Universität Bremen

Faculty 4: Production Engineering

Institute of Space Systems

Masterthesis

Development and experimental Verification of a Guidance for a Planetary Landing Vehicle using a Quadro-Copter

Roman Grötzinger
Enrolment No.: 304 771 7

26.03.2018

Supervisor: Dr.-Ing. Stephan Theil
Second Assessor: Dr.-Ing. Andreas Rittweger

Kurzfassung

Die vorliegende Arbeit mit dem deutschen Titel "Entwicklung und experimentelle Überprüfung der Bahnregelung für ein Planetenlandefahrzeug mit einem Quadrocopter" hat die Zielsetzung das Testen der Algorithmen zur Bahnführung und -regelung von Raumfahrzeugen mit geringem Aufwand zu ermöglichen. Hierfür wurde durch das Deutsche Zentrum für Luft- und Raumfahrt in Bremen ein Android-basierter Quadrocopter angeschafft, der als Grundlage für die Implementierungen dient.

Zunächst wurden alle Schnittstellen implementiert, die nötig sind um Flugversuche in einer kontrollierten Laborumgebung zu absolvieren. Der Lageregler des Copters wurde weiterentwickelt, was die Präzision und das Ansprechverhalten der Drohne auf Kommandos deutlich verbessert. Für Positions- und Bahnverfolgung wurden übergelagerte Regelkreise in MATLAB/ Simulink entwickelt und umgesetzt. Ein Algorithmus, der das Bahnführungsproblem durch einen polynomiellen Ansatz löst, wurde implementiert und erfolgreich in verschiedenen Missionen des Quadrocopters getestet. Die Funktionsfähigkeit des Konzepts konnte damit belegt werden.

Die Implementierung in MATLAB/Simulink ermöglicht ein einfaches und schnelles Austauschen oder Erweitern von entwickelten Funktionalitäten, sodass dem Deutschen Zentrum für Luft- und Raumfahrt in Bremen in Zukunft günstige und einfache Testmöglichkeiten für die Entwicklungen von Regelungs- und Bahnführungssystemen zur Verfügung stehen.

Abstract

The objective of this thesis is to enable the testing of guidance algorithms for spacecraft applications with low effort.
The German Aerospace Center in Bremen has acquired an experimental Android-based quadro-copter for this purpose. This copter serves as the basis for all implementations in the frame of this thesis.

All interfaces required for copter flight operations in a controlled laboratory environment have been implemented as a prerequisite. The on-board attitude controller of the copter was further enhanced. This improved both, the accuracy and responsiveness of the copter to reference attitude angle inputs. Overlaid control loops for credible position and trajectory following were developed and realized in MATLAB/Simulink.
A polynomial guidance algorithm was implemented and successfully tested in various test missions for the copter. This all together proved the viability of the concept.

As the implementations are realised in MATLAB/Simulink, exchanging or expanding of functionalities is easily made possible. This subsequently provides the German Aerospace Center in Bremen with the possibility to test future developments of guidance and control systems in an easy and cost-effective way.

Table of Contents

List of Figures

List of Tables

Nomenclature

Symbol	Description	Unit
A	State matrix	–
a	Acceleration	m/s^2
B	Control matrix	–
C	Output matrix	–
e	State error	–
$G(s)$	Transfer function	–
g	Gravity acceleration	m/s^2
H	Pre-filter matrix	–
I	Identity matrix	–
i	Unit vector	–
I_α	Integral state of α	–
J	Cost function	–
K	Feedback gain matrix	–
l	Length specification	mm
k	Index of discrete time-step	–
m	Mass	kg
N	Number of samples	–
P	Solution of Algebraic Riccati Equation	–
p	Measured position	m
Q	Weighting matrix for state vector	–
R	Weighting matrix for input vector	–
r	Position vector	m
T	Thrust force	N
t	Time	s

Symbol	**Description**	**Unit**
Ts	Sample time	s
t_f	Arrival time	s
T_{i2b}	Transformation matrix	–
u	Control vector	–
v	Velocity	m/s
w	Reference input value	–
x	State vector	–
$\mathbf{0}$	Zero matrix	–
κ	Reference point index	–
ϕ	Roll angle	rad
θ	Pitch angle	rad
ψ	Yaw angle	rad
$\dot{()}$	First time derivative	$()/s$
$\ddot{()}$	Second time derivative	$()/s^2$
$()_{ref}$	Reference value	–
$()_{corr}$	Corrective values	–
$()_0$	Initial value	–
$()_f$	Final value	–
$()_{CR}$	Crossrange portion of vector	–
$()_{DR}$	Downrange portion of vector	–
$()_r$	Room coordinate reference frame	–
$()_b$	Body fixed coordinate reference frame	–

1 Introduction

1.1 Setting

This thesis was realised at the German Aerospace Center (DLR) Bremen Institute of Space Systems in the Guidance Navigation and Control (GNC) Systems department. At the University of Bremen, the Institute of Space Systems is organisationally assigned to the faculty 4, Production Engineering.
To make future space missions possible or to improve existing technologies in terms of performance, the Institute of Space Systems conducts research into relevant system technologies with a focus on the behaviour and influence of cryogenic fuels in tanks, landing technologies, attitude and orbit control systems, avionics systems and high-precision optical measurement systems. [1]
The GNC department has recently acquired an experimental quadro-copter from the University of Bremen research group: Parallel Computing for Embedded Sensor Systems. Its characteristics are described in further detail in Chapter 1.3. With this drone, the experimental testing of guidance algorithms developed for landing vehicles shall be enabled in an easy and cost-effective way with the results of this thesis.

1.2 Objectives

The top-level objectives of this thesis are:

- Implementation of a computer-based remote control for an experimental quadro-copter
- Development of the guidance for landing
- Performing and evaluating test flights in a lab environment

With the given hard- and software requirements in the frame of the thesis, these objectives translate to the following work packages:

- Sending data from a running Simulink simulation to an application on an Android device connected to the PC via USB.
- Implementing a USB remote controller (joypad) as a human interface device (HID) data source in Simulink
- Extending the existing "Flight" application (Android) provided with the drone for the use with the USB remote controller
- Fix optical tracking markers on the quadro-copter so that the position and attitude of the drone can be reliably determined by a tracking system
- Implement the tracking system as a continuous data source for a Simulink model running on a PC
- Implementing a simple position-hold controller in Simulink (proof of concept for the closed loop system)
- Integrating trajectory-following by target-point guidance
- Include measures to ensure safe flight-test execution
- Verify all implementations by performing autonomous test flights in the lab

1.3 The Quadro-Copter

The copter used in this thesis is based on the DJI Flame Wheel F450 Almost Ready to Fly (ARF) bare-bone drone platform, shown in Figure 1.1 below. As the kit does not come with a flight controller, the platform is equipped with a non-rooted Android smartphone and an Arduino Nano. The Android phone controls the four independent propellers by sending four pulse width (PW) commands to the Arduino Nano. These commands are forwarded from the Arduino to the electronic speed controllers (ESCs) which directly set the voltages of the brushless motors to which the propellers are attached. [2]
In Chapter 1.3.1 the used hardware is described in further detail.

Figure 1.1: The DJI Flame Wheel F450 ARF Kit [3]

As the drone utilises the Android device's built-in sensors, the body-fixed coordinate system of the quadro-copter is directly derived from Androids Sensor Application Programming Interface (API) coordinate system definition: when a device is held in its default orientation (portrait mode with home-button down) as shown in Figure 1.2, the X-axis is horizontal and points to the right, the Y-axis is vertical and points up, and the Z-axis points toward the outside of the screen face. [4]
The software framework utilized in the Android implementations has been developed by the Parallel Computing for Embedded Sensor Systems (PCESS) research group of the University of Bremen[1]. The group, under the lead of Prof. Dr. Matthew Holzel, is a collaboration between the DLR and the University of Bremen.
Chapter 1.3.2 describes the software framework, on which the work of this thesis is based, in further detail.

[1]The website of the research group is accessible via: `http://www.pcessflight.com` (as of March 2018)

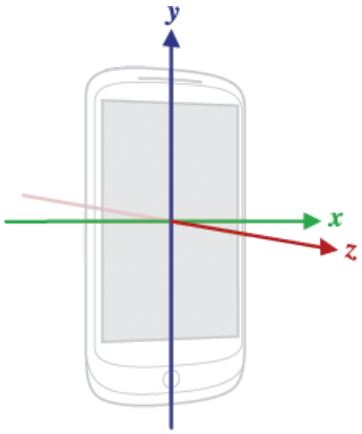


Figure 1.2: Android Sensor API coordinate system (relative to device) [4]

1.3.1 Hardware

As already mentioned above, the drone is based on the F450 airframe kit from the popular drone-manufacturer DJI. The kit is aimed at developers and offers a starting point for custom quadro-copter controller implementations, as it consists of the pure basics: the airframe itself, the motors, the ESCs and the propellers[2]. The basic specifications are shown in Table 1.1 below.

Table 1.1: DJI F450 basic specifications [5]

Model	Flame Wheel 450 (F450)
Frame Weight	282g
Diagonal Wheelbase	450mm
Takeoff Weight	800g - 1600g
Propeller	24 x 12.7cm (Diameter / Thread Pitch)
Battery	3S LiPo
Motor	23 × 12mm
ESC	15A OPTO

On the bare airframe the on-board Android phone is centrally fixed on the top plate using velcro. This allows for easy accessibility of the touch screen. To forward the PW commands to the ESCs, the Arduino Nano, placed in a protective casing, is fixed to the lower plate of the airframe also using velcro. The wiring of the ESCs to the

[2]DJI 9450 Self-tightening Propellers for the "DJI Phantom 3" have shown to serve as replacements and are used in the frame of this thesis

Arduino board is schematically shown in Figure 1.3 below. The PWM pins D6, D9, D10, D11 on the Arduino Nano are used respectively for motors f1, f2, f3, f4[3]. In the lower compartment of the airframe the 3S LiPo battery is accommodated and held in place by a velcro lace.

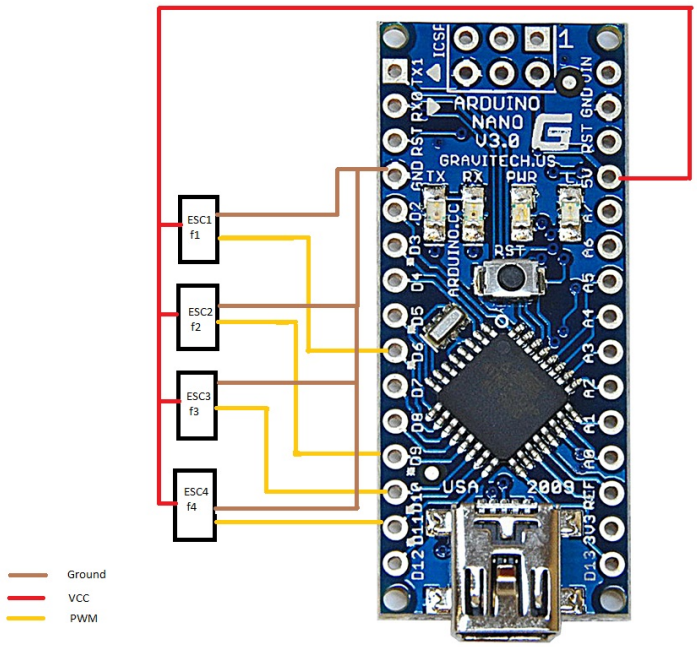


Figure 1.3: The wiring between the ESCs and the Arduino Nano board [6]

For additional safety, a remote controlled switch, connecting and completely disconnecting the battery is included in the drone setup. Figure 1.4 below shows the installed switch along its remote-controller. When first receiving power, the switch is by default in the "off" position, so no power is provided to the drone. Before any operations the user therefore has to press the "A" button on the remote-switch.
In the event of an emergency, this provides the user with a kill-switch: in any given moment, the user can instantly cut power from the drone and therefore stop the propellers by pressing the "B" button.
At start-up (After pressing the "A" button), when the ESCs first receive power, the state of the drone can be determined by the sounds emitted from the ESCs as listed in Table 1.2 below.

[3]The motor1 (f1) is the top left one to the phone (+Y/-X quadrant) and the rest are numbered in clockwise manner

Figure 1.4: The remote-controlled power switch

Table 1.2: ESC Sound Description [5]

Sound	**Description**
♪1234	Ready
BBBBBB...	No signal input to the ESC or throttle stick is not in the bottom position

1.3.2 Software Framework

As mentioned above, the quadro-copter used for the work on this thesis features an on-board Android smartphone as its flight controller. The only inertial measurement unit (IMU) in the on-board system is the one in the Android phone. For the PCESS research group, the standard operating mode of the drone is the use of a second hand-held "Master" Android device. Its orientation is mimicked by the on-board "Slave" device, allowing the user to control the drone. The signal flow of this operating mode is depicted in Figure 1.5 below.

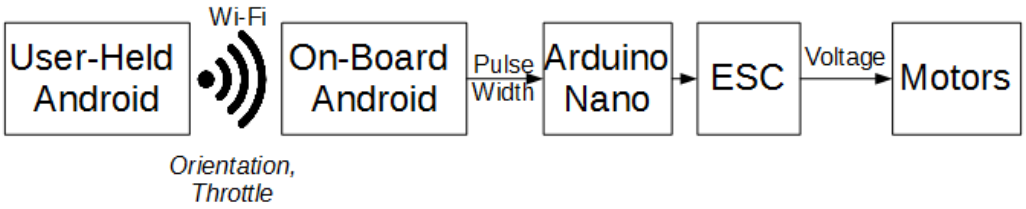


Figure 1.5: Block diagram of the signal flow [2]

In order for the "Slave" device to follow the orientation of the "Master" device, a controller is running on the "Slave" which is minimizing the difference between the master and slave roll, pitch, and yaw angles, that is the vector of errors. [7]

$$[e_\phi(t), e_\theta(t), e_\psi(t)] = [\phi_M(t), \theta_M(t), \psi_M(t)] - [\phi_{Sl}(t), \theta_{Sl}(t), \psi_{Sl}(t)] \tag{1-1}$$

Since only Android measurements are used for the determination of the angles in (1-1), they are expressed in the frame defined by the Android API. Figure 1.6 below shows the quadro-copter's body fixed coordinate system, with the Android device attached.

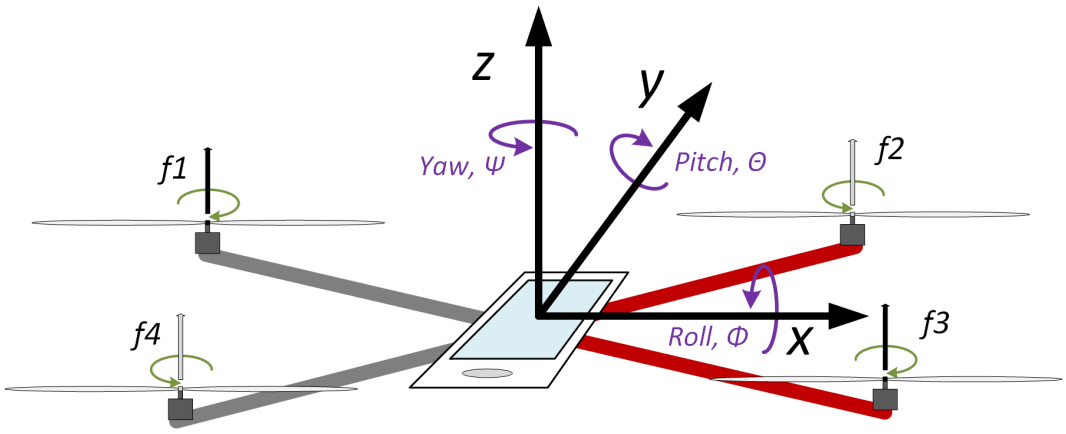


Figure 1.6: Schematics of copter body fixed coordinate system with Android device

For the interfaces depicted in Figure 1.5, the PCESS research group developed a Software Development Kit (SDK) which is heavily utilized in all Android implementations throughout this thesis.

The PcessSDK

For running the on-board control algorithms, the "Slave" device needs its own orientation measurement along with the orientation of the "Master" device. In Androids standard sensor API, the Android SensorManager is included, which produces SensorEvents. However, this is not usable for the setup shown in Figure 1.5, because Google protects the SensorEvent constructor so that one cannot re-instantiate SensorEvents that have been sent over a network, and one cannot create custom SensorEvents. Also SensorEvents are not serializable. [8]

SimpleEvents

To avoid these issues, the PCESS research group has developed the PcessSDK which, as a core functionality, contains the *SimpleEventManager*. Its task is to deal with *SimpleEvents* which omit the previously mentioned shortcomings of SensorEvents and are particularly useful for networked applications because they: [8]

1. Hold a reference to the ID of the device which produced the event.
2. Have a public constructor.

To use the SimpleEventManager, it must be initialized with a valid context by calling `SimpleEventManager.initialize`. Typically this method is called towards the top of the `onCreate` method in the first Activity that is started, although it also doesn't hurt to call this function more than once.
By calling the `SimpleEventManager.registerListener` method, it is then possible to register for SimpleEvents. The type of the event has to be specified along with the desired sampling period between events as an integer number of microseconds. [8]
At this point it is very important to understand that Android is not a real-time system, and hence, user-created threads have lower priority than several Android system threads. Figure 1.7 illustrates this problem, that the specified sampling rate cannot be guaranteed. Although the orientation measurements are technically available at approximately 100 Hz on the Xperia Z1 Compact[4], the figure shows that the time between consecutive events can be significantly longer than 10 ms. [9]

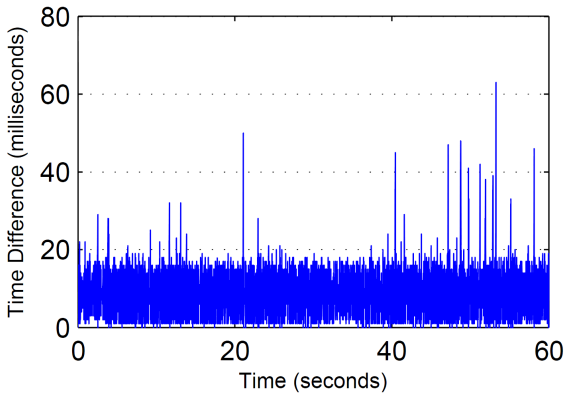


Figure 1.7: Time between consecutive quaternion measurements on the the Xperia Z1 Compact [9]

[4]This is the device used in the frame of this thesis, but any other Android smart-phone with a gyro can be used

To access the data of any SimpleEvent, e.g. a sensor value, a *SimpleEventListener* needs to be implemented that actually receives the SimpleEvents of the respective type. This is achieved by including the `OnEvent` method in the activity.
A Minimal Working Example (MWE), called *SimpleEventMWE*, for handling SimpleEvents is provided as part of the PcessSDK in the "examples" folder.

The WiFi Connection

The backbone for sending SimpleEvents between devices is the *WifiService* class, that is also part of the PcessSDK. This class establishes a Wifi Direct connection between two devices. The connection runs as a service, which means that it runs in the background even when the app is closed. Any serializable object can be transmitted via the Wifi Direct, but the synergy with the SimpleEvents is even greater.
A backdoor has been included in the WifiService class, so that when two devices connect, the set of available SimpleEvent-Types on each device (including the onboard sensors as well as the custom events) are shared with each other. That means that after a "Master" and "Slave" are connected, the "Slave" device can register for SimpleEvents on the "Master" in almost exactly the same way that it registers for SimpleEvents on its own device. For example, if the "Slave" wants to register for accelerometer measurements on itself, then it could register for those measurements by calling: [8]

```
1 SimpleEventManager.registerListener(this, Sensor.TYPE_ROTATION_VECTOR,
    SensorManager.SENSOR_DELAY_UI);
```

However, with the Wifi Direct link established between the two devices via the WifiService class, registering for the same type of event on the peer device can be implemented like this, without having to deal with any *send* or *receive* method: [8]

```
1 String peer = getPeerDeviceID(); //Get ID of peer Device
2 SimpleEvent event = new SimpleEvent(Sensor.TYPE_ROTATION_VECTOR, peer);
3 SimpleEventManager.registerListener(this, event,
    SensorManager.SENSOR_DELAY_UI);
```

Flight Controller on the Slave

For controlling the motors, the flight controller running on the "Slave" device must extend the *FlightController* class included in the PcessSDK. It therefore has to define the following three methods:

```
1 public Set<SimpleEvent> getRequiredEvents();
2 public void update(List<Double> measurements);
3 public Map<SimpleEvent, int[]> getMeasurementMap();
```

Each of these methods serves a specific purpose: [8]

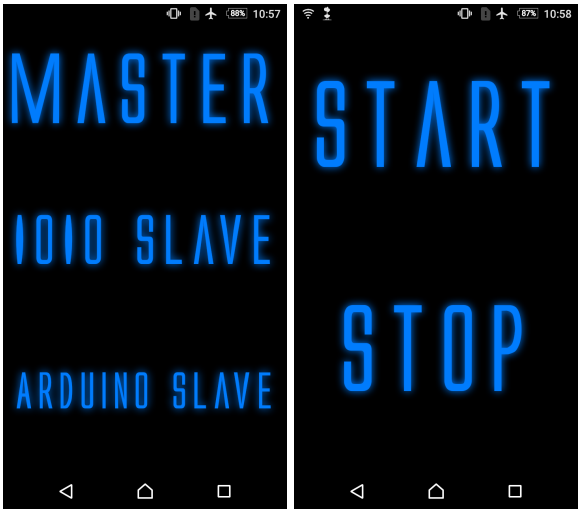
1. The *getRequiredEvents* method defines the Set of SimpleEvents that must be received before the controller on the "Slave" device will start.

2. The *update* method is where the controller will calculate the pulse width commands for each motor. It will be called at periodic intervals with the list of measurements that the controller needs.

3. The *getMeasurementMap* method defines how the measurements will be put into the list that the update method gets.

Using the Preinstalled Flight Software

For the previously described operating mode with the "Slave" device mimicking the movements of a hand-held "Master" device, there are currently two working controllers provided in the App: An Linear-Quadratic-Integral (LQI)-based controller which was designed around a linearised model of the system and a PID controller which allows the user to change the proportional, integral, and derivative gains of the controller during the flight.
Hereafter the basic steps required to use these controllers in their intended ways are briefly described. This should provide a basic understanding on how the parts of the PcessSDK are applied.
First step is making sure that the WiFi is enabled on the phones. The "Slave" device is then connected to the ESC control board via a Micro-USB cable. This control board can either be an Arduino Nano (like on the copter used for the work on this thesis) or an IOIO board. When opening the "Flight" App on the "Slave" device, the user then has to select the respective board from the two bottom options as shown in the screenshot in Figure 1.8a below.

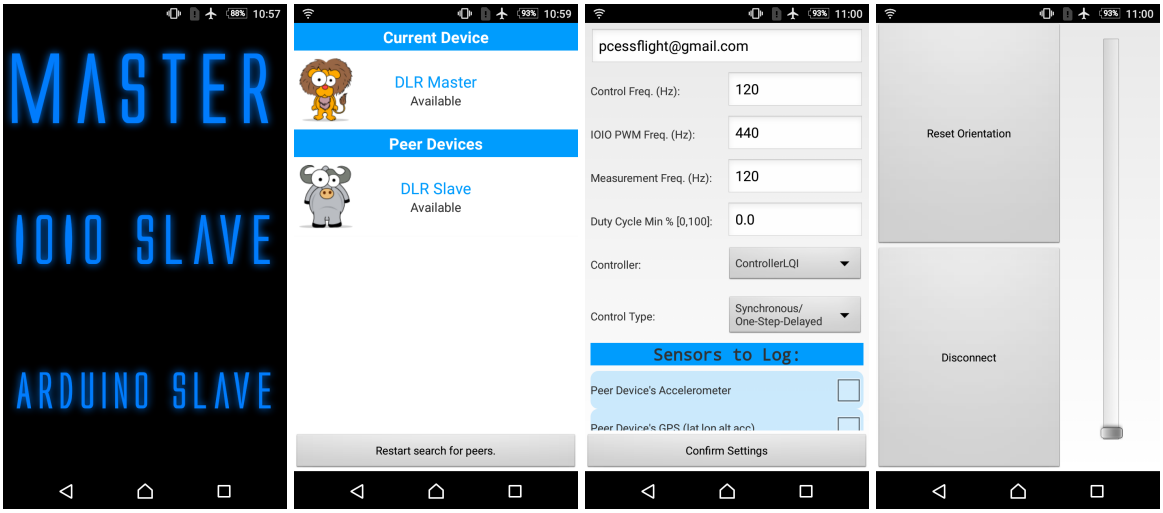


(a) Flight-App landing screen (b) Slave device GUI

Figure 1.8: Flight App GUI sequence on "Slave" device

On the next screen, as shown in Figure 1.8b, the user then starts the flight controller by pressing "START". The stylized bird in the very left top of the phones' statusbar indicates the running controller. No action is then required on the "Slave" device until the end of the flight, when the controller is halted by pressing "STOP".
On the hand-held "Master" device, after selecting "MASTER" on the landing screen as shown in Figure 1.9a, the user can connect to a peer device. The "Master" device automatically starts searching for peers to connect to. When it finds available peers, they will be displayed, as shown in Figure 1.9b.
Connection is established by long-pressing the name of the respective "Slave" device. The status note should now change from *Available* to *Invited* to *Connected*.
Shortly after the status changes to *Connected*, the settings screen is displayed, as shown in Figure 1.9c. This should take no more than 30 seconds. [7]
In the flight settings, the parameters that will hold for the duration of the flight are configured. The most important setting is selecting the controller used to control the attitude of the "Slave" device from the drop-down list. As already mentioned before, a LQI and a PID controller are implemented for the mimicking operating mode. Further below, the necessary steps to take for developing a controller and having it appear in this list are described.

Note: The PID controller also seems to allow changing the inertia matrix of the drone, but actually those parameters are just there for demonstration purposes. They are not actually used to configure the PID controller in its current iteration.



(a) Flight-App landing screen

(b) Wi-Fi connection to peer device

(c) Flight settings menu on "Master"

(d) GUI on "Master" device (LQI Controller)

Figure 1.9: Flight App GUI sequence on "Master" device

At the bottom of the screen, the sensors to be logged during the flight are selected. One should be careful not to select too many, since this will add overhead that might interfere with functionality of the controller. At the top of the screen, an email address to receive all of the log files from the flight can be entered. The log files will also be saved in the "Movies" folder of the "Slave" device, so they can be downloaded after the flight. Specifically, each flight should generate a subfolder in the "Movies" folder with the time that the flight started. If the flight was successful, then one should see "EventLogs" in that folder for each of the sensors that were selected for logging. Each of these logs is identified with a *Device.ID()* to differentiate between measurements on the "Master" and "Slave". Typically, you can tell which ID corresponds to the "Master" by examining the ID of the *EventLog_Gain*, since the gain events only occur on the hand-held device. Changing any of the other settings is strongly depreciated.

