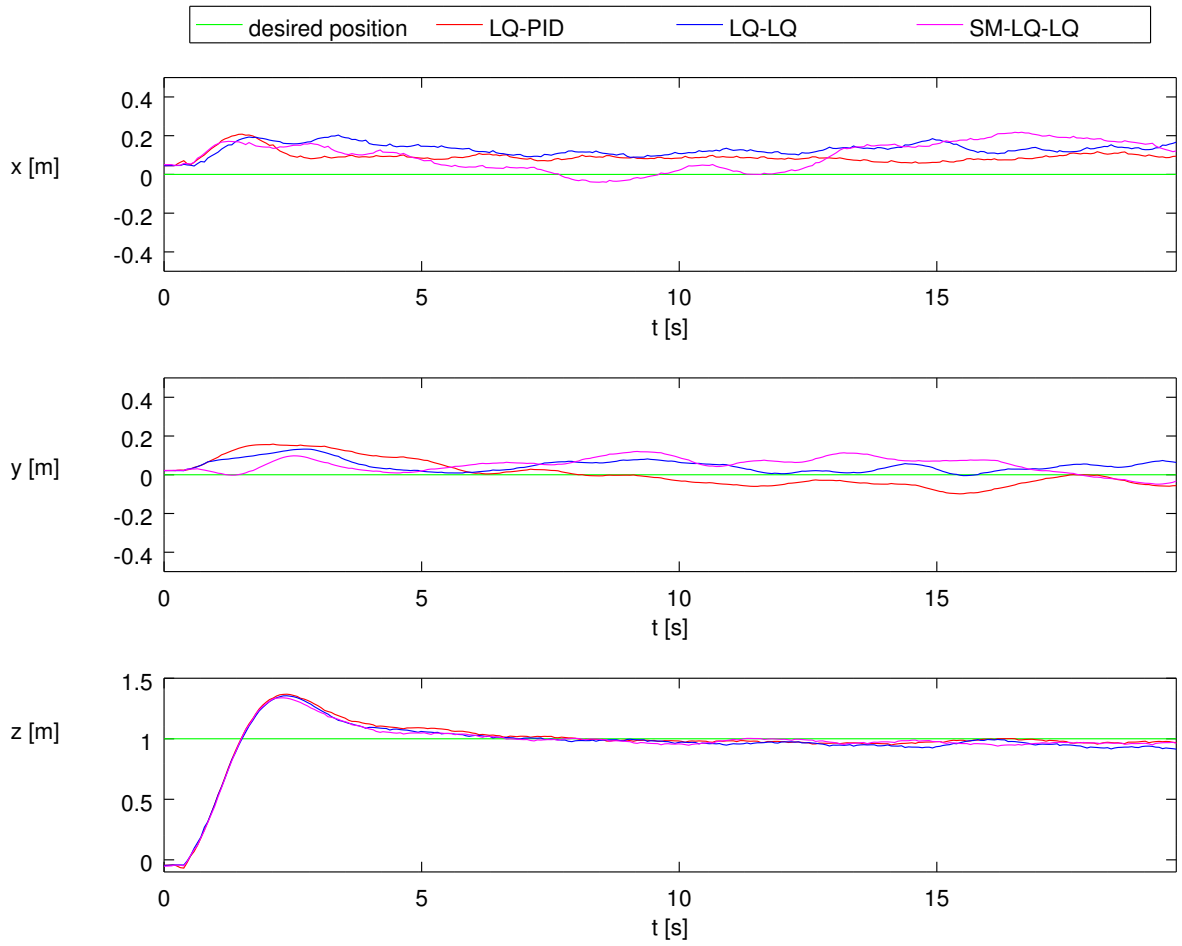

Abbildung 6.8: 3 weitere Testflüge mit den verschiedenen Regelmechanismen. Der berechnete Roll- bzw. Pitch-Winkel wurden sowohl beim LQ-LQ- als auch beim SM-LQ-LQ-Regler um 2rad erhöht bzw. um 1rad verringert.

testet wurde, ob sich der Crazyflie in einer Umgebung um den aktuellen Zielpunkt befindet. Reduziert man aber diese Umgebung, so konnte es vorkommen, dass der Quadcopter nicht in diese verkleinerte Umgebung eindringen konnte.

Auf der beiliegenden CD befinden sich auch Videos zu den Abbildungen in diesem Abschnitt. Dabei handelt es sich jeweils um die Aufnahmen der ersten Kamera.

