Alma Mater Studiorum · Università di Bologna

Scuola di Ingegneria e Architettura

Dipartimento di Ingegneria dell'Energia Elettrica e dell'Informazione
"Guglielmo Marconi" – DEI

Corso di Laurea in Ingegneria dell'Automazione

Implementation of an Autonomous Navigation Algorithm with Collision Avoidance for an Unmanned Aerial Vehicle

Tesi di Laurea
in
Controlli Automatici

Relatore:
Chiar.mo Prof. Lorenzo Marconi

Presentata da:
Florian Mahlknecht

Correlatore:
Prof. Nicola Mimmo

II Sessione
Anno Accademico 2017/2018

Abstract

In the last few years an increasing demand in multicopter vehicle applications is taking place. Almost all of them involve autonomous trajectory following and collision avoidance as a security measure. Although commonly used flight controller software foresees sophisticated solutions to those problems, advanced implementations are still rare. This thesis elaborates those issues in a specific use case regarding a coverage flight for agricultural purposes. The currently used technologies are exploited to implement tracking of a general trajectory defined up to the 1st order derivative as well as a rudimentary collision avoidance approach, applied to the inner multicopter control loops. The higher level control is performed by an on-board companion computer and implemented in Robot Operating System (ROS), which opens up the possibility of modular extensions. The collision avoidance in contrast, is implemented directly in the Flight Control Unit (FCU) firmware to provide fast reactive evasion maneuvers. The achieved performances are evaluated and compared with simulation results.

Keywords: Coverage Trajectory Following, Collision Avoidance, Unmanned Air Vehicles (UAV), Autopilot, Quadrotors, UAV Position Control

Abstract (italiano)

Negli ultimi anni si sta verificando una crescente domanda nelle applicazioni per gli aeromobili a pilotaggio remoto, comunemente noti come droni. Quasi tutte le applicazioni prevedono la navigazione autonoma su una traiettoria prestabilita e un sistema per evitare collisioni in volo come misura di sicurezza. Sebbene i software di controllo di volo attuali prevedano soluzioni sofisticate a tali problemi, le implementazioni avanzate sono ancora rare. Questa tesi elabora tali questioni in un caso d'uso specifico riguardante un volo di copertura per scopi agricoli. Le tecnologie attualmente utilizzate vengono sfruttate per implementare il tracciamento di una traiettoria generale definita fino alla derivata prima e un approccio rudimentale per l'esclusione delle collisioni, applicato agli anelli di controllo interni. Il controllo di alto livello viene eseguito da un computer di bordo, usando il Robot Operating System (ROS) per un'eventuale estensione modulare. Al contrario, il sistema per evitare ostacoli in volo, viene implementato direttamente nel firmware della Flight Control Unit (FCU) per fornire manovre di evasione più reattive possibile. Le prestazioni raggiunte vengono valutate e confrontate tramite i dati della simulazione.

Keywords: Tracciamento traiettoria di copertura, evitare collisioni, aeromobile a pilotaggio remoto (APR), autopilota, quadrirotore, APR controllato in posizione

Acronyms

CC Companion Computer. vii, 6–8, 10, 19, 22, 49, 51, *Glossary:* Companion Computer

CCW Counter Clockwise. 13

CNC Computer Numerical Control. 53

CPU Central Processing Unit. 67

CW Clockwise. 13

EMI Electromagnetic Interference. 57

ENU East North Up. 11, 12, 23, 29

FCU Flight Control Unit. iii, v, vii, 5–8, 10, 19, 29, 41, 49, 67, *Glossary:* Flight Control Unit

FSM Finite State Machine. 18, 19, 49

GPS Global Positioning System. 1, 10, 14, 68

I/O Input / Output. 68, *Glossary:* I/O port

IMU Inertial Measurement Unit. 1

LIDAR Light Detection and Ranging. 38, 41, 51

LP Low Pass. 42, 47, 48

MAVLINK Micro Air Vehicle Communication Protocol. 8, 10, 19, 30, 41, 42

MAVROS ROS interface for MAVLINK. 8, 10, 19, 20, 30, 41, 57

MGRS Military Grid Reference System. 14

NED North East Down. 11–13, 22, 38, 42

PC Personal Computer. 2

ROS Robot Operating System. iii, v, vii, 6–8, 10, 15, 16, 19–21, 23, 30, 42, *Glossary:* Robot Operating System

RPY Roll Pitch Yaw. 13, 45, 48

RTL Return To Land. 57

SITL Software In The Loop. 29

UART Universal Asynchronous Receiver-Transmitter. viii, 9, 10, *Glossary:* Universal Asynchronous Receiver-Transmitter

UAV Unmanned Arial Vehicle. 1, 2, 4–8, 11, 16, 48, 59, 67

UDP User Datagram Protocol. 30

UTM Universal Transverse Mercator. viii, 13, 14, 23, *Glossary:* Universal Transverse Mercator

WGS World Geodetic System. viii, 10, 13, 17, 19, 23, *Glossary:* World Geodetic System

XML Extensible Markup Language. 9

Contents

Chapter I

Introduction

Unmanned Arial Vehicles (UAVs), colloquially also known as *drones*, are drawing a lot of interest in the recent years. Advancements and price drops in sensor technology and microcontrollers, especially the packaging into an Inertial Measurement Unit (IMU) of gyroscopes and accelerometers, together with the availability of Global Positioning System (GPS), allow to build a relatively low-cost *mobile robot* with considerably good precision for outdoor flight tasks. These developments opened up many new use cases, providing attractive approaches in various fields, e.g. logistics.

Generally, there are still many improvements needed to realize robust, safe and truly autonomous drones. This written work elaborates the implementation with *currently* most common drone technologies of a *path following navigation algorithm with collision avoidance*. In particular, the focus is on automatic trajectory generation for a given path, guaranteeing constraints on internal state variables, such as velocity or acceleration. The collision avoidance is then performed in a reactive manner, *without* any need for modifying or updating the previously generated trajectory.

This thesis is part of the outcome of a group project code-named *bambi*, whose main objective is to *automate* an existing quad-copter mission for agriculture purposes. In essence, the flight task consists in scanning an agricultural field with an infrared sensor to detect the body heat of animals, in order to save them from danger of death through mowing machines.

1 Motivation

Just as the Bambi Project, almost all UAV flight applications entail the necessity to implement the following two features:

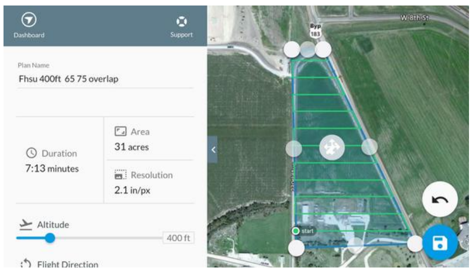
1. Automated trajectory following
2. Collision avoidance

1.1 Trajectory Following

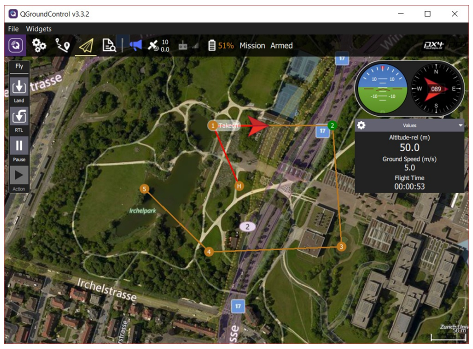
Following a trajectory is one of the most basic tasks an UAV has to accomplish. Many papers have been released covering the topics *position control* and *waypoint navigation*, e.g. [1] or [2], which together constitute the currently used solution to realize *autonomous* drone flights.

Waypoint navigation basically consists in reaching a list of defined global waypoints, i.e. GPS coordinates, by using a straight line as interconnection. It is used in countless applications such as surveillance or aerial imaging.

The so-called *flight missions* are generated by end-user software running on mobile devices or personal computers (PCs), fig. 1.1 shows two examples. Already the quantity of mission planner applications available in the common online stores shows the elevated use of trajectory following tasks.


Plan Name
Fhsu 400ft 65 75 overlap
Area 31 acres
Duration 7:13 minutes
Resolution 2.1 in/px
Altitude 400 ft

(a) Drone Deploy Aerial Mapping Software


Altitude-rel (m) 50.0
Ground Speed (m/s) 5.0
Flight Time 00:00:53

(b) QGroundControl[1]

Figure 1.1: Waypoint Generation Software

However, from a control point of view, those solutions give very limited freedom in the trajectory design, i.e. the waypoints are reached using *straight lines* (as it can be seen in fig. 1.1) and usually a constant velocity control signal of around 5 m/s.

1.2 Collision Avoidance

Collision avoidance[2] is another important research focus in the field of UAVs, since it represents the main security aspect. Usually, such systems try to keep a minimum distance in flight from any kind of obstacles. Especially in urban environments, the presence of trees, houses, light poles, etc. constitutes a huge problem in the safe execution of flight tasks. Not only are eventual crashes a huge risk of damage for the (often expensive) on-board electronic equipment, but they constitute also serious safety concerns for people nearby.

1.3 The Bambi Project

The Bambi Project aims to robotize a coverage flight task on agriculture fields for saving wildlife, currently carried out by hand.

The mission is so far guided by a pilot and at least one assistant, which saves eventually found animals in the field. Through an analog video transmitter, the signal of the thermal camera is brought to the ground and continuously checked for anomalies by the pilot. If zones at higher temperature are found, the drone *hovers*[3] over the potential animal, until it is found and saved by the ground crew.

Various organizations and projects have been realized to promote the usage of UAVs to save mainly fawns, e.g.:[4]

- In Switzerland: `www.rehkitzrettung.ch`

[1]Taken from `https://docs.px4.io/en/flying/missions.html`, accessed on 2018-09-18
[2]Also known as *obstacle* avoidance
[3]i.e. stays in the same position in the air
[4]Links have been accessed on 2018-09-19

- In Germany: `www.wildretter.de`[5], `rehkitzrettung-reichelsheim.de` and `rehkitzrettung-gera.de`

Some of them provide even online reservation and registration for new fields.

1.3.1 Importance in Agriculture

In grassland areas, especially those surrounded by forests, it is a common problem, that wild animals hide in cultivated fields. The work with mowing machines then involuntary leads to thousands of fatalities. The international council for game and wildlife conservation, published a mowing guide for agriculture fields, in which an estimate of the yearly losses can be found:

"In Germany alone, the volume of wildlife losses resulting from grassland management amounts at a conservative estimate to 500,000 individuals, of which approximately 90,000 are fawns." [3, p. 4]

It is therefore a significant issue, which draws the attention mainly from *two* parties:

the farmers are interested in avoiding brutal interruptions and keeping their harvest clean

the rangers are interested in preserving the population of wild animals

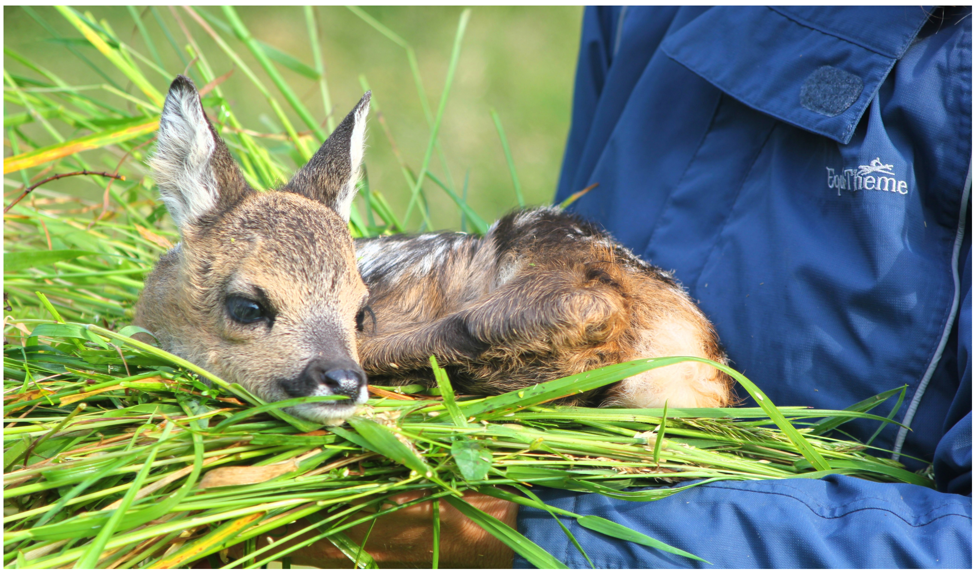


Figure 1.2: Saved fawn[6]

Figure 1.2 shows a saved fawn. The crucial aspect is that they hide in a very efficient way. In higher grass it is impossible to spot them even from just a few meters of distance.

The most common method to overcome the problem is the use of infrared sensors:

> Juveniles in the meadows can be tracked down by means of so-called "wildlife detectors" and moved to safety. There are several prototypes on the market, most of which function on the principle of infrared sensors which detect the body heat of the animals. [...] However, the limitations of this technique are quickly reached: in order to track down the animals, a certain temperature difference between the animal body temperature and the environment temperature is required [...] [3, p. 10]

[5]EU research project

[6]Gently provided by "Verein Rehkitzrettung Schweiz/Rehkitzrettung.ch"

The former issue regarding the need of a significant temperature difference is the reason why coverage flights are usually carried out in the morning, from 5AM to 7AM.

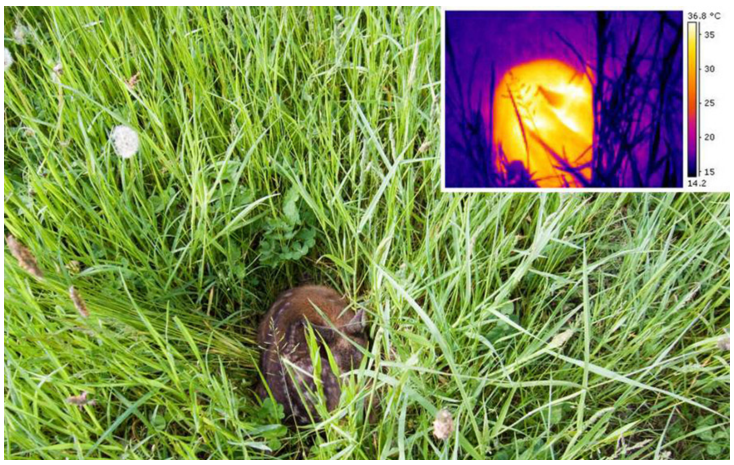


Figure 1.3: Thermal capture of fawn

Figure 1.3 shows the record of a thermal camera. Modern equipment allows the detection from a flight altitude of more than 50m[7].

1.4 General UAV usage for Wildlife Tracking

The usage of UAVs has been considered more generally for tracking down animals. For several years already research works are published, using drones for various surveillance tasks in the field of biology, see [4].

In the *Journal of Ecology and Environment* in 2017 e.g., Han proposed the usage of UAVs to monitor waterbirds, see [5]. In [6] monitoring cattle has been evaluated. [7] documents the usage of a custom-build drone for whale surveillance.

1.5 Summary

The use cases shown, contain all together the need for *trajectory following* and – as a safety measure – *collision avoidance*. Contenting the demand for improvements in this research fields is therefore crucial to further enhance the advantages of UAVs.

2 State of the art

The currently deployed solutions shall briefly be inspected and analyzed.

2.1 Trajectory Following

As discussed for fig. 1.1 on page 2, it is of common practice to define *waypoints* on special software for later mission execution on UAVs. The workflow is thereby:

[7]See `https://www.rehkitzrettung.ch/infos/infos-ueber-bfh-hafl-methode`, accessed on 2018-09-20

1. Use mission planning software to get a list of waypoints
2. Upload the *mission* to the Flight Control Unit (FCU)
3. Start the execution on the vehicle using a waypoint navigation autopilot

Typically, a few tens of waypoints are defined, which makes the trajectory quite sharp-cornered and harsh.

2.1.1 Waypoint Navigation

Once the *mission* is uploaded to the FCU, special flight modes[8] trigger the waypoint navigation. The vehicle will follow a *straight line* between waypoints at a *constant velocity*. PX4, one of the most used flight controller softwares e.g., implements this behavior in the FlightTaskAutoLine C++ class: `https://github.com/PX4/Firmware/blob/370fddc1158fde39b14467ec7ed098376c76632e/src/lib/FlightTasks/tasks/AutoLine/FlightTaskAutoLine.hpp#L37-L38`.

2.2 Collision Avoidance

There is a huge research effort to improve collision avoidance approaches. The problem is discussed not only in the research area of UAVs, but also in the promising sector of *self-driving* cars.

The major challenge is to *detect* the presence of obstacles. The most common approach is to use a *2D* laserscan for measuring the distance from any eventual obstacle located in a plane of reference around the vehicle, as discussed e.g. in [8]. Modern approaches include computer vision, often in combination with convolutional neural networks trying to enhance the quality of the information on obstacles, see e.g. [9].

The reaction to obstacles, then, depends on the way *path planning* is handled. Usually, approaches are distinguished between *online* and *offline* path generation. In online path generation, the planned trajectory is generated in-flight, i.e. updated as obstacles are detected. The model typically implies to have a *navigation goal* for the mobile robot. In the offline path generation method by contrast, the trajectory is computed *once*, usually even before the flight.

The most common and, at the same time, one of first ideas is the *potential field approach*. Basically, every obstacle is modeled as a pole in a *repulsive* potential field, while the *navigation goal* is represented in a *attractive* potential field. The gradients of those potentials are then used as a reference acceleration for the UAV.

Collision avoidance and path planning are often treated together. Despite of the research effort, a simple and effective solution, to be used as a *security measure* is therefore still missing.

2.3 Related Publications

Besides the already mentioned papers, many other related publications are available. Regarding trajectory following, an approach for smooth trajectory generation using Bezier curves, meeting dynamic constraints, is discussed in [10]. Minimum *snap* trajectories for UAVs are treated in [11].

A good introduction into obstacle avoidance can be found in [12]. A real time implementation is discussed e.g. in [13].

[8]In the flight controller software PX4 e.g., the mode AUTO

In addition to the publications regarding UAV applications for biological surveys, a similar use case has been analyzed in [14]. It treats the tracking of humans through the usage of thermal cameras mounted on UAVs.

3 Innovation

In this thesis, an *implementation* using broadly adopted platforms such as PX4 and Robot Operating System (ROS) is presented. A Companion Computer (CC) is used to provide setpoint updates to the FCU. This brings the advantage of using the flexible ROS environment as the main programming platform.

Regarding trajectory generation, the implemented setpoint updates from the CC provide more flexibility in the trajectory design. It enables to use control techniques such as *velocity feed-forwarding* on those mobile robot platforms. A simple constant velocity trajectory generation is provided. Unlike waypoint navigation systems, the presented implementation allows to freely use any trajectory defined up to the 1^{st} order derivative, which yields a better tracking performance.

For the collision avoidance system a simple approach with a 2D laserscan is used. The proposed application of the obstacle avoidance signal directly in the acceleration control loop yields a high agility in the reactive maneuvers with shorter delays. With respect to similar systems, it has the advantage of getting along without *online path updates*. It works in *any* flight mode as a *security measure*. Common disadvantages of 2D obstacle detection systems are ,however, *not* addressed.

4 Thesis Outline

In chapter II, the problem of controlling the UAV to follow a predefined path is discussed and a simple solution, i.e. a *constant velocity trajectory* is provided and its performance evaluated.

Chapter III on page 37 discusses the implementation of an appropriate *collision avoidance* approach and presents the simulation results.

Chapter IV on page 51 is dedicated to the presentation of the project work and provides the outcomes of some field experiments.

Finally, chapter V on page 59 comments the accomplished work and gives an outlook on possible future improvements and developments.

Chapter II

UAV Position Control

In this chapter the problem of controlling the UAV's position is addressed in detail. Currently available technologies are exploited and their performance is evaluated, using a simple solution to the trajectory generation problem.

1 System Architecture

Figure 2.1 shows the main system components. The most important ones are two:

Flight Control Unit (FCU) To handle the low-level control, the widely spread flight controller Pixhawk is used, flashed with the most recent PX4 flight-stack.

On-board Companion Computer (CC) The high-level control is performed by the *Raspberry Pi 3* as an on-board computer, using ROS[1] as an implementation framework.

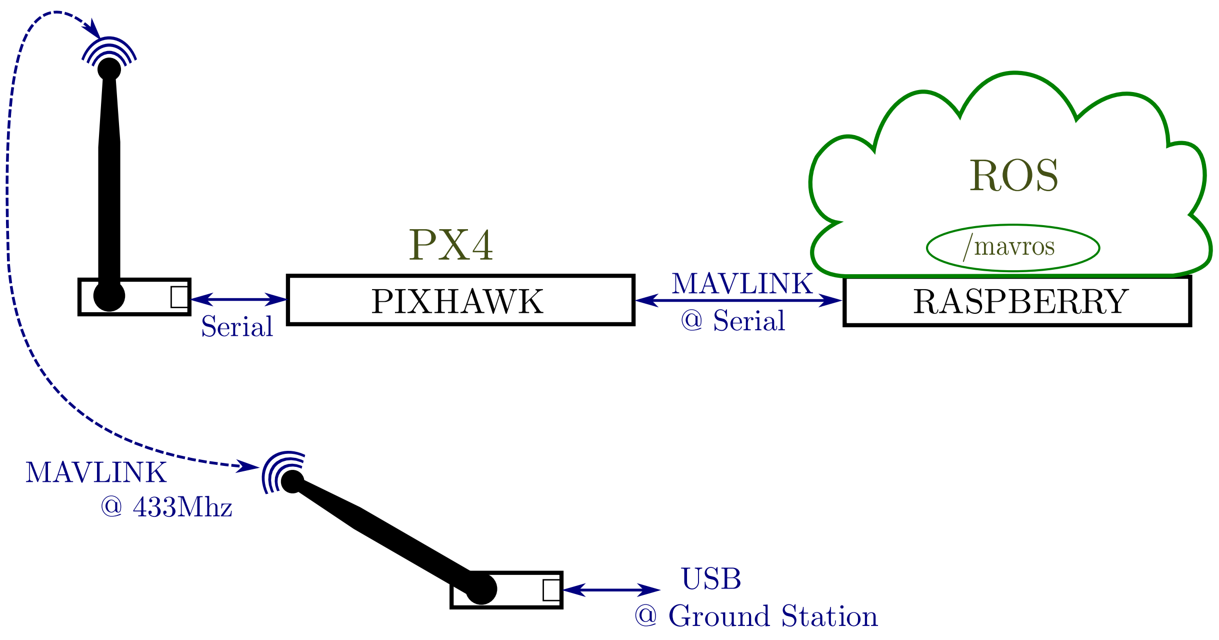


Figure 2.1: Basic System Architecture

[1]See [15] or, for a brief introduction, [16]

As fig. 2.1 on page 7 illustrates, all the major implementation work can be conveniently carried out, using ROS on the Raspberry Pi. The crucial ROS Node, which allows the communication with the PX4 flight-stack, is a ROS interface for MAVLINK (MAVROS), and available as part of the software collection around PX4.