After confirming the settings, the "Master" device's graphical user interface (GUI) for the flight controller is displayed, as shown in Figure 1.9d and Figure 1.10. There are buttons for disconnecting the WiFi and for resetting the reference orientation. Both controllers have a SeekBar which essentially controls the average thrust of the four motors, from 0 to 100% power. To prevent unintentional disconnects during flight, the disconnect button is disabled unless the throttle is below 5%.
Before the controller on the "Slave" device initialises, the *required Events*, as previ-

ously introduced, have to be received by the "Slave". For the LQI controller this is an orientation-reset event and a change in value of the throttle seek bar. As the PID controller allows for controller gain changes during flight, as shown in the user interface screenshot in Figure 1.10, each of the gain-dials also has to be moved[5].

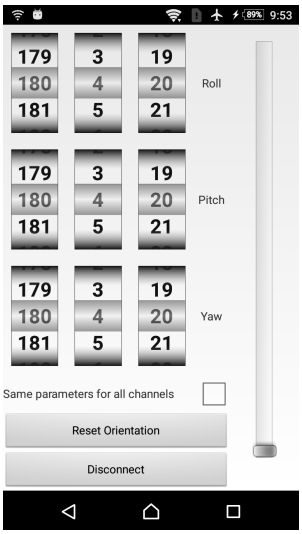


Figure 1.10: GUI on "Master" device for the PID controller

When the orientation is reset, the "Slave" device will emit a beep sound. This is to give some reassurance that the "Slave" actually received the command. After all requred events are received by the "Slave", the controller is running and the ESCs receive the PW signals. At this point the drone can be powered with button "A" on the remote controlled switch introduced in Chapter 1.3.1. Confirmation that indeed all required events have been received by the "Slave" at this point is given by the sound emitted by the ESCs. Like presented in Table 1.2, the ESCs emit a four tone jingle if ready to go, or a continuous beep otherwise.

Development of a new flight controller

The PCESS research group provided complete instructions for the development of a new flight controller (in [8]), hereafter only the basic steps are mentioned.
A flight controller consists of two parts:

1. The GUI on the "Master" device and the logic providing the reference orientation for the "Slave"

2. The feedback controller that runs on the "Slave" device

[5]Gains that are known to work well are: Roll: (180,2,20); Pitch: (160,2,20); Yaw: (150,2,10) [7]

The current naming convention used in the PcessSDK is, that components of type 1 are called *XControllerFragment*, while components of type 2 are called *XController*. For example, there is a *PIDController* and a *PIDControllerFragment*.

The interface shown to the user during the flight is not liable to any specific requirements and is heavily dependent on the type of implemented controller. However, a good minimal working example is the *LQIControllerFragment* since it contains the basic components that almost every *XControllerFragment* will need for hand-held applications:

1. A button to trigger a disconnect from the slave. This button simply runs the command:

```
((MasterActivity) getActivity()).onDisconnectRequested();
```

2. A *SeekBar* to control the average motor thrust.

3. For applications that track the orientation of the "Master": A button to reset the reference orientation, that is, the orientation of the "Master" device at which the drone should "do nothing".

As previously mentioned, the *XController* running on the "Slave" must extend the *FlightController* class and therefore must implement the methods *getRequiredEvents*, *update* and *getMeasurementMap*. Rather than setting up the controller from scratch, it is strongly recommended to just copy the existing code from one of the existing controllers, as they provide a perfect starting point and only customize them to the intended application.

The last step in the development of a new controller is to make it appear in the settings screen shown in Figure 1.9c. To achieve this, the classes have to be recognized by the *Flight* framework. This is done by modifying the *FlightControllerFactory* in 3 places[6]:

1. Adding a simple class name for the developed controller class to the *controllerClassStrings* in the static initialization block. For instance, if the controller class is *XController*, then one would add the line:

```
controllerClassStrings.add(XController.class.getSimpleName());
```

[6]If the developed controller contains custom constructor settings, like the PID controller does, additional steps have to be taken. Refer to Chapter 3 of [8] for the necessary steps.

2. In the *newInstance* method, the new controller object is created if it is requested. So add an "else if" statement of the form:

```
} else if (
controllerClassString.equals(XController.class.getSimpleName())) {
    return new XController();
```

3. The "Master" device's interface fragment (*XControllerFragment*) has to be returned from the *newControlFragment* method by inserting an "else if" statement of the form:

```
} else if (
controllerClassString.equals(XController.class.getSimpleName())) {
    return XControllerFragment.newInstance(settings);
```

1.4 The TEAMS Facility

The Test Environment for Applications of Multiple Spacecraft (TEAMS) facility is a laboratory of the GNC department located at DLR Bremen. The lab consists of a highly smooth granite platform measuring $4m \, x \, 5m$ and a DTrack infra-red tracking system. With the air cushion vehicles, shown in Figure 1.11, the TEAMS laboratory emulates the force- and torque-free dynamics of satellites.

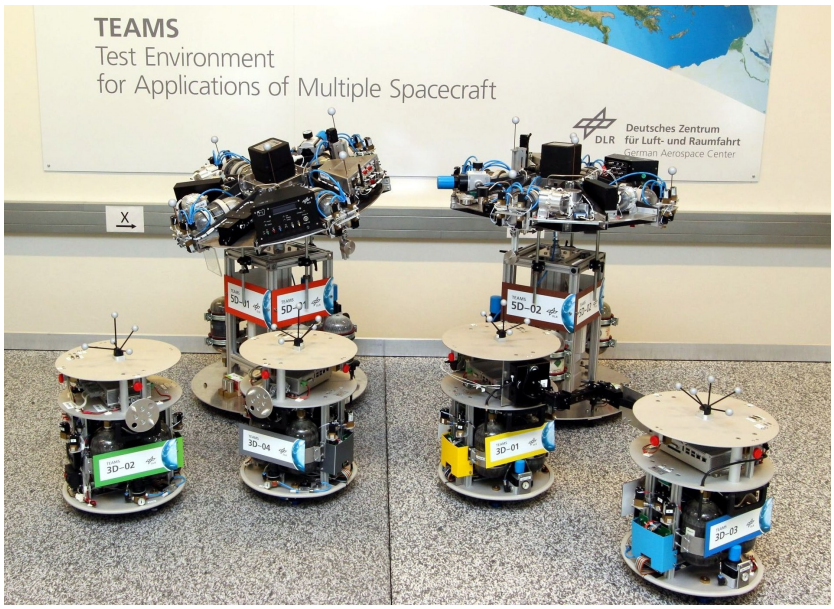


Figure 1.11: The TEAMS vehicles on the granite table [10]

Five passive tracking targets, highly reflective balls as shown in Figure 1.12 are mounted on each vehicle in a different configuration, so that the tracking system is able to distinguish between them.

Figure 1.12: Passive spherical tracking markers [11]

The attitude and position of the detected vehicles is broadcast via a local wireless network. Figure 1.13 below, illustrates the functional principle of the DTrack system. Information on the position and attitude of all vehicles is therefore available for all participants. Additionally the vehicles are each equipped with fibre-optic gyroscopes and accelerometers.

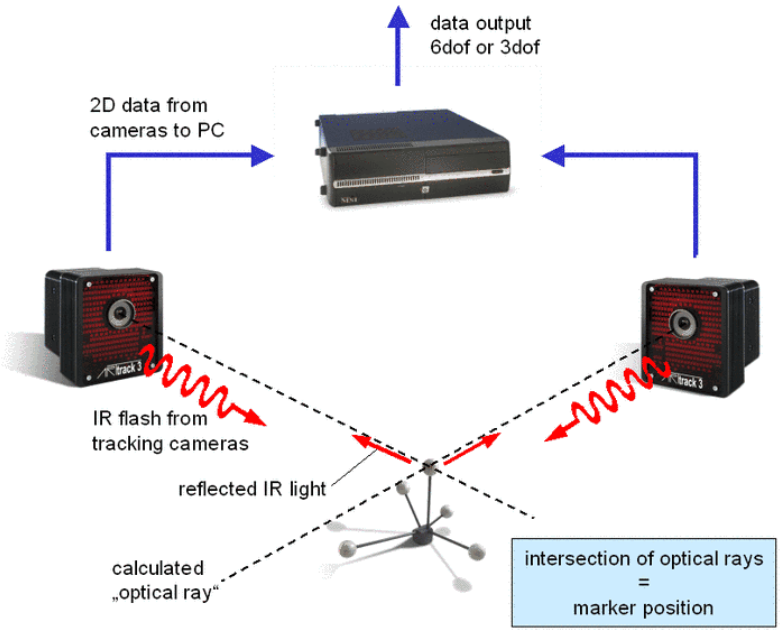


Figure 1.13: Functional principle of DTrack system [12]

The vehicles' QNX real-time operating system, running on the x86 Atom Z530 on a PC/104 stack on-board computer, allows automatic generation of C code with Simulink Coder. This ensures easy development and testing of algorithms for guidance, navigation and control. [10, 12].

The origin of the room fixed coordinate system is, as shown in Figure 1.14 below, in the center of the granite platform with the Z-axis pointing upwards and the X-axis along the longer side of the table.

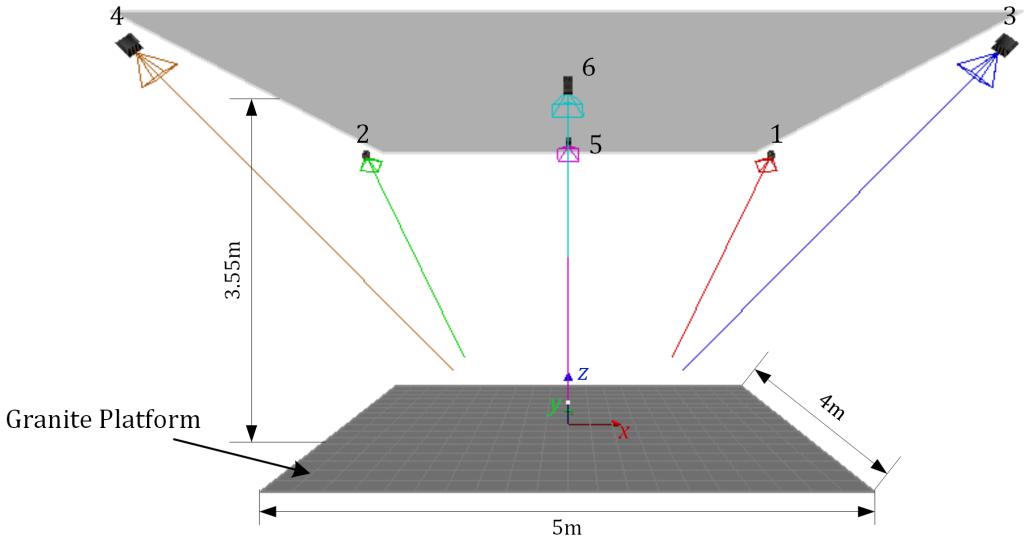


Figure 1.14: The TEAMS facility tracking area dimensions [13]

The ceiling height above the platform is 3.55m, however due to the cameras pointing downwards, the height, at which targets are in the field of view of enough cameras to ensure reliable tracking, ends at approximately 2m above the platform, as suggested by the manufacturer in his proposal for the system ([13]). The available volume for tracked drone flight in the TEAMS facility is therefore limited to this height.

2 Solution Approach

Hereafter the overall approaches taken to achieve the objectives and to fulfil the work packages unrolled in Chapter 1.2 are described.
In Chapter 2.1, the system architecture is presented from a high-level point of view and therefore introduces the main subsystems and their interfaces. In Chapter 2.2 and Chapter 2.3 the subsystems and their interfaces are described in further detail, setting the stage for understanding the implementations documented in Chapter 3. Chapter 2.4 describes the modifications to the copter's hardware.

2.1 System Architecture

The system architecture with all relevant elements is schematically depicted in Figure 2.1 below.

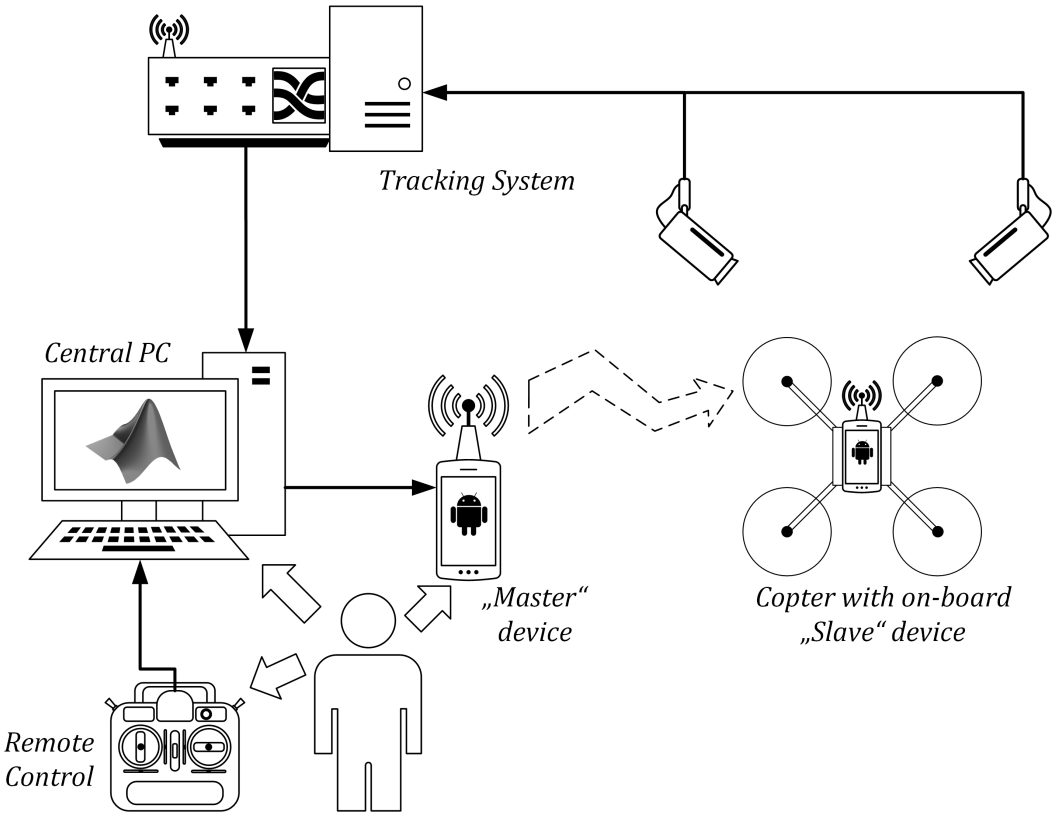


Figure 2.1: High-level system architecture schematics

The system consists of the quadro-copter with the on-board "Slave" Android device, the "Master" device, the DTrack Tracking system, a USB remote controller and a central PC with MATLAB and Simulink.

2.2 Subsystem Descriptions

Hereafter a brief introduction to the subsystems is given along with their intended tasks within the overall solution approach.

The "Slave" Android device is a Sony Xperia Z1 Compact, as shown in Figure 2.2. It is used just as in the PCESS research groups use-case, where it is fixed to the quadro-copter. An on-board attitude controller implemented in Java is running on it. This controller calculates the PW commands for the four motors, so that the drone follows reference attitude angles and actions the input average motor thrust.

Figure 2.2: The Sony Xperia Z1 Compact

The "Master" Android device is also a Sony Xperia Z1 Compact. It is used to forward the reference attitude angles to the "Slave" device, fully utilizing the software framework of the PcessSDK. The reference angles for the developments in the frame of this thesis are not generated by the movement and orientation of the "Master" device, but rather sent to it from the central PC.

The USB Remote Controller used is a *Modelcraft* 6-channel joypad. As shown in Figure 2.3 it is shaped like a common remote controller for RC flight models. It connects to a Windows PC via USB and is recognised as a standard joystick by Windows. Additional drivers are not needed[7]. The USB remote controller is used

[7]Tested under Windows 7 and Windows 10

as the hand-held controller. It can provide the reference attitude angles for the drone along with the throttle from the position of its flight-sticks. There is a two-position switch and a rotary knob as additional input sources to e.g. trigger events or behaviour.

Figure 2.3: The used USB Remote Controller

The Tracking System is the DTrack system installed in the TEAMS facility as introduced in Chapter 1.4. It is used to provide the current 3D position and orientation of the drone to the central PC.

The central PC is running a Simulink simulation, where the data from the tracking system and the input from the remote controller are acquired. In this Simulink simulation guidance calculations and trajectory controllers can be easily implemented utilizing the tracking data and user input to calculate reference angles for the quadrocopter, so that it executes a desired mission.

2.3 Interface Descriptions

This section focuses on the interfaces between the subsystems introduced previously. Hereafter the interfaces and the transmitted data are described, whereas the necessary implementations to make them work are described in detail in Chapter 3. Figure 2.4 below shows the general signal flow block diagram of the whole system.

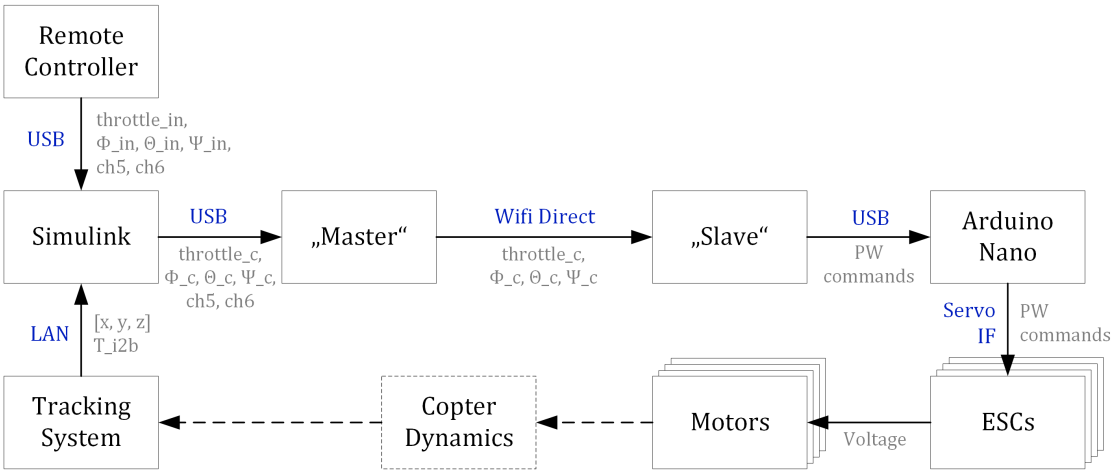


Figure 2.4: Signal Flow Block Diagram as implemented

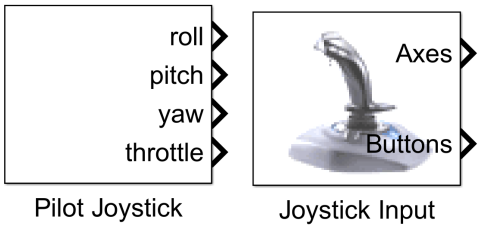
The interfaces beginning from the "Master" device to the motors are handled by the PcessSDK, as already described in the introduction to the drone in Chapter 1.3. For the work in the frame of this thesis, these interfaces are applied unmodified so no further explanations are given here.
The interfaces developed within the scope of this thesis are the in- and outputs to Simulink. On that account, the general description and the requirements on these interfaces are described hereafter.

Remote Controller → Simulink

The current position of the flight-sticks, the rotary knob and the two-position switch on the remote control shall be available to a running Simulink simulation on the connected PC as floating point numbers that allow clear association with the current state. This required behaviour is in fact already available ready to use in two commercial Simulink toolboxes:

1. Option 1 would be the *Pilot Joystick* block included in the Simulink Aerospace Blockset as shown in Figure 2.5a.

2. Option 2 would be the *Joystick Input* block included in Simulink 3D Animation™as shown in Figure 2.5b.



(a) *Pilot Joystick* Block from Simulink Aerospace Blockset [14]

(b) *Joystick Input* Block from Simulink 3D Animation [15]

Figure 2.5: Available Simulink Blocks for joystick input

However, as both toolboxes are not available on the current MATLAB installation on the PC in the TEAMS facility, a custom solution that is fulfilling the requirements without utilizing additional commercial software pieces has had to be found. See Chapter 3.1 for the description.

Tracking System → Simulink

The software of the tracking system by Advanced Realtime Tracking GmbH (ART) consists of two parts. The front-end software is installed on a remote PC which is connected to the controller via Ethernet. A GUI enables the user to control the tracking system completely from here, in the particular case of this thesis, the PC in the TEAMS facility.
The Linux-based back-end software runs on the controller, which does all necessary calculations for position and attitude determination. The data is broadcast by the Controller via a TCP/IP connection. The front-end software acquires the information on this channel. [11]
In addition to the front-end software, a SDK is provided by ART[8] for custom developments. The C++ libraries provided in the SDK are used to develop a S-function block for Simulink to receive the navigation messages in a running simulation. See Chapter 3.2 for the description.

[8]Alternatively available from: `https://www.schneider-digital.com/support/download/Tools-Ressourcen/ART_Tracking/DTrack2-Controller/SDK/` (as of March 2018)

Simulink → Android

The requirement for this interface is to send the controller calculations to an Android application running on the "Master" phone connected to the PC. The data source of the interface on the PC is a running Simulink simulation.
In the specific use-case of this thesis, the PC in the TEAMS facility generates the data: floating point numbers representing the calculated reference angles for the copter and other data to be visualized on the "Master" device for identification by the operator.
This functionality is achieved by utilizing the Android Debugging Bridge (ADB), which is a command line tool by Google for Android developers. The ADB is included in the Android SDK Platform-Tools package in the SDK manager as part of the Android Studio installation which is anyhow needed for Android application developments[9].
To use ADB with a device connected to a PC via USB, *USB debugging* has to be enabled in the device system settings under Developer options. [16]
To exploit ADB functionality for communication with a running application on the "Master", the *port forwarding* functionality is utilized. *Port forwarding* forwards requests on a specific TCP port on the PC to a TCP port on the connected device. Therewith the Simulink simulation communicates to a specific TCP port on the host PC, ADB then forwards the communication to the Android device, where the application can listen to the data received on the port.
Simulink does not support a TCP connection in its standard installation, however the functionality is included in the Instrument Control Toolbox.

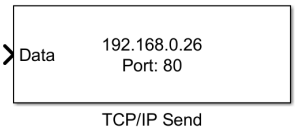


Figure 2.6: *TCP/IP Send* from Simulink Instrument Control Toolbox [17]

The *TCP/IP Send* Block from the Simulink Instrument Control Toolbox, as shown in Figure 2.6, sends data to the specified remote port at the end of the simulation or at fixed intervals during a simulation. However this commercial toolbox is not available on the current installation on the PC in TEAMS, so again a custom solution was implemented. See Chapter 3.3 for the description.

[9] Alternatively, the Platform-Tools can be downloaded as a stand-alone package from: `https://developer.android.com/studio/releases/platform-tools.html` (as of March 2018)

2.4 Hardware Changes

Hereafter the changes to the quadro-copters hardware are described. In Chapter 2.4.1 the accessories added to the copter are shown and in Chapter 2.4.2 the mounting of the tracking markers is illustrated. As all of the hardware additions add mass to the copter, a detailed mass breakdown of the drone is conducted in Chapter 2.4.3.

2.4.1 Quadro-Copter Accessories

In order to increase the crash resistance of the drone, propeller guards and landing gear have been added. Figure 2.7 below shows the copter with the additions.

Figure 2.7: The copter with landing gear and propeller guard accessories

As they are commercial readily available accessories no further explanation is given here. The added mass is accounted for in Chapter 2.4.3.

2.4.2 Tracking Markers

To enable unmistakeable tracking of the copter's position and attitude, five markers have been fixed to the drone in a unique configuration. Premise for the undertaking was, that the fixations shall be fully reversible, lightweight and cheap to accomplish. Hence the existing mounting screws for the copter assembly are used. As the threads in the markers are sized at M3 and the drone uses M2.5 threads for assembly, thread adapters are utilized, as shown in Figure 2.8.

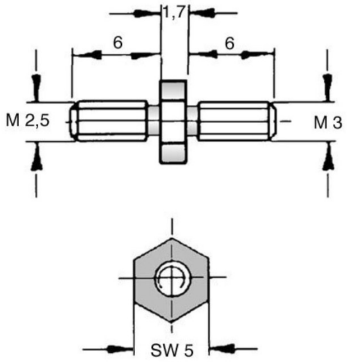


Figure 2.8: Thread Adapter M2.5 to M3 (male/male) [18]

To achieve a marker pattern that allows for unique identification of the orientation, each of the five markers is mounted in a different height. This is done by stacking standard M3 standoffs, each with a length of 30mm. To preserve the markers threading, a metallic standoff is used at the top of the stack.
Figure 2.9 shows a sketch of the F450 top plate. In the figure, the positions of the screws used to fix tracking markers are marked and numbered. At positions 1-4, the standard M2.5x6 bolts [5] are replaced with the thread adapters shown in Figure 2.8. As the "Slave" device, which is centrally fixed to the copters top plate, covers all but the four screws, the fifth marker is fixed to the arm of motor 1 using a M3 bolt at position 5.

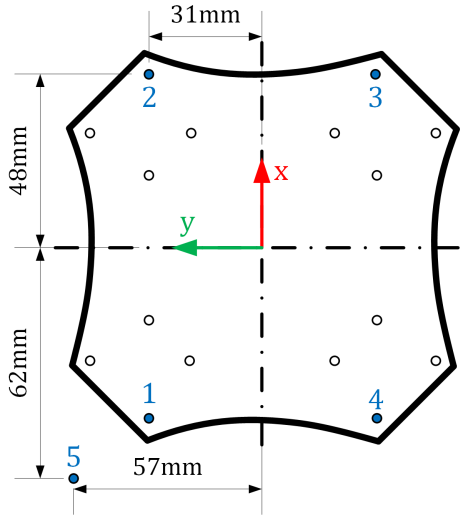


Figure 2.9: X/Y-Positions of the markers relative to the center of the top plate

Table 2.1 below lists the 3D coordinates of the marker balls center relative to the center of the top plate in their current configuration. This information can be very

handy in the calibration process. Note that these are calculated distances which is why they may differ in reality. When in doubt rely on the automated calibration process of the DTrack2 system[10].

Table 2.1: Marker Positions in X, Y and Z coordinates, relative to top plate center

Marker #	**X**	**Y**	**Z**
1	-48mm	31mm	116.7mm
2	48mm	31mm	86.7mm
3	48mm	-31mm	206.7mm
4	-48mm	-31mm	146.7mm
5	-62mm	57mm	59.3mm

2.4.3 Copter Mass

Table 2.2 gives a detailed breakdown of the copters weighted mass.