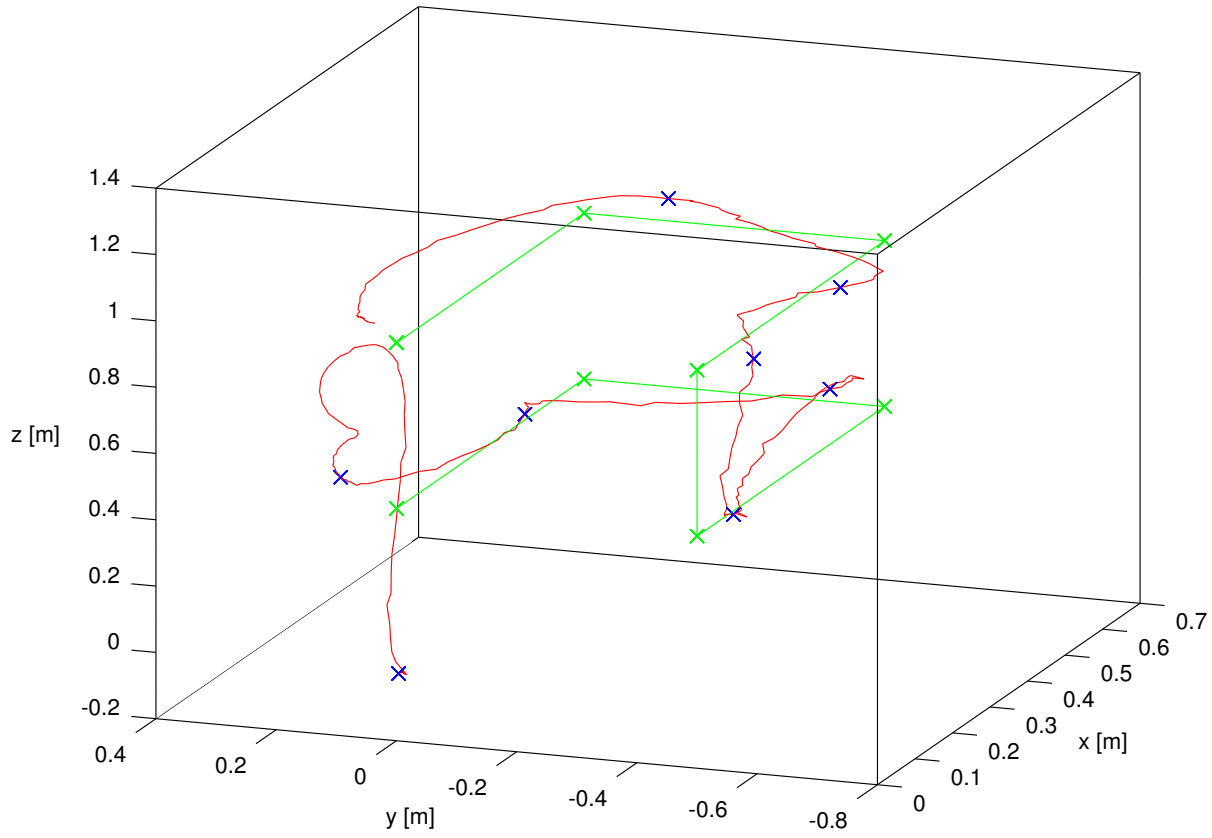

Abbildung 6.9: Crazyflie soll der grünen Linie nachfliegen.

Kapitel 7

Fazit

7.1 Zusammenfassung

In Kapitel 3 wurde zunächst das Kameramodell vorgestellt, das *OpenCV* verwendet. Bei der Kalibrierung haben wir festgestellt, dass das Modell mit 8 Parametern keine nennenswerte Verbesserung gegenüber dem Modell mit 5 Parameter mit sich bringt (zumindest bei den in dieser Arbeit verwendeten Kameras). Anschließend haben wir gesehen, wie man 3D-Punkte mit Hilfe einer Stereo-Kamera rekonstruieren kann. Um den Crazyflie in dem entstehenden Bildmaterial zu suchen, wurden zwei Trackingverfahren vorgestellt, die darauf basieren, dass die auf dem Crazyflie verbaute LED die hellste Lichtquelle darstellt.

In Kapitel 4 wurden Ergebnisse aus der linear-quadratischen Regelungstheorie aufgezeigt. Um die Lösung der algebraischen Riccati Gleichung zu bestimmen, wurde ein Verfahren hergeleitet, das auf der Matrix Signumsfunktion basiert.

In Kapitel 5 wurde das Hybride Erwetierte Kalman Filter ausgehend von der Rekursiven Methode der kleinsten Quadrate hergeleitet. Dieser Filter ist für zeitkontinuierliche Systemdynamiken und zeitdiskreten Messungen entworfen.

In Kapitel 6 wurden diese Ergebnisse schließlich angewendet um eine kameraunterstützte Positionsregelung umzusetzen. Dabei haben wir gesehen, dass man bei der Positionsregelung den Ladezustand der Batterie nicht ignorieren darf. Nach einer Anpassung der Modellkonstanten wurde die Positionsregelung schließlich in der Praxis mit dem LQ-PID-, dem LQ-LQ- und dem SM-LQ-LQ-Regler getestet. Es hat sich gezeigt, dass der LQ-LQ und SM-LQ-LQ-Regler zunächst immer entlang der positiven x- und y-Achse versetzt war. Grund dafür ist wahrscheinlich ein nicht ganz zutreffendes mathematisches Modell. Durch künstliches Anpassen des Roll- und Pitch-Winkels, die der äußere LQ-Regler berechnet, konnte man darauf aber realtiv einfach reagieren und somit lieferten auch der LQ-LQ- und SM-LQ-LQ-Regler gute Ergebnisse.

7.2 Ausblick

Als nächstes könnte man Versuchen eine Positionsregelung umzusetzen, ohne die Systemdynamik aufzuteilen. Das Erweiterte Kalmanfilter kann auch mit der Situation umgehen, dass Zustandskomponenten in verschiedenen Samplingraten abgetastet werden (unterschiedliche H_k, bzw. R_k in Algorithmus 6). In dieser Anwendung muss man aber bedenken, dass die Stereo-Kamera in etwa alle 1000/30 Millisekunden neue Bilder liefert, die Zeit zur Berechung der neuen Position kann aber nicht vernachlässigt werden. In der Regel wird der Fall eintreten, dass während der Berechnung der Position zum Zeitpunkt t_k bereits mehrere neue Euler- und Gyro-Daten zum Zeitpunkt t_{k+i} mit $i > 0$ eintreffen.

Ein naiver Ansatz wäre nun einen zweiten HEKF zu starten, der mit dem Zustand des ersten HEKF zum Zeitpunkt t_{k-1} initialisiert wird. Solange die Positionsberechnung zum Zeitpunkt t_k noch nicht abgeschlossen ist, wird der zweite HEKF mit den Euler- und Gyro-Daten zum Zeitpunkt t_{k+i} mit $i = 1, ..., I$ aktualisiert. Sobald die Position berechnet wurde, kann man damit den ersten HEKF aktualisieren, muss dann aber zusätzlich die Aktualisierungen des ersten HEKF mit den Euler- und Gyro-Daten zum Zeitpunkt t_{k+i} mit $i = 1, ..., I$ auch noch durchführen. Je größer I ist, desto länger wird also dieses „auf den aktuellen Stand bringen“ des ersten HEKF dauern und im schlimmsten Fall kann dies zur Instabilität des Systems führen.