1.1 PX4

PX4 is an open-source autopilot system designed for low-cost UAVs. Started as a student project at the Computer Vision and Geometry Lab at ETH Zürich[2] in 2009, it presents nowadays the de-facto standard in the drone industry. Besides the software, the project provides open hardware designs which several vendors are currently producing. Documentation and general information on the project can be found under `px4.io`[3].

The following main advantages made up the design choice of using PX4:

Simulation As a open source project, it supports many simulation choices in order to be able to *test* new code prior to an application on the real vehicle. Given the complexity of the system, without a simulation option it would be practically impossible to develop new features.

OFFBOARD mode The autopilot software provides the so-called OFFBOARD mode, which enables *full control* of the vehicle by the CC.

Velocity feed-forward In the OFFBOARD mode, a feed-forward control path is enabled, which potentially allows a better control with respect to other available autopilots.

1.1.1 Flight Modes Overview

The currently implemented flight modes for multicopters in PX4 are[4]:

- Position
- Altitude
- Stabilized
- Rattitude
- Acro
- Takeoff
- Land
- Hold
- Return
- Mission
- Follow Me
- Offboard

Relevant for *autonomous* copter missions are the former 7, especially the OFFBOARD mode. In the former the Micro Air Vehicle Communication Protocol (MAVLINK) is used to receive setpoint updates from a CC, which enable *full control* of the vehicle's position, up to the 1st order derivative.

1.1.2 MAVLINK

MAVLINK is a parallel outcome of the PX4 project used for communication between the FCU and the ground station or the CC. It is designed as a *marshalling* library, i.e. serializing the defined messages for transmission.

Every message is in the following byte format, listed in table 2.1 on the facing page:

[2]ETH Zürich (Swiss Federal Institute of Technology)

[3]Accessed on 2018-09-20

[4]Compare with `https://docs.px4.io/en/flight_modes/`, accessed on 2018-09-20

Field name	**Index (Bytes)**	**Purpose**
Start-of-frame	0	Denotes the start of frame transmission (0xFE)
Payload-length	1	Length of payload (n)
Packet sequence	2	Sending sequence (allows detection of packet losses)
System ID	3	Identification of the SENDING system
Component ID	4	Identification of the SENDING component
Message ID	5	Identification of the message (defines what the payload "means")
Payload	6 to (n+6)	The message data
CRC	(n+7) to (n+8)	Check-sum of the entire packet

Table 2.1: General MAVLINK Message Format

The different message types are then defined in Extensible Markup Language (XML) files, available at `https://mavlink.io/en/messages/common.html`[5].

1.1.3 Interfacing Options

The used platform is the first version of the Pixhawk series, shown in fig. 2.2. Details can be found under `https://docs.px4.io/en/flight_controller/pixhawk.html`[5].

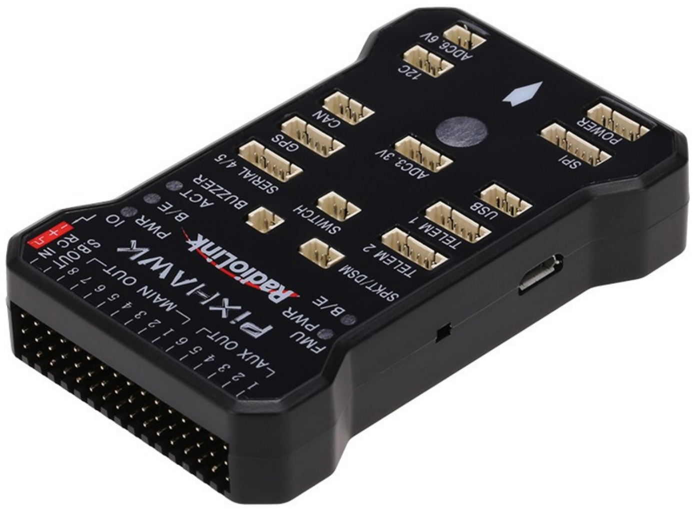


Figure 2.2: Pixhawk

The most relevant interfaces are:

I^{2}C through an I^{2}C splitter, the ground lidar and the external compass are connected

GPS through a dedicated Universal Asynchronous Receiver-Transmitter (UART) port

[5]Accessed on 2018-09-21

Serial connection to the Raspberry Pi (used as CC) through an UART port

The serial connection to the Raspberry Pi was set-up for the OFFBOARD mode, configuring a few routing settings on the Pixhawk and the installation of MAVROS on the Raspberry Pi.

1.2 MAVROS

As already mentioned, a translation layer between MAVLINK and ROS is needed[6], in order to allow ROS programming on the Raspberry Pi. MAVROS has been designed to satisfy those exact needs.

The ROS node `/mavros` contains all publishers / subscribers for interacting through MAVLINK with the FCU. The most important topics for controlling the vehicle in OFFBOARD mode are:

- `/mavros/setpoint_raw/local` which transmits the setpoint to the vehicle's controller, referred to the local system of reference.
- `/mavros/set_mode` ROS service to change the flight mode.

2 Problem Description

The robotized flight mission implemented in the Bambi Project[7] includes various stages, inter alia, the *execution of the coverage flight*. It basically consists in following a *geometric path*, i.e. a list of World Geodetic System (WGS) 84 coordinates.[8]

The central problem is therefore to continuously provide the PX4 flight controller with *setpoint* updates that guarantee the trailing of the given path.

2.1 Mathematical Definition of the Control Problem

Assuming for now to have an inertial Euclidean reference frame, the path may be defined as a finite sequence:

$$\begin{aligned} \langle \boldsymbol{p}_k \rangle_{k \leq n} &= \langle \boldsymbol{p}_1, \boldsymbol{p}_2, \dots, \boldsymbol{p}_n \rangle \\ with\ \boldsymbol{p}_k &\in \mathbb{R}^3 \end{aligned} \tag{2.1}$$

The problem is to obtain the control setpoint $\boldsymbol{r}^*(t)$, i.e. the *time dependent trajectory*, to be used as an input for the PX4 FCU.

$$\begin{aligned} \boldsymbol{r}^*(t) : \ [0, t_n] &\to \mathbb{R}^3 \\ t &\mapsto \left(x^*(t), y^*(t), z^*(t)\right) \end{aligned} \tag{2.2}$$

Where $t_n \in \mathbb{R}^+$ is the time in seconds needed for getting to the last point $\boldsymbol{p}_n = \boldsymbol{r}^*(t_n)$. The function $\boldsymbol{r}^*(t)$, which needs to be found, must satisfy the following condition:

[6]See fig. 2.1 on page 7
[7]see section 1.3 on page 2
[8]For GPS navigation see e.g. [17], the reference systems are discussed in section 3 on the facing page

$$\begin{aligned}\exists \langle t_j \rangle_{j \le n} \in \mathbb{R}^+ \;\mid\; \forall\, i \le j \le n \qquad & t_i \le t_j \\ & \wedge\, \boldsymbol{r}^*(t_j) = \boldsymbol{p}_j\end{aligned} \tag{2.3}$$

In other words, it must be possible to identify time instances in which the trajectory passes through the given points. Mathematically, this is accomplished with the existence of the finite real increasing monotonic sequence $\langle t_j \rangle_{j \le n}$. Note that $t_1 = 0$ in order to constrain the trajectory to start from $\boldsymbol{p}_1$.

The time dependent trajectory $\boldsymbol{r}^*(t)$ which needs to be generated has to obey some dynamical constraints, like velocity limits:

$$\|\dot{\boldsymbol{r}}^*(t)\| \le v_{max} \;\; \forall\, t \in [0, t_n] \tag{2.4}$$

$$\|\ddot{\boldsymbol{r}}^*(t)\| \le a_{max} \;\; \forall\, t \in [0, t_n] \tag{2.5}$$

It is worth to emphasize that the actual implementation does not ask for a time-continuous function, but a discrete sequence of setpoints, equidistant in time of a certain *period* $\frac{1}{f_{sp}}$ where $f_{sp} \in \mathbb{R}^+$ is the *setpoint rate* in $[\text{Hz}]$. In this way the control signal can equally be defined as:

$$\begin{aligned}\langle \boldsymbol{r}^*{}_k \rangle_{k<=m} &= \langle \boldsymbol{r}^*{}_1, \boldsymbol{r}^*{}_2, \dots, \boldsymbol{r}^*{}_m \rangle \\ \text{with } \boldsymbol{r}^*{}_k &= \boldsymbol{r}^*\Big(\frac{k}{f_{sp}}\Big)\end{aligned} \tag{2.6}$$

Which implies that $t_f = \frac{m}{f_{sp}} \rightarrow m = f_{sp}\ t_f$, i.e. the higher the rate, the more setpoints are needed (considering the same trajectory).

3 Coordinate Frames

For a meaningful discussion, common reference systems are needed. There are many different conventions for coordinate frames, the most important ones shall be discussed briefly.

3.1 Local Reference Frames

For controlling the UAV two main representations are used:

- East North Up (ENU)
- North East Down (NED)

These are right-handed *local* reference frames, i.e. the origin is set in a determined position.

3.1.1 NED Frame

As shown in fig. 2.3, the NED convention is applied to an *external* reference frame as well as to the *body* reference frame. The external reference frame lets the x^E-axis point to global *north*, the y^E-axis point to global *east* and the z axis pointing *down*, to complete the right-handed reference system.

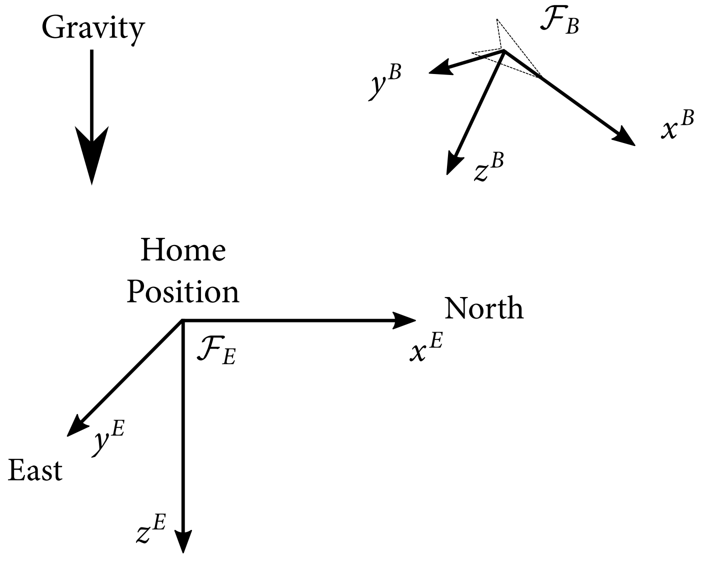


Figure 2.3: NED Frame

The *body* reference frame is made-up in the same manner, with the nose pointing in x^B direction.

3.1.2 ENU Frame

In an analogous way, the ENU convention, given in fig. 2.4, lets the x^E-axis point to global *east*, the y^E-axis point to global *north* and the z axis pointing *up*.

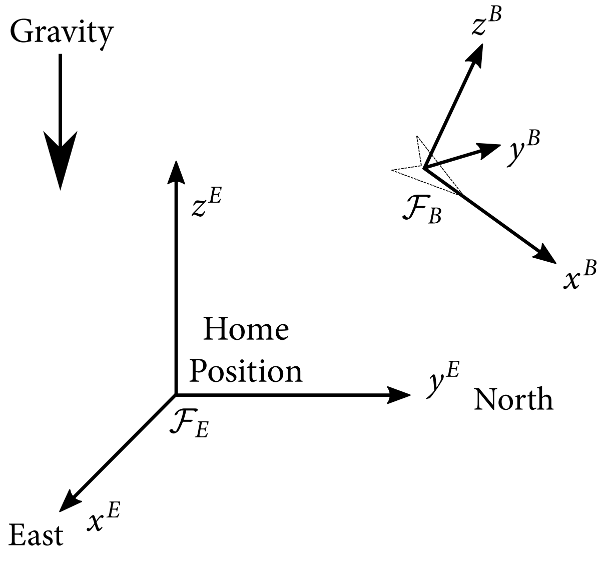


Figure 2.4: ENU Frame

The *body* reference frame lets the nose point again in x^B direction, but with z^B pointing downwards instead.

3.1.3 RPY Angles

It is worth to point out that the body systems presented in sections 3.1.1 and 3.1.2 are not coherent with the general adapted Roll Pitch Yaw (RPY) *orientation angles* used in robotics. The former can be defined as:

Roll ϕ counter-clockwise (ccw)[9] around the z axis, which is the *approach* direction

Pitch θ ccw around the y axis, which is the *sliding* direction

Yaw ψ ccw around the x axis which is the *normal* direction

Since in the NED frame e.g., the nose points in the x^B direction, instead of pointing in the *approach direction z*, the RPY angles the NED reference system will not yield the expected results.

Furthermore, the YAW angle in PX4 will generally be measured considering it with respect to global north in the NED frame, which, seen from above means in the *clockwise (cw)* direction, since the z^B axis points downwards.

3.2 Global Reference Frames

Globally there are two important standards:

- World Geodetic System (WGS)
- Universal Transverse Mercator (UTM)

Meanwhile WGS maps global locations using an *reference ellipsoid* in a 3-dimensional polar reference frame, UTM provides a set of local, metric, 2-dimensional Cartesian reference frames.

3.2.1 WGS 84

As fig. 2.5 on the next page shows, WGS uses the commonly known coordinate mapping involving longitude λ and latitude ϕ.

[9] ccw is the counter-clockwise direction considering the top view from the axis onto the other two 90° distant axis

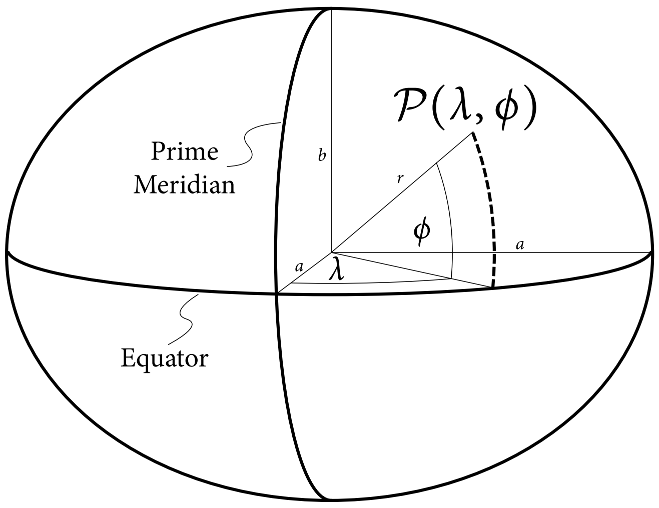


Figure 2.5: World Geodetic System

The geoid[10] is approximated by a *reference ellipsoid*. The last version from 1984, which is nowadays used also in GPS, has inter alia the following parameters (used in fig. 2.5):

- $a \approx 6378.137\,\text{km}$
- $b \approx 6356.314\,\text{km}$

It is worth to emphasize that the distinction from a proper sphere is rather small, given the small difference in distance between the two main axis of around 20 km, which is less than 1 %. As fig. 2.5 illustrates, all the coordinate pairs of (λ, ϕ) need a common reference which is given by the prime meridian and the equator.

3.2.2 UTM

The UTM system provides a set of 2-dimensional Cartesian coordinate frames. Referring to any possible reference ellipsoid, the Earth is segmented into 60 zones, each being a 6° band of longitude. In each zone a secant transverse Mercator projection is applied. For details and formulas see e.g. [18].

Although, strictly speaking latitude bands are not part of UTM, but rather defined in the Military Grid Reference System (MGRS), they are commonly used. Each UTM (longitude) zone is divided into 20 latitude bands. Each latitude band is 8° high, and is identified by letters starting from ‘‘C’’ at 80°S, following the alphabet until ‘‘X’’, without the letters ‘‘I’’ and ‘‘O’’, because of their similarity to 1 and 0 respectively.

The UTM tiles over Europe are illustrated in fig. 2.6 on the facing page.

[10] the shape of earth’s water surface under the influence of gravity and rotation alone, i.e. no wind effects etc.

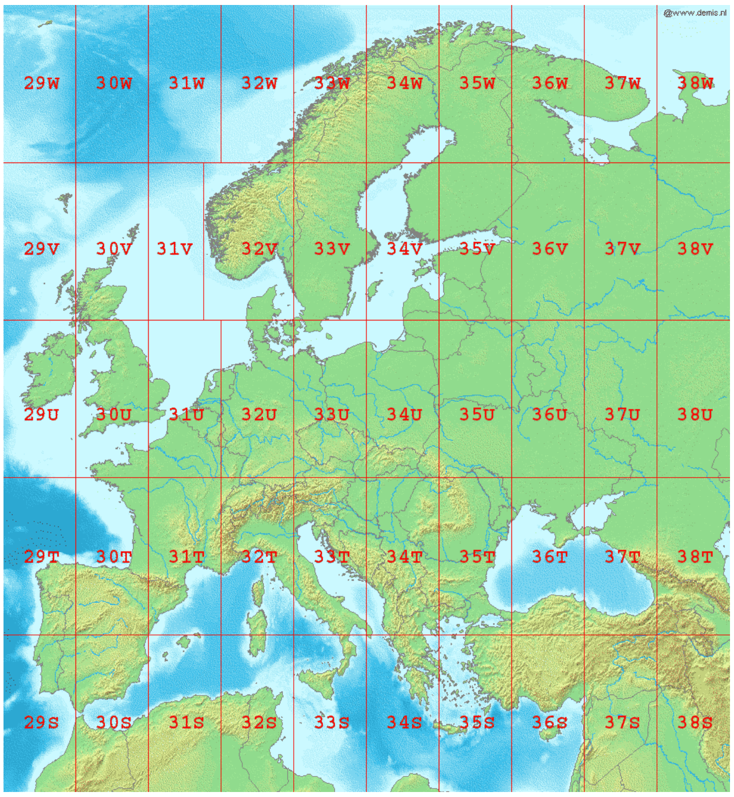


Figure 2.6: UTM Grid

4 ROS Node Architecture

The Bambi Project uses various ROS nodes that implement different tasks, which need to be accomplished during the flight mission. For a better illustration however, only the relevant nodes involved in *trajectory following* and the most important related topics are shown in fig. 2.7.

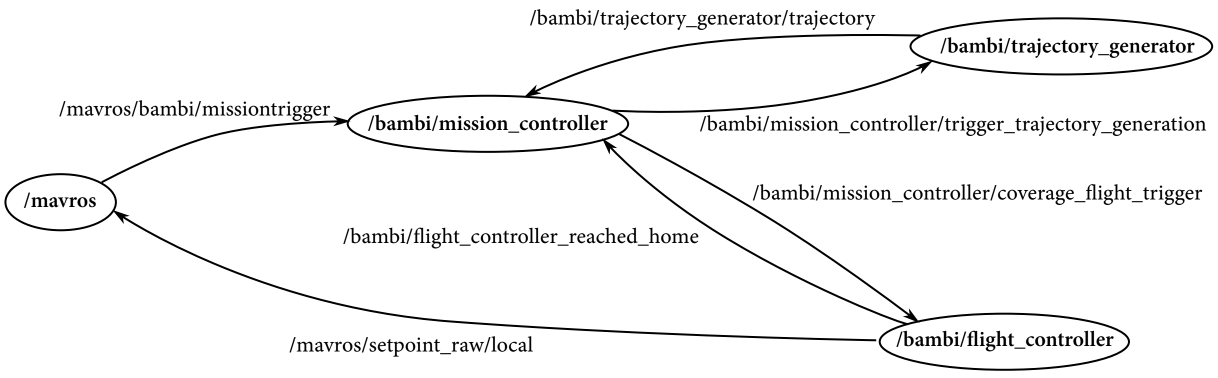

/bambi/trajectory_generator/trajectory
/bambi/trajectory_generator
/mavros/bambi/missiontrigger
/bambi/mission_controller
/bambi/mission_controller/trigger_trajectory_generation
/mavros
/bambi/mission_controller/coverage_flight_trigger
/bambi/flight_controller_reached_home
/mavros/setpoint_raw/local
/bambi/flight_controller

Figure 2.7: ROS-Graph of trajectory-following related ROS nodes

The *mission controller node* handles the whole mission procedure. In particular, for the coverage flight it relies on other two nodes:

Trajectory generator Generates a list of setpoints, for a certain update rate which follow the trajectory

Flight controller Handles the communication with the flighstack and actually publishes the generated setpoints

4.1 Problem Division

In this way the problem has been split up in two smaller subproblems:

1. Generating the *list of control setpoints* from a geometric path
2. Controlling the UAV by using the list of generated setpoints

This problem division is reflected in the chosen ROS nodes. The real control problem lies naturally in *generating* the list of setpoints, meanwhile the controlling node is a more technical implementation issue. The following sections are therefore dedicated to the discussion of those technical details, including the flight controller node implementation.

4.2 Relevant messages definitions

The communication between the ROS nodes is defined by *messages* used in different ROS topics. There are already standard ROS messages available, but in order to have meaningful messages for the special purpose of using relative altitudes over ground, the Bambi Project introduced some custom ROS messages. The relevant ones are given in listing 2.1 on the facing page.

```
####################### bambi_msgs/GeoPosition2D #######################

float64 latitude
float64 longitude


############## bambi_msgs/GeoPositionWithRelativeAltitude ##############

bambi_msgs/GeoPosition2D geopos_2d
float32 altitude_over_ground


########################### bambi_msgs/Path ############################

bambi_msgs/GeoPositionWithRelativeAltitude[] geometric_path


################# bambi_msgs/DynamicFlightConstraints ##################

float32 max_velocity
float32 max_acceleration


################### bambi_msgs/PathWithConstraints ####################

bambi_msgs/Path path
bambi_msgs/DynamicFlightConstraints flight_constraints


######################### bambi_msgs/Trajectory ########################

float32 sample_rate
mavros_msgs/PositionTarget[] setpoints
```

Listing 2.1: Relevant Bambi ROS message definitions

As it is reported, the `bami_msgs/GeoPosition2D` carries the WGS 84 coordinates, i.e. latitude and longitude, *without* altitude information. `bami_msgs/GeoPosition2DWithRelativeAltitude` adds then the relative altitude, represented as a floating point value. The *geometric path* is simply an array of those relative altitude points.