Table 2.2: Copter mass breakdown

Item	**Description**	**Mass**
Assembled Airframe	including wiring, Motors, ESCs, Arduino and Power Switch	804.4g
Smartphone	Sony Xperia Z1 compact including protective case	177.8g
Propellers	4x DJI 9450 @ 12.3g each	49.2g
Battery	Type: 3S LiPo	175.1g
Propeller Guards	4x 30.4g	121.6g
Landing Gear	4x 16.7g	66.8g
Tracking Markers	16x Polymer 30mm Spacer @ 0.8g each 5x Brass 15mm Spacer @ 2g each 5x Marker Balls @ 4g each	42.8g
	Total mass:	**1437.7g**

[10]Instructions on the calibration process are given in Appendix C

3 Interface Implementation

In this chapter, the implementations of the interfaces introduced in Chapter 2.3 are explained.

3.1 USB Remote Controller Interface

To read the input from the previously introduced USB remote controller in a MATLAB Simulink environment without the use of any commercial toolbox, the *HebiRobotics/MatlabInput* library is utilized. This library, created by Florian Enner and published to the MathWorks File Exchange, enables keyboard and joystick input into MATLAB. It makes use of the Java library *JInput* and does not support code generation. [19]
Two classes are provided with the library:

1. `HebiJoystick.m` basically is a drop-in replacement for *vrjoystick* for users who do not have access to the Simulink 3D Animation toolbox.

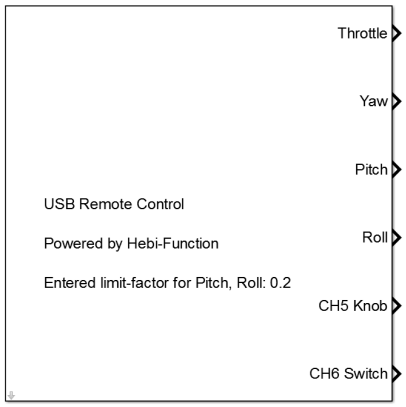
2. `HebiKeyboard.m` provides similar functionality as `HebiJoystick.m`, but for keyboard inputs.

For the USB remote controller the *HebiJoystick* class is used. It creates a joystick object for the ID that is passed to it as a positive integer number. The operating systems driver for the connected joystick assigns this number. When only one joystick is connected, its ID is usually '1'.
The *HebiJoystick* class defines several methods for interacting with the connected joystick. However, only the very basic functionality of connecting to and reading from the connected remote controller is needed. The minimum working example below shows the procedure for reading the values from all axis of joystick 1 once and then closing the connection.

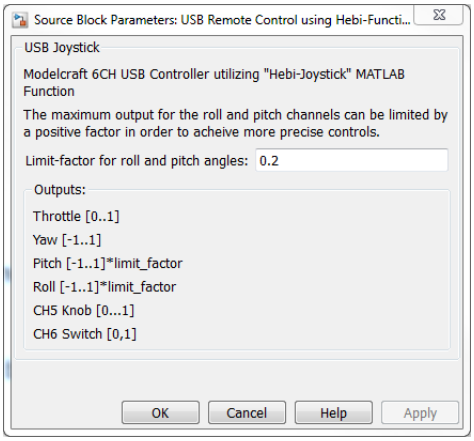
```
joy = HebiJoystick(1); %Create joystick object for joystick 1
axis_read = axis(joy); %Get the State-Vector of the axes
close(joy);            %Close the joystick object
```

The example above creates the state-vector in the MATLAB workspace, however the information is needed in a running Simulink simulation as described in Chapter 2.3. As the *HebiJoystick* class does not support code generation, this is achieved by using the *Interpreted MATLAB Function* block inside a masked Simulink subsystem. The

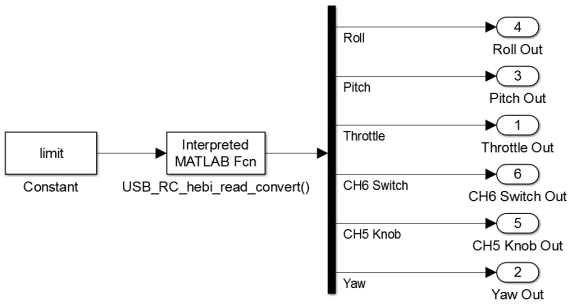
joystick object is created as a global variable at initialisation of the simulation, utilizing the *Callback* functionality of Simulink. In the same manner, the connection is closed at simulation end-time. Figure 3.1a below shows the developed block for continuously reading the states from the USB remote controller.



(a) Simulink Block for USB RC readings



(b) Block Mask



(c) Inside the Subsystem

Figure 3.1: The developed Block for USB Remote Controller readings in Simulink

Figure 3.1b shows the mask of the subsystem block where further explanations on the function are given. Here the user also has the possibility to enter the limits for the reference roll and pitch angles to enable significantly finer control in manual flight. The underlying functionality is shown in Figure 3.1c, basically the *Interpreted MATLAB Fcn* block and the assignment of the channels to the output ports.
In Table 3.1 the mapping of the channels for the "Modelcraft 6CH USB Controller" to their referenced functionality in the subsystem block is given.
The sign convention for the flight angles follows the common flight mechanics convention for aircraft body fixed system, where the Z-axis points down and the X-axis points forwards [20, p.9]. In this thesis however the copters body fixed Z-axis is

defined pointing up. Therefore the pitch and roll angles are multiplied with -1 after the block output.

Table 3.1: Mapping of USB RC channels to reference values

RC Function	Ref. Function	"Hebi"Axis Ch. #	Position	Output
Left Stick	Throttle	3	Down	0
			Up	+1
	Yaw	6	Left	-1
			Right	+1
Right Stick	Pitch	2	Down	-limit
			Up	+limit
	Roll	1	Left	-limit
			Right	+limit
Rotary Knob	ch5	5	Left	0
			Right	+1
Switch	ch6	4	Down	0
			Up	+1

3.2 Tracking Interface to Simulink

3.2.1 Receiving Navigation Data

The manufacturer of the tracking system ART provides the previously mentioned SDK that is utilized to implement a Simulink S-function block. The code for the S-function is written in C++, where the DTrack libraries are included. The basic template for any S-function can be opened by typing the following into the MATLAB command line:

```
edit([matlabroot,'\simulink\src\sfuntmpl_basic.c']);
```

Following this template the following methods have been implemented:

- `mdlInitializeSizes`: This function is used by Simulink to determine the S-function block's interface characteristics. Hence, here the dimensions of the

output ports are defined.

- `mdlInitializeSampleTimes`: This function specifies the sample time of the S-function block. In the present implementation the sample time is set via parameter:

```
ssSetSampleTime(S, 0, mxGetScalar(ssGetSFcnParam(S, 0)));
```

- `MDL_START`: This function is called once at start of model execution. Therefore the object for receiving the tracking information is created here:

```
dt = new DTrack2("", 0, port);
```

- `mdlOutputs`: This function is called by the Simulink engine at each time step to create the new information for the output ports. The navigation information is received and allocated to the respective output ports here. The ID of the body to be tracked is specified as a parameter of the S-function. This ID has to be consistent with the calibration data in the DTrack2 front-end software:

```
//Get Body ID to be tracked
int body2beTracked = (int)mxGetScalar(ssGetSFcnParam(S,3));
//Read the tracking inftormation for this body
body = dt->get_body(body2beTracked-1);
```

- `mdlTerminate`: This function is called at the termination of the simulation run. Therefore this is the place to free the allocated memory by deleting the created object instances:

```
delete dt;
```

The *.cpp* file created after this manner then has to be complied for the use in Simulink. This is performed with the *mex*-command in MATLAB. The used libraries have to be specified in the command as well.

```
mex Sfun_DTrack_singleBody_on_PC_sf.cpp ...
    DTrack2.cpp DTrackSDK.cpp DTrackNet.cpp DTrackParse.cpp ...
    -l"ws2_32"
```

It has been found, that in order to successfully compile the DTrack libraries on a 64bit system, the library *ws2_32.lib* has to be used.

When using the S-function block in a simulation that is running in Accelerator mode, it is crucial to prevent the Simulink engine from trying to compile the *.cpp* source of the S-function again. This is because the used libraries can not successfully be applied in this build. However, this second build attempt can be suppressed by having a MATLAB function called *rtwmakecfg.m* in the same directory as the S-function sources. The contents of this function have to be:

```
function makeInfo = rtwmakecfg
   % Don't build in accelerator mode
   if(strcmp(get_param(bdroot, 'SimulationMode'), 'accelerator'))
       makeInfo = {};
       return;
   end
end
```

With these considerations, the DTrack tracking information can successfully be received in Simulink at runtime. As already introduced, the tracking information includes the bodys position in mm in the room coordinate system as well as the 3x3 transformation matrix from body coordinates to room coordinates.

3.2.2 Determination of Euler Angles

Euler angles are a convenient way to express a vehicles' attitude, as their values are directly visually identifiable:

- ϕ - Phi, Roll-angle, rotation around x-axis
- θ - Theta, Pitch-angle, rotation around y-axis
- ψ - Psi, Yaw-angle, rotation around z-axis

Transformation matrices are the result of combining subsequent rotations around the coordinate system axes in predefined orders. According to [20, p.57], the transformation matrix from inertial (i) to body-fixed (b) coordinates can be expressed with the rotation sequence ψ, θ, ϕ.

$$T_{i2b} = T_{b2i}^{T} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & cos(\phi) & sin(\phi) \\ 0 & -sin(\phi) & cos(\phi) \end{bmatrix} \begin{bmatrix} cos(\theta) & 0 & -sin(\theta) \\ 0 & 1 & 0 \\ sin(\theta) & 0 & cos(\theta) \end{bmatrix} \begin{bmatrix} cos(\psi) & sin(\psi) & 0 \\ -sin(\psi) & cos(\psi) & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad (3\text{-}1)$$

This sequence is also called the *321-Rotation*, as the rotation around the Z-axis is applied first following the rotation around the Y-axis and lastly the rotation around the X-axis.

Note: The user has to be aware, that the angles displayed in the DTrack2 front-end software are determined by assuming a 123-rotation sequence. These displayed angles therefore are not suitable for identifying the actual copter orientation with respect to the room.

Expanding (3-1) and shortening $sin(\alpha), cos(\alpha)$ to $s(\alpha), c(\alpha)$ yields:

$$T_{i2b} = \begin{bmatrix} c(\psi)c(\theta) & s(\psi)c(\theta) & -s(\theta) \\ c(\psi)s(\theta)s(\phi) - s(\psi)c(\phi) & s(\psi)s(\theta)s(\phi) + c(\psi)c(\phi) & c(\theta)s(\phi) \\ c(\psi)s(\theta)c(\phi) + s(\psi)s(\phi) & s(\psi)s(\theta)c(\phi) - c(\psi)s(\phi) & c(\theta)c(\phi) \end{bmatrix} \tag{3-2}$$

Utilizing (3-2), the euler angles for a 321-sequence can be determined for a given rotation matrix:

$$\begin{aligned} tan(\phi) &= \frac{T_{i2b}(2,3)}{T_{i2b}(3,3)} \\ \Rightarrow \phi &= atan2(T_{i2b}(2,3), T_{i2b}(3,3)) \end{aligned} \tag{3-3}$$

$$\theta = -asin(T_{i2b}(1,3)) \tag{3-4}$$

$$\begin{aligned} tan(\psi) &= \frac{T_{i2b}(1,2)}{T_{i2b}(1,1)} \\ \Rightarrow \psi &= atan2(T_{i2b}(1,2), T_{i2b}(1,1)) \end{aligned} \tag{3-5}$$

3.2.3 Velocity and Angular Rates

Along the position and attitude of the tracked drone, its velocity and angular rates are particularly useful for control applications. These values can be determined as the time derivative of the respective measurements.
As the position and attitude measurements of the DTrack system are subject to measurement noise, these high frequencies lead to spikes in the derivatives. The simplest way to evade their effects is by post processing the raw derivative through a

low-pass filter. Figure 3.2 below shows a side by side comparison between untreated and filtered derivatives for the velocity in X-direction.

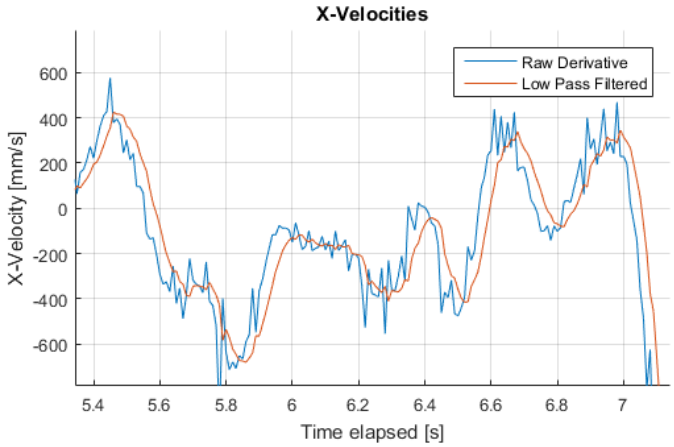


Figure 3.2: Comparison between unfiltered and filtered verlocity yielded from derivative

It can clearly be seen, that the spikes in the raw derivative are smoothed out at the cost of a small time delay. The used transfer function of the first-order low pass filter for this example is shown in Equation (3-6). The time-constant of 0.03 seconds can also be read out of Figure 3.2, as it is precisely the time-offset between the two graphs.

$$G(s)_{LP} = \frac{4.8}{0.03 \cdot s + 1} \tag{3-6}$$

The value of 0.03 seconds is chosen deliberately for the application with the DTrack system. The tracking information is broadcast at about 60Hz, so a new measurement is available around every 0.016 seconds. Using 0.03 seconds as the time constant of the filter means that the filtered output is determined from the last two measurements.

There are certainly also other means of determining the velocities and angular rates of the copter. A Kalman filter has also been tested, however as the data from the tracking system is currently the only data source, the estimator has no additional supporting data. The estimating filter therefore can also only act similarly to a low-pass filter. With a direct feedback channel from the on-board attitude measurement to the filter, the (actual) attitude combined with the motor thrust could be used to predict accelerations and therefore also the copters velocity. However, at this point other solutions were discarded due to the substantial additional complexity without definitive benefits over the simple low-pass filter solution previously introduced.

3.3 Data Interface Simulink to Android

As described in Chapter 2.3, the ADB included in the Android platform tools is utilized. ADB is a command-line tool and MATLAB is capable of executing command-line code with the *dos*-function.
ADB's TCP port-forwarding can therefore be called as a MATLAB command:

```
dos(['cd ' adb_path, ' && adb forward tcp:38300 tcp:38300']);
```

Here `adb_path` is a string-variable containing the full path to the platform-tools directory. The TCP port number 38300 has been chosen for this thesis and is hard-coded in the solution. Generally speaking any port in the range of 1024-49151 (user ports) can be used, however it should be ensured that the port is not otherwise registered or reserved by a service in order to avoid any interferences. This can be checked in the Service Name and Transport Protocol Port Number Registry[11] by the Internet Assigned Numbers Authority (IANA).

For sending TCP messages from a running Simulink simulation, the *jtcp.m*-function developed by Kevin Bartlett and published to the MathWorks File Exchange is used. The *jtcp.m* program uses Matlab's ability to call Java code to enable it to send and/or receive TCP packets. [21]
The function is subsequently used to request a TCP connection on port 38300 on the localhost computer (IP adress 127.0.0.1), because this is the port that is forwarded to the connected Android device.

```
global tcp_obj;
tcp_obj =
    jtcp('REQUEST','127.0.0.1',38300,'timeout',5000,'serialize',false);
pause(0.5); %Wait to establish connection
```

Disabling serialisation simplifies the handling of the transmitted data, but on the other hand limits read and write operations of *jtcp.m* to variables of type "int8", what is not a problem for the intended data. The function returns an object of type *jTcpObj*, which is used to handle the communication for the established connection.

The calculated values for the reference euler angles, the throttle and the state of the rotary knob and switch on the USB remote controller shall be transmitted to

[11] The complete list is available at: `https://www.iana.org/assignments/service-names-port-numbers/service-names-port-numbers.xhtml` (as of March 2018)

the "Master" device. As all of these six values are floating point numbers, a simple protocol has been defined to enable the distinction between the received numbers on the Android device:
The data is written in a string, with the channels separated with a line-break. Thereby the length of the data can be arbitrary, as the receiving peer just reads until it receives a line-break. Correct allocation of the floating point data to its channel on the receiver side is ensured by adding a one-character identifier in front of the data.
The MATLAB code below shows the short and simple way of sending all channels to the "Master" device via TCP. The floating point accuracy in this example is four decimals, but any precision could be applied here.

```
global tcp_obj; %Get access to the global jTcp object
data=sprintf('T%.4f\nY%.4f\nP%.4f\nR%.4f\n5%.4f\n6%.4f\n', ...
            throttle_in,yaw_in,pitch_in,roll_in,ch5_in,ch6_in);
data=int8(data); %Convert to int8
jtcp('WRITE',tcp_obj,data); %Send Data
```

At the end of operations the TCP connection should be terminated and also the port forwarding on ADB should be removed. All these operations can again be carried out in a MATLAB script:

```
global tcp_obj; %Get access to the global jTcp object
tcp_obj = jtcp('CLOSE',tcp_obj); %Terminate Connection
pause(0.5); %Wait to be sure
clear tcp_obj; %Delete the global variable
%Remove the TCP port forwarding
dos(['cd ' adb_path, ' && adb forward --remove-all']);
```

In order to use the above described functionality in Simulink the *WRITE* part of the code is implemented in a function that is called trough the *Interpreted MATLAB Fcn*-Block at simulation run-time. The establishing connection and ADB port forwarding part is implemented as callback during initialisation of the simulation, whereas the termination part is called at the end of the run.

4 Android Flight Controller

This chapter is dedicated to explaining the Android developments made in the frame of this thesis. The work extends the *Flight* application by the PCESS research group as introduced in Chapter 1.3.2, by a new flight controller. In Chapter 4.1 the implementations for the "Master" device (The *ControllerUSBRemoteFragment.java*) are presented, whereas Chapter 4.2 focuses on the flight controller algorithms running on the "Slave" device mounted to the copter (The *ControllerUSBRemote.java*).
All developments for Android are written in Java and carried out in Android Studio (version 3.0).

4.1 Master Device

4.1.1 Receiving Data from Simulink

Receiving data from the running Simulink Simulation is realised using TCP port forwarding. The Android device therefore has to read on the forwarded port. To ensure uninterrupted receiving, the functionality is run in a separate thread.
In the minimal example below, the necessary steps to establish a *InputStreamReader* on the utilized TCP port 38300 are shown:

```
server = new ServerSocket(38300);
server.setSoTimeout(TIMEOUT * 2000);
TCPconnection = server.accept();
socketInBuffer= new BufferedReader(new
    InputStreamReader(TCPconnection.getInputStream()));
```

After successful connection, a method generating events from the receiving data can be continuously called from the established thread.

```
if (TCPconnection!=null) {
     //Successful Connecton!
     connected = true;
    //Generate USB Events while connected
    while (connected & !USB_Thread.isInterrupted() & !server.isClosed()){
          usb_event();
    }
}
```

4.1.2 Custom SimpleEvents

The SimpleEvents, as part of the framework developed by the PCESS research group and introduced in Chapter 1.3.2, offer an extremely convenient way to share data between Android devices.
The custom SimpleEvents created for the interface between the two Android devices in the frame of this thesis are organised in the following way:

- The event of type: *TYPE_USB_REMOTE* contains the essential data for drone-flight. Its data array therefore contains the commands for throttle, yaw, pitch and roll in this order: [*throttle*, *yaw*, *pitch*, *roll*].
- The event of type: *TYPE_USB_CH5* contains the position of the rotary knob as a single floating point value.
- The event of type: *TYPE_USB_CH6* contains the position of the two point switch on the RC as a single floating point value.

The events are created and dispatched in the previously introduced method *usb_event()*. This method also reads the TCP input stream from the *BufferedReader*:

```
try {
    inSt = socketInBuffer.readLine();
} catch (IOException e) {
    e.printStackTrace();
    Log.d("USB READ ERROR", "Readline not successful");
}
```

The previously introduced protocol of delimiting the data from the different channels with a line-break comes in particular handy at this point, as it allows for convenient automatic distinction of the message part by utilizing Javas' *readLine()* command.
The *readLine()* command delivers a received data as a String. According to the defined protocol, the channel is encoded in the first character. Therefore the next step is separating the first character from the data in the message:

```
Channel = inSt.substring(0, 1);
ChannelData = Float.parseFloat(inSt.substring(1));
```

At this point a simple *switch-case* logic sorts the received data in their respective SimpleEvents data-arrays. Upon receiving all the data for a type of event, it is dispatched for all listeners. Below is an example for dispatching the *TYPE_USB_REMOTE* event, other types work accordingly.

```
1 //All Channels needed for USB_Remote Event Received!
2 time = SystemClock.elapsedRealtimeNanos();
3 simpleEventSource.dispatchEvent(new
    SimpleEvent(SimpleEvent.TYPE_USB_REMOTE, time, dataSet));
```

At this point is is extremely important to understand, that the frequency at which the custom SimpleEvents are created and dispatched can in theory be arbitrarily high. However it has been found, that frequencies above about 100Hz are very likely to cause crashes of the Android operating system. The way this is counteracted within the setup of this thesis is limiting the rate at which the data is sent from Simulink to Android under 100Hz.

4.1.3 Human Interface Design

On the GUI for the "Master" device the connection to the "Slave" device via Wifi and the connection to the PC via USB are managed. Therefore there are buttons for connecting and disconnecting the USB connection as well as a button disconnecting both, the Wifi and USB connection. Figure 4.1 below shows the layout of the GUI.

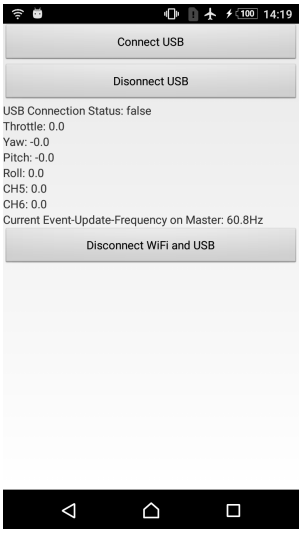


Figure 4.1: GUI Design of USBRemoteControllerFragment

Along the buttons, also the SimpleEvent values are displayed for the channels, as the "Master" device itself also registered to receive the events. The USB connection status can be inspected as well as the frequency at which the events occur at the "Master" device.

4.2 On-board Attitude Controller

4.2.1 Data Acquisition

In order to be a valid on-board controller function within the PCESS SDK, as described in Chapter 1.3.2, the *ControllerUSBRemote.java* has to implement the functions *getMeasurementMap*, *getRequiredEvents* and *update*. As previously introduced, the *MeasurementMap* defines where which data for the controller is stored. In the case of the on-board controller developed in the frame of this thesis, this data is the reference attitude and throttle received from the "Master" and the attitude of the "Slave". This data is allocated to the map as following:

```
1  Map<SimpleEvent, int[]> mm = new HashMap<SimpleEvent, int[]>();
2  for (String peer : getPeerDeviceIDs()) {
3       /* USB Remote Controller from the Master*/
4       SimpleEvent USBevent = new
       SimpleEvent(SimpleEvent.TYPE_USB_REMOTE, peer);
5       int[] usbMap={ 0, 1, 2, 3};
6       //0: Throttle, 1: Yaw, 2: Pitch, 3: Roll
7       mm.put(USBevent,usbMap);
8       /* Rotation vector from the Slave */
9       mm.put(new SimpleEvent(Sensor.TYPE_GAME_ROTATION_VECTOR), new
       int[] { 4, 5, 6, 7 });
10      /* Angular Velocities of the Slave */
11      mm.put(new SimpleEvent(Sensor.TYPE_GYROSCOPE), new int[] { 8, 9,
       10 });
12      break;
13 }
```

Again it can be seen here how simple the exchange of data between the connected devices really is with the PCESS SDK, as the only necessary part is adding *peer* to the SimpleEvent in order to receive the data.
The *TYPE_USB_REMOTE* event on the peer device (that is in this context the "Master") has been defined as the *RequiredEvent*. The flight controller therefore does not start and send any data to the motors until the reference data on the "Master" device has been received at least once.
The *update* method is where the actual data for the motors is calculated. Hence this is the place where the control algorithms are implemented. Different approaches are described and compared in the sections directly hereafter.

4.2.2 PID Controller

The first approach was to directly use the advanced PID controller that has been developed by the PCESS research group and described in [9]. The idea behind this approach was, that the model-free nature of the PID controller lends itself well to the application in the system with small differences in its (mechanical) properties.
The PID control law has been advanced by the PCESS research group in order to take into account that the sensor measurements on an Android system are not obtained at regular intervals, eliminate integrator wind-up and derivative kick. In its asynchronous time-discrete form, the control law that minimises the error vector between the "Master" (subscript M) and the drone (subscript D)

$$e_k = \begin{bmatrix} e_{\phi k} \\ e_{\theta k} \\ e_{\psi k} \end{bmatrix} = \begin{bmatrix} \phi_{Mk} \\ \theta_{Mk} \\ \psi_{Mk} \end{bmatrix} - \begin{bmatrix} \phi_{Dk} \\ \theta_{Dk} \\ \psi_{Dk} \end{bmatrix} \tag{4-1}$$

is:

$$u_k = K_p e_k + K_i \sum_{k=1}^{n} (t_k - t_{k-1}) e_k + K_d \frac{e_k - e_{k-1}}{t_k - t_{k-1}} \tag{4-2}$$

The derivative kick is omitted by the approximation:

$$\frac{e_k - e_{k-1}}{t_k - t_{k-1}} \approx \frac{\alpha_{Dk} - \alpha_{Dk-1}}{t_k - t_{k-1}} \tag{4-3}$$

where $\alpha_D \in \mathbb{R}$ denotes the angles vector of the drones orientation. [9]
The PID controllers for each of the copters rotational axes are decoupled, that means that K_p, K_i and K_d are chosen for each channel independently. In Table 4.1 below the gains applied in the test flights of the PCESS research group are listed.

Table 4.1: PID gains for attitude channels

PID Controller on	K_p	K_i	K_d
Roll	180	4	20
Pitch	160	4	20
Yaw	105	4	15

Also utilizing these gains, test flights have been carried out successfully in the TEAMS facility within the system setup as shown in Figure 2.1 on page 18.

Figure 4.2 below shows logged data from such a test flight, exemplary for the pitch axis. The angle measurements are in this case acquired with the DTrack tracking system.

