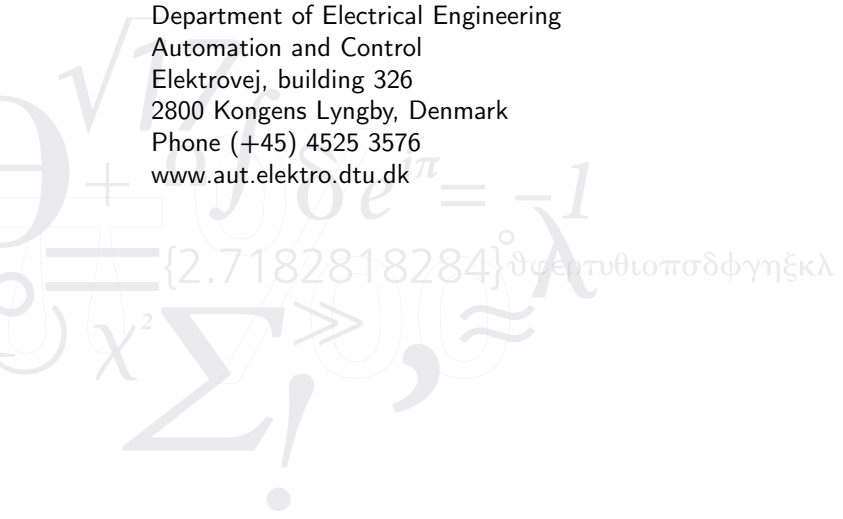
Flight Controller for Quad Rotor Helicopter in X-configuration

Kristian Sloth Lauszus

Kongens Lyngby 2015

Technical University of Denmark
Department of Electrical Engineering
Automation and Control
Elektrovej, building 326
2800 Kongens Lyngby, Denmark
Phone (+45) 4525 3576
www.aut.elektro.dtu.dk



Summary (English)

The goal of the thesis is to describe the development of a flight controller for a quad rotor helicopter in x-configuration, commonly known as a quadcopter. The flight controller will be implemented on a TI Tiva EK-TM4C123GXL Launchpad development board.

A variety of sensors including a gyroscope, an accelerometer, and a magnetometer will be used in order to stabilize and control the quadcopter.

The development that goes into such a system will be described in detail and a theoretical model will be introduced and simulated. The performance will then be evaluated on the final flight controller by logging the data in real time. Finally, this data will be compared to the simulated results.

The firmware is written in C and is available under the GPLv2 license at: `https://github.com/Lauszus/LaunchPadFlightController`.

Summary (Danish)

Formålet med afhandlingen er at beskrive udviklingen af en flight controller til en quad rotor helikopter i x-konfiguration - bedre kendt som en quadcopter. Flight controlleren vil blive implementeret på et TI Tiva EK-TM4C123GXL Launchpad development board.

En række sensorer inklusiv gyroskop, accelerometer og magnetometer vil blive brugt til at stabilisere og styre quadcopteren.

Udviklingen af systemet vil blive beskrevet fra et praktisk standpunkt, hvorefter en teoretisk model vil blive udledt og simuleret. Ydeevnen for den færdige flight controller vil dernæst blive evalueret ved at logge data fra systemet i realtid. Disse data vil derefter blive sammenlignet med de simulerede resultater.

Kildekoden er skrevet i C og er tilgængelige under GPLv2 licens på følgende side: `https://github.com/Lauszus/LaunchPadFlightController`.

Preface

This thesis was composed by Kristian Sloth Lauszus (s123808) at Department of Electrical Engineering at the Technical University of Denmark (DTU) in the period February 2, 2015 to June 26, 2015. This report is made as a 15 ECTS-point project in fulfilment of the requirements for acquiring a B.Sc. in Electrical Engineering at DTU.

The thesis deals with the development of a flight controller for a fixed pitch quad rotor helicopter in x-configuration also known as a quadcopter. X-configuration refers to the position of the motors, as the motors form an X with respect to the body of the quadcopter. Throughout this thesis the term quadcopter will be used to characterize such a helicopter.

In the first part of the thesis a theoretical model of the system will be presented. An explanation will be presented on how the orientation and altitude can be estimated using a 3-axis gyroscope, 3-axis accelerometer, 3-axis magnetometer, barometer and ultrasonic sensor.

The second part uses the the theoretical model and the equations, derived in part 1, to implement the system on a microcontroller. The hardware components are introduced first, followed by an introduction to the different flight modes. A description of the software implemented and the different PID controllers will then be described. Finally, the Android application written for this project is discussed.

