mng rämibühl
Mathematisch-Naturwissenschaftliches Gymnasium

Fly by wire for model airplanes

Jonathan Hungerbühler

Maturitätsarbeit
im Fach Informatik

Betreuende Lehrperson:
Lukas Fässler

7. Januar 2020

Fly by wire for model airplanes

Jonathan Hungerbühler

Maturitätsarbeit
im Fach Informatik

Betreuende Lehrperson:
Lukas Fässler

7. Januar 2020

Comments on the title page

The official logo of the Mathematisch-Naturwissenschaftliches Gymnasium Rämibühl adorns the head of the title page. It is available on the website www.mng.ch as svg file (Scalable Vector Graphics) and can be scaled without any loss of quality. Background and font are in classical sepia (background RGB BinHex FBF0D9, CMYK 0/4/14/2, font RGB BinHex 5F4B32, CMYK 0/21/47/63).

The bastard title picks up the logo again. To create a reference to the work, I underlaid the logo with a casual brush stroke in Zürich blue (RGB BinHex 0066CC, CMYK 100/44/0/0). This reminds of the color of the sky, in which the airplanes cavort.

ABSTRACT

The intention of this project was to create a functioning fly by wire system (FBW system) with a dedicated aircraft. To achieve this all relevant physical aspects regarding rigid body movement and fluid dynamics where considered. Said information was then used to construct a model of a fixed-wing aircraft. The resulting airframe was then fitted with the appropriate electrical systems to serve as a testing platform for the proposed flight controller and its associated algorithms. These form the main part of the project, ranging from telemetry handling, data acquisition and extended linear quadric state estimation to solution approaches for optimal control problems via closed loop policies. This allowed the implementation of a two axis attitude tracking servo as well as a yaw damping mechanism that fully manages the rudder during flight manoeuvres. After the model aircraft was equipped with the FBW system, numerous flight tests were made to determine the tuning parameters. The most successful flight extended over about 200 meters.

CONTENTS

1 Introduction

1.1 Problem description

The goal of this project was the creation of a working model aircraft and its associated control systems. One side involved the physical aspects of designing a flying object. These are required to build a working prototype and finally dictate the flying characteristic of the proposed aircraft. Secondly the plane was fitted with actuators and sensors, allowing for the implementation of a computer-assisted *fly-by-wire* (FBW) system. This system made headlines after the ditching of US Airways Flight 1549 on the Hudson river on January 15, 2009. Airbus A320-214 with 155 people on board was equipped with a state of the art FBW system. During its initial climb from LaGuardia Airport, New York, the airplane hit a flock of Canada Geese, resulting in loss of engine power. Captain Chesley Sullenberger managed, thanks to the FBW system, to safely make an emergency landing on the Hudson river in the middle of New Cork City (see Figure 1.1). Soon after, the incident became known as the "Miracle on the Hudson". The incident and the role of the FBW system has been reported and analysed in [14]).

Figure 1.1: The "Miracle on the Hudson".

Systems like this replace the manual controls of an aircraft by connecting the mechanical systems to a digital controller. This allows a pilot to command an aircraft without having to directly interface with every individual component, resulting in greater ease-of-use, safety and setting the base for future fully autonomous flight.

1.2 Notions and notations

In this section we introduce some of the main notions which we use later with a brief description. The precise definitions, if needed, will be given in the following chapters.

We will use the standard abbreviation UAV for *unmanned aerial vehicle* or *drone*. The mechanical structure of an aircraft is called *airframe*. The main body section of the aircraft is referred to as *Fuselage*, carrying the *payload*, which is the carrying capacity of an aircraft. In this work we will consider only a fixed-wing design, i.e., a flying machine with fixed wings (airfoils), in contrast to a rotary-wing aircraft such as a helicopter.

A *Fly-By-Wire* system describes an electrical system, connecting the aircraft's actuators via a *flight controller* to the pilot's controls. The pilot may control an airplane via changing its *attitude* that consists out of *roll*, *pitch*pitch and *yaw*. These describe the angle between the wings and the horizon in the plane perpendicular to the flight direction, angle of the flight direction relative to the horizon line and the orientation in earth's horizontal plane respectively.

The *lift* is the component of the resulting force acting on a wing which is perpendicular to the direction of the flight and opposite to the force of gravity. The component of the force parallel to the direction of the flight is called *drag force* or *air resistance*. This force is overcome by the propeller propulsion. The *angle of attack* AoA is the angle between the main reference line of the airplane and the incoming air flow. The *Reynolds number* Re can be understood as the ratio of inertia forces to viscosity forces in a given fluid. The *Mach number* is the dimensionless ratio between the relative velocity of a body in a fluid and the speed of sound in that fluid. The *wing loading* is the ratio between the weight of an aircraft and the area of its wing.

The *wing chord* refers to the imaginary line connecting the nose of the airfoil and the trailing edge of the airfoil.

An *Analog to Digital Converter (ADC)* is a device that maps an electrical voltage range to a range of numbers.

A *microcontroller* is a System on a Chip (SoC) that consists out of a processing unit and memory directly executing the given source code. This is similar to a *single board computer* that refers to a computer with all required components such as memory, interfaces and a power supply, consisting out of one single *PCB*. *PCB* stands for Printed Circuit Board and is a copper plated fibreglass plate used to connect individual electronic components.

A *task manager* is a program or circuit that is responsible for managing the execution of software.

Telemetry refers to transmission of measurement data and commands over a remote data link (usually wireless). The *downlink* describes the dataflow from a remote location towards a centre and the *uplink* the opposite direction away from the central unit. The transmission method of said data is called *message transport.*The transported data is encoded into *data packets* that are organised by the software layer. The hardware layer divides the payload further into *frames* that are fundamental data units on a physical level. This can be done over a *serial data bus* that transmits data in series rater than in *parallel*. The payload may be protected against transmission errors by using *parity*, forming to a checksum. The *baud rate* describes the transmission speed of a data link. *Chip-to-chip* communication is defined as the data exchange between automated systems.

An *AHRS (Attitude Heading Reference System)* is a system that tracks the angular orientation of an object in 3D-space. *Euler angles* are the orientation in 3D-space consisting out of three values, forming a touple that describes the rotation around each individual axis. Rotations can also be represented by *quaternions* that consist out of a real part, forming a rotation axis in 3D-space and an imaginary scalar encoding the rotation angle around said axis.

State space is a mathematical model, where all states of a system can be represented as a vector inside a space, that contains all possible system states. A *state transfer function* transforms said states inside the space according to the *system dynamics*.

A *closed loop* system is a self regulating system that consists out of a policy that gets direct state feedback from the plant to control.

A *Jacobian* matrix generalizes the derivative of a real valued function of one variable. If $f : \mathbb{R}^n \to \mathbb{R}^m$, then the Jacobian matrix in the point $x \in \mathbb{R}^n$ is given by

$$J_f(x) = \left(\frac{\partial f_i(x)}{\partial x_j} \right) \in \mathbb{R}^{m \times n}.$$

The linearization of f in a point x_0 is then

$$f(x) = f(x_0) + J_f(x_0)(x - x_0).$$

2 Mechanics and fluid dynamics

The mechanical design of the UAV was centred around maximising the vehicle's efficiency to increase possible real-world applications. The airframe had to be modelled around a large payload carrying capacity and should offer good fuel efficiency.

The set design criteria are best met by a fixed-wing design, which creates lift through immovable airfoils. This also comes with the added benefit of greater stability compared to rotary wings, further increasing system reliability.

2.1 Lift

The majority of an aircraft's lift is generated by the main airfoil (see [9]): The wings generate an upward force from the airstream they experience during flight (see Figure 2.1). The

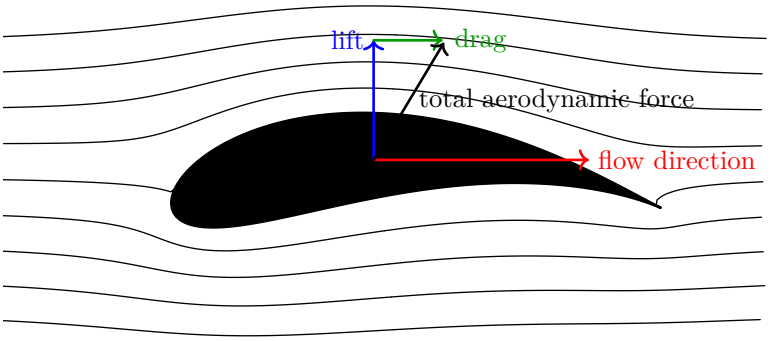

Figure 2.1: Joukowski profile in the complex plane with laminar flow lines. See [11] and [16].

phenomenon which applies here is Bernoulli's principle which states that the increase in the speed of a fluid on the upper side of the wing which is forced by the geometry of the profile leads to a decrease in static pressure.

Every airfoil has a specific lift coefficient C_L which is dependant on the wing profile, the angle of attack towards to the oncoming airstream, the Reynolds number and its Mach number, of which the latter can be ignored, as the proposed design is considered only to fly well under the speed of sound in air. The Reynolds number describes the ratio of a fluids internal forces F_I to its viscosity force F_V. More precisely, let v be the fluids relative velocity in m/s with respect to the moving object, l a characteristic linear dimension of the object in meters (typically its length), and let ν denote the fluids kinematic viscosity in m^2/s, then

the Reynolds number is defined by

$$\mathrm{Re} = \frac{F_I}{F_V} = \frac{v\,l}{\nu}. \tag{2.1}$$

Experiments show that the lift force L of a body moving in a fluid is proportional to the relevant surface A (e.g. the area of the wing), the density ρ of the fluid and the square of the velocity v. The proportionality coefficient is called *lift coefficient*. Hence, the lift coefficient C_L is a dimensionless number given by

$$C_L = \frac{2L}{\rho\, v^2 A}. \tag{2.2}$$

The numerical value of C_L can therefore be determined by experimentally measuring L on the right hand side of (2.2) for *one* arbitrary velocity v, assuming that ρ and A are known. Once we have C_L we can solve (2.2) for the lift force L and find

$$L = \frac{1}{2} C_L \rho\, v^2 A. \tag{2.3}$$

2.2 Wing Loading

Another important parameter of airfoil design is the wing loading W_L (see [25] or [8]), which is the quotient of the aircraft's mass m and the wing area A:

$$W_L = \frac{m}{A}. \tag{2.4}$$

Not only does a wing have to be designed to withstand a certain load to allow for flight with a desired payload but it is also one of the main factors that influence the aircraft's stall speed which we consider next.

2.3 Angle of Attack and Stall

The lift coefficient of a wing changes with the angle to the oncoming airstream (the angle of attack AoA), hence influencing the generated lift. This combined with engine power are the main ways of a fixed-wing aircraft to control its climb rate and altitude. The AoA over the main wing is controlled using the elevator at the tail, forcing air up, hence tail down, and therefore levering the nose upwards. Experimentally, one finds that the effective lift coefficient C_L^{eff} depends approximately affinely linear on the AoA α, namely

$$C_L^{\mathrm{eff}} \simeq C_L^{\mathrm{ini}}\Big(1 - \frac{\alpha}{\hat{\alpha}}\Big). \tag{2.5}$$

Here, C_L^{ini} denotes the lift coefficient for horizontal flight, i.e., $\alpha = 0$. So, every wing has a characteristic so called stall angle $\hat{\alpha}$ at which the lift vanishes. At this angle a flow separation at the tip of the wing occurs, resulting in turbulence directly over the wing surface. This results in a dramatic lift reduction and requires the pilot to recover by immediately reducing the AoA. The stall angle is specific to each airfoil design and can be determined experimentally. To do this a large amount of strings is attached directly to the wing surface. If rotated in an airstream the onset of turbulence is made visible by the strings not laying flat to the surface but being distorted by the turbulence. It is important to determine the stall angle of the aircraft to later constrain the FBW-system to only allow safe angles. Plane

stalling can also be expressed as a function of airspeed. This links the lift L_0 needed to maintain altitude at a given speed v to the AoA α providing said lift. Utilising (2.5) in the lift equation (2.3) and the critical angle $\hat{\alpha}$, the stall speed $\hat{v}$ can be expressed as follows:

$$\hat{v} \simeq \sqrt{\frac{2L_0}{\rho A C_L^{\text{ini}}(1-\frac{\alpha}{\hat{\alpha}})}}\,. \tag{2.6}$$

A safe flight at a given angle α is therefore only possible for speed $v \geq \hat{v}$. The stall speed is mainly useful when determining the maximum payload for cruising and restricting the throttle control to safe limits. For more information we refer to [26] and [6].

2.4 Wing Sweep

The amount the wing tips are angled in reference to their root is referred to as wing sweep (see Figure 2.2). As we will explain below, the wing sweep plays a decisive role for maintaining sufficient lift during flight (see, e.g., [7], [18] and [24]).

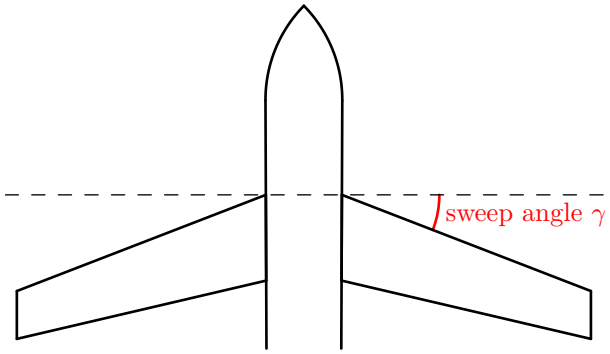

Figure 2.2: Wing sweep.

Even if an airfoil is only subjected to a subsonic airstream the area near its front is still capable of creating a localised flow that can exceed subsonic speeds. This area of supersonic flow then creates shockwaves, detaching the upper airflow from the wing surface, which results in dramatically increased drag as well as reduced lift. This famous problem was first observed on the *P-38 Lightning*, engineered by *Lockheed* during the Second World War. The sudden lift reduction that occurred with certain airspeeds combined with the placement of the horizontal stabiliser caused the plane to enter a nearly unrecoverable nosedive (see Figure 2.3).

On an aircraft with no wing sweep the airstream flows fully parallel to the wing chord. This velocity component is referred to as chord wise flow and is the one accelerated through the airfoil, reducing the critical mach number. The flow over a swept wing contains another span wise component that follows the wing tip and is therefore not subject to the acceleration caused by the wing profile. This reduces the cord-wise flow, as the overall flow is expressed as a vector V consisting of the span-wise flow S and the chord-wise flow C (see Figure 2.4).

$$V = \begin{pmatrix} S \\ C \end{pmatrix}, \qquad |V| = \sqrt{S^2 + C^2} = |C|\sec\gamma\,. \tag{2.7}$$

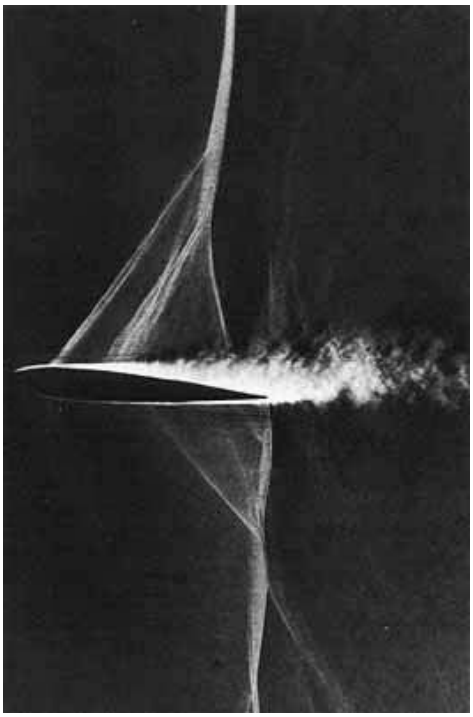

Figure 2.3: Schlieren photograph of transonic flow over an airfoil. The nearly vertical shock wave is followed by boundary layer separation that adversly affects lift, drag, an other flight parameters (from [2]).

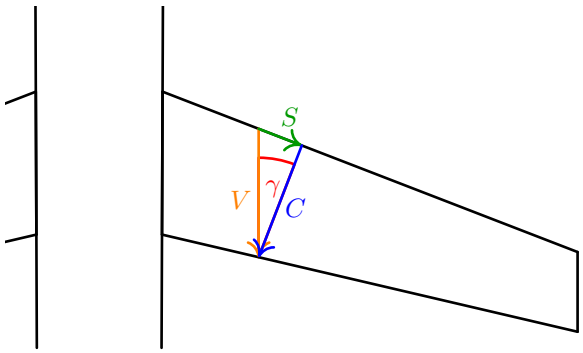

Figure 2.4: Decomposition of the flow velocity.

Assuming the airspeed $|V|$ is the same, a wing with a larger span-wise flow S leads to a reduction of the cord-wise component. This results in an aircraft that is capable of reaching higher velocities $|V|$ relative to the cord-wise flow, effectively delaying the occurrence of flow separation and therefore increasing the critical mach number. While this is even needed on commercial passenger planes the proposed airframe in this paper is way under the size and speed limits to justify a swept wing design.

2.5 Dihedral and Wing Placement

The dihedral angle Γ of a wing refers to the amount the wing surface is angled upwards or downwards (also known as anhedral) from a horizontal position (see Figure 2.5). Its main

purpose is to influence the aircrafts overall lateral stability (see [22] or [28]). An airfoil always creates lift in the direction perpendicular to the surface. If a plane with a dihedral angle $\Gamma \neq 0$ is turned around the roll axis the lift components facing up are unequal, resulting in a rolling force itself, that either increases or counteracts the original roll movement (see, e.g., [1]).