As an *input* to the trajectory generation node, a `bami_msgs/PathWithConstraint` message is used. It contains the *geometric path* (i.e. `bami_msgs/Path`) and the dynamical constraints object. The output of the trajectory generation node, is nothing but a list of *setpoints* paired with the *setpoint rate* f_{sb}, i.e. the `bami_msgs/Trajectory` message.

The setpoints are defined as `mavros/PositionTarget` messages, shown in listing 2.2 on the next page.

```
###################### mavros_msgs/PositionTarget ######################
std_msgs/Header header

uint8 coordinate_frame
# ... [coordinate frame constants]

uint16 type_mask
# ... [type mask constants]

geometry_msgs/Point position
geometry_msgs/Vector3 velocity
geometry_msgs/Vector3 acceleration_or_force
float32 yaw
float32 yaw_rate
```

Listing 2.2: Position Target (Setpoint) Message Definition

The trajectory generator node, from a certain point of view, *converts* incoming `bami_msgs/PathWithConstraint` messages into `bami_msgs/Trajectory` messages. This implies introducing the dependency on time, since the outcome is nothing but $\langle \boldsymbol{r}^*{}_k \rangle_{k<=m}$ defined in eq. (2.6) on page 11.

4.3 Mission Controller

The mission controller node is build as a finite state machine (FSM). Figure 2.8 shows a simplified version of the complete state machine diagram.

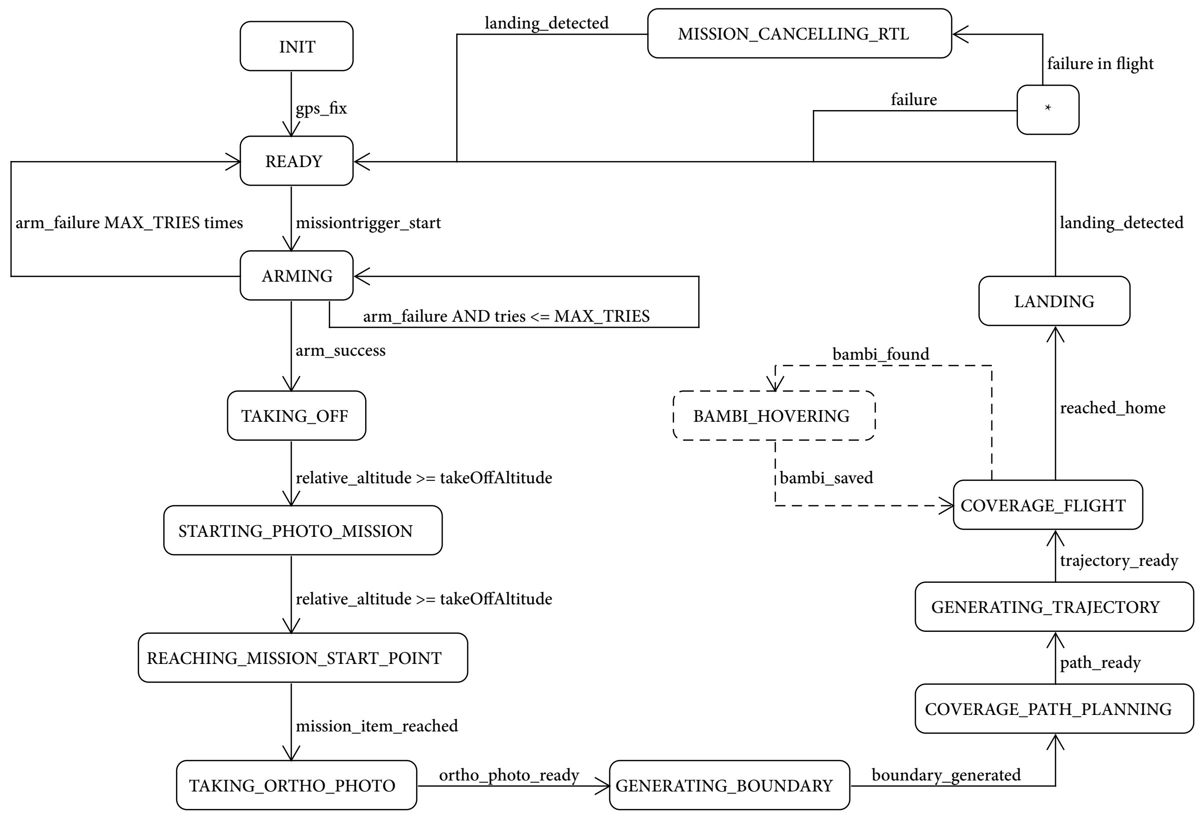


Figure 2.8: Mission Controller Finite State Machine

The Bambi Project flight mission gets triggered by a dedicated MAVLINK message, fired from the groundstation and routed through the Pixhawk to the CC. It contains all necessary information, such as flight altitudes, thermal sensor footprint and mission start point. In the first few states of the FSM diagram in fig. 2.8 on page 18, the multicopter gets armed and takes off, by using mainly the MAVROS interface. Once taken off, the altitude has to be reached before starting a waypoint navigation mission to the given WGS 84 mission coordinates. As the mission start point is reached, the agriculture field boundary is generated and a *geometric coverage path* is produced by using other ROS nodes. At this point the nodes relevant for trajectory generation are reached out.

The `GENERATING_TRAJECTORY` state indeed publishes the geometric path with the relevant dynamic constraints, which triggers the trajectory generator node. The topic `/bambi/mission_controller/trigger_trajectory_generation` is used for this purpose. Once the list of setpoints has been generated, the coverage flight is started, by using the flight controller node.

It is foreseen, but not implemented yet, to pause the coverage flight when a thermal anomaly has been detected. Through the modular structure of ROS this can easily be added through an additional ROS node, which would trigger the `bambi_found` signal on the mission controller. The hovering then gets handled by the *fligh controller* node, which pauses the setpoint publishing, stopping at the current setpoint.

Once the *flight controller* node finishes the setpoint list, the `reached_home` signal is fired which triggers then an automatic landing handled by the PX4 FCU.

4.4 Flight Controller

The flight controller node makes use of the set point list and is implemented as another, although smaller, FSM, shown in fig. 2.9.

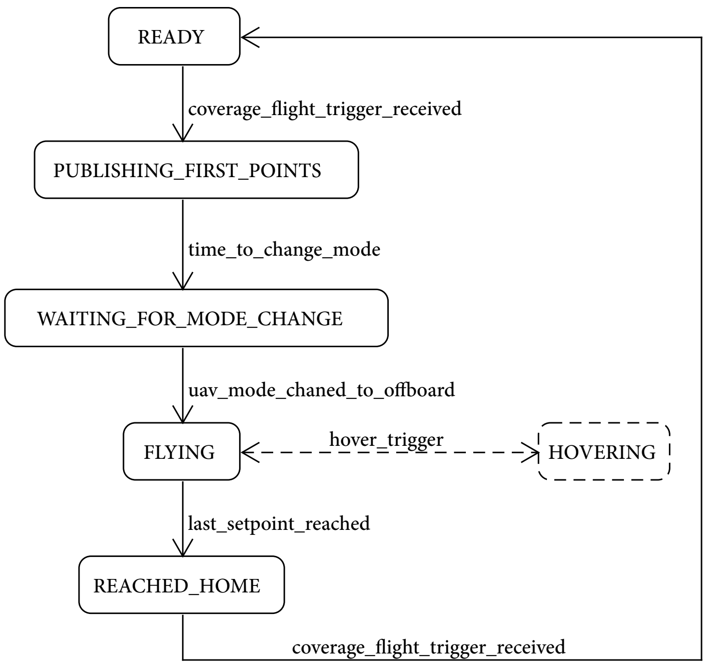


Figure 2.9: Flight Controller Finite State Machine

Starting State	**Ending State**	**Condition**	**Action**
`READY`	`FIRST_SETPOINTS`	start received	save trajectory and start publishing the *first* setpoint
`FIRST_SETPOINTS`	`MODE_CHANGE`	2s passed	change to OFFBOARD mode
`MODE_CHANGE`	`FLYING`	mode changed	continuously publish setpoints
`FLYING`	`HOVERING`	hover trigger	publish current setpoint
`HOVERING`	`FLYING`	hover trigger	continue with setpoints
`FLYING`	`REACHED_HOME`	last setpoint reached	inform *mission controller*
`REACHED_HOME`	`READY`	stop received	free up memory

Table 2.2: Flight Controller State Transition Table[11]

The state transition table (table 2.2) defines the behavior of the flight controller. When the coverage flight trigger is received, the trajectory is saved internal to the flight controller node and publishing the *first* setpoint is started. In fact, the documentation and the PX4 implementation requires setpoints to be published already before the mode is switched to OFFBOARD:

"Before entering Offboard mode, you must have already started streaming setpoints. Otherwise the mode switch will be rejected."[12]

In the flight controller implementation 2 s pass before trying to switch mode. Before publishing the actual trajectory the confirmation of the mode update is awaited, i.e. the MAVROS mode topic is expected to change.

4.4.1 Implementation

Most of the ROS nodes in the Bambi Project are implemented using C++[13]. A short code snippet from the *flight controller* node class shown in listing 2.3 on the next page.

The class has been designed in such a way that a clean implementation of the state machine is guaranteed. This is achieved by handling *all*[14] the state changes in *one* method, namely `handleStateMachineCommand(...)`. The parameter `command` assumes the value of any possible state change trigger, from fig. 2.9 on page 19, which is represented by the nested enum class `Command`. The second argument, a generic void pointer, eventually identifies a data carrying object, depending on the command, such as the trajectory in case of a coverage flight trigger.

The `handleStateMachineCommand` method is called internally from callback methods, denoted by the suffix `cb_`, which implement mainly ROS subscriber callback functions. In the implementation of the command-handling method, a switch-case directive decides in each state the action to be taken, according to the given command.

Furthermore it is worth to notice that the members `m_trajectory` together with `m_index` save the current publishing state, i.e. the setpoint list and the index of the current setpoint which

[11]Note that in the table, for representation reasons, `PUBLISHING_FIRST_SETPOINTS` and `WAITING_FOR_MODE_CHANGE` have been changed to `FIRST_SETPOINTS` and `MODE_CHANGE` respectively

[12]See `https://dev.px4.io/en/ros/mavros_offboard.html`, accessed on 2018-09-23

[13]As a C++ reference the book [19] has been used

[14]With exception of `REACHED_HOME`, which is identified and handled in the `spin()` method

has to be published. Additionally, `m_rate` holds the used setpoint publishing rate and gets defined by the incoming coverage flight trigger.

```
//...
namespace bambi {
namespace flight_controller {
class FlightControllerNode
{
public:
    FlightControllerNode(const ros::NodeHandle& nodeHandle);
    void spin();

    enum class State {
        READY,
        PUBLISHING_FIRST_POINTS, // wait a few seconds before changing mode
        WAITING_FOR_MODE_CHANGE,
        FLYING,
        HOVERING,
        REACHED_HOME
    };
    enum class Command {
        COVERAGE_FLIGHT_TRIGGER_RECEIVED,
        TIME_TO_CHANGE_MODE,
        UAV_MODE_CHANGED_TO_OFFBOARD,
        HOVER_TRIGGER
    };
private:
    ros::NodeHandle m_nodeHandle;
    ros::Publisher m_publisherSetPosition;
    //...
    State m_state;
    boost::shared_ptr<bambi_msgs::Trajectory> m_trajectory;
    size_t m_index;
    ros::Rate m_rate;
    //...

    void cb_trigger_coverage_flight(CoverageFlightTrigger& trajectory);
    //...
    void changeState(FlightControllerNode::State newState);
    void handleStateMachineCommand(Command command, const void* msg);
};
}
}
```

Listing 2.3: Flight Controller Class Definition

The actual setpoint publishing is accomplished in the `spin()` method. Listing 2.4 on the next page shows the former *crucial* code snippet from the flight controller node class implementation.

Since `m_rate` is set to the ROS node handle object, calling `spinOnce()` guarantees the publishing to be at the desired rate, currently implemented as 50 Hz. As it can be seen, only in the `FLYING` state, `m_index` gets incremented. The rest o the code is almost self-explaining.

Anyhow, it should be pointed out that a clean implementation requires the use of a *mutex* to avoid a race condition when changing state in the `spin()` method. Since the ROS callbacks may be called from different threads, it is possible that the command-handling method overlaps with the execution of `spin()`. Due to performance reasons the mutex is acquired only if necessary, i.e. when the index reaches the end. For simplicity however, this is not shown in listing 2.4.

```
void FlightControllerNode::spin() {
    ros::AsyncSpinner spinner(4);
    spinner.start();

    while (ros::ok()) {
        switch (m_state) {
        case State::PUBLISHING_FIRST_POINTS:
        case State::WAITING_FOR_MODE_CHANGE:
            m_missionStartPositionTarget.header.stamp = ros::Time::now();
            m_publisherSetPosition.publish(m_missionStartPositionTarget);
            break;
        case State::FLYING:
            ++m_index;
            if (m_index < m_trajectory->setpoints.size()) {
                m_trajectory->setpoints[m_index].header.stamp = ros::Time::now();
                m_publisherSetPosition.publish(m_trajectory->setpoints[m_index]);
            } else {
                // reached home --> return index to save value
                --m_index;
                std_msgs::Bool b;
                b.data = true;
                m_publisherReachedHome.publish(b);
                changeState(State::REACHED_HOME);
            }
            break;
        case State::REACHED_HOME: // continue to publish last setpoint
            m_trajectory->setpoints[m_index].header.stamp = ros::Time::now();
            m_publisherSetPosition.publish(m_trajectory->setpoints[m_index]);
        }
        ros::spinOnce();
        m_rate.sleep();
    }
    spinner.stop();
}
```

Listing 2.4: Setpoint publishing in FlightController

4.4.2 Relative Altitude Handling

Since the NED reference frame is relative to the home position (see section 3.1.1 on page 12) and the altitude profile is not known at priori, keeping a desired altitude relative to ground would be challenging: The internally generated setpoint list, i.e. the trajectory, contains altitudes meant to be *relative to ground*, meanwhile the strict external NED frame $\mathcal{F}_E$ is relative to the home position's altitude, which generally is different.

Fortunately there are PX4 settings, which enable altitude control relative to ground even in OFFBOARD mode. Therefore, no special implementation on the CC is required.

4.5 Trajectory Generator

The trajectory generator needs to generate the actual setpoint. As already discussed, as an input the topic `/bambi/mission_controller/trigger_boundary` provides a `PathWithConstraints` message, i.e. the geometric path in WGS 84 format and the maximum velocity and acceleration magnitudes[15].

Given that the setpoints have to be carried out in the local ENU frame with the home position as origin, first of all the global coordinates have to be transformed into local ones.[16] This work is accomplished by using the `geodesy` ROS package[17].

In particular, listing 2.5 shows the implemented conversion, from WGS 84 to UTM to the local ENU reference frame with the origin in the home position.

```
for(auto point : pathWithConstraints.path.geometric_path){
    geographic_msgs::GeoPoint geoPoint
            = geodesy::toMsg(point.geopos_2d.latitude,
                             point.geopos_2d.longitude);
    geodesy::UTMPoint UTMPoint(geoPoint);
    Point3dRelAltitude pointXYZ_relAlt;
    pointXYZ_relAlt.x = UTMPoint.easting - utmHomePoint.easting;
    pointXYZ_relAlt.y = UTMPoint.northing - utmHomePoint.northing;
    pointXYZ_relAlt.alt = point.altitude_over_ground;
    m_pPathXYZ_relAltitude->push_back(pointXYZ_relAlt);
}
```

Listing 2.5: WGS84 to ENU coordinate frame converstion

In this way all the geometric path is in the local ENU coordinate frame used by PX4. In this way we got nothing but the mathematical path defined in eq. (2.1) on page 10.

Note that there is *no relation with time* yet. To define the relation with time is – again – the *control problem* (section 2.1 on page 10). Its solution is elaborated in the following sections.

[15] See section 4.2 on page 16

[16] See section 3 on page 11

[17] Installable with `sudo apt install ros-kinetic-geodesy`, see `http://wiki.ros.org/geodesy`, package author **O'Quin Jack**, accessed on 2018-09-02

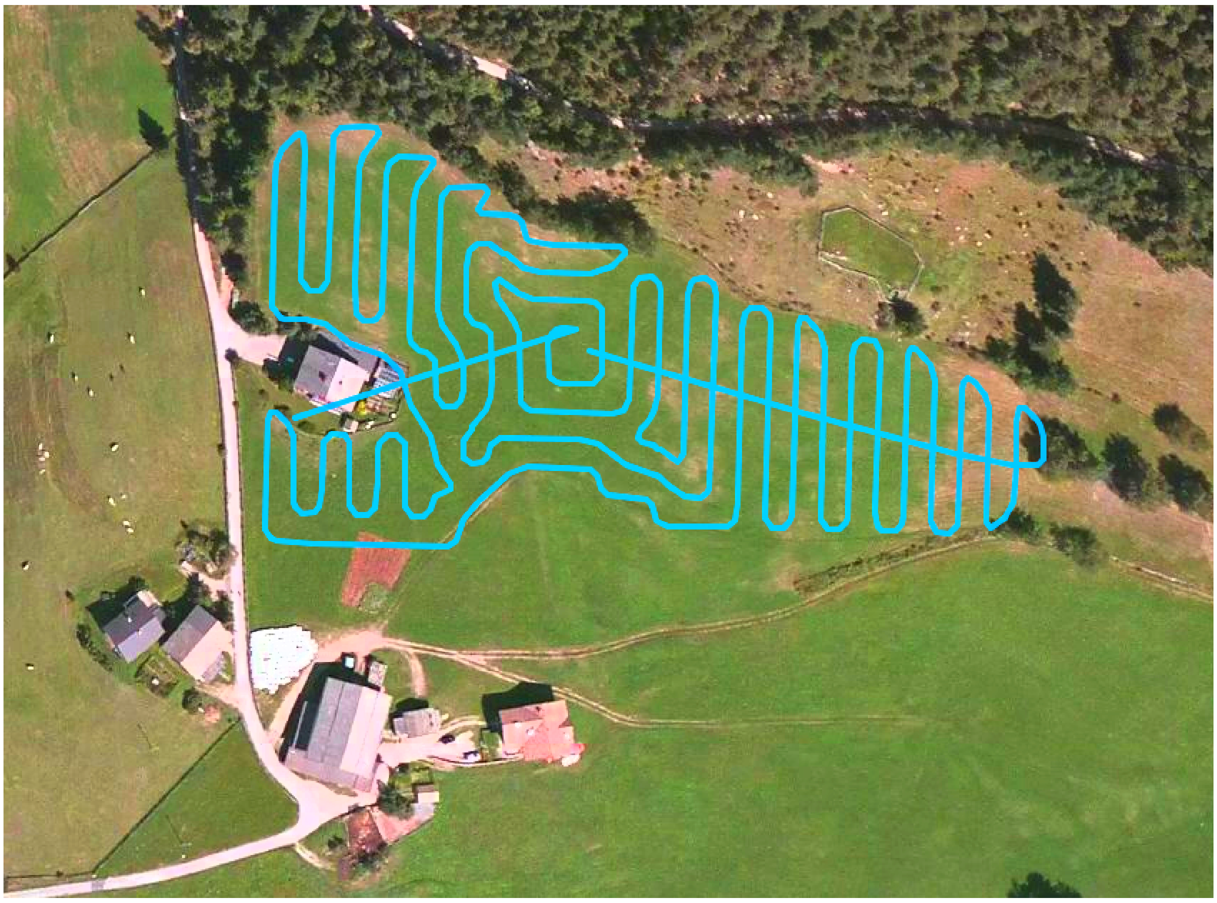


Figure 2.10: Sample Coverage Path

Figure 2.10 shows a typical *geometric* path to follow during a *BAMBI* mission. As one can guess this path has already been smoothed using a BSpline interpolation[18]. The task is therefore to follow *as close as possible* the given path, without introducing any systematic error, such as further interpolation.

5 Constant velocity trajectory

The simplest approach to solve the control problem, defined in section 2.1 on page 10, is a constant velocity trajectory, guaranteeing a constrained change in position per unit time. It is important to point out, that a constant velocity trajectory can *not* guarantee constraints in accelerations. However, it conveniently implies, that the spatial distance between two setpoints is constant, namely $nominalDistance = v_{max}/f_{sp}$, where f_{sp} denotes the setpoint rate in [Hz].

This reduces the problem into dividing the path in equally spaced segments, getting the new setpoints $\langle \boldsymbol{r^*}_k \rangle_{k<=m}$ out of the the geometric path $\langle \boldsymbol{p}_k \rangle_{k \leq n}$ (see eqs. (2.1) and (2.2) on page 10). Figure 2.11 on the facing page illustrates the problem.

[18]See e.g. [20]

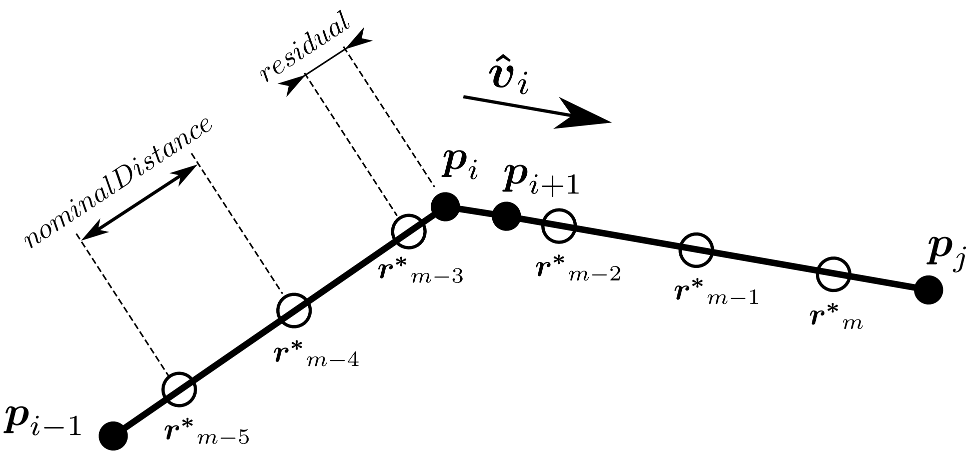


Figure 2.11: Spatial sampling of the geometric path

Reducing the problem to *spatial* path sampling, allows to solve exclusively a *geometric* problem. Figure 2.11 sketches a general situation in which a sequential algorithm could be, i.e. at position i in the list of setpoints $\boldsymbol{p}[]$.

