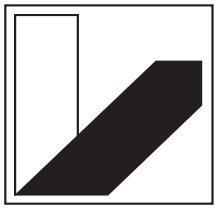


UNIVERSITÄT
BAYREUTH

Aspekte der mathematischen Modellierung eines Quadcopters

Bachelorarbeit

von

Matthias Höger

FAKULTÄT FÜR MATHEMATIK, PHYSIK UND INFORMATIK
MATHEMATISCHES INSTITUT

Datum: 30. September 2014

Betreuung:
Prof. Dr. L. Grüne
Dipl.-Math. T. Jahn

Inhaltsverzeichnis

Symbolverzeichnis

Modellkonstanten

b	Konstante, die vom Auftriebskoeffizienten des Quadrocopters 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 Quadrocopters
k	Konstante, die vom Strömungswiderstandskoeffizienten der Rotoren abhängt
m	Masse des Quadrocopters

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
$u = \begin{pmatrix} \omega_1 & \omega_2 & \omega_3 & \omega_4 \end{pmatrix}^T$	Kontrolle interpretiert als Winkelgeschwindigkeiten
$u = \begin{pmatrix} u_1 & u_2 & u_3 & u_4 \end{pmatrix}^T$	Kontrolle interpretiert als Raten

Parametrisierung

κ	Parameter für die Modellkonstanten
t_j	j-ter Messzeitpunkt
$\hat{x}_j$	gemessener Zustand zum Zeitpunkt t_j
$\hat{u}_j$	gemessene Kontrolle zum Zeitpunkt t_j
y_0	Parameter für den Startzustand
$y_\kappa(j)$	simulierter Zustand zum Zeitpunkt t_j

Sonstiges

$G(t)$	Ableitung der Lösung eines AWP nach dem Anfangswert
$G^p(t)$	Ableitung der Lösung eines parameterabhängigen AWP nach den Parametern
I_n	$n \times n$ Einheitsmatrix
0_n	$n \times n$ Nullmatrix

Kapitel 1

Einleitung

1.1 Crazyflie Nano Quadcopter

Im Bereich des Modellfluges findet sich inzwischen neben den klassischen Flugobjekten, dem Flugzeug und dem Helikopter, auch der Quadrocopter (engl.: quadcopter oder quadrotor) wieder. Dabei handelt es sich um ein Luftfahrzeug, das von vier in einer Ebene liegenden Rotoren betrieben wird.

Der von *Bitcraze*[1] entwickelte *Crazyflie Nano Quadcopter* ist ein vergleichsweise kleiner Vertreter dieser Klasse. Er wiegt lediglich circa 19 Gramm und hat eine Spannweite von etwa 90 Millimeter. Weitere Spezifikationen[2] des Modells sind unter anderem

- bis zu 7 Minuten Flugzeit mit dem Standard-Akku (170mAh Lithium-Polymer-Akkumulator)
- aufladbar per standardmäßigen Micro-USB-Kabel
- 32 Bit Mikrocontroller: STM32F103CB
- 3-Achsen Gyroskop
- 3-Achsen Accelerometer

Der große Vorteil dieses Modells ist, dass er von Entwickler für Entwickler erstellt wurde. Bei der gesamten Software, die im Zusammenhang mit dem *Crazyflie* steht, handelt es sich um Open Source. Der Quellcode, der in C und Python geschriebenen Programme, liegt also offen vor und kann somit auch vom Anwender verändert werden. Dies bietet natürlich optimale Voraussetzungen, um den Quadrocopter seinen eigenen Wünschen entsprechend zu modifizieren. Dabei ist man nicht nur auf sich alleine gestellt. *Bitcraze Wiki*[3] bietet bereits eine Menge an Informationen unter anderem über den *Crazyflie*. Sollte man dennoch

[1] 29.09.2014 http://www.bitcraze.se/
[2] 29.09.2014 http://www.bitcraze.se/crazyflie
[3] 29.09.2014 http://wiki.bitcraze.se/

noch ungeklärte Fragen haben, kann man sich auch über das *Bitcraze Forum*[4] direkt an die Entwickler und andere Forenmitglieder wenden. Nach eigener Erfahrung wird dort bereits innerhalb weniger Tage geantwortet.

Fliegen lässt sich der *Crazyflie* zum Beispiel mit einem USB-Controller. Dazu startet man den in Python geschriebenen Client, der unter anderem den Output des Controllers einliest und über eine USB-Antenne an den Quadrocopter schickt.

1.2 Ziel der Arbeit

Bei einem Quadrocopter handelt es sich um ein sehr instabiles Flugobjekt. Ohne einer entsprechenden Regelung ist er so gut wie gar nicht flugtauglich. Standardmäßig kommt beim *Crazyflie* ein PID-Regler (proportional integral derivative) zum Einsatz. PID-Regler sind die in der Industrie am meist verbreitetsten Feedback-Regler[5]. Gründe dafür sind zum Beispiel, dass sie meistens oft intuitiv verständlich sind und man auf ein mathematisches Modell des zu regelnden Systems verzichten kann. Letzteres ist aber wiederum auch ein Nachteil. Modellbasierte Regelungen, wie zum Beispiel die modellprädiktive Regelung, beziehen zusätzliche Informationen über die Dynamik des Systems mit ein, eben in Form eines mathematischen Modells, und können so ein Feedback mit besseren Eigenschaften erzeugen.

Hauptziel dieser Arbeit wird es nun sein, ein Modell für den *Crazyflie* aufzustellen. Dabei wird es vor allem um die Bestimmung der unbekannten Modellkonstanten gehen. Wenn dies gelingt, so kann man sich an die Realisierung von Alternativen zum PID-Regler wagen.

1.3 Aufbau

Im ersten Teil wird die Software des *Crazyflies* etwas näher betrachtet. In Kapitel 3 werden die mathematischen Grundlagen für diese Arbeit gelegt. Anschließend wird aufbauend auf den Ergebnissen einer Seminararbeit [4] ein mathematisches Modell für den *Crazyflie* aufgestellt. Dabei wird vor allem die Parametrisierung des Modells eine zentrale Rolle spielen. Zum Schluss werden die in der Arbeit erbrachten Ergebnisse noch einmal zusammengefasst.

[4] 29.09.2014 http://forum.bitcraze.se/index.php

[5] 29.09.2014 http://neutron.ing.ucv.ve/eiefile/Control%20I/Astrom_notas.pdf, S. 216

Kapitel 2

Crazyflie

2.1 Vorarbeiten der Entwicklungsumgebung

Um mit dem *Crazyflie* arbeiten zu können, muss zunächst die Entwicklungsumgebung eingerichtet werden. Im Folgenden wird darauf eingegangen, welche Schritte dafür unter Ubuntu 14.04 notwendig sind.

Die meisten Programme sind bereits im Repository enthalten. Daher kann man sie im Terminal mit dem Befehl

```
$ sudo apt-get -y install mercurial python2.7 python-usb python-pygame
      python-qt4 qt4-designer openocd build-essential
```

installieren.

Mercurial[1] ist ein verteiltes Versionskontrollsystem. Die Python-Programme werden für den Client benötigt, auf den in Abschnitt 2.2.2 näher eingegangen wird. *OpenOCD*[2] ist ein On-Chip Debugger und mit *build-essential*[3] lassen sich Debian-Pakete bauen.

Da der *STM32F103CB* ein ARM-basierter Mikrocontroller ist, wird auch die *GNU ARM Embedded Toolchain* benötigt. Es ist eine Sammlung von Cross-Compilern und Debuggern, mit der sich entsprechende Programme übersetzen lassen.[4]

In den offiziellen Paketquellen ist die *GNU ARM Embedded Toolchain* nicht enthalten. Man kann aber dem System ein PPA (Personal Package Archive) als Paketquelle hinzufügen, die diese bereitstellt

```
$ sudo add-apt-repository ppa:terry.guo/gcc-arm-embedded
```

Nach einem Update kann man nun den ARM-Compiler installieren:

[1] 29.09.2014 http://mercurial.selenic.com
[2] 29.09.2014 http://openocd.sourceforge.net
[3] 29.09.2014 http://packages.ubuntu.com/de/lucid/build-essential
[4] 29.09.2014 http://wiki.ubuntuusers.de/Archiv/GNU_ARM-Toolchain

```
$ sudo apt-get update
$ sudo apt-get install gcc-arm-none-eabi
```

Nun sind alle erforderlichen Programme installiert. Es wird sich aber zeigen, dass man zum Ansprechen der USB-Antenne Root-Rechte benötigt. Um die Programme nicht immer als Superuser aufrufen zu müssen, geben wir die USB-Antenne für den User frei. Dies kann man mit dem Programm *udev*[5] bewerkstelligen, das unter anderem für die Rechteverwaltung von Geräten zuständig ist.[6]

Dazu gibt man im Terminal

```
$ sudo groupadd plugdev
$ sudo usermod -a -G plugdev <username>
```

ein. Anschließend erstellt man eine Datei, in der man die Rechte für die USB-Antenne festlegt. Die Namen solcher Dateien sollten von der Form *xx-descriptive-name.rules* sein, wobei *xx* für eine Zahl steht.[7] Wir wollen unsere Datei *99-crazyradio.rules* nennen und erzeugen diese mit dem Befehl

```
$ sudo touch /etc/udev/rules.d/99-crazyradio.rules
```

Nun öffnen wir die noch leere Datei zum Beispiel mit dem Editor *Nano*

```
$ sudo nano /etc/udev/rules.d/99-crazyradio.rules
```

und fügen ihr folgende Zeile hinzu

```
SUBSYSTEM=="usb", ATTRS{idVendor}=="1915", ATTRS{idProduct}=="7777",
MODE="0664", GROUP="plugdev"
```

Mit *Strg-O* speichert man die Datei und mit *Strg-X* wird der Editor wieder verlassen.

Nach einem Neustart des Rechners kann man nun auch ohne Root-Rechte auf die USB-Antenne zugreifen.

Natürlich brauchen wir jetzt auch noch die Software, die von *Bitcraze* bereitgestellt wird. Dazu lädt man die Projekte *crazyflie-clients-python*, *crazyflie-firmware*, *crazyflie-bootloader* und *crazyradio-firmware* herunter, die auf folgenden Seiten zu finden sind:

- https://github.com/bitcraze/crazyflie-clients-python
- https://github.com/bitcraze/crazyflie-firmware
- https://github.com/bitcraze/crazyflie-bootloader

[5] 29.09.2014 http://manpages.ubuntu.com/manpages/karmic/en/man7/udev.7.html
[6] 29.09.2014 http://wiki.ubuntuusers.de/udev
[7] 29.09.2014 http://hackaday.com/2009/09/18/how-to-write-udev-rule

- https://github.com/bitcraze/crazyradio-firmware

Anschließend entpackt man die Dateien in einem beliebigen Ordner. Sie sollten aber alle im gleichen Verzeichnis liegen.

Um die C-Programme zu bearbeiten, werden wir die Entwicklungsumgebung *CodeBlocks* verwenden. Sollte diese noch nicht vorhanden sein, so installiert man sie mit dem Befehl

```
$ sudo apt-get install codeblocks
```

Nun muss ein Projekt erstellt werden:

- erzeuge im Crazyflie-Firmware Ordner ein leeres Projekt
- füge dem Projekt die einzelnen Source-Files der Crazyflie-Firmware hinzu
- spezifiziere das Makefile
  - Rechtsklick auf das Projekt
  - gehe zu *Properties...* → *Project settings*
  - setze einen Haken bei *This is a custom Makefile*
- passe das Build target *Debug* an
  - Rechtsklick auf das Projekt
  - gehe zu *Build options...* → *Debug* → *Make commands*
  - fülle die Felder wie folgt aus

Build project/target:	$make -f $makefile all
Clean project/target:	$make -f $makefile clean
Ask if rebuild is needed:	$make -q -f $makefile all

Um aus CodeBlocks heraus direkt eine neue Software auf den Quadrocopter spielen zu können, müssen wir ein zusätzliches Build target erstellen:

- erstelle zusätzliches Build target *cload*
  - Rechtsklick auf das Projekt
  - gehe zu *Properties...* → *Build Targets* → *Add*
  - erzeuge ein Build target mit dem Namen *cload*
- passe das Build target *cload* an
  - Rechtsklick auf das Projekt
  - gehe zu *Build options...* → *cload* → *Make commands*

– fülle die Felder wie folgt aus
  Build project/target: $make -f $makefile cload
  Clean project/target: $make -f $makefile clean
  Ask if rebuild is needed: $make -q -f $makefile cload

Ist das Build target *Debug* ausgewählt und kompiliert man das Projekt, so wird im Crazyflie-Firmware Ordner die Datei *cflie.bin* erstellt. Diese Datei wird auf den Crazyflie geladen, indem man das Build target auf *cload* umstellt und erneut kompiliert. Dabei ist folgendes zu beachten:

- natürlich muss die USB-Antenne an den Rechner angesteckt sein
- der Quadrocopter darf erst eingeschalten werden nachdem man auf *build* gedrückt hat
- der Crazyflie-Firmware Ordner und der Crazyflie-Client Ordner müssen sich im selben Verzeichnis befinden

Um den für diese Arbeit verwendeten Client zu starten, wechselt man im Terminal in das Verzeichnis des Crazyflie-Clients. Dann geht man eine Stufe weiter in den Ordner *lib*. Dort startet man das Main-Programm des Clients mit dem Befehl

```
$ python my_main.py
```

Dabei sollten schon die USB-Antenne angesteckt und der Quadrocopter eingeschaltet sein.