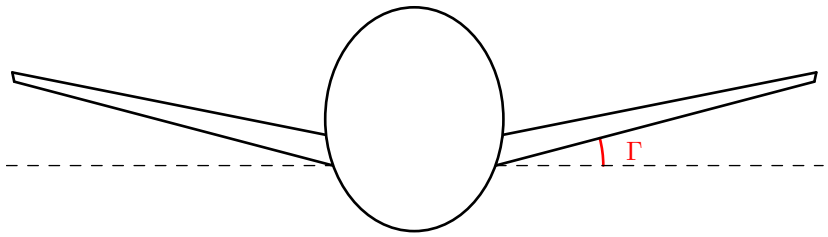

Figure 2.5: Dihedral angle.

The wing placement relative to the center of mass of the aircraft (CoM) dictates the flight stability. A low-wing design generally gives a plane less stability, as the generated lift force acts offset to the CoM, giving the whole system a similar characteristic to an inverted pendulum: an unstable system, that needs to be kept in balance externally. The most common configurations are either planes with a high-mounted wing and an anhedral angle such as the *Antonov An-255* or low-wing designs with a dihedral angle, seen on most passenger and light cargo planes. Both provide a balance between wing placement induced and dihedral stability, as an imbalance leads to uncontrollability through either too much instability or stability. The proposed UAV-model got a top mounted wing to allow for easier assembly, as the wing would only need to be attached to the top of the fuselage rather than to be incorporated into it. Because the design in question is used for development a slight dihedral angle was introduced additionally, to create a more stable testing platform.

2.6 Control Surfaces

The control surfaces of an aircraft are used to change its attitude (see [17]). They do this by deflecting the airstream in a certain direction. As they are offset from the CoM they create a torque on the airframe, changing the plane's attitude over time (see Figures 2.6 and 2.7).

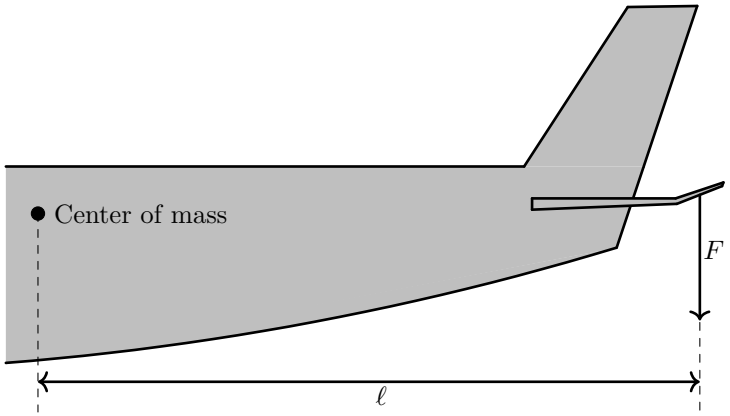

Figure 2.6: Torque ℓF induced by elevator.

The proposed model features an elevator positioned on the horizontal stabiliser at the tail of the aircraft. Its function is to control the plane's pitch angle. Also mounted at the tail is the rudder, a part of the vertical stabiliser that mainly controls the sideslip. This describes the horizontal angle between the plane's heading and the oncoming airstream. The rudder plays an important role in making coordinated turns, in which the sideslip is kept minimal and the gravity experienced on board is perpendicular to the floor despite being influenced by a centrifugal force acting outwards. The effectiveness of the rudder and elevator is increased due to their placement on the tail, reducing the required deflection angle and surface size. This is also the reason for the aileron placement. These are located at the tips of the main wing and are able to set the plane into a rolling motion. This is the main control used to turn the aircraft and change its heading. One important thing to consider is adverse yaw. This describes a slight change in the opposite yaw direction compared to the one navigating towards, caused by a change in drag by deflecting the ailerons. To overcome this the rudder is utilised to minimise sideslip. Furthermore the aileron range of motion can be made slightly asymmetrical by allowing the upward deflection to be slightly larger. This results in a smaller drag difference while still producing the desired lift difference on each side (see Figure 2.7).

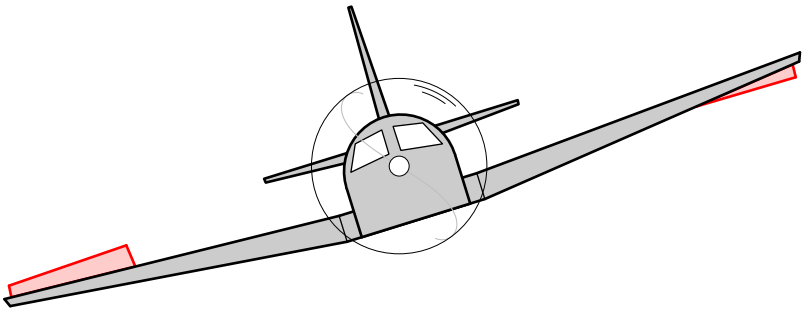

Figure 2.7: Ailerons: The rising wing tip creates more drag compared to the sinking one, as it has to generate lift. This can be overcome by increasing the upwards deflection of the lowering side, matching the drag created by air resistance.

The effectiveness of all control surfaces also depends on the current airspeed, as larger velocities demand a smaller deflection to generate the same torque. Some planes are therefore equipped with split control surfaces, allowing for a reduction in size during cruise and additionally offering redundancy. Most FBW-systems are also able to apply software scaling to the surface deflection, based on the current airspeed. This speed-scaling however was not implemented in the proposed system, as the speed differences of the model are rather small and the later discussed closed loop controller (see Section 4.2) was designed to withstand changes in actuator effectiveness. This also removes another parameter, that, if tuned poorly, could compromise the flight performance.

2.7 Navigation

Fixed wing aircraft navigate by changing their altitude above the ground and their heading, wich describes their yaw angle relative to north. The altitude is managed by changing the lift generated by the wings. This is done by either changing engine power, affecting airspeed or by altering the pitch angle, resulting in changed AoA. Both influence the generated lift and therefore a change in the plane's vertical velocity, also referred to as climb rate. To maintain a certain altitude the plane has to keep the climb rate theoretically at zero, only compensating

for external influences such as turbulence or weather disturbances. The proposed flight controller could track a given altitude only by changing the pitch angle. This would add simplicity and give the pilot the freedom of choosing the flying speed. Planes change their heading by performing a coordinated turn. This manoeuvre is performed by rolling the plane sideways using the ailerons and the rudder to correct for sideslip. This results in the generated lift pointing diagonally. The vertical part keeps the plane in the air but needs to be increased using the elevator, as it is now lower compared to level flight. This effect can be quantified by looking at Figure 2.8: For level flight, the lift $L_0 = mg$, while $L_\varphi = mg \sec\varphi = L_0 \sec\varphi > L_0$ for $\varphi > 0$. The other lift component points towards a hypothetical turn center C and keeps the plane on a circular path with a constant radius R. Said radius R can be calculated from the airspeed v and the roll angle φ (the so called bank angle), as is easily derived from Figure 2.8: The gravitational force is mg (where m is the mass of the aircraft) and the centripetal force is $mR\omega^2$ (ω denotes the angular velocity).

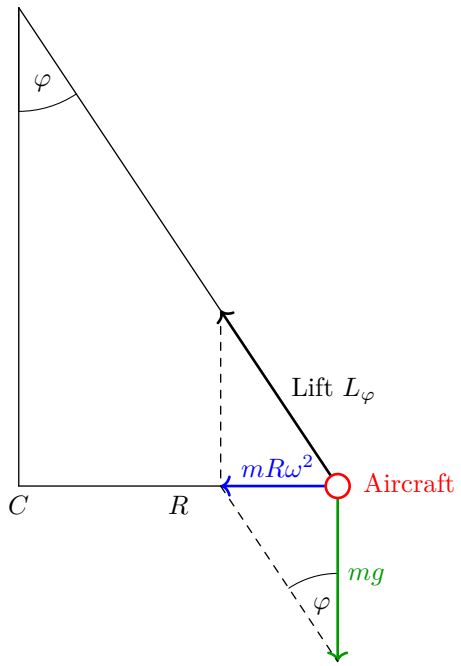

Figure 2.8: Aircraft during turn flight.

Indeed, we read off that $\tan\varphi = \frac{R\omega^2}{g}$. Recall, that the angular velocity ω and the velocity v are connected via $\omega = \frac{v}{R}$. Hence, we obtain

$$R = \frac{v^2}{g \tan\varphi}\,. \tag{2.8}$$

For more information we refer, e.g., to [30], [31] or [15].

2.8 Material

The construction material of the airframe had to feature a low density while still offering sufficient rigidity. One cost effective match for this is foam board, a material consisting mostly of low density polyethylene foam. This foam is cast into sheets and then sandwiched between two layers of bentonite reinforced paper. This material has the additional benefit of

being easy to build with as it can be cut with cutters and is compatible with most glues. The airplane was fully constructed out of this material, only with the addition of a few 3d-printed parts such as motor brackets and wooden dowels.

2.9 Model

As a starting point a model that best matched the desired criteria was chosen. In this case it was a set of plans for the *Simple Cub*, designed by Josh Bixler at *FliteTest*. The design was then modified to fit the required electronics, increase structural stability, feature an acceptable wing-loading and reduce adverse yaw.

Figure 2.9 shows a photo of the model that was created in the course of this matura thesis.

Figure 2.9: The final model aircraft

3 Electrical aspects

We start with an overview over the full electrical system in Figure 3.2. We discuss all relevant components in the subsequent sections, housed all inside compartments of the fuselage as shown in figure 3.1.

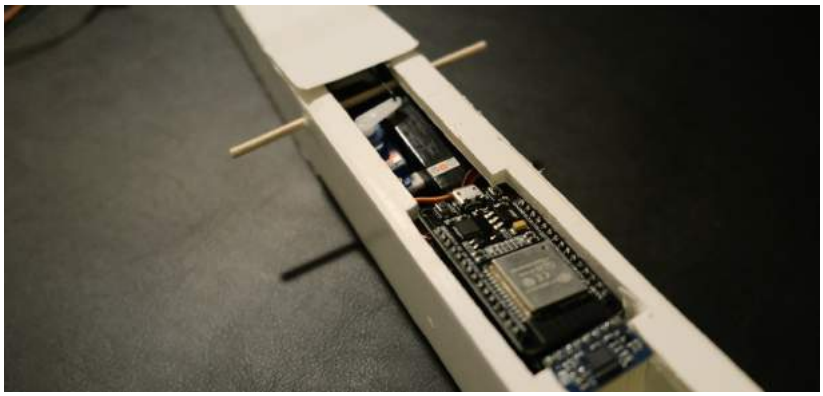

Figure 3.1: The electrical components inside a compartment of the fuselage.

3.1 Controller

The controller forms the core of the flight control system. It is responsible for acquiring sensor measurements, handling the telemetry data and commanding all control surfaces of the aircraft. For the use on a model-scale plane the controller had to additionally have a small form factor and be energy efficient. The two main options that matched these criteria where either a micro controller or a single board computer like for example a *Raspberry Pi*.

The main difference between these two types is the way software is executed on the device. A single board computer runs an operating system that forms a layer between the actual hardware and the application side. It is responsible for interfacing with different physical components and managing software execution. Most operating systems use a non-deterministic task manager for this, wich is unproblematic for most applications such as consumer use, as they are not time critical. This however could pose problems for the use as a flight controller, because the stabilisation of an aircraft is both dependant on a high update frequency and a reliable execution order. This could theoretically be fixed by heavily customising the deployed operating system but would result in a lot of additionally required work. A micro controller however would overcome this issue, as they either run the compiled programs directly on the hardware or employ a special real time operating system. These are heavily stripped down forms of operating systems, that only provide rudimentary

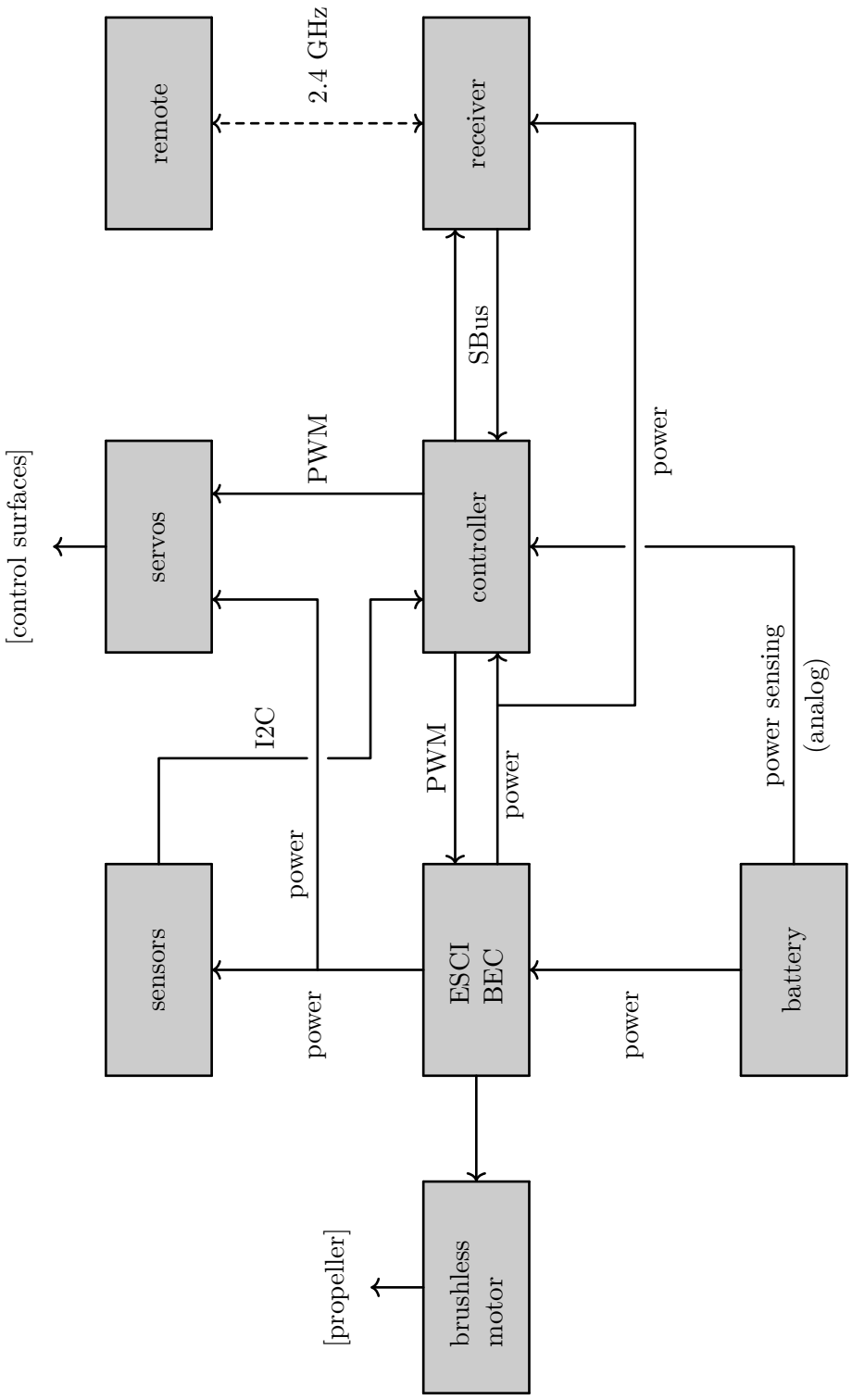

Figure 3.2: System Overview.

functions such as program multitasking, which if possible are executed in real time or at least scheduled in a fully deterministic order. Additional benefits of micro controllers are a lower energy consumption and higher system reliability, as hardware and software are less complex.

At the beginning of the project a stock *Arduino Mega 2560 Rev. 3* was utilised but was later changed to an *ESP-32* due to a higher performance demand (see [13]). The new micro controller featured a 32-bit design, allowing single-cycle floating point operations, dual processor cores for real time multitasking and a 240MHz clock speed. The latter would even enable to software-emulate certain communication protocols.

3.2 Telemetry

To command the model aircraft a wireless communication link was required. This was achieved with a normal hobby rc-remote made by *FrSky*. This remote transmitted the user control inputs over a 2.4GHz wireless link to a receiver on the plane. The receiver itself was able to control the onboard motors directly to allow for manual flight, as well as outputting all revived signals over a data bus, giving the system an uplink connection for control commands. Additionally the receiver also offered downlink capability by accepting data over the bus and transmitting it back to the remote, providing the pilot with information like battery voltage or the plane's current altitude.

The data bus utilised by *FrSky* is called SBus, an uni-directional, serial data bus that was originally developed by *Futaba*. This protocol had to be custom implemented on the *ESP-32*, as it is rather uncommon. The utilised SBus consists of two layers: a hardware and a software transport. The hardware transport is responsible for physically carrying the information over the wires in the form of voltage changes. To transmit the information most transports encode the data into smaller units called frames, which are then decoded again by the receiving end. On a hardware level SBus resembles a universal serial standard called UART (Universal asynchronous receiver-transmitter). Both protocols are asynchronous, meaning that they do not require an additional clock signal (see Figure 3.3), as it is already contained within in the data signal. This reduces the amount of data lines needed to one, which also restricts the information flow to one single direction (without employing a complex collision avoidance). The system has to be doubled to allow for a duplex connection (see Figure 3.4) by connecting the receiving signal of one chip to the receiver of the other one and vice versa (see also [20] or [5]).