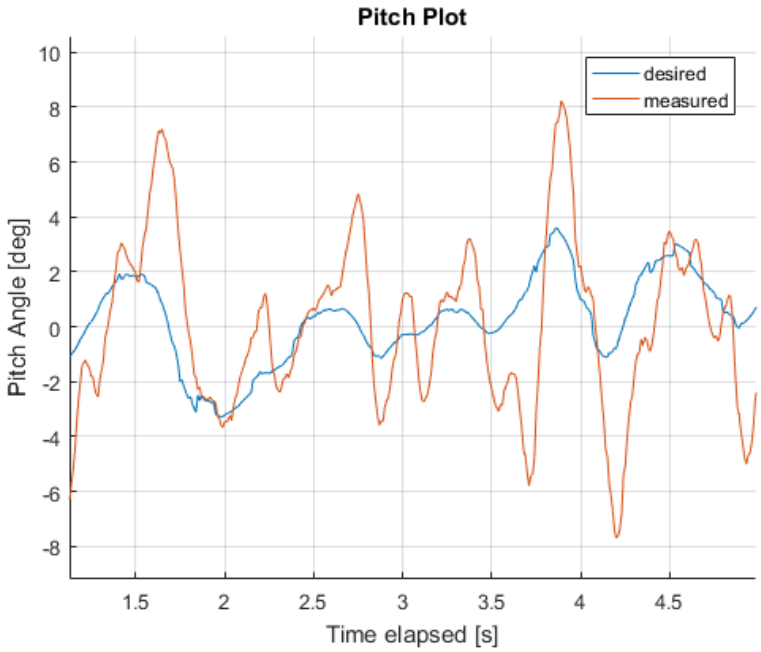


Figure 4.2: Plot from Flight log with PID attitude controller

It can clearly be seen from the flight log, that the controller can successfully track the desired value. Little time-delay between the reference and the measured value shows the good responsiveness. Therefore manual as well as autonomously controlled test flights are definitely feasible.
However the plot also shows, that there is clearly overcompensation by the controller and that there are oscillations left in the controller output. These observations match the observed behaviour by the PCESS research group in [9].
Summarizing it can be said, that the PID controller in its presented state is reacting fast to its inputs, but lacks precision. Without knowledge of the system's dynamic behaviour, finding improved values for the PID controller gains is a matter of spending a lot of time on flight tests and analysing the results. The pursued approach however is finding a simple model that adequately describes the attitude dynamics and then calculating controller gain matrices with standard methods.

4.2.3 System Model

In order to improve the on-board controller for the quadro-copter, it is essential to first create a model of its attitude dynamics. A linear state-space representation has been conceived in [9, 22]. This model structure forms the basis for the developments in this thesis. With the state vector containing the copter's euler angles with their respective rates:

$$X(t) = \begin{bmatrix} \phi(t) & \dot{\phi}(t) & \theta(t) & \dot{\theta}(t) & \psi(t) & \dot{\psi}(t) \end{bmatrix}^T \tag{4-4}$$

And the control input vector:

$$U(t) = \begin{bmatrix} U_1(t) & U_2(t) & U_3(t) \end{bmatrix}^T \tag{4-5}$$

The time-continuous linear state space system for the simplified rotational dynamics is given as:

$$\begin{aligned} \dot{X}(t) &= \underbrace{\begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}}_{\mathbf{A}} X(t) + \underbrace{\begin{bmatrix} 0 & 0 & 0 \\ \frac{l}{J_{xx}} & 0 & 0 \\ 0 & 0 & 0 \\ 0 & \frac{l}{J_{yy}} & 0 \\ 0 & 0 & 0 \\ 0 & 0 & \frac{1}{J_{zz}} \end{bmatrix}}_{\mathbf{B}} U(t) \\ Y(t) &= \underbrace{\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix}}_{\mathbf{C}} X(t) \end{aligned} \tag{4-6}$$

where $J_{ii} \in \mathbb{R}$ denotes the inertia element about the copter's axis i and $l \in \mathbb{R}$ is the distance from the center of mass to the center of each rotor in the xy plane. For the DJI F450 model, with its diagonal wheelbase (diagonal distance between the motor spinning axes) of $450mm$, because of symmetry follows $l = 0.225m$. [5]
There certainly are several different ways to determine the copters inertia. In [2] the PCESS research groups attempt to identify the rotational dynamics from flight data is described. Beside from measuring the properties, which would require a sophisti-

cated test-stand, calculating the inertia is the most traditional way. [23, p.232ff]
As the mechanical properties, most notably the mass, compared to the copter used by the PCESS research group have changed with the hardware additions (refer to Chapter 2.4) new inertia values have to be determined.
For the sake of simplicity solid cuboid inertia is assumed. Figure 4.3 below schematically shows the assumed dimensions of the representing cuboid. It has been chosen to accommodate the bulk of the copters mass in it, the far protruding but relatively light accessories (the landing legs, the propeller guard and the marker fixations) lie outside the cuboid. This was deliberately done, in order to lessen the error posed by assuming the solid cuboid with uniformly distributed mass (constant density).

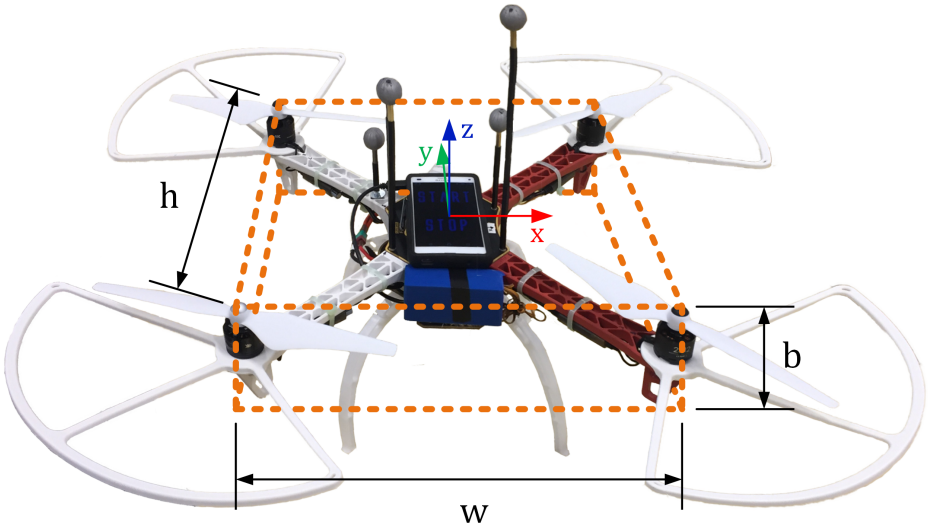


Figure 4.3: Sketch of approximated cuboid for copter inertia calculations

The inertia elements about the cuboids axes are given as: [23, p.234]

$$\begin{aligned} J_{xx} &= \frac{m}{12}(h^2 + b^2) \\ J_{yy} &= \frac{m}{12}(w^2 + b^2) \\ J_{zz} &= \frac{m}{12}(w^2 + h^2) \end{aligned} \tag{4-7}$$

For the utilized drone with its properties: $h = w \approx 0.32m$, $b \approx 0.08m$ and $m \approx 1.5kg$, inertia calculations yield:

$$\begin{aligned} J_{xx} = J_{yy} &= 0.0136\, kg\, m^2 \\ J_{zz} &= 0.0256\, kg\, m^2 \end{aligned} \tag{4-8}$$

4.2.4 LQI Controller

With the modelled system matrices introduced in Equation (4-6), the PCESS research group developed a LQI controller for the copter attitude as described in [9]. Due to the changed mechanical properties and therefore changed system matrices, new controller gains had to be calculated. Hereafter initially follows a brief introduction to the utilized methodology.
A LQI controller is an optimal state-feedback controller with incorporated integral states. Figure 4.4 below shows the general structure of a closed-loop system with LQI control.

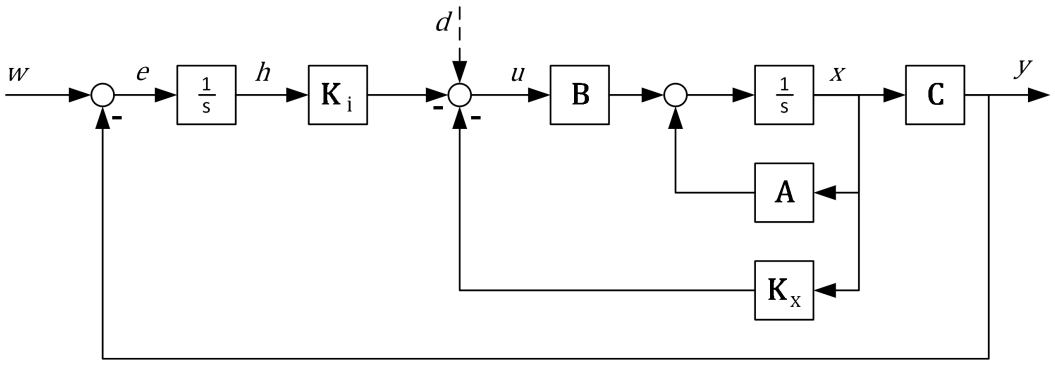


Figure 4.4: Block diagram of LQI closed loop system

For controller development it is convenient to extend the state-vector with the integral states, which yields the extended state space description:

$$\begin{bmatrix} \dot{x} \\ \dot{h} \end{bmatrix} = \underbrace{\begin{bmatrix} \mathbf{A} & \mathbf{0} \\ \mathbf{-C} & \mathbf{0} \end{bmatrix}}_{\hat{\mathbf{A}}} \begin{bmatrix} x \\ h \end{bmatrix} + \underbrace{\begin{bmatrix} \mathbf{B} \\ \mathbf{0} \end{bmatrix}}_{\hat{\mathbf{B}}} \mathbf{K} \tag{4-9}$$

An output equation is not needed here, as the relationship $y = \mathbf{C}\,x$ is already included in the controller equation. [24, p.193]
The optimal controller for the extended system is now found with the Riccati design method ([24, p.215ff]). In this design approach, the controller of the form

$$u = -\mathbf{K}\,z = -\begin{bmatrix} K_x & K_i \end{bmatrix} \begin{bmatrix} x \\ h \end{bmatrix} \tag{4-10}$$

minimizes the cost function:

$$J(u) = \int_0^\infty \left\{ z^T(t)\mathbf{Q}z(t) + u^T(t)\mathbf{R}u(t) \right\} dt \tag{4-11}$$

where **Q** and **R** are symmetrical and positive definite weighting matrices for the course of the state- and actuating-variables.
$J(u)$ will be minimal for the controller gain:

$$\mathbf{K} = \mathbf{R}^{-1}\hat{\mathbf{B}}^T\mathbf{P} \tag{4-12}$$

where **P** is the symmetrical and positive definite solution of the Matrix Riccati Equation:

$$\hat{\mathbf{A}}^T\mathbf{P} + \mathbf{P}\hat{\mathbf{A}} - \mathbf{P}\hat{\mathbf{B}}\mathbf{R}^{-1}\hat{\mathbf{B}}^T\mathbf{P} = -\mathbf{Q} \tag{4-13}$$

For higher order systems the Riccati equation (4-13) is solved numerically. In [25] the "Schur method" is described as a solution approach. This algorithm has been utilized in the frame of this thesis[12].

The PCESS research group achieved the best flight performance when choosing the weighting matrices **Q** and **R** as diagonal matrices with the diagonal elements given in the vectors q and r respectively as: [9]

$$\begin{aligned} q &= \begin{bmatrix} 1.0 & 0.1 & 1.0 & 0.1 & 1.0 & 0.1 & 40.0 & 40.0 & 10.0 \end{bmatrix} \\ r &= \begin{bmatrix} 3.0 & 3.0 & 3.0 \end{bmatrix} \end{aligned} \tag{4-14}$$

The same weighting matrices are again applied and with using the extended system model in (4-9), with the matrices **A**, **B** and **C** being defined in (4-6), the feedback gain matrices $\mathbf{K_x}$ and $\mathbf{K_i}$ were found numerically to be

$$\begin{aligned} \mathbf{K_x} &= \begin{bmatrix} 2.05 & 0.53 & 0 & 0 & 0 & 0 \\ 0 & 0 & 2.05 & 0.53 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1.21 & 0.31 \end{bmatrix} \\ \mathbf{K_i} &= \begin{bmatrix} -3.65 & 0 & 0 \\ 0 & -3.65 & 0 \\ 0 & 0 & -1.83 \end{bmatrix} \end{aligned} \tag{4-15}$$

[12] Appendix D shows the MATLAB- implementation of the algorithm

With these controller gains a LQI flight controller for the on-board device has been implemented and test flights have been performed successfully in the TEAMS facility within the system setup as shown in Figure 2.1 on page 18.
In Figure 4.5 below, logged data from such a test flight is shown exemplary for the pitch axis. The angle measurements are in this case acquired with the DTrack tracking system.

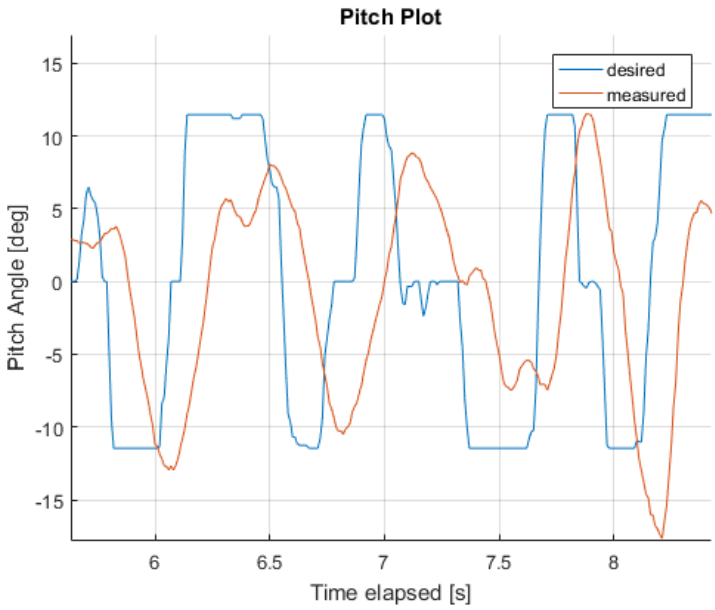


Figure 4.5: Plot from flight log with LQI attitude controller

The plot resembles the results gathered by the PCESS research group and presented in [9]. While the LQI controller is capable of tracking the reference commands much more precisely than the PID controller from Chapter 4.2.2, it is definitely inferior in terms of agility and response time. The copter reacts to reference inputs with substantial delay and, while the USB remote controller would allow for precise commands, the pilot tends to over-steer, because the drone seems non-responsive to the inputs. In their conclusions in [9], the PCESS research group suggests to find optimized PID controller gains with the results of the LQI controller calculation as starting points, as the LQI controller already has a PID-like structure.
However, as also stated in [24, p.192], the generally low control speed of a LQI controller is a known problem. The characteristic is easy to understand and clearly identifiable from the block diagram in Figure 4.4. The reference vector $\mathbf{w}$ first has to pass an integrator, before it can – via the two proportional matrices K_i and B – influence the plant. In the frame of this thesis therefore finding an improved solution for the on-board attitude controller based on the LQI design has been pursued.

4.2.5 LQI Controller Adaptation

The first impulse to make the controller faster is to increase the weighting on the integral states and therefore achieve higher gains in K_i. This subsequently in fact leads to higher control inputs and therefore definitely increases the responsiveness of the copter. At this point simulations were carried out, incorporating the system dynamics described in Equation (4-6), in order to find weighting matrices that result in a faster controller. Starting point are the previously used values, presented in (4-14). The relevant values are the integrator weights in $\mathbf{Q}$. Rising the values for the pitch and roll channels from 40.0 to 4000.0 results in the controller gain matrices to be:

$$\mathbf{K_x} = \begin{bmatrix} 8.75 & 1.04 & 0 & 0 & 0 & 0 \\ 0 & 0 & 8.75 & 1.04 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1.21 & 0.31 \end{bmatrix}; \quad \mathbf{K_i} = \begin{bmatrix} -36.51 & 0 & 0 \\ 0 & -36.51 & 0 \\ 0 & 0 & -1.83 \end{bmatrix} \tag{4-16}$$

The simulated behaviour of both, the original and the high gain controller, is displayed in Figure 4.6 below.

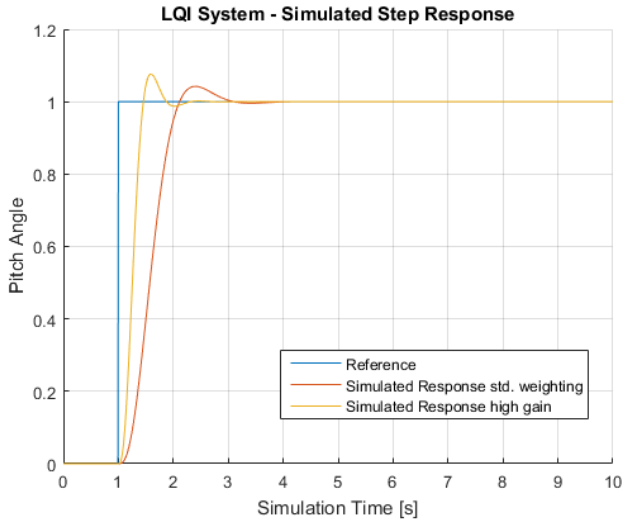


Figure 4.6: Comparison between simulated step responses of LQI system with standard vs. high gains

The weighting for the yaw channel has been left unchanged for two reasons: First of all, the yaw response does not contribute to subjective flying responsiveness and second, overreactions in the yaw axis can lead to a quick loss of altitude as the speed of two of the four rotors is drastically and abruptly reduced.

As the simulated step response for the high gain LQI controller generally shows desirable behaviour, its gain matrices (4-16) have been applied to the on-board attitude controller.
It was not possible to perform successful flight tests with this on-board controller. Figure 4.7 below shows the plot of the short data log of such a test flight. It can clearly be seen, that the drone becomes unstable.

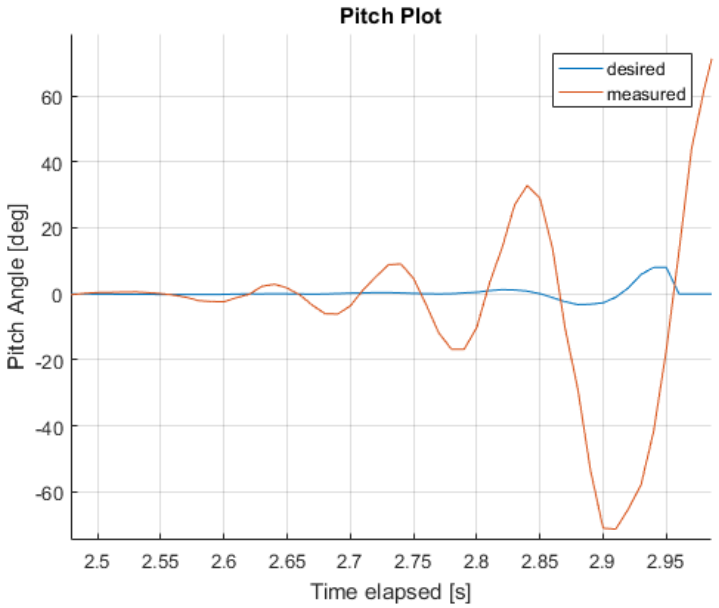


Figure 4.7: Plot from flight log with high gain LQI attitude controller (instable)

Obviously the highly simplified and linearised model of the attitude dynamics shown in (4-6) reaches its limits and the differences to the real system come to light with the aggressive controller.
However it is also vital to understand, that it is not solely the reference vector that is amplified and then fed to the plant, but rather the error vector. Therefore the copter over-reacts to very little attitude errors, even if the reference value is zero. The copter's counter-reaction to the previous over-reaction is naturally delayed by the integrator and what follows is an even greater over-steer in the opposite direction and thus leading to instability.
The observed instability of course can be counteracted with smaller values for the integrator weighting, however this would subsequently increase the response time of the copter and therefore bringing back the problem that was attempted to be solved in the first place.

At this point it is worthwhile to take a look at a simpler control system, namely the Linear-Quadratic-Regulator (LQR) controller. This controller is also an optimal controller for the given system of **A**, **B** and **C**. In contrast to the LQI controller, the system matrices are not extended with additional integral states. However, the optimal controller of the form

$$u = -\mathbf{K}_x\, x \tag{4-17}$$

is again found by solving the algebraic Riccati equation (4-13). Steady state accuracy of the LQR closed loop system is achieved by introducing a pre-filter matrix **H** into the reference channel. The values of this matrix can be found with the solution for the controller feedback gain matrix: [24, p.189]

$$\mathbf{H} = -\left[\mathbf{C}(\mathbf{A} - \mathbf{B}\,\mathbf{K}_x)^{-1}\mathbf{B}\right]^{-1} \tag{4-18}$$

Figure 4.8 below shows the general block diagram of a LQR closed loop system with the pre-filter matrix.

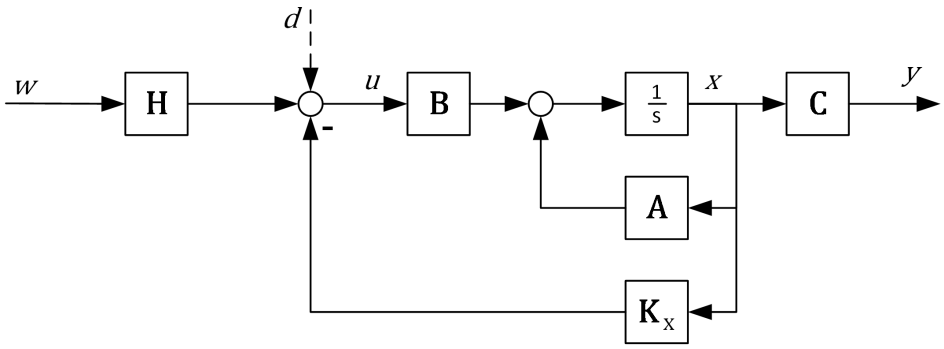


Figure 4.8: Block diagram of LQR closed loop system with pre-filter for steady-state-accuracy

With appropriate choices for the weighting matrices **R** and **Q**, the LQR system with pre-filter can follow a reference input fast and precisely. The advantage in control speed can be easily understood by comparing the block diagrams for the LQR and the LQI systems. For LQR systems, the reference vector w doesn't have to pass a integrator and thus can instantly influence the plant.
The LQR controller however is unable to counteract a constant disturbance d, that influences the system right at the input of the plant. For the present system with the quadro-copter drone, the sources of such constant disturbances are easily imaginable. Definite examples are non-perfectly horizontal mounting of the on-board Android device (extremely likely, as it is positioned by hand and fixed by velcro) or uneven

lift generated by the propellers (e.g. due to mechanical deviation of propellers, divergent motor properties or speed). So as constant disturbances to the drone are generally expected, a LQR controller with pre-filter cannot be the solution for the attitude controller problem despite its previously mentioned advantages.

At this point, the idea of combining a LQR controller with pre-filter and a LQI controller into one control-system is pursued. Figure 4.9 below shows the block diagram of the general idea with the contributions of each control method being outlined. In [20, Chapter 13.1.3] a similar controller structure is proposed as an capable solution, however without affiliating it with an optimal control problem.

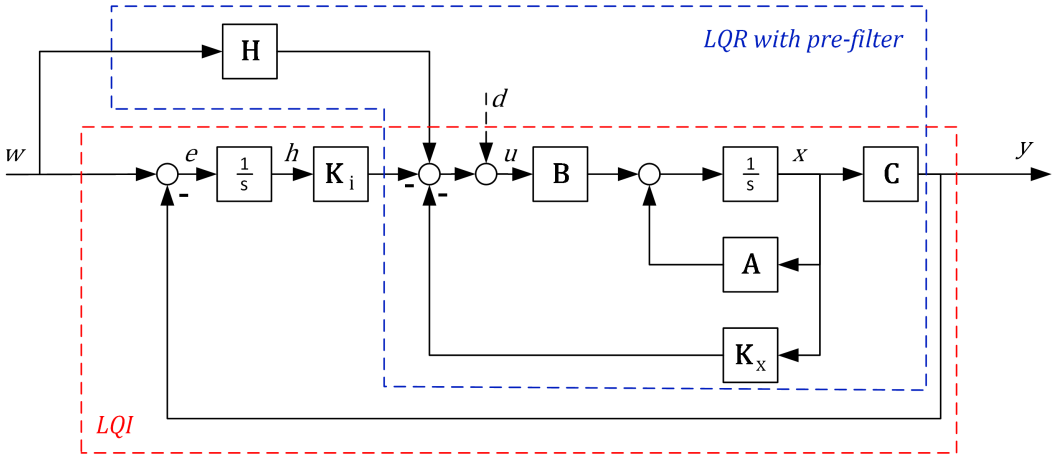


Figure 4.9: Combined control structure of LQI and LQR with pre-filter

In [20] the method is in general referred to "State space control with feed-back integration and proportional reference break-in"[13]. Linking this terminology to the optimal control method applied in this thesis, the term "LQI controller with proportional reference break-in", or in short "LQI-P", appears reasonable and therefore is used hereafter.

The general behaviour (exemplary for a step as reference) of the LQI-P system can be qualitatively described as following:

- While the integral state h is initially zero, the plant instantly receives the boosted reference value $\mathbf{H} \cdot w$.
- The plant, which is stabilized by $\mathbf{K_x}$, follows this inflated reference.

[13] dt: "Zustandsvektor-Rückführung mit integrierendem Ausgangsgrößenregler und Führungsgrößenaufschaltung"

- As long as the output has not reached the true reference w, the integrating part also contributes to increasing the plant input.
- When the output y reaches the reference input ($e = w - y = 0$) the integrating part of the system (slowly) starts to counteract the proportional feed-forward.
- At this point there is overshoot noticeable, but steady-state-accuracy ($e_\infty = 0$) is ensured by the integrating part.

In order to limit the overshoot, one additional enhancement of the system is conducted at this point. The integrator part of the LQI-P system can be replaced with a PI-structure as displayed in Figure 4.10 below. This final control structure is meaningfully referred to as a LQ-PI-P controller in this thesis.

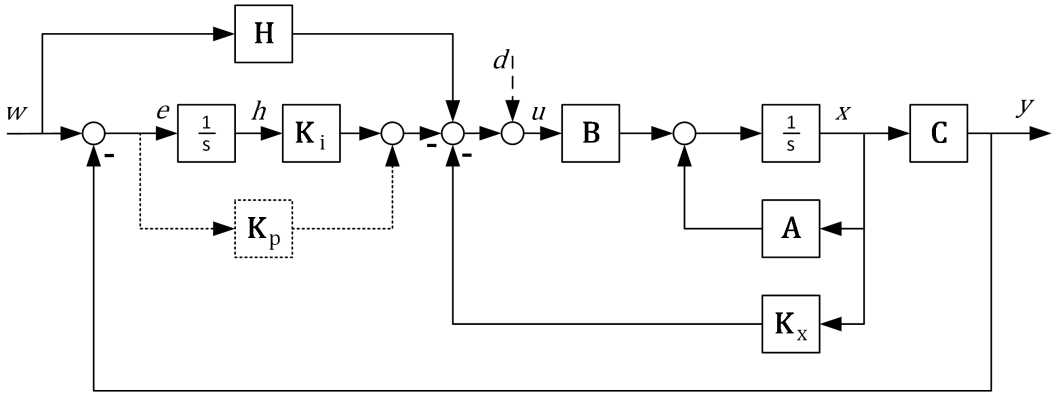


Figure 4.10: Block diagram of LQ-PI-P closed loop system

The proportional part of the PI-structure is dotted, because there are no dynamics between $\mathbf{K_x}$ and $\mathbf{K_p}$, so its contributions to the closed-loop behaviour are incorporated by adapting values in $\mathbf{K_x}$.
The controller output equation for the LQ-PI-P controller can be read out of the block diagram in Figure 4.10 and, neglecting disturbances, is given as:

$$u = -\mathbf{K_x}\, x - \mathbf{K_i}\, h + \mathbf{H}\, w \tag{4-19}$$

While the suspected behaviour of the LQ-PI-P controller seems very promising, the task of finding the matrices in Equation (4-19) still remains. "Springer - Flugregelung" ([20]) generally describes the control approach, however gives no calculation method. Hereafter the pursued approach of finding the matrices for the LQ-PI-P optimal control method is introduced by following the idea to split the problem into two separate contributions of the LQI and the LQR components.