2.2 Software

Nachdem nun die Entwicklungsumgebung eingerichtet ist, können wir uns der Software widmen. Diese ist in vier Teile aufgegliedert:

- Bootloader
- Crazyradio-Firmware
- Crazyflie-Firmware
- Client

Mit dem Bootloader lässt sich die Firmware auf dem Crazyflie aktualisieren. Die Crazyradio-Firmware ist für die Kommunikation über die USB-Antenne verantwortlich. Auf die genaue Funktionsweise dieser beiden Programmteile wird nicht näher eingegangen, da dies für unser Anliegen nicht nötig ist.

Im Gegensatz dazu müssen wir die Crazyflie-Firmware und den Client zumindest ansatzweise etwas genauer betrachten. Aus ihnen erhalten wir die notwendigen Telemetriedaten.

2.2.1 Crazyflie-Firmware

Die Crazyflie-Firmware ist für alles zuständig, was sich direkt auf dem Quadrocopter abspielt. Hier werden zum Beispiel die Daten ausgelesen und weiterverarbeitet, die der Gyroskop und der Accelerometer liefern.

Abbildung 2.1 zeigt die grobe Struktur der Crazyflie-Firmware.

```
Folder description:
./              | Root, contains the Makefile
 + init         | Contains the main.c
 + config       | Configuration files
 + drivers      | Hardware driver layer
 |  + src       | Drivers source code
 |  + interface | Drivers header files. Interface to the HAL layer
 + hal          | Hardware abstaction layer
 |  + src       | HAL source code
 |  + interface | HAL header files. Interface with the other parts of the program
 + modules      | Firmware operating code and headers
 |  + src       | Firmware tasks source code and main.c
 |  + interface | Operating headers. Configure the firmware environement
 + utils        | Utils code. Implement utility block like the console.
 |  + src       | Utils source code
 |  + interface | Utils header files. Interface with the other parts of the program
 + scripts      | Misc. scripts for LD, OpenOCD, make, version control, ...
 |              | *** The two following folders contains the unmodified files ***
 + lib          | Libraries
 |  + FreeRTOS  | Source FreeRTOS folder. Cleaned up from the useless files
 |  + STM32F... | Library folder of the St STM32 peripheral lib
 |  + CMSIS     | Core abstraction layer
```

Abbildung 2.1: Aufbau der Crazyflie-Firmware (vgl. https://github.com/bitcraze/crazyflie-firmware)

Für die Parametrisierung eines mathematischen Modells eines Quadrocopters werden wir Telemetriedaten benötigen. Dazu werden wir nun die Funktionen betrachten, mit denen man die entsprechenden Zustände des *Crazyflies* abfragen und auch loggen kann. Außerdem wird gezeigt, wie man die einzelnen Motoren ansteuert.

Auslesen der IMU-Daten Über die Intertial Measurement Unit (IMU) gelangen wir an die Daten des Gyroskops und des Accelerometers. Dazu ruft man folgende Funktion auf, die in der Datei *hal/src/imu.c* deklariert ist

```
void imu9Read(Axis3f* gyroOut, Axis3f* accOut, Axis3f* magOut);
```

Bei den Übergabeparametern handelt es sich jeweils um einen Zeiger auf die in *hal/interface/imu_types.h* definierte Struktur *Axis3f*, die drei Float-Werte zusammenfasst: für jede

Koordinatenachse ein Wert.

Dereferenziert man nach dem Aufruf dieser Funktion *gyroOut* und *accOut*, so erhält man die Daten des Gyroskops in Grad pro Sekunde beziehungsweise des Accelerometers in Vielfachen der Erdbeschleunigung g. Bei den Beschleunigungswerten ist zu beachten, dass sie die Erdbeschleunigung enthalten. Das heißt, wenn der Quadrocopter still auf einer waagrechten Fläche liegt, so zeigt der Wert für die Beschleunigung in z-Richtung in etwa den Wert 1 an. Dies entspricht einer Beschleunigung von $1g$ nach oben. Verrechnet man dies mit der Erdbeschleunigung in negative z-Richtung, so erhält man die tatsächliche Beschleunigung von $0g$.

magOut würde die Daten des Magnetometers enthalten. Der vorliegende *Crazyflie* hat einen solchen aber nicht verbaut, weshalb dieser Zeiger in unserem Fall stets auf eine *Axis3f*-Variable verweist, deren Einträge alle Null sind.

Sensfusion In der Datei *moduls/src/sensfusion6.c* werden die Informationen des Gyroskops und des Accelerometers zusammengefasst, um daraus weitere Informationen abzuleiten.

Mit der Funktion

```
void sensfusion6GetEulerRPY(float* roll, float* pitch, float* yaw);
```

kann man die Euler-Winkel[8] Roll, Pitch und Yaw abfragen.

Die Funktion

```
float sensfusion6GetAccZWithoutGravity(const float ax, const float ay,
                                       const float az);
```

bekommt die drei Werte des Accelerometers übergeben und liefert die Beschleunigung des Quadrocopters in vertikaler Richtung (im Inertialsystem, vgl Kapitel 4) zurück. Dabei wurde der Anteil der Erdbeschleunigung bereits abgezogen. Ruft man diese Funktion auf, wenn der Quadrocopter still auf einer waagerechten Fläche liegt, so bekommt man den Wert 0 zurück. Der Wert wird hier auch wieder in Vielfache der Erdbeschleunigung g angegeben.

Commander Ein Quadrocopter wird in der Regel direkt von einem Anwender per Fernbedienung / Joystick gesteuert. Die Eingaben des Piloten können in der Datei *modules/src/commander.c* abgefragt werden.

Mit

```
void commanderGetThrust(uint16_t* thrust);
```

bekommt man die Information über den gewünschten Schub.

Dereferenziert man nach dem Aufruf der Funktion

[8] Auf die Euler-Winkel wird in Kapitel 4 näher eingegangen

```
void commanderGetRPY(float* eulerRollDesired,
                     float* eulerPitchDesired,
                     float* eulerYawDesired);
```

eulerRollDesired und *eulerPitchDesired*, so erhält man die vom Anwender gewünschten Roll- und Pitchwinkel. Dereferenzierung von *eulerYawDesired* liefert dagegen die gewünschte Winkeländerung des Yaws. Dies entspricht der typischen Steuerung eines Quadrocopters.

Motoren Die Motoren, die die einzelnen Rotoren antreiben, steuert man über die Funktion

```
void motorsSetRatio(int id, uint16_t ratio);
```

die in *drivers/src/motors.c* implementiert ist. Mit *id* gibt man an, welcher Motor angesprochen werden soll. *id* sollte dabei einen der in *drivers/interface/motors.h* vordefinierten Werte *MOTOR_M1* bis *MOTOR_M4* annehmen.

ratio gibt an, wie schnell sich der Motor beziehungsweise der Rotor drehen soll. Hat *ratio* den Wert 65535, so soll sich der Rotor mit der (unbekannten) maximalen Geschwindigkeit drehen. Bei einem Wert von 0 dreht sich der Rotor nicht.

Logging Anders als man vielleicht annehmen würde, wird nicht direkt auf dem *Crazyflie* entschieden, welche Daten zu welchen Zeitpunkten geloggt werden sollen. Dies geschieht in Wirklichkeit auf der Seite des Clients, auf den wir im nächsten Abschnitt näher eingehen werden.

Auf dem *Crazyflie* wird lediglich eine sogenannte *Table of Content* (*ToC*) angelegt, die alle Variablen enthält, die geloggt werden können. Diese Sammlung an Variablen kann man Dank der Makros in *moduls/interface/log* sehr leicht seinen eigenen Bedürfnissen anpassen. Dies wird nun an Hand eines Beispiels erklärt.

Angenommen wir wollen in *modules/src/stabilizer.c* die aktuellen Euler-Winkel Roll, Pitch und Yaw in die *ToC* aufnehmen. Der entsprechende Code könnte wie folgt aussehen

```
1  #include "log.h"
2
3  static float eulerRollActual;
4  static float eulerPitchActual;
5  static float eulerYawActual;
6
7  //diverse Funktionen
8  //...
9  //nach den Funktionen werden Variablen dem ToC hinzugefuegt
10
11 LOG_GROUP_START(euler)
12 LOG_ADD(LOG_FLOAT, roll, &eulerRollActual)
13 LOG_ADD(LOG_FLOAT, pitch, &eulerPitchActual)
```

```
14 LOG_ADD(LOG_FLOAT, yaw, &eulerYawActual)
15 LOG_GROUP_STOP(euler)
```

Zunächst muss die Header-Datei *log.h* eingebunden werden, damit man die dort definierten Makros verwenden kann. In den Zeilen 3 bis 5 werden hier die Variablen deklariert, die dem *ToC* hinzugefügt werden sollen. Hierbei ist zu beachten, dass diese Variablen global definiert werden müssen. Ob sie nun zusätzlich *static* sind oder nicht, spielt keine Rolle.

Nach allen Funktionen, also am Ende der Datei, beginnen wir nun, die gewünschten Variablen dem *ToC* hinzuzufügen. Dazu starten wir in Zeile 11 mit dem Makro *LOG_GROUP_START(<name>)* eine neue Gruppe, die den (beliebigen) Namen *euler* bekommt. In den Zeilen 12 bis 14 fügen wir dieser Gruppe mit der Anweisung *LOG_ADD(<typ>, <name>, <adresse>)* unsere globalen Variablen hinzu. *typ* muss dabei einen der in *modules/interface/log.h* vordefinierten Werte annehmen. *name* kann hier wieder frei gewählt werden und dient als eindeutige ID innerhalb dieser Gruppe. *adresse* enthält die Adresse der entsprechenden Variable.

Zum Schluss wird die Gruppe in Zeile 15 mit *LOG_GROUP_STOP(<name>)* wieder abgeschlossen, wobei *name* mit dem Namen übereinstimmen muss, den man der Gruppe beim Erzeugen in Zeile 11 zugewiesen hat.

Theoretisch kann man nun analog auch mehrere solche Gruppen erzeugen, sowohl im gleichen File als auch in verschiedenen Dateien. Dabei sollte darauf geachtet werden, dass man mehreren Gruppen nicht den gleichen Namen zuweist

2.2.2 Crazyflie Client

Der Client ist ein Python-Programm, das auf dem Rechner des Benutzers läuft. Seine Hauptaufgaben sind:

- Starten und Beenden der Funkverbindung zum *Crazyflie*
- Steuerung des Piloten (zum Beispiel Output eines USB-Controllers) einlesen und an *Crazyflie* weiterleiten

Zudem kann der Client aber auch Daten vom *Crazyflie* anfordern.

Wir werden hier nur darauf eingehen, wie man Flugdaten loggen und in eine Text-Datei abspeichern kann. Dazu betrachten wir das Main-File *my_main.py* des Clients, die sich im Ordner *lib* befindet. Wir werden als Beispiel die Euler-Winkel loggen, die, wie im Abschnitt 2.2 beschrieben, dem *ToC* hinzugefügt worden sind.

Zunächst wird im Konstruktor *__init__* der Klasse *StartCrazyflie* eine Textdatei erzeugt, die wir *euler.txt* nennen wollen, in der die Daten abgespeichert werden sollen.

```
self._fileHandle = open('euler.txt', 'a')
```

_fileHandle bezeichnet dabei die Variable, mit der wir auf die Datei zugreifen können. Mit *open* wird eine neue Textdatei mit dem entsprechenden Namen erzeugt, falls sie noch nicht existiert. Existiert bereits eine Datei mit diesem Namen, so bewirkt die Option *a*, dass alle folgenden Einträge an das Ende der Datei angefügt werden (engl.: append).

Wenn eine Verbindung zum *Crazyflie* erfolgreich aufgebaut worden ist, wird die Funktion *_connected* aufgerufen. Hier legt man fest, welche Daten mit welcher Periode geloggt werden sollen. Dazu wird ein Objekt der Klasse *LogConfig* mit

```
conf = LogConfig("Euler", period_in_ms)
```

erzeugt. Der erste Parameter steht für den Namen des LogConfig-Objekts und spielt eher keine Rolle. Der zweite Parameter ist schon wichtiger. Er gibt an, mit welcher Periode die Daten geloggt werden sollen. Dabei ist zu beachten, dass die Periode nur ganzzahlige Vielfache von 10ms sein können. (Im Konstruktor von LogConfig wird eine ganzzahlige Division von *period_in_ms* durch 10 durchgeführt.)

Nachdem nun das Objekt *conf* erstellt wurde, können wir ihm die Informationen über die zu loggenden Daten hinzufügen. Voraussetzung dafür ist, dass diese Daten auch im *ToC* liegen. In unserem Beispiel haben wir den *ToC* wie folgt um die Euler-Winkel erweitert.

```
1 LOG_GROUP_START(euler)
2 LOG_ADD(LOG_FLOAT, roll, &eulerRollActual)
3 LOG_ADD(LOG_FLOAT, pitch, &eulerPitchActual)
4 LOG_ADD(LOG_FLOAT, yaw, &eulerYawActual)
5 LOG_GROUP_STOP(euler)
```

Mit diesen Informationen können wir jetzt dem Objekt *conf* die Euler-Winkel hinzufügen

```
conf.add_variable("euler.roll", "float")
conf.add_variable("euler.pitch", "float")
conf.add_variable("euler.yaw", "float")
```

Der erste String-Parameter hängt davon ab, wie die entsprechenden Daten in der *ToC* angelegt worden sind. Der Teilstring vor dem Punkt (hier *euler*) ist identisch mit dem Gruppennamen (vgl. Programmcode Zeile 1). Der Teilstring nach dem Punkt (hier z.B. *roll*) entspricht der ID, mit der die entsprechende Variable identifiziert wird (vgl. Programmcode Zeile 2).