As the illustration already points out, in essence, there may be *two* cases:

1. *More than 1* sample has to be generated between two geometric points, i.e. between $\boldsymbol{p}_{i-1}$ and $\boldsymbol{p}_i$ in the illustration, 3 samples are needed
2. *No sample at all* has to be generated, i.e. between $\boldsymbol{p}_i$ and $\boldsymbol{p}_{i+1}$ no sample is needed for satisfying the velocity constraint and the setpoint rate f_{sp}

For being able to sample an appropriate segment $(\boldsymbol{p}_i, \boldsymbol{p}_j)$, $\boldsymbol{p}_j$ must be identified in the list of points from the geometric path. A next position j has therefore to be determined, which is distant at least $nominalDistance$. This is essentially accomplished by iterating through the geometric path, starting from $i + 1$, until a next point guaranteeing the required distance is found. It is therefore clear, that the segment $(\boldsymbol{p}_i, \boldsymbol{p}_j)$ to be sampled, is generally *not* chosen by $j = i + 1$, as illustrated also in fig. 2.11.

Note that this approach may also *skip* “angles”, i.e. the original shape could get lost. This can be observed in the following cases:

1. If there is a *residual* distance which could not be sampled in the previous step, the corner point $\boldsymbol{p}_i$ e.g. will not be part of the setpoint trajectory. Indeed, considering the samples $\boldsymbol{r}^*{}_{m-3}$ and $\boldsymbol{r}^*{}_{m-2}$ it is evident that a straight line connecting them will not pass through $\boldsymbol{p}_i$.
2. If the point $\boldsymbol{p}_{i+1}$ was not on the segment $(\boldsymbol{p}_i, \boldsymbol{p}_j)$, and neither distant more than $nominalDistance$, the generated samples would lie anyways on the segment $(\boldsymbol{p}_i, \boldsymbol{p}_j)$ without considering the eventual corner introduced by $\boldsymbol{p}_{i+1}$.

In any case, those introduced errors are governed by the sample rate and can be seen as a natural outcome of the used procedure. The algorithm undeniably has to be *lossy*, considering the *discrete* format of the coordinates. If those kind of issues had an impact in a practical implementation, the sample rate simply would have to be increased.

Once the segment to be sampled is chosen, it is anyhow *always* possible to determine a unit vector $\hat{\boldsymbol{v}}_i$ which points from $\boldsymbol{p}_i$ to $\boldsymbol{p}_j$. Using that vector the sample generation is quite intuitive.

In particular, multiplies of *nominalDistance* are used to add to the position $\boldsymbol{p}_i$ equally spaced portions in direction of $\hat{\boldsymbol{v}}_i$.

It is worth to point out that numerical problems involving the calculation of the unit vector $\hat{\boldsymbol{v}}_i$ are avoided implicitly by choosing the segment distance to be greater than *nominalDistance*.

Out of the given figure with the provided reasoning, an algorithm can be deduced. Algorithm 2.1 shows a pseudocode of the used implementation.

Algorithm 2.1 Constant Velocity Setpoint Generation

```
Input: $\boldsymbol{p}_1 \dots \boldsymbol{p}_n$
Output: $\boldsymbol{r}^*{}_1 \dots \boldsymbol{r}^*{}_m$

function GenerateSetpoints($\boldsymbol{p}[\,], v_{max}, f_{sp}$)
    $nominalDistance \leftarrow v_{max}/f_{sp}$
    $m \leftarrow 0$                                         ▷ $m$ denotes the number of setpoints
    $i \leftarrow 1$                                         ▷ $i$ denotes the index in the path points list
    $m \leftarrow m + 1$
    $\boldsymbol{r}^*[m] \leftarrow \boldsymbol{p}[i]$                                ▷ Insert first point
    $residual \leftarrow 0.0$
    $reachedEnd \leftarrow$ false

    while not $reachedEnd$ do
        $j \leftarrow i$
        repeat
            $j \leftarrow j + 1$
            if $j > n$ then
                $reachedEnd \leftarrow$ true
            end if
            $distance = residual + $ Distance($\boldsymbol{p}[i], \boldsymbol{p}[j]$)
        until $distance \geq nominalDistance$

        if not $reachedEnd$ then
            $s \leftarrow$ Floor($distance/nominalDistance$)          ▷ $s$ samples between $i$ and $j$
            $\hat{\boldsymbol{v}} \leftarrow (\boldsymbol{p}[j] - \boldsymbol{p}[i])/$Distance($\boldsymbol{p}[i], \boldsymbol{p}[j]$)   ▷ Unit Vector from $\boldsymbol{p}[i]$ to $\boldsymbol{p}[j]$
            for $k \leftarrow 1$ to $s$ do
                $\boldsymbol{r}^*{}_{new} \leftarrow \boldsymbol{p}[i] + (k \cdot nominalDistance - residual)\,\hat{\boldsymbol{v}}$
                $m \leftarrow m + 1$
                $\boldsymbol{r}^*[m] \leftarrow \boldsymbol{r}^*{}_{new}$
            end for
            $residual \leftarrow$ Distance($\boldsymbol{r}^*[m], \boldsymbol{p}[j]$)
            $i \leftarrow j$
        end if
    end while
    $m \leftarrow m + 1$
    $\boldsymbol{r}^*[m] \leftarrow \boldsymbol{p}[n]$                                ▷ Insert last point
    return $\boldsymbol{r}^*[\,]$
end function
```

Note that algorithm 2.1 on page 26 assumes the existence of the functions Floor and Distance which round floating point values downward to the next integer value and calculate the euclidean distance respectively.

The actual implementation differs just in some technical details, i.e. different object types and the fact that the velocity is constrained just in a 2D plane, since we do not predict the landscape ground profile, see section 4.4.2 on page 22. It is therefore not useful to consider the velocity limitations along the z-axis, because the actual ground profile may vary significantly and hence also the related *actual* velocity along z^B. In the actual flight dynamics enough rest trust has to be reserved for correct ground profile following.

Listing 2.6 shows a brief code snippet from the implementation, containing the initialization of the setpoint object (PositionTarget, see listing 2.2 on page 18).

```
    //prepare PositionTarget message
    PositionTarget posTargetLocal;
    posTargetLocal.coordinate_frame = PositionTarget::FRAME_LOCAL_NED;
    posTargetLocal.type_mask = PositionTarget::IGNORE_AFX |
            PositionTarget::IGNORE_AFY |
            PositionTarget::IGNORE_AFZ |
            PositionTarget::IGNORE_YAW_RATE;
```

Listing 2.6: Setpoint object preparation in TrajectoryGeneratorNode

As it can be seen, some properties for the reference frames have to be set, as well as flags made for ignoring the unused acceleration values, to enable a valid object recognition in the PX4 firmware implementation.

6 Velocity Feed Forward Control

Since the trajectory is now defined, i.e.[19]

$$\boldsymbol{r}^*(t) = \begin{pmatrix} x^*(t) \\ y^*(t) \\ z^*(t) \end{pmatrix} \tag{2.7}$$

the control behavior can easily be improved by adding a velocity feed forward path. As discussed in section 1.1 on page 8, PX4 currently just supports a feed forward path for the first order derivate, i.e. only velocity feed-forward.

Feed-forward control is a well known approach (see e.g. [21]), which introduces control action *in absence of feedback errors*. This basically means that, instead of waiting for a position error, the internal velocity controller is already triggered by the provided values.

To calculate the velocity values, the first order derivative in time of eq. (2.7), i.e. $\dot{\boldsymbol{r}}^*(t)$ is needed. Performing the numerical derivatives of the generated setpoint sequence (see eq. (2.6) on page 11) can be accomplished in different ways. The most simple approach is the finite difference method. In [22] such local methods, among others, are compared with *global methods*. The author suggests to prefer global over local methods.

[19]see also eq. (2.2) on page 10

In the specific usecase, the applied setpoint frequency is about 50 Hz, which comes down to a period of 0.02 s. Although 32-bit floating point precision should give fairly satisfying results, a global method has been used. It consists in building up a spline interpolation and exploit its derivatives.

6.1 Implementation

Professor Enrico Bertolazzi from the University of Trento published online[20] a C++ library providing spline interpolation. The generated splines are carried out with derivatives up to the 4th order.

Three curves $x^*(t), y^*(t), z^*(t)$ are prepared inside the setpoint generation loop of algorithm 2.1 on page 26. Later on in the trajectory generator node the following code snippet in listing 2.7 is sufficient to provide the needed velocity values.

```
    curveX->build();
    curveY->build();
    curveZ->build();

    for (int i = 0; i < m_pPositionTrajectoryENU->size(); ++i) {
        // better computation than addition in loop
        double t = sampleTime * i;
        mavros_msgs::PositionTarget& setpoint
                = m_pPositionTrajectoryENU->operator[](i);
        setpoint.velocity.x = curveX->D(t);
        setpoint.velocity.y = curveY->D(t);
        setpoint.velocity.z = curveZ->D(t);
        // head always towards where we are going
        setpoint.yaw = atan2(setpoint.velocity.y, setpoint.velocity.x);
        setpoint.yaw_rate = atan2(curveY->DD(t), curveX->DD(t));
    }
```

Listing 2.7: Derivative calculation in TrajectoryGeneratorNode

The `build()` method call on the spline objects at the beginning of listing 2.7, interpolates the previously added points by calculating internally the coefficients. Later on, the derivatives can be easily be computed by using `D(t)`, which yields the 1st order derivatives.

To get the right yaw heading the direction of the *velocity* vector is computed. Most conveniently, the function `atan2` from the math library is used. The yaw *rate*, i.e. the angular velocity of the yaw angle, in contrast is given by the direction of the acceleration vector.

[20]See `https://github.com/ebertolazzi/Splines`, referring inter alia to [20]

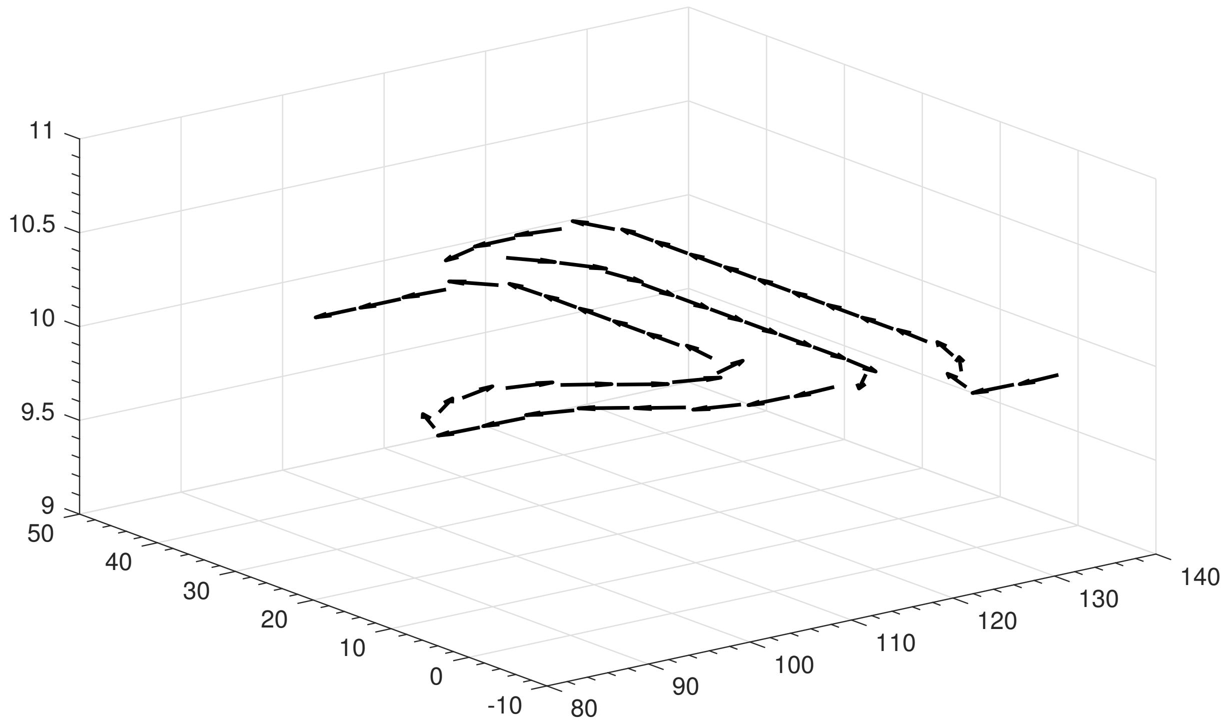


Figure 2.12: Velocity Vector Plot

To qualitatively control the results from the algorithms, the velocity vectors can be plotted along the generated trajectory. Figure 2.12 shows a 3-dimensional representation in the ENU frame of a chosen sector from the trajectory. As it can be seen, the vectors have – as desired – a fixed length, in this specific case of 4 m/s.

7 Simulation Results

As pointed out already in section 1.1 on page 8, PX4 supports various simulation frameworks. Major plugins, including laser distance sensor and more, are freely available.[21].

The used simulation options are evaluated in a master thesis from the Polytechnic University of Milan [23].

7.1 SITL Environment

One possible choice for simulating FCUs is known as Software In The Loop (SITL). It implies pure *software* simulation running *in the loop*, which basically means that physical actuators are simulated in a (software) physics engine, similar to game engines, which provide *software feedback* of various simulated sensors. In this way the FCU runs *in the loop*, i.e. with feedbacks from the physics simulation.

[21]See e.g. `https://dev.px4.io/en/simulation/gazebo.html`, accessed on 2018-09-06

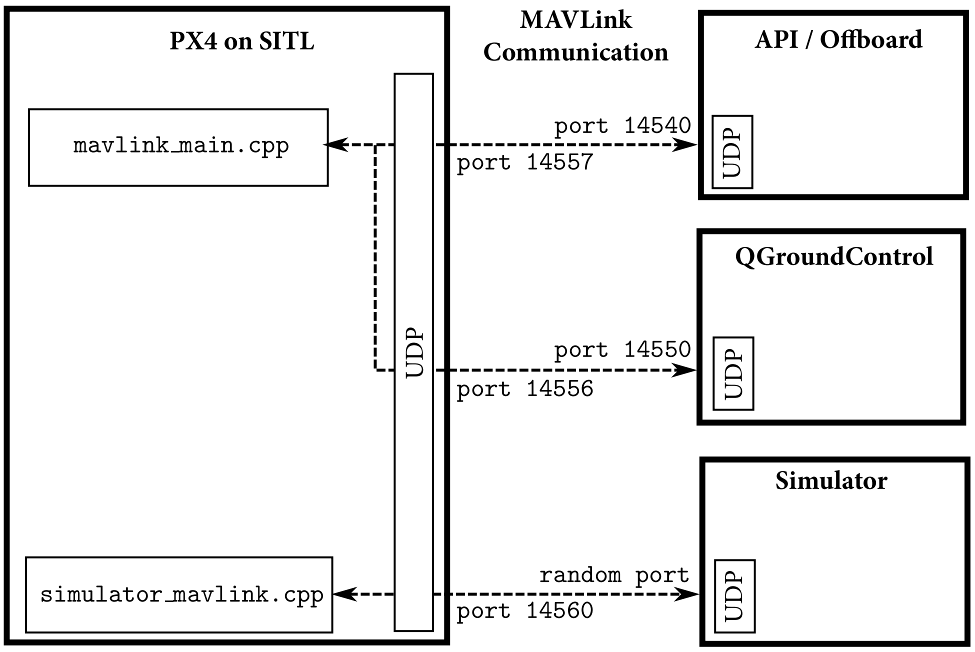


Figure 2.13: Simulation Architecture

Figure 2.13[22] shows the general communication architecture between the different components in the simulation environment. MAVLINK, routed through User Datagram Protocol (UDP), is used to exchange the relevant information. Since it all happens on the *same* physical system, different UDP *ports* are used. Note that for any two-way communication *two* different UDP sockets on the receiver sides are opened, at the specified port.

The simulator in the right lower corner is the actual physics engine. Different physics engines are supported in PX4. However, mainly for the availability of models and advanced sensors such as 2-dimensional laser scanners, *Gazebo* has been chosen.

MAVROS together with the Bambi autopilot is running in the represented API / Offboard box. From the QGroundControl software the mission trigger can be fired, after which the simulation autonomously carries out the whole coverage flight.

The simulation can be supervised through the 3D rendering and, more importantly, analyzed through the regularly generated flight logs.

7.1.1 Gazebo

Gazebo is a common open source *robot* simulator which provides a lot of different environments for indoor and outdoor simulations. It includes a robust physics engine, a graphical visualization and good integration with ROS.

The simulation for the Bambi Project has been customized in various aspects. On of them is the home position, which has been updated to the real test field, by setting the environment variables `PX4_HOME_LAT`, `PX4_HOME_LON` and `PX4_HOME_ALT`.

The simulation is started by executing the provided ROS "Iris" model launch files with updated 3D worlds.

[22]See also `https://dev.px4.io/en/simulation/#sitl-simulation-environment`, accessed on 2018-09-24

7.1.2 Simulated 3D World

The use of the `staticMapPlugin` shipped with Gazebo allows to render Google Earth tiles as a ground plane. Although the altitude profile could not be included in this way, the realistic ground image effectively marked the real field borders.

On top of the flat ground, some 3D objects were placed in order to provide obstacles and and more realistic look. Figure 2.14 shows a screen-shot of the modified Gazebo simulation world.

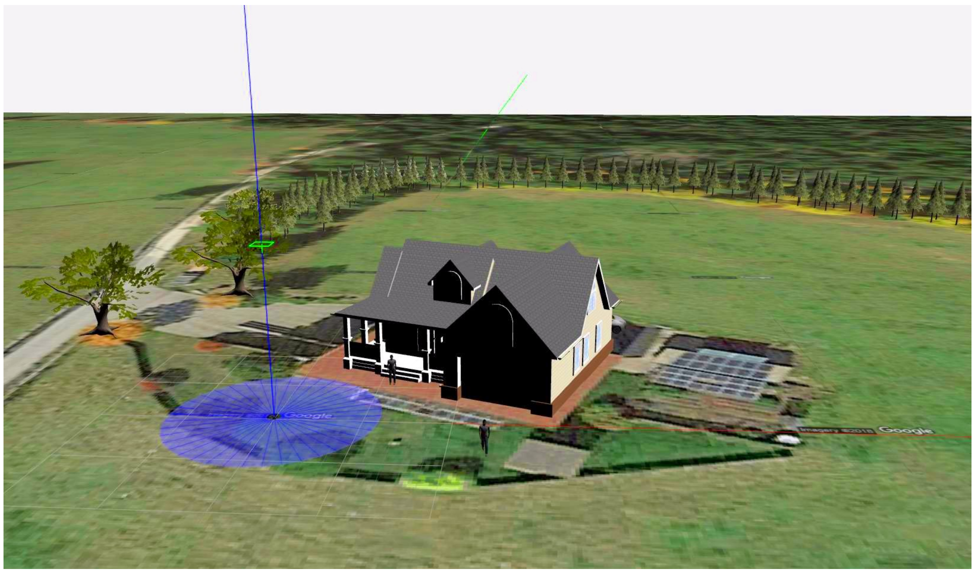


Figure 2.14: Gazebo Simulated 3D World

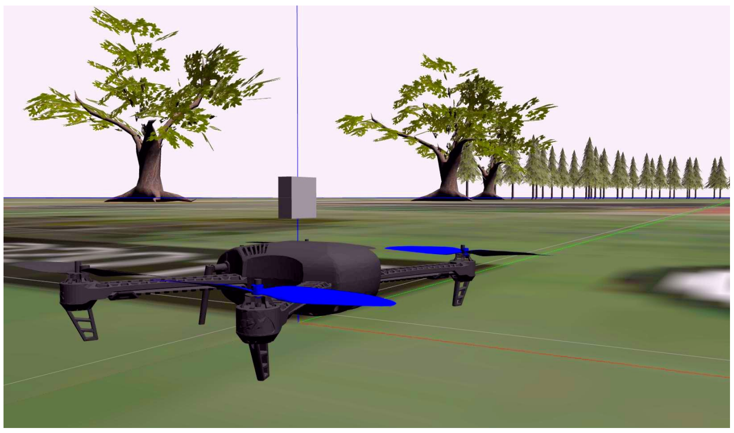


Figure 2.15: Simulated IRIS Model

As a model, the "Iris" quadcopter was used, shown in fig. 2.15. The main reason for the choice is the availability of sensors that can be mounted on top of the model, without the need for programming or modeling their 3D appearance.

7.2 Tracking Performance Evaluation

To compare the tracking performance, a BAMBI mission has been triggered, and the `*.ulg` flight log file has been analyzed. This could be accomplished mainly by using the `ulog2csv` program, which is part of the PX4 software collection.

Figures 2.16 and 2.17 on this page and on the next page show the test flights, with and without velocity feed forwarding. Both flights have been performed using a constant velocity trajectory of 4 m/s.

At the first look, the standard position tracking without velocity feed forwarding seems to have a better performance, especially in the right part of the graph. The turning points are handled way better by the simpler approach, i.e. with far less overshoot. In the velocity feed-forwarding case, those overshoots amount to almost 2 m.

The estimated position in the standard approach is more often directly on the desired track than in the velocity feed-forwarding case.

Anyhow, it is important to clarify that those plots lack of time dependency, i.e. the relation with time is lost, when plotting x^E against y^E (see section 3.1.1 on page 12).