In the final part the implementation will be tested and the result will be compared to the a theoretical model simulated using Simulink. The conclusion to the project is outlined and the future work is then proposed.

Along with the report a CD-rom is attached, containing the source code for the flight controller and Android application, video demonstrations, Matlab scripts, Simulink model, 3D model and other relevant files.

I would like to thank my supervisor Søren Hansen for his advice and guidance, especially during the final phase of this project.

Kristian Sloth Lauszus

Contents

Nomenclature

β	Filter coefficient for accelerometer LPF
$\Delta\phi$, $\Delta\theta$, $\Delta\psi$	Delta angle estimate by integrating the angular velocity
Δt	Delta time
δ	Filter coefficient for heading complimentary filter
$\dot{\phi}$, $\dot{\theta}$, $\dot{\psi}$	Angular velocity in the body frame
ϵ	Filter coefficient for velocity complimentary filter
γ	Filter coefficient for acceleration complimentary filter
α	Angular acceleration vector in the body frame
$\dot{\omega}$	Angular acceleration vector in the inertial frame
ω	Angular velocity vector in the inertial frame
τ	Torque vector in the body frame
$\tilde{\mathbf{a}}$	Acceleration vector in the inertial frame
$\mathbf{a_{acc-lpf}}$	Low-pass filtered accelerometer data vector
$\mathbf{a_{acc}}$	Accelerometer data vector
$\mathbf{a}$	Acceleration vector in the body frame
$\mathbf{F}$	Force vector
$\mathbf{I}$	Moment of inertia

$\mathbf{M_{mag}}$	Magnetometer data vector
$\mathbf{M}$	Magnetic field vector in the body frame
$\mathbf{r}$	Displacement vector
$\mathbf{R}(\phi, \theta, \psi)$	Rotation matrix
Ω_0	Average angular velocity at hover
Ω_n	Angular velocity of motor n
ϕ, θ, ψ	Roll, pitch, and yaw angle
ζ	Filter coefficient for height complimentary filter
b	Thrust coefficient
c_{air}	Speed of sound in air
d	Drag coefficient
d_{sonar}	Distance measured using sonar
ESC_n	Output sent to the ESC n in the range [-100:100]
F_T	Thrust force in the body frame
g	Gravitational acceleration (9.82 $^{m}/_{s^2}$)
h	Height above ground estimated using z-acceleration and barometer
h_{acc}	Height above ground estimated using z-acceleration
h_{baro}	Height above ground estimated using barometer
h_{ground}	Height of ground above sea level estimated using barometer
h_{sea}	Height above sea level estimated using barometer
h_{sonar}	Height above ground estimated using sonar
k	Discrete time unit
L	Distance from center of mass to point between two motors
m	Total mass of the quadcopter
M_{declin}	Magnetic declination
p	Pressure measured using barometer

p_0	Pressure at sea level (101325 Pa)
T_c	Temperature in degrees ℃
$T_{0.1c}$	Temperature in 0.1 degrees ℃
v_z	Velocity in the z-axis estimated using z-acceleration and barometer
v_{acc}	Velocity in the z-axis estimated using z-acceleration
v_{baro}	Velocity in the z-axis estimated using barometer
w	Width of the sonar echo pulse

Chapter 1

Introduction

The use of quadcopters and similar aircraft are becoming more and more widespread, in industrial use as well as among hobbyist. Common for all of these is that they need some kind of computer to control and stabilize the system. This is typically implemented on a microcontroller, which receives inputs from a variety of sensors like accelerometers, gyroscopes, barometers etc.

These systems are often referred to as drones, which are commonly associated with military operations but they are in non-military use too. More peaceful uses include 3D mapping of areas, search-and-rescue operations[5] or just for recreational use.

The purpose of this assignment is to make an universal flight controller that can be used with any quadcopter in x-configuration. Initially I had the following goals for this project:

- Use a gyroscope to get the quadcopter flying
- Use an accelerometer to get self-level mode working
- Use a magnetometer to reduce drift along yaw axis
- Use an ultrasonic sensor to measure distance at close proximity to ground

- Use a barometer to estimate the altitude far above the ground
- Use the altitude estimate from the ultrasonic sensor to stay in same absolute height within $\pm 10\,\mathrm{cm}$
- Use the altitude estimate from the barometer to stay at a fixed height with no more than $\pm 2\,\mathrm{m}$ drift

Part I

Part 1 - Quadcopter theory

Chapter 2

Theoretical model

In this chapter a theoretical model for the quadcopter will be derived. This should provide a better understanding of the physical system and how it behaves. The chapter will go through the theory needed in order to relate the forces applied by the motors and the resulting orientation in space. The space equations will then be used to develop a simulation model in Simulink in part 3. The different parameters will be calculated by measuring the final system, which will be introduced in part 2 of this thesis. Note standard SI-units are used if no units are noted.