Der zweite String-Parameter gibt an, welchen Datentyp die zu speichernde Variable haben soll. Die erlaubten String-Werte sind in der Klasse *LogTocelement*, die sich in der Datei *lib/cflib/crazyflie/log.py* befindet, festgelegt. Man kann diesen Parameter auch weglassen. Dann wird die Variable in dem Datentyp abgespeichert, mit dem er in den *ToC* aufgenommen worden ist. In diesem Beispiel hätte man den zweiten Parameter also auch jeweils immer weglassen können.

Anschließend wird *conf* dem Crazyflie-Objekt *_cf* hinzugefügt.

```
self._cf.log.add_config(conf)
```

Für den Anwender kann es durchaus auch hilfreich sein, wenn man die Textdatei mit Titeln versieht, damit er weiß, welche Spalte für welche Variable steht. Dazu ruft man folgende Funktion auf:

```
self._print_title(self._fileHandle, conf)
```

Letztendlich muss noch dafür gesorgt werden, dass jedes Mal, wenn neue Daten vom Crazyflie ankommen, diese auch in das Textfile abgespeichert werden. Dies geschieht durch:

```
conf.data_received_cb.add_callback(lambda timestamp, data, log_conf:
self._print_tele(self._fileHandle, timestamp, data, log_conf))
```

add_callback erhält als Parameter eine Funktion, in unserem Fall eine Lambda-Funktion. Jedes Mal, wenn *Crazyflie* ein Paket losschickt und dieses beim Client ankommt, wird diese Lambda-Funktion aufgerufen. *timestamp* gibt dabei den Zeitpunkt an, wann das Paket losgeschickt worden ist. In *data* sind die Werte der laut *log_conf* zu loggenden Variablen zum Zeitpunkt *timestamp*. Die Lambda-Funktion leitet diese Parameter weiter an die Methode *_print_tele*, die schließlich die enthaltenen Daten in die durch *_fileHandle* gegebene Textdatei schreibt.

Wenn man nun die Verbindung zum Crazyflie beendet oder diese abbricht, so soll unser Handle *_fileHandle* auch wieder geschlossen werden. Dies bewerkstelligt man in der Funktion *_close_fileHandles*, indem man folgende Zeile hinzufügt

```
self._fileHandle.close()
```

Beachte: *Crazyflie* wird für ein so angelegtes LogConfig-Objekt in der jeweiligen Periode ein Paket mit den entsprechenden Informationen losschicken. Diese Pakete können aber nicht beliebig viele Daten fassen. Stattdessen muss man mit einem Speicher von 240 Bit auskommen. Möchte man trotzdem mehr Daten loggen, so muss man die einzelnen Variablen auf mehrere LogConfig-Objekte aufteilen.

Außerdem sei darauf aufmerksam gemacht, dass beim Zeitstempel *timestamp* bei zu langer Flugzeit auch die Grenzen des Datentyps überschritten werden und die Zeit quasi wieder bei 0 anfängt.

Kapitel 3

Mathematische Grundlagen

Bevor wir uns der Modellierung eines Quadrocopters widmen, benötigen wir zunächst einige mathematische Grundlagen, die in Kapitel 4 Anwendung finden werden.

3.1 Eigenschaften von Lösungen von Differentialgleichungen

In diesem Abschnitt werden wir Lösungen von parameterabhängigen Anfangswertproblemen betrachten.

Definition 3.1 Seien $n, k \in \mathbb{N}$ und $f : \mathbb{R} \times \mathbb{R}^n \times \mathbb{R}^k \to \mathbb{R}^n$ eine stetige Funktion.
Das vom Parameter $p \in \mathbb{R}^k$ abhängige Anfangswertproblem für die gewöhnliche Differentialgleichung

$$\frac{d}{dt}x(t) = f(t, x, p) \tag{3.1}$$

besteht darin, zu gegebenem $t_0 \in \mathbb{R}$ und $x_0 \in \mathbb{R}^n$ eine Lösungsfunktion $x(t)$ zu finden, die (3.1) erfüllt und für die zudem die Anfangsbedingung

$$x(t_0) = x_0(p) \tag{3.2}$$

gilt.

Das folgende Lemma gibt eine äquivalente Formulierung des Anfangswertproblems an. Diese Äquivalenz wird im weiteren Verlauf dieses Abschnitts sehr hilfreich sein.

Lemma 3.2 Sei $x : J \to \mathbb{R}^n$ stetig differenzierbar mit $J \subset \mathbb{R}$ offen. Dann sind äquivalent

- x löst das Anfangswertproblem (3.1), (3.2) für ein $t_0 \in J$, $x_0 \in \mathbb{R}^n$ und $p \in \mathbb{R}^k$
- x erfüllt für alle $t \in J$ die Integralgleichung

$$x(t) = x_0 + \int_{t_0}^{t} f(s, x(s), p)ds \tag{3.3}$$

Beweis Die Hinrichtung folgt direkt durch Integrieren von (3.1) bezüglich t unter Berücksichtigung der Anfangsbedingung (3.2). Differenzieren von (3.3) nach t liefert die Rückrichtung [3, S. 3].

Wir werden uns vor allem für die Ableitungen der Lösung eines parameterabhängigen Anfangswertproblems interessieren.

Definition 3.3 x löse das Anfangswertproblem (3.1), (3.2) für ein $t_0 \in J$, $x_0 \in \mathbb{R}^n$ und $p \in \mathbb{R}^k$. Dann definiert (3.4) die Ableitung von x nach dem Anfangswert x_0 und (3.5) die Ableitung nach dem Parameter

$$G(t; t_0, x_0, p) := \frac{\partial}{\partial x_0} x(t; t_0, x_0, p) \tag{3.4}$$

$$G^p(t; t_0, x_0, p) := \frac{\partial}{\partial p} x(t; t_0, x_0, p) \tag{3.5}$$

Wir werden zur besseren Übersicht statt $G(t; t_0, x_0, p)$ und $G^p(t; t_0, x_0, p)$ kurz $G(t)$ und $G^p(t)$ schreiben.

Satz 3.4 Es gilt:

$$G(t) = I_n + \int_{t_0}^{t} \frac{\partial}{\partial x} f(s, x(s; t_0, x_0, p), p) \cdot G(s)ds \tag{3.6}$$

$$G^p(t) = \frac{\partial}{\partial p} x_0 + \int_{t_0}^{t} \left(\frac{\partial}{\partial x} f(s, x(s; t_0, x_0, p), p) \cdot G^p(s) + \frac{\partial}{\partial p} f(s, x(s; t_0, x_0, p), p) \right) ds \tag{3.7}$$

Beweis Der Nachweis von (3.6) und (3.7) erfolgt durch differenzieren von x, das sich gemäß Lemma 3.2 durch eine Integralgleichung darstellen lässt.

Für die Ableitung nach dem Anfangswert x_0 gilt:

$$\begin{aligned} G(t) &= \frac{\partial}{\partial x_0} x(t; t_0, x_0, p) \\ &= \frac{\partial}{\partial x_0} \left(x_0 + \int_{t_0}^{t} f(s, x(s; t_0, x_0, p), p) ds \right) \\ &= I_n + \int_{t_0}^{t} \frac{\partial}{\partial x} f(s, x(s; t_0, x_0, p), p) \cdot G(s) ds \end{aligned}$$

Dabei bezeichnet I_n die $n \times n$ Einheitsmatrix.

Analog gilt für die Ableitung nach dem Parameter p

$$\begin{aligned} G^p(t) &= \frac{\partial}{\partial p} x(t; t_0, x_0, p) \\ &= \frac{\partial}{\partial p} \left(x_0 + \int_{t_0}^{t} f(s, x(s; t_0, x_0, p), p) ds \right) \\ &= \frac{\partial}{\partial p} x_0 + \int_{t_0}^{t} \left(\frac{\partial}{\partial x} f(s, x(s; t_0, x_0, p), p) \cdot G^p(s) + \frac{\partial}{\partial p} f(s, x(s; t_0, x_0, p), p) \right) ds \end{aligned}$$

Folgerung 3.5 Mit Lemma 3.2 folgt aus den Integralgleichungen (3.6) und (3.7) deren Äquivalenz zu den sogenannten *Variationsdifferentialgleichungen*

$$\frac{\partial}{\partial t} G(t) = \frac{\partial}{\partial x} f(t, x(t; t_0, x_0, p), p) \cdot G(t) \tag{3.8}$$

$$G(t_0) = I_n \tag{3.9}$$

und

$$\frac{\partial}{\partial t} G^p(t) = \frac{\partial}{\partial x} f(t, x(t; t_0, x_0, p), p) \cdot G^p(t) + \frac{\partial}{\partial p} f(t, x(t; t_0, x_0, p), p) \tag{3.10}$$

$$G^p(t_0) = \frac{\partial}{\partial p} x_0 \tag{3.11}$$

Folgender Satz gibt Auskunft über die Lösbarkeit der Variationsdifferentialgleichungen:

Satz 3.6 Es existiere eine lokal eindeutige Lösung x des Anfangswertproblems (3.1), (3.2) für ein $t_0 \in J$, $x_0 \in \mathbb{R}^n$ und $p \in \mathbb{R}^k$. Zudem sei f stetig differenzierbar. Dann existieren die Lösungen der Variationsdifferentialgleichungen (3.8), (3.9) sowie (3.10), (3.11) und sind lokal eindeutig.

Aus formellen Gründen werden für den Beweis des Satzes folgende zwei Abbildungen notwendig sein:

- Sei m eine natürliche Zahl. Die Funktion d_m ist wie folgt definiert

$$\begin{aligned} d_m : \mathbb{R}^{n\times n} &\to \mathbb{R}^{n\cdot m\times n\cdot m} \\ A &\mapsto \begin{pmatrix} A & 0_n & \dots & 0_n \\ 0_n & A & \dots & 0_n \\ \vdots & \vdots & \ddots & \vdots \\ 0_n & 0_n & \dots & A \end{pmatrix} \end{aligned}$$

  wobei 0_n die $n \times n$ Nullmatrix bezeichnet.

- Die Funktion v bildet eine Matrix B auf den Spaltenvektor ab, den man erhält, wenn man die Spalten von B untereinander schreibt

$$\begin{aligned} v : \mathbb{R}^{n\times m} &\to \mathbb{R}^{n\cdot m\times 1} \\ B = \begin{pmatrix} b_1 & \dots & b_m \end{pmatrix} &\mapsto \begin{pmatrix} b_1 \\ \vdots \\ b_m \end{pmatrix} \end{aligned}$$

  wobei $b_i \in \mathbb{R}^{n\times 1}$

Für $A \in \mathbb{R}^{n\times n}$ und $B, C \in \mathbb{R}^{n\times m}$ lässt sich leicht nachrechnen, dass folgende Rechenregeln gelten:

$$\begin{aligned} v(B + C) &= v(B) + v(C) \\ v(A \cdot B) &= d_m(A) \cdot v(B) \end{aligned}$$

Beweis (von Satz 3.6) Betrachte zunächst die Integralgleichung (3.6)

$$G(t) = I_n + \int_{t_0}^{t} \frac{\partial}{\partial x} f(s, x(s; t_0, x_0, p), p) \cdot G(s) ds$$

Wenn wir diese Matrizen als Vektoren interpretieren, indem wir die einzelnen Spalten un-

tereinander schreiben, so ist dies mit $G(t) \in \mathbb{R}^{n\times n}$ offensichtlich äquivalent zu

$$\begin{aligned} v\left(G(t)\right) &= v\left(I_n + \int_{t_0}^{t} \frac{\partial}{\partial x} f(s, x(s; t_0, x_0, p), p) \cdot G(s) ds\right) \\ &= v(I_n) + \int_{t_0}^{t} v\left(\frac{\partial}{\partial x} f(s, x(s; t_0, x_0, p), p) \cdot G(s)\right) ds \\ &= v(I_n) + \int_{t_0}^{t} d_n\left(\frac{\partial}{\partial x} f(s, x(s; t_0, x_0, p), p)\right) \cdot v\left(G(s)\right) ds \end{aligned}$$

Bringe alles auf eine Seite und definiere

$$F(t, v(G)) := v\left(G(t)\right) - v(I_n) - \int_{t_0}^{t} d_n\left(\frac{\partial}{\partial x} f(s, x(s; t_0, x_0, p), p)\right) \cdot v\left(G(s)\right) ds \tag{3.12}$$

$$= 0 \tag{3.13}$$

Es gilt

$$\frac{\partial}{\partial v(G)} F(t, v(G)) = I_{n\cdot n} - \int_{t_0}^{t} d_n\left(\frac{\partial}{\partial x} f(s, x(s; t_0, x_0, p), p)\right) ds$$

Nach Lemma 3.2 ist dies äquivalent zum Anfangswertproblem

$$\frac{\partial}{\partial t}\left(\frac{\partial}{\partial v(G)} F(t, v(G))\right) = -d_n\left(\frac{\partial}{\partial x} f(s, x(s; t_0, x_0, p), p)\right) \tag{3.14}$$

$$\frac{\partial}{\partial v(G)} F(t_0, v(G)) = I_{n\cdot n} \tag{3.15}$$

Der *Satz von Picard-Lindelöf* liefert nun die Existenz von $\tilde{\epsilon}, \tilde{\delta} > 0$, sodass in der Umgebung $\tilde{W} := \tilde{U} \times \tilde{V}$ von $(t_0, I_{n\cdot n})$ mit

$$\begin{aligned} \tilde{U} &:=]t_0 - \tilde{\epsilon}, t_0 + \tilde{\epsilon}[\\ \tilde{V} &:= \{M \in \mathbb{R}^{n^2\times n^2} : \|M - I_{n\cdot n}\| < \tilde{\delta}\} \end{aligned}$$

eine eindeutige Lösung $\frac{\partial F}{\partial v(G)} : \tilde{U} \to \tilde{V}$ existiert, die (3.14) und (3.15) erfüllt. $\|\cdot\|$ bezeichne dabei die Frobenius-Norm.