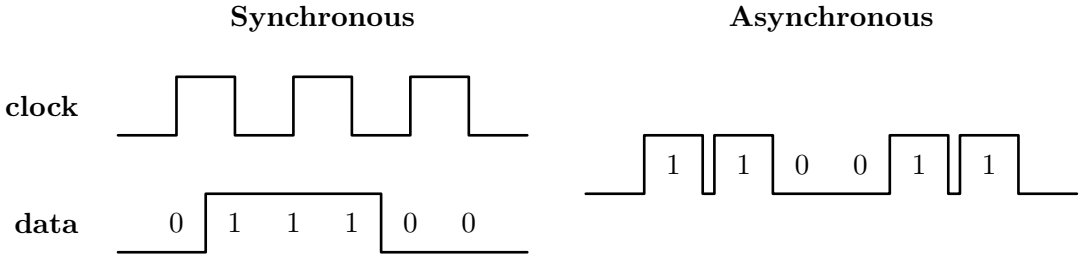

Figure 3.3: Synchronous vs. Asynchronous data transmission

While UART dictates the overall structure of a frame (see Figure 3.5) its exact contents can be adjusted to fit the desired application. This includes one that exactly matches an SBus frame, requiring a payload size of 8-bits, one parity and two stop bits. These notify

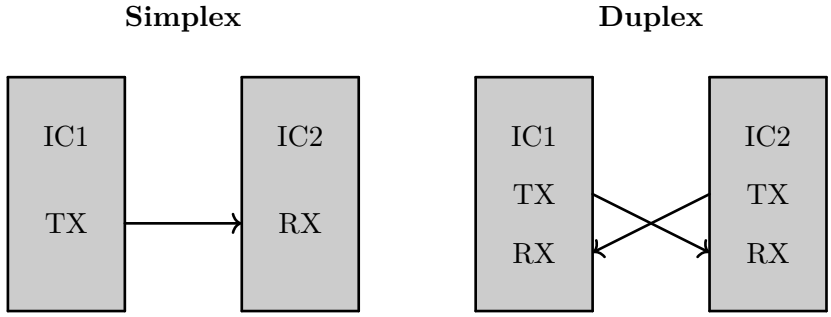

Figure 3.4: Simplex vs. (full) Duplex communication link

the receiver of the begin/end of a frame and offer a measure against data corruption by the means of parity.

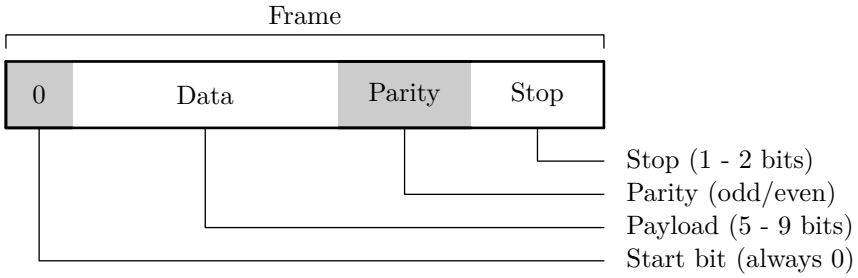

Figure 3.5: The structure of an SBus frame.

Additionally SBus utilises a baud-rate of 100′000 bit/s and an inverted logic level compared to standard UART, which can be easily configured within the code (see Code 3.1).

```
1 bus->begin(100000, SERIAL_8E2, 16, 17, true, 1000);
```

Code 3.1: Configuration of the UART bus

Secondly comes the software transport layer. It describes how the data is structured before it is encoded into frames. In most cases this is done by creating data packets that contain a certain payload and sometimes additional information like timestamps or sender information. As the purpose of SBus is to transmit analog user inputs in the form of lever positions with a high reliability it only utilises one packet type with a fixed size. These packets contain one header byte that indicates the start of a packet, followed by 22 bytes that contain 16 channels of which every one represents a lever position encoded into 11 bit integer numbers. The packet is then closed by sending one byte containing binary flags to indicate if the connection with the remote has been lost and finally an empty stop byte. This structure allows the packets to be decoded easily and also makes the whole system stateless, meaning that the connection can be instantly resumed after interrupts without requiring some handshake procedure. The software to decode the SBus packets was fully custom implemented and is an improved version of the code written by Brian R. Taylor from *Bolderflight Systems* (See 6.1 and 6.2).

With the ability to receive the user inputs from the hobby receiver the flight controller was now able to control the plane based on the pilot controls and the onboard sensor data. The setup would also allow to completely bypass the control system in emergencies by simply

forwarding the user inputs to the motors just like the receiver would have done, again giving fully manual control of the plane.

3.3 Sensors

The plane was fitted with numerous sensors to gather data about certain flight variables such as air pressure or linear acceleration. These sensors have to first be configured by the controller at takeoff to adjust parameters like sensitivity and update rate, requiring a link from the controller to multiple sensing devices. During flight the captured data now has to be transported the opposite way from multiple devices towards the controller. This requirement for multi-device duplex rules out SBus/UART, as they only support device-to-device communication. The most common data bus, that is supported by most sensors is I2C, another serial bus that was developed by *NXP Semiconductors* (formerly *Philips Semiconductors*) in 1982. Unlike the above mentioned protocols it is synchronous (see Figure 3.3), meaning that the clock signal is carried separately by one conductor, while the second one only transmits the data's logic level. I2C falls into the category of Master-Slave-Buses, meaning that every system has one master that is responsible for managing traffic and one to multiple slave devices (see Figure 3.6 and [3]). This architecture allows for a half-duplex connection, where the master can write directly to a device and also retrieve data by sending a read-request and waiting for the device to respond. The system is similar to full duplex as it is bi-directional but restricted by the fact that traffic can only go one way at once and is regulated by the master. To discriminate between the different devices everyone has a pre-defined one-byte address, allowing for a maximum of 256 connected devices. The master can now address every device individually by including the corresponding id in every message sent. This theoretically poses a security risk as all devices still receive every message. This can however be ignored in most applications, as the protocol is intended for chip-to-chip communication within a single machine and is therefore not directly accessible from the outside.

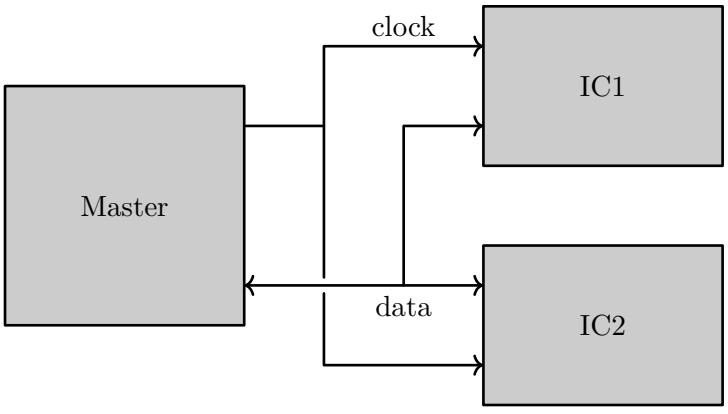

Figure 3.6: A typical I2C network.

To enable assisted FBW the plane had to capture information about its current altitude and attitude. As the altitude is not directly measurable it had to be estimated using a temperature and an air-pressure sensor. The data was then combined using a sensor fusion algorithm, discussed later. Secondly the plane needed an attitude heading reference system (AHRS). This system tracks its orientation angles, consisting of roll, pitch and yaw compared to a reference frame. Here the heading describes the angle between north and the plane's heading direction and can be easily calculated from the yaw angle. The aircraft's spacial

orientation could only be directly captured with the use of a flywheel gyroscope of some sort. The problem with these sensors is that they are difficult to source, relatively heavy and quite energy inefficient, as their flywheel has to be kept at a certain velocity. These downsides can be overcome on larger, higher budget aircraft like passenger planes or fighter jets but clearly not on a small model. Therefore the attitude had to be also measured indirectly through a MEMS gyroscope. MEMS stands for Micro Electro Mechanical System and describes a technology used to create microscopic mechanical devices using the same silicon wavers as integrated circuits. This allows for the production of tiny, super light weight and energy efficient sensors that are mainly used in consumer electronics such as phones or smartwatches. Compared to the flywheel version a MEMS gyroscope is only able to track the angular velocity instead of absolute angles and was therefore combined with the linear acceleration and earth's magnetic field measurements using another fusion algorithm.

Therefore final sensor range of the plane included only MEMS devices consisting of:

- Barometer to capture air pressure
- Temperature sensor to enhance altitude estimations
- Gyroscope to capture angular velocities
- Accelerometer to estimate a gravity vector and external forces on the plane
- Magnetometer for navigation and to enhance attitude estimation

3.4 Servos

The flight control surfaces, consisting of rudder, elevator and ailerons are all driven by servo motors. These are electric motors with an integrated positional encoder to provide closed-loop-feedback. This allows the system to control the exact deflection angle of the control surfaces even under the influence of external forces, such as drag and turbulence. To qualify for use on a model plane these actuators had to be compact and light, which perfectly matches commercially available hobby servos for rc-models. These use a potentiometer as encoder and feature a simple controller that can be commanded by sending pulse-width-modulated signals (PWM) over a single wire. This is done by applying a rectangular waveform with a period duration of 20ms, that is driven by the flight controller. The demanded position is then encoded by varying the duty cycle of said signal, usually ranging from 1ms to 2ms (see Figure 3.7). This leads to an unidirectional protocol that allows the servo position to be updated with a frequency of 50Hz but provides no way for data from the servo to reach the controller. This means that the system is fully reliant on the internal servo loop and cannot be monitored. An advantage of the PWM protocol however is its ease of use and robustness, as the signal is really resistant against external noise and the square wave being able to be generated on the most minimal hardware. For more information see [32] and [33].

3.5 Powertrain

The main motor has to have enough power to propel the entire plane forward, forcing air over the airfoils, generating lift. As the rest of the project is already fully electric and to reduce cost and complexity an electric solution was chosen. Brushless electric motors are ideal for this application, as they offer a lot of power in a small form-factor. These motors do not use

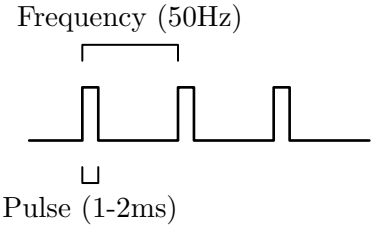

Figure 3.7: PWM Signal

brushes like a classical DC-motor but are instead driven by three-phase alternating current, that directly powers the stator. They are interfaced via an electronic speed controller (ESC) that can change the rotational speed by changing the frequency of the drive signal. Most ESCs are also capable of motor feedback by measuring the signal induced back by the motor coils, allowing for more accurate speed control. The protocol to communicate with most hobby motor controllers is conveniently the same as the one used by the servos, with the only difference being the controlled value changing from a stationary angle to rotational velocity. An ESC is usually connected straight to the battery of the model plane as the main motor has the biggest energy draw. Therefore it is also fitted with a battery eliminator circuit (BEC), designed to protect the connected battery from over-discharge. To power the rest of the electronics the controller also features an auxiliary power supply, usually regulated at 5 volts. In the project this was fed directly into the main power distribution wiring, powering the flight controller, sensors, servo motors and the telemetry transceiver (see also [29]).

The battery of the airplane was a multi-cell LiPo battery, chosen because of its high power density. Despite this it makes up for a substantial amount of the aircraft's mass and therefore its placement needs extra consideration regarding the CoM. It is able to provide a voltage, larger than its individual cells by chaining multiple in series. These can then be simply discharged by connecting them to desired load such as the ESC. Here it is also possible to measure the battery terminal voltage and current draw to estimate its charge and notify the pilot of the remaining flight time. This can be done by connecting it to a voltage divider to proportionally step-down the voltage, a current sensing resistor and then use an analog to digital converter (ADC) of the microcontroller to convert the analog voltage into a digital measurement (see Figures 3.8 and 3.9).

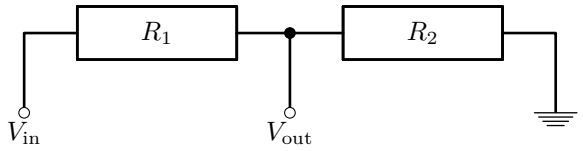

Figure 3.8: A simple voltage divider: $V_{\text{out}} = V_{\text{in}} \frac{R_2}{R_1+R_2}$.

The charging of LiPo batteries is a bit more complicated, as a specific charge procedure has to be followed to ensure equal cell voltage and prevent overheating. This can be done using a commercially available universal battery charger. Figure 3.10 shows the assembled powertrain consisting out of the motor, the ESC and the LiPo battery.

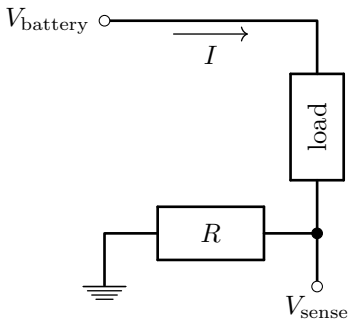

Figure 3.9: A low-side current sensing circuit: $I = \frac{V_{\text{sense}}}{R}$.

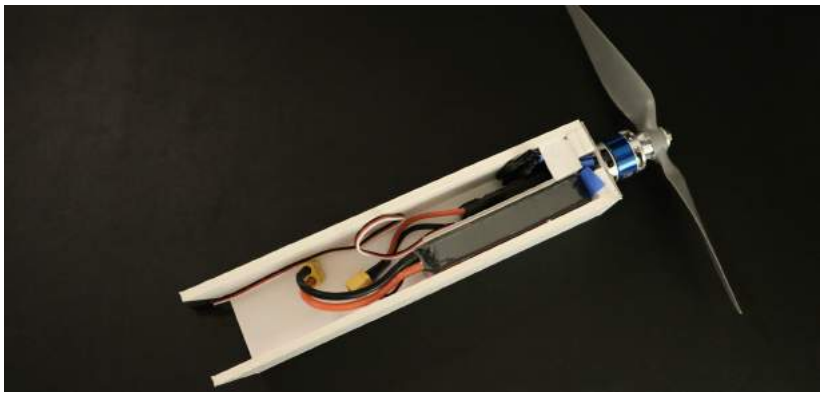

Figure 3.10: The powertrain inside its dedicated compartment before being mounted inside the fuselage.

4 Software

4.1 Digital signal processing

Many integrated sensors are designed to fit a certain envelope regarding price and overall size, compromising their performance. This combined with with the unavoidable nature of any sensing system biasing its observations through measurement errors can lead to some noisy and inaccurate data streams. The following are approaches used to overcome these issues and deliver more accurate and stable data to the flight control system.

4.1.1 Digital low pass filter

Many sensors offer a high update rate, leaving only little time to do the actual measurement. This makes the system susceptible to sensor noise, that can be seen while comparing the signal to a reference (see Figure 4.1).

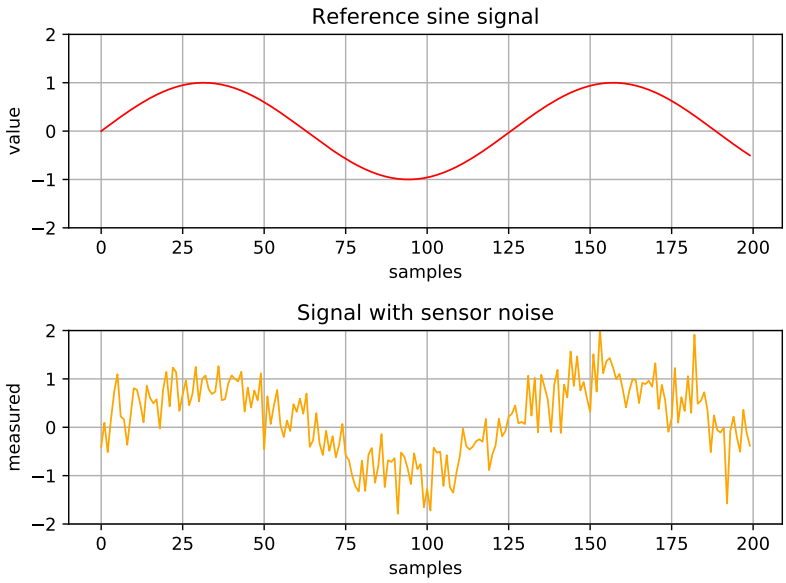

Figure 4.1: A reference signal with period $T = \frac{\pi}{20}$, and one with gaussian noise of $\mu = 0$ and $\sigma = 0.4$

The simplest way to deal with this noise is the implementation of a low pass filter. These filters have a given cutoff frequency F_c and suppress the parts of a signal that exceed said threshold. Every implementation does not discriminate exactly at the cutoff point but grad-

ually increases the signal attenuation (see Figure 4.2). The steepness of the cutoff slope is a characteristic of every filter implementation and is generally referred to as the filter order (see also Figure 4.2). A larger order describes a smaller cutoff band but may also increase the filter's response time (see also [10]).

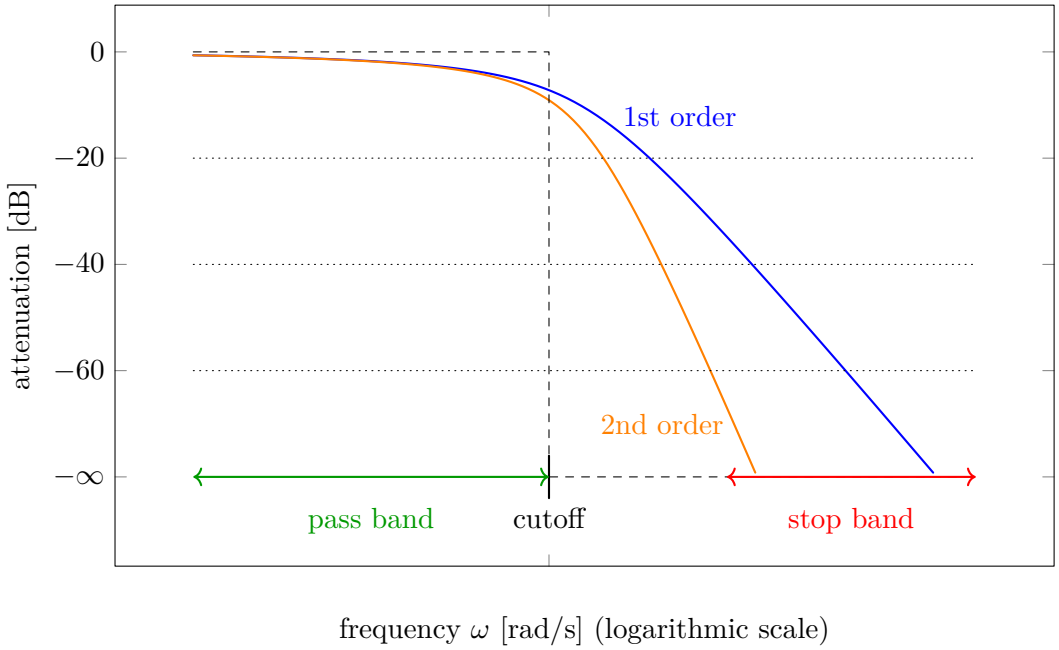

Figure 4.2: Low pass filter. The dashed line indicates a hypothetical ideal filter.