Since the focus of this thesis is not to develop a thorough theoretical model, but rather to develop the actual flight controller, several aspect has been simplified. Thus, the model does not take into account the effect of wind, air resistance, and motor dynamics. It assumes that all propellers and motors are identical and that the quadcopter is perfect symmetrical.

Disclaimer

This work is inspired by André B.Ø. Bertelsen & Mikkel Wessel Thomsen [1, Chap. 2 & 4] and Andrew Gibiansky [6]. However the equations have been modified to be used with a quadcopter in x-configuration, as the quadcopter modeled in the prior work is based upon a quadcopter in plus-configuration i.e. where the x-axis go through the front motor.

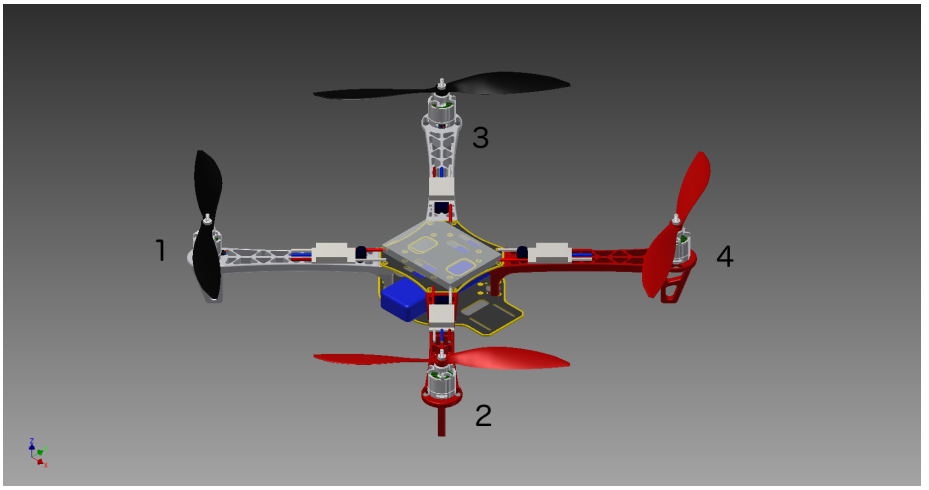


Figure 2.1: 3D rendering of the quadcopter. The numbers indicates the corresponding motor number

2.1 Establishment of the coordinate system

Figure 2.2 shows the right-hand coordinate system that is used in this project. It is important to be able to differentiate between the quadcopter coordinate system and the inertial frame coordinate system. The coordinate system for the quadcopter is defined by the orientation of the quadcopter with the x-axis facing forward between the two motor 2 and 4, the y-axis is facing left between motor 3 and 4, and the z-axis is facing upward (please refer to figure 2.1). This coordinate system will be referred to as the body frame. On the other hand, the inertial frame is defined by the ground; thus, the z-axis is pointing vertical up with respect to gravity.

Roll ϕ and pitch θ are defined in the body frame and both follow the right-hand rotation rule, while yaw ψ follows the left-hand rotation rule. This is done so yaw follows the rudder stick input on the remote control. The remote controller will be discussed in chapter 4.4.

In order for the different calculations to work it is necessary to orientate the sensors so all axes align up with the body frame. If this is not possible on the PCB due to various reasons. It can easily be remapped in code by swapping the different axes.

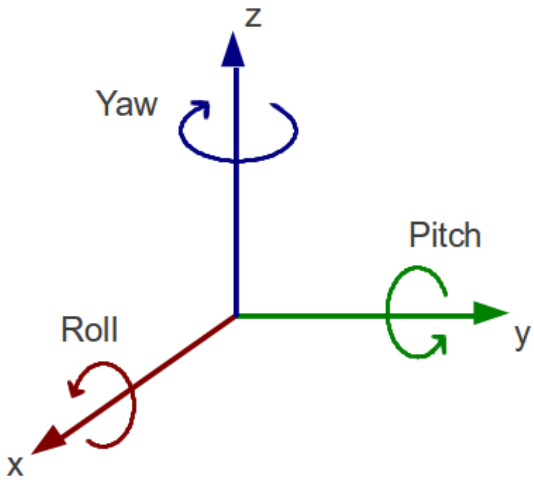


Figure 2.2: Coordinate system used in this thesis
Original: http://doc.aldebaran.com/2-1/family/nao_t2/joints_t2.html