Da f stetig differenzierbar ist, ist $\frac{\partial F}{\partial v(G)}$ stetig. Weil zudem $\det(I_{n\cdot n}) = 1$ gilt, kann die Umgebung $\tilde{W} = \tilde{U} \times \tilde{V}$ auf eine Umgebung $W := U \times V$ von $(t_0, I_{n\cdot n})$ eingeschränkt werden, sodass für alle $t \in U$ gilt

$$\det\left(\frac{\partial}{\partial v(G)} F(t_0, v(G))\right) \neq 0$$

$\frac{\partial F}{\partial v(G)}$ ist also auf dem Intervall U regulär. Die Voraussetzungen des *Satzes über implizite Funktionen* sind also erfüllt und (3.12) lässt sich eindeutig nach $v(G(t))$ auflösen. Da v eine Bijektion ist, ist damit auch $G(t)$ eindeutig bestimmt.

Mit Lemma 3.2 folgt nun die eindeutige Existenz der Lösung $G(t)$ der Variationsdifferentialgleichung (3.8), (3.9) auf dem Intervall U.

Für die Variationsdifferentialgleichung (3.10) und (3.11) verfährt man analog.

Bemerkung Da die Variationsdifferentialgleichungen (3.8) - (3.11) von $x(t)$ abhängen, können G und G^p nur zeitgleich zur Lösung des ursprünglichen Anfangswertproblems berechnet werden.

3.2 Resultate aus der Kontrolltheorie

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 [2].

Definition 3.7 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 3.8 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 3.9 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 3.10 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

$$QA + A^T Q + M - (QB + R)N^{-1}(B^T Q + R^T) = 0 \tag{3.16}$$

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

Lemma 3.11 Falls das linear-quadratische optimale Steuerungsproblem eine optimale Wertefunktion der Form $V(x) = x^T Q x$ 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

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

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

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

Dann gilt:

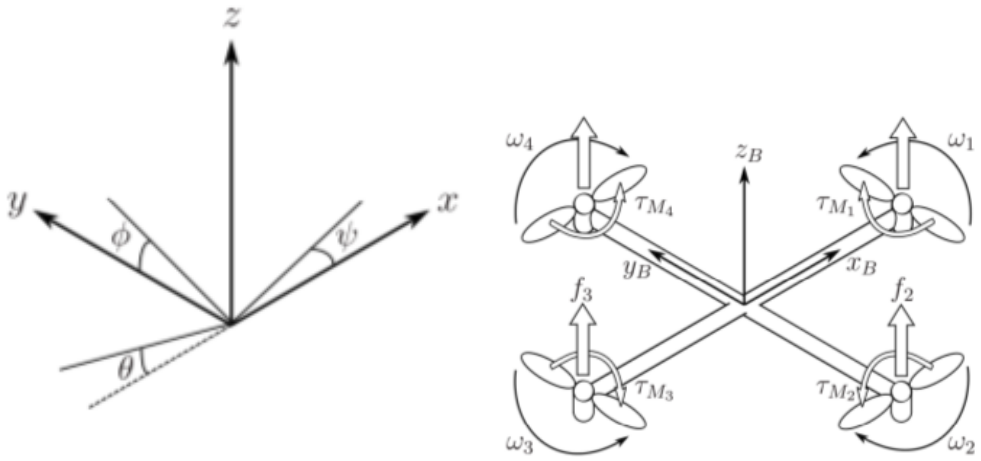
Ein lineares Feedback $F \in \mathbb{R}^{m \times n}$ stabilisiert den Nullpunkt $x^* = 0$ des nichtlinearen Kontrollsystems lokal exponentiell genau dann, wenn F die Linearisierung global exponentiell stabilisiert.

Kapitel 4

Mathematische Modellierung des Crazyflies

Nun liegen uns die entscheidenden Grundlagen vor und wir können uns der mathematischen Modellierung des *Crazyflies* widmen. Dazu betrachten wir zunächst kurz ein Modell eines Quadrocopters, das bereits in einer Seminararbeit hergeleitet worden ist [4].

4.1 Mathematisches Modell eines Quadcopters



(a) Inertialsystem (b) Hauptachsensystem

Abbildung 4.1: Bei der Modellierung werden zwei Koordinatensysteme verwendet

Bei der mathematischen Modellierung eines Quadrocopters kommen in der Regel zwei verschiedene Koordinatensysteme zum Einsatz. Zum einen wird ein raumfestes Koordinatensystem verwendet (vgl. 4.1(a)), das auch Inertialsystem genannt wird. Zum anderen hat

man ein Koordinatensystem, dessen Ursprung im Massenschwerpunkt des Quadrocopters liegt (vgl. 4.1(b)). Dieses System, auch Hauptachsensystem oder Body-Frame genannt, dreht und verschiebt sich mit dem Quadrocopter.

Im Inertialsystem werden neben der Position $\xi = \begin{pmatrix} x & y & z \end{pmatrix}^T \in \mathbb{R}^3$ und der Geschwindigkeit $v = \begin{pmatrix} v_x & v_y & v_z \end{pmatrix}^T \in \mathbb{R}^3$ des Flugobjekts auch die sogenannte Euler-Winkel $\eta = \begin{pmatrix} \phi & \theta & \psi \end{pmatrix}^T \in \mathbb{R}^3$ angegeben, mit denen man die Koordinatensysteme in das jeweils andere überführen kann. Dabei wird ϕ Roll-Winkel, θ Pitch-Winkel und ψ Yaw-Winkel genannt. Es ist zu beachten, dass es verschiedene Euler-Winkel-Konventionen gibt. Im Folgenden werden wir die *zyx-Konvention* benutzen, die in der Fahrzeugtechnik gebräuchlich ist [1].

Im Hauptachsensystem werden die Winkelgeschwindigkeiten $\nu = \begin{pmatrix} p & q & r \end{pmatrix}^T \in \mathbb{R}^3$ der Drehungen des Quadrocopters um die Achsen des Hauptachsensystems angegeben. Dabei gilt:

- p gibt die Winkelgeschwindigkeit der Drehung um die x_B-Achse (Bodyframe) an
- q gibt die Winkelgeschwindigkeit der Drehung um die y_B-Achse (Bodyframe) an
- r gibt die Winkelgeschwindigkeit der Drehung um die z_B-Achse (Bodyframe) an

Winkelgeschwindigkeit sind jeweils positiv, wenn man ausgehend vom Ursprung in Richtung der entsprechenden (positiven) Achse blickt und sich der Quadrocopter dabei im Uhrzeigersinn dreht.

Mit dem Zustand $x = \begin{pmatrix} \xi^T & v^T & \eta^T & \nu^T \end{pmatrix}^T \in \mathbb{R}^{12}$ und der Kontrolle $u = \begin{pmatrix} \omega_1 & \omega_2 & \omega_3 & \omega_4 \end{pmatrix}^T \in \mathbb{R}^4$, wobei ω_i für die Winkelgeschwindigkeit des Rotors i steht, kann man folgendes Kontrollsystem für einen Quadrocopter herleiten [4]

$$\dot{x} = f(x,u) = \begin{pmatrix} v \\ \frac{1}{m}(G + R_\eta T) \\ W_\eta^{-1}\nu \\ I^{-1}(-\nu \times I \cdot \nu - \Gamma + \tau) \end{pmatrix} \tag{4.1}$$

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 $C_\alpha := \cos(\alpha)$

[1] 29.09.2014 http://de.wikipedia.org/wiki/Eulersche_Winkel

- der Schubkraft $T = \begin{pmatrix} 0 \\ 0 \\ b(\omega_1^2 + \omega_2^2 + \omega_3^2 + \omega_4^2) \end{pmatrix}$
- 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 $\Gamma = I_r(\omega_1 - \omega_2 + \omega_3 - \omega_4) \begin{pmatrix} q \\ -p \\ 0 \end{pmatrix}$
- den externen Drehmomenten $\tau = \begin{pmatrix} 0 & -db & 0 & db \\ -db & 0 & db & 0 \\ -k & k & -k & k \end{pmatrix} \begin{pmatrix} \omega_1^2 \\ \omega_2^2 \\ \omega_3^2 \\ \omega_4^2 \end{pmatrix}$
- und den Modellkonstanten g, m, I_r, d, b, k sowie $I = \begin{pmatrix} I_x & 0 & 0 \\ 0 & I_y & 0 \\ 0 & 0 & I_z \end{pmatrix}$

Tabelle 4.1 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_r	$\mathrm{kg{\cdot}m^2}$	Trägheitsmoment der Rotoren
I_x, I_y, I_z	$\mathrm{kg{\cdot}m^2}$	Trägheitsmomente des Quadrocopters
b	kg·m	Konstante, die vom Auftriebskoeffizienten des Quadrocopters abhängt
k	$\mathrm{kg{\cdot}m^2}$	Konstante, die vom Strömungswiderstandskoeffizienten der Rotoren abhängt

Tabelle 4.1: Modellkonstanten und ihre Bedeutung

Die Rotationsmatrizen R_η bzw. W_η^{-1} bilden die Schubkraft T bzw. die Winkelgeschwindigkeiten ν vom Hauptachsensystem ins Inertialsystem ab. Der Zusammenhang dieser beiden

Matrizen wird erkennbar, wenn man beide herleitet.

Wie bereits erwähnt, verwenden wir für die Euler-Winkel die zyx-Konvention. Laut dieser erhält man das Hauptachsensystem ausgehend vom Inertialsystem wie folgt:[2]

1. Drehung um die z-Achse des Inertialsystems mit dem Yaw-Winkel ψ
   $\Rightarrow$ Rotationsmatrix $R_1(\psi)$
2. Drehung um die neue y-Achse mit dem Pitch-Winkel θ
   $\Rightarrow$ Rotationsmatrix $R_2(\theta)$
3. Drehung um die neue x-Achse mit dem Roll-Winkel ϕ
   $\Rightarrow$ Rotationsmatrix $R_3(\phi)$

Um nun also einen Vektor w im Inertialsystem ins Hauptachsensystem abzubilden, muss man diesen in der richtigen Reihenfolge dreimal rotieren:

$$\begin{aligned} w_b &= R_3(\phi)R_2(\theta)R_1(\psi)w \\ &= \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos(\phi) & \sin(\phi) \\ 0 & -\sin(\phi) & \cos(\phi) \end{pmatrix} \begin{pmatrix} \cos(\theta) & 0 & -\sin(\theta) \\ 0 & 1 & 0 \\ \sin(\theta) & 0 & \cos(\theta) \end{pmatrix} \begin{pmatrix} \cos(\psi) & \sin(\psi) & 0 \\ -\sin(\psi) & \cos(\psi) & 0 \\ 0 & 0 & 1 \end{pmatrix} w \end{aligned}$$

Die Umkehrung erhält man, wenn man die entsprechenden Inversen der Rotationsmatrizen von links multipliziert. Nach dem anschließenden Ausmultiplizieren erhält man die Rotationsmatrix R_η

$$\begin{aligned} w &= \begin{pmatrix} \cos(\psi) & \sin(\psi) & 0 \\ -\sin(\psi) & \cos(\psi) & 0 \\ 0 & 0 & 1 \end{pmatrix}^{-1} \begin{pmatrix} \cos(\theta) & 0 & -\sin(\theta) \\ 0 & 1 & 0 \\ \sin(\theta) & 0 & \cos(\theta) \end{pmatrix}^{-1} \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos(\phi) & \sin(\phi) \\ 0 & -\sin(\phi) & \cos(\phi) \end{pmatrix}^{-1} w_b \\ &= \begin{pmatrix} \cos(\psi) & -\sin(\psi) & 0 \\ \sin(\psi) & \cos(\psi) & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} \cos(\theta) & 0 & \sin(\theta) \\ 0 & 1 & 0 \\ -\sin(\theta) & 0 & \cos(\theta) \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos(\phi) & -\sin(\phi) \\ 0 & \sin(\phi) & \cos(\phi) \end{pmatrix} w_b \\ &=: R_\eta w_b \end{aligned}$$

Betrachten wir jetzt den Zusammenhang zwischen der Änderung der Euler-Winkel $\dot{\eta} = \begin{pmatrix} \dot{\phi} & \dot{\theta} & \dot{\psi} \end{pmatrix}^T$ und den Winkelgeschwindigkeiten ν im Hauptachsensystem. Wir haben gesehen, dass in der zyx-Konvention der Yaw-Winkel ψ noch zwei weitere Rotationen, der Pitch-Winkel θ einer weiteren Rotation und der Roll-Winkel ϕ keiner weiteren Rotation unterliegt. Für kleine Änderungen der Euler-Winkel kann man dann folgende Beziehung angeben [5, S. 71]

[2] 29.09.2014 http://de.wikipedia.org/wiki/Eulersche_Winkel

$$\begin{aligned}\nu &= R_3(\phi)R_2(\theta)\begin{pmatrix}0\\0\\\dot{\psi}\end{pmatrix} + R_3(\phi)\begin{pmatrix}0\\\dot{\theta}\\0\end{pmatrix} + \begin{pmatrix}\dot{\phi}\end{pmatrix}\\ &= \begin{pmatrix}1 & 0 & -\sin(\theta)\\ 0 & \cos(\phi) & \sin(\phi)\cos(\theta)\\ 0 & -\sin(\phi) & \cos(\phi)\cos(\theta)\end{pmatrix}\dot{\eta}\\ &=: W_\eta\dot{\eta}\end{aligned}$$

Invertiert man W_η, so erhält man schließlich die Rotationsmatrix, die die Winkelgeschwindigkeiten ν auf die Änderung der Euler-Winkel abbildet

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

4.2 Anpassen des Modells an Crazyflie

Wie wir bereits in Abschnitt 2.2.1 gesehen haben, können wir von *Crazyflie* die Winkelgeschwindigkeiten ν und die Euler-Winkel η abfragen. Dabei ist aber a priori weder die Orientierung des Hauptachsensystems, noch die verwendete Konvention der Euler-Winkel klar.