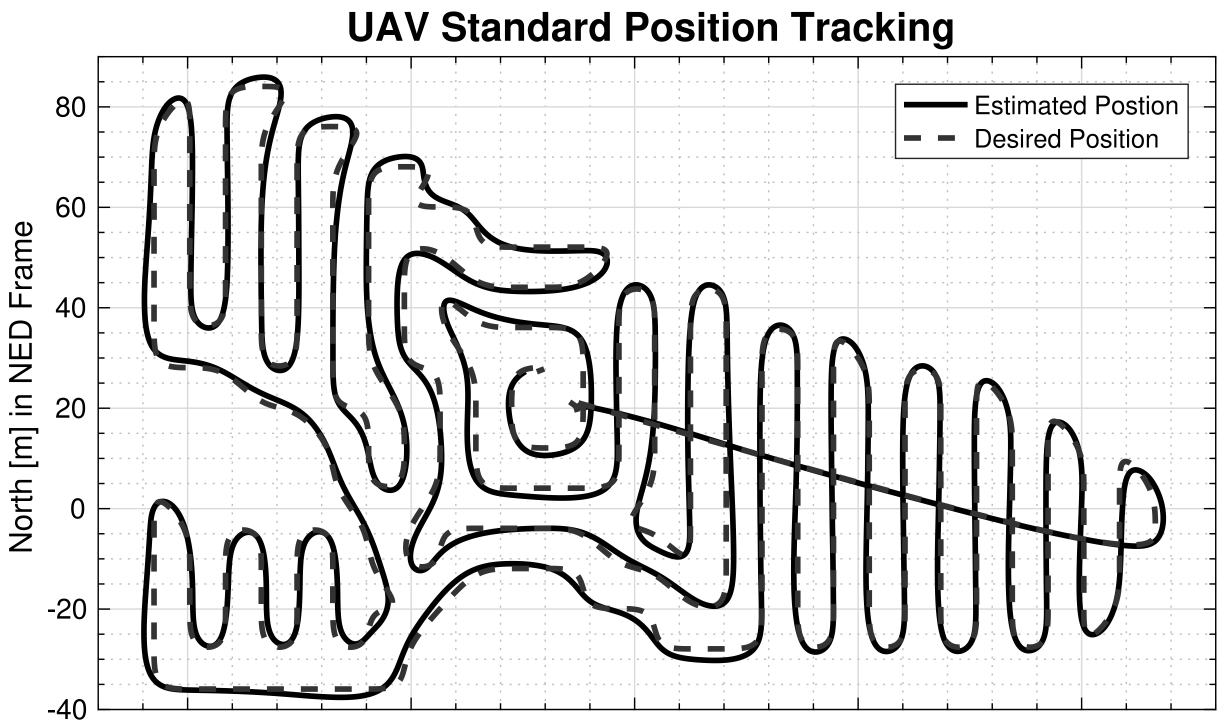


Figure 2.16: Simulation Standard Position Tracking Performance

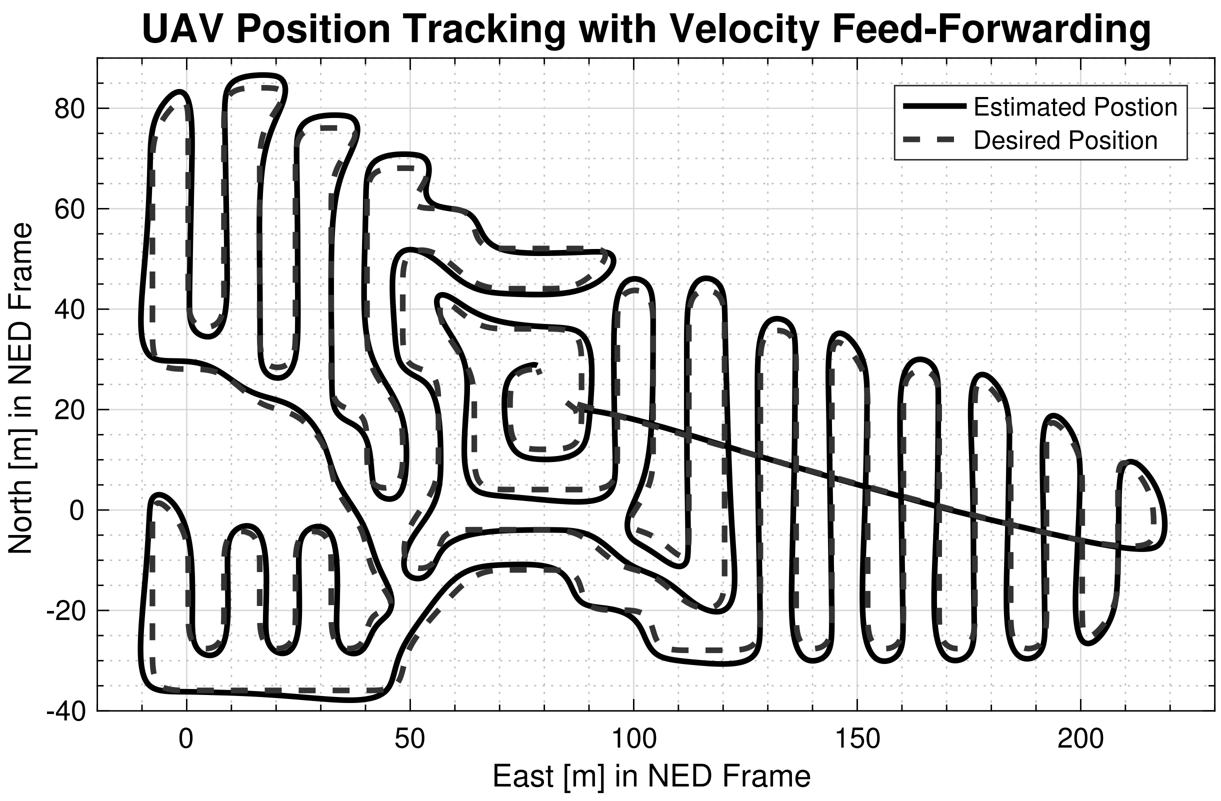


Figure 2.17: Simulation Velocity Feed-Forwarding Tracking Performance

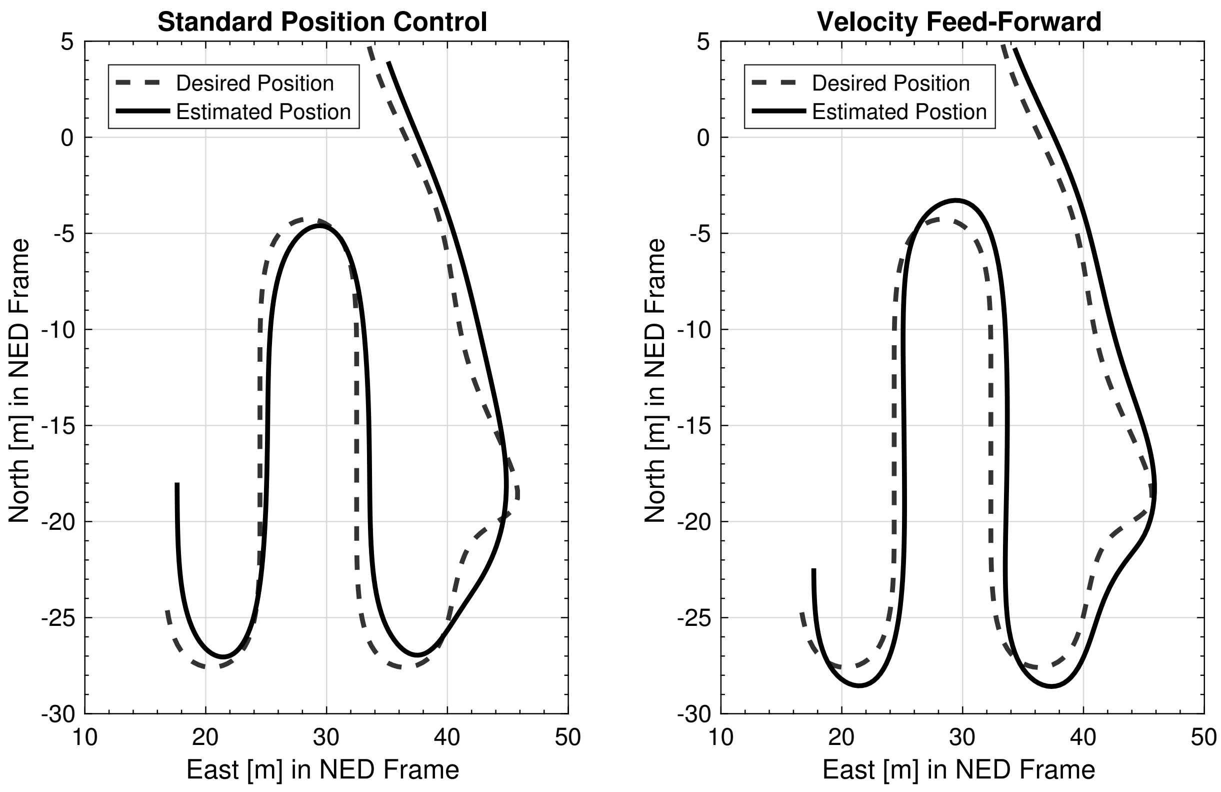


Figure 2.18: Simulation Position Tracking Comparison

Figure 2.18 gives a better insight on the tracking differences. It clearly can be observed, that velocity feed forwarding, although presenting a bigger overshoot at the turning points, *preserves* the original shape of the trajectory much better.

It is also evident, that without velocity feed-forwarding the actual position lacks far more behind the desired position. This can be seen considering the starting point of the setpoint curve with respect to the starting point of the estimated position curve.

To go further into detail in the time dependency, table 2.3 can be considered. It gives a comparison on the tracking errors in the 2-dimensional plane, i.e.

$$e(t) = \sqrt{(\Delta x)^2 + (\Delta y)^2} \tag{2.8}$$

	Standard Position Control	**Velocity Feed-Forward**
Mean Value	5.507 m	2.195 m
Standard Deviation	1.308 m	1.482 m
Minimum	0.693 m	0.001 m
Maximum	8.397 m	5.589 m

Table 2.3: Position Tracking Error Comparison

With a difference of more than 3 m in the mean position tracking error, the responsiveness of a velocity feed-forward approach is verified as expected.

Even better insights give the boxplots in fig. 2.19. Eminently, the tracking error during velocity feed forward reaches *basically zero* in some cases, which shows the effective control action of the feed forward path. Using just a feedback action, indeed entails a minimum error of approximately 70 cm.

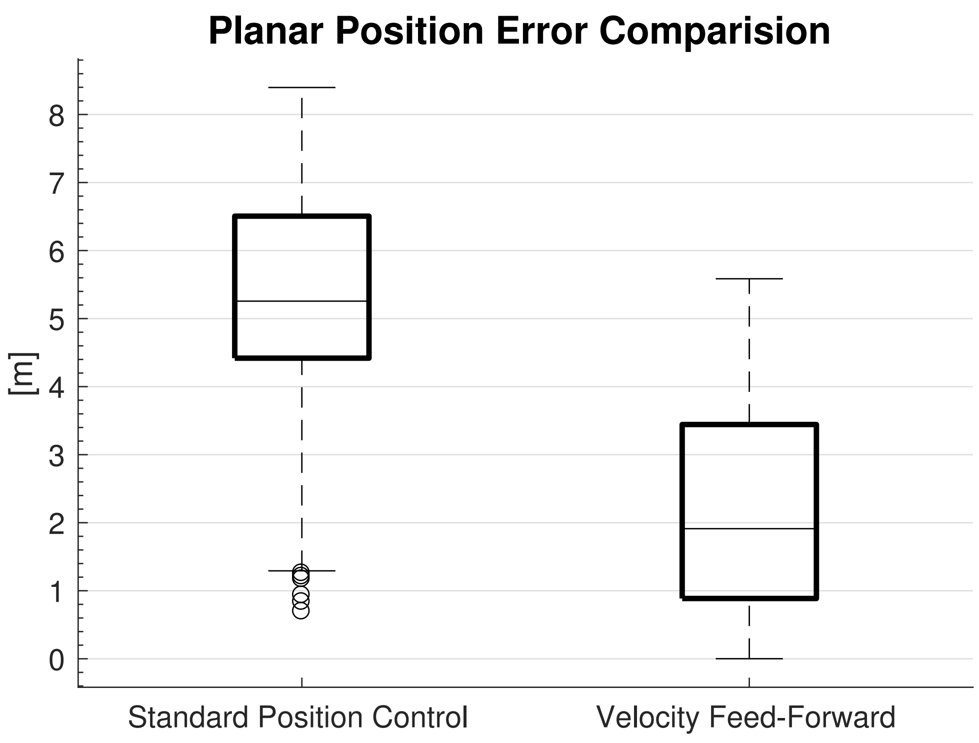


Figure 2.19: Position Error Comparison

To further improve tracking, the use of an acceleration feed-forward path could be useful, but it is not available in PX4 yet.[23]

The real problem however, is that the multicopter dynamics are not considered during trajectory generation. Maintaining a constant velocity of 4 m/s in the sharp turns is physically impossible, which is why significant overshoots occur.

[23] See `https://github.com/px4/Firmware/blob/c5b954daccdcc5195e146844f42699cdce63650d/src/lib/FlightTasks/tasks/Offboard/FlightTaskOffboard.cpp#L167`

Chapter III

Collision Avoidance

In this chapter a simple but effective security measure is developed, avoiding collisions during the mission's flight by detecting obstacles through a 2D laserscan. Simulation results are presented and discussed.

1 Use Case Analysis

The Bambi Project, performing a coverage flight over a cultivated grassland field, in the nominal flight behavior *should not* encounter major obstacles. The mountainous landscape challenges the autopilot at most with some alone standing trees or transmission towers. Major obstacles such as forests or living houses are *already excluded* by the generated coverage path.

In this way, the collision avoidance has just to be a *security measure*, since encountering obstacles is *not* the nominal case. A security measure has to react *as fast as possible*, avoiding any kind of delays: The flight velocity is planned to be around 5 m/s, which means e.g. just 500 ms of delay would introduce a change of 2.5 m in position, towards the obstacle in the worst case.

2 Control Signal Application

Figure 3.1 shows the control diagram of the PX4 flight-stack.

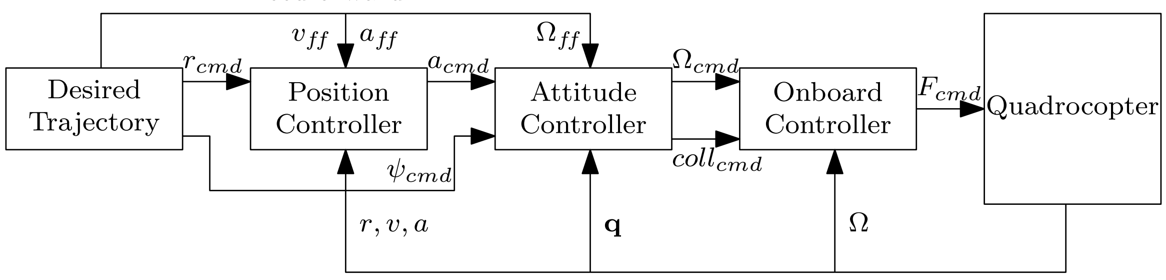


Figure 3.1: PX4 Multi Copter Controller Diagram[1]

[1]Taken from [24, p. 5]

Since a detected obstacle needs to influence in some way the multicopter's position, at a certain level in the control loop, an additional control component has to be applied, namely the one responsible for performing an evasive maneuver.

It is of common knowledge that inner control loops are generally faster. If the difference of the smallest time constant is of more than one order of magnitude, some design techniques for the outer loops allow even to neglect the existence of the inner loops.

Since reaction time is a major concern in a security measure, it is most convenient to apply the control signal in the *inner* loops. The control signals in the inner loops however, refer to different systems of reference. The attitude controller e.g. acts on orientation angles of the body reference frame $\mathcal{F}_B$. The $\boldsymbol{a}_{cmd}$ is still expressed in the external NED reference frame $\mathcal{F}_E$, hence it is advisable to apply the collision avoidance signal to $\boldsymbol{a}_{cmd}$. In this way the change of system of reference from the laser scan, attached to the body frame, to $\boldsymbol{a}_{cmd}$, i.e. the NED frame, is realizable without major complications. In particular the object avoidance control signal is applied as a *sum* to $\boldsymbol{a}_{cmd}$. In this way the trajectory tracking performed by the position controller will not be stopped, as desired.

In the PX4 implementation, $\boldsymbol{a}_{cmd}$ is known as normalized thrust setpoint $\boldsymbol{F}_{sp}$.

3 Distance Sensing

The distance sensing, as already discussed in section 2.2 on page 5 will be performed by a 2D laser scan. There are many device available on the market, such as ``RPLidar''. Sample rates can be in the order of 10 000 Hz.

3.1 Available Hardware

To avoid excessive development expenses, a low-cost solution is chosen. It consists of a single distance measurement rotated by a stepper motor, which is under control by an Arduino Mini. The used Light Detection and Ranging (LIDAR) device is the *Benewake Mini*. It measures under good conditions up to 8 m and more, at a sampling frequency of 100 Hz.

4 Collision Avoidance Thrust Definition

Supposing to have arbitrary radial distance values $d(\theta)\,\forall\theta$ in the *body* NED frame[2], the question arises in which direction should the vehicle counteract in order to *avoid a collision*. The intuitive approach is to define a unit direction vector pointing to the opposite direction:

$$\hat{a}_{colission} = \frac{1}{\int_0^{2\pi} w\big(d(\theta)\big)d\theta} \int_0^{2\pi} w\big(d(\theta)\big) \begin{pmatrix} -\cos\theta \\ -\sin\theta \end{pmatrix} d\theta \tag{3.1}$$

Where $w(d)$ is a weight function, that generally should account *more* for smaller distances, since they present a higher risk. It is worth noting that the resulting direction $\hat{a}_{colission}$ from eq. (3.1) is nothing but the integral mean value of the weight function, to be applied in the local *body NED* frame.

[2]Note that this means having clockwise angles starting from body-north when seen from above, see section 3 on page 11

In practice though, the used sensors will always have a minimum and a maximum measurable distance, so just in some cases a *valid measurement* can be made. To account for this fact in eq. (3.1) on page 38, we may introduce

$$\delta_{valid}(d) = \begin{cases} 1 \text{ if } D_{min} <= d <= D_{max} \\ 0 \text{ otherwise} \end{cases} \tag{3.2}$$

This can be easily incorporated in the weight function:

$$w(d) = \delta_{valid}(d)\, w^*(d) \tag{3.3}$$

Note that in this way, eq. (3.1) on page 38 becomes valid if, and only if

$$\exists\, \theta \in [0, 2\pi] \mid \delta_{valid}\big(d(\theta)\big) \neq 0 \tag{3.4}$$

In the actual implementation this gets implemented by some `if` directive.

4.1 Thrust Magnitude

The magnitude, or *how intensive* the reaction is, should be determined by the smallest distance, or in other words, considering the highest weight:

$$\max_{\theta \in [0, 2\pi]} w(d(\theta)) \tag{3.5}$$

Note that this is not equivalent to making the mean reaction: It considers *always* the worst case, i.e. the highest danger, instead of making «a mean value of dangers».

To get a reasonable guess on a right weighting approach, a simple example calculation shall be considered. Suppose a vehicle moving at a given velocity v_0 with respect to some inertial reference frame, has to stop at a certain distance d_{stop} using a constant acceleration of magnitude a_{req}. The governing equation is:

$$x(t) = v_0\, t - \frac{1}{2}\, a_{req}\, t^2 \tag{3.6}$$

At a time t_f, where the vehicle should have been stopped, the following conditions can be imposed:

$$\begin{cases} x(t_f) = d_{stop} \\ \dot{x}(t_f) = 0 \end{cases} \Rightarrow \begin{cases} v_0\, t_f - \frac{1}{2}\, a_{req}\, t_f^2 = d_{stop} \\ v_0 - a_{req}\, t_f = 0 \end{cases} \tag{3.7}$$

$$\tag{3.8}$$

Solving the system by eliminating t_f yields:

$$a_{req} = \frac{v_0^2}{2\, d_{stop}} \tag{3.9}$$

This means that the required acceleration to stop the vehicle is inversely proportional to the available braking distance and quadratically proportional to the initial speed.

A first approach may therefore be:

$$w(d) \propto \frac{1}{d} \tag{3.10}$$

5 Implementation

To accomplish the implementation, eqs. (3.1) and (3.5) on page 38 and on page 39 have to be *discretized*, i.e.:

$$\hat{a}_{colission} = \frac{1}{\sum_{i=1}^{n} w\left(d_i\right)} \sum_{i=1}^{n} w\left(d_i\right) \begin{pmatrix} -\cos\left(i\,\frac{2\pi}{n}\right) \\ -\sin\left(i\,\frac{2\pi}{n}\right) \end{pmatrix} \tag{3.11}$$

where d_i with $1 \le i \le n$ denotes the sample in the i^{th} segment. Note that the segment size $\Delta\theta$ cancels out, since it can be brought out of the sum in numerator and denominator. As a weight function, the one with the incorporated validation from eq. (3.3) on page 39 has to be used.

The actual weight can be taken as:

$$w^*(d) = \frac{D_{max}}{d} - 1 \tag{3.12}$$

with $D_{min} \le d \le D_{max}$. Figure 3.2 shows an example plot with $D_{max} = 6\,\text{m}$.

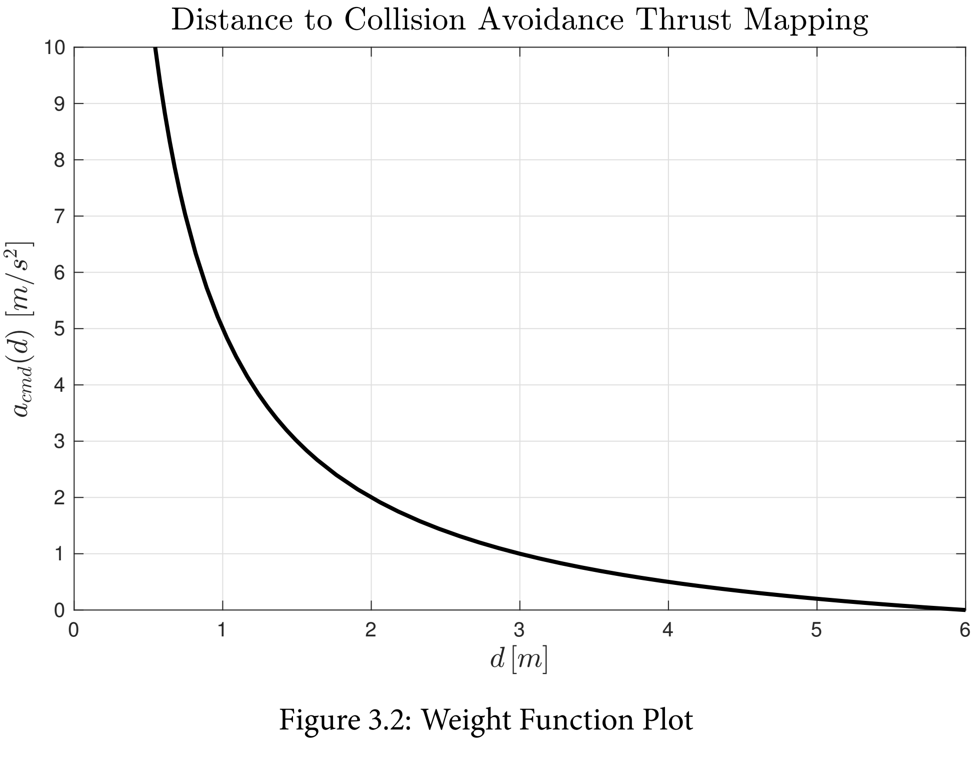


Figure 3.2: Weight Function Plot

Starting from the maximum detectable distance D_{max} a *smooth* control action is introduced. This means that when an obstacles enters the scanning field, no *abrupt* reaction occurs, but rather a softly increasing thrust against the obstacle is applied.