This filter is commonly found in analog electronics, such as audio equipment but also well established in the field of digital signal processing. Here it is based on the assumption, that the frequency of the unwanted noise is higher than the maximum fluctuation rate of the real world variable. This means that the sensor noise can be smoothed by setting the cutoff F_c between the real world fluctuation and the sample frequency of the sensor.

As the proposed implementation of said filter has to run on the limited hardware of the flight controller a heavily simplified version of Stephen Butterworth's approach was utilized (see [27]). Here y is the filter output at any given time, Δt is the time since the last update, x is the filter input, and F_c is the cutoff frequency. The update equation is given by

$$y(t + \Delta t) = (1 - a)x(t) + y(t)a, \quad \text{where } a = e^{-2\pi F_c}\Delta t.$$

This implementation corresponds to a first order filter that is also commonly referred to as single pole. Its transfer function in the complex frequency plane contains one single peak. A low pass filter offers an efficient way to reduce induced sensor noise as said error is attenuated by the filter. This however leads to a tradeoff between a small cutoff to allow for strong data smoothing but also induces a high delay and a high cutoff that ensures a fast response time but provides less smoothing (see Figure 4.3).

4.1.2 Kalman Filter

The classic Kalman filter, also known as linear quadric estimator is one of the most common methods used for system state estimation and sensor fusion. It was mainly developed by

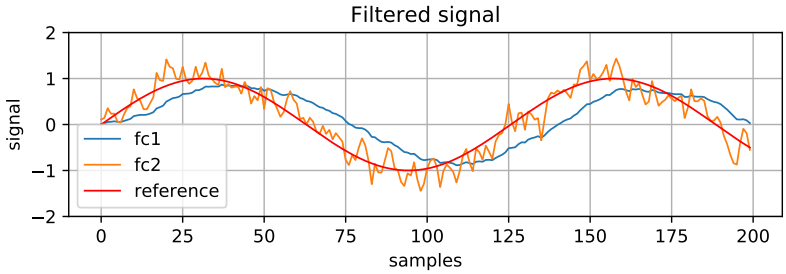

Figure 4.3: The reference signal compared to the noise signal with $\mu = 0$ and $\sigma = 0.4$, filtered with $F_{c_1} = 9.8 \cdot 10^{-3}$ and $F_{c_2} = 1.1 \cdot 10^{-1}$. The blue signal has less noise but shows a phase shift relative to the reference. Note how the blue signal is slightly attenuated, as its frequency is close to the cutoff.

Rudolf Kálmán around 1960 and played a key role in the Apollo program, forming the core of the computer guidance system.

First a mathematical model of the system in question has to be created. This model consists of defining a plant state $x \in \mathbb{R}^n$, that contains all relevant variables. Next a state transition function, represented by matrix $A \in \mathbb{R}^{n \times n}$, has to be defined, that describes the theoretical future outcome $\hat{x} \in \mathbb{R}^n$, given the current plant state x:

$$\hat{x} = Ax.$$

The filter now works in a two step process, where first an estimate $\hat{x}$ using the equation above is made. As the true plant state is never truly known every state is associated with an uncertainty, represented in a covariance matrix $P = \mathrm{Cov}(x)$. This also has to be applied to the prediction step, giving the theoretical next state $\hat{x}$ with its associated uncertainty $\hat{P} = \mathrm{Cov}(\hat{x})$, derived from the previous covariance $\mathrm{Cov}(x)$:

$$\hat{P} = \mathrm{Cov}(\hat{x}) = \mathrm{Cov}(Ax) = A\,\mathrm{Cov}(x)A^T = APA^T.$$

As the state transition function F is idealistic it is never fully accurate. The plant is subject to external influences that cannot be included in the mathematical model. This results in every prediction increasing the uncertainty, what can be implemented in the filter by defining an external noise matrix Q, that describes the added uncertainty for every step.

In some cases the external influences on the plant are based on other known variables of the system. In this case it is possible to decrease the prediction uncertainty Q by modelling the external factors into the prediction equation. Here $u \in \mathbb{R}^m$ represents the known system variables and matrix $B \in \mathbb{R}^{n \times m}$ describes the effects said variables have on the plant state x. This resembles the classic discrete state space formula:

$$\hat{x} = Ax + Bu.$$

This leads to the final filter equations for the prediction step:

$$\hat{x}_k = A_k \hat{x}_{k-1} + B_k u_k \tag{4.1}$$
$$P_k = A_k P_{k-1} A_k^T + Q_k. \tag{4.2}$$

The second step of one filter cycle is the update step. It combines the predicted state $\hat{x}$ with some real sensor measurements $z \in \mathbb{R}^m$. To do this the predicted state $\hat{x}$ is transferred to the expected sensor results $\hat{z}$ with the measurement matrix $H \in \mathbb{R}^{n \times m}$:

$$\hat{z} = H\hat{x}.$$

This allows the filter to compare $\hat{z}$ with the gathered measurements z to improve the filter state and decrease the associated uncertainty. One other advantage is that the entire system state does not have to be measured directly by the sensors but can be derived from their results.

To get the required filter equations the system is simplified to one dimension. A Gaussian distribution $\mathcal{N}(\mu, \sigma^2)$ of a random variable X with a variance of σ^2 and a mean μ is expressed by its density function:

$$g(x, \mu, \sigma) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

This means, that for small Δx the probability that X takes a value in the interval $(x, x+\Delta x)$ is given by $g(x, \mu, \sigma)\Delta x$. Two random variables X_1, X_2 with Gaussian distributions $\mathcal{N}(\mu_1, \sigma_1^2)$ (representing the distribution of the expected sensor results $\hat{z}$), and $\mathcal{N}(\mu_2, \sigma_2^2)$ (representing the gathered noisy measurements z) represent a random variable $(X_1, X_2) \in \mathbb{R}^2$ with density

$$(x_1, x_2) \mapsto g(x_1, \mu_1, \sigma_1) g(x_2, \mu_2, \sigma_2),$$

i.e., the probability that the vector valued random variable (X_1, X_2) assumes a value in $(x_1, x_1 + \Delta x_1) \times (x_2, x_2 + \Delta x_2)$ is given by $g(x_1, \mu_1, \sigma_1) g(x_2, \mu_2, \sigma_2) \Delta x_1 \Delta x_2$. Along the diagonal $X_1 = X_2$ (i.e., the event that the measurements $\hat{z}$ which correspond to the predicted state $\hat{x}$ agree with the actually gathered measurements z) the density is the product

$$g : x \mapsto g(x, \mu_1, \sigma_1) g(x, \mu_2, \sigma_2).$$

This corresponds to a conditional probability to find a value X_1 in $(x, x + \Delta)$ subject to the condition that also X_2 takes the value in the same interval. To get the density which corresponds to this conditional probability one has to normalize g such that its integral over $\mathbb{R}$ equals 1. Observe, that

$$g(x) = \frac{1}{2\pi\sigma_1\sigma_2} \exp\left(-\frac{(x-\mu_1)^2}{2\sigma_1^2} - \frac{(x-\mu_2)^2}{2\sigma_2^2} \right).$$

Adding the two fractions and completing the square reveals that the resulting distribution is again a Gaussian distribution $\mathcal{N}(\mu, \sigma^2)$ with

$$\mu = \frac{\mu_1\sigma_2^2 + \mu_2\sigma_1^2}{\sigma_1^2 + \sigma_2^2} \tag{4.3}$$

$$\sigma^2 = \frac{\sigma_1^2\sigma_2^2}{\sigma_1^2 + \sigma_2^2}. \tag{4.4}$$

These terms can be rewritten by using the factor

$$K = \frac{\sigma_1^2}{\sigma_1^2 + \sigma_2^2} \tag{4.5}$$

introduced by Kalman. We obtain

$$\mu = \mu_1 + K(\mu_2 - \mu_1) \tag{4.6}$$

$$\sigma^2 = \sigma_1^2 - K\sigma_1^2. \tag{4.7}$$

K is referred to as the Kalman gain, because it *reduces* the variance (the uncertainty) σ_1^2 of the predicted state to the smaller value $\sigma^2 = \sigma_1^2 - K\sigma_1^2$ by comparing the prediction with the actual measurements. This was Kalman's groundbreaking idea (see [4] for the complete theory).

In higher dimensions, one has to replace the variance by the covariance matrix: Here, the sensor data $z \in \mathbb{R}^m$ with their associated covariance $R := \mathrm{Cov}(z)$ are compared to the expected measurements $\hat{z} = H\hat{x}$ and their covariance $\mathrm{Cov}(\hat{z}) = HPH^T$ (where $P := \mathrm{Cov}(\hat{x})$), resulting in the final estimated state $\hat{x}'$ with its covariance $P' := \mathrm{Cov}(\hat{x}')$. The corresponding equations are

$$\begin{aligned} K &= H_k P_k H_k^T (H_k P_k H_k^T + R_k)^{-1} && (4.8)\\ H_k \hat{x}'_k &= H_k \hat{x}_k + K(z - H_k \hat{x}_k) && (4.9)\\ H_k P'_k H_k^T &= H_k P_k H_k^T - K H_k P_k H_k^T, && (4.10) \end{aligned}$$

where we used the same colors as in (4.5), (4.6), and (4.7) to make the correspondence visible. This results in the following filter update equations

$$\begin{aligned} K' &= P_k H_k^T (H_k P_k H_k^T + R_k)^{-1} && (4.11)\\ \hat{x}'_k &= \hat{x}_k + K'(z - H_k \hat{x}_k) && (4.12)\\ P'_k &= P_k - K' H_k P_k && (4.13) \end{aligned}$$

where we use K' connected to the Kalman gain via $K = H_k K'$.

4.1.3 Sensor Fusion

To give the proposed aircraft the ability to stabilise itself during flight the system needs to acquire real time data. This is done through a range of on board sensors that feed the information directly to the flight controller. As this data is subject to measurement errors and may not be directly correlated to a desired variable it has to be estimated. This is done through an algorithm that fuses together different related pieces of data to estimate the wanted value.

Altitude

As it is difficult to directly measure the altitude of an object directly through on-board sensors it had to be estimated using available data. In the proposed system this was done through a barometer, and a vertical accelerometer. A simple variant of the *Barometric formula* is the so called *International altitude formula*: Normalized by the values for temperature $T = 15°\mathrm{C} = 288.15\,\mathrm{K}$, air pressure at see level $p_0 = 1013.25\,\mathrm{hPa}$, temperature gradient $\Delta T = 0.65\,\mathrm{K/m}$ (temperature decrease per 100 height meters), one gets

$$h = \frac{288.15\,\mathrm{K}}{0.65\,\mathrm{K/m}} \left(1.0 - \left(\frac{p}{1013.25\,\mathrm{hPa}}\right)^{\frac{1}{5255}}\right) \tag{4.14}$$

which gives a good estimate for altitudes $h < 11\,\mathrm{km}$ as a function of the measured air pressure p in hPa. A slightly better approximation which takes into account the current weather condition is possible if the altitude $\hat{h}$ above sea level of the launch site and the corresponding air pressure $\hat{p}$ are known. In this case, one just replaces the constant in (4.14) in front of the bracket by

$$C = \frac{\hat{h}}{1.0 - \left(\frac{\hat{p}}{1013.25\,\mathrm{hPa}}\right)^{\frac{1}{5255}}}.$$

Then, the modified formula

$$h = C \left(1.0 - \left(\frac{p}{1013.25\,\mathrm{hPa}}\right)^{\frac{1}{5255}}\right) \tag{4.15}$$

trivially gives the correct altitude at the launch site.

The problem with only this approach is that the barometer sensor data are really noisy and therefore has to be put through a digital low pass filter. This increases the accuracy but also increases the time for real changes to propagate through the filter, compromising a fast response time. This results in a measurement that offers a high drift resistance over longer time periods, as the filter averages the noisy data to one theoretically more accurate one. The accuracy of the used air pressure sensor is remarkable: By (4.15) it indicates changes in altitude of 20 cm correctly.

On the other hand the vertical position h can also be computed given the starting value $h_0 = h(0)$, starting velocity $v_0 = v(0)$ and the vertical acceleration $a(t)$:

$$h(t) = h_0 + v_0 t + \int_0^t \int_0^\tau a(u) du \, d\tau. \tag{4.16}$$

If the acceleration is measured at the time steps $t_k = k\Delta t$, the formula (4.16) can be discretized and hence estimated as follows:

$$\begin{aligned} v(t_{k+1}) &= v(t_k) + a(t_k)\Delta t \\ h(t_{k+1}) &= h(t_k) + v(t_k)\Delta t. \end{aligned}$$

This approach offers a really fast response to changes in the real-world value. The downside here is that the measured variable $a(t)$ is subject to the sensor noise, meaning that these errors are gradually accumulated by the integrator.

The methods proposed above both have their own advantages and disadvantages that seem to complement each other. The slow-response approach could be used to correct the measurement drift while the other one could increase the system response time over shorter time periods. To fuse both approaches the above mentioned Kalman filter was utilised. Here the state contained the altitude h, the vertical velocity v and vertical acceleration a:

$$x = \begin{pmatrix} h & v & a \end{pmatrix}^T.$$

The system in question can now be described as discrete state space equation

$$x_{k+1} = \begin{pmatrix} 1 & \Delta t & \frac{1}{2}\Delta t^2 \\ 0 & 1 & \Delta t \\ 0 & 0 & 1 \end{pmatrix} x_k$$

To find the system noise Q a method for the creation of discrete white nose for systems with timestep Δt consisting of derivatives up to order 2 was used:

$$Q = GG^T, \text{ where } G = \begin{pmatrix} \frac{1}{2}\Delta t^2 \\ \Delta t \\ 1 \end{pmatrix}$$

The measurement matrix H and a covariance matrix R of the measurement. More precisely, with $\sigma_h^2 > 0$ describing the barometer and $\sigma_a^2 > 0$ the accelerometer noise variance, we have

$$H = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix}, \quad \text{with the measurement estimate being } \hat{z} = Hx,$$

and

$$R = \begin{pmatrix} \sigma_h^2 & 0 \\ 0 & \sigma_a^2 \end{pmatrix}.$$

An initial state x_0 with associated covariance matrix P representing the uncertainty of the state is given by

$$x_0 = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}, \quad P = \begin{pmatrix} \sigma_h^2 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & \sigma_h^2 \end{pmatrix}.$$

This allows the system to estimate the not measured vertical velocity, representing the aircraft climb rate, and enhance the altitude and acceleration measurements. To improve the system performance even further the data from the accelerometer was first smoothed by a digital low pass filter (see Figure 4.4).

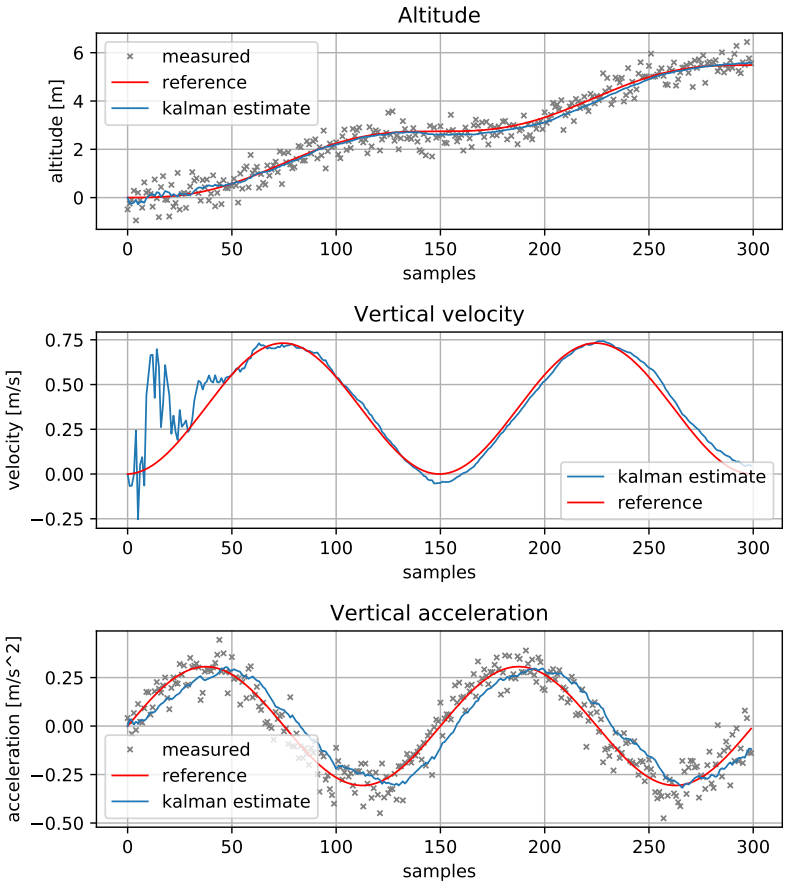

Figure 4.4: The results of the proposed altitude estimator compared to the reference signal. The system shows good performance after an initiation period of 50 samples. Note that the acceleration estimate is not as accurate as the other components. This is because the state transition A directs the information flow towards the primitives, i.e., velocity and altitude.