2.2 Motor output to motor angular velocity

The output sent to the ESC (Electronics Speed Controller) is a value between -100 and 100. The resulting angular velocity Ω_n of motor n can be assumed to be linear proportional[1, p. 25-26]. This relationship can be estimated by measuring the angular velocity of the motors at several different outputs. The quadcopter follows the motor-layout shown in figure 2.1.

Figure 2.3 shows a linear plot of the angular velocity in $^{\text{rad}}/_{\text{s}}$ as a function of the motor output. The measurements were conducted by first applying black tape to all the motors and then applying a strip of reflective tape. The angular velocity in RPM (Revolutions Per Minute) was measured using a tachometer and then converted to $^{\text{rad}}/_{\text{s}}$. All the data used to generate the plot can be seen in appendix B. From the linear approximation it follows that there is the following relationship between the output sent to an ESC and the angular velocity of the corresponding motor:

$$\Omega_n = 6.4056\ ESC_n + 677.87\,^{\text{rad}}/_{\text{s}} \tag{2.1}$$

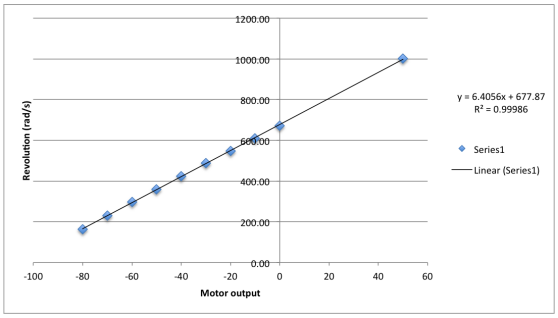


Figure 2.3: Motor output to $^{\text{rad}}/_{\text{s}}$

2.3 Estimation of the thrust coefficient

The thrust coefficient b relates the angular velocity of a motor and the resulting thrust force F_t:

$$b = \frac{F_T}{\Omega_n^2} \tag{2.2}$$

By substituting the thrust force with the force from gravity mg and inserting the average angular velocity at hover for all motors Ω_0, the thrust coefficient can be calculated:

$$b = \frac{mg}{4\Omega_0^2} \tag{2.3}$$

Note that the average angular velocity at hover is multiplied by 4 as there are four rotors.

The total mass of the quadcopter m was measured to be:

$$m = 1.162\,\text{kg} \tag{2.4}$$

The angular velocity of each motor at hover was logged (the full data-set can be found in appendix F) and then converted to $^{\text{rad}}/_{\text{s}}$ using 2.1:

$$\begin{aligned}\Omega_1 &= 630.04\,{}^{\mathrm{rad}}/_{\mathrm{s}}\\ \Omega_2 &= 672.93\,{}^{\mathrm{rad}}/_{\mathrm{s}}\\ \Omega_3 &= 606.41\,{}^{\mathrm{rad}}/_{\mathrm{s}}\\ \Omega_4 &= 564.34\,{}^{\mathrm{rad}}/_{\mathrm{s}}\end{aligned} \tag{2.5}$$

The average angular velocity can then be calculated:

$$\Omega_0 = 618.43\,{}^{\mathrm{rad}}/_{\mathrm{s}} \tag{2.6}$$

By inserting 2.4 and 2.6 into 2.3 I get:

$$\begin{aligned}b &= \frac{mg}{4\Omega_0^2}\\ &= \frac{1.162\ 9.82}{4\ 618.43^2}\\ &= 7.459 \times 10^{-6}\,{}^{\mathrm{kg\ m}}/_{\mathrm{rad}^2}\end{aligned} \tag{2.7}$$

Thus, by multiplying 2.7 with the angular velocity squared I can calculate the resulting thrust force for motor n:

$$F_T = b\Omega_n^2 \tag{2.8}$$

2.4 Estimation of the torque

The torque vector τ is defined as:

$$\tau = \mathbf{r} \times \mathbf{F} \tag{2.9}$$

F denotes the force vector and **r** is the displacement vector i.e the distance from the point where the torque is measured to where the force is applied. In this case it is the distance from the center of mass to the point between two motors, as the quadcopter is assumed to be perfect symmetrical.