Die Ausrichtung des körperfesten Koordinatensystems bzw. des Gyro- und Accelerometers ergibt sich durch das entsprechende Datenblatt. Nach [7, S. 21] stimmt das Hauptachsensystem mit dem in Abbildung 4.1(b) gezeigten System überein.

Um Klarheit über die verwendete Euler-Winkel-Konvention zu erlangen, müssen wir nur feststellen, wie die Euler-Winkel im Sourcecode berechnet werden. In der Datei *moduls/-src/sensfusion6.c* zeigt sich, dass die Rotationsmatrix vom Inertial- ins Hauptachsensystem durch Quaternionen $q_0, ..., q_3$ mit der Eigenschaft $\sum_{i=0}^{3} q_i = 1$ repräsentiert werden. Aus diesen Quaternionen werden die Euler-Winkel wie folgt berechnet:

$$\begin{aligned}\text{roll} &= \text{atan2}\left(2(q_0q_1 + q_2q_3), q_0^2 - q_1^2 - q_2^2 + q_3^2\right)\\ \text{pitch} &= \text{asin}\left(2(q_1q_3 - q_0q_2)\right)\\ \text{yaw} &= \text{atan2}\left(2(q_0q_3 + q_1q_2, q_0^2 + q_1^2 - q_2^2 - q_3^2\right)\end{aligned}$$

Wie man [6, S. 39] entnehmen kann, entspricht dies fast der Berechnung der Euler-Winkel in der zyx-Konvention. Der Roll-Winkel ϕ und der Yaw-Winkel ψ stimmen überein, aber der Pitch-Winkel θ hat das falsche Vorzeichen. Man könnte das nun korrigieren, allerdings hätte dies auch Auswirkungen auf den PID-Regler, der momentan den Crazyflie stabilisiert. Um

diesen jetzt nicht auch anpassen zu müssen, werden wir daher die Berechnung der Euler-Winkel so lassen. Wenn wir dann aber den Pitch-Winkel abfragen und für unsere Zwecke benutzen wollen, müssen wir diesen dann natürlich mit -1 multiplizieren.

Die Euler-Winkel liegen uns also im Prinzip in der zyx-Konvention vor und wir können die Rotationsmatrizen R_η und W_η^{-1} benutzen, wie sie in Abschnitt 4.1 hergeleitet worden sind. Mit diesen Matrizen kann man rückwirkend auch überprüfen, ob es sich tatsächlich um diese Konvention gehandelt hat.

Dazu nimmt man in einem Testflug zum Beispiel die Winkelgeschwindigkeiten ν_j und die Euler-Winkel η_j zu den Zeitpunkten t_j auf. Nun vergleicht man die Differenzenquotienten $\frac{\eta_{j+1}-\eta_j}{t_{j+1}-t_j}$ mit den ins Inertialsystem transformierte Winkelgeschwindigkeiten $W_{\eta_j}^{-1}\nu_j$. Für den Datensatz aus Anhang A wurden diese Werte in Abbildung 4.2 eingetragen. Die über Differenzenquotienten approximierten Änderungen der Euler-Winkel (rot) stimmen gut mit den transformierten Winkelgeschwindigkeiten (blau) überein, das auf die korrekte Wahl der Transformationsmatrix W_η^{-1} schließen lässt. Dies bestätigt erneut die Verwendung der zyx-Konvention.

Kommen wir nun zur Kontrolle u. Wie wir im Abschnitt 2.2.1 gesehen haben, können wir Motor i auf eine bestimmte Rate, die wir u_i nennen werden, setzen. Diese Rate entspricht aber nicht der Winkelgeschwindigkeit der einzelnen Rotoren und damit nicht den einzelnen Komponenten der Kontrolle u aus dem System (4.1). Daher muss nun eine Beziehung zwischen diesen Raten u_i und den Winkelgeschwindigkeiten ω_i angegeben werden.

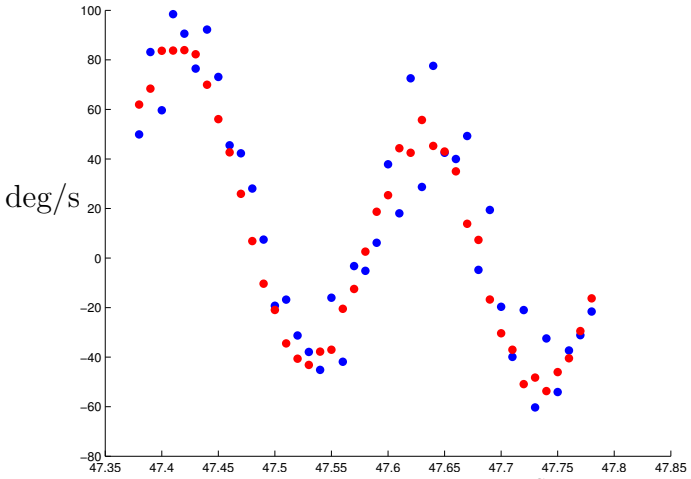
Annahme: Es besteht ein linearer Zusammenhang zwischen den Raten u_i und den Winkelgeschwindigkeiten ω_i.[3] Dazu existiert für alle Motoren eine gemeinsame Konstante $c > 0$, so dass gilt

$$\omega_i = cu_i, i = 1, ..., 4$$

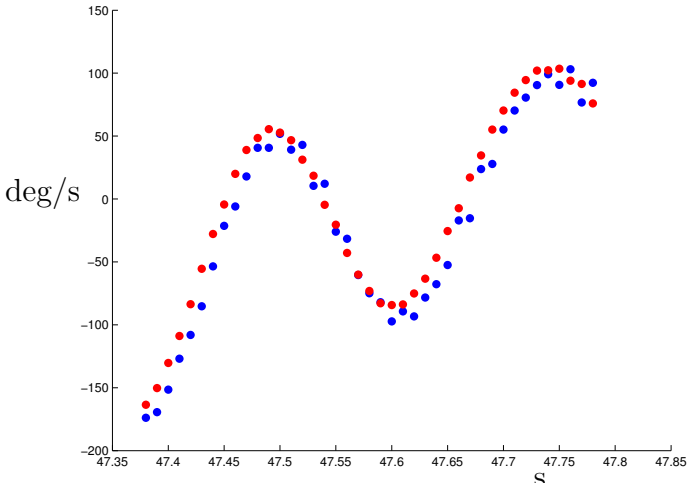
Ersetzen wir nun die Winkelgeschwindigkeiten durch diese Beziehung, so erhalten wir ein auf den Crazyflie zugeschnittenes Modell. Allerdings sind jetzt noch die Modellkonstanten unbekannt. Im folgenden Abschnitt werden wir sehen, wie wir diese bestimmen können.

Beachte: Es ist auch entscheidend, welchen Umlaufsinn die einzelnen Rotoren haben. Ein Sichttest ergibt, dass sich die Rotoren 1 und 3 entgegen dem Uhrzeigersinn und die Rotoren 2 und 4 im Uhrzeigersinn drehen (bei Betrachtung des Quadrocopters von oben). Dies entspricht also gerade der Umlaufrichtung, wie sie in Abbildung 4.1(b) zu sehen ist. Das Modell muss diesbezüglich also nicht weiter angepasst werden.

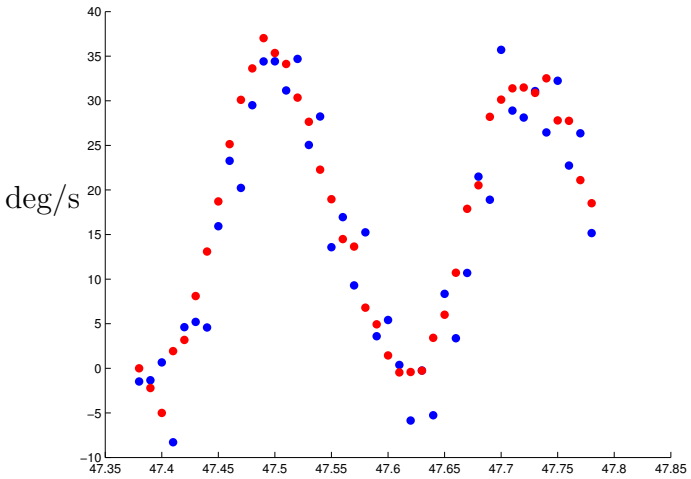
[3] 29.09.2014 http://forum.bitcraze.se/viewtopic.php?f=11&t=959



(a) Roll-Rate



(b) Pitch-Rate



(c) Yaw-Rate

Abbildung 4.2: Änderungen der Euler-Winkel. rot: approximiert durch Differenzenquotienten, blau: transformierte Winkelgeschwindigkeiten ν

4.3 Parametrisierung

Die Bestimmung von Modellkonstanten wird Parametrisierung genannt. Ein paar Konstanten können wir direkt durch Messungen angeben. So gelten für den Abstand d von Rotor zum Mittelpunkt, die Masse m des Quadrocopters und die Erdbeschleunigung g

$$d = 0.045 \text{ m} \tag{4.2}$$

$$m = 0.020 \text{ kg} \tag{4.3}$$

$$g = 9.81 \text{ ms}^{-2} \tag{4.4}$$

Um die restlichen Parameter $(b, c, k, I_r, I_x, I_y, I_z)$ zu bestimmen, werden wir die Methode der kleinsten Quadrate für nichtlineare Probleme verwenden. Dabei werden wir nicht auf das System (4.1) zurückgreifen, sondern betrachten zunächst das vereinfachte System

$$\dot{x} = f(x,u) = \begin{pmatrix} W_\eta^{-1}\nu \\ I^{-1}(-\nu \times I \cdot \nu - \Gamma + \tau) \end{pmatrix} \tag{4.5}$$

$$= \begin{pmatrix} p + \sin(\phi)\tan(\theta)q + \cos(\phi)\tan(\theta)r \\ \cos(\phi)q - \sin(\phi)r \\ \frac{\sin(\phi)}{\cos(\theta)}q + \frac{\cos(\phi)}{\cos(\theta)}r \\ I_x^{-1}(I_y - I_z)qr - I_x^{-1}I_r c(u_1 - u_2 + u_3 - u_4)q + I_x^{-1}dbc^2(u_4^2 - u_2^2) \\ I_y^{-1}(I_z - I_x)pr + I_y^{-1}I_r c(u_1 - u_2 + u_3 - u_4)p + I_y^{-1}dbc^2(u_3^2 - u_1^2) \\ I_z^{-1}(I_x - I_y)pq + I_z^{-1}kc^2(u_4^2 - u_3^2 + u_2^2 - u_1^2) \end{pmatrix} \tag{4.6}$$

mit dem Zustand $x = \begin{pmatrix}\eta^T & \nu^T\end{pmatrix}^T \in \mathbb{R}^6$ und der Kontrolle $u = \begin{pmatrix}u_1 & u_2 & u_3 & u_4\end{pmatrix}^T \in \mathbb{R}^4$. Der Grund für diese Einschränkung der Systemdynamik liegt darin, dass wir ohne weitere Hilfsmittel die Position ξ und Geschwindigkeit v des Quadrocopters nicht bestimmen können. Um aber eine sinnvolle Optimierung mittels kleinste Quadrate durchführen zu können, müssen wir die einzelnen Zustände messen können, was im System (4.5) mit dem Zustand $x = \begin{pmatrix}\eta^T & \nu^T\end{pmatrix}^T$ der Fall ist.

Das Prinzip der kleinsten Quadrate-Methode ist folgendes:
Man zeichnet in einem Testflug die einzelnen Zustände $\hat{x}_j = \begin{pmatrix}\eta_j^T & \nu_j^T\end{pmatrix}^T$, die Motorraten $\hat{u}_j \in \mathbb{R}^4$ sowie den jeweiligen Messzeitpunkt t_j auf. Auf diese Weise erhält man eine Menge von Tupeln $(\hat{x}_j, \hat{u}_j, t_j), j = 0, ..., N$. Nun löst man das Optimierungsproblem[4]

$$\min_{\kappa \in P} \sum_{j=0}^{N} \|\hat{x}_j - y_\kappa(j)\|^2$$

wobei P die Menge der zulässigen Parameter angibt und $y_\kappa(j)$ die simulierten Zustände zum Messzeitpunkt t_j sind.

[4]Bei der verwendeten Norm wird es sich immer um die 2-Norm handeln

Die simulierten Zustände erhält man wie folgt:

- Wähle einen Startzustand $y_\kappa(0) \in \mathbb{R}^6$
- Betrachte für $j = 0, ..., N-1$ das Anfangswertproblem

$$\dot{z} = f_\kappa(z, \hat{u}_j), z(t_j) = y_\kappa(j) \tag{4.7}$$

mit zugehöriger Lösung $z(t; t_j, y_\kappa(j), \hat{u}_j, \kappa)$ und definiere iterativ

$$y_\kappa(j+1) := z(t_{j+1}; t_j, y_\kappa(j), \hat{u}_j, \kappa) \tag{4.8}$$

Dabei bezeichne f_κ unsere Dynamik f aus (4.5), wobei dort der Abstand d durch den gemessenen Wert 0.045m ersetzt wird und die restlichen Modellkonstanten durch die einzelnen Komponenten des Parametervektors κ.