Calculation Method of Controller Matrices

The controller matrices are calculated by applying the calculation methods for a "PI" extension of a LQI system and the pre-filter methodology for a LQR system, both developed and shown in [24, Chapter 8].
In the first step, we neglect the presence of the pre-filter **P** and find the the optimal controller for the "LQ-PI" system of the form:

$$u = -\mathbf{F}\begin{bmatrix} x \\ h \end{bmatrix} = -\begin{bmatrix} \mathbf{K_x} - \mathbf{K_p}\,\mathbf{C}, & \mathbf{K_i} \end{bmatrix}\begin{bmatrix} x \\ h \end{bmatrix} \tag{4-20}$$

The matrix **F** is found numerically by solving the algebraic Riccati equation (4-13) for the extended state space system (4-9) and by choosing appropriate diagonal matrices for **Q** and **R**. Choosing **Q** as diagonal matrix guarantees the stability of the inner loop of the LQI system. [26]

$\mathbf{K_i}$ is now uniquely defined as the right-hand part of **F**. For the left-hand part of **F** (in the following referred to as $\mathbf{F}_l$) however, we have an infinite number of solutions for $\mathbf{K_x}$ and $\mathbf{K_p}$. A practical solution is to choose $\mathbf{K_p}$ as a diagonal matrix, with non-zero elements at each position of the reference vector that is intended to be amplified. With fixing $\mathbf{K_p}$, $\mathbf{K_x}$ is now given as:

$$\mathbf{K_x} = \mathbf{F}_l + \mathbf{K_p}\,\mathbf{C} \tag{4-21}$$

For finding the feed-forward matrix **P** we now neglect the presence of the integrator[14] and calculate **H** as the suitable pre-filter matrix, that would ensure steady-state-accuracy of the "inner" part of the LQI system. This resembles a LQR system with the state feedback gain given as $\mathbf{F}_l$. **H** is therefore defined as: [20, p.575]

$$\mathbf{H} = -\left[\mathbf{C}(\mathbf{A} - \mathbf{B}\,\mathbf{F_l})^{-1}\mathbf{B}\right]^{-1} \tag{4-22}$$

For a decoupled system, as it is true for the simplified system in (4-6), **P** will be a diagonal matrix. At this point it is also possible to manually set single elements to zero, if boosting the control speed is not desired for the respective axis. This should go hand in hand with the chosen zero-elements on the diagonal of $\mathbf{K_p}$.

[14] This assumption is also considered valid in [20, p.581]

Results

Following this introduced method, the matrices are calculated for the approximated system of the attitude dynamics of the copter. The same weighting matrices used by the PCESS research group and given in (4-14) are again applied. Choosing $\mathbf{K_p}$ as:

$$\mathbf{K_p} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix} \tag{4-23}$$

yields from the previously introduced calculation method:

$$\begin{aligned} \mathbf{K_x} &= \begin{bmatrix} 3.05 & 0.53 & 0 & 0 & 0 & 0 \\ 0 & 0 & 3.05 & 0.53 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1.21 & 0.31 \end{bmatrix} \\ \mathbf{K_i} &= \begin{bmatrix} -3.65 & 0 & 0 \\ 0 & -3.65 & 0 \\ 0 & 0 & -1.83 \end{bmatrix} \\ \mathbf{H} &= \begin{bmatrix} 2.05 & 0 & 0 \\ 0 & 2.05 & 0 \\ 0 & 0 & \mathbf{0} \end{bmatrix} \end{aligned} \tag{4-24}$$

The element $H(3,3)$ is manually set to zero, as with the choice of $\mathbf{K_p}$ it was decided that the control speed of the yaw axis shall not be accelerated.

Note: The reasoning behind this is, that with a quadro-copter, the yaw axis is actuated by varying the speed of two of the four motors with the same rotational direction. Therefore with a fast jump in the reaction to a yaw command, the copter could loose altitude quickly. As a result, "sluggish" behaviour on the yaw channel is even desirable.

Another simulation is conducted with the calculated results. In Figure 4.11 below, the results of a simulated step response of the LQ-PI-P closed loop system are displayed along the LQI controller step response as comparison.

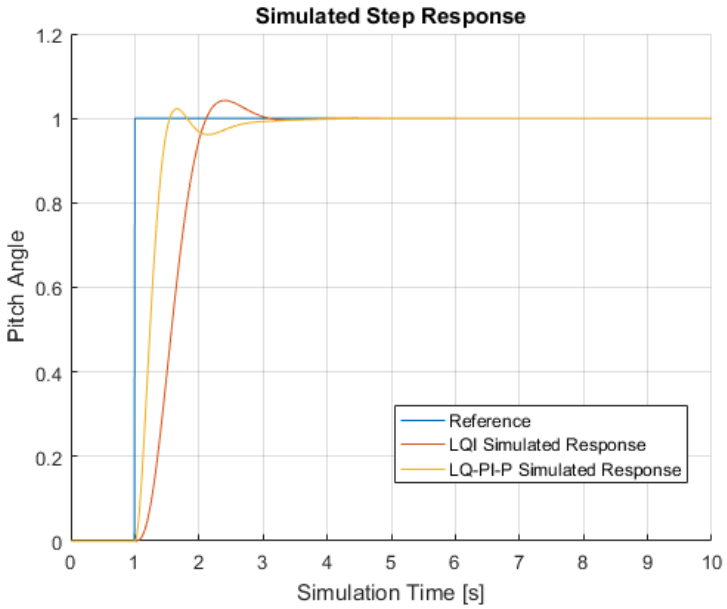


Figure 4.11: Comparison between the simulated step response of LQI and LQ-PI-P system

The simulation results definitely support the intended controller behaviour, with fast reaction time and steady-state-accuracy obtained with small controller gains. There is little overcompensation, however time elapsed until the final reference value is reached is almost identical to the LQI controller, in the form as applied by the PCESS research group.
With the calculated results for the gain matrices in (4-24) a new on-board attitude controller is implemented with the vectors defined as:

$$\begin{aligned} w(t) &= \begin{bmatrix} \phi_M(t) & \theta_M(t) & \psi_M(t) \end{bmatrix}^T \\ x(t) &= \begin{bmatrix} \phi_D(t) & \dot{\phi}_D(t) & \theta_D(t) & \dot{\theta}_D(t) & \psi_D(t) & \dot{\psi}_D(t) \end{bmatrix}^T \\ h(t) &= \begin{bmatrix} I_{e\phi} & I_{e\theta} & I_{e\psi} \end{bmatrix}^T \end{aligned} \tag{4-25}$$

where $I_{e\alpha}$ refers to the integral state of the control error for the respective channel. The values for the state vector $x(t)$ are obtained by on-board attitude measurements of the Android device.
Successful test flights have been performed in the TEAMS facility within the system setup as shown in Figure 2.1 on page 18. The LQ-PI-P controller showed that it held up to the expectations raised by the simulation results. In Figure 4.12 below logged data from such a test flight is shown exemplary for the pitch axis. The angle measurements are in this case acquired with the DTrack tracking system.

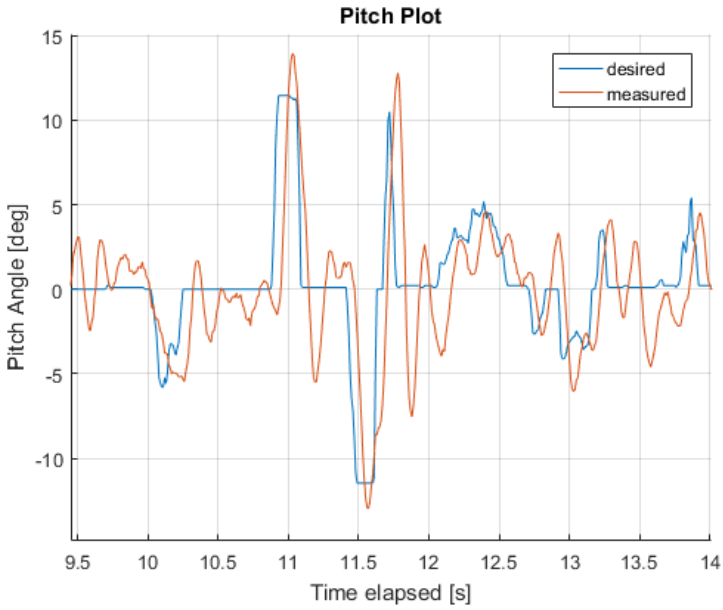


Figure 4.12: Plot from flight log with LQ-PI-P on-board attitude controller

The controller is extremely successful in following the reference signal with little delay and with precision. The LQ-PI-P controller therefore is applied as the on-board attitude controller for the further work in the frame of this thesis.

5 Guidance Development

With a precisely working on-board attitude control for the copter it is now possible to create a new overlaid control loop, which is closed by the position information measured by the optical tracking system in the TEAMS facility. Figure 5.1 below schematically shows the intended layers of the two control loops.

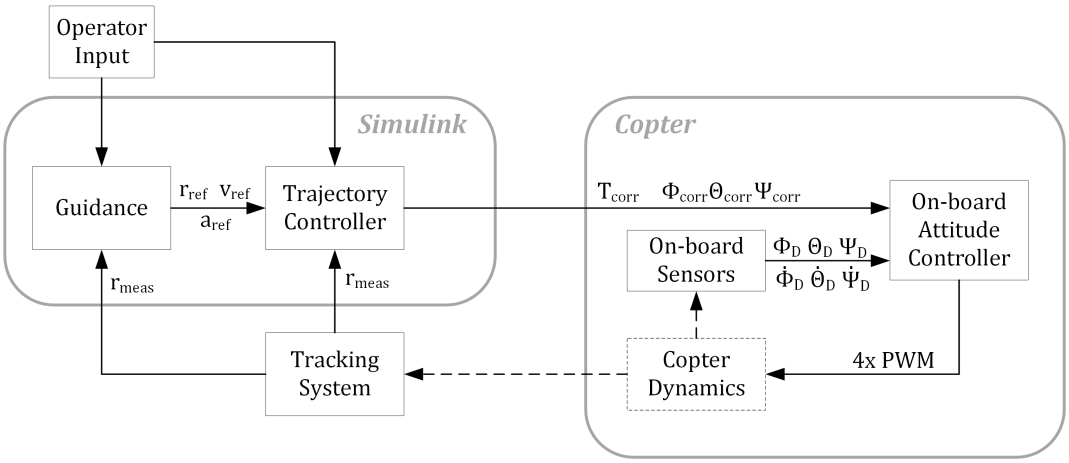


Figure 5.1: Layers of control for trajectory following

As shown in the block diagram, the approach pursued is to calculate a corrective throttle command and corrective euler angle commands in Simulink and – by utilizing the system interfaces as described in Chapter 2.3 – therefore control the drones position in the room.

In Chapter 5.1 the development of a controller that holds the copter static at one position is described. This first controller also serves as a proof for the viability of the overall layered closed loop system as depicted in Figure 5.1.
In Chapter 5.2 an advanced controller, capable of performing a flight between coordinate points is introduced. This consequently enables the application of more complex guidance algorithms, e.g. the polynomial guidance as introduced in Chapter 5.3.

5.1 Position Controller

5.1.1 Approach

The task of controlling the 3D-position of the drone is divided into two parts:

- Controlling the horizontal movement
- Controlling the vertical movement (the altitude)

When developing the linear state space models for the copter's translational dynamics, a number of simplifications and assumptions have been made.
It is assumed that the copter is moving slowly enough that aerodynamic drag forces can be neglected. In order to simplify calculations regarding the direction of the general thrust, it is assumed that the resulting thrust force of all four rotors T_{zb} is always directed along the positive Z-axis in the copter's body frame. At the same time, the general magnitude of the thrust along the positive Z-axis in the room frame of the TEAMS facility is always assumed to be $T_{zr} \approx m \cdot g$, where m is the mass of the copter and g is the gravitational acceleration of the earth ($\approx 9.81 \frac{m}{s^2}$). This subsequently means, that the copter is always assumed to be close to a hovering state.

It can be seen from the sketch in Figure 5.2, that with the pre-condition of a constant force of $T_{zr} = m \cdot g$, any rotation of the copter around its body-fixed X- or Y-axis only results in a force, that is parallel to the rooms X/Y-Plane.

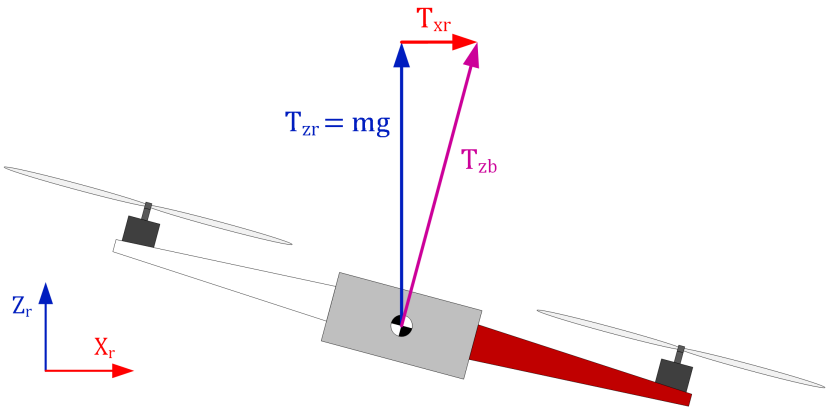


Figure 5.2: Sketch of simplified forces for copter hover flight

5.1.2 Horizontal Position Controller Development

With the previously introduced findings following the general assumptions, a simple linear state space representation for the horizontal movement of the copter in the room can be deducted as a starting point for the controller developments.

$$\begin{bmatrix} \dot{p}_{xr} \\ \dot{v}_{xr} \\ \dot{p}_{yr} \\ \dot{v}_{yr} \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} p_{xr} \\ v_{xr} \\ p_{yr} \\ v_{yr} \end{bmatrix} + \begin{bmatrix} 0 & 0 \\ \frac{1}{m} & 0 \\ 0 & 0 \\ 0 & \frac{1}{m} \end{bmatrix} \begin{bmatrix} T_{xr} \\ T_{yr} \end{bmatrix} \tag{5-1}$$

It can be seen, that the copter's mass is included in the input matrix of the state space representation in Equation (5-1). Having this very specific property of the copter as part of the outer control loop has been deemed disadvantageous at this point. However, the previously made assumptions allow for an easy normalisation of all involved forces with the fixed force $T_{zr} = m \cdot g$. This yields for the normalized horizontal forces:

$$\begin{aligned} u_{xr} &= \frac{T_{xr}}{m \cdot g} \\ u_{yr} &= \frac{T_{yr}}{m \cdot g} \end{aligned} \tag{5-2}$$

While these normalisations are certainly useful to make the developed controller more universal, the state space description has to be modified further, in order to allow an operator to specify the target point of the copter. Therefore the position-variable in the state-vector is replaced by the deviation from the specified coordinates:

$$\begin{aligned} dp_{xr} &= p_{xr} - r_{xr} \\ dp_{yr} &= p_{yr} - r_{yr} \end{aligned} \tag{5-3}$$

Where $\begin{bmatrix} r_{xr} & r_{yr} \end{bmatrix}$ is the reference position for the drone (hereby the target) specified by the operator. The state vector is now again extended by the horizontal acceleration and integrating states of the position error (following the sign convention of LQI controller development) in order to achieve greater precision. The time-discrete state space representation with the sampling time Ts is:

$$\begin{bmatrix} dp_{xr}[k+1] \\ v_{xr}[k+1] \\ a_{xr}[k+1] \\ dp_{yr}[k+1] \\ v_{yr}[k+1] \\ a_{yr}[k+1] \\ I_{dpxr}[k+1] \\ I_{dpyr}[k+1] \end{bmatrix} = \begin{bmatrix} 1 & Ts & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & \frac{Ts}{2} & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & Ts & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & \frac{Ts}{2} & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ -Ts & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & -Ts & 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} dp_{xr}[k] \\ v_{xr}[k] \\ a_{xr}[k] \\ dp_{yr}[k] \\ v_{yr}[k] \\ a_{yr}[k] \\ I_{dpxr}[k] \\ I_{dpyr}[k] \end{bmatrix} + \begin{bmatrix} \frac{g\,Ts^2}{2} & 0 \\ \frac{g\,Ts}{2} & 0 \\ g & 0 \\ 0 & \frac{g\,Ts^2}{2} \\ 0 & \frac{g\,Ts}{2} \\ 0 & g \\ 0 & 0 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} u_{xr}[k] \\ u_{yr}[k] \end{bmatrix} \tag{5-4}$$

As shown in the state space representation, the update for the velocity is determined by the mean of the expected acceleration from the (normalized) force input and the previous value of the (measured) acceleration.
For the discrete-time state space equation it is possible to find an optimal controller, as described in [27, p.536f]. The optimal controller of the form

$$u(k) = -\mathbf{K}\,x(k) \tag{5-5}$$

minimises the discrete cost function

$$J(u(k)) = \sum_{k=0}^{\infty} \left(x^T(k)\mathbf{Q}x(k) + u^T(k)\mathbf{R}u(k) \right) \tag{5-6}$$

where $\mathbf{Q}$ and $\mathbf{R}$ are again symmetrical and positive definite weighting matrices for the course of the state- and actuating-variables.
The controller is found by solving the discrete-time algebraic Riccati equation:

$$\mathbf{P} = \mathbf{Q} + \mathbf{A^T P A} - \mathbf{A^T P B} \left(\mathbf{R} + \mathbf{B^T P B} \right)^{-1} \mathbf{B^T P A} \tag{5-7}$$

With the positive definite solution $\mathbf{P}$ of the equation, the controller gain is defined by:

$$\mathbf{K} = \left(\mathbf{R} + \mathbf{B^T P B} \right)^{-1} \mathbf{B^T P A} \tag{5-8}$$

For larger systems, as present here for the horizontal position controller, the discrete-time algebraic Riccati equation (5-7) is solved numerically. In the frame of this thesis the MATLAB function *DPRE*, which was developped by Ivo Houtzager and published to the MathWorks File Exchange is utilized. [28]

Good performance was achieved for the controller, when choosing the weighting matrices **Q** and **R** as diagonal matrices with the diagonal elements given in the vectors q and r respectively as:

$$\begin{aligned} q &= \begin{bmatrix} 9.0 & 3.0 & 1.0 & 9.0 & 3.0 & 1.0 & 2.0 & 2.0 \end{bmatrix} \\ r &= \begin{bmatrix} 20.0 & 20.0 \end{bmatrix} \end{aligned} \tag{5-9}$$

It should be stressed out here, that the controller output values are "punished" by choosing rather large values in **R**. This was done deliberately, because in order to achieve a steady position, it is very counter-productive if the controller over-shoots the target due to large actuating variables.
Following the same logic as for the LQI controller, the calculation result – obtained with the weighting from (5-9) – is split up in two controller gains for the state vector and the integral states respectively:

$$\begin{aligned} \mathbf{K_{x,hpos}} &= \begin{bmatrix} 0.4002 & 0.3278 & 0.0016 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0.4002 & 0.3278 & 0.0016 \end{bmatrix} \\ \mathbf{K_{i,hpos}} &= \begin{bmatrix} -0.1291 & 0 \\ 0 & -0.1291 \end{bmatrix} \end{aligned} \tag{5-10}$$

With these controller gains, the optimal horizontal normalized forces can be calculated for the current time that eliminate the deviation between the measured and the commanded horizontal position:

$$\begin{bmatrix} u_{xr}[k] \\ u_{yr}[k] \end{bmatrix} = -\mathbf{K_{x,hpos}} \begin{bmatrix} dp_{xr}[k] \\ v_{xr}[k] \\ a_{xr}[k] \\ dp_{yr}[k] \\ v_{yr}[k] \\ a_{yr}[k] \end{bmatrix} - \mathbf{K_{i,hpos}} \begin{bmatrix} I_{dpxr}[k] \\ I_{dpyr}[k] \end{bmatrix} \tag{5-11}$$

Now these determined forces in the room frame have to be converted to euler angle commands in copter body fixed coordinates. To achieve this translation, we once again take a look at the transformation matrix method and our initial assumption, that the total thrust in the body frame is only directed along the positive Z-axis. With

the *321-Rotation* sequence presented in Chapter 3.2.2 we can transform the body thrust vector in room coordinates:

$$T_r = T_{b2i}\, T_b = \begin{bmatrix} sin(\phi)cos(\psi) + cos(\phi)sin(\theta)cos(\psi) \\ cos(\phi)sin(\theta)sin(\psi) - sin(\phi)cos(\psi) \\ cos(\phi)cos(\theta) \end{bmatrix} \tag{5-12}$$

normalizing the vector again with the T_{zr} value yields

$$\begin{bmatrix} u_{xr} \\ u_{yr} \\ u_{zr} \end{bmatrix} = \begin{bmatrix} \frac{sin(\phi)cos(\psi)+cos(\phi)sin(\theta)cos(\psi)}{cos(\phi)cos(\theta)} \\ \frac{cos(\phi)sin(\theta)sin(\psi)-sin(\phi)cos(\psi)}{cos(\phi)cos(\theta)} \\ 1 \end{bmatrix} \tag{5-13}$$

From Equation (5-13), the relations between the body euler angles and the normalized horizontal forces can be obtained. With assuming small angles for pitch and roll (ϕ, $\theta < 0.14$ ($\approx 8°$), [20, p.56]) and solving for the angles yields:

$$\begin{aligned} \theta_{corr} &= u_{yr}\, sin(\psi) + u_{xr}\, cos(\psi) \\ \phi_{corr} &= u_{xr}\, sin(\psi) - u_{yr}\, cos(\psi) \end{aligned} \tag{5-14}$$

Note: To force the controller output to the valid realm of the small angle approximation, it has been experimentally determined that linear scaling of the normalized horizontal forces by multiplying them with 0.08 works best.

The calculated pitch and roll angles to reach and hold the target point in the horizontal plane are subsequently sent to the Android device on the quadro-copter, which then itself utilizes the on-board attitude controller to follow these desired angles.
In order to be able to target a three-dimensional point in the room, it is necessary to implement a second controller, that controls the drone's altitude (z-component of target point).

5.1.3 Altitude Controller Development

The altitude controller, similarly to the horizontal position controller, is also based on the general assumption, that the copter is already in a flying state that is close to hovering. A simple linear state space representation of the vertical movement can therefore also be deducted from the assumption and serves as a starting point to the controller development:

$$\begin{bmatrix} \dot{p}_{zr} \\ \dot{v}_{zr} \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} p_{zr} \\ v_{zr} \end{bmatrix} + \begin{bmatrix} 0 \\ \frac{1}{m} \end{bmatrix} \Delta T_{zr} \tag{5-15}$$

It can be seen from the state space representation, that the input is now a "force-delta" in Z-direction. To go along the horizontal position controller, it is normalized with $m \cdot g$. At this point similar steps are taken to allow for targeting an arbitrary z-coordinate in the room. The state of the position is replaced with the deviation from a reference coordinate:

$$dp_{zr} = p_{zr} - r_{zr} \tag{5-16}$$

The state vector is again extended by the vertical acceleration and the integrating state of the deviation in position to achieve greater (and also steady-state-) accuracy. For the time-discrete state space representation with the sampling Time Ts follows:

$$\begin{bmatrix} dp_{zr}[k+1] \\ v_{zr}[k+1] \\ a_{zr}[k+1] \\ I_{dpzr}[k+1] \end{bmatrix} = \begin{bmatrix} 1 & Ts & 0 & 0 \\ 0 & 1 & \frac{Ts}{2} & 0 \\ 0 & 0 & 0 & 0 \\ -Ts & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} dp_{zr}[k] \\ v_{zr}[k] \\ a_{zr}[k] \\ I_{dpzr}[k] \end{bmatrix} + \begin{bmatrix} \frac{g\,Ts^2}{2} \\ \frac{g\,Ts}{2} \\ g \\ 0 \end{bmatrix} \Delta u_{zr}[k] \tag{5-17}$$

Again the update for the vertical velocity is approximated to be the mean of the previous measured acceleration and the contribution from the force input.
From the discrete-time state space representation for vertical movement (5-17) again an optimal controller can be calculated by solving the discrete-time algebraic Riccati equation as previously introduced. The weighting for the state- and input-variables is chosen as the diagonal matrix $\mathbf{Q}$ with its diagonal elements defined in the vector q below and the scalar factor R, as we only have one input variable.

$$\begin{aligned} q &= \begin{bmatrix} 200.0 & 10.0 & 1.0 & 20.0 \end{bmatrix} \\ R &= 5.0 \end{aligned} \tag{5-18}$$

Following the same logic as for the LQI controller, the calculation result – obtained with this weighting from (5-18) – is split up in two controller gains for the vertical state vector and the integral state respectively:

$$\begin{aligned} \mathbf{K_{x,vpos}} &= \begin{bmatrix} 1.5547 & 0.6466 & 0.0032 \end{bmatrix} \\ K_{i,vpos} &= -0.4305 \end{aligned} \tag{5-19}$$

With these controller gains, the optimal vertical "delta" force is calculated for the current time that eliminates the deviation between the measured and the commanded z-position:

$$\Delta u_{zr}[k] = -\mathbf{K_{x,vpos}} \begin{bmatrix} dp_{zr}[k] \\ v_{zr}[k] \\ a_{zr}[k] \end{bmatrix} - K_{i,vpos}\, I_{dpzr}[k] \tag{5-20}$$

Now again the task remains, to translate this normalized force to a "delta" throttle signal. Assuming, that the initial throttle signal results in a hovering drone and therefore in $T_{zr} = m \cdot g$, the normalized force Δu_{zr} can be converted to a throttle command by:

$$\Delta throttle = \Delta u_{zr} \cdot \frac{throttle_{cmd}}{m \cdot g} \tag{5-21}$$

As previously mentioned, the overall thrust, that results from a throttle command is always assumed to be aligned with the body fixed Z-axis of the drone. However the force that ensures the vertical position is the thrust force in the rooms Z-direction. Therefore the pitch and roll attitude of the copter have to be taken into account for the corrective throttle command.
From the calculation results of the horizontal position controller, we have knowledge of the attitude commands for the drone. We can therefore utilize this knowledge and correct the throttle command with the attitude angles, that the drone will try to engage in the next time-step and hence feed-forward functionality is easily achieved for the throttle command that is sent to the drone:

$$throttle_{corr} = \frac{throttle_{cmd} + \Delta throttle}{cos(\phi_{corr}) \cdot cos(\theta_{corr})} \tag{5-22}$$

5.1.4 Heading Controller

The heading poses no constraint for possible movement of a quadro-copter, and therefore is of little importance for fulfilling the task of reaching a target point or following a reference trajectory. Therefore only a very simple PI-controller has been implemented. What actually is commanded by the remote controller yaw-stick input is the desired yaw-rate of the copter. The idea is, that the copter keeps turning in the desired direction as long as the control-stick is moved. As soon as it is returned to its zero position, the copter shall maintain its current heading, hence keeping the yaw-rate $\dot{\psi} \approx 0$. The PI-control law for controlling the yaw-rate is:

$$\dot{\psi}_{corr}[k] = K_{p,\,yaw}\left(\dot{\psi}_{cmd}[k] - \dot{\psi}_{meas}[k]\right) + K_{i,\,yaw}\sum_{k=1}^{n}\left((t_k - t_{k-1})(\dot{\psi}_{cmd}[k] - \dot{\psi}_{meas}[k])\right) \quad (5\text{-}23)$$

Controller gains that have experimentally proven to work well are:

$$\begin{aligned} K_{p,\,yaw} &= 0.2 \\ K_{i,\,yaw} &= 0.05 \end{aligned} \quad (5\text{-}24)$$

As described in Chapter 3.2.3, the yaw-rate measurement is currently determined by deriving the angular measurement and low-pass filtering the results. This leads to a rather noisy input for the controller, however it can be seen in the data plot in Figure 5.3, that the controller is successful in keeping the heading steady during the flight.