See Code 6.7 in Chapter 6 for the Python code which belongs to this section.

Attitude

The attitude of the aircraft, consisting of roll, pitch and yaw, had to be estimated from the angular velocities measured by the MEMS gyroscope. This can be done through an integrator, that combines all measured velocities $v(t) \in \mathbb{R}^3$ at time t to get the current

orientation $x \in \mathbb{R}^3$, based on the initial position x_0:

$$x(t) = x_0 + \int_0^t v(t)dt$$

In the presented system the angles are integrated as Euler angles, which makes the system prone to gimbal locking. This is a situation, where two of the three Euler axis enter a parallel alignment, reducing the system's degrees of freedom by one. If the following update step contains a component on the locked axis other than zero, said information is lost. This would represent a catastrophic failure of the system, resulting in the flight controller loosing orientation. Therefore another representation for 3-dimensional rotations was used, consisting of a quaternion (see, e.g., [12]). Giving the orientation state another degree of freedom prevents it from ever entering the above described situation. The quaternion q can be interpreted as containing a real component $q_R \in \mathbb{R}^3$, representing a rotation axis in 3D-space and an imaginary part $q_I \in \mathbb{R}$, describing the rotation angle around said axis. The aircraft's orientation can still be represented in the classic Euler format, given that said information is not chained together, meaning that only the integrator was forced to utilise it.

Another problem with integrating the raw data lies in the sensor error that is present in each measurement. This leads to a buildup of errors over time that render orientation tracking relying only on this method almost unusable. To successfully track a value through integration would only be possible if the measurements would fully represent the truth, making this approach a purely hypothetical solution. To overcome this the system had to gather additional information that related to the searched attitude but where not subject to error buildup induced drift. This can be done through measuring earth's gravity, assuming gravity is present in the environment in question and would form a homogenous, parallel field. Another information source was the on-board magnetometer, providing earth's magnetic field, again assuming that said field was approximately homogenous with a parallel orientation. This information is not fully accurate due the approximations but not subject to drift, offering a good counterpart to fuse with the gyroscope's velocity data.

4.1.4 Extended Kalman Filter

The AHRS system of the airplane was based on another sensor fusion algorithm, consisting of a modified Kalman filter. The state of this filter $x \in \mathbb{R}^7$ consists of a quaternion q, that represents the aircraft's rotation in reference to earth and a bias vector $b \in \mathbb{R}^3$ for each gyroscope axis, that serves as the filter's estimate of the measurement error, making the filter an error state Kalman filter. These are a special category of filters that improve their performance by additionally estimating biases of connected sensors as part of the full system state:

$$x = \begin{pmatrix} q_0 & q_1 & q_2 & q_3 & b_0 & b_1 & b_2 \end{pmatrix}^T$$

In this case this is only applicable to the raw gyroscope data v that is corrected using the estimated bias b into $\hat{v}$:

$$\hat{v} = \begin{pmatrix} b_0 & 0 & 0 \\ 0 & b_1 & 0 \\ 0 & 0 & b_2 \end{pmatrix} v$$

As the gyroscope delivers rotational velocity it can be used to predict where the system's state is probably moving. This is done by applying the velocities to the state quaternion, while also adjusting the gyroscope biases according to the current orientation q. This can

be represented as a state transition function as follows:

$$x(t + \Delta t) = Ax(t)$$

where

$$A = \begin{pmatrix} 1 & -\delta_0 & -\delta_1 & -\delta_2 & \varepsilon_1 & \varepsilon_2 & \varepsilon_3 \\ \delta_0 & 1 & \delta_2 & -\delta_1 & -\varepsilon_0 & -\varepsilon_2 & \varepsilon_3 \\ \delta_1 & -\delta_2 & 1 & \delta_0 & -\varepsilon_3 & -\varepsilon_0 & \varepsilon_1 \\ \delta_2 & \delta_1 & -\delta_0 & 1 & \varepsilon_2 & -\varepsilon_1 & \varepsilon_0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix},$$

and

$$\delta = \hat{v}\Delta t \text{ and } \varepsilon = q\frac{\Delta t}{2}.$$

A standard deviation L is derived from the current orientation but excludes the bias:

$$L = \begin{pmatrix} -\varepsilon_1 & -\varepsilon_2 & -\varepsilon_3 \\ \varepsilon_0 & -\varepsilon_3 & \varepsilon_2 \\ \varepsilon_3 & \varepsilon_0 & -\varepsilon_1 \\ -\varepsilon_2 & \varepsilon_1 & \varepsilon_0 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

This standard deviation L is now scaled with the standard deviation Q of the sensor axis. The variance is then given by LQL^T. This results in the recursion

$$P_{k+1} = A_k P_k A_k^T + L_k Q L_k^T, \text{ where } Q = \begin{pmatrix} \sigma_0^2 & 0 & 0 \\ 0 & \sigma_1^2 & 0 \\ 0 & 0 & \sigma_2^2 \end{pmatrix}$$

This completes the prediction step, that is now combined with additional sensor data to improve the made prediction. The roll and pitch angles where both corrected using the gravity vector, provided by the accelerometer, as both axis are aligned perpendicular to it. Utilising this comes with the added advantage of always referencing both rotation axis to the horizon. The roll Φ and pitch θ can be directly calculated given the gravity vector g:

$$\begin{aligned} \Phi &= \arctan\left(\frac{g_x}{g_z}\right) \\ \theta &= \arctan\left(\frac{g_y}{\|g_{xz}\|}\right) \end{aligned}$$

Here the y-axis points in flight direction, the z-axis is the vertical axis, and g_{xz} denotes the projection of the gravity vector to the xy-plane. It should be noted here, that the computed angles have to be adjusted for their sign to allow for full device inversion and prevent division by 0. Therefore a new function $\text{atan}(y, x)$ was introduced, that replaced all conventional arctan operations:

$$\text{atan}(y, x) = \begin{cases} \arctan(\frac{y}{x}) + \pi & \text{if } y > 0 \\ \arctan(\frac{y}{x}) & \text{if } y < 0 \\ \pi & \text{if } y > 0 \text{ and } x = 0 \\ 0 & \text{if } y < 0 \text{ and } x = 0 \\ \text{undefined} & \text{if } x = 0 \text{ and } y = 0 \end{cases}$$

To obtain the measurement $\hat{z}$ the reverse approach was taken by converting the state quaternion q into the corresponding gravity vector. After some calculations one gets:

$$\hat{z} = \frac{\zeta}{\|\zeta\|}, \text{ where } \zeta = \begin{pmatrix} -2(q_1 q_3 - q_0 q_2) \\ -2(q_0 q_1 + q_2 q_3) \\ q_0^2 - q_1^2 - q_2^2 + q_3^2 \end{pmatrix}$$

This however results in a non-linear function to convert the quaternion q of the filter state into the corresponding values. As a Kalman filter only works for linear systems the model had to be adjusted. This was done by calculating the Jacobian matrix H_a for the measurements to locally linearise the system. This implementation is also referred to as an extended Kalman filter, meaning that its working envelope has been extended to non-linear systems. The partial linearisation offers a good approximation given the system in question is not heavily curved around the estimation range of the filter. For the Jacobian one finds

$$H_a = 2 \begin{pmatrix} q_2 & -q_3 & q_0 & -q_1 & 0 & 0 & 0 \\ -q_1 & -q_0 & -q_3 & -q_2 & 0 & 0 & 0 \\ -q_0 & q_1 & q_2 & -q_3 & 0 & 0 & 0 \end{pmatrix}$$

This matrix is now utilised like the normal measurement matrix, meaning that given the accelerometer sensor variance $\sigma^2 = (\sigma_0^2, \sigma_1^2, \sigma_2^2)^T$, the Kalman gain K_a can now be computed as follows:

$$K_a = P H_a^T (H_a P H_a^T + R)^{-1}, \text{ where } R = \begin{pmatrix} \sigma_0^2 & 0 & 0 \\ 0 & \sigma_1^2 & 0 \\ 0 & 0 & \sigma_2^2 \end{pmatrix}$$

This results in the classic Kalman gain that is then multiplied with the measurement difference $z - \hat{z}$ and added to the state x to perform the update step:

$$\begin{aligned} x_{k+1} &= x_k + K_a(z - \hat{z}), \text{ where } z = \frac{g}{\|g\|} \\ P_{k+1} &= P_k - K_a H_a P_k \end{aligned}$$

Here, g is the gravity vector. The update step described above is not able to correct the yaw axis, as it is parallel to gravity and therefore remains constant for all yaw changes. This was overcome by adding the magnetic field vector into the equations, again with the added benefit of the filter being automatically calibrated towards north. To calculate the device heading the measured field vector $m \in \mathbb{R}^3$ had to be transferred from the sensor's reference frame to earth by rotating it around the state quaternion q, resulting in m':

$$m' = \mathbb{I} - 2 \begin{pmatrix} q_1^2 - q_2^2 & q_0 q_1 - q_2 q_3 & q_0 q_2 + q_1 q_3 \\ q_0 q_1 + q_2 q_3 & q_0^2 - q_2^2 & q_1 q_2 - q_0 q_3 \\ q_0 q_2 - q_1 q_3 & q_1 q_2 + q_0 q_3 & q_0^2 - q_1^2 \end{pmatrix} m$$

The rotated vector now was projected onto earth's horizon plane, giving the two dimensional heading vector h:

$$h = \begin{pmatrix} m'_0 \\ m'_1 \end{pmatrix}$$

The yaw angle ψ can now be calculated using the $\text{atan}(y, x)$ function defined above. Note that the system is referenced to north, meaning that the yaw angle is always the same as the aircraft's heading, reducing the complexity:

$$\psi = \text{atan}(h_1, h_0)$$

The new measurement $\hat{z}$ is computed similar to above, by converting the quaternion into the gravity vector, but with the addition of the magnetic heading ψ, that is computed from the quaternion as follows:

$$\hat{z} = \begin{pmatrix} -2(q_1 q_3 - q_0 q_2) \\ -2(q_0 q_1 + q_2 q_3) \\ q_0^2 - q_1^2 - q_2^2 + q_3^2 \\ \operatorname{atan}(2q_1 q_2 + 2q_0 q_3, 2q_0^2 + 2q_1^2 - 1) \end{pmatrix}$$

Adding another dimension to the measurements z and $\hat{z}$ also demands an updated measurement Jacobian H_m:

$$H_m = \begin{pmatrix} 2q_2 & -2q_1 & -2q_0 & -4q_0^2 q_3 + 4q_1^2 q_3 - 2q_3 - 8q_1 q_2 q_0 \\ -2q_3 & -2q_0 & 2q_1 & 4q_2 q_0^2 - 4q_1^2 q_2 - 2q_2 - 8q_1 q_0 q_3 \\ 2q_0 & -2q_3 & 2q_2 & 2q_1(2q_1 q_1 + 2q_0 q_0 - 1) \\ -2q_1 & -2q_2 & -2q_3 & 2q_0(2q_0 q_0 + 2q_1 q_1 - 1) \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}^T$$

To get the measurement noise this time the variance from the accelerometer σ_a^2 and magnetometer σ_m^2 had to be combined with the gravity vector g magnetic field m as follows:

$$\begin{aligned} \theta &= \arcsin(g_0) \\ \phi &= \arcsin\left(-\frac{g_1}{\cos(\theta)}\right) \end{aligned}$$

This is implemented into

$$R_m = M \begin{pmatrix} \sigma_{a0}^2 & 0 & 0 & 0 & 0 & 0 \\ 0 & \sigma_{a1}^2 & 0 & 0 & 0 & 0 \\ 0 & 0 & \sigma_{a2}^2 & 0 & 0 & 0 \\ 0 & 0 & 0 & \sigma_{m0}^2 & 0 & 0 \\ 0 & 0 & 0 & 0 & \sigma_{m0}^2 & 0 \\ 0 & 0 & 0 & 0 & 0 & \sigma_{m0}^2 \end{pmatrix} M^T$$

where

$$M = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & (-\cos(\theta)(m_2 \sin(\phi) - m_1 \cos(\phi)) \\ 0 & 0 & 0 & (-m_2 \cos(\phi)^2 \sin(\theta) - m_0 \cos(\theta) \cos(\phi) - m_2 \sin(\phi) \sin(\theta)) \\ 0 & 0 & 0 & (m_0 \sin\phi \cos(\theta + m_1 \sin(\phi)^2 \sin(\theta) + m_1 \cos(\phi)^2 \sin(\theta)) \end{pmatrix}^T$$

The result can now be used to compute the new Kalman gain:

$$K_m = P H_m^T (H_m P H_m^T + R_m)^{-1}$$

This gives the proposed algorithm two different update functions, making the filter more flexible. As most magnetometers have a slower measurement rate than gyroscopes or accelerometers due to physical limitations it is now possible to choose an update routine based on the available data rather than waiting for it, resulting in a good performing attitude estimator. (see Figure 4.5, an implemetation can be found at 6.5 and 6.6).

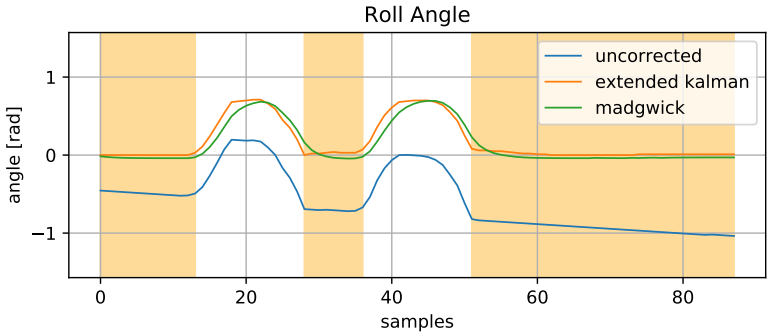

Figure 4.5: The system was put on a flat table and then tilted rapidly for two times. The orange sections indicate when the sensor was laying flat on the surface, serving as a reference for the roll, where $\phi = 0$. The blue line serves to demonstrate the system drift without any correction. Here the proposed algorithm was also compared to one proposed by S. Madgwick. His approach is based on a gradient-descent algorithm, that gradually reduces the accumulated sensor error.

4.2 Control

As described in Section 2.6 the aircraft is able to change its attitude by deflecting the control surfaces. As these do not correspond directly to the desired outcome a control algorithm had to be implemented. A classic optimal control problem can be described as a plant (the aircraft) that has a state $x \in \mathbb{R}^n$. Every state x_k at timestep k can be derived from the previous one x_{k-1} using a transfer function, represented by a matrix $A \in \mathbb{R}^{n \times n}$ and unknown external noise e:

$$x_{k+1} = Ax_k + e_k \tag{4.17}$$

The plant can be influenced by applying action $u \in \mathbb{R}^m$. This action affects the plant state x in a way specified by matrix $B \in \mathbb{R}^{n \times m}$. Therefore a discrete state space equation can be formed using the known plant dynamics A and B:

$$x_{k+1} = Ax_k + Bu_k + e_k \tag{4.18}$$

The goal is now to find a policy that, given the current state x_k results in the action u_k that drives the following plant states closest to a specified target y over a minimal amount of iterations. This is done by a closed loop controller (see Figure 4.6), meaning that it gets the current plant state as direct feedback to generate an appropriate response. To achieve this two options where explored: Model Predictive Control (see [21]) and a modified version of Proportional Integral Derivative Control (see [19]).

4.2.1 Model Predictive Control

Predictive controllers try to predict the plant's behavior using the known dynamics A and B to generate an efficient state-trajectory towards the desired outcome. For this a prediction horizon h is set, that limits the iteration steps to project from every real plant state. The control problem can also be further constrained by placing restrictions on the controller output. This can be done to prevent the output from potentially damaging the plant. In the case of an aircraft this could be over-deflection of the control surfaces that would damage the hinges or abrupt changes that would pose an unnecessary load on the airframe.

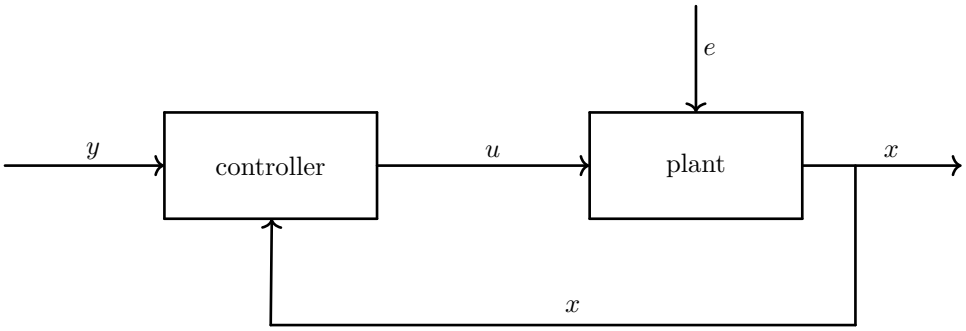

Figure 4.6: A closed feedback loop of a state-space system. Here $y \in \mathbb{R}^m$ is the target state, $u \in \mathbb{R}^n$ the controller output, $x\mathbb{R}^m$ the actual system state and e indicates external influences on the system. The plant is the system to be controlled and is described with the state space dynamics $A \in \mathbb{R}^{m \times m}$ and $B \in \mathbb{R}^{m \times n}$.

The working principle of predictive controllers can now be illustrated on a tree diagram (see Figure 4.7). The tree starts with the current state x_k and then leads to an amount of possible future states $x^0_{k+1}, ...x^n_{k+1}$ (note that the superscript serves as an index and not as an operator). Said amount then expands into even more possible states, that are reachable with their associated action u^n and the iteration equation (4.18).