Als Startzustand $y_\kappa(0)$ kann man zum Beispiel den ersten gemessenen Zustand $\hat{x}_0$ nehmen. Da aber dieser auf Grund von Messungenauigkeiten auch fehlerbehaftet sein könnte, werden wir stattdessen unser Optimierungsproblem um den Parameter $y_0 \in \mathbb{R}^6$ erweitern und verwenden diesen dann als Startzustand $y_p(0)$.

Das Optimierungsproblem ist damit gegeben durch

$$\min_{y_0, \kappa} \|\hat{x}_0 - y_0\|^2 + \sum_{j=1}^{N} \|\hat{x}_j - y_\kappa(j)\|^2 \tag{4.9}$$

Um dieses Problem zu lösen, verwenden wir die von Matlab bereitgestellte Routine *lsqnonlin*. *lsqnonlin* löst nichtlineare kleinste Quadrate Probleme der Form

$$\min_p \|F(p)\|^2,$$

wobei $F : \mathbb{R}^n \to \mathbb{R}^m$ eine nichtlineare Zielfunktion ist.

Angepasst auf das Optimierungsproblem (4.9) muss F also wie folgt gewählt werden

$$F(\bar{p}) = \begin{pmatrix} \hat{x}_0 - y_0 \\ \hat{x}_1 - y_\kappa(1) \\ \vdots \\ \hat{x}_N - y_\kappa(N) \end{pmatrix}, \text{mit } \bar{p} = \begin{pmatrix} y_0 \\ \kappa \end{pmatrix} \tag{4.10}$$

Bevor wir nun versuchen das Minimierungsproblem zu lösen, betrachten wir erneut das System gegeben durch Gleichung (4.6), wobei die Modellkonstante d bereits fest gesetzt wurde. Offensichtlich erhält man ein komplett identisches System, wenn man zum Beispiel die Konstante c mit einem beliebigen, positiven Wert λ multipliziert und dafür die Konstante I_r durch λ sowie die Konstanten b und k durch λ^2 teilt. Insbesondere wird es keine eindeutige Lösung des Optimierungsproblems (4.9) geben.

Um dieses Problem zu umgehen, werden wir einzelne Modellkonstanten zusammenfassen. Dazu definieren wir

$$\begin{aligned} I_{r,c} &:= I_r c \\ b_c &:= bc^2 \\ k_c &:= kc^2 \end{aligned}$$

und ersetzen in Gleichung (4.6) die entsprechenden Produkte.

Tatsächlich haben wir damit aber das Problem nur verlagert. Man kann nun erneut die Modellkonstanten $I_x, I_y, I_z, I_{r,c}, b_c$ und k_c mit einem beliebigen, positiven Wert λ durch multiplizieren und erhält wieder eine identische Systemdynamik. Diesmal wollen wir nicht noch weitere Konstanten zusammenfassen, da dadurch auch der physikalische Sinn verloren geht. Der ist dahingehend wichtig für uns, weil wir anhand diesem abschätzen können, in welcher Größenordnung sich die Konstanten befinden sollten. Entsprechend kann man an der Lösung des Optimierers erkennen, ob diese brauchbar bzw. realistisch ist.

Stattdessen werden wir die Zielfunktion F in (4.10) um weitere Komponenten ergänzen. Das ursprüngliche System (4.1) gibt auch eine Differentialgleichung für die Geschwindigkeit in vertikaler Richtung (z-Achse des Inertialsystems) an

$$\begin{aligned} \dot{v}_z &= -g + bc^2 \cos(\phi)\cos(\theta)(u_1^2 + u_2^2 + u_3^2 + u_4^2) && (4.11)\\ &= -g + b_c \cos(\phi)\cos(\theta)(u_1^2 + u_2^2 + u_3^2 + u_4^2) && (4.12)\\ &=: f_a(x,u) && (4.13) \end{aligned}$$

mit $x = \begin{pmatrix} \phi & \theta & \psi & p & q & r \end{pmatrix}^T \in \mathbb{R}^6$ und $u = \begin{pmatrix} u_1 & u_2 & u_3 & u_4 \end{pmatrix}^T \in \mathbb{R}^4$.

$\dot{v}_z$ entspricht aber gerade der vertikalen Beschleunigung a im Inertialsystem, die wir beim *Crazyflie* nach Abschnitt 2.2.1 auch abfragen können.

Wir werden im Testflug also noch zusätzlich die vertikale Beschleunigung $\hat{a}_j \in \mathbb{R}$ zum Messzeitpunkt t_j für $j = 0, ..., N$ aufnehmen und ergänzen die Zielfunktion F in (4.10) zu

$$F(\bar{p}) = \begin{pmatrix} \hat{x}_0 - y_0 \\ \hat{x}_1 - y_\kappa(1) \\ \vdots \\ \hat{x}_N - y_\kappa(N) \\ \hat{a}_0 - h_\kappa(0) \\ \vdots \\ \hat{a}_N - h_\kappa(N) \end{pmatrix}, \text{mit } \bar{p} = \begin{pmatrix} y_0 \\ \kappa \end{pmatrix} \tag{4.14}$$

wobei h_κ definiert wird über die Funktion f_a aus Gleichung (4.13)

$$h_\kappa(j) := f_a(y_\kappa(j), \hat{u}_j), j = 0, ..., N$$

Beim Optimieren sollten diese zusätzlichen Komponenten von F den Parameter, der für die Konstante $b_c = bc^2$ steht, in der passenden Größenordnung fixieren und damit auch indirekt die Parameter für die Konstanten k_c und $I_{r,c}$.

Mit diesen Vorüberlegungen können wir schon versuchen das Optimierungsproblem

$$\min_{\bar{p}} \|F(\bar{p})\|^2$$

mit F und $\bar{p}$ aus Gleichung (4.14) zu lösen. Man kann lsqnonlin aber auch zusätzlich die Jacobi-Matrix J von F übergeben. Dadurch können die Berechnungen schneller und mit besserem Ergebnis durchgeführt werden.

Bei der Berechnung der Jacobi-Matrix ist zu beachten, dass F die Terme $y_\kappa(1), ..., y_\kappa(N)$ enthält. Diese Resultieren aber aus den Lösungen $z_{j-1}(t) := z(t; t_{j-1}, y_\kappa(j-1), \hat{u}_{j-1}, \kappa)$ von den Anfangswertproblemen (4.7), die sowohl von y_0 als auch von den Parametern κ abhängen.

Seien dazu für $j = 1, ..., N$

- $G_j(t) := \frac{\partial}{\partial y_0} z_{j-1}(t) \in \mathbb{R}^{6\times 6}$
  die Ableitung der simulierten Trajektorie nach dem Anfangswert y_0
- $G_j^\kappa(t) := \frac{\partial}{\partial \kappa} z_{j-1}(t) \in \mathbb{R}^{6\times 6}$
  die Ableitung der simulierten Trajektorie nach den Parametern κ
- $A_j(t) := \frac{\partial}{\partial y_0} f_a(z_{j-1}(t), \hat{u}_j) \in \mathbb{R}^{1\times 6}$
  die Ableitung der simulierten, vertikalen Beschleunigung nach dem Anfangswert y_0
- $A_j^\kappa(t) := \frac{\partial}{\partial \kappa} f_a(z_{j-1}(t), \hat{u}_j) \in \mathbb{R}^{1\times 6}$
  die Ableitung der simulierten, vertikalen Beschleunigung nach den Parametern κ

Mit $G_0(t_0) := I_6$, $G_0^\kappa(t_0) := 0_6$, $A_0(t_0) := \frac{\partial}{\partial y_0} f_a(y_0, \hat{u}_0)$ und $A_0^\kappa(t_0) := \frac{\partial}{\partial \kappa} f_a(y_0, \hat{u}_0)$ ist die Jacobi-Matrix J von F gegeben durch

$$J(\bar{p}) = -\begin{pmatrix} G_0(t_0) & G_0^\kappa(t_0) \\ G_1(t_1) & G_1^\kappa(t_1) \\ \vdots & \vdots \\ G_N(t_N) & G_N^\kappa(t_N) \\ A_0(t_0) & A_0^\kappa(t_0) \\ \vdots & \vdots \\ A_N(t_N) & A_N^\kappa(t_N) \end{pmatrix}, \text{mit } \bar{p} = \begin{pmatrix} y_0 \\ \kappa \end{pmatrix}$$

In Abschnitt 3.1 wurde gezeigt, wie man die Matrix-Trajektorien $G_j(t)$ und $G_j^\kappa(t)$ für $j > 0$ berechnen kann.[5] Mit diesen Ergebnissen kann man dann auch die Matrizen $A_j(t)$ und $A_j^\kappa(t)$

[5] Die Anfangswerte in (3.9) und (3.11) müssen dabei durch $G_{j-1}(t_j)$ und $G_{j-1}^\kappa(t_j)$ ersetzt werden

für $j > 0$ explizit angeben. Es gilt

$$\begin{aligned}
A_j(t) &= \frac{\partial}{\partial y_0} f_a(z_{j-1}(t), \hat{u}_{j-1}) \\
&= -\begin{pmatrix} \sin(\phi_{j-1}(t))\cos(\theta_{j-1}(t)) \\ \cos(\phi_{j-1}(t))\sin(\theta_{j-1}(t)) \end{pmatrix}^T G_j(t)_{1..2,1..6} b_c \|\hat{u}_{j-1}\|^2 \\
A_j^\kappa(t) &= \frac{\partial}{\partial \kappa} f_a(z_{j-1}(t), \hat{u}_{j-1}) \\
&= -\begin{pmatrix} \sin(\phi_{j-1}(t))\cos(\theta_{j-1}(t)) \\ \cos(\phi_{j-1}(t))\sin(\theta_{j-1}(t)) \end{pmatrix}^T G_j^\kappa(t)_{1..2,1..6} b_c \|\hat{u}_{j-1}\|^2 + \\
&+ \begin{pmatrix} \cos(\phi_{j-1}(t))\cos(\theta_{j-1}(t))\|\hat{u}_{j-1}\|^2 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix}^T
\end{aligned}$$

wobei $\phi_{j-1}(t)$ bzw. $\theta_{j-1}(t)$ die erste bzw zweite Komponente von $z_{j-1}(t)$ bezeichnet und die Notation $M_{n1..n2,m1..m2}$ die Teilmatrix einer Matrix M beschreibt, bei der man nur die Zeilen $n1$ bis $n2$ und Spalten $m1$ bis $m2$ betrachtet. Außerdem wurde hier verwendet, dass die erste Komponente des Parameters κ für die Modellkonstante b_c steht.

$A_0(t_0)$ und $A_0^\kappa(t_0)$ können direkt ausgerechnet werden, da sie nur von y_0 und $\hat{u}_0$, und damit insbesondere nicht von einer Lösung eines Anfangswertproblems abhängen.

Wir wollen uns nun die Ergebnisse von *lsqnonlin* betrachten. Der zugrunde liegende Datensatz ist Anhang A zu entnehmen. Es wurde stets der Algorithmus *trust-region-reflective* verwendet, mit dem *lsqnonlin* das Minimierungsproblem lösen soll, da dieser Algorithmus obere und untere Schranken für die Parameter zulässt. In diesem Fall wurde nur die untere Schranke für die Systemparameter κ auf 0 gesetzt.

Die Differentialgleichungen wurden mit *ode45* gelöst

Tabelle 4.2 zeigt die unterschiedlichen Ergebnisse, die man erhält, wenn man *lsqnonlin* mit bzw. ohne Jacobi-Matrix aufruft. Es ist zu erkennen, dass der Testlauf mit Übergabe der Jacobi-Matrix nicht nur ein etwas besseres Ergebnis liefert, sondern auch bedeutend schneller ist.

Abschnitt 3.1 hat aber gezeigt, dass G_j und G_j^κ nur zeitgleich zum ursprünglichen Anfangswertproblem (4.7) gelöst werden können. Das heißt, dass man in diesem Fall *lsqnonlin* die Jacobi-Matrix nur in Form eines Anfangswertproblems übergeben. Pro Funktionsauswertung müssen also anstatt N sechsdimensionale Differentialgleichungssysteme N 78-dimensionale Systeme gelöst werden. Tabelle 4.2 enthält eine Erklärung, warum trotz des erheblichen

Mehraufwandes die Laufzeit von *lsqnonlin* deutlich reduziert wird, wenn man zeitgleich die Jacobi-Matrix berechnen lässt.

Wenn *lsqnonlin* keine Informationen über die Jacobi-Matrix bekommt, so werden mehr Iterationen[6] und vor allem viel mehr Funktionsauswertungen[7] von F benötigt, um ein Ergebnis zu erzielen. Dies ist letztendlich für die längere Laufzeit verantwortlich.

Die große Differenz zwischen Funktionsauswertungen und Iterationen verdeutlicht, dass die Optimierungs-Routine ohne die Jacobi-Matrix oft neue Parameter vorschlägt, die das bisherige Ergebnis nicht verbessern.

Kommen wir nun zu den eigentlichen Resultaten der Parametrisierung. Tabelle 4.3 enthält neben den Startparametern auch das Ergebnis von der Routine *lsqnonlin*, der man die Jacobi-Matrix zur Verfügung gestellt hat. Der zugrunde liegende Datensatz ist wieder Anhang A zu entnehmen.

Um das Ergebnis zu bewerten, betrachten wir Abbildung 4.3. Abbildung 4.3(a) zeigt die vertikale Beschleunigung und Abbildung 4.3(b) die Euler-Winkel und Winkelgeschwindigkeiten. Dabei sind die gemessenen Zustände jeweils in blau und die simulierten Zustände in rot eingetragen.