As pointed out already, in the PX4 implementation $\boldsymbol{a}_{cmd}$ is implemented as a *normalized thrust* setpoint $\boldsymbol{F}_{sp}$, i.e. the value 1.0 means full usage of the available thrust. So unlike the axis label in fig. 3.2 on page 40 suggests, the real implementation uses a dimensionless thrust action of the same magnitude.

The magnitude gets, as already mentioned, summed to the thrust desired by the position controller. Afterwards a saturation is applied, in order to maintain the vehicle's attitude under secure limits. It thus makes sense to apply also values greater than 1.0, in order to eventually be able to cancel out and act against the desired flight direction, which may be towards the obstacle.

As it can be seen in fig. 3.2 on page 40 at half of the detectable range, full thrust is applied.

The further crucial step is to get the distance information into the PX4 firmware. A standard MAVLINK message is used for this purpose. Any kind of LIDAR can simply transmit a message of the given type through MAVLINK to the FCU, and obstacle avoidance gets handled at firmware level.

5.1 MAVLINK Message

The standardized MAVLINK[3] message #330 can grab all the information of a 2D laserscan. A compact version of the definition is shown in table 3.1[4]

Field Name	Type	Description
time_usec	uint64_t	Timestamp
sensor_type	uint8_t	Class id of the distance sensor type
distances	uint16_t[72]	Distance of obstacles around the UAV with index 0 corresponding to local North. A value of 0 means that the obstacle is right in front of the sensor. A value of max_distance +1 means no obstacle is present. A value of UINT16_MAX for unknown/not used. In a array element, one unit corresponds to 1cm.
increment	uint8_t	Angular width in degrees of each array element
min_distance	uint16_t	Minimum distance the sensor can measure
max_distance	uint16_t	Maximum distance the sensor can measure

Table 3.1: MAVLINK Obstacle Distance Message Definition

5.1.1 Gazebo Laser Scan

For Gazebo, as already mentioned in section 7.1.1 on page 30 there exists a plugin which emulates the RPLidar. It can be configured in terms of minimum distance, maximum distance and sampling frequency. Fortunately it can be set up to collaborate with MAVROS for directly publishing the required mavlink message.

During the implementation one might encounter some open bugs like `https://github.com/PX4/Firmware/issues/9156`, which are quite recent, underlining again the active development and large community working with PX4. The following configuration however, worked out as expected:

[3]See section 1.1.2 on page 8 on page 8

[4]See `http://mavlink.org/messages/common#OBSTACLE_DISTANCE`, accessed on 2018-09-08

```xml
<sensor name="laser" type="ray">
  <!-- ... -->
  <plugin name="laser" filename="libRayPlugin.so" />
  <plugin name="gazebo_ros_head_rplidar_controller"
          filename="libgazebo_ros_laser.so">
    <robotNamespace>mavros</robotNamespace>
    <topicName>obstacle/send</topicName>
    <frameName>rplidar_link</frameName>
  </plugin>
  <!-- ... -->
</sensor>
```

Listing 3.1: Iris RPLidar Configuration

The crucial configuration is the right setup of the ROS namespace and topic in listing 3.1. Furthermore, to avoid invalid readings, the minimum distance has to be set to 40 cm for the iris model.

5.2 PX4 Firmware Modification

By following the controller scheme from fig. 3.1 on page 37 into the firmware of PX4, it results that the right point to adapt the control logic for collision avoidance is in the file `mc_pos_control_main.cpp`.

In listing 3.2 on the next page, a simplified version of the code modification is shown.

The first part consists in parsing the MAVLINK message as specified by table 3.1 on page 41. Basically the program iterates through all the segments which have been measured by considering only the valid onces. This makes up an implemented version of the mathematical function δ_{valid} from eq. (3.2) on page 39. Along the valid segments, the smallest one is determined as well as the weight for determining the direction as specified in eq. (3.11) on page 40. Unlike in the equation, there is no practical need to divide by the length to obtain a unit vector, since the resulting angle, given by `atan2()` would remain unchanged. However, this calculation is only performed if there is at least one valid angle, which is nothing but the condition provided in eq. (3.4) on page 39.

It is important to point out that the gazebo simulation provided the obstacle distance position counter-clockwise from body *south*, not as specified by the MAVLINK message in table 3.1 on page 41. This is why a rather particular angle calculation is carried out, i.e. $\frac{\pi}{2} - \theta_{obstacle}$.

In order to transform the angle from the body reference frame $\mathcal{F}_B$ to the external NED frame $\mathcal{F}_E$[5], the state variable *yaw* is used. The function `fmod()` then normalizes the angle, in such a way that it is in the interval $[0, 2\pi]$.

The minimum distance is then filtered, with a Low Pass (LP) filter for experimental reasons, to see if eventual oscillations can be damped. This makes it necessary to internally save the last valid angle, because it may happen that the filtered thrust is still active while no obstacle is detected anymore.

[5]see section 3 on page 11

```
int numberOfSummedAngles = 0;
float segmentWidthInRad = math::radians((float)_obstacle.increment);
float obstacleLocalN = 0.f;
float obstacleLocalE = 0.f;
float objectAvoidanceThrustMagnitude = 0.f;
_ca_thrust.lowest_distance = _obstacle.max_distance;

for (int i = 0; i < 72; ++i) {
    if (_obstacle.distances[i] == UINT16_MAX) {
        break; // END REACHED
    }

    if (_obstacle.distances[i] <= _obstacle.min_distance ||
            _obstacle.distances[i] >= _obstacle.max_distance) {
        // skip invalid element
        continue;
    }

    // VALID SEGMENT
    ++numberOfSummedAngles;

    float weight = -1.f + _obstacle.max_distance / _obstacle.distances[i];

    if (_ca_thrust.lowest_distance > _obstacle.distances[i]) {
        _ca_thrust.lowest_distance = _obstacle.distances[i];
    }

    obstacleLocalE += weight * sinf(i*segmentWidthInRad);
    obstacleLocalN += weight * -cosf(i*segmentWidthInRad);
}

// filter minimum distance
_ca_thrust.lowest_distance_filtered =
        _lpFilterObjectAvoidance.apply(_ca_thrust.lowest_distance);

if (numberOfSummedAngles > 0) {
    float angleLocalNE = M_PI/2.f - atan2(obstacleLocalN, obstacleLocalE);
    float angleGlobalNE = fmod((_states.yaw + obstacleAngleLocalNE), 2*M_PI);
    // angle valid for update
    _lastAngle = obstacleAngleGlobalNE;
}

float weight = -1.f +
        _obstacle.max_distance / _ca_thrust.lowest_distance_filtered;

weight = std::fmax(weight, 0.f);

_resultantThrust(0) = -cosf(_lastAngle) * weight;
_resultantThrust(1) = -sinf(_lastAngle) * weight;
```

Listing 3.2: Collission Avoidance Thrust Calculation

5.2.1 Thrust Application

Listing 3.3 shows the actual application of the previously calculated thrust. As previously mentioned, a saturation is applied to retain safe attitude angles.

```
Vector2f thrust_desired_NE;
thrust_desired_NE(0) = thr_sp(0) + _resultantObjectAvoidanceThrust(0);
thrust_desired_NE(1) = thr_sp(1) + _resultantObjectAvoidanceThrust(1);

float thrust_max_NE_tilt = fabsf(thr_sp(2)) * tanf(constraints.tilt);
float thrust_max_NE = sqrtf(THR_MAX*THR_MAX - thr_sp(2)*thr_sp(2));
thrust_max_NE = math::min(thrust_max_NE_tilt, thrust_max_NE);

thr_sp(0) = thrust_desired_NE(0);
thr_sp(1) = thrust_desired_NE(1);

// Saturate thrust in NE-direction.
if (thrust_desired_NE * thrust_desired_NE > thrust_max_NE * thrust_max_NE) {
    float mag = thrust_desired_NE.length();
    thr_sp(0) = thrust_desired_NE(0) / mag * thrust_max_NE;
    thr_sp(1) = thrust_desired_NE(1) / mag * thrust_max_NE;
}
```

Listing 3.3: Collission Avoidance Thrust Application

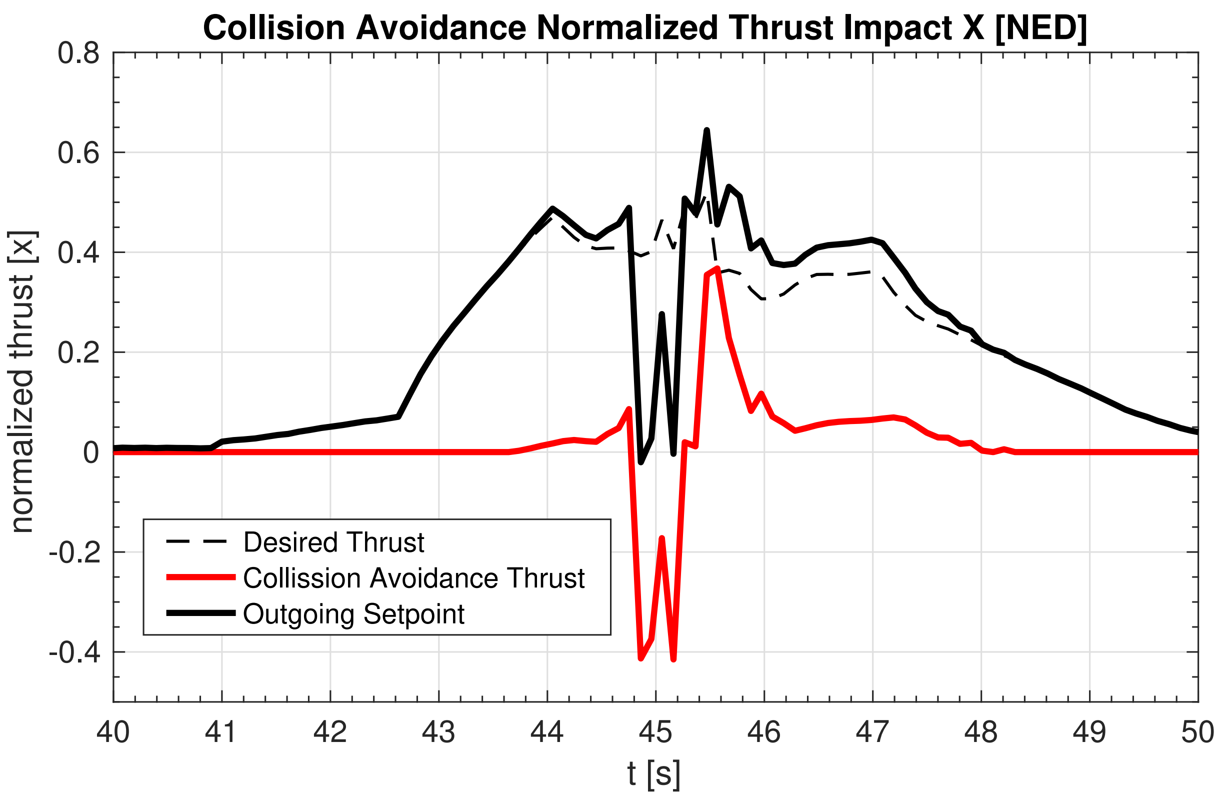
5.2.2 Logging

Since the controller has to satisfy realtime constraints, logging the values has to be performed in another thread. Inter alia for this reason, PX4 uses special uORB topics. For the newly implemented collision avoidance, a dedicated uORB topic has been added to the firmware. The topic saves inter alia the thrust magnitude and the measured distance. This allows to generate the graphs presented in the next section out of the flight logs, just in the same manner as in section 7 on page 29.

6 Simulation Results

As it can be seen from fig. 3.3 on the facing page, the collision avoidance thrust is summed up to the desired thrust coming from the position controller. Once an obstacle is detected, the impact on the xy axis is noticable and anti-proportional increasing with decreasing distance. Thanks to the pole in 0 of the weight function (see fig. 3.2 on page 40), a collision of the object, assuming a valid measurement, is almost impossible.

To obtain those test results, an auto mission has been created which piloted the drone towards a 3D house. In this case, a successful evasive maneuver could be performed, as we see from the x^E and y^E thrust graphs.



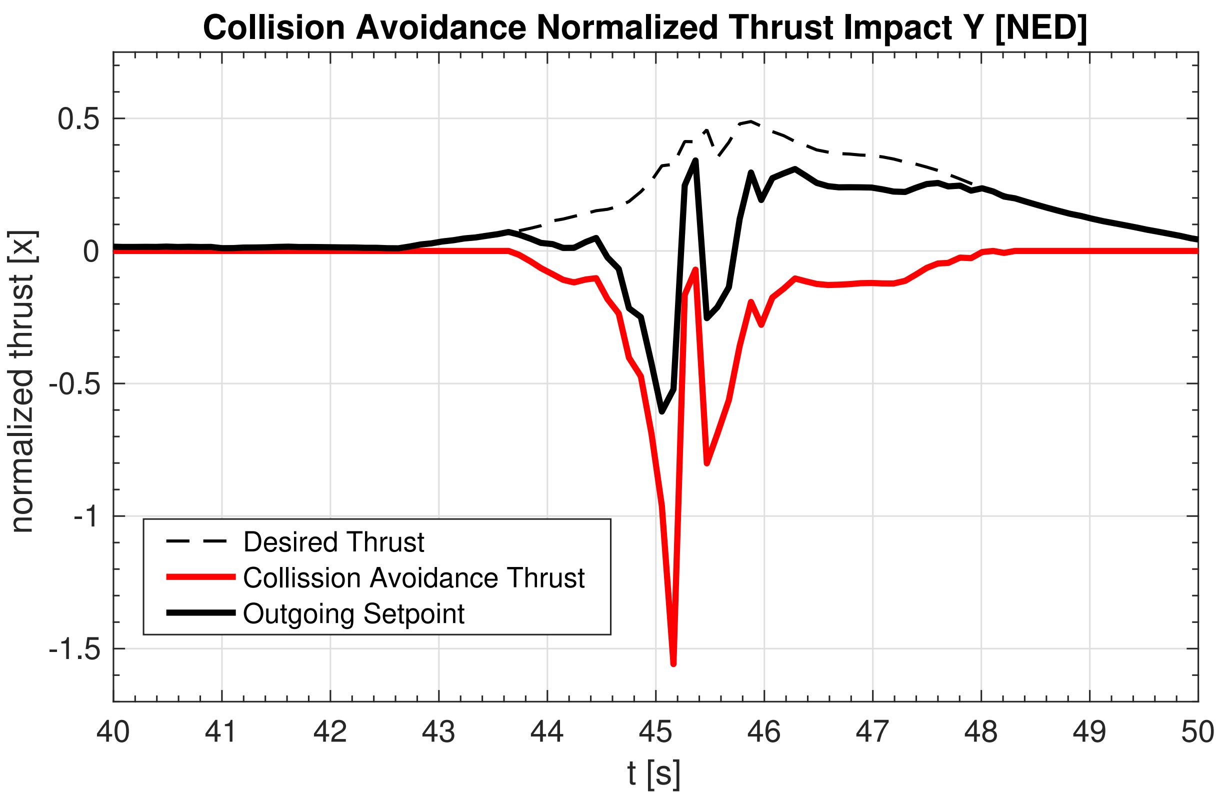
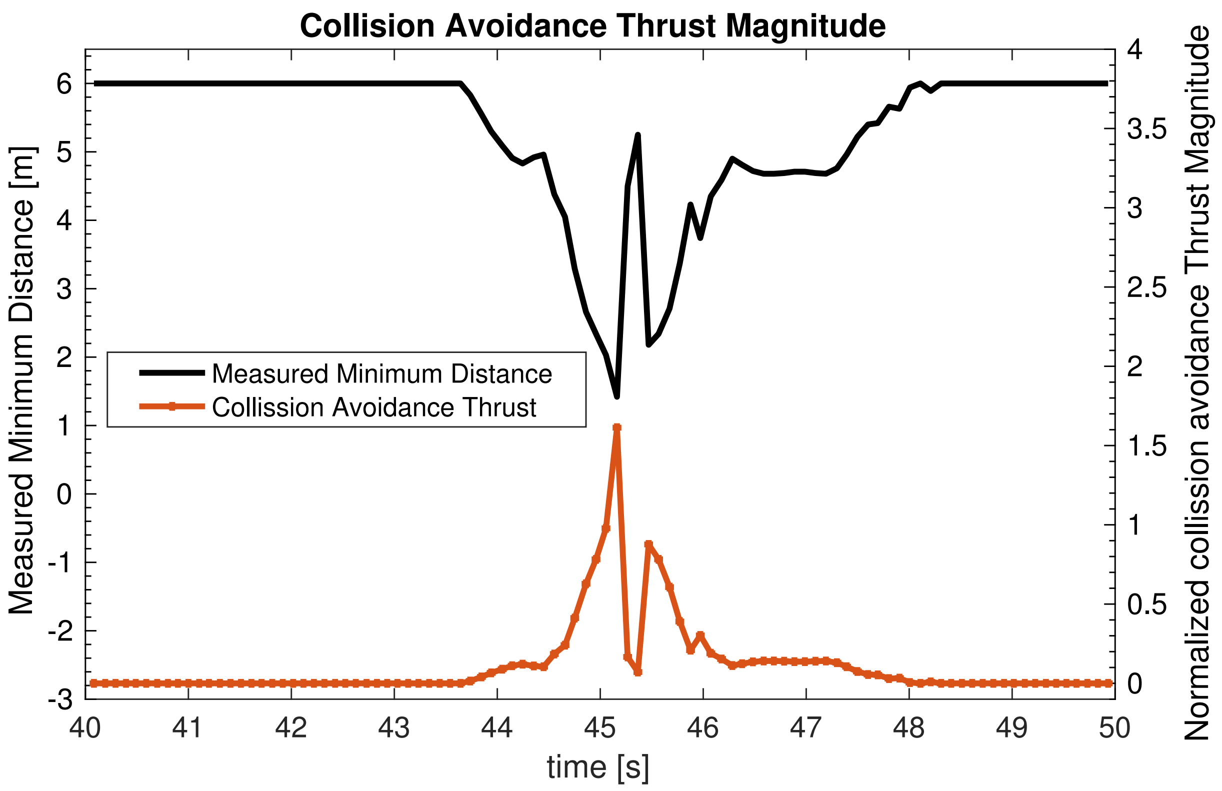
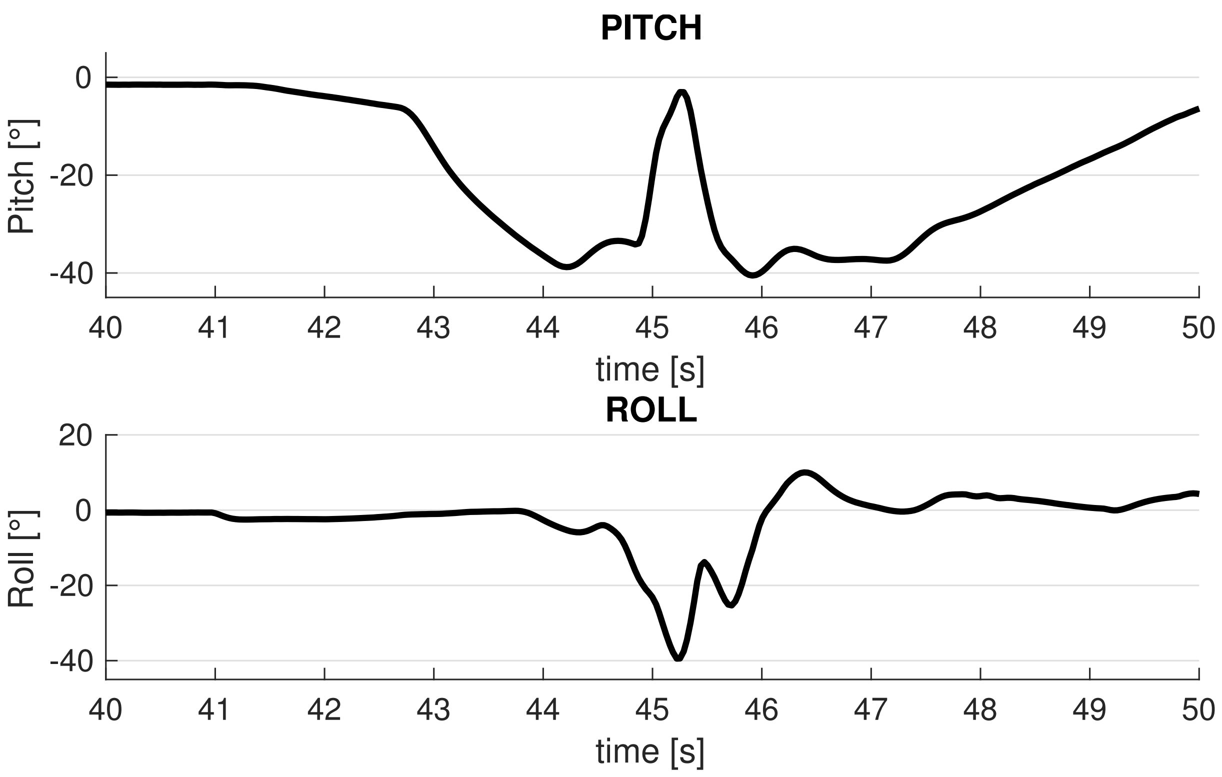


Figure 3.3: Collision Avoidance normalized Thrust Impact in XY [NED]

Figure 3.4a on the following page illustrates the mapping from the distance to the thrust *magnitude* (instead of considering x and y separately). The respective RPY behavior is given in fig. 3.4b on the next page. Even though the thrust mapping increases smoothly starting from the maximum distance D_{max}, a relative abrupt stop can be noticed following the roll angle.