Eine Alternative dazu ist ein HEKF, der extra für solch eine Situation zugeschnitten ist. In der Fachliteratur wird dies mit „EKF for Out-of-Sequence Measurements“ bezeichnet.

Als nächsten Schritt wäre die Umsetzung eines Modell-prädiktiven Regelansatzes denkbar. Dabei wird wahrscheinlich die echtzeitfähigkeit des Reglers die größte Herausforderung stellen. Der große Vorteil dieses Regeltypes wäre aber, dass auch Nebenbedingungen an den Zustand und die Kontrolle gestellt werden können.

Ein letzter Anregungspunkt wäre die Verbesserung der Algorithmen zum Tracken des Crazyflies. Aktuell wird immer im gesamten Bild nach dem Quadcopter gesucht. Nachdem wir aber die Systemdynamik kennen und Informationenen über die Position sowie die Geschwindigkeit haben, sollte man auch einen Suchbereich berechnen können, in dem sich der Crazyflie (höchstwahrscheinlich) befindet. Eventuell kann man die Bildverarbeitung dabei auch so stark beschleunigen, dass die Messdaten wieder in geordneter Reihenfolge vorliegen. Ob dies dann auch der Fall ist, kann nur ein Praxistest zeigen.

Interessant wäre natürlich auch die Umsetzung von Trackingalgorithmen, die nicht mehr so stark auf passende Lichtverhältnisse im Raum angewiesen sind. Sollten diese Algorithmen zu rechenintensiv sein, so könnte man diese auf die GPU auslagern, da *OpenCV* auch eine Cuda-Schnittstelle anbietet. Im Anhang B wird ein Verfahren vorgestellt, wie ein solcher Trackingalgorithmus aussehen könnte. Um diesen auch in der Praxis einzusetzen, müssten aber noch Vorkehrungen getroffen werden, die ihn robuster und schneller machen.

Anhang A

Beispiel einer Main-Funktion

Listing A.1 enthält ein Beispiel eines Main Progammes. Zunächst müssen einige Header-Files eingebunden werden. Ab Zeile 18 beginnt eine Methode, mit der man den in dieser Arbeit verwendeten AR-Marker (vgl. Abbildung 3.5) erstellen und in die Datei *arucomarker.png* abspeichern kann.

Mit der Methode *cntrThread* in Zeile 43 kann der Anwender neue Positionsdaten über die Standardeingabe eingeben. Dabei gibt es drei Möglichkeiten: Ist die erste Zahl eine 1, so will der Anwender einen einzelnen Zielpunkt setzen, die durch die nächsten drei Koordinaten festgelegt wird. Ist die erste Zahl eine 2, so wird eine in der Main-Funktion vordefinierte Serie an Zielpunkten nachgeflogen. Ist die erste Zahl eine 0, so wird das Programm beendet.

In Zeile 80 beginnt die Main-Funktion. In Zeile 84 wird die Gtk-Umgebung initialisiert. Dies ist nur notwendig, wenn man die Kameras unter Zuhilfenahme der Klasse *Calibration-GUI* kalibrieren will. In Zeile 86 wird ein *StereoCamera*-Objekt erzeugt. Der Übergabeparameter entspricht dabei der ID, die das Betriebssystem der ersten Kamera zugewiesen hat. Es wird dabei angenommen, dass die zweite Kamera die darauffolgende ID zugewiesen wurde, also hier die 2. Man sollte dabei bedenken, dass der Rechner bei einem Neustart den USB-Kameras nicht immer in der gleichen Reihenfolge ihre ID's zuweist. Es ist also nicht sichergestellt, dass die Kamera mit der ID=1 beim nächsten Systemstart wieder die ID=1 zugewiesen bekommt. Möchte man bereits bestimmte Kameraparameter wieder verwenden, so sollte man die Kameras (immer in der gleichen Reihenfolge) erst nach dem Systemstart mit dem Rechner verbinden. Der Testrechner verfügt über eine integrierte Kamera, die die ID=0 besitzt. Daher starten hier die ID's der USB-Kameras mit 1.

Hier sei auch erwähnt, dass im Konstruktor der Kamera-Klasse das *v4l2*-Programm aufgerufen wird um den Autofokus der Kamera zu deaktivieren. Falls die verwendete Kamera keinen Autofokus besitzt, so kann es sein, dass an dieser Stelle das Programm abbricht. Gegebenenfalls muss die Anweisung dann auskommentiert werden.

In Zeile 87 wird ein 6×9 Schachbrettmuster erzeugt, wobei die Seitenlänge der Quadrate 0.026 Meter beträgt. Anschließend wird ein Objekt der Klasse *CalibrationGUI* erzeugt. Neben der Stereo-Kamera und dem Schachbrett wird der Pfad übergeben, in dem die aufgenommenen Bilder abgespeichert werden.

In Zeile 92 wird die graphische Oberfläche gestartet. Dabei wird ein Fenster erzeugt, wie es in Abbildung A.1 zu sehen ist. Dort kann man nun auswählen, ob man eine der folgenden Operationen nur an der ersten, nur an der zweiten Kamera oder an der Stereo-Kamera ausführen möchte:

- **Record:** Die ausgewählte(n) Kamera(s) nehmen solange Bilder auf, bis auf den Button *Stop Recording* gedrückt wird.
- **Calibrate:** Sofern für die ausgewählte(n) Kamera(s) schon Bilder aufgenommen wurden, wird nun die Kalibrierung durchgeführt.
- **Save:** Sofern die ausgewählte(n) Kamera(s) schon kalibriert wurden, kann man über diesen Button die Kameraparameter in einer Datei abspeichern.
- **Load:** Lädt die Parameter für die ausgewählte(n) Kamera(s) aus einer Datei.
- **Exit:** Beendet die graphische Oberfläche.