In den Berechnungen werden die Winkel und Winkelgeschwindigkeiten in Rad bzw. Rad pro Sekunde gemessen. In der graphischen Darstellung werden dagegen die Einheiten Grad bzw. Grad pro Sekunde verwendet, da diese intuitiver sind. Entsprechend wurden die Beschleunigungswerte in Abbildung 4.3(a) ebenfalls mit $180/\pi$ multipliziert, damit die Plots vergleichbar bleiben.

Die Euler-Winkel (Graphiken auf der linken Seite von Abb. 4.3(b)) werden offensichtlich gut approximiert. Hier liegt nur eine Abweichung von maximal circa 3 Grad vor. Bei den Winkelgeschwindigkeiten (Graphiken auf der rechten Seite) wird die Charakteristik der gemessenen Werte auch relativ gut angenähert, wobei die Abweichungen betragsmäßig doch schon deutlich größer sind als wie bei den Euler-Winkeln.

Dagegen kann man in Abbildung 4.3(a) schon bei der gemessenen vertikalen Beschleunigung kaum ein Muster erkennen. Dies ist vielleicht auf starke Messungenauigkeiten zurückzuführen.[8] Dementsprechend kann man hier auch nicht erwarten, dass die simulierten Werte die gemessenen hinreichend gut approximieren. Für die Parametrisierung muss das aber nicht unbedingt ein schlechtes Zeichen sein. Dieser Zustand wurde ja nur eingeführt, um für den

[6] Der Iterationszähler wird nur erhöht, wenn *lsqnonlin* den aktuellen Parameter akzeptiert, d.h. die Zielfunktion ist mit diesem Parameter kleiner als mit dem bisher besten Parameter

[7] Der Zähler für die Funktionsauswertungen wird für jeden neuen Parameter erhöht, insbesondere auch dann, wenn der Parameter nicht akzeptiert wird.

[8] Unter der Annahme, dass die vertikale Beschleunigung stärker unter Messfehlern leidet, wurde bei den Berechnungen der Anteil dieses Zustandes in der Zielfunktion (4.14) durch 10 geteilt, damit er nicht so stark ins Gewicht fällt.

Parameter, der die Modellkonstante b_c repräsentiert, die passende Größenordnung zu finden.

Vor allem die geringe Abweichung der Euler-Winkel lässt darauf hoffen, dass der gefundene Parametersatz für eine Feedback-Regelung des *Crazyflies* hinreichend gut ist.

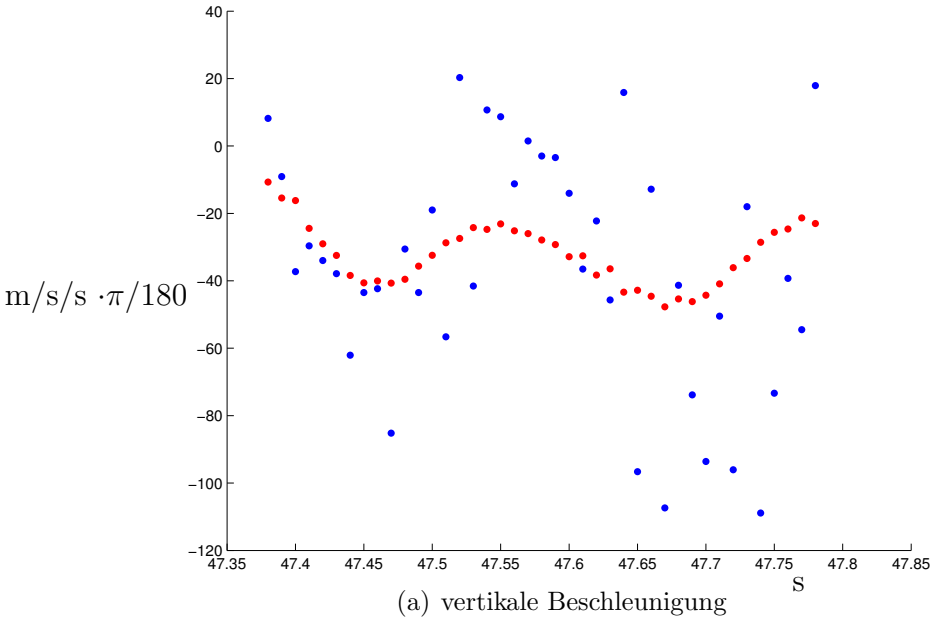
Anmerkung In Abschnitt 2.2.2 wurde erwähnt, dass die Variablen, die einem LogConfig-Objekt hinzugefügt werden, höchstens alle 10ms abgespeichert werden können. Der PID-Regler auf dem *Crazyflie* aktualisiert aber die Motorraten alle 2ms. Um dieses Defizit an Informationen zu umgehen, wurde die PID-Regelung so angepasst, dass sie auch nur noch alle 10ms neue Kontrollen an die Motren gibt.

Jacobi-Matrix	ja	nein
Iterationen	3	13
Funktionsauswertungen	4	182
Laufzeit in Sekunden	1.35	52.88
κ_1 ($\widehat{=} b_c$)	$2.5e^{-11}$	$1.7e^{-11}$
κ_2 ($\widehat{=} I_x$)	$6.7e^{-04}$	$6.6e^{-04}$
κ_3 ($\widehat{=} I_y$)	$3.4e^{-04}$	$3.3e^{-04}$
κ_4 ($\widehat{=} I_z$)	$7.7e^{-04}$	$1.4e^{-03}$
κ_5 ($\widehat{=} I_{r,c}$)	$2.9e^{-06}$	$2.8e^{-06}$
κ_6 ($\widehat{=} k_c$)	$1.1e^{-12}$	$2.6e^{-12}$
$\min_{\bar{p}} \|F(\bar{p})\|^2$	24.7	27.7

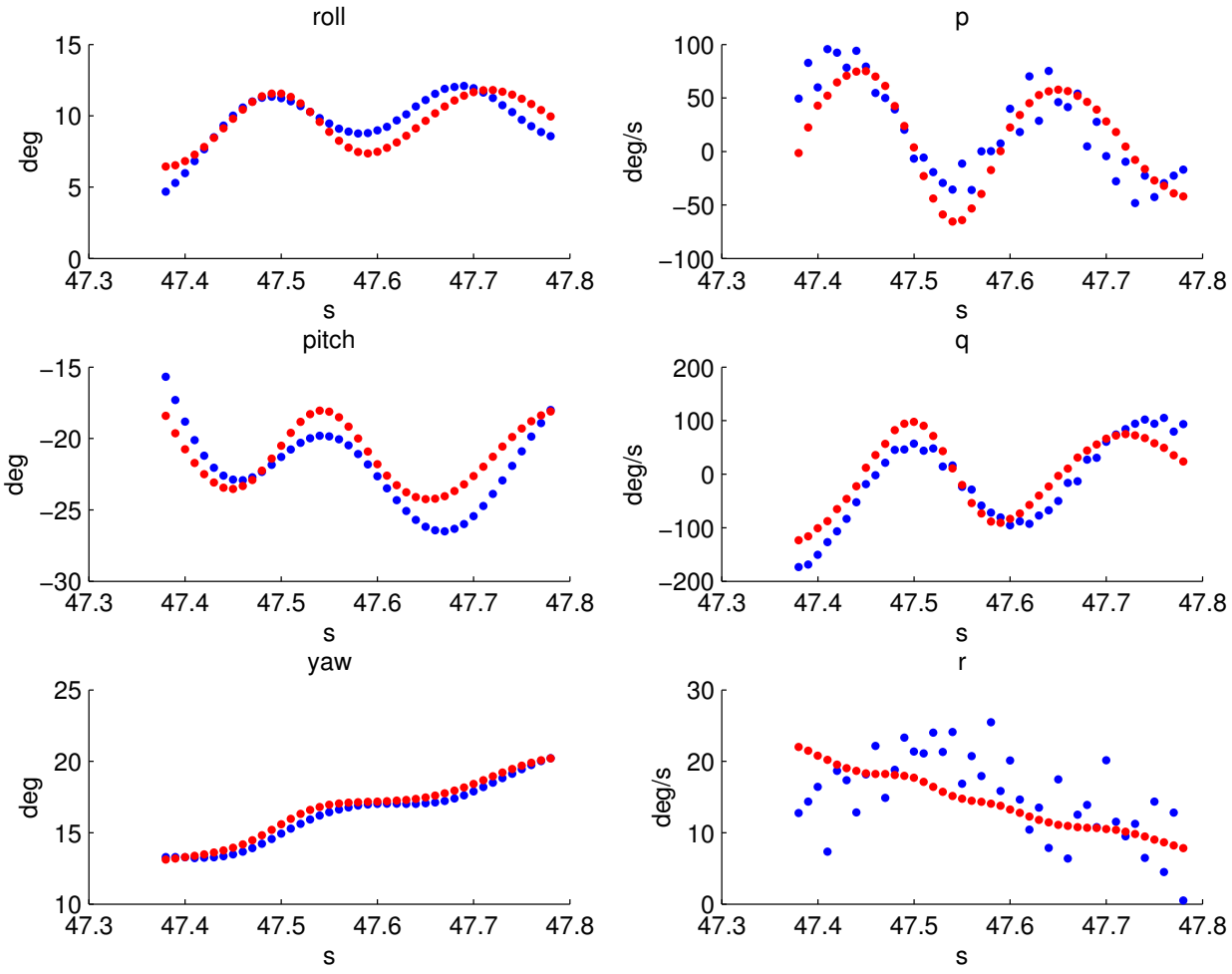
Tabelle 4.2: Vergleich von *lsqnonlin*: mit und ohne Übergabe der Jacobi-Matrix

Parameter	**Startwert**	**Ergebnis von *lsqnonlin***
κ_1 ($\widehat{=} b_c$)	$2.312088e^{-11}$	$2.523667957651499e^{-11}$
κ_2 ($\widehat{=} I_x$)	$7.895581ee^{-04}$	$6.702131924055546e^{-04}$
κ_3 ($\widehat{=} I_y$)	$2.928249e^{-04}$	$3.382571185081749e^{-04}$
κ_4 ($\widehat{=} I_z$)	$7.580681e^{-04}$	$7.689040163619214e^{-04}$
κ_5 ($\widehat{=} I_{r,c}$)	$2.896164e^{-06}$	$2.893229941196927e^{-06}$
κ_6 ($\widehat{=} k_c$)	$8.228593e^{-13}$	$1.103337935808926e^{-12}$

Tabelle 4.3: Ergebnis von *lsqnonlin*



(a) vertikale Beschleunigung



(b) links Euler-Winkel und rechts Winkelgeschwindigkeiten

Abbildung 4.3: Vergleich zwischen gemessenen Werten (blau) und simulierten Werten (rot)

4.4 Feedbackregelung des Crazyflie

Mit dem Ergebnis der Parametrisierung können wir nun ein lineares Feedback für das System (4.5) konstruieren.

Ein Gleichgewicht dieses Systems ist offensichtlich gegeben durch

$$x^* = \begin{pmatrix} \eta^* \\ 0 \end{pmatrix} \in \mathbb{R}^6, \eta^* \in \mathbb{R}^3 \text{ beliebig}$$

$$u^* = \begin{pmatrix} \lambda^* \\ \lambda^* \\ \lambda^* \\ \lambda^* \end{pmatrix} \in \mathbb{R}^4, \lambda^* \in \mathbb{R}^4 \text{ beliebig}$$

Satz (3.12) liefert eine Aussage darüber, wann ein lineares Feedback das System lokal exponentiell stabilisiert. Eine Voraussetzung des Satzes ist aber, dass das betrachtete Gleichgewicht in $(0, 0)$ liegt. Daher transformieren wir das System:

$$\tilde{f}(x, u) = f(x + x^*, u + u^*) \tag{4.15}$$

Die Linearisierung des Systems (4.15) ist gegeben durch die Matrizen

$$A = \frac{\partial}{\partial x} \tilde{f}(0, 0) \tag{4.16}$$

$$B = \frac{\partial}{\partial u} \tilde{f}(0, 0) \tag{4.17}$$

Wir setzen die Modellkonstanten auf die in der Parametrisierung gefundenen Werte:

$$\begin{aligned} d &= 0.045 \\ b_c &= 2.523667957651499e - 11 \\ I_x &= 6.702131924055546e - 04 \\ I_y &= 3.382571185081749e - 04 \\ I_z &= 7.689040163619214e - 04 \\ I_{r,c} &= 2.893229941196927e - 06 \\ k_c &= 1.103337935808926e - 12 \end{aligned}$$

Mit $\eta^* = \begin{pmatrix}\phi^* & \theta^* & \psi^*\end{pmatrix}^T$ gelten dann für A und B

$$A = \begin{pmatrix} 0 & 0 & 0 & 1 & \sin(\phi^*)\tan(\theta^*) & \cos(\phi^*)\tan(\theta^*) \\ 0 & 0 & 0 & 0 & \cos(\phi^*) & -\sin(\phi^*) \\ 0 & 0 & 0 & 0 & \frac{\sin(\phi^*)}{\cos(\theta^*)} & \frac{\cos(\phi^*)}{\cos(\theta^*)} \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

$$B = \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & -\frac{2}{I_x} db_c \lambda^* & 0 & \frac{2}{I_x} db_c \lambda^* \\ -\frac{2}{I_y} db_c \lambda^* & 0 & \frac{2}{I_y} db_c \lambda^* & 0 \\ -\frac{2}{I_z} k_c \lambda^* & \frac{2}{I_z} k_c \lambda^* & -\frac{2}{I_z} k_c \lambda^* & \frac{2}{I_z} k_c \lambda^* \end{pmatrix}$$

Nun müssen wir eine geeignete quadratische Kostenfunktion g finden. Mit der Notation aus Definition (3.8) setzen wir R gleich der Nullmatrix entsprechender Dimension und

$$M = 10^8 \cdot \begin{pmatrix} 300 & 0 & 0 & 0 & 0 & 0 \\ 0 & 300 & 0 & 0 & 0 & 0 \\ 0 & 0 & 300 & 0 & 0 & 0 \\ 0 & 0 & 0 & 5 & 0 & 0 \\ 0 & 0 & 0 & 0 & 5 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

$$N = \begin{pmatrix} 0.5 & 0 & 0 & 0 \\ 0 & 0.5 & 0 & 0 \\ 0 & 0 & 0.5 & 0 \\ 0 & 0 & 0 & 0.5 \end{pmatrix}$$

Dabei wurde M so gewählt, dass in erster Linie die Abweichungen der Euler-Winkel bestraft werden.