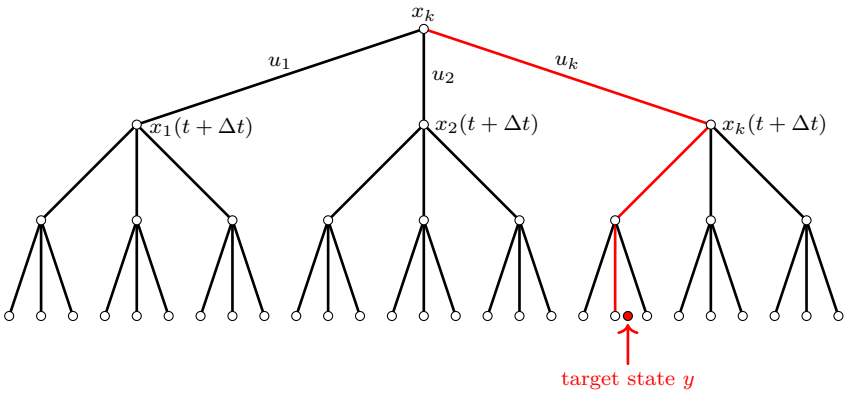

Figure 4.7: Prediction tree with the best possible branch (in red).

To quantify the error the plant would generate over a branch of the tree a cost function f has to be defined. This function ranks every predicted state x on the branch on how close it resembles the target state y. Finally the branch of the closest x is chosen with the corresponding sequence of u's that also fit any possible restrictions. A possible example of such a cost function is:

$$f(x) = |x - y| \tag{4.19}$$

Said function is often enhanced by assigning fixed weights to individual components, represented by a vector $w \in \mathbb{R}^n$. This implementation allows the controller to prioritise certain aspects of the system state, such as ones that are more relevant to the desired plant behaviour. Hence, equation (4.19) is modified as follows:

$$f(x) = \sum_{i=1}^{n} w_i |x_i - y_i| \tag{4.20}$$

After finding a matching system trajectory that best minimises the cost function its root action u is used as the actual controller output (see [21])., while the rest is discarded and computed again in the next step.

The main advantage of the system is, that very complex plants even with cross-dependencies between different state components can be modelled. Furthermore a large prediction horizon is able to prevent overshoots, oscillation or the system being trapped inside a local cost-minimum. The biggest downside to deploying this sort of algorithm however is the computational load and complexity, especially considering the limited hardware of the proposed model airplane.

4.2.2 Proportional Integral Derivative Controller

The classic Proportional Integral Derivative (PID) controller is one of the most common algorithms used for closed-loop system control. Compared to more complex approaches it is only applicable to individual components x_k of the plant state $x \in \mathbb{R}^n$. Therefore it is not suitable for systems that feature strong cross-dependencies between different components of the state vector. For simplicity, we now suppress the index and just write x instead of x_k. The algorithm works by applying a cost function f to the state component x with respect to its corresponding target y to calculate the current plant error. In most cases this is done by simply computing the difference. In this implementation however the difference was additionally squared:

$$f(x) = (x - y)^2. \tag{4.21}$$

The reason for taking the square instead of the absolute value of the difference ist twofold: lager errors are are punished more severely, and the square function is differentiable everywhere. The controller output u at time t is then calculated by taking the weighted sum of the current error, the derivative of the cost function with respect to the time t and the integrated error since $t = 0$. The corresponding tuning parameters are real numbers K_D, K_I and K_D:

$$u(t) = K_P f(x(t)) + K_I \int_0^t f(x(\tau))d\tau + K_D \frac{d}{dt} f(x(t)). \tag{4.22}$$

The values of x in the time interval $[0, t]$ are results of the real-world plant, hence the value of $u(t)$ cannot be expressed mathematically since the values of x are only available in the measurement time steps

$$t_i = i\Delta t, \quad i = 0, 1, \ldots, k, \text{ with } t = t_k.$$

The integration and the derivative in (4.22) are then approximated by a Riemann sum and a difference quotient:

$$\int_0^t f(x(\tau))d\tau \approx \sum_{i=1}^{k} f(x(t_i))\Delta t \tag{4.23}$$

$$\frac{d}{dt} f(x(t)) \approx \frac{f(x(t_k)) - f(x(t_{k-1}))}{\Delta t} \tag{4.24}$$

The different weights K_P, K_I, K_D are used to tune the controller to the desired behavior. K_P describes the controller's proportional response to the current plant error. K_I is used to reduce error buildup that would be caused by the proportional part never reaching the target state. This would lead to a steady increase of the error sum, giving the output more authority than just the proportional. Finally K_D is implemented to prevent overshoots. By

tracking the change rate of the current error the controller can predict how the plant is going to respond to a given u and can therefore be used to dampen the proportional and integral components that would otherwise result in the system surpassing the target value, possibly resulting in an oscillation.

As the airplane is constantly adjusting target y the controller had to be modified to provide better responses. Rapid changes of the target value can lead to a situation called integral buildup. As it is impossible for the plant to instantly reach the exact target value the integrated error is increased at every target change. If this is repeated frequently the built-up integrator slowly begins to corrupt the controller output. This can be overcome by resetting the integral at every target adjustment. With a sufficiently high change rate however, one can completely neglect the integrator part, as the frequent resets never allow the integral to reach values of any significance. Therefore the error integrator was removed from the proposed controller design. To optimise the controller further for fast target tracking a feed forward path was implemented. This adds the current weighed state directly to the output $u(t)$:

$$u(t) = K_P f(x(t)) + K_D \frac{d}{dt} f(x(t)) + K_F x(t) \tag{4.25}$$

(see also [19]). In case of an airplane this can be interpreted as the pilot directly applying controls to achieve a certain outcome and not just doing so after the plane deviated from its intended path.

4.2.3 Implementation

After some experiments the results of the Model Predictive Controller where not able to justify the added complexity compared to the simple PDFF Controller (see [23]). This is further supported by the fact that an airplane's pitch and roll angles are not sufficiently codependent and therefore allow component individual controllers. As control surface deflection causes a force to act on the airframe, the resulting angular velocity around the corresponding axis was used as target variable, forming a single order control problem. This results in two PDFF controllers controlling the plane's roll and pitch rates by deflecting the ailerons and elevator. This results in the ground commander now being able to control the turn rates of the aircraft instead of directly deflecting its control surfaces. To manage the rudder a similar approach was taken. As described above the rudder is mainly used to keep sideslip to a minimum. Therefore the PDFF Controller's target value was set to 0 and the feedback system state the linear acceleration along the pitch axis. This would allow for fully automatic rudder control, even while performing coordinated turns. To control the plane over an absolute attitude and not just angular velocities a simplified part of the model predictive controller was implemented. It consisted of the controller doing a prediction x_{pred} based on the current attitude x, angular rate x' and the prediction horizon h.

$$x_{\text{pred}} = x + x' \cdot h \tag{4.26}$$

From this predicted position one can calculate the angular velocity x'_{dem} needed to instead reach the target value y at the prediction horizon h. Here h is another unable parameter that represents the airplane's response time.

$$x'_{\text{dem}} = \frac{y - x_{\text{pred}}}{h} \tag{4.27}$$

x'_{dem} can now be fed to the PDFF controller as a target value, reducing this second order control problem to a first degree one. One additional benefit of the approach above is its

tangential nature as it approaches a constant target value. This has a damping effect on the plane while tracking a certain attitude, greatly reducing unwanted oscillation, while still being reactive to externally induced positional errors (see Figure 4.8).

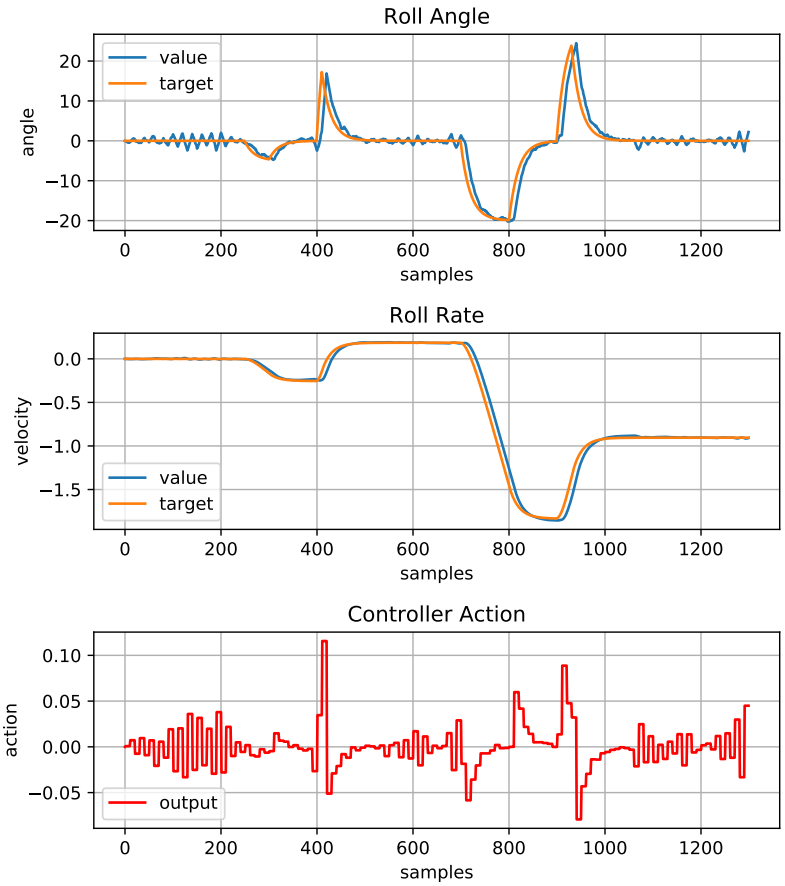

Figure 4.8: This diagram shows the position tracking servo inside a simulation with artificially added turbulence.

This forms the attitude servo system of the FBW-controller. The pilot can command the aircraft's attitude over the turn rates and the system tracks the resulting angle. This allows the controller to limit the attitude to safe angles and also allows the plane to maintain the given pitch and roll even while experiencing external forces without any additional pilot input.

5 Conclusion

This finalises the project, resulting in a working model airplane that was used to develop a fly by wire system. The system consisted of an altimeter, accelerometer and AHRS system, forming a basic inertial navigation system. The implemented filters are flexible and easily adaptable to include more state variables and support redundant sensors. The controller is able to control the flight control surfaces, that have all been motorised, accepting either the desired attitude angles or turning rates for roll and pitch as an input. This forms a good foundation to add a full autopilot to the system. The main requirement for this would be the addition of a GPS receiver giving the plane absolute positioning capability and also providing additional altitude data that could be fused into the altimeter. The main challenge of the project was its broad range of different topics. It was quite difficult to cover and document every subject in enough detail to get acceptable results, especially given the fact, that most of the fields involved are not covered in the school curriculum, requiring extensive research.

6 Appendix: Code

This chapter contains a selection of the most important pieces of code that where used for testing and in the flight controller.

```
#ifndef SBUS_h
#define SBUS_h

#include "Arduino.h"

#define SBUS_BAUD     100000
#define SBUS_TIMEOUT  7000
#define SBUS_HEADER   0x0F
#define SBUS_END      0x00
#define SBUS2_END     0x04
#define SBUS_PAYLOAD  24
#define SBUS_FAILSAFE 0x08
#define SBUS_LOSS     0x04
#define SBUS_CHANNELS 16

class SBUS{
  public:
    SBUS(HardwareSerial& bus);
    void start();
    bool read(uint16_t* channels, bool* failsafe, bool* lost_packet);

  private:
    HardwareSerial& bus;
    bool terminated;
    uint8_t buffer[SBUS_PAYLOAD];

    bool parse();
};

#endif
```

Code 6.1: The header file of the custom SBus implementation.

```
#include "SBUS.h"

SBUS::SBUS(HardwareSerial& bus){
  this->bus = &bus;
}

void SBUS::start(){
  this->index = 0;
  this->bus->begin(SBUS_BAUD, SERIAL_8E2, 16, 17, true, 1000);
}

bool SBUS::parse(){
```

```
13    // check for timeout
14    if((micros() - this->ts_prev) > SBUS_TIMEOUT){
15      this->index = 0;
16    }
17
18    // read uart bytes
19    while(this->bus->available() > 0){
20      this->ts_prev = micros();
21      uint8_t current = this->bus->read();
22
23      if(this->index == 0){
24        // find header
25        if(current == SBUS_HEADER && this->terminated){
26          this->index += 1;
27        }
28      }else{
29        uint8_t pointer = this->index - 1;
30
31        if(pointer < SBUS_PAYLOAD){
32          this->buffer[pointer] = current;
33          this->index++;
34
35        }else{
36          this->index = 0;
37          this->terminated = (current == SBUS_END || (current & 0x0F) ==
      SBUS2_END);
38          return this->terminated && (pointer == SBUS_PAYLOAD);
39        }
40      }
41
42      this->terminated = (current == SBUS_END || (current & 0x0F) == SBUS2_END)
      ;
43    }
44
45    // incomplete packet
46    return false;
47  }
48
49  bool SBUS::read(uint16_t* channels, bool* failsafe, bool* lost_packet){
50    if(parse()){
51
52      // receiver state info
53      if(failsafe){
54        *failsafe = this->buffer[22] & SBUS_FAILSAFE;
55      }
56
57      if(lost_packet){
58        *failsafe = this->buffer[22] & SBUS_LOSS;
59      }
60
61      // split payload into 11-bit values
62      if(channels){
63        channels[0]  = (uint16_t) ((this->buffer[0] | this->buffer[1] << 8) & 0
      x07FF);
64        channels[1]  = (uint16_t) ((this->buffer[1] >> 3 | this->buffer[2] <<
      5) & 0x07FF);
65        channels[2]  = (uint16_t) ((this->buffer[2] >> 6 | this->buffer[3] << 2
       |this->buffer[4] << 10) & 0x07FF);
66        channels[3]  = (uint16_t) ((this->buffer[4] >> 1 | this->buffer[5] <<
      7) & 0x07FF);
67        channels[4]  = (uint16_t) ((this->buffer[5] >> 4 | this->buffer[6] <<
      4) & 0x07FF);
```

```
68        channels[5]  = (uint16_t) ((this->buffer[6] >> 7 | this->buffer[7] << 1
      | this->buffer[8] << 9) & 0x07FF);
69        channels[6]  = (uint16_t) ((this->buffer[8] >> 2 | this->buffer[9] <<
     6) & 0x07FF);
70        channels[7]  = (uint16_t) ((this->buffer[9] >> 5 | this->buffer[10]<<
     3) & 0x07FF);
71        channels[8]  = (uint16_t) ((this->buffer[11] | this->buffer[12] << 8) &
      0x07FF);
72        channels[9]  = (uint16_t) ((this->buffer[12] >> 3| this->buffer[13] <<
     5) & 0x07FF);
73        channels[10] = (uint16_t) ((this->buffer[13] >> 6| this->buffer[14] <<
     2 | this->buffer[15] << 10) & 0x07FF);
74        channels[11] = (uint16_t) ((this->buffer[15] >> 1| this->buffer[16] <<
     7) & 0x07FF);
75        channels[12] = (uint16_t) ((this->buffer[16] >> 4| this->buffer[17] <<
     4) & 0x07FF);
76        channels[13] = (uint16_t) ((this->buffer[17] >> 7| this->buffer[18] <<
     1 | this->buffer[19] << 9) & 0x07FF);
77        channels[14] = (uint16_t) ((this->buffer[19] >> 2| this->buffer[20] <<
     6) & 0x07FF);
78        channels[15] = (uint16_t) ((this->buffer[20] >> 5| this->buffer[21] <<
     3) & 0x07FF);
79      }
80      return true;
81    }
82    return false;
83  }
```

Code 6.2: The implementation of the SBus software interface as described in Section 3.2. It is an improved and optimised version of the code by B. Taylor from Bolderflight.

```
1  #ifndef ADVMATH_h
2  #define ADVMATH_h
3
4  // set memory range to 0x00
5  void clear(float* in, int length);
6
7  // clone memory range
8  void clone(float* in, int length, float* out);
9
10 // add memory ranges
11 void add(float* in_a, float* in_b, int length, float* out);
12
13 // subtract memory ranges
14 void subtract(float* in_a, float* in_b, int length, float* out);
15
16 // subtract range from identity matrix [size x size]
17 void idSubtract(float* in, int size, float* out);
18
19 // multiply memory range with scalar
20 void scale(float* in, float value, int length, float* out);
21
22 // normalise memory range
23 void normalise(float* in, int length, float* out);
24
25 // store transposed copy of matrix [m x n]
26 void transpose(float* in, int m, int n, float* out);
27
28 // matrix multiplication of [m x c] and [c x n], stores copy
29 void multiply(float* in_a, float* in_b, int m, int c, int n, float* out);
30
31 // matrix multiplication of [m x c] and [n x c]^T, stores copy
```

```
32 void transMultiply(float* in, float* in_t, int m, int c, int n, float* out);
33
34 // invert matrix [size x size]
35 bool invert(float* in, int size, float* out);
36
37 // compare memory ranges
38 bool equals(float* in_a, float* in_b, int length);
39
40 // convert quaternion to euler angles
41 void getEuler(float* quat, float* euler);
42
43 // convert euler angles to quaternion
44 void getQuat(float* euler, float* quat);
45
46 // quaternion multiplication, stores copy
47 void multiplyQuat(float* p, float* q, float* out);
48
49 // rotate vector by quaternion, stores copy
50 void axisRotate(float* vector, float* quat, float* out);
51
52 // angle difference, clamps PI * 2
53 float enclosed(float a, float b);
54
55 #endif
```