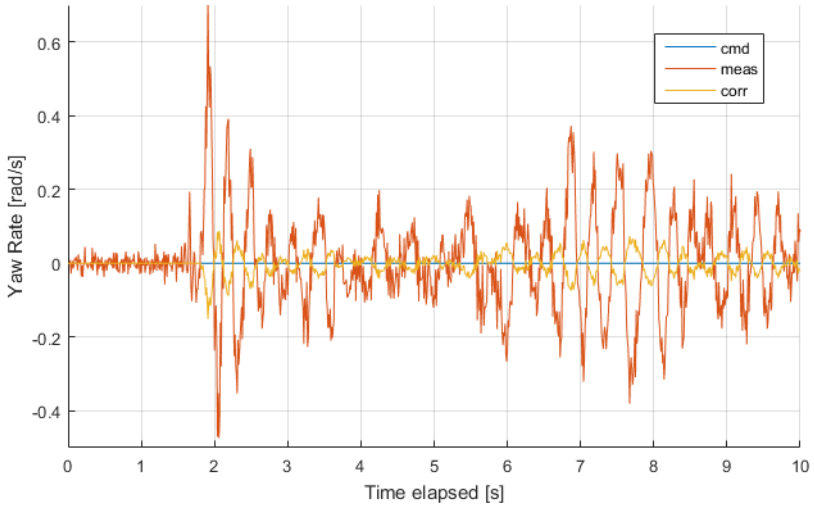


Figure 5.3: Flight data plot of yaw-rate measurement and controller output

5.1.5 Position Following

Now with position controllers for horizontal movement and altitude in place, it is possible to assign target coordinates to the copter. During flight tests the success of the developed controllers in reaching and maintaining position at a target point has been proven. In Figure 5.4 below, the position error for the three directions is shown for an exemplary flight. The take-off position is some distance away from the targeted position at $r_{ref} = \begin{bmatrix} 200 & -200 & 1000 \end{bmatrix}^T$ (in mm). It can be seen, that the controller manages to reach the target-coordinates and is afterwards able to maintain the position with accuracy of around $\pm 20\,cm$ in X/Y-direction and $\pm 10\,cm$ in Z-direction.

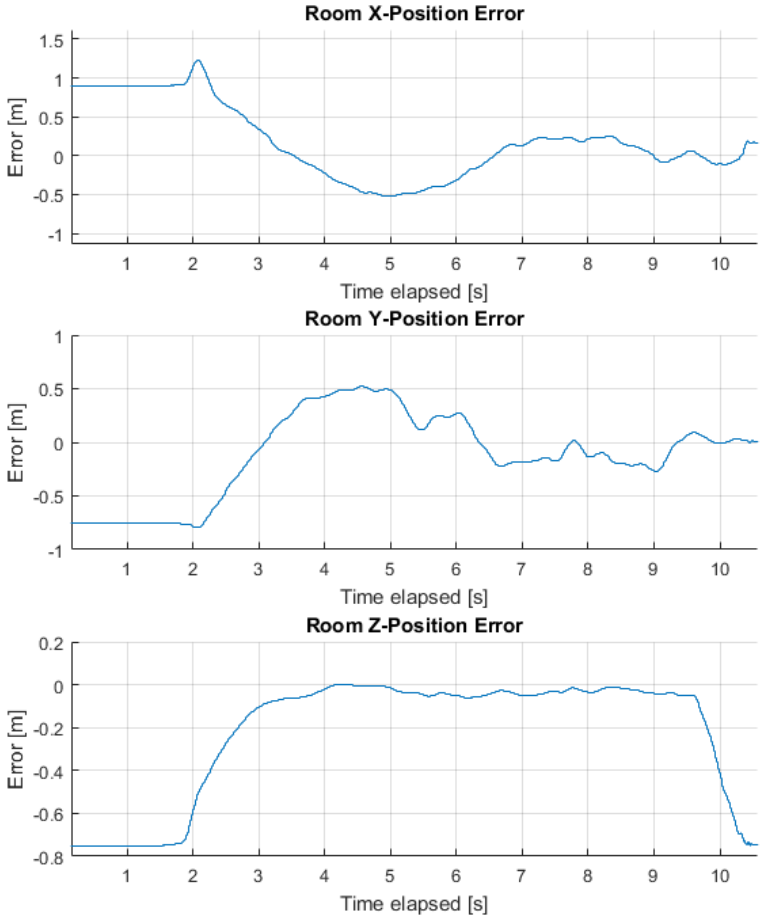


Figure 5.4: Flight data plot of position controller

The position controller could definitely also follow a path by defining a list of reference coordinates. The quadro-copter thereby would move to each coordinate by

setting it as the reference position, starting with the first coordinate in the list. Upon reaching the target position within some threshold of the current target position, the next coordinate in the list would be set as the reference position. In Figure 5.5 below, the two-dimensional XY-plot for the same test flight as given previously is shown. From this plot we can get a better idea how the position controller would perform, when pushed into the role of following a path.

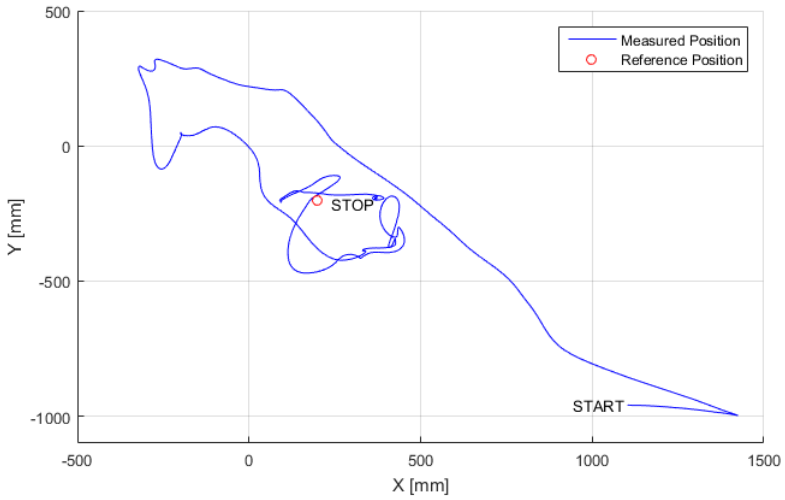


Figure 5.5: XY-plot of flight with position controller

While the position controller itself certainly would be capable of moving between predefined points, there is no possibility to control the speed of the copter on its way between them and due to the integral parts of the controller, overshooting the target points is expected. The only way of counteracting this would be by predefining a large set of reference points and thereby minimizing the distances between them. For longer or more complicated mission scenarios this results in a huge effort that has to be undertaken. All this makes the utilization of the position controller for trajectory-following tasks very disadvantageous. For that reason a dedicated controller for transitioning between target points has been developed.

The position controller however still remains useful. It can for example be used to reach the first position in a path-defining list as a coarse but robust controller.

5.2 Trajectory Following

5.2.1 Approach

The basic principle of the trajectory controller is similar to the position controller described in Chapter 5.1. The copter's movement is once again divided into horizontal and vertical movement. However for the horizontal movement between two reference points the input variables are no longer (normalized) forces in the rooms X/Y-directions, but instead expressed in parallel and perpendicular to a reference line, that connects the two X/Y-positions of the reference points.
In Figure 5.6 below a sketch of this idea is given. The cownrange (DR) direction in this reference frame is pointing from the first (or start) position to the second. The crossrange (CR) direction is is pointing 90 degrees to the left, so that it forms a regular coordinate system with the positive Z-direction of the room-frame.

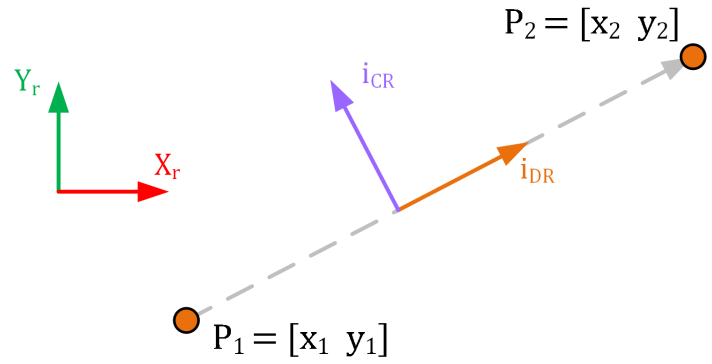


Figure 5.6: Crossrange/Downrange directions between two points

With linear algebra the unit vectors representing the CR and DR directions can be calculated from the X/Y-coordinates of the reference points:

$$i_{DR} = \frac{1}{\left\|P_2 - P_1\right\|} \begin{bmatrix} x_2 - x_1 \\ y_2 - y_1 \end{bmatrix} \tag{5-25}$$

$$i_{CR} = \frac{1}{\left\|P_2 - P_1\right\|} \begin{bmatrix} y_1 - y_2 \\ x_2 - x_1 \end{bmatrix} \tag{5-26}$$

If $P_2 = P_1$, the equations above would not result in a credible result (division by zero). However in this case, there also is no horizontal line to follow. The CR/DR-system

is in this special case expressed as unit vectors in the horizontal room coordinate directions:

$$i_{DR} = \begin{bmatrix} 1 \\ 0 \end{bmatrix} \tag{5-27}$$

$$i_{CR} = \begin{bmatrix} 0 \\ 1 \end{bmatrix} \tag{5-28}$$

The system states of the copter now can easily be converted to the CR/DR-system through the dot product of the unit vectors i_{DR} and i_{CR} and the X/Y-values of the respective state:

$$v_{DR} = i_{DR} \bullet \begin{bmatrix} v_{xr} \\ v_{yr} \end{bmatrix} \tag{5-29}$$

$$v_{CR} = i_{CR} \bullet \begin{bmatrix} v_{xr} \\ v_{yr} \end{bmatrix} \tag{5-30}$$

$$a_{DR} = i_{DR} \bullet \begin{bmatrix} a_{xr} \\ a_{yr} \end{bmatrix} \tag{5-31}$$

$$a_{CR} = i_{CR} \bullet \begin{bmatrix} a_{xr} \\ a_{yr} \end{bmatrix} \tag{5-32}$$

The crossrange position is expressed as a distance from the horizontal line, that connects the two points. The downrange position is defined as the distance to go on that line in order to reach the target.

$$p_{CR} = i_{CR} \bullet \begin{bmatrix} p_{xr} - x_2 \\ p_{yr} - y_2 \end{bmatrix} \tag{5-33}$$

$$p_{DR} = i_{DR} \bullet \begin{bmatrix} p_{xr} - x_2 \\ p_{yr} - y_2 \end{bmatrix} \tag{5-34}$$

With these definitions in place, it is possible to adapt the state space equation for horizontal movement (5-4) to the CR/DR-system and find an optimal controller for it.

5.2.2 Controller Development

The time-discrete state space representation for horizontal movement in the CR/DR-system between two target points is given with the sampling time Ts as:

$$\begin{bmatrix} p_{CR}[k+1] \\ v_{CR}[k+1] \\ a_{CR}[k+1] \\ p_{DR}[k+1] \\ v_{DR}[k+1] \\ a_{DR}[k+1] \\ I_{p,CR}[k+1] \\ I_{p,DR}[k+1] \end{bmatrix} = \begin{bmatrix} 1 & Ts & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & \frac{Ts}{2} & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & Ts & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & \frac{Ts}{2} & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ -Ts & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & -Ts & 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} p_{CR}[k] \\ v_{CR}[k] \\ a_{CR}[k] \\ p_{DR}[k] \\ v_{DR}[k] \\ a_{DR}[k] \\ I_{p,CR}[k] \\ I_{p,DR}[k] \end{bmatrix} + \begin{bmatrix} \frac{g\,Ts^2}{2} & 0 \\ \frac{g\,Ts}{2} & 0 \\ g & 0 \\ 0 & \frac{g\,Ts^2}{2} \\ 0 & \frac{g\,Ts}{2} \\ 0 & g \\ 0 & 0 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} u_{CR}[k] \\ u_{DR}[k] \end{bmatrix} \tag{5-35}$$

As shown in the state space representation, the update for the velocities is again determined by the mean of the expected acceleration from the (normalized) force input and the previous value of the (measured) acceleration.

From the discrete-time state space representation for vertical movement (5-35) an optimal trajectory controller can be calculated by solving the discrete-time algebraic Riccati equation (5-7) as previously introduced. The weighting for the state- and input-variables is chosen as the diagonal matrices **Q** and **R** with their diagonal elements defined in the vectors q and r below respectively.

$$\begin{aligned} q &= \begin{bmatrix} 17.0 & 2.0 & 0.1 & 18.0 & 5.0 & 0.1 & 1.0 & 3.0 \end{bmatrix} \\ r &= \begin{bmatrix} 20.0 & 20.0 \end{bmatrix} \end{aligned} \tag{5-36}$$

Following the same logic as for the LQI controller, the calculation result – obtained with this weighting from (5-36) – is split up in two separate gain matrices of the trajectory controller for the state vector and the integral states respectively:

$$\begin{aligned} \mathbf{K_{x,traj}} &= \begin{bmatrix} 0.8512 & 0.4913 & 0.0024 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0.9734 & 0.6019 & 0.0030 \end{bmatrix} \\ \mathbf{K_{i,traj}} &= \begin{bmatrix} -0.1793 & 0 \\ 0 & -0.3089 \end{bmatrix} \end{aligned} \tag{5-37}$$

5.2.3 Application

With the controller gains from (5-37), the optimal normalized forces in CR/DR-direction can be calculated from the control law:

$$\begin{bmatrix} u_{CR}[k] \\ u_{DR}[k] \end{bmatrix} = -\mathbf{K_{x,traj}} \begin{bmatrix} p_{CR}[k] \\ v_{CR}[k] \\ a_{CR}[k] \\ p_{DR}[k] \\ v_{DR}[k] \\ a_{DR}[k] \end{bmatrix} - \mathbf{K_{i,traj}} \begin{bmatrix} I_{p,CR}[k] \\ I_{p,DR}[k] \end{bmatrix} \tag{5-38}$$

In order to convert the trajectory controller output to forces in the room reference frame, the unity vectors are again applied:

$$\begin{bmatrix} u_{xr} \\ u_{yr} \end{bmatrix} = i_{CR} \cdot u_{CR} + i_{DR} \cdot u_{DR} \tag{5-39}$$

Now again Equation (5-14) can be applied to convert the forces to reference angles for the drone.

The transition to the CR/DR-system itself does not automatically solve the task of following a line between two points better, however it enables the possibility of specifying reference values to states, that are conductive to reaching the target point. One example is, to replace the downrange velocity value in the state vector by a difference of the measured to a desired constant downrange velocity.

Note: Extending the other state variables with a constant reference to an error state does not make sense at this point. Any constant deviation from the crossrange position is pointless and specifying a constant crossrange velocity would guide the copter away from the target. Constant reference accelerations in any direction would push the force output of the controller to its limits. Lastly any target value for the downrange position other than zero is contradictory to reaching the target point at all.

At this point, a list of reference points can be delivered to the controller. The idea is, that the coarse position controller is utilized to reach the first point (within a threshold) in the list. Then the crossrange and downrange unit vectors are calculated for the distance to the next point. Now the trajectory controller is utilized to reach the next point in the list with constant downrange velocity. Upon reaching this point (again within some threshold) the process is repeated until the last point in the list is reached. Then again the position controller is utilized to hold this point. Correct altitude is in all cases held by the altitude controller described in Chapter 5.1.3.
In Figure 5.7 and Figure 5.8 the plots from a test flight that constitutes of a return trip from $\begin{bmatrix}1100 & -800 & 1200\end{bmatrix}$ to $\begin{bmatrix}0 & 800 & 1200\end{bmatrix}$ are shown.

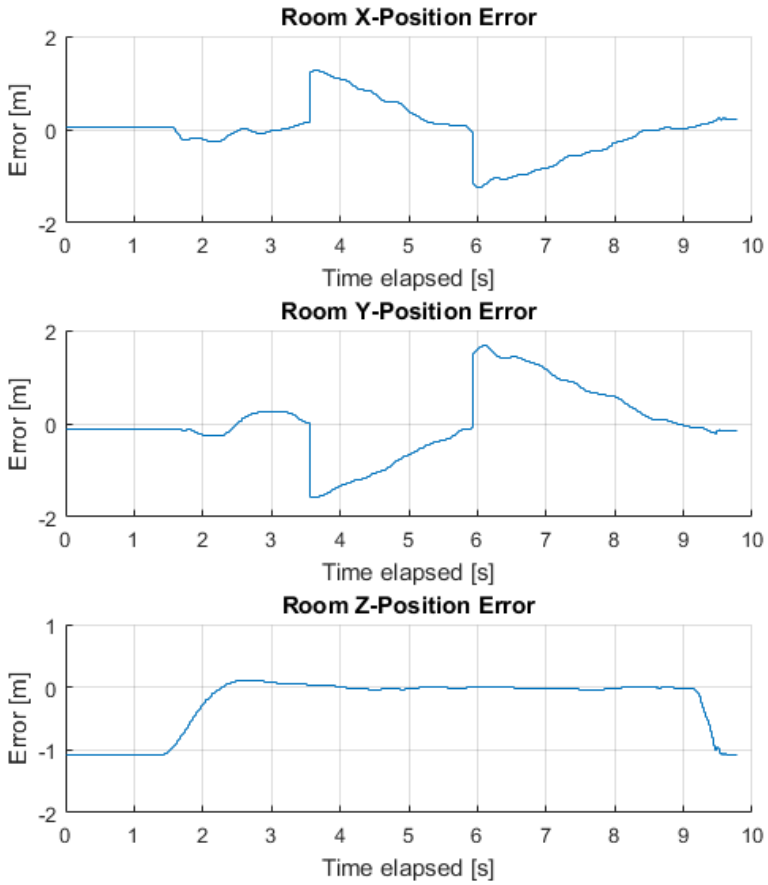


Figure 5.7: Flight data plot of trajectory controller and constant downrange velocity

It is clearly identifiable, that the start point is reached after about 3.5 s and the second point is reached after about 6 s through the jumps in the position error when the reference point is updated. Between the points, the position error is virtually

declining linearly, indicating success in maintaining a constant downrange velocity. In the XY-plot of the horizontal movement in Figure 5.8, it can clearly be seen, that there is overshoot present when reaching the target. This is easily understandable, because the controller maintains constant downrange velocity until the point is actually reached. After reaching the target point, the reference velocity instantly changes. The controller reacts with large output values, nonetheless cancelling out the velocity naturally takes some time in which the copter still moves downrange.

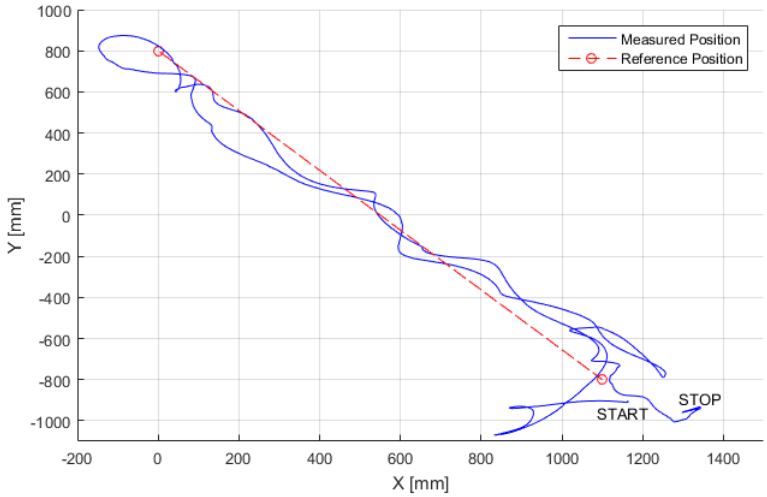


Figure 5.8: XY-plot of test flight with trajectory controller and constant downrange velocity

A much more powerful approach would be to to change the reference velocity for the drone while it is on its way between reference points. If stopping at a target point would be desired, the reference velocity could be gradually decreased before the target point and therefore avoid overshoot.
This is where more advanced guidance algorithms come into play. Their goal is to calculate a reference trajectory that connects the two coordinate points while also satisfying the boundary conditions for the points, that are defined as the states of the vehicle. One method is the polynomial guidance approach pursued in the frame of this thesis and described hereafter.

5.3 Polynomial Guidance

A very well proven guidance method is the polynomial based trajectory generation, as it was used in the Apollo missions, that landed men on the Moon and returned them safely back to Earth.
The method thereby clearly showed its applicability for planetary landing problems and hence is pursued in the frame of this thesis.

5.3.1 Approach

The reference trajectory for the final phases in lunar descent in the Apollo missions was defined as a vector polynomial function, that satisfies a two-point boundary value problem with a total of five degrees of freedom for each of the three components in the vectors. A fourth order polynomial function is the minimum order, that satisfies the constraints. With fixing the time at which the target shall be reached, the reference trajectory can be expressed as a function of time: [29, p.14]

$$\underline{R}_{RG} = \underline{R}_{TG} + \underline{V}_{TG}\, t + \underline{A}_{TG}\, \frac{t^2}{2} + \underline{J}_{TG}\, \frac{t^3}{6} + \underline{S}_{TG}\, \frac{t^4}{24} \tag{5-40}$$

where $\underline{R}_{RG}$ is the position vector on the reference trajectory at time t and $\underline{R}_{TG}$, $\underline{V}_{TG}$, $\underline{A}_{TG}$, $\underline{J}_{TG}$ and $\underline{S}_{TG}$ are the position, velocity, acceleration, jerk, and snap at the target point.
If jerk and snap are replaced with constant values, the acceleration profile from Equation (5-40) is defined as: [30, p.63]

$$a(t) = c_0 + c_1\, t + c_2\, t^2 \tag{5-41}$$

In order to obtain another degree of freedom and therefore enable considering the vehicles initial acceleration in the reference trajectory, the polynomial order for the acceleration profile has to be increased by one, which yields: [30, p.64]

$$a(t) = c_0 + c_1\, t + c_2\, t^2 + c_3\, t^3 \tag{5-42}$$

where $c_i \in \mathbb{R}$ are coefficients to be determined. By integrating Equation (5-42) twice, the velocity and position profiles are obtained:

$$\begin{aligned} v(t) &= v_0 + c_0\, t + \frac{1}{2} c_1\, t^2 + \frac{1}{3} c_2\, t^3 + \frac{1}{4} c_3\, t^4 \\ r(t) &= r_0 + v_0\, t + \frac{1}{2} c_0\, t^2 + \frac{1}{6} c_1\, t^3 + \frac{1}{12} c_2\, t^4 + \frac{1}{20} c_3\, t^5 \end{aligned} \tag{5-43}$$

Known values for the boundary conditions are $a(0) = a_0$, $v(0) = v_0$, $r(0) = r_0$, $a(t_f) = a_f$, $v(t_f) = v_f$, $r(t_f) = r_f$. Fixing the arrival time to t_f enables solving the system of linear equations for the constants:

$$\begin{aligned} c_0 &= a_0 \\ c_1 &= \frac{3}{t_f^3} \left(20\, r_f - 20\, r_0 - 8\, v_f\, t_f - 12\, v_0\, t_f + a_f\, t_f^2 - 3\, a_0\, t_f^2\right) \\ c_2 &= -\frac{6}{t_f^4} \left(30\, r_f - 30\, r_0 - 14\, v_f\, t_f - 16\, v_0\, t_f + 2\, a_f\, t_f^2 - 3\, a_0\, t_f^2\right) \\ c_3 &= \frac{10}{t_f^5} \left(12\, r_f - 12\, r_0 - 6\, v_f\, t_f - 6\, v_0\, t_f + a_f\, t_f^2 - a_0\, t_f^2\right) \end{aligned} \tag{5-44}$$

By substituting the coefficients from (5-44) in (5-42) and (5-43), the acceleration, velocity and position are obtained as a function of time from 0 to t_f.
So if the arrival time at the target is known, there will always be a trajectory that connects the two points and fulfils the boundary conditions.

5.3.2 Implementation

Using the arrival time as the fixed parameter to define the trajectory on the one hand ensures that a trajectory is found, on the other hand however, this can lead to extreme values in the reference velocities or acceleration. The polynomial trajectory calculation at this point cannot consider potential limitations of the vehicle that is supposed to follow the trajectory. Choosing a different fixed variable – e.g. the maximum velocity v_{max} – however does not directly lead to a solution of Equations (5-42) and (5-43), as the time *when* v_{max} should occur is unknown.
There is a possibility to fix other variables than the arrival time with a simple but effective work-around, though. The basic idea is, that – starting from a low value for t_f – the reference trajectory is calculated for each iteration of an increased arrival time until the requirement for the maximum velocity v_{max} (or equivalently a_{max}) is no

longer violated. This is possible because as a general rule the velocity and acceleration values decrease if a longer time is allowed for covering a given distance.
For the present application with the quadro-copter, fixing the maximum allowed reference velocity on the trajectory was chosen as the solution approach. It is important to note here, that the proposed algorithm fails to converge and deliver a credible reference trajectory, if the initial or final velocities exceed the required velocity limit. In the implementation in the frame of this thesis the velocity vectors are therefore linearly scaled down, so that their norm is lower than the specified maximum velocity. This certainly causes some error in the result, however it also means that the direction of the boundary condition vectors remains unchanged. In Figure 5.9 the basic flowchart for the guidance algorithm is shown.