Lemma (3.11) gibt nun an, wie man ein lineares Feedback $\tilde{F}$ für das System (4.15) konstruiert:

$$\tilde{F} = -N^{-1} B^T Q$$

wobei Q die symmetrische und positiv definite Lösung der algebraischen Riccati-Gleichung (3.16) ist.

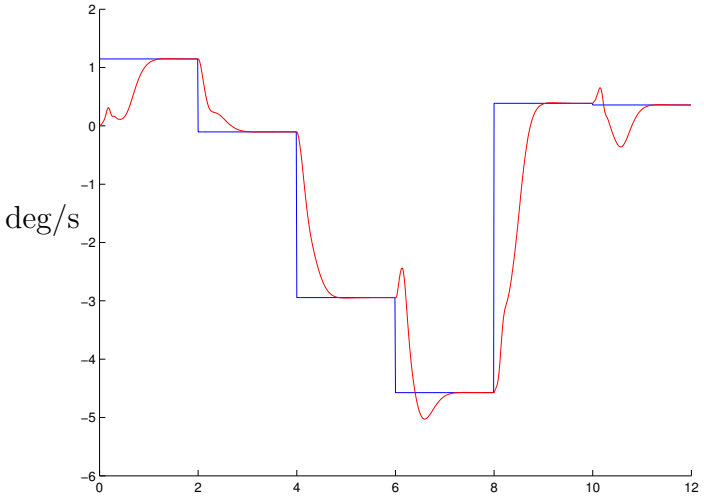
Aus dem aktuellen Zustand $\hat{x}$ des Quadrocopters und dem Feedback $\tilde{F}$ für das transformierte System konstruiert man die Kontrolle u für das ursprüngliche System (4.5) wie folgt:

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

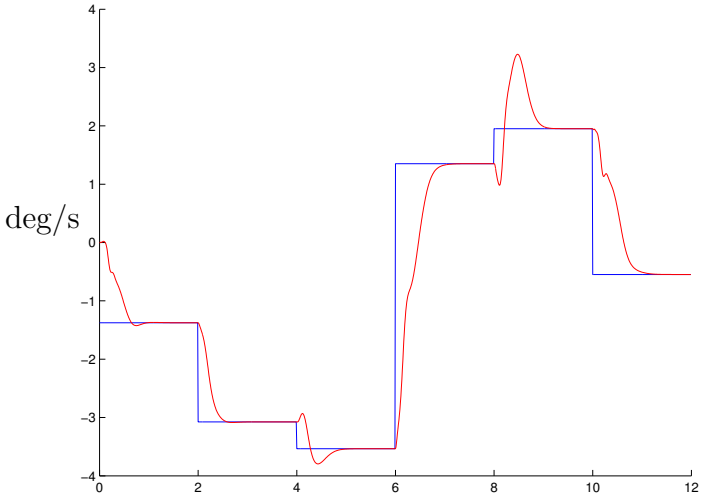
Mit der Matlab-Routine *lqr*, mit der man das Feedback $\tilde{F}$ berechnen kann, können nun Simulationen durchgeführt werden.[9] Abbildung 4.4 zeigt das Ergebnis einer dieser Tests. In rot ist der Verlauf der simulierten Euler-Winkel und in blau sind die gewünschten Euler-Winkel angetragen. Der Schub wurde hier Konstant gehalten.

In der Simulation passen sich die Euler-Winkel innerhalb einer Sekunde den vorgegebenen Werten an. Das Feedback scheint also zumindest für die angegebenen Modellkonstanten hinreichend gut zu sein. Ob sich damit auch der *Crazyflie* regeln lässt, kann nur ein Praxistest zeigen. Dieser steht allerdings noch aus.

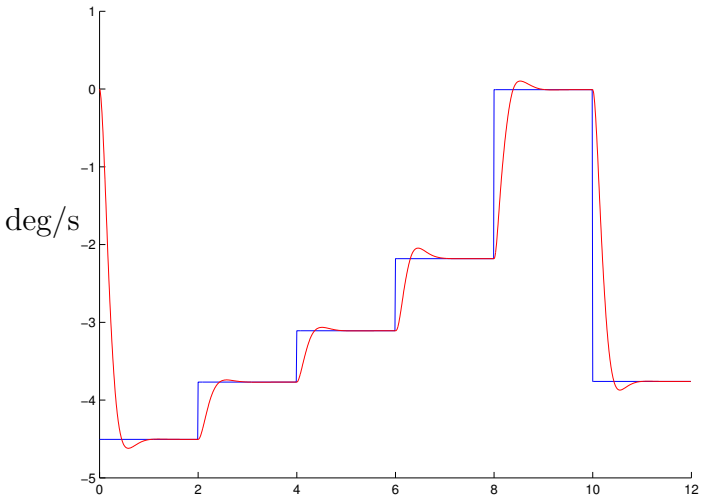
[9] Anhand der Ergebnisse der Simulationen wurde die Skalierung der Matrizen M und N gewählt



(a) Roll-Winkel



(b) Pitch-Winkel



(c) Yaw-Winkel

Abbildung 4.4: gewünschte Euler-Winkel (blau) und Simulierte Euler-Winkel (rot)

Kapitel 5

Fazit

5.1 Zusammenfassung

Am Anfang der Arbeit wurde auf wichtige Bestandteile der Software des *Crazyflies* eingegangen. Dies soll den Einstieg für Personen, die künftig mit dem Quadrocopter arbeiten wollen, erleichtern.

In Kapitel 3 haben wir die mathematischen Grundlagen für diese Arbeit gelegt. Dabei haben wir gesehen, wie man Lösungen von parameterabhängige Anfangswertproblemen nach dem Anfangswert und den Parametern ableitet. Wie sich später gezeigt hat, kann man mit diesen Kenntnissen Optimierungsprobleme erheblich schneller lösen. Im zweiten Teil dieses Kapitels wurden Resultate aus der Kontrolltheorie aufgelistet, die angeben, wann und wie man ein lineares Feedback konstruieren kann.

Im Hauptteil der Arbeit wurde zunächst ein mathematisches Modell eines Quadrocopters vorgestellt. Dabei wurden die Rotationsmatrizen, mit denen man die Zustände zwischen dem Inertial- und dem Hauptachsensystem transformieren kann, näher betrachtet. Anschließend ist das Modell auf den *Crazyflie* angepasst worden. Bei der Parametrisierung der Modellkonstanten sind wir unter anderem darauf eingegangen, wie man die Laufzeiten des Optimierers reduzieren kann. Die Ergebnisse der Parameterbestimmung wurde anhand von Grafiken bewertet und interpretiert. Zum Schluss wurde ein lineares Feedback, basierend auf die in Abschnitt (4.3) gefundenen Parameter, für den *Crazyflie* konstruiert.

5.2 Ausblick

Der nächste Schritt wird auf jeden Fall der Praxistest des linearen Feedbacks sein. Dabei muss noch geklärt werden, wie man diesen umsetzt. Das Feedback wurde in der Simulation mit Matlab berechnet. Die Kommunikation mit dem *Crazyflie* läuft aber über den Python-Client. Wenn man also weiterhin mit Matlab die Kontrollen berechnen will, so muss man dafür sorgen, dass Python und Matlab untereinander Daten austauschen können.

Ist dieses technische Problem gelöst, kann man sich wieder der Regelung widmen. In dieser Arbeit wurde nur auf die Feedback-Regelung des vereinfachten System (4.5) eingegangen. Eine weitere interessante Aufgabe ist die Positions-Regelung des *Crazyflies* über das System (4.1). Dazu wurde bereits ein zusätzliches LED am Quadrocopter angebracht, so dass man ihn in einem abgedunkelten Raum mit Kameras tracken kann. Mit dieser Regelung kann man zwei Ziele verfolgen: entweder man gibt eine Position vor, an dem der Quadrocopter schweben bzw. hin fliegen soll, oder man gibt eine Strecke vor, die er abfliegen soll. Dabei wird letzteres sicherlich eine größere Herausforderung darstellen.

Anhang A

Verwendeter Datensatz

Der für diese Arbeit verwendete Datensatz beruht auf den Log-Daten in den Dateien *9_acc_motor.txt* und *9_euler_gyro.txt*. Diese Daten wurden weiterverarbeitet und in *prepared_data.txt* abgespeichert. Die Daten in den Zeilen 3085 bis 3125 dieser Datei wurden für die Berechnungen dieser Arbeit verwendet.

Anhang B

Inhalt der beiliegenden CD

Die beiliegende CD enthält neben der PDF-Version dieser Arbeit alle Programme, die in diese Arbeit eingegangen sind. Zum einen handelt es sich dabei um die (zum Teil modifizierte) Software von *Bitcraze*. Zum anderen liegen die Matlab-Implementierungen bei, die zum Einsatz kamen. Außerdem befinden sich die Quellen als PDF-Dokumente auf der CD.

Im Folgenden ist ein Dateiverzeichnis der CD aufgeführt. Dabei werden im Bitcraze-Ordner nur Dateien aufgeführt, die verändert worden sind, oder die Funktionen enthalten, die in Abschnitt 2.2 vorgestellt worden sind..

/BA_Matthias_Hoeger.pdf	PDF-Version dieser Arbeit
/Bitcraze	Ordner, der die Software von Bitcraze enthält
/bootloader	Ordner, der den Bootloader enthält
/crazyradio-firmware	Ordner, der die Firmware des Crazyradio enthält
/crazyflie-firmware	Ordner, der die Firmware des Crazyflie enthält
/config/config.h	Datenrate auf 2Mbit/s gesetzt
/drivers/src/motors.c	enthält die Funktion *motorsSetRatio*
/hal/src/imu.c	enthält die Funktion *imu9Read*
/hal/src/radiolink.c	Datenrate auf 2Mbit/s gesetzt
/init/main.c	main-Funktion
/modules/interface/log.h	enthält die Makros zum anlegen des *ToC*
/modules/src/commander.c	enthält die Funktionen *commanderGetThrust* und *commanderGetRPY*
/modules/src/sensfusion6.c	enthält die Funktionen *sensfusion6GetEulerRPY* und *sensfusion6GetAccZWithoutGravity*
/modules/src/stabilizer.c	hier wird der Quadrocopter stabilisiert. Die geloggten Variablen stammen aus dieser Datei
/crazyflie-firmware-mod.cbp	CodeBlocks-Projektdatei
/crazyflie-clients-python	Ordner, der den Client enthält

/lib/my_main.py	main-Programm des Clients
/lib/9_acc_motor.txt	Log-Daten eines Testfluges
/lib/9_euler_gyro.txt	weitere Log-Daten des gleichen Testfluges
/Matlab	Ordner mit Matlab-Programmen
/9_acc_motor.txt	unbearbeitete Daten eines Testfluges
/9_euler_gyro.txt	weitere unbearbeitete Daten des gleichen Testfluges
/animation.m	animiert Quadrocopter (nur anhand von Euler-Winkel)
/calc_feedback.m	berechnet ein Feedback
/datafilehandling.m	liest Datensätze ein und verarbeitet sie weiter
/dynamics.m	enthält das mathematische Modell des Quadrocopters
/final_estimate_param.m	enthält lsqnonlin-Routine ohne Jacobi-Matrix
/final_estimate_param_jacobian.m	enthält lsqnonlin-Routine mit Jacobi-Matrix
/main.m	main-Programm für die Parametrisierung
/main_test_feedback.m	main test feedback
/prepared_data.txt	bearbeitete Daten
/verify_w_inv.m	Test, ob es sich bei den Euler-Winkel um die zyx-Konvention handelt
/visualize_gap.txt	für den Anwender gedacht: entspricht prepared_data.txt wobei Lücken in der Übertragung durch Leerzeilen dargestellt werden
/Quellen	Dieser Ordner enthält die angegebenen Quellen als PDF-Dokumente

Literaturverzeichnis

[1] KÖRKEL S.: *Numerische Mathematik I.* Vorlesungsmitschrift von S. Wolf und J. Vihharev, Universität Heidelberg, Wintersemester 2004/2005.

[2] GRÜNE L.: *Mathematische Kontrolltheorie.* Vorlesungsskript, Universität Bayreuth, Wintersemester 2013/2014

[3] GRÜNE L.: *Numerische Methoden für gewöhnliche Differentialgleichungen.* Vorlesungsskript, Universität Bayreuth, Sommersemester 2010

[4] PIRKELMANN S.: *Steuerung und Regelung eines Quadrocopters.* Seminararbeit, Universität Bayreuth, Sommersemester 2014

[5] HOVER F. S., TRIANTAFYLLOU M. S.: *System design for uncertainty.* Massachusetts Institute of Technology, 2009

[6] BERNER P.: *Technical Concepts Orientation, Rotation, Velocity and Acceleration, and the SRM.* 2008

[7] InvenSense Inc.: *MPU-6000 and MPU-6050 Product Specification Revision 3.2.* Document Number: PS-MPU-6000A-00, Release Date: 11.16.2011

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 30.09.2014

(Matthias Höger)