Code 6.3: The header file of the custom vector math library.

```
1  #include "Arduino.h"
2  #include "AdvMath.h"
3
4  void clear(float* in, int length){
5    memset(in, 0x00, length * 4);
6  }
7
8  void clone(float* in, int length, float* out){
9    memcpy(out, in, length * 4);
10 }
11
12 void add(float* in_a, float* in_b, int length, float* out){
13   for(int i = 0; i < length; i++){
14     out[i] = in_a[i] + in_b[i];
15   }
16 }
17
18 void subtract(float* in_a, float* in_b, int length, float* out){
19   for(int i = 0; i < length; i++){
20     out[i] = in_a[i] - in_b[i];
21   }
22 }
23
24 void idSubtract(float* in, int size, float* out){
25   for(int i = 0; i < size * size; i++){
26     out[i] = (i / size == i % size ? 1.0 : 0.0) - in[i];
27   }
28 }
29
30 void scale(float* in, float value, int length, float* out){
31   for(int i = 0; i < length; i++){
32     out[i] = in[i] * value;
33   }
34 }
35
```

```
36 void normalise(float* in, int length, float* out){
37   float mag_sq = 0.0;
38
39   for(int i = 0; i < length; i++){
40     mag_sq += in[i] * in[i];
41   }
42
43   scale(in, 1.0 / sqrt(mag_sq), length, out);
44 }
45
46 void transpose(float* in, int m, int n, float* out){
47   for(int i = 0; i < m; i++){
48     for(int j = 0; j < n; j++){
49       out[m * j + i] = in[n * i + j];
50     }
51   }
52 }
53
54 void multiply(float* in_a, float* in_b, int m, int c, int n, float* out){
55   for(int i = 0; i < m; i++){
56     for(int j = 0; j < n; j++){
57       out[n * i + j] = 0.0;
58
59       for(int k = 0; k < c; k++){
60         out[n * i + j] += in_a[c * i + k] * in_b[n * k + j];
61       }
62     }
63   }
64 }
65
66 void transMultiply(float* in, float* in_t, int m, int c, int n, float* out){
67   for(int i = 0; i < m; i++){
68     for(int j = 0; j < n; j++){
69       out[n * i + j] = 0.0;
70
71       for(int k = 0; k < c; k++){
72         out[n * i + j] += in[c * i + k] * in_t[c * j + k];
73       }
74     }
75   }
76 }
77
78 bool invert(float* in, int size, float* out){
79   int pivrow = 0;
80   int pivrows[size];
81
82   clone(in, size * size, out);
83
84   for(int k = 0; k < size; k++){
85     float value = 0;
86     for(int i = k; i < size; i++){
87       float entry = fabs(out[i * size + k]);
88
89       if(entry >= value){
90         value = entry;
91         pivrow = i;
92       }
93     }
94
95     // singular matrix check
96     if(out[pivrow * size + k] == 0.0){
97       return false;
```

```
98      }
99
100     if(pivrow != k){
101       for(int j = 0; j < size; j++){
102         float tmp = out[k * size + j];
103         out[k * size + j] = out[pivrow * size + j];
104         out[pivrow * size + j] = tmp;
105       }
106     }
107
108     pivrows[k] = pivrow;
109
110     float inv = 1.0 / out[k * size + k];
111     out[k * size + k] = 1.0;
112
113     for(int j = 0; j < size; j++){
114       out[k * size + j] *= inv;
115     }
116
117     for(int i = 0; i < size; i++){
118       if(i != k){
119         float tmp = out[i * size + k];
120         out[i * size + k] = 0.0;
121         for(int j = 0; j < size; j++){
122           out[i * size + j] -= out[k * size + j] * tmp;
123         }
124       }
125     }
126   }
127
128   for(int k = size - 1; k >= 0; k--){
129     if(pivrows[k] != k){
130       for(int i = 0; i < size; i++){
131         float tmp = out[i * size + k];
132         out[i * size + k] = out[i * size + pivrows[k]];
133         out[i * size + pivrows[k]] = tmp;
134       }
135     }
136   }
137
138   return true;
139 }
140
141 bool equals(float* in_a, float* in_b, int length){
142   for(int i = 0; i < length; i++){
143     if(in_a[i] != in_b[i]){
144       return false;
145     }
146   }
147
148   return true;
149 }
150
151 void getEuler(float* quat, float* euler){
152   euler[0] = atan2(2.0 * quat[2] * quat[3] + 2.0 * quat[0] * quat[1], 2.0 *
       quat[0] * quat[0] + 2.0 * quat[3] * quat[3] - 1.0);
153   euler[1] = asin(-2.0 * quat[1] * quat[3] + 2.0 * quat[0] * quat[2]);
154   euler[2] = atan2(2.0 * quat[1] * quat[2] + 2.0 * quat[0] * quat[3], 2.0 *
       quat[0] * quat[0] + 2.0 * quat[1] * quat[1] - 1.0);
155 }
156
157 void getQuat(float* euler, float* quat){
```

```
158   float cos_[3], sin_[3];
159
160   for(int i = 0; i < 3; i++){
161     cos_[i] = cos(euler[i] / 2.0);
162     sin_[i] = sin(euler[i] / 2.0);
163   }
164
165   quat[0] = cos_[2] * cos_[1] * cos_[0] + sin_[2] * sin_[1] * sin_[0];
166   quat[1] = cos_[2] * cos_[1] * sin_[0] - sin_[2] * sin_[1] * cos_[0];
167   quat[2] = cos_[2] * sin_[1] * cos_[0] + sin_[2] * cos_[1] * sin_[0];
168   quat[3] = sin_[2] * cos_[1] * cos_[0] - cos_[2] * sin_[1] * sin_[0];
169 }
170
171 void multiplyQuat(float* p, float* q, float* out){
172
173   out[0] = p[0] * q[0] - (p[1] * q[1] + p[2] * q[2] + p[3] * q[3]);
174   out[1] = p[0] * q[1] + q[0] * p[1] + p[2] * q[3] - p[3] * q[2];
175   out[2] = p[0] * q[2] + q[0] * p[2] + p[3] * q[1] - p[1] * q[3];
176   out[3] = p[0] * q[3] + q[0] * p[3] + p[1] * q[2] - p[2] * q[1];
177 }
178
179 float enclosed(float a, float b){
180   float angle = fmod(a - b, PI * 2);
181
182   if(angle >= PI){
183     angle -= (PI * 2);
184   }
185
186   return angle;
187 }
188
189 void axisRotate(float* vector, float* quat, float* out){
190   float r = quat[0];
191   float i = quat[1];
192   float j = quat[2];
193   float k = quat[3];
194
195   out[0] = 2 * (r * vector[2] * j + i * vector[2] * k - r * vector[1] * k + i
        * vector[1] * j) + vector[0] * (r * r + i * i - j * j - k * k);
196   out[1] = 2 * (r * vector[0] * k + i * vector[0] * j - r * vector[2] * i + j
        * vector[2] * k) + vector[1] * (r * r - i * i + j * j - k * k);
197   out[2] = 2 * (r * vector[1] * i - r * vector[0] * j + i * vector[0] * k + j
        * vector[1] * k) + vector[2] * (r * r - i * i - j * j + k * k);
198 }
```

Code 6.4: The implementation of the custom vector math library. It was especially designed to be computation and memory efficient. This was done by utilising pointers to the memory locations containing the vectors and matrices. The code is mostly self written with a few exceptions of code ported from other projects.

```
1  #ifndef AHRS_h
2  #define AHRS_h
3
4  #include "AdvMath.h"
5  #include "Arduino.h"
6
7  class AHRS{
8  public:
9    void initialise(uint32_t sample_time);
10   bool update(float* gyr, float* acc, float* mag);
11
12   float getRoll();
```

```
13    float getPitch();
14    float getYaw();
15
16  private:
17
18    bool sample(float* gyr, float* acc, float* mag);
19    void setup();
20
21    void gyrUpdate(float dt, float* gyr);
22    void accUpdate(float* acc_raw);
23    void magUpdate(float* mag_raw, float* acc_raw);
24
25    // flow management
26    uint32_t init_time;
27    uint32_t ts_prev;
28    bool running;
29    bool started;
30
31    // used for sensor sampling
32    uint32_t gyr_count;
33    uint32_t acc_count;
34    uint32_t mag_count;
35    uint32_t quat_count;
36
37    float gyr_bias[3], gyr_m2[3];
38    float acc_mean[3], acc_m2[3];
39    float mag_mean[3], mag_m2[3];
40    float quat_mean[4], quat_m2[4];
41
42    float x_[7]; // filter state
43    float euler[3]; // euler rotation
44    float initial_euler[3]; // start reference
45
46    float P_[7 * 7]; // state uncertainty
47    float Q_[3 * 3]; // process noise
48    float Ra_[3 * 3];// sensor noise
49    float R_[6 * 6]; // sensor noise
50    float M_[4 * 6]; // output matrix
51    float Ma_[3 * 3];// output matrix
52
53    float K_[7 * 4]; // kalman gain
54    float Ka_[7 * 3];  // kalman gain
55  };
56
57  #endif
```

Code 6.5: The header file of the AHRS system. See Sections 3.3, 4.1.3, and 4.1.4.

```
1  #include "AHRS.h"
2  #include "AdvMath.h"
3  #include "Arduino.h"
4
5  void AHRS::initialise(uint32_t sample_time){
6    this->init_time = sample_time;
7    this->running = false;
8    this->started = false;
9
10   this->gyr_count = 0;
11   this->acc_count = 0;
12   this->mag_count = 0;
13   this->quat_count = 0;
14
```

```
15    clear(this->euler, 3);
16    clear(this->initial_euler, 3);
17
18    clear(this->gyr_bias, 3);    clear(this->gyr_m2, 3);
19    clear(this->acc_mean, 3);    clear(this->acc_m2, 3);
20    clear(this->mag_mean, 3);    clear(this->mag_m2, 3);
21    clear(this->quat_mean, 4);   clear(this->quat_m2, 4);
22  }
23
24  bool AHRS::update(float* gyr, float* acc, float* mag){
25    if(!this->running){
26      if(!this->started){
27        this->started = true;
28        this->ts_prev = micros();
29      }
30
31      if(this->sample(gyr, acc, mag) && (micros() - this->ts_prev) >= this->
        init_time){
32        this->setup();
33      }
34    }else{
35      uint32_t now = micros();
36      float dt = ((float)(now - this->ts_prev)) / 1000000.0f;
37      this->ts_prev = now;
38
39      this->gyrUpdate(dt, gyr);
40
41      if(acc != NULL){
42        if(mag != NULL){
43          this->magUpdate(mag, acc);
44        }else{
45          this->accUpdate(acc);
46        }
47      }
48
49      float norm[4], quat[4] = {this->x_[0], this->x_[1], this->x_[2], this->x_
        [3]};
50
51      normalise(quat, 4, norm);
52      getEuler(norm, this->euler);
53    }
54
55    return this->running;
56  }
57
58  void AHRS::setup(){
59    clone(this->quat_mean, 4, this->x_);
60    clone(this->gyr_bias, 3, &this->x_[4]);
61
62    /* store initial position */ {
63      float quat[4];
64      normalise(this->quat_mean, 4, quat);
65      getEuler(quat, this->initial_euler);
66    }
67
68    float gyr_variance[3], acc_variance[3], mag_variance[3], quat_variance[4];
69
70    scale(this->gyr_m2, 1.0f / (this->gyr_count - 1), 3, gyr_variance);
71    scale(this->acc_m2, 1.0f / (this->acc_count - 1), 3, acc_variance);
72    scale(this->mag_m2, 1.0f / (this->mag_count - 1), 3, mag_variance);
73    scale(this->quat_m2, 1.0f / (this->quat_count - 1), 4, quat_variance);
74
```

```
75    clear(this->P_, 7 * 7);
76    clear(this->Q_, 3 * 3);
77    clear(this->Ra_, 3 * 3);
78    clear(this->R_, 6 * 6);
79    clear(this->M_, 4 * 6);
80    clear(this->Ma_, 3 * 3);
81
82    // initialize state covariance
83    this->P_[0] = quat_variance[0];
84    this->P_[8] = quat_variance[1];
85    this->P_[16] = quat_variance[2];
86    this->P_[24] = quat_variance[3];
87    this->P_[32] = gyr_variance[0];
88    this->P_[40] = gyr_variance[1];
89    this->P_[48] = gyr_variance[2];
90
91    // gyroscope covariance matrix
92    this->Q_[0] = gyr_variance[0];
93    this->Q_[4] = gyr_variance[1];
94    this->Q_[8] = gyr_variance[2];
95
96    // accelerometer covariance matrix
97    this->Ra_[0] = acc_variance[0];
98    this->Ra_[4] = acc_variance[1];
99    this->Ra_[8] = acc_variance[2];
100
101   // accelerometer & magnetometer covariance matrix
102   this->R_[0] = acc_variance[0];
103   this->R_[7] = acc_variance[1];
104   this->R_[14] = acc_variance[2];
105   this->R_[21] = mag_variance[0];
106   this->R_[28] = mag_variance[1];
107   this->R_[35] = mag_variance[2];
108
109   // output matrix
110   this->M_[0] = 1.0f;
111   this->M_[7] = 1.0f;
112   this->M_[14] = 1.0f;
113   this->Ma_[0] = 1.0f;
114   this->Ma_[4] = 1.0f;
115   this->Ma_[8] = 1.0f;
116
117   this->ts_prev = micros();
118   this->running = true;
119 }
120
121 bool AHRS::sample(float* gyr, float* acc, float* mag){
122   if(gyr != NULL){
123     this->gyr_count += 1;
124
125     for(int i = 0; i < 3; i++){
126       float delta = gyr[i] - this->gyr_bias[i];
127       this->gyr_bias[i] += delta / ((float) this->gyr_count);
128       this->gyr_m2[i] += delta * (gyr[i] - this->gyr_bias[i]);
129     }
130   }
131
132   if(acc != NULL){
133     this->acc_count += 1;
134
135     for(int i = 0; i < 3; i++){
136       float delta = acc[i] - this->acc_mean[i];
```

```
137       this->acc_mean[i] += delta / ((float) this->acc_count);
138       this->acc_m2[i] += delta * (acc[i] - this->acc_mean[i]);
139     }
140   }
141
142   if(mag != NULL){
143     this->mag_count += 1;
144
145     for(int i = 0; i < 3; i++){
146       float delta = mag[i] - this->mag_mean[i];
147       this->mag_mean[i] += delta / ((float) this->mag_count);
148       this->mag_m2[i] += delta * (mag[i] - this->mag_mean[i]);
149     }
150   }
151
152   if(acc != NULL && mag != NULL){
153     this->quat_count += 1;
154
155     // estimate the quaternion for initial state covariance
156     float acc_[3], mag_[3], euler[3], quat[4];
157     normalise(acc, 3, acc_);
158     normalise(mag, 3, mag_);
159
160     euler[1] = asin(acc_[0]);
161     euler[0] = asin(-acc_[1] / (cos(euler[1])));
162     euler[2] = atan2(mag_[2] * sin(euler[0]) - mag_[1] * cos(euler[0]),
163       mag_[0] * cos(euler[1]) + mag_[1] * sin(euler[1]) * sin(euler[0]) +
    mag_[2] * sin(euler[1]) * cos(euler[0]));
164
165     getQuat(euler, quat);
166
167     for(int i = 0; i < 4; i++){
168       float delta = quat[i] - this->quat_mean[i];
169       this->quat_mean[i] += delta / ((float) this->quat_count);
170       this->quat_m2[i] += delta * (quat[i] - this->quat_mean[i]);
171     }
172   }
173
174   //return if enough samples are present
175   return (this->gyr_count > 2 && this->acc_count > 2 && this->mag_count > 2
    && this->quat_count > 2);
176 }
177
178 void AHRS::gyrUpdate(float dt, float* gyr){
179   float dx = 0.5f * dt * (gyr[0] - this->x_[4]);
180   float dy = 0.5f * dt * (gyr[1] - this->x_[5]);
181   float dz = 0.5f * dt * (gyr[2] - this->x_[6]);
182
183   float dx0 = 0.5f * dt * this->x_[0];
184   float dx1 = 0.5f * dt * this->x_[1];
185   float dx2 = 0.5f * dt * this->x_[2];
186   float dx3 = 0.5f * dt * this->x_[3];
187
188   // state transition matrix
189   float F[7 * 7];
190   clear(F, 49);
191
192   F[0] = 1.0f;   F[1] = -dx;    F[2] = -dy;    F[3] = -dz;    F[4] = dx1;
    F[5] = dx2;    F[6] = dx3;
193   F[7] = dx;     F[8] = 1.0f;   F[9] = dz;     F[10] = -dy;   F[11] = -dx0;
    F[12] = -dx2;  F[13] = dx3;
194   F[14] = dy;    F[15] = -dz;   F[16] = 1.0f;  F[17] = dx;    F[18] = -dx3;
```

```
    F[19] = -dx0;  F[20] = dx1;
195 F[21] = dz;    F[22] = dy;    F[23] = -dx;   F[24] = 1.0f;  F[25] = dx2;
    F[26] = -dx1;  F[27] = dx0;
196
197 F[32] = 1.0f;  F[40] = 1.0f;  F[48] = 1.0f;
198
199 // process devia
200 float L[7 * 3];
201 clear(L, 21);
202
203 L[0] = -dx1;   L[1] = -dx2;   L[2] = -dx3;
204 L[3] = dx0;    L[4] = -dx3;   L[5] = dx2;
205 L[6] = dx3;    L[7] = dx0;    L[8] = -dx1;
206 L[9] = -dx2;   L[10] = dx1;   L[11] = dx0;
207 L[12] = 1.0f;  L[16] = 1.0f;  L[20] = 1.0f;
208
209 /* P = F * P * F.T + L * Q * L.T */ {
210   float tmp_a[49], tmp_b[49], tmp_c[21];
211
212   multiply(F, this->P_, 7, 7, 7, tmp_b);
213   transMultiply(tmp_b, F, 7, 7, 7, tmp_a);
214
215   multiply(L, this->Q_, 7, 3, 3, tmp_c);
216   transMultiply(tmp_c, L, 7, 3, 7, tmp_b);
217
218   add(tmp_a, tmp_b, 49, this->P_);
219 }
220
221 /* x = F * x */ {
222   float tmp[7];
223
224   multiply(F, this->x_, 7, 7, 1, tmp);
225   clone(tmp, 7, this->x_);
226 }
227 }
228
229 void AHRS::accUpdate(float* acc_raw){
230 float acc[3];
231 normalise(acc_raw, 3, acc);
232
233 // measurement jacobian
234 float Ha[3 * 7];
235 clear(Ha, 21);
236
237 float dq0 = 2.0f * this->x_[0];
238 float dq1 = 2.0f * this->x_[1];
239 float dq2 = 2.0f * this->x_[2];
240 float dq3 = 2.0f * this->x_[3];
241
242 Ha[0] = dq2;    Ha[1] = -dq3;  Ha[2] = dq0;   Ha[3] = -dq1;
243 Ha[7] = -dq1;   Ha[8] = -dq0;  Ha[9] = -dq3;  Ha[10] = -dq2;
244 Ha[14] = -dq0;  Ha[15] = dq1;  Ha[16] = dq2;  Ha[17] = -dq3;
245
246 /* Ka = P * Ha.T * (Ha * P * Ha.T + Ma * Ra * Ma.T)^-1 */ {
247   float tmp_a[21], tmp_b[9], tmp_c[9];
248
249   multiply(Ha, this->P_, 3, 7, 7, tmp_a);
250   transMultiply(tmp_a, Ha, 3, 7, 3, tmp_b);
251
252   multiply(this->Ma_, this->Ra_, 3, 3, 3, tmp_a);
253   transMultiply(tmp_a, this->Ma_, 3, 3, 3, tmp_c);
254
```

```
255     add(tmp_b, tmp_c, 9, tmp_a);
256     invert(tmp_a, 3, tmp_b);
257
258     transMultiply(this->P_, Ha, 7, 7, 3, tmp_a);
259     multiply(tmp_a, tmp_b, 7, 3, 3, this->Ka_);
260   }
261
262   /* x = x + Ka * (acc - ha) */ {
263     float ha[3], tmp[7];
264
265     float sq0 = this->x_[0] * this->x_[0];
266     float sq1 = this->x_[1] * this->x_[1];
267     float sq2 = this->x_[2] * this->x_[2];
268     float sq3 = this->x_[2] * this->x_[2];
269
270     ha[0] = acc[0] - (-2.0f * (this->x_[1] * this->x_[3] - this->x_[0] * this
      ->x_[2]));
271     ha[1] = acc[1] - (-2.0f * (this->x_[0] * this->x_[1] + this->x_[2] * this
      ->x_[3]));
272     ha[2] = acc[2] + sq0 - sq1 - sq2 + sq3;
273
274     multiply(this->Ka_, ha, 7, 3, 1, tmp);
275     add(this->x_, tmp, 7, this->x_);
276   }
277
278   /* P = (I - Ka * Ha) * P */ {
279     float tmp_a[49], tmp_b[49];
280
281     multiply(this->Ka_, Ha, 7, 3, 7, tmp_a);
282     idSubtract(tmp_a, 7, tmp_a);
283     multiply(tmp_a, this->P_, 7, 7, 7, tmp_b);
284
285     clone(tmp_b, 7 * 7, this->P_);
286   }
287 }
288
289 void AHRS::magUpdate(float* mag_raw, float* acc_raw){
290   float acc[4], mag[3];
291   normalise(acc_raw, 3, acc);
292   normalise(mag_raw, 3, mag);
293
294   float dq0 = 2.0f * this->x_[0];
295   float dq1 = 2.0f * this->x_[1];
296   float dq2 = 2.0f * this->x_[2];
297   float dq3 = 2.0f * this->x_[3];
298
299   float sq0 = this->x_[0] * this->x_[0];
300   float sq1 = this->x_[1] * this->x_[1];
301
302   // measurement jacobian
303   float H[4 * 7];
304   clear(H, 28);
305
306   H[0] = dq2;    H[1] = -dq3;  H[2] = dq0;   H[3] = -dq1;
307   H[7] = -dq1;   H[8] = -dq0;  H[9] = -dq3;  H[10] = -dq2;
308   H[14] = -dq0;  H[15] = dq1;  H[16] = dq2;  H[17] = -dq3;
309
310   H[21] = (-4.0f * sq0*this->x_[3] + 4.0f * sq1 * this->x_[3] - dq3 - 8.0f *
      this->x_[1] * this->x_[2] * this->x_[0]);
311   H[22] = (4.0f * this->x_[2] * sq0 - 4.0f * sq1 * this->x_[2] - dq2 - 8.0f *
       this->x_[1] * this->x_[0] * this->x_[3]);
312   H[23] = dq1 * (dq1 * this->x_[1] + dq0 * this->x_[0] - 1.0f);
```

```
313   H[24] = dq0 * (dq0 * this->x_[0] + dq1 * this->x_[1] - 1.0f);
314
315   float h_scale = pow(dq1 * this->x_[2] + dq0 * this->x_[3], 2.0f) + pow(dq1
        * this->x_[1] + dq0 * this->x_[0] - 1.0f, 2.0f);
316
317   for(int i = 21; i < 25; i++) H[i] /= h_scale;
318
319   // noise jacobian
320   float theta = asin(acc[0]);
321   float ct = cos(theta);
322
323   float phi = asin(-acc[1] / ct);
324   float st = sin(theta);
325   float cp = cos(phi);
326   float sp = sin(phi);
327
328   float m_scale = pow(mag[2] * sp - mag[1] * cp, 2.0f) + pow(mag[0] * ct +
        mag[2] * cp * st + mag[1] * sp * st, 2.0f);
329
330   this->M_[21] = (-ct * (mag[2]*sp - mag[1]*cp));
331   this->M_[22] = (-mag[2] * cp * cp * st - mag[0] * ct * cp - mag[2] * sp *
        sp * st);
332   this->M_[23] = (mag[0] * sp * ct + mag[1] * sp * sp * st + mag[1] * cp * cp
         * st);
333
334   for(int i = 21; i < 24; i++) this->M_[i] /= m_scale;
335
336   /* Km = P * H.T * (H * P * H.T + M * R * M.T)^-1 */ {
337     float tmp_a[28], tmp_b[16], tmp_c[28];
338
339     multiply(H, this->P_, 4, 7, 7, tmp_a);
340     transMultiply(tmp_a, H, 4, 7, 4, tmp_b);
341
342     multiply(this->M_, this->R_, 4, 6, 6, tmp_a);
343     transMultiply(tmp_a, this->M_, 4, 6, 4, tmp_c);
344
345     add(tmp_b, tmp_c, 16, tmp_a);
346     invert(tmp_a, 4, tmp_b);
347
348     transMultiply(this->P_, H, 7, 7, 4, tmp_a);
349     multiply(tmp_a, tmp_b, 7, 4, 4, this->K_);
350   }
351
352   /* x += K * (acc - h) */ {
353     float h[4], tmp[7];
354
355     acc[3] = atan2(mag[2] * sp - mag[1] * cp, mag[0] * ct + mag[1] * st * sp
        + mag[2] * st * cp);
356
357     h[0] = acc[0] - (-2.0f * (this->x_[1] * this->x_[3] - this->x_[0] * this
        ->x_[2]));
358     h[1] = acc[1] - (-2.0f * (this->x_[0] * this->x_[1] + this->x_[2] * this
        ->x_[3]));
359     h[2] = acc[2] - (-(sq0 - sq1 - this->x_[2] * this->x_[2] + this->x_[3] *
        this->x_[3]));
360     h[3] = acc[3] - atan2(2.0f * this->x_[1] * this->x_[2] + 2.0f * this->x_
        [0] * this->x_[3], 2.0f * sq0 + 2.0f * sq1 - 1.0f);
361
362     multiply(this->K_, h, 7, 4, 1, tmp);
363     add(this->x_, tmp, 7, this->x_);
364   }
365
```

```
366   /* P = (I - K * H) * P */ {
367     float tmp_a[49], tmp_b[49];
368
369     multiply(this->K_, H, 7, 4, 7, tmp_a);
370     idSubtract(tmp_a, 7, tmp_a);
371
372     multiply(tmp_a, this->P_, 7, 7, 7, tmp_b);
373     clone(tmp_b, 7 * 7, this->P_);
374   }
375 }
376
377 float AHRS::getPitch(){
378   return enclosed(this->initial_euler[0], this->euler[0]);
379 }
380
381 float AHRS::getRoll(){
382   return enclosed(this->initial_euler[1], this->euler[1]);
383 }
384
385 float AHRS::getYaw(){
386   return enclosed(this->initial_euler[2], this->euler[2]);
387 }
```