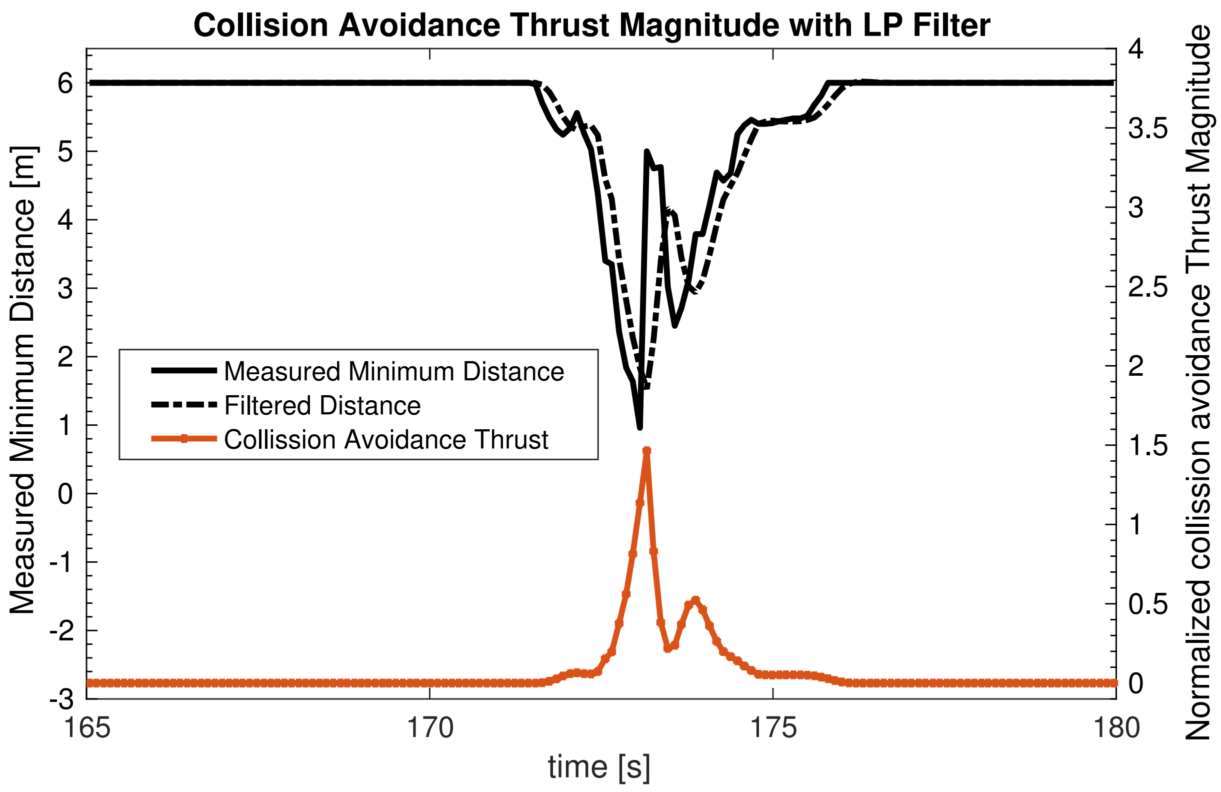
(a) Minimum Distance mapped to Thrust Magnitude



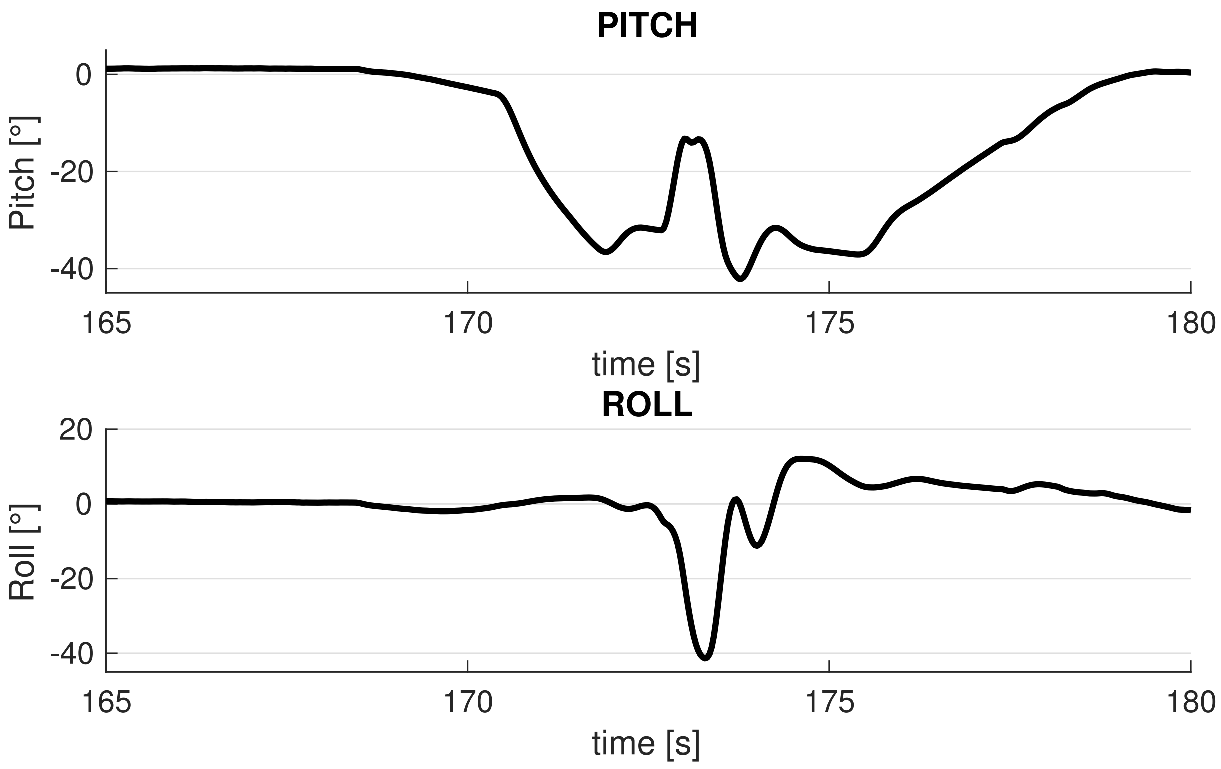
(b) Roll Pitch Behavior during Active Collision Avoidance

Figure 3.4: Collision Avoidance Behavior

Figure 3.5a shows the same graph, but with a LP filter applied.



(a) Minimum Distance mapped to Thrust Magnitude using Low-Pass Filter



(b) Roll Pitch Behavior during Active Collision Avoidance with LP filter enabled

Figure 3.5: Collision Avoidance Behavior with LP filter enabled

The LP filter is useful for reducing oscillations caused by instable measurements, but at the same time increases the delay in the reactive control action. Indeed, comparing fig. 3.5b on page 47 with the RPY behavior in the unfiltered version in fig. 3.4b on page 46, it can be observed that the oscillations slightly decreased.

In fig. 3.5a on page 47 the peaks of unfiltered distance and thrust magnitude are not perfectly aligned anymore, which is due to the introduced filter delay. Indeed the UAV gets closer to the obstacle. A deeper evaluation reveals that the minimum measured distance in the chosen maneuver is:

- 1.42 m without LP filter
- 0.96 m with LP filter

Although the introduced delay is rather small, it has a considerable impact. The used filter cut-off frequency is 100 kHz allowing anyhow a fast reaction. Given the high frequency however, also the effect of oscillation damping is almost negligible.

6.1 Qualitative Evaluation

Generally the object avoidance yielded satisfying results in the simulation, even though wrong recognitions, such as the distance to the earth ground in case of an inclined vehicle, have been observed. When the obstacle is unavoidable and presented a local minimum the multicopter successfully stopped by canceling out the position controller's desired thrust.

Complex obstacles however, such as transmission towers given in fig. 3.6 with a thin skeleton frame could not be avoided successfully. This is due to the intrinsic poor information quality provided by the 2D laserscan, i.e. just a *distance* measurement.

Anyhow, provided that the imposed trajectory roughly avoids major obstacles, the resulting behavior is satisfactory.

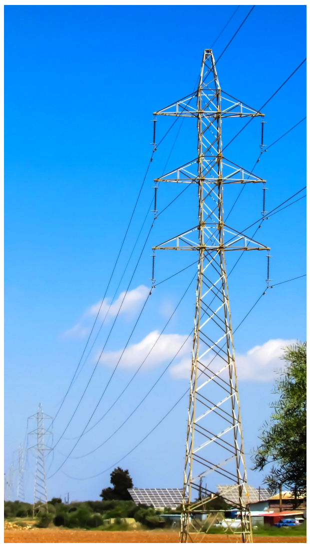


Figure 3.6: Transmission Tower

6.1.1 Advantages and Disadvantages

To summarize, the pros and cons of the proposed method shall be listed. The disadvantages include:

- The vehicle may get stuck in local minima, if the encountered obstacle's shape presents one and the position controller keeps imposing an orthogonal direction to the obstacle's surface
- No reference trajectory correction is planned, which implies that the time lost by sidestepping obstacles may compromise the complete execution of the trajectory
- The algorithm relies on reliable measurements, since no sophisticated filter e.g. for each angular segment is adopted
- The low cost hardware solution (see section 3.1 on page 38) yields a low sample rate which is unfeasible for higher speeds

Among the advantages are:

- Easy, straight-forward implementation approach
- Low latency which fully exploits the sensor's sample rate
- As a security measure, it is active in *any* possible flightmode
- No CC needed, since all is handled on the FCU, i.e. at *firmware* level

6.2 Comparison with other methods

In comparison to [8], the provided implementation has a lack of filtering options and focuses less on obstacle *detection*. As many other works, [8] treats *online* obstacle avoidance using a *goal* position, which was not feasible in the presented use case. However, the obstacle detection uses the same 2D laser scan, but its data elaboration is more sophisticated and could be improved in the proposed implementation.

[25] on the other hand, treats a very similar approach for providing a *low cost* collision avoidance solution. For this purpose, in the former publication, a combination of infrared and sonar distance sensors is used. A more sophisticated control, in PID manner and a the usage of a FSM are proposed. However, the performance with active trajectory following has not been evaluated. A deeper performance comparison is therefore rather difficult, also because the implementation is carried out on dedicated platforms, i.e. not using the famous PX4 software bundle.

To the knowledge of the author, the proposed *simple* implementation in PX4 *position controller* has not been considered so far.

Chapter IV

Experimental Results

In this chapter some practical results achieved during the Bambi Project are presented. The custom build drone is shown as a reference implementation, explaining the major drawbacks and difficulties arisen during the project work.

1 Reference Implementation

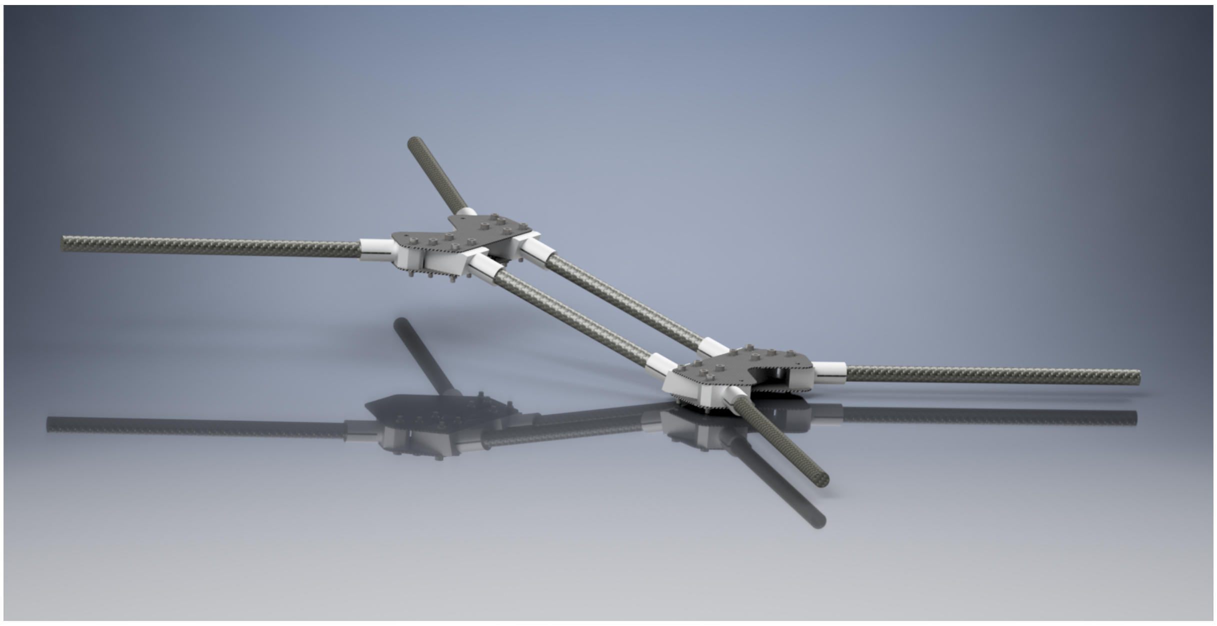
The required *minimum* equipment for the Bambi Project mission drone is:

- Ground LIDAR to measure the altitude
- 2D laser scanner
- CC (i.e. Raspberry Pi)
- external GPS
- Pixhawk
- thermal camera
- visual camera

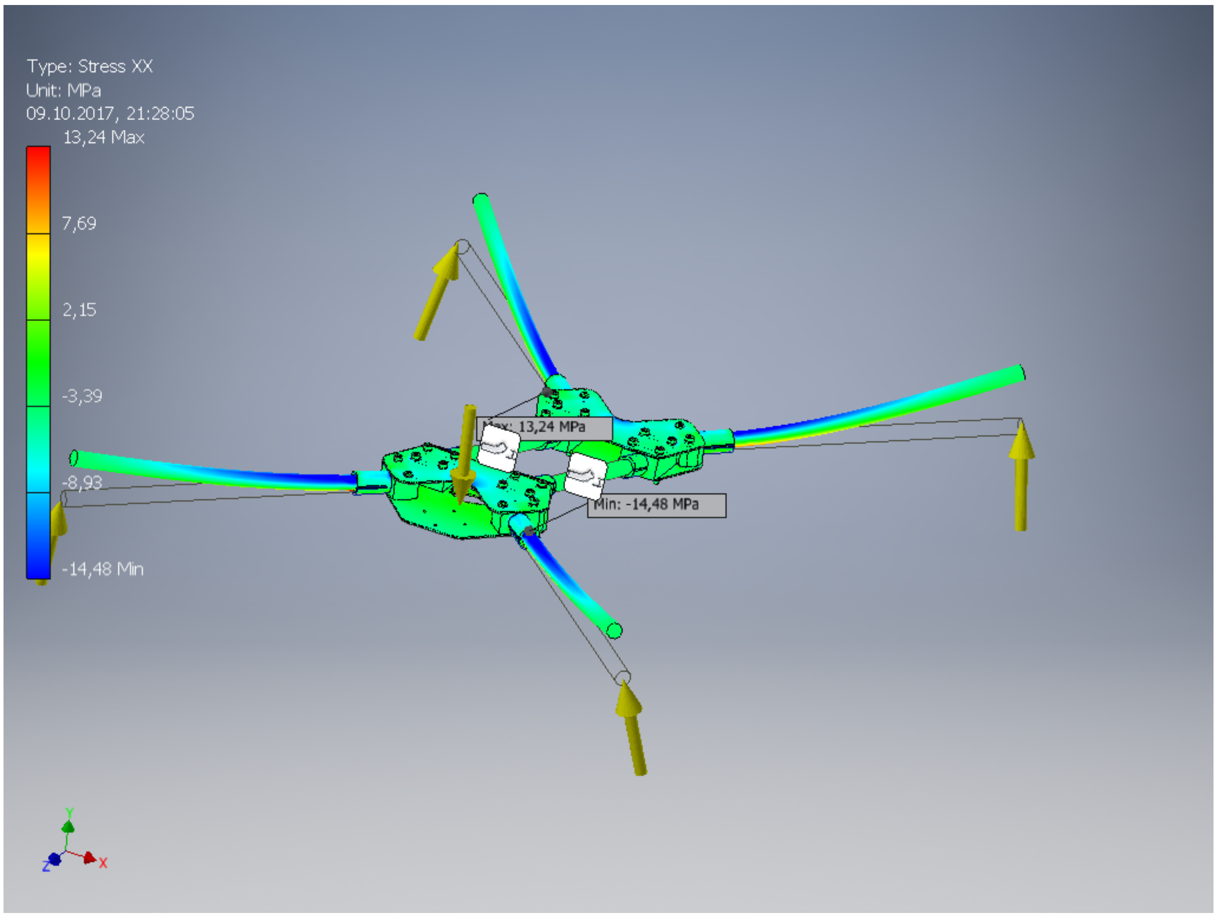
To mount those devices all together onto a standard drone frame is barely possible. This is the reason, why a custom build drone was constructed.

1.1 Drone Frame

A *foldable* drone frame has been designed building on carbon fiber tubes, mainly because of their availability on the market, limiting the need for customized pieces to a few aluminum works. Figures 4.1a and 4.1b on the following page show the 3D CAD design.



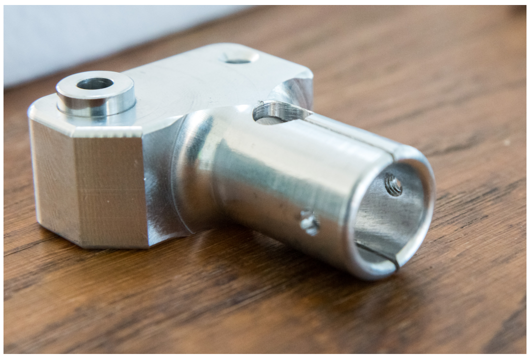
(a) 3D Model Rendering



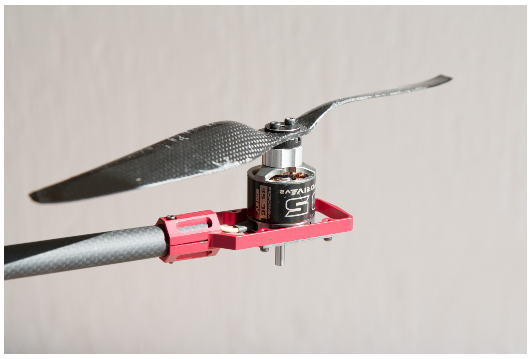
(b) Stress Analysis for weight reduction on Plates

Figure 4.1: Design Jobs carried out in Inventor

Figure 4.1 shows the design tasks carried out in Inventor.
Figures 4.2a and 4.2b on the facing page show some close-up of the physical frame. The



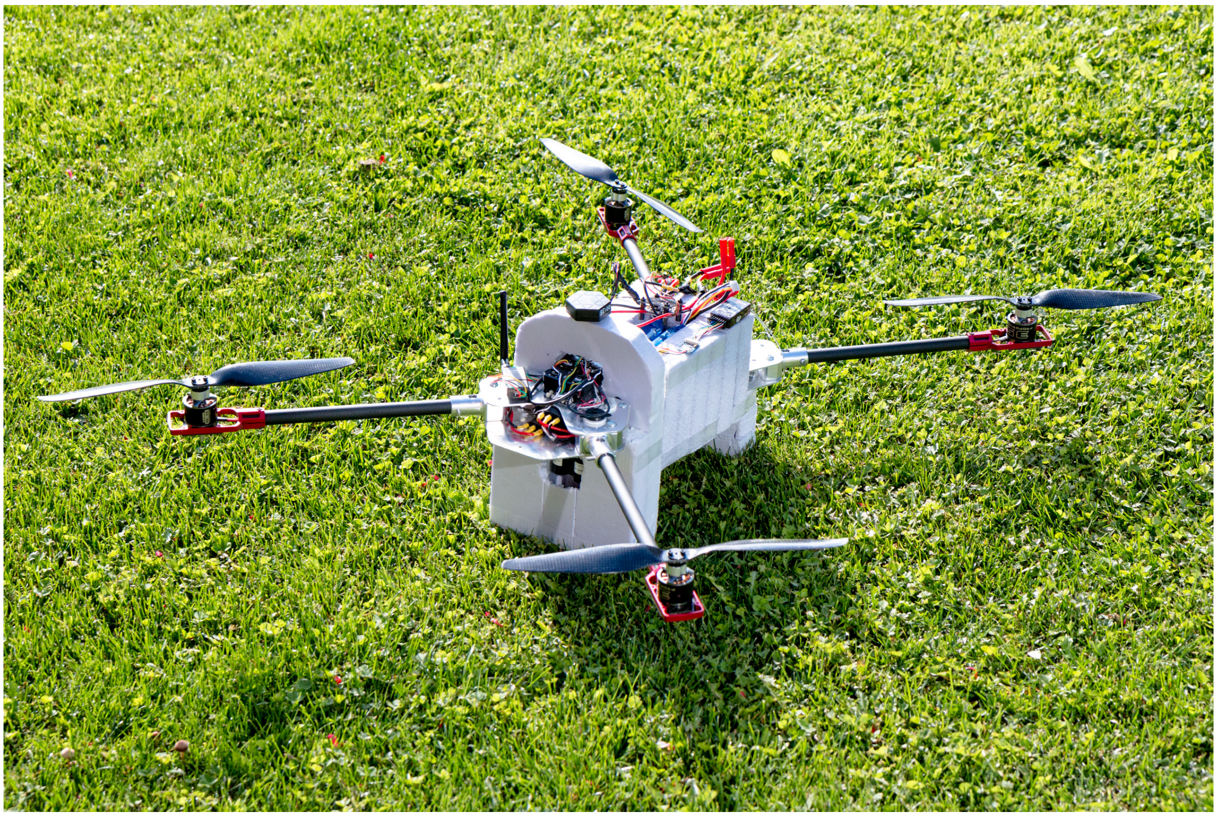
(a) Revolute Joint allowing folding functionality



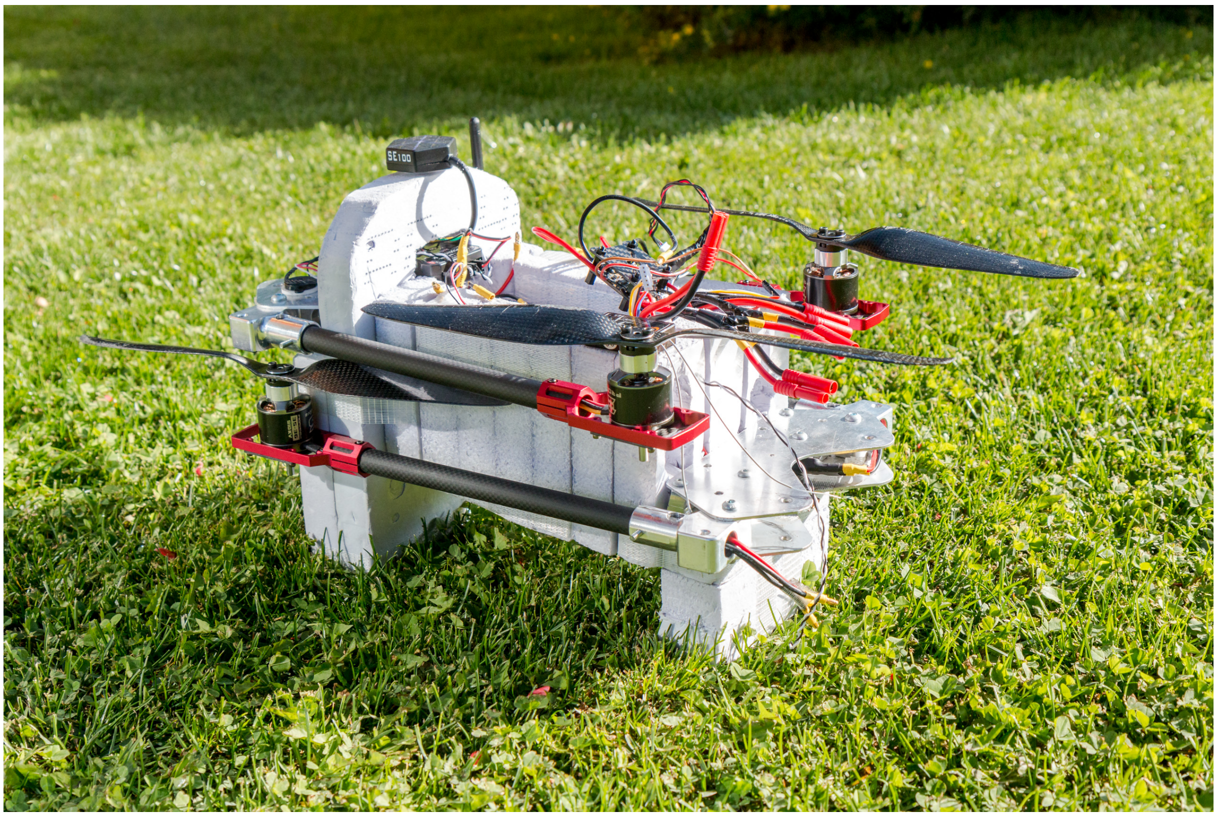
(b) Arm Close-Up

Figure 4.2: Frame Realization

aluminum tube connectors have been realized by a Computer Numerical Control (CNC) machine.