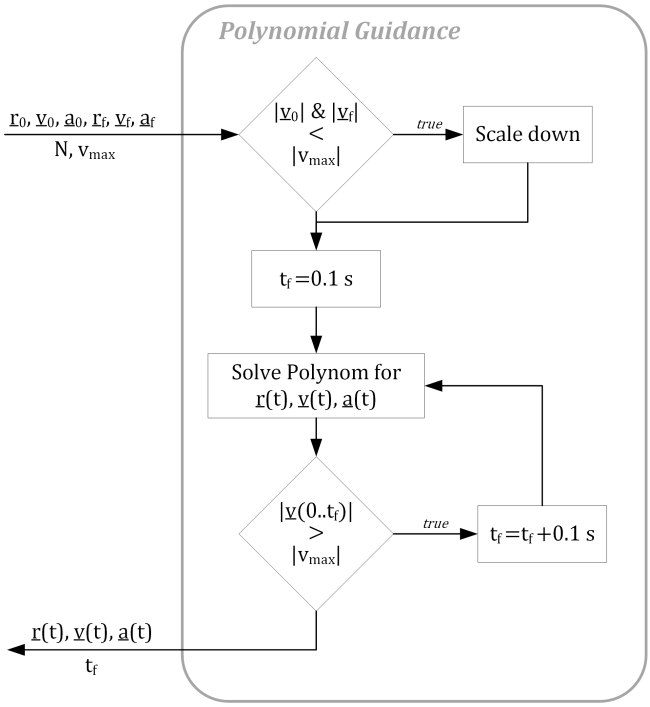


Figure 5.9: Flowchart of polynomial guidance algorithm for v_{max} boundary condition

The calculated trajectory that is returned by the algorithm contains N data points, each consisting of a three-dimensional position, velocity and acceleration vector. Also the time t_f, in which the final state will be reached within all boundary conditions is returned.
In Figure 5.10 below, the state vector plots for an exemplary trajectory is shown for a transition from $r_0 = \begin{bmatrix} 2 & 2 & 1 \end{bmatrix}^T$, $v_0 = \begin{bmatrix} 0 & -1 & 0 \end{bmatrix}^T$, $a_0 = \begin{bmatrix} 1 & 0 & 0 \end{bmatrix}^T$ to $r_f = \begin{bmatrix} 5 & 1 & 3 \end{bmatrix}^T$, $v_f = \begin{bmatrix} 0 & -1 & 0 \end{bmatrix}^T$, $a_f = \begin{bmatrix} 0 & 0 & 0 \end{bmatrix}^T$ with $v_{max} = 3\,\frac{m}{s}$ and $N = 10$.

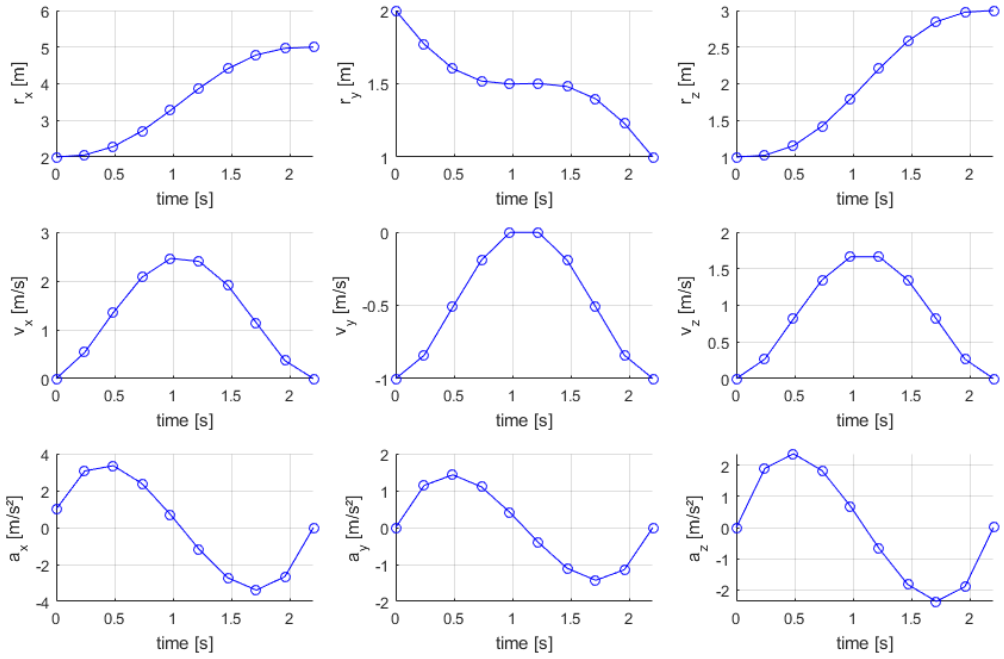


Figure 5.10: Calculated states of reference trajectory example

While staying below $v_{max} = 3\,\frac{m}{s}$ in total speed, the target is reached after 2.2 s. In the three-dimensional plot of the trajectory shown in Figure 5.11 below, the success of the algorithm can clearly be seen. The reference velocity vector of the inter-states is drawn in with red arrows, showing the change in its direction and magnitude.

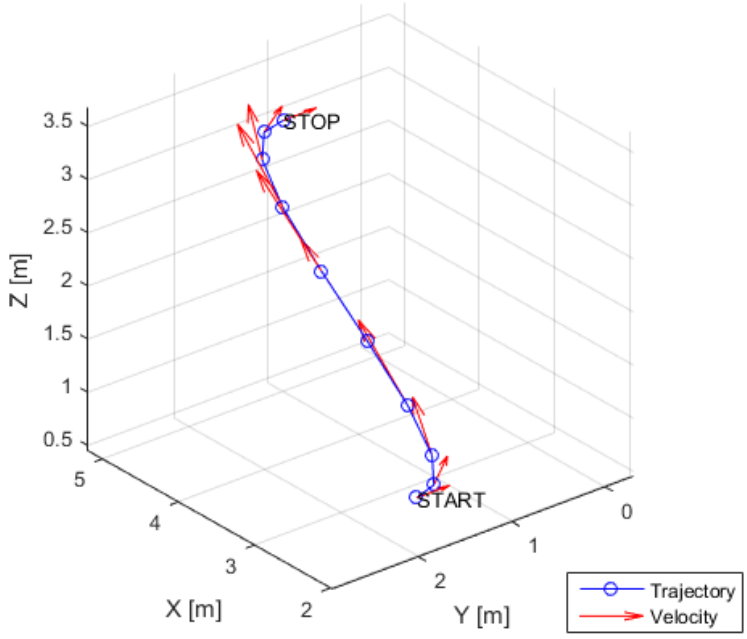


Figure 5.11: 3D-plot of reference trajectory example

5.3.3 Application

The polynomial guidance algorithm is clearly capable of generating a set of state variables, that connect two target points while as well complying with the boundary conditions of these states. The set consists of inter-states, that contain the reference state at a discrete point in time. The time in this frame is described from the initial state at $t = 0$ onwards to t_f in discrete steps of

$$Ts_{inter} = \frac{t_f}{N-1} \tag{5-45}$$

where $N > 2 \cap N \in \mathbb{N}$ is the specified number of states in the set.

The polynomial guidance algorithm now can be utilized together with the horizontal trajectory following controller and the altitude controller to guide the copter. Again, the coarse position controller is used to approach the first reference point. When the point is reached within a certain threshold, a reference trajectory is then calculated, that connects the current state of the copter with the next point in the reference list. At the discrete time points the values of the inter-states are set as references for all states of the trajectory controller and the altitude controller. To be compatible with the controller for horizontal movement, the reference states are converted to the crossrange/downrange system with the formulas developed in Chapter 5.2.1.

The adapted control law for the altitude controller in order to enable it to follow the reference trajectory is:

$$\Delta u_{zr}[k] = -\mathbf{K_{x,vpos}} \begin{bmatrix} dp_{zr}[k] \\ v_{zr}[k] - v_{z,ref}[\kappa] \\ a_{zr}[k] - a_{z,ref}[\kappa] \end{bmatrix} - K_{i,vpos}\, I_{dpzr}[k] \tag{5-46}$$

where

$$dp_{zr}[k] = p_{zr}[k] - r_{z,ref}[\kappa] \tag{5-47}$$

Similarly the control law for the horizontal trajectory controller is extended to incorporate the reference values for all states:

$$\begin{bmatrix} u_{CR}[k] \\ u_{DR}[k] \end{bmatrix} = -\mathbf{K_{x,traj}} \begin{bmatrix} p_{CR}[k] \\ v_{CR}[k] - v_{CR,ref}[\kappa] \\ a_{CR}[k] - a_{CR,ref}[\kappa] \\ p_{DR}[k] \\ v_{DR}[k] - v_{DR,ref}[\kappa] \\ a_{DR}[k] - a_{DR,ref}[\kappa] \end{bmatrix} - \mathbf{K_{i,traj}} \begin{bmatrix} I_{p,CR}[k] \\ I_{p,DR}[k] \end{bmatrix} \tag{5-48}$$

with the horizontal position errors defined as:

$$p_{CR}[k] = i_{CR} \bullet \begin{bmatrix} p_{xr}[k] - p_{x,ref}[\kappa] \\ p_{yr}[k] - p_{y,ref}[\kappa] \end{bmatrix} \tag{5-49}$$

$$p_{DR}[k] = i_{DR} \bullet \begin{bmatrix} p_{xr}[k] - p_{x,ref}[\kappa] \\ p_{yr}[k] - p_{y,ref}[\kappa] \end{bmatrix} \tag{5-50}$$

The factor $\kappa \in [2:1:N]$ represents the index of the inter-state, that is targeted next on the reference trajectory. It is increased by 1 with each time-step of Ts_{inter}.

With the controller adaptations in place, test flights are again conducted in the TEAMS facility. A $m\,x\,6$ reference matrix, containing the coordinates and velocity vectors, is defined prior to the flight. The definition is, that each row in the matrix represents one node of the trajectory. The first three numbers in this row represent the reference room coordinates x, y, z in millimetres. The second three numbers represent the respective reference velocity components in millimetres per second.
Where it would generally be possible to also specify the reference acceleration vector for the node points, it was refrained from implementing it in the frame of this thesis and an acceleration of $a_{ref} = \begin{bmatrix} 0 & 0 & 0 \end{bmatrix}^T$ is assumed for all node points in the reference matrix. With the acceleration fixed to zero in the reference nodes, this subsequently also means that the commanded reference velocity is assumed constant for the respective node and thus increasing accuracy.
During flight, the m node points are approached one after the other and the last position in the list is held by the copter. In between the nodes, the copter follows the calculated reference trajectory, that is connecting its *measured* states at $t_0 = 0$ to the next reference target.

Hereafter in Table 5.1 an exemplary set of trajectory nodes is given that incorporates different positions and reference velocities. The goal of the guidance algorithm is to shape the corner between nodes 2 and 3 as a smooth 90 degree turn. The vertical reference velocity in node 5 is set to a negative value in order to already command the copter to approach towards node 6.

Table 5.1: Exemplary matrix of nodes for a trajectory

Node #	$X_{ref}[mm]$	$Y_{ref}[mm]$	$Z_{ref}[mm]$	$V_{x,ref}\left[\frac{mm}{s}\right]$	$V_{y,ref}\left[\frac{mm}{s}\right]$	$V_{z,ref}\left[\frac{mm}{s}\right]$
1	1000	-800	1000	0	0	0
2	1000	-100	1000	0	1000	0
3	500	600	1000	-1000	0	0
4	0	600	1000	0	0	0
5	1000	-800	1000	0	0	-500
6	1000	-800	0	0	0	0

The successful execution of the test flight can be observed in the three-dimensional plot in Figure 5.12.

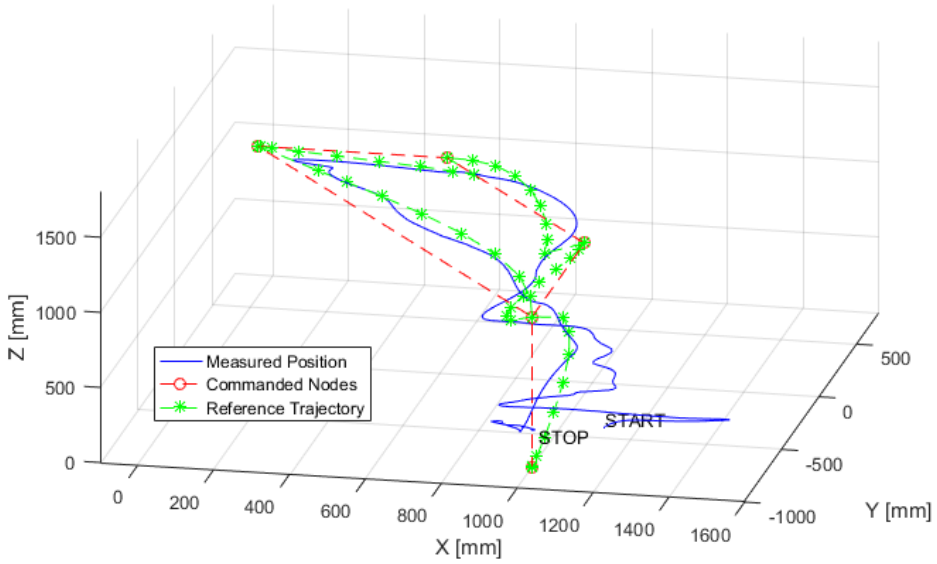


Figure 5.12: Data log 3D-plot for trajectory example

The logic is set up, so that the guidance switches to the next node in the reference matrix, as soon as the target point is reached within a certain threshold. At this point

it is important to note, that the controllers for altitude and horizontal movement are set-up as feedback controllers. This means that the the copter will definitely always lag behind its reference values for bounded controller gains. For a target point with specified reference velocity $|v_f| = 0$ this also means, that the controller tries to maintain zero velocity while not having arrived at the target point. This behaviour can clearly be observed in the target node 4 in Figure 5.12. In the current set-up of the horizontal movement controller this behaviour is counteracted to some degree by the integrating part – especially the downrange position error integrator which is weighted stronger for that particular reason – and the fact that the trajectory update is performed when reaching the threshold (currently set to $25\,cm$). The contribution of the integrating controller part however can lead to violation of the boundary conditions for velocity and acceleration.

For the Apollo lunar landing guidance, the issue of unreachable target points was also considered. The solution approach was, to update the trajectory and switch to the next target point long before reaching the target point. This means that the target point for a certain mission phase lies far beyond the portion of the trajectory that is actually flown. [29, p.14]

Summarizing it can be said that for a feedback trajectory controller there is always a compromise that has to be made between the precision for reaching the target position and the accuracy at which the boundary conditions are met.

6 Test Missions

This chapter gives examples for test missions that were performed in order to proof the viability of the developed solutions. Chapter 6.1 discusses a landing trajectory and Chapter 6.2 focuses on forming arbitrary complex forms with the polynomial guidance approach.

6.1 Landing Guidance

The polynomial guidance approach in general already has proven its suitability for spacecraft landing tasks with the Apollo missions. However, there are important aspects that have to be considered to ensure safe landing guidance.
The polynomial guidance in the implemented form is unable to consider from which direction the target point is approached. The algorithm only generates a reference trajectory that fulfils the boundary conditions. However for the example shown in Figure 6.1, that is a trajectory from $r_0 = \begin{bmatrix} 0 & 1 & 3 \end{bmatrix}^T$, $v_0 = \begin{bmatrix} 0 & 0 & 0 \end{bmatrix}^T$, $a_0 = \begin{bmatrix} 0 & 0 & -3 \end{bmatrix}^T$ to $r_f = \begin{bmatrix} 10 & 1 & 0 \end{bmatrix}^T$, $v_f = \begin{bmatrix} 0 & 0 & 0 \end{bmatrix}^T$, $a_f = \begin{bmatrix} 0 & 0 & 0 \end{bmatrix}^T$, it can clearly be seen that the vehicle would impact the ground way before reaching the target.

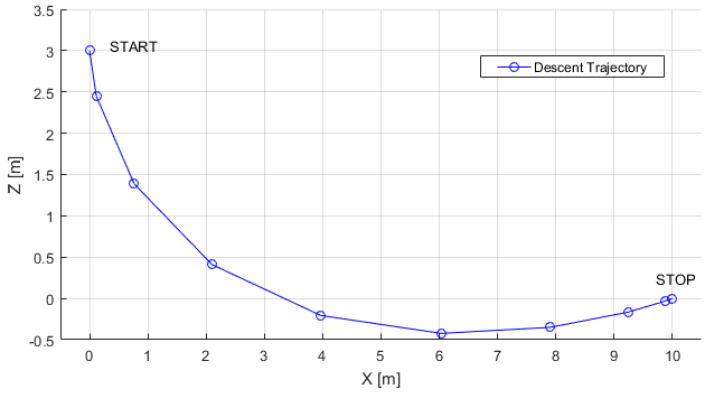


Figure 6.1: Calculated descent trajectory without angle considerations

The solution for that issue is to divide the trajectory into more phases to enforce the desired behaviour. This exact approach was also incorporated into the lunar descent trajectory for the Apollo missions. In Figure 6.2 below a typical descent path for the Apollo missions is shown. It can clearly be seen that the approach phase ends almost above the targeted landing site from where on the terminal descent is performed.

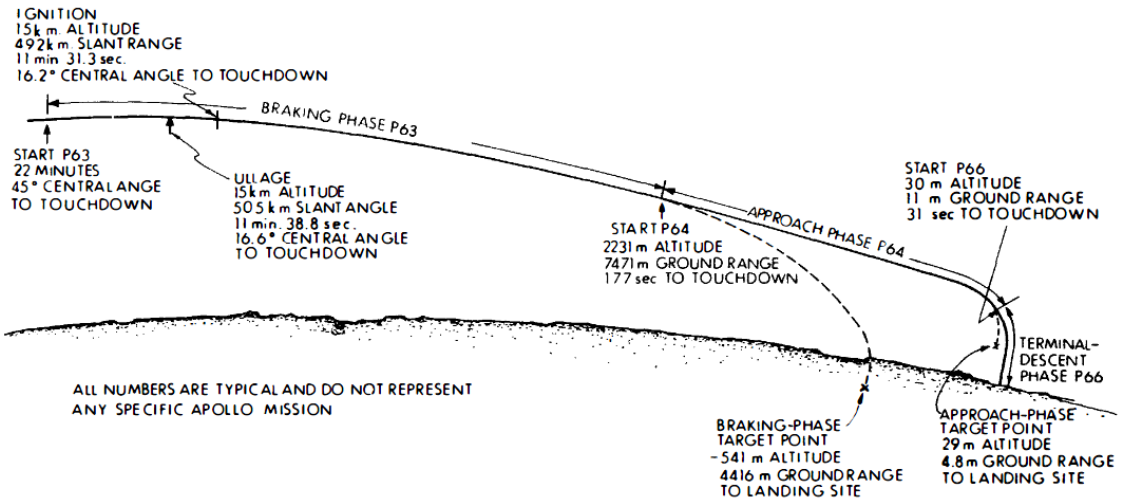


Figure 6.2: Typical lunar descent trajectory for Apollo [29]

Similar behaviour can be achieved for our exemplary trajectory when including an additional node at $r_i = \begin{bmatrix} 9.5 & 1 & 1 \end{bmatrix}^T$ with its boundary conditions set to $v_i = \begin{bmatrix} 0 & 0 & -1 \end{bmatrix}^T$ and $a_i = \begin{bmatrix} 0 & 0 & 0 \end{bmatrix}^T$. This forces the trajectory to first recover from the initial dive at $t = 0$ and then transition to a descent from right above the landing site.

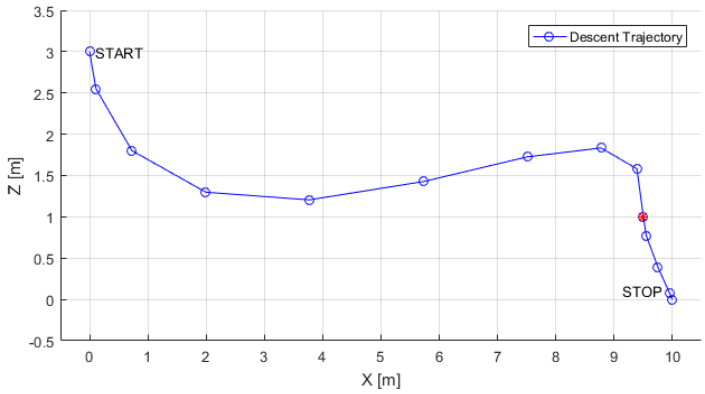


Figure 6.3: Calculated descent trajectory with additional node for impact angle constraints

For applications with the quadro-copter this is especially important, since the last measurement values are used as the starting point of the trajectory. As an operator one therefore has little control over the initial states that are incorporated in the trajectory.
In Table 6.1 the reference nodes for a landing trajectory applying this principle are given.

Table 6.1: Nodes for a landing trajectory

Node #	$X_{ref}[mm]$	$Y_{ref}[mm]$	$Z_{ref}[mm]$	$V_{x,ref}\left[\frac{mm}{s}\right]$	$V_{y,ref}\left[\frac{mm}{s}\right]$	$V_{z,ref}\left[\frac{mm}{s}\right]$
1	0	-1000	1200	0	0	0
2	1400	-1000	1000	0	0	-1000
3	1600	-1000	0	0	0	0

This reference node matrix has been utilized in a test flight. Figure 6.4 shows a plot of the calculated reference trajectory along with the measured position.

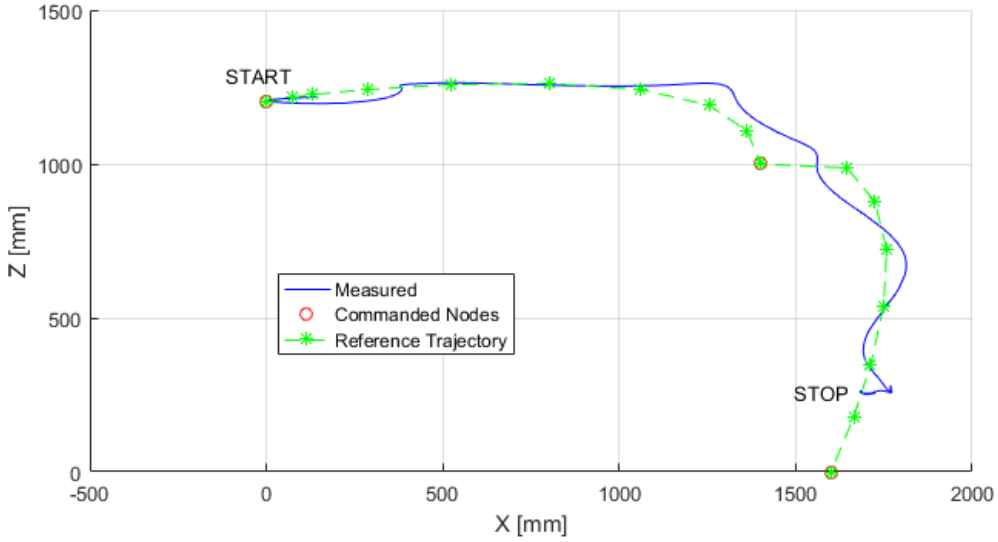


Figure 6.4: Descent trajectory test flight

The success of calculating a reference trajectory from the polynomial guidance approach for a landing scenario can clearly be seen. The accuracy of the developed trajectory controllers is also once more proven with this test-flight.

6.2 Shaping Complex Trajectories

The choice of the velocity vector in a node point of the trajectory offers a significant handle to the operator to influence the shape of a trajectory. In Chapter 6.1, it has been utilized to ensure a valid descent path. With a smart choice of the reference velocity vector for the node points, it is possible to shape complex forms with little nodes. The principle is made obvious hereafter with the example of a horizontal figure "8"-trajectory. Table 6.2 gives the nodes that are used to create the reference shape.

Table 6.2: Nodes to generate a reference trajectory shaped like a figure eight

Node #	$X_{ref}[mm]$	$Y_{ref}[mm]$	$Z_{ref}[mm]$	$V_{x,ref}\left[\frac{mm}{s}\right]$	$V_{y,ref}\left[\frac{mm}{s}\right]$	$V_{z,ref}\left[\frac{mm}{s}\right]$
1	1000	0	1000	700	700	0
2	1000	1300	1000	-1000	0	0
3	1000	0	1000	700	-700	0
4	1000	-1300	1000	-1000	0	0
5	1000	0	1000	700	700	0

A simulation clearly proves the successful shaping of the trajectory, as observable in Figure 6.5. The reference velocity vectors, shown as red arrows in the figure, serve illustrative purposes only and are not to scale.

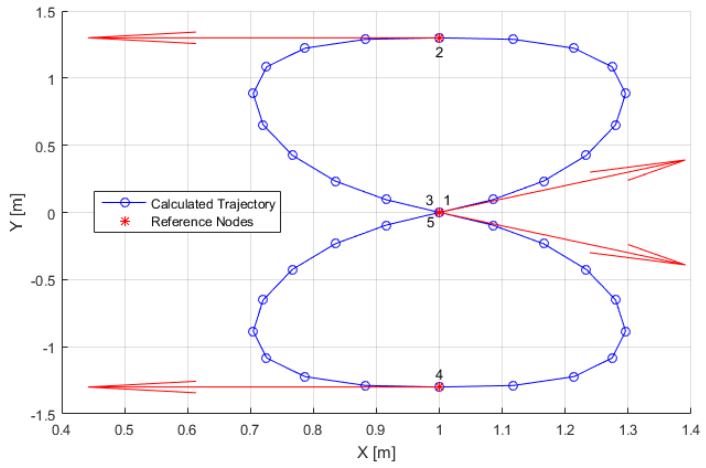


Figure 6.5: Simulation result for figure "8" trajectory

After expanding the reference matrix from Table 6.2 with start and landing nodes and include a second lap, flight tests were once again conducted. In Figure 6.6 the logged flight data is displayed along the on-flight calculated reference trajectory.