Wird die Aufnahme der Bilder über den Button *Stop Recording* beendet, so werden dem Nutzer nacheinander alle Bilder angezeigt, in denen das Schachbrettmuster gefunden wurde. Dabei werden auch die Eckpunkte eingezeichnet, die das Programm berechnet hat (vgl. Abbildung A.2). Es kann vorkommen, dass einzelne Eckpunkte nicht mit den realen Eckpunkten übereinstimmen, was oft daran liegt das Schachbrettmuster stark verschwommen ist. Dies kann passieren, wenn das Muster zu schnell bewegt wird. Ein Beispiel dafür ist in Abbildung A.3 zu sehen. Da sich solche Ausreißer negativ auf die Kalibrierung auswirken würden, steht dem Anwender ein Dialogfenster zur Verfügung (vgl. Abbildung A.4) um für jede Aufnahme zu entscheiden, ob das Bild übernommen oder verworfen werden soll. Die übernommenen Bilder (ohne den eingezeichneten Eckpunkten) werden dabei je nach Auswahl in den Ordner „Cali/Cam1/“ , „Cali/Cam2/“ bzw. „Cali/Stereo/“ abgespeichert, wenn „Cali/“ der Übergabeparameter bei der Erstellung des *CalibrationGUI*-Objektes war. Damit bei der Kalibrierung auch sicher nur die gerade aufgenommenen Bilder verwendet werden, werden vor Beginn der Aufnahme alle Dateien im Ordner „Cali/Cam1/“ , „Cali/Cam2/“ bzw. „Cali/Stereo/“ (ohne Hinweis) gelöscht!

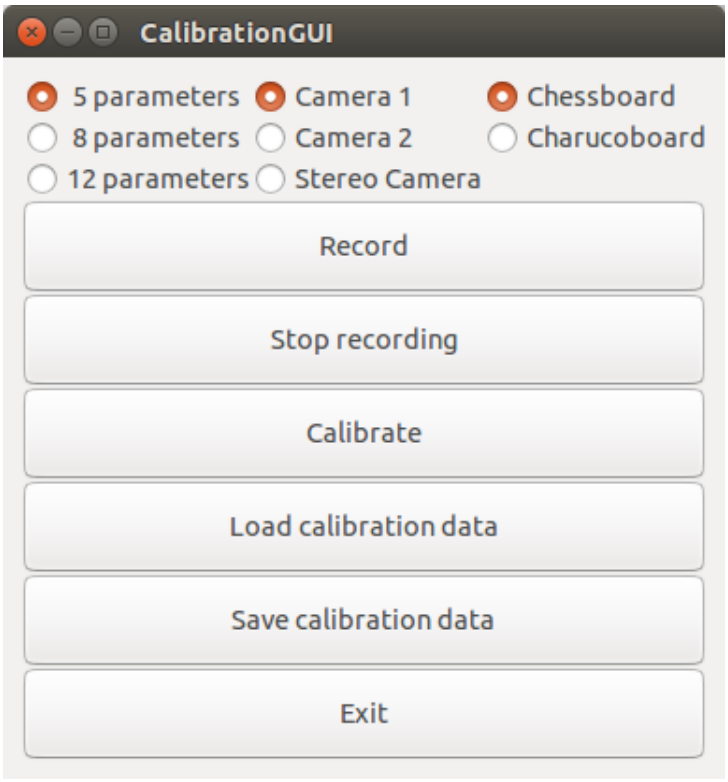

Abbildung A.1: Main Fenster

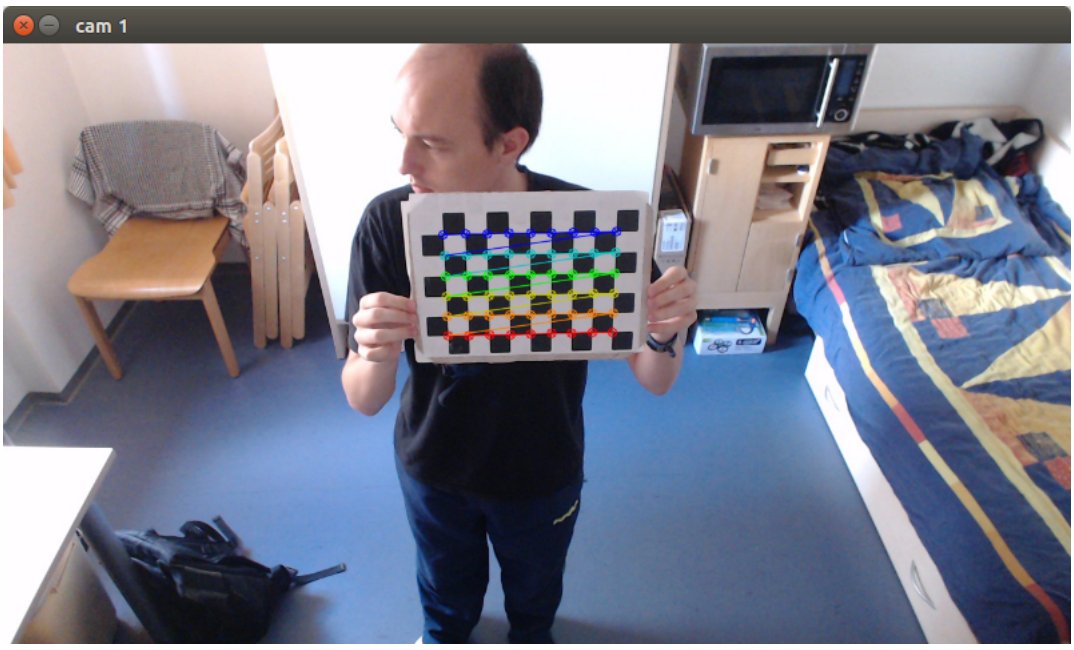

Abbildung A.2: Die Ecken des Schachbrettmusters wurden richtig erkannt.