(a) Expanded

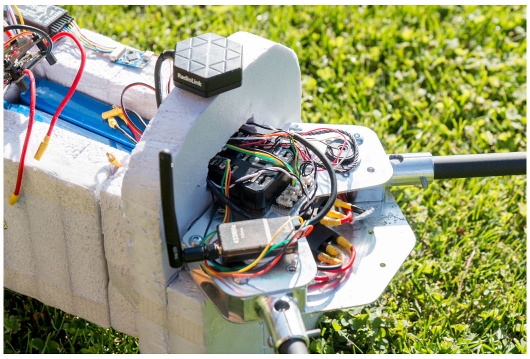


(b) Folded

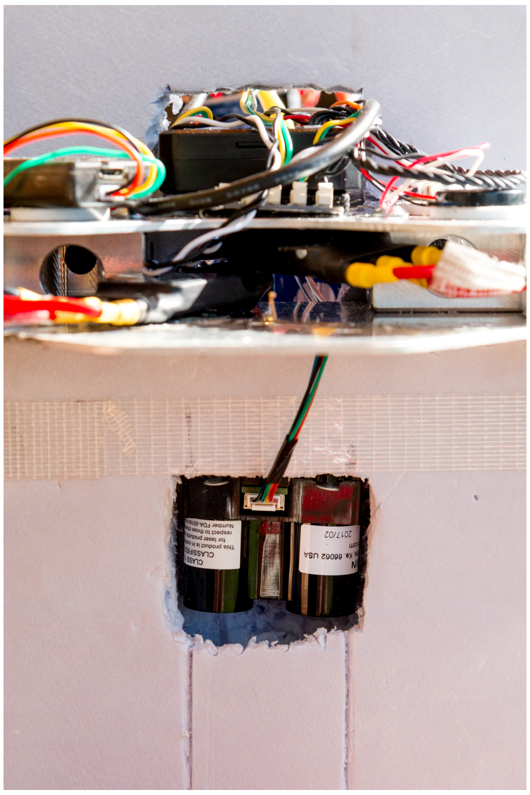
Figure 4.3: Reference Drone

1.2 Mounted Equipment

Not all of the planned equipment have been mounted yet. Figure 4.4 shows a selection.



(a) Pixhawk with external GPS



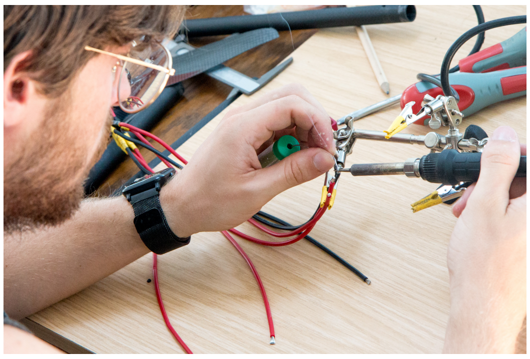
(b) Lidar for relative altitude measurements

Figure 4.4: Mounting of Components

2 Bambi Project Work

The Bambi project has been carried out in 17 intensive days. Figures 4.5a and 4.5b give some impressions.

(a) Bambi Project Work



(b) Soldering Tasks

3 Field Experiments

Various flight tests have been performed. They mainly involved the correct setup of the PID regulators used for multicopter control. Unfortunately the OFFBOARD mode could not be tested due to lack of time.

Figure 4.6 shows one of the typical agricultural fields where tests have been performed.

Figure 4.6: Test Sight on the Field

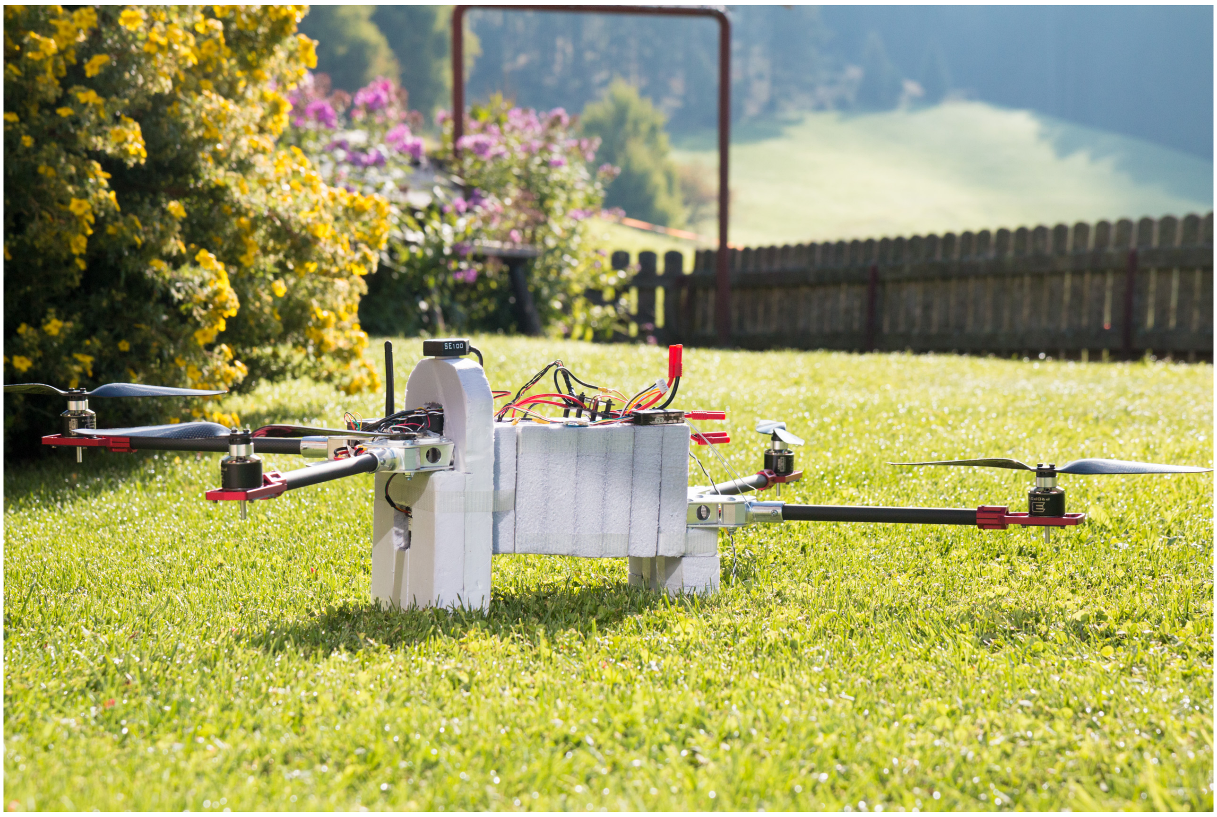


Figure 4.7: Drone Side View

The drone, almost ready to fly, from a side-view perspective, in fig. 4.7.

4 Lessons learned

The most useful practical lessons shall quickly be listed:

PID Tuning Unlike ArduPilot firmware, the new PX4 does not provide PID autotune procedure for security reasons. For custom build drones, this implies that PID tuning has to be carried out by hand which can eventually be dangerous. In the specific test case, the default PX4 parameters constituted an *unstable* system. The following may help to prevent crashes:

- use the *GeoFence* parameters to enforce Return To Land (RTL) in case of danger
- in case of danger close the outer control loop by enabling some position trajectory control (auto land, RTL, etc.) to *stabilize* the system

Electromagnetic Interference (EMI) pay attention to coupling of actuator currents and magnetic compass sensors. Increasing the distance between affected components and eventually twisting the cables should be effective countermeasures.

Raspberry Pi Build times, especially of MAVROS may exceed practical limits (2 h to 3 h). A cross compile setup can be very useful.

Chapter V

Conclusion and Future Work

This work provided an implementation approach, for controlling the position of an UAV in an elegant way from a *Companion Computer*. The currently most used systems such as ROS, PX4 and MAVLINK, have been exploited to *improve* the tracking performance and provide a low-cost collision avoidance solution.

All the Bambi Project is available as open source software, hosted at github under `https://github.com/BambiSaver`. Some documentation is published on the special purpose media-wiki under `https://wiki.bambi.florian.world`.

1 Future Work

There are a lot of improvements for multicopter UAVs, in the agricultural field use case, to be addressed:

1. Flight Time – Battery Performance
2. Sophisticated Collision Avoidance
3. Trajectory tracking performance enhancements

1.1 Trajectory Tracking

Although the provided solution of velocity feed forward control improved the overall tracking performance, it presented overshoots of several meters, which is clearly unacceptable on the field. Implementing also an acceleration feed forward path may improve the situation, together with more sophisticated trajectory planning algorithms.

In particular the application of optimal control theory has been evaluated, but could not be implemented due to lack of time. It was foreseen to use the optimal control framework proposed by Nicola dal Bianco, available at `https://github.com/stavoltafunzia/Maverick`. In the related PhD thesis lap time optimization using optimal control theory was implemented, see [26]. The problem can be modeled in an analogous way.

1.2 Collision Avoidance

One possible approach is to combine the currently used stereo-cameras for collision avoidance with the low-level control action presented in this work. The common approach to make online trajectory modifications may not always be suitable, as it was the case in the presented flight mission. However, stereo vision systems could avoid the major drawbacks encountered with a 2D laser scan, by improving obstacle *detection*.

A more straight forward improvement, perhaps avoiding unnecessary control actions, could be, to integrate the component v_0^2 from eq. (3.9) on page 39 into the thrust impact.

The proposed approach however, *must* be evaluated on an experimental system, in order to be able to correctly judge its feasibility.

Appendix A

References

[1] T. Puls, M. Kemper, R. Küke, and A. Hein, "Gps-based position control and waypoint navigation system for quadrocopters," in *2009 IEEE/RSJ International Conference on Intelligent Robots and Systems*, Oct. 2009, pp. 3374–3379. DOI: 10.1109/IROS.2009.5354646.

[2] A. Roberts and A. Tayebi, "Adaptive position tracking of vtol uavs," *IEEE Transactions on Robotics*, vol. 27, no. 1, pp. 129–142, Feb. 2011, ISSN: 1552-3098. DOI: 10.1109/TRO.2010.2092870.

[3] I. C. for Game and W. Conservation, "Mowing mortality in grassland ecosystems," pp. 4–10, 2013. [Online]. Available: http://www.cic-wildlife.org/wp-content/uploads/2013/04/Mowing_guide_EN.pdf.

[4] K. S. Christie, S. L. Gilbert, C. L. Brown, M. Hatfield, and L. Hanson, "Unmanned aircraft systems in wildlife research: Current and future applications of a transformative technology," *Frontiers in Ecology and the Environment*, vol. 14, no. 5, pp. 241–251, 2016. DOI: 10.1002/fee.1281. eprint: https://esajournals.onlinelibrary.wiley.com/doi/pdf/10.1002/fee.1281. [Online]. Available: https://esajournals.onlinelibrary.wiley.com/doi/abs/10.1002/fee.1281.

[5] Y.-G. Han, S. H. Yoo, and O. Kwon, "Possibility of applying unmanned aerial vehicle (uav) and mapping software for the monitoring of waterbirds and their habitats," *Journal of Ecology and Environment*, vol. 41, no. 1, p. 21, May 2017, ISSN: 2288-1220. DOI: 10.1186/s41610-017-0040-5. [Online]. Available: https://doi.org/10.1186/s41610-017-0040-5.

[6] J. G. A. Barbedo and L. V. Koenigkan, "Perspectives on the use of unmanned aerial systems to monitor cattle," *Outlook on Agriculture*, p. 0 030 727 018 781 876, 2018.

[7] V. Pirotta, A. Smith, M. Ostrowski, D. Russell, I. D. Jonsen, A. Grech, and R. Harcourt, "An economical custom-built drone for assessing whale health," *Frontiers in Marine Science*, vol. 4, p. 425, 2017, ISSN: 2296-7745. DOI: 10.3389/fmars.2017.00425. [Online]. Available: https://www.frontiersin.org/article/10.3389/fmars.2017.00425.

[8] Y. Peng, D. Qu, Y. Zhong, S. Xie, J. Luo, and J. Gu, "The obstacle detection and obstacle avoidance algorithm based on 2-d lidar," in *2015 IEEE International Conference on Information and Automation*, Aug. 2015, pp. 1648–1653. DOI: 10.1109/ICInfA.2015.7279550.

[9] N. Pokhrel, "Drone obstacle avoidance and navigation using artificial intelligence," en, G2 Pro gradu, diplomityö, 2018-05-14, pp. 95 + 7. [Online]. Available: `http://urn.fi/URN:NBN:fi:aalto-201806012988`.

[10] J. A. S. Jayasinghe and A. M. B. G. D. A. Athauda, "Smooth trajectory generation algorithm for an unmanned aerial vehicle (uav) under dynamic constraints: Using a quadratic bezier curve for collision avoidance," in *2016 Manufacturing Industrial Engineering Symposium (MIES)*, Oct. 2016, pp. 1–6. DOI: `10.1109/MIES.2016.7780258`.

[11] D. Mellinger and V. Kumar, "Minimum snap trajectory generation and control for quadrotors," in *2011 IEEE International Conference on Robotics and Automation*, May 2011, pp. 2520–2525. DOI: `10.1109/ICRA.2011.5980409`.

[12] M. I. Ribeiro, "Obstacle avoidance," *Instituto de Sistemas e Robótica, Instituto Superio Técnico*, p. 1, 2005. [Online]. Available: `http://users.isr.ist.utl.pt/~mir/pub/ObstacleAvoidance.pdf`.

[13] D. Jie, M. Xueming, and P. Kaixiang, "Real-time dynamic obstacle avoidance for mobile robots," in *2010 11th International Conference on Control Automation Robotics Vision*, Dec. 2010, pp. 844–847. DOI: `10.1109/ICARCV.2010.5707283`.

[14] J. Kümmerle, T. Hinzmann, A. S. Vempati, and R. Siegwart, "Real-time detection and tracking of multiple humans from high bird's-eye views in the visual and infrared spectrum," in *Advances in Visual Computing*, G. Bebis, R. Boyle, B. Parvin, D. Koracin, F. Porikli, S. Skaff, A. Entezari, J. Min, D. Iwai, A. Sadagic, C. Scheidegger, and T. Isenberg, Eds., Cham: Springer International Publishing, 2016, pp. 545–556, ISBN: 978-3-319-50835-1.

[15] M. Quigley, K. Conley, B. P. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Y. Ng, "Ros: An open-source robot operating system," in *ICRA Workshop on Open Source Software*, 2009.

[16] J. M. O'Kane, *A Gentle Introduction to ROS*. Independently published, Oct. 2013, Available at `http://www.cse.sc.edu/~jokane/agitr/`, ISBN: 978-1492143239.

[17] M. P. Ananda, H. Bernstein, K. E. Cunningham, W. A. Feess, and E. G. Stroud, "Global positioning system (gps) autonomous navigation," in *IEEE Symposium on Position Location and Navigation. A Decade of Excellence in the Navigation Sciences*, Mar. 1990, pp. 497–508. DOI: `10.1109/PLANS.1990.66220`.

[18] J. W. Hager, J. F. Behensky, and B. W. Drew, "The universal grids: Universal transverse mercator (utm) and universal polar stereographic (ups). edition 1," DEFENSE MAPPING AGENCY HYDROGRAPHIC/TOPOGRAPHIC CENTER WASHINGTON DC, Tech. Rep., 1989.

[19] U. Breymann, *Der C++-Programmierer: C++ lernen - professionell anwenden - Lösungen nutzen*. Hanser, 2011, ISBN: 9783446428416. [Online]. Available: `https://books.google.it/books?id=x%5C_J4uAAACAAJ`.

[20] F. N. Fritsch and R. E. Carlson, "Monotone piecewise cubic interpolation," *SIAM Journal on Numerical Analysis*, vol. 17, no. 2, pp. 238–246, 1980.

[21] H. W. Smith and E. J. Davison, "Design of industrial regulators. integral feedback and feedforward control," *Proceedings of the Institution of Electrical Engineers*, vol. 119, no. 8, pp. 1210–1216, Aug. 1972, ISSN: 0020-3270. DOI: `10.1049/piee.1972.0233`.

[22] K. Ahnert and M. Abel, "Numerical differentiation of experimental data: Local versus global methods," *Computer Physics Communications*, vol. 177, no. 10, pp. 764–774, 2007, ISSN: 0010-4655. DOI: https://doi.org/10.1016/j.cpc.2007.03.009. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0010465507003116.

[23] A. Cardamone, "Implementation of a pilot in the loop simulation environment for uav development and testing," 2017. [Online]. Available: https://www.politesi.polimi.it/bitstream/10589/135202/3/2017_07_Cardamone.pdf.

[24] D. Brescianini, M. Hehn, and R. D'Andrea, "Nonlinear quadrocopter attitude control. technical report," en, Tech. Rep., 2013. DOI: 10.3929/ethz-a-009970340.

[25] N. Gageik, P. Benz, and S. Montenegro, "Obstacle detection and collision avoidance for a uav with complementary low-cost sensors," *IEEE Access*, vol. 3, pp. 599–609, 2015, ISSN: 2169-3536. DOI: 10.1109/ACCESS.2015.2432455.

[26] N. D. Bianco, R. Lot, and M. Gadola, "Minimum time optimal control simulation of a gp2 race car," *Proceedings of the Institution of Mechanical Engineers, Part D: Journal of Automobile Engineering*, vol. 232, no. 9, pp. 1180–1195, 2018. DOI: 10.1177/0954407017728158. eprint: https://doi.org/10.1177/0954407017728158. [Online]. Available: https://doi.org/10.1177/0954407017728158.

Appendix B

Supplementary Information

1 List of Figures

2 List of Tables

3 List of Listings

4 List of Algorithms

Appendix C

Glossary

Bambi Project is a project involing the usage of an UAV for saving animals in mountainous grassland agriculture fields by using a thermal camera to detect their body heat. 1, 2, 10, 15, 16, 19, 20, 30, 37, 51, 59

C++ is a general purpose programming language with inter alia object-oriented features. 20, 28

Companion Computer is an auxiliary on-board computing device, which communicates with the FCU to usually handle high-level flight tasks such as computer vision. vii, 6, 7

Flight Control Unit is usually a microcontroller which performs the main flight control tasks, such as stabilizing the vehicles attitude or holding the position, by running a *flight controller software* such as PX4. iii, v, vii, 5, 7

I/O Device is a device which exchanges data with the Central Processing Unit (CPU) through an I/O port, e.g. a hard disk drive. vii, 68

I/O Port is an electronic component integrated in the CPU bus system which enables connected devices to exchange data with the CPU, e.g. a serial port. vii, 67, 68

Microcontroller is a small computer on a single integrated circuit. 67

Multicopter or multirotor is a rotor-wing aircraft with more than two rotors, providing the advantage of *simpler rotor mechanics* (e.g. typically no swashplate needed). iii, 8, 19, 35, 38, 48, 56

Open Source Software is free software distributed under a license which grants access to the source code. 30, 59

Pixhawk is an open source hardware platform developed for the PX4 project. 7, 9, 10, 19, 51

PX4 is *the professional autopilot*, a widely used flight controller software inter alia for multi copters. 5–8, 10, 13, 19, 20, 22, 23, 27, 29–31, 35, 37, 38, 40–42, 44, 49, 57, 59, 67

Raspberry Pi is a open-source single-board computer. 7, 8, 10, 51, 57

Robot Operating System is a development environment which provides libraries and tools for creating robot applications. iii, v, vii, 6

Universal Asynchronous Receiver-Transmitter is a serial Input / Output (I/O) communication port with configurable transmission speeds and data formats.. viii, 9

Universal Transverse Mercator is a coordinate system which maps global positions to a defined set of local 2-dimensional Cartesian reference frames by *transverse mercator projection*. viii, 13

World Geodetic System is a standard in cartography used for global navigation (also in GPS) which approximates the globe through a *reference ellipsoid* in different versions: the latest is WGS 84. viii, 10, 13