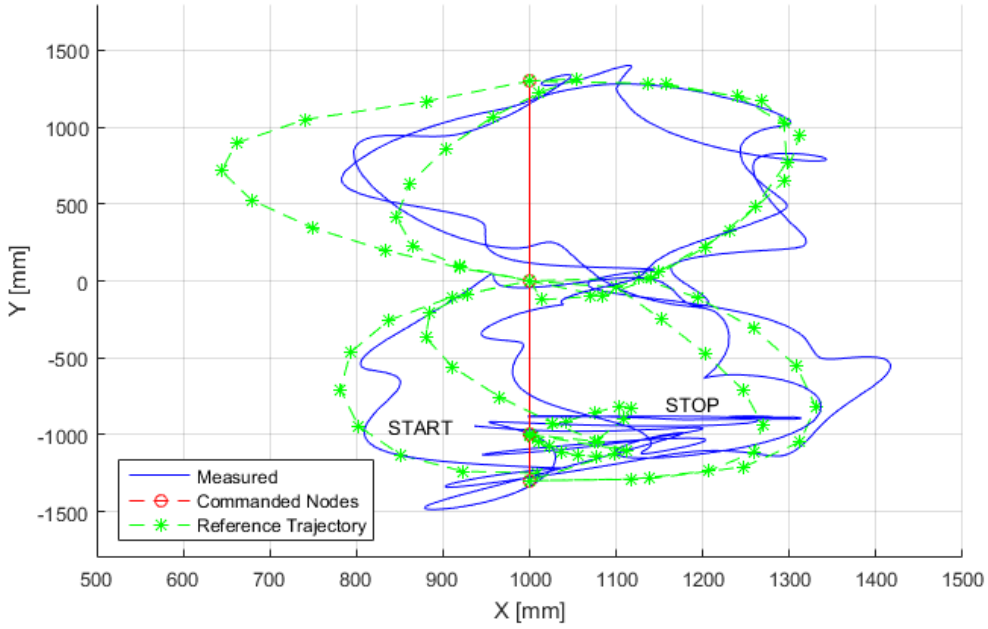


Figure 6.6: Flight result for figure "8" trajectory

The test-flight generally showed that it is also possible to calculate more complex reference trajectories with few reference nodes and the trajectory controller is capable of following it. However, it has to be pointed out, that the particular shaping of the figure eight poses challenges to the CR/DR trajectory controller in its current implementation. The trajectory controller is weighted in a way, so that it puts more emphasis into following the downrange position and velocity rather than their crossrange counterparts. Because the downrange direction is defined as a straight horizontal line connecting the node points, the reference velocity vectors for the nodes 2 and 4 in the figure eight trajectory only have a crossrange component. So either additional nodes in between or a differently weighted controller presumably would increase the accuracy of following the reference trajectory. In order to keep flight-test results consistent throughout the thesis, it was refrained form doing so.

7 Conclusions

This thesis has shown, that it is possible to experimentally verify the practicality of guidance algorithms by applying them to test-flights of a quadro-copter. The interface implementations realised to enable the flight operations in the TEAMS facility, have proven their capabilities in test-flights.
A lot of effort has been put into the improvement of the on-board attitude controller. The applied LQ-PI-P design has shown that it outperforms both former attitude controllers, the PID and the LQI design, which have been used by the PCESS research group. This is at least valid for the relevant reference angle range $< 10°$.
The developed feedback controllers for position tracking have proven that they are very capable in both, holding a referenced position and also following a reference trajectory. The remaining inaccuracy is in the ballpark of $\pm 20\,cm$ for the horizontal position and $\pm 10\,cm$ for the vertical position.
The polynomial guidance approach showed that it still offers a very capable solution for calculating a reference trajectory. The simplicity of the underlying equations even makes it possible to calculate the trajectories in-flight. Besides generating trajectories that enable a safe landing, smart combinations of reference points and velocity vectors can generate virtually every desired path shape.

8 Outlook

For future work, the first steps should be to further increase the accuracy of the developed controllers. The definite starting point for further improving the on-board attitude controller is determining the inertia of the drone with higher accuracy. This subsequently provides a more accurate state-space representation and with the proposed method of calculating the gain matrices for the LQ-PI-P design should result in a further improved attitude reference following.
To improve the accuracy of the trajectory controller, the first efforts should go into implementing an improved navigation filter, that determines the copter's velocity and acceleration with higher accuracy and averts the currently present time-delay. To implement a viable Kalman filter, a feedback path of the attitude measurements made by the "Slave" device is a prerequisite. However this should generally be possible within the present interfaces.
Including and testing other guidance algorithms than the polynomial guidance is conveniently enabled with the implementations made in this thesis. The polynomial guidance is outsourced to a MATLAB function, so switching to a different algorithm solely requires changing a single line of code.

Further along the line of future developments it is imaginable to abandon the DTrack tracking system and include an option to acquire the copter's position from e.g. GPS measurements. Obviously this configuration requires the feedback data path from the copter to the PC. This however would enable to perform flight tests outdoors at a much larger scale than in the confined spaces of the TEAMS facility.

Future evolutions of the copter to be utilized as a test-rig for guidance algorithms could also abandon the Android device as the on-board controller entirely. By utilizing an on-board IMU with higher accuracy and a real time-capable computer, this could further improve accuracy of both, the attitude and trajectory controllers.

Bibliography

[1] DLR. Institute of Space Systems, 2015. URL http://www.dlr.de/irs/en/Portaldata/46/Resources/2015_dokumente/Institutsbroschu_re_Bremen_ENG_ONLINE_251115.pdf. Brochure, Last Acessed: 12.03.2018.

[2] Mohammad Alsharif and Matthew Hölzel. System Identification of a Quadcopter's Rotational Dynamics Using Android Flight Data. In *IEEE Conference on Control Applications (CCA)*, Buenos Aires, Argentina, 19 - 22 September 2016.

[3] MHM Modellbau. Flame Wheel F450 ARF Quadrocopter-Bausatz mit E3xx Set, 2018. URL https://www.mhm-modellbau.de/part-CP.MX.540005.php. Last Accessed: 08.02.2018.

[4] Android. Sensors Overview, 2017. URL https://developer.android.com/guide/topics/sensors/sensors_overview.html. Last Accessed: 12.03.2018.

[5] DJI. Flame Wheel 450 User Manual, 2015. URL http://dl.djicdn.com/downloads/flamewheel/en/F450_User_Manual_v2.2_en.pdf. Last Accessed: 25.02.2018.

[6] Matthew Holzel. Migration from IOIO to Arduino Nano. University of Bremen, Research Group: Parallel Computing for Embedded Sensor Systems, 2017.

[7] Matthew Holzel. Using the Preinstalled Flight Software. University of Bremen, Research Group: Parallel Computing for Embedded Sensor Systems, 2017.

[8] Matthew Holzel. Developing a new Flight Controller. University of Bremen, Research Group: Parallel Computing for Embedded Sensor Systems, 2017.

[9] Mohammad Alsharif, Yunus Arslantas, and Matthew Hölzel. A Comparison between Advanced Model-Free PID and Model-Based LQI Attitude Control of a Quadcopter Using Asynchronous Android Flight Data. In *25th Mediterranean Conference on Control and Automation (MED)*, Valletta, Malta, 3 - 6 July 2017.

[10] Markus Schlotterer. TEAMS - Test Environment for Applications of Multiple Spacecraft. Booklet, Institute of Space Systems, Robert-Hooke-Straße 7, 28539 Bremen, 2017.

[11] ART Advanced Realtime Tracking GmbH. Homepage, 2018. URL https://ar-tracking.com. Last Acessed: 15.02.2018.

[12] Markus Schlotterer and Stephan Theil. Testbed for on-orbit servicing and formation flying dynamics emulation. In *AIAA Guidance, Navigation, and Control Conference*, volume AIAA 2010-8108, Toronto, Ontario Canada, 2 - 5 August 2010.

[13] ART Advanced Realtime Tracking GmbH. Camera positions for 4/6 ARTtrack2 cameras. DLR-Bremen-"Granittische " (500cm_x_400cm), 2009.

[14] MathWorks. Aerospace Blockset - Pilot Joystick, 2018. URL `https://de.mathworks.com/help/aeroblks/pilotjoystick.html`. Last Accessed: 24.02.2018.

[15] MathWorks. Simulink 3D Animation - Joystick Input, 2018. URL `https://de.mathworks.com/help/sl3d/joystickinput.html`. Last Accessed: 24.02.2018.

[16] Android. Android Debug Bridge (adb), 2018. URL `https://developer.android.com/studio/command-line/adb.html`. Last Accessed: 25.02.2018.

[17] MathWorks. Simulink Instrument Control Toolbox - TCP/IP Send, 2018. URL `https://de.mathworks.com/help/instrument/tcpipsend.html`. Last Accessed: 25.02.2018.

[18] Ettinger Elektronik-Bauelemente. Abstandsbolzen M2,5 - M3 Type E / Aussen - Aussengewinde; Katalognummer: 05.21.201, 2018. URL `https://www.ettinger.de/de/product/05.21.201`. Last Accessed: 05.03.2018.

[19] Florian Enner. MathWorks File Exchange - HebiRobotics/MatlabInput, 2018. URL `https://de.mathworks.com/matlabcentral/fileexchange/61306-hebirobotics-matlabinput`. Last Accessed: 06.03.2018.

[20] Rudolf Brockhaus, Wolfgang Alles, and Robert Luckner. *Flugregelung*, volume 3. Springer-Verlag Berlin Heidelberg, 2011. ISBN 978-3-642-01442-0.

[21] Kevin Bartlett. MathWorks File Exchange - jtcp, 2018. URL `https://de.mathworks.com/matlabcentral/fileexchange/24524-jtcp-actionstr-varargin-`. Last Accessed: 06.03.2018.

[22] Samir Bouabdallah. *Design and Control of Quadrotors with Application to Autonomous Flying*. PhD thesis, École Polytchnique Fédérale de Lausanne, 2007.

[23] Alfred Böge. *Technische Mechanik*, volume 29. Vieweg+Teubner Verlag, 2011. ISBN 978-3-8348-1355-8.

[24] Kai Michels. Control Engineering, November 2012. University of Bremen, Lecture Script.

[25] Laub, Alan. A Schur Method for solving Algebraic Riccati Equations, 1978. URL `http://dspace.mit.edu/bitstream/handle/1721.1/1301/R-0859-05666488.pdf`. Massachusets Institute of Technology, Last Accessed: 14.03.2018.

[26] Zhengping Feng, Jimao Zhu, and Robert Allen. Design of LQI Control Systems with Stable Inner Loops. December 2007.

[27] Jan Lunze. *Regelungstechnik 2*, volume 8. Springer Vieweg Verlag, 2014. ISBN 978-3-642-53943-5.

[28] Ivo Houtzager. MathWorks File Exchange - DPRE, 2018. URL `https://de.mathworks.com/matlabcentral/fileexchange/21379-dpre`. Last Accessed: 19.03.2018.

[29] Allan R. Klumpp. Apollo Lunar-Descent Guidance. Technical Report R-695, Massachusetts Institute of Technology, June 1971.

[30] Yunus Emre Arslantaş. *Development of a Reachability Analysis Algorithm for Space Applications*. PhD thesis, Universität Bremen, 2017.

Glossary

ADB Android Debugging Bridge 23, 34, 35

API Application Programming Interface 3, 7

ARF Almost Ready to Fly 3

CR crossrange 67–70, 85

DLR German Aerospace Center 1, 3, 15

DR cownrange 67–70, 85

ESC electronic speed controller 3–6, 10, 13

GNC Guidance Navigation and Control 1, 15

GUI graphical user interface 12, 13, 22, 38

HID human interface device 2

IANA Internet Assigned Numbers Authority 34

IMU inertial measurement unit 6, 87

LQI Linear-Quadratic-Integral 10, 11, 44, 46, 48–52, 54, 58, 60, 63, 69

LQR Linear-Quadratic-Regulator 49–52

MWE Minimal Working Example 9

PCESS Parallel Computing for Embedded Sensor Systems 3, 6–8, 13, 19, 36, 37, 39–46, 53, 54, 86

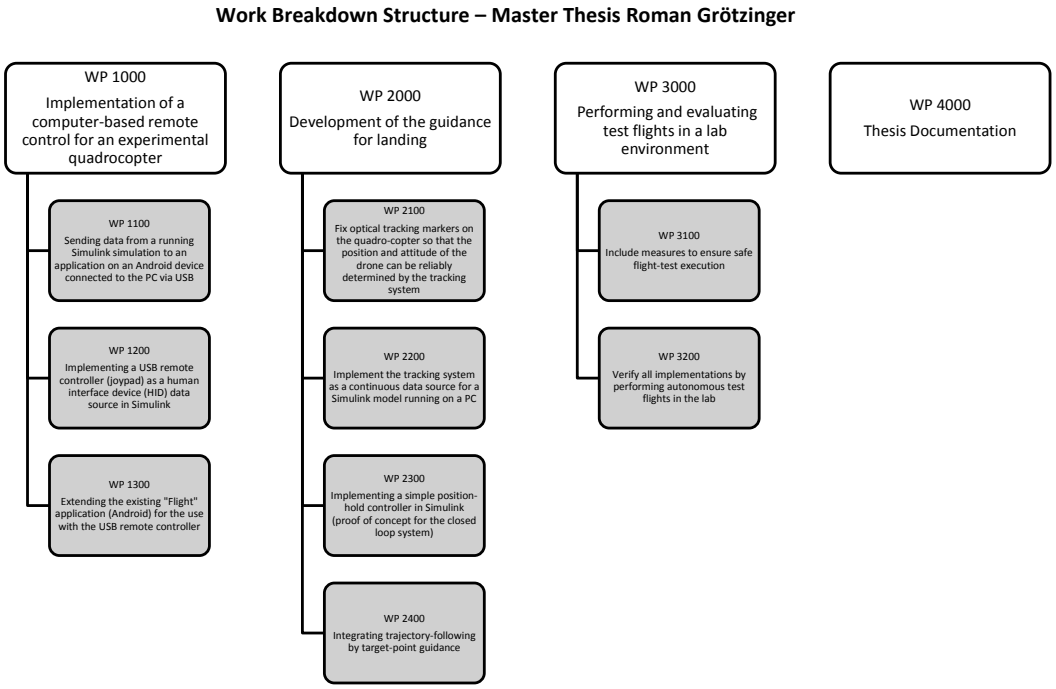
PW pulse width 3, 4, 13, 19

SDK Software Development Kit 7, 22, 23, 29, 39

TEAMS Test Environment for Applications of Multiple Spacecraft 15, 17, 20, 22, 23, 40, 46, 54, 56, 57, 78, 86, 87

Appendices

A Work Breakdown Structure



B User Manual

The necessary steps that have to be performed to make the present implementations work are described hereafter in short step-by-step instructions:

1. Fix the "Slave" device centrally on the copter and connect it to the Arduino Nano with the Micro-USB cable. Then open the Flight App and select "Arduino Slave", then "START".

2. Connect the battery cable to the copter.

3. Launch the Flight App on the "Master" device and select "Master".

4. Connect to the "Slave" by long-pressing the device name. If no peer is visible, make sure that WiFi is enabled on both devices.

5. After successful connection select the "ControllerUSBRemote" from the "Controller" drop-down menu and then confirm the settings.

6. Now the "Master" GUI is displayed. At this point connect the "Master" device and the remote controller to the central PC via USB.

7. Boot the DTrack2 system and start the tracking operations. At this point it can be verified that the copter is recognized by the tracking system and its orientation is correctly conceived. If this is not the case, re-calibration is necessary, refer to Appendix C for the instructions on the calibration process.

8. Launch MATLAB on the PC and open a Simulink model that contains at least the developed blocks for reading the USB Remote Controller input, to send the reference commands to Android and the source block for the tracking system. The guidance controller logic in between can be defined by the operator.

9. In order for the blocks to properly work, the following functions need to be on the MATLAB path:

   a) *HebiJoystick.m* together with the implementations: *USB_RC_hebi_read_convert.m*, *init_joy.m* and *close_joy.m*.

   b) *jtcp.m* together with: *adb_tcp_38300_forward.m*, *send_to_Android.m* and *tcp_close.m*.

c) The Android platform tools folder *"platform-tools"*. In the current implementation the folder needs to be in the same directory as *adb_tcp_38300_forward.m*.

10. To start operations first press the "Connect USB" button on the "Master". Then the operator has 60 seconds to start the Simulink simulation before the connection attempt times out.

11. The connection to the "Master" device is autonomously performed upon the initialisation of the Simulation via callback functions.

12. When the Simulation is running, success of the connection can be verified by the received values displayed on the "Master" device.

13. At this point the "Slave" device simultaneously receives the values and therefore starts the attitude controller.

14. Now the battery power can be forwarded to the copter by pressing "A" on the kill switch controller.

15. The ESCs of the copter should now emit a four tone jingle that indicates valid PW commands.

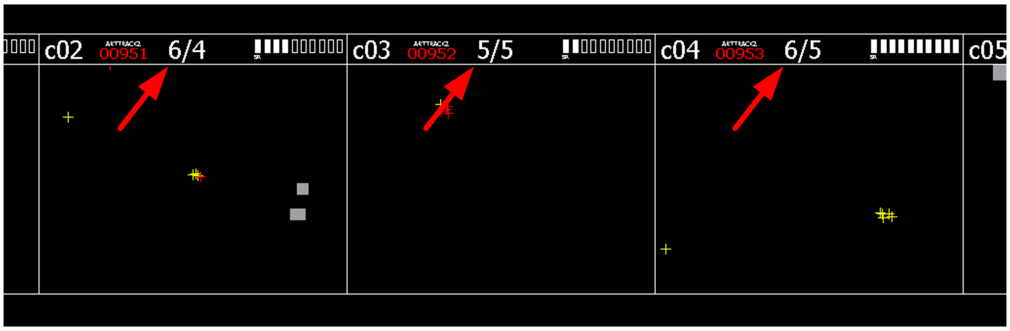
16. Now flight operations can be started by commanding a reference throttle over 0.5.

17. In the current implementation the two-position switch on ch6 is used to toggle the autonomous operations. When flipped up, towards the the operator, the pilot is in full command of the reference angles. When flipped down, away from the operator, the reference values are calculated by the guidance controller to fulfil the specified mission. The operator then has no authority over roll and pitch.

C DTrack2 Body Calibration Instructions

Hereafter the basic steps that are necessary to calibrate the copter for the DTrack2 tracking system are presented. New calibrations can regularly be required, especially if the fixations of the tracking markers had to be replaced after a hard landing.

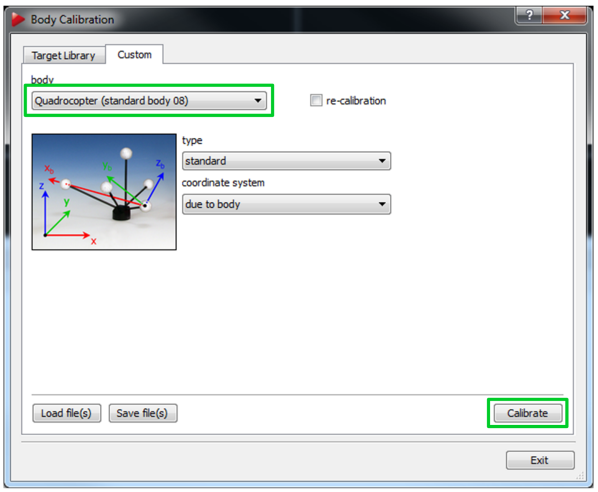
1. Make sure that the copter (with its five markers in a unique pattern) is placed on the table and that all five markers are detected of at least three different tracking cameras. This is indicated by the numbers highlighted in the front-end software screen-shot below:



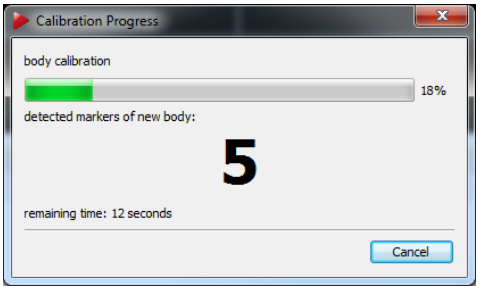
2. It can be helpful to physically cover or remove additional markers present in the visible range of the cameras so that the calibration process does not take them into account.

3. At this point the active tracking has to be stopped.

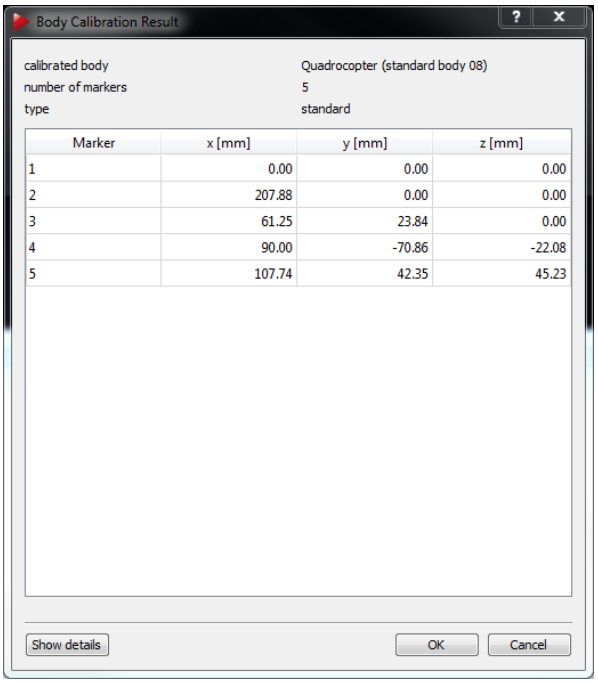
4. Go to: "Calibration" → "Body" and then select the Quadrocopter in the drop-down menu and press "Calibrate".



5. Now the calibration process is started and the markers are detected. Here, all five targets should show up.

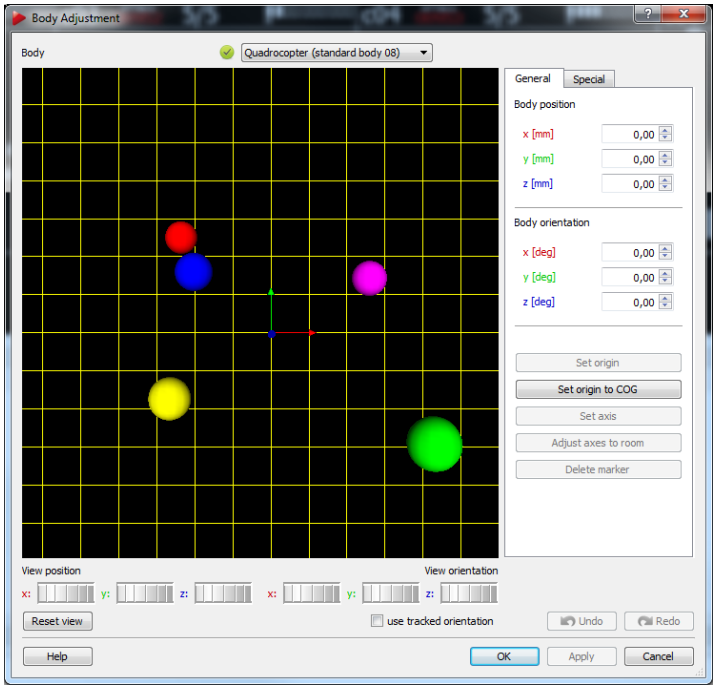


6. After the calibration process the detected markers that are associated with the calibrated body are displayed. At this point it is possible to eliminate markers if they were erroneously taken into account.



7. After the calibration process is finished, go to: "Calibration" → "Body Adjustment" and again select the Quadrocopter from the drop-down menu.

8. By default the center of the body fixed coordinate system is set in the center of one marker. Press "Set origin to COG" and dial in the physical orientation in the "Body orientation". Now the orientation of the tracking coordinate system complies with the body fixed coordinate system.



9. To position the coordinate system precisely in X/Y direction, it can be very helpful to mark the physical position of the copter on the table, turn it in 90° increments and dial the X/Y position in, so that the measured X/Y position is independent from the orientation.

10. The tracked Z-position should be set, so that the zero is placed directly on the screen-face of the "Slave" device.

D MATLAB Implementation of Schur-Algorithm

Hereafter the MATLAB implementation of the "Schur" algorithm, described in [25], is given. The function *riccati_controller_gain* solves the time-continuous algebraic Riccati equation. It returns the controller gain matrix K, so that $u = -Kx$ is the optimal controller for the input system **A**, **B** with the symmetric and positive definite weighting matrices **Q** and **R**.

```
function K = riccati_controller_gain(A, B, Q, R)
%% Check Input
  % Q and R have to be symmetrical and positive definite
  if ~(issymmetric(Q))
     error('Matrix Q has to be symmetric');
  end
  if ~(issymmetric(R))
     error('Matrix R has to be symmetric');
  end
  if ~(all(eig(Q) > eps) == 1)
      error('Matrix Q has to be positive definite');
  end
  if ~(all(eig(R) > eps) == 1)
      error('Matrix R has to be positive definite');
  end

%% Solve Riccati Equation
  %Step 1:
  %Calculate the Hamiltonian H
  H = [A, -(B*inv(R)*B'); -Q, -A'];
  % Schur Decomposition
  [U, S] = schur(H);
  %Reorder to use only stable Eigenvalues (Left-Half-Plane -> 'lhp')
  [U, ~] = ordschur(U,S,'lhp');
  [m,n] = size(U);
  U11 = U(1:(m/2), 1:(n/2));
  U21 = U((m/2+1):m, 1:(n/2));

  %Solution to Riccati Equation
  P = U21*inv(U11);

%% Controller Gains
  %Output so that u=-K*x is the optimal controller
  K=inv(R)*transpose(B)*P;
end
```

Universität Bremen

Nachname **Grötzinger** Matrikelnr. **3047717**

Vorname/n **Roman**

Diese Erklärungen sind in jedes Exemplar der Bachelor- bzw. Masterarbeit mit einzubinden.

Urheberrechtliche Erklärung

Hiermit versichere ich, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.
Alle Stellen, die ich wörtlich oder sinngemäß aus anderen Werken entnommen habe, habe ich unter Angabe der Quellen als solche kenntlich gemacht.

26.03.2018

Datum Unterschrift

Erklärung zur Veröffentlichung von Abschlussarbeiten

Die Abschlussarbeit wird zwei Jahre nach Studienabschluss dem Archiv der Universität Bremen zur dauerhaften Archivierung angeboten.
Archiviert werden:

1) Masterarbeiten mit lokalem oder regionalem Bezug sowie pro Studienfach und Studienjahr 10 % aller Abschlussarbeiten
2) Bachelorarbeiten des jeweils der ersten und letzten Bachelorabschlusses pro Studienfach und Jahr.

- (●) Ich bin damit einverstanden, dass meine Abschlussarbeit im Universitätsarchiv für wissenschaftliche Zwecke von Dritten eingesehen werden darf.
- ( ) Ich bin damit einverstanden, dass meine Abschlussarbeit nach frühestens 30 Jahren (gem. §7 Abs. 2 BremArchivG) im Universitätsarchiv für wissenschaftliche Zwecke von Dritten eingesehen werden darf.
- ( ) Ich bin nicht damit einverstanden, dass meine Abschlussarbeit im Universitätsarchiv für wissenschaftliche Zwecke von Dritten eingesehen werden darf.

26.03.2018

Datum Unterschrift