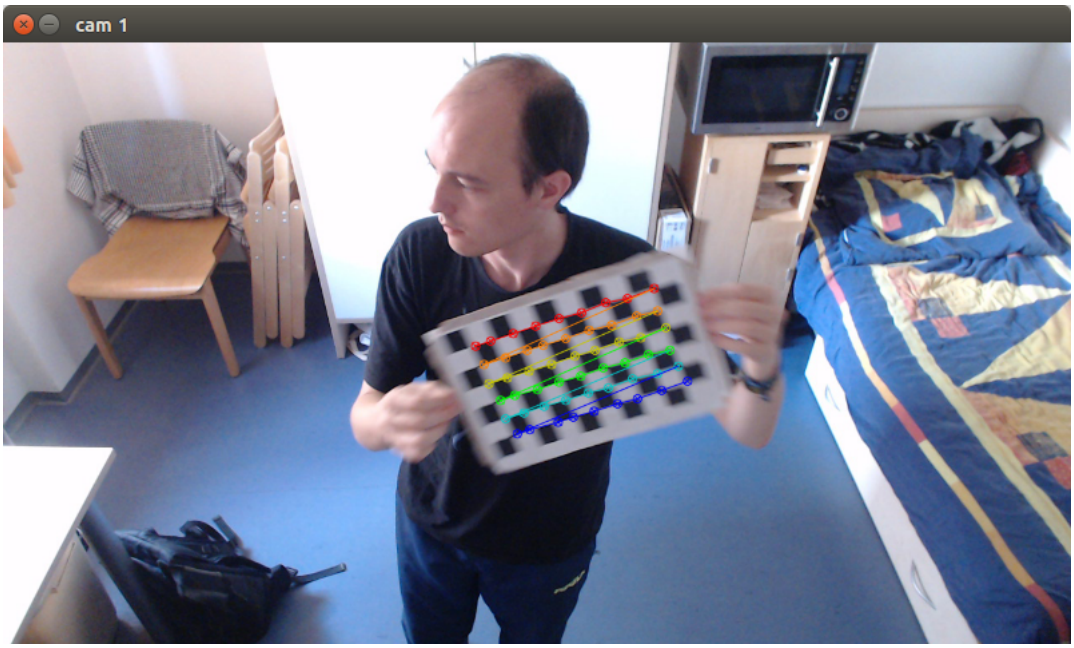

Abbildung A.3: Das Schachbrettmuster wurde hier zwar erkannt, aber manche Eckpunkte konnten nicht gut bestimmt werden. Diese Aufnahme sollte nicht zum Kalibrieren verwendet werden.

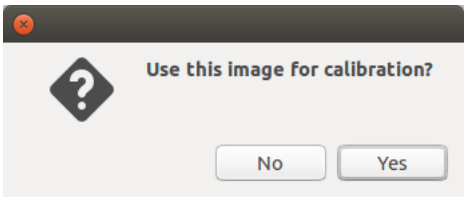

Abbildung A.4: Das Dialogfenster um eine Aufnahme für die Kalibrierung zu übernehmen oder zu verwerfen.

Nach dem die Stereo-Kalibrierung durchgeführt worden ist, wird das Weltkoordinatensystem initialisiert. Dazu wird in Zeile 95 zunächst der AR-Marker erstellt, mit dem der Anwender die Initialisierung durchführen will. Die Initialisierung selbst erfolgt schließlich in Zeile 96, wobei neben dem Marker auch die Länge in Metern des ausgedruckten Markers übergeben wird. Je nachdem wie der Anwender das Weltkoordinatensystem haben möchte, platziert er den AR-Marker im Raum entsprechend. Ihm wird dabei das Bild der ersten Kamera angezeigt. Wird der AR-Marker auf dem Bild erkannt, so wird dessen Orientierung durch Einzeichnen eines Koordinatensystems gekennzeichnet (vgl. Abbildung A.5). Bei Bestätigung durch den Anwender, würde dieses Koordinatensystem dem Weltkoordinatensystem entsprechen. Der Anwender kann durch drücken der Taste „q“ die Wahl bestätigen.

Dabei muss aber das Bild im Fokus sein (nicht das Fenster mit der Standardeingabe!). Rot signalisiert dabei die positive x-Achse, grün die positive y-Achse und blau die positive z-Achse.

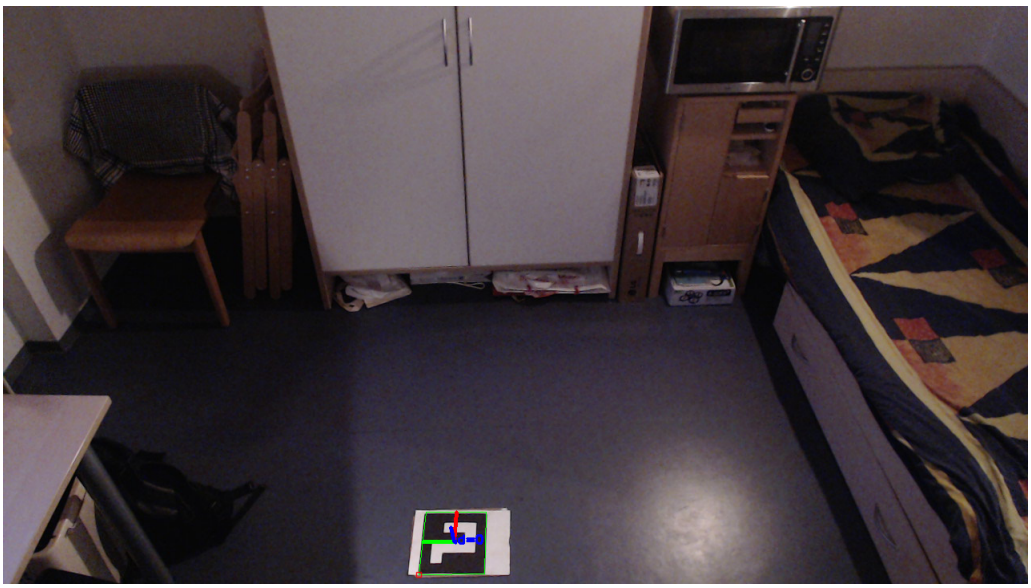

Abbildung A.5: Wenn der AR-Marker erkannt wird, wird dessen Orientierung durch ein eingezeichnetes Koordinatensystem dargestellt.