Code 6.6: The implementation of the AHRS system from section 4.1.3, based on an extended Kalman filter. The code was custom developed to be efficient and require only the lightweight math library (see Code 6.3 and Code 6.4). Its structure is roughly based on proposals by B. Taylor from Bolderflight. The AHRS system samples the sensor data at startup, allowing the system to calibrate itself by determining the initial state and determine the sensor noise. This is done by sampling the initial sensor measurements that, assuming the system remains stationary, give the individual noise variances.

```
1  import numpy as np
2  from filterpy.kalman import KalmanFilter
3
4  # discrete time step
5  dt = 0.01
6
7  # load sensor noise
8  stddev = np.loadtxt('sensor-stats.np')
9
10 filter = KalmanFilter(dim_x=3, dim_z=2)
11 filter.x = np.zeros(3)  # x -> [pos, vel, acc]
12
13 filter.F = np.array([
14     [1, dt, 0.5 * dt ** 2],
15     [0, 1, dt],
16     [0, 0, 1]
17 ])
18
19 filter.P = np.diag([stddev[0] ** 2, 1, stddev[1] ** 2])
20
21 filter.H = np.array([
22     [1, 0, 0],
23     [0, 0, 1]
24 ])
25
26 filter.Q = np.outer(np.array([
27     0.5 * dt ** 2,
28     dt,
29     1
30 ]) * 0.05)
31
```

```
32 filter.R = np.diag([stddev[0] ** 2, stddev[1] ** 2])
33
34 # load recorded sensor data
35 data = np.loadtxt('sensor-data.np')
36 states = []
37
38 # z -> [pos, acc]
39 for z in data:
40     filter.predict()
41     filter.update(z)
42
43     states.append(filter.x[:])
44
45 np.array(states).tofile('altimeter-result.np')
```

Code 6.7: The altimeter code. This version is written in Python to enhance readability. Most of the code exists both in C++ and Python, due to Python's ease of use and better better debugging tools, enabling easier development and testing. Later all code had to be converted to C++ to be compiled for execution on the actual flight controller. The examples above serve to illustrate all the optimisation steps that had been taken to make the code more efficient. See Section 4.1.3.

Dank

Ich danke meinem Lehrer, Lukas Fässler, für die Begleitung und die Betreuung dieser Arbeit, sowie für die Unterstützung während deren Entstehung. Meiner Familie danke ich für den moralischen Beistand und die zahlreichen Diskussionen, in deren Verlauf einige Ideen zu dieser Arbeit entstanden.

BIBLIOGRAPHY

[1] Federal Aviation Administration. *Pilot's Encyclopedia of Aeronautical Knowledge*. Federal Aviation Administration, 2007.

[2] D.D. Baals, W.R. Corliss, and United States. *Wind Tunnels of NASA*. NASA SP. Scientific and Technical Information Branch, National Aeronautics and Space Administration, 1981.

[3] C. Benavente-Peces, N. Cam-Winget, E. Fleury, and A. Ahrens. *Sensor Networks: 6th International Conference, SENSORNETS 2017, Porto, Portugal, February 19-21, 2017, and 7th International Conference, SENSORNETS 2018, Funchal, Madeira, Portugal, January 22-24, 2018, Revised Selected Papers*. Communications in Computer and Information Science. Springer International Publishing, 2019.

[4] S.M. Bozic. *Digital and Kalman Filtering: An Introduction to Discrete-Time Filtering and Optimum Linear Estimation, Second Edition*. Dover Books on Engineering. Dover Publications, 2018.

[5] F. Carden, R.P. Jedlicka, and R. Henry. *Telemetry Systems Engineering*. Artech House telecommunications library. Artech House, 2002.

[6] P.A. Craig. *Stalls & spins*. Tab practical flying series. McGraw-Hill, 1993.

[7] F.A.A. *Airplane Flying Handbook: Federal Aviation Administration*. Skyhorse Publishing, 2007.

[8] J.P. Fielding. *Introduction to Aircraft Design*. Cambridge Aerospace Series. Cambridge University Press, 1999.

[9] United States. National Advisory Committee for Aeronautics. *Technical Note - National Advisory Committee for Aeronautics*. Number Nr. 2091-2100 in Technical Note - National Advisory Committee for Aeronautics. National Advisory Committee for Aeronautics, 1950.

[10] P. Gaydecki and Institution of Electrical Engineers. *Foundations of Digital Signal Processing: Theory, Algorithms and Hardware Design*. IEE circuits and systems series: Institution of Electrical Engineers. Institution of Engineering and Technology, 2004.

[11] Nikolai Zhukovsky Joukowsky. Über die Konturen der Tragflächen der Drachenflieger. *Zeitschrift für Flugtechnik und Motorluftschiffahrt*, 1:281–284, 1910.

[12] J.B. Kuipers. *Quaternions and Rotation Sequences: A Primer with Applications to Orbits, Aerospace, and Virtual Reality*. Princeton paperbacks. Princeton University Press, 1999.

[13] A. Kurniawan. *MicroPython for ESP32 Development Workshop*. PE Press, 2017.

[14] W. Langewiesche. *Fly By Wire: The Geese, The Glide, The 'Miracle' on the Hudson*. Penguin Books Limited, 2010.

[15] T.C. Lyon. *Practical Air Navigation*. Civil aeronautics bulletin. U.S. Government Printing Office, 1940.

[16] J.H. Mathews and R.W. Howell. *Complex Analysis for Mathematics and Engineering*. G – Reference, Information and Interdisciplinary Subjects Series. Jones and Bartlett, 2006.

[17] A.F. Molland and S.R. Turnock. *Marine Rudders and Control Surfaces: Principles, Data, Design and Applications*. Elsevier Science, 2011.

[18] R.L. Naeseth, T.G. Gainer, United States. National Aeronautics, Space Administration, and Langley Research Center. *Low-speed Investigation of the Effects of Wing Sweep on the Aerodynamic Characteristics of Parawings Having Equal-length Leading Edges and Keel*. NASA technical note. NASA, 1963.

[19] A. O'Dwyer. *Handbook of PI and PID Controller Tuning Rules*. Imperial College Press, 2006.

[20] D. Patranabis. *Telemetry Principles*. McGraw-Hill Education, Pvt Limited, 1999.

[21] S.V. Raković and W.S. Levine. *Handbook of Model Predictive Control*. Control Engineering. Springer International Publishing, 2018.

[22] L.V. Schmidt. *Introduction to Aircraft Flight Dynamics*. AIAA Education Series. American Institute of Aeronautics & Astronautics, 1998.

[23] S.K. Singh. *Process Control: Concepts Dynamics And Applications*. Prentice-Hall Of India Pvt. Limited, 2009.

[24] H.A. Soule and United States. *Influence of Large Amounts of Wing Sweep on Stability and Control Problems of Aircraft*. National Advisory Committee for Aeronautics. Technical Note. National Advisory Committee for Aeronautics, 1946.

[25] United States. *Technical Note - National Advisory Committee for Aeronautics*. Number Nr. 3191-3200 in Technical Note - National Advisory Committee for Aeronautics. National Advisory Committee for Aeronautics, 1954.

[26] R. Stowell. *The Light Airplane Pilot's Guide to Stall/spin Awareness: Featuring the PARE Spin Recovery Checklist*. Rich Stowell Consulting, 2007.

[27] L. Tan. *Fundamentals of Analog and Digital Signal Processing*. AuthorHouse, 2008.

[28] E. Torenbeek. *Synthesis of Subsonic Airplane Design: An introduction to the preliminary design of subsonic general aviation and transport aircraft, with emphasis on layout, aerodynamic design, propulsion and performance*. Springer Netherlands, 2013.

[29] N. Vaughan. *Integrated Powertrains and Their Control*. Wiley, 2001.

[30] J.S. Wolper. *Understanding Mathematics for Aircraft Navigation*. McGraw-Hill Education, 2001.

[31] D. Wyatt and M. Tooley. *Aircraft Communications and Navigation Systems*. CRC Press, 2013.

[32] G.W. Younkin. *Industrial Servo Control Systems: Fundamentals And Applications, Revised And Expanded*. Fluid power and control. CRC Press, 2002.

[33] T. Zhang, M. Nakamura, S. Goto, and N. Kyura. *Mechatronic Servo System Control: Problems in Industries and their Theoretical Solutions*. Lecture Notes in Control and Information Sciences. Springer Berlin Heidelberg, 2004.

INDEX

List of Figures

List of Codes

Eigenständigkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende schriftliche Arbeit selbstständig und nur unter Zuhilfenahme der in den Verzeichnissen oder in den Anmerkungen genannten Quellen angefertigt habe.

Zürich, den 7. Januar 2020 Jonathan Hungerbühler

I confess that in 1901 I said to my brother Orville that man would not fly for fifty years.

— Wilbur Wright

If I had to choose, I would rather have birds than airplanes.

— Charles Lindbergh

Le plus beau rêve qui a jamais hanté le cœur des hommes depuis Icare est aujourd'hui réalité.

— Louis Blériot