In Zeile 98 wird ein Enum belegt, der festlegt, welcher Regeltyp verwendet werden soll. Anschließend wird in einem Thread die oben erwähnte Methode *cntrThread* gesatartet, die über die Standardeingabe Kommandos vom Anwender einliest. In diesem Fall sollen so die gewünschten Positionsdaten eingegeben bzw. aktualisiert werden.

Ab Zeile 101 wird eine Serie von Zielpunkten angelegt, die der Crazyflie auf Wunsch nachfliegt. Danach wird ein *CCrazyRadio*-Objekt erstellt, das die Kommunikation über die Antenne verwaltet. Anschließend wird mit diesem Objekt, dem ausgewähltem Reglertyp und der erzeugten Stereo-Kamera ein *CCrazyflie*-Objekt erzeugt. In der darauffolgenden while-Schleife werden diesem Objekt die aktuell gewünschten Positionsdaten übermittelt. Soll der Crazyflie der vordefinierten Serie von Zielpunkten folgen, so wird dies in Zeile 125 durchgeführt. Die zweite Übergabeparameter gibt dabei an, ob die Trajektorie wiederholt nachgeflogen werden soll (true), oder ob der Crazyflie beim erreichen des letzten Zielpunktes von *vecSetpoints* an dieser Stelle schweben soll.

Die eigentliche Berechnung der Regelung und das Senden von Daten zum Crazyflie geschieht mit dem Aufruf von *cflieCopter.cycle()*.

Listing A.1: Main Programm

```
1 #include <iostream>
```

```
2
3  #pragma GCC diagnostic ignored "-Wdeprecated-declarations"
4  #include <gtkmm.h>
5  #pragma GCC diagnostic warning  "-Wdeprecated-declarations"
6
7  #include "CCrazyRadio.h"
8  #include "CCrazyflie.h"
9  #include "StereoCamera.h"
10 #include "Controller.h"
11 #include "CalibrationGUI.h"
12 #include "Chessboard.h"
13 #include "joystick.h"
14 #include "aruco.hpp"
15 #include "charuco.hpp"
16 #include <vector>
17
18 void create_aruco_marker()
19 {
20     int markerId = 0;
21     int borderBits = 1;
22     int markerSize = 2000; //pixel
23
24 //     int dictionaryId = 0;
25 //     cv::aruco::Dictionary dictionary = cv::aruco::getPredefinedDictionary(cv
       ::aruco::PREDEFINED_DICTIONARY_NAME(dictionaryId));
26     cv::aruco::Dictionary dictionary = cv::aruco::generateCustomDictionary(1,
           4);
27
28     cv::Mat markerImg;
29     cv::aruco::drawMarker(dictionary, markerId, markerSize, markerImg,
           borderBits);
30
31     cv::imshow("marker", markerImg);
32     cv::waitKey(0);
33
34     std::string filename = "arucomarker.png";
35     cv::imwrite(filename, markerImg);
36 }
37
38 bool shutdown, newTrajectory = false, followTrajectory;
39 double posX, posY, posZ, thrust = 0;
40 bool sendSetPoints = false;
41 bool threadstillrunning = true;
42
43 void cntrThread()
44 {
45     threadstillrunning = true;
46     shutdown = false;
47
```

```
48      std::cout << "Enter command:" << std::endl;
49      std::cout << "  1 x y z     to set single setpoint" << std::endl;
50      std::cout << "  2           to use predefined trajectory" << std::endl;
51      std::cout << "  0           to exit" << std::endl << std::endl;
52
53      int mode;
54
55      while(!shutdown)
56      {
57          std::cin >> mode;
58          switch(mode)
59          {
60          case 0:
61              sendSetPoints = false;
62              shutdown = true;
63              break;
64          case 1:
65              sendSetPoints = true;
66              std::cin >> posX >> posY >> posZ;
67              followTrajectory = false;
68              break;
69          case 2:
70              sendSetPoints = true;
71              followTrajectory = true;
72              newTrajectory = true;
73              break;
74          }
75      }
76
77      threadstillrunning = false;
78  }
79
80  int main(int argc, char *argv[])
81  {
82      // false: dont set locale
83      // default value is true. but it changes the "usual" locale so the decimal
            separator in sscanf (CCrazyRadio.cpp) would be ',' instead of '.'
84      Gtk::Main app(argc, argv, false);
85
86      StereoCamera stereo(1);
87      Chessboard cb(6, 9, 0.026);
88      std::string dirStr("Cali/");
89
90      CalibrationGUI caliGui(stereo, dirStr, cb);
91
92      app.run(caliGui);
93
94      //dictionary with only 1 marker and a marker size of 4x4 bits
95      cv::aruco::Dictionary dictionary = cv::aruco::generateCustomDictionary(1,
```

```
        4);
96      stereo.initializeWorldFrame(dictionary, 0.197);
97
98      Regulator regulator = SENDMOTORRATIOS_LQ6DIM_LQ6DIM;
99      std::thread t(cntrThread);
100
101     std::vector<Eigen::Matrix<float,3,1>> vecSetpoints(8);
102     vecSetpoints[0] << 0.0,  0.0, 0.5;
103     vecSetpoints[1] << 0.5,  0.0, 0.5;
104     vecSetpoints[2] << 0.5, -0.5, 0.5;
105     vecSetpoints[3] << 0.0, -0.5, 0.5;
106
107     vecSetpoints[4] << 0.0, -0.5, 1.0;
108     vecSetpoints[5] << 0.5, -0.5, 1.0;
109     vecSetpoints[6] << 0.5,  0.0, 1.0;
110     vecSetpoints[7] << 0.0,  0.0, 1.0;
111
112     CCrazyRadio crRadio("radio://0/10/2M");
113     if(crRadio.startRadio())
114     {
115         CCrazyflie cflieCopter(&crRadio, regulator, stereo);
116
117         while(cflieCopter.cycle())  //cycle() sends the data
118         {
119             cflieCopter.setSendSetpoints(sendSetPoints);
120
121             if(followTrajectory)
122             {
123                 if(newTrajectory)
124                 {
125                     cflieCopter.setFollowTrajectoryPoints(vecSetpoints, false)
                            ;
126                     newTrajectory = false;
127                 }
128             }
129             else
130                 cflieCopter.setSendpointPosition(posX, posY, posZ);
131
132             if(shutdown)
133             {
134                 cflieCopter.stopMotors();
135                 sendSetPoints = false;
136                 break;
137             }
138         }
139     }
140     else
141         std::cout << "start radio failed" << std::endl;
142
```

```
143     if(t.joinable())
144         t.join();
145 
146     return 0;
147 }
```

Anhang B

Modifizierte Bewegungserkennung

Der Nachteil der in Abschnitt 3.2 vorgestellten Verfahren ist, dass die Robustheit der Algorithmen stark von den Lichtverhältnissen im Arbeitsbereich abhängen. Sollte man zum Beispiel keine Möglichkeit haben Sonneneinstrahlungen komplett abzuschirmen, so werden beide Trackingverfahren in der Regel scheitern. Deswegen wurde auch noch ein drittes Verfahren betrachtet, das auf der Erkennung von Bewegungen basiert. Die Bewegungserkennung kann man umsetzen, indem man das aktuelle Bild mit dem vorherigen Bild (das Referenzbild) vergleicht. Hat sich in der Szene etwas verändert, so haben die entsprechenden Pixel auch einen anderen Wert. Analog zum Verfahren 3.2.2, kann nun der Bereich bestimmt werden, in dem sich die meisten Pixelwerte verändert haben.

In unserer Anwendung sind wir aber gar nicht unbedingt an einer Bewegungserkennung interessiert. Schwebt zum Beispiel der Quadcopter an einem Punkt und bewegt sich somit idealerweise überhaupt nicht, so würde die Bewegungserkennung den Quadcopter auch nicht registrieren. Daher passen wir das Verfahren so an, dass nicht Bewegungen erkannt werden, sondern Objekte, die ursprünglich nicht im Arbeitsbereich vorhanden waren. Wir nehmen also am Anfang ein Referenzbild auf, auf dem sich der Quadcopter nicht befindet. Startet man nun das Trackingverfahren, so vergleicht man das aktuelle Bild immer mit dem Referenzbild anstatt mit dem vorherigen Bild. So wird der Quadcopter immer erkannt, selbst wenn er bewegungslos im Raum schwebt. Wir nennen dieses Verfahren modifizierte Bewegungserkennung, auch wenn es streng genommen gar keine Bewegungen wahrnimmt.

Der unten angegebene Algorithmus zeigt dieses Verfahren als Pseudo-Code. Zunächst wird das aktuelle Bild und das am Anfang aufgenommene Referenzbild der Methode übergeben. Anschließend wird die pixelweise Differenz dieser Bilder gebildet und der Betrag davon genommen. *OpenCV* stellt dafür die Methode *cv::absdiff* zur Verfügung. Mit *threshold* wird dann ein Schwarz-Weiß-Bild erzeugt. Als Schwellwert wurde in dieser Arbeit der Wert 40 verwendet um geringe Änderungen auf Grund von Messrauschen zuzulassen. In den Zeilen 4 und 5 werden die morphologischen Filter Erosion und Dilation angewendet. Erosion

(*cv::erode*) sorgt dafür, dass die weißen Flächen im Bild verkleinert werden bzw. komplett verschwinden, wenn sie schon sehr klein sind. In diesem Fall sollen sie dafür sorgen, dass Rauschsignale verschwinden. Mit der Dilation (*cv::dilate*) werden die weißen Flächen im Bild vergrößert. So kann man weiße Flächen, die eigentlich zusammenhängend sein sollten, es aber auf Grund von Rauschen nicht sind, wieder vereinigen. Abschließend wird nach der größten Kontur gesucht und angenommen, dass es sich dabei um den Crazyflie handelt. Genauere Informationen zu den morphologischen Filtern kann man zum Beispiel [8, S. 191ff.] entnehmen.

Dieses Verfahren ist in der Lage auch bei hellen Lichtverhältnissen den Quadcopter zu tracken. Allerdings sollte hier bedacht werden, dass sich im Arbeitsbereich sonst kaum etwas ändern sollte. Daher sollten Lichtverhältnisse auch konstant sein. Fällt zum Beispiel während dem Tracken ein Schatten in den Arbeitsbereich so wird dieser registriert und ist dann oftmals auch der größte Bereich einer Änderung. In der Praxis hat sich dieser Algorithmus aber noch als zu anfällig für verändernde Lichtverhältnisse gezeigt und war zudem relativ langsam. Bekommt man dies aber in den Griff, so wäre man nicht mehr auf die LED des Crazyflies angewiesen.

Algorithmus 7 Suche nach dem Bereich mit der größten Veränderungen.

```
1:  procedure findMotion(img, refimg)
2:      img ← absdiff(refimg, img)
3:      img ← threshold(img, 40, 255)
4:      img ← erode(img)
5:      img ← dilate(img)
6:      contours ← findContours(img)
7:      for each currentContour in contours do
8:          currentAera ← contourAera(currentContour)
9:          if currentAera > maxAera then
10:             bestContour ← currentContour
11:             maxAera ← currentAera
12:     if maxAera = 0 then
13:         return (false, 0)
14:     else
15:         rectangle ← boundingRect(bestContour)
16:         loc ← center(rectangle)
17:         return (true, loc)
```

Anhang C

Inhalt der beiliegenden CD

Die beiliegende CD enthält neben der PDF-Version dieser Arbeit, den bei dieser Arbeit entstandenen Quellcode sowie Abbildungen und Videos.

/MA_Matthias_Hoeger.pdf	PDF-Version dieser Arbeit
/Source-Code	Ordner, der den Source-Code enthält
/c-bootloader	Ordner, der den Bootloader enthält, der sich auf dem CF befindet
/crazyradio-firmware	Ordner, der die Firmware des Crazyradio enthält
/crazyflie-firmware	Ordner, der die Firmware des Crazyflie enthält
/python-client	Ordner, der den Python Client enthält
/c-client	Ordner, der den C/C++ - Client enthält
/Abbildungen_und_Videos	Ordner, der Abbildungen und Videos dieser Arbeit enthält

Literaturverzeichnis

[1] Bitcraze. *Loco positioning system*. Siehe `https://wiki.bitcraze.io/doc:lps:index`, Aufgerufen am 21.06.2016.

[2] Bitcraze. *LPS Node*. Siehe `https://wiki.bitcraze.io/projects:lps:node#configuring_the_node`, Aufgerufen am 23.06.2016.

[3] Bitcraze. *motors.c*. Siehe `https://github.com/bitcraze/crazyflie-firmware/blob/master/src/drivers/src/motors.c`, Aufgerufen am 26.06.2016.

[4] Bitcraze. *Bitcraze LPS: Second update*, April 2016. Siehe `https://www.youtube.com/watch?v=addbBgewzOU`, Aufgerufen am 21.06.2016.

[5] Bitcraze. *DWM1000 crazyflie autonomous flight first demo*, Januar 2016. Siehe `https://www.youtube.com/watch?v=HZC8yJE12r4`, Aufgerufen am 21.06.2016.

[6] Bitcraze. *Loco Positioning Getting Started with Early Access*, Juni 2016. Siehe `https://www.youtube.com/watch?v=eKgVNE0Ozo8`, Aufgerufen am 21.06.2016.

[7] Gary Rost Bradski and Adrian Kaehler. *Learning Opencv, 1st Edition*. O'Reilly Media, Inc., first edition, 2008.

[8] Wilhelm Burger and Mark James Burge. *Digitale Bildverarbeitung - Eine algorithmische Einführung mit Java, 3. Auflage*. X.media.press. Springer, 2015.

[9] decaWave. *SenSor DWM1000 Module*. Siehe `http://www.decawave.com/products/dwm1000-module`, Aufgerufen am 23.06.2016.

[10] Lars Grüne. Numerische Methoden für gewöhnliche Differentialgleichungen. Vorlesungsskript, 2010. Siehe `http://num.math.uni-bayreuth.de/de/team/Gruene_Lars/lecture_notes/num2/num2_4.pdf`, Aufgerufen am 30.06.2016.

[11] Lars Grüne. Mathematische Kontrolltheorie. Vorlesungsskript, 2014. Siehe `http://num.math.uni-bayreuth.de/de/team/Gruene_Lars/lecture_notes/mkt1/mkt1_3.pdf`, Aufgerufen am 23.05.2016.

[12] Richard Hartley and Andrew Zisserman. *Multiple View Geometry in Computer Vision.* Cambridge University Press, second edition, 2004.

[13] Nicholas J. Higham. *Functions of Matrices: Theory and Computation.* Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2008.

[14] Matthias Höger. Aspekte der Modellierung eines Quadcopters. Bachelorarbeit, Universität Bayreuth, 2014. Siehe `http://num.math.uni-bayreuth.de/de/thesis/2014/Hoeger_Matthias/BA_Matthias_Hoeger.pdf`, Aufgerufen am 23.05.2016.

[15] Matthias Höger. Regelung eines Quadcopters. Seminararbeit, Universität Bayreuth, 2015.

[16] Itseez. *calibration.cpp.* Siehe `https://github.com/Itseez/opencv/blob/master/modules/calib3d/src/calibration.cpp`, Aufgerufen am 11.06.2016.

[17] Itseez. *The OpenCV Reference Manual: Release 3.0.0-dev*, Juni 2014. Siehe `http://docs.opencv.org/3.0-beta/opencv2refman.pdf`, Aufgerufen am 23.05.2016.

[18] Keith Lantz. *A Linux C++ joystick object*, Oktober 2011. Siehe `http://www.keithlantz.net/2011/10/a-linux-c-joystick-object/`, Aufgerufen im Oktober 2015.

[19] Vasile Sima. *Algorithms for linear-quadratic optimization.* New York, NY: Marcel Dekker, 1996.

[20] Dan Simon. *Optimal State Estimation: Kalman, H Infinity, and Nonlinear Approaches.* Wiley-Interscience, 2006.

[21] Jan Winkler. *Crazyflie Nano C++ Client Library*, May 2013. Siehe `https://github.com/fairlight1337/libcflie`, Aufgerufen im Dezember 2015.

[22] x-io Technologies. *Open source IMU and AHRS algorithms*, Juli 2012. Siehe `http://www.x-io.co.uk/open-source-imu-and-ahrs-algorithms/`, Aufgerufen am 28.06.2016.

[23] Saeid Yazdani. *Installing OpenCV 3.1.0 on Ubuntu*, Januar 2016. Siehe `http://embedonix.com/articles/image-processing/installing-opencv-3-1-0-on-ubuntu/`, Aufgerufen im Januar 2016.

Erklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig und nur unter Verwendung der angegebenen Quellen und Hilfsmittel angefertigt habe.

Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen.

Bayreuth, den 30. Juni 2016

(Matthias Höger)