By replacing **F** with the thrust force found in 2.8, the torque due to the thrust force along the x- and y-axis is given by:

$$\begin{aligned} \tau_x &= Lb(\Omega_3^2 + \Omega_4^2 - \Omega_1^2 - \Omega_2^2) \\ \tau_y &= Lb(\Omega_1^2 + \Omega_3^2 - \Omega_2^2 - \Omega_4^2) \end{aligned} \tag{2.10}$$

Where the distance L was measured to be:

$$L = 18\,\text{cm} \tag{2.11}$$

2.5 Estimation of the moment of inertia

In order to estimate the moment of inertia **I** of the system, a 3D model was created in a CAD program. A rendering of the model can be seen in figure 2.1.

In order to calculate the moment of inertia it is very important that the correct weight distribution is used; thus, the mass of all components and their placement have been carefully measured. From my 3D model the moment of inertia was found to be:

$$\mathbf{I} = \begin{bmatrix} 12.091 & -0.002 & 0.000 \\ -0.002 & 11.719 & 0.000 \\ 0.000 & 0.000 & 22.923 \end{bmatrix} \times 10^{-3}\ \text{kg m}^2 \tag{2.12}$$

This can be approximated to:

$$\mathbf{I} \approx \begin{bmatrix} 12.091 & 0 & 0 \\ 0 & 11.719 & 0 \\ 0 & 0 & 22.923 \end{bmatrix} \times 10^{-3}\ \text{kg m}^2 \tag{2.13}$$

Thus, the moment of inertia is a diagonal matrix as expected since the quadcopter is assumed to be symmetrical.

The 3D model can be found in appendix F.

2.6 Estimation of the drag coefficient

According to [6] the relationship between the torque around the z-axis and the angular velocities of the motors is given by:

$$\tau_z = d(\Omega_2^2 + \Omega_3^2 - \Omega_1^2 - \Omega_4^2) \tag{2.14}$$

Where d is the so-called drag coefficient.

Using Newton's second law for rotation I get:

$$\tau = \mathbf{I}\alpha \tag{2.15}$$

Where α is the angular acceleration. Thus, the drag coefficient can be calculated according to the following formula:

$$d = \frac{\alpha_z I_{zz}}{\Omega_2^2 + \Omega_3^2 - \Omega_1^2 - \Omega_4^2} \tag{2.16}$$

In order to estimate the drag coefficient the quadcopter was set to spin with a constant acceleration around yaw. The angular velocity from the gyroscope, the time, and the output to the motors were then logged. A graph of the logged angular velocity around the z-axis $\dot{\psi}(t)$ can be seen in figure 2.4. The full dataset can be found on the attached CD-ROM. As observed the angular velocity is increasing linearly as expected. From the data I calculated the relationship between the time and the angular velocity around the z-axis:

$$\dot{\psi}(t) = 0.4695t + 0.238 \tag{2.17}$$

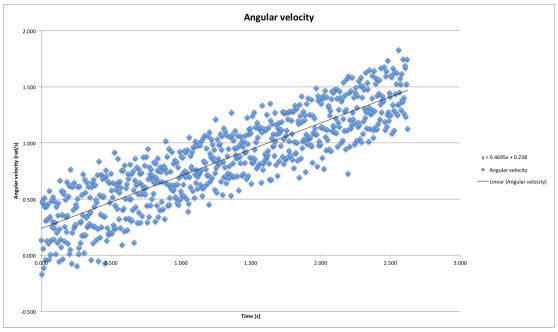


Figure 2.4: Angular velocity at constant angular acceleration

The angular acceleration around the z-axis can then be found:

$$\alpha_z = \frac{\dot{\psi}(t_2) - \dot{\psi}(t_1)}{t_2 - t_1} \tag{2.18}$$

Thus from my data I estimated the angular acceleration to be:

$$\begin{aligned} \alpha_z &= \frac{1.4690 - 0.2380}{2.6220} \\ &= 0.4695 \, {}^{\mathrm{rad}}/_{\mathrm{s}^2} \end{aligned} \tag{2.19}$$

The average motor output to each motor was then converted to ${}^{\mathrm{rad}}/_{\mathrm{s}}$ using 2.1:

$$\begin{aligned} \Omega_1 &= 449.38 \, {}^{\mathrm{rad}}/_{\mathrm{s}} \\ \Omega_2 &= 595.33 \, {}^{\mathrm{rad}}/_{\mathrm{s}} \\ \Omega_3 &= 508.79 \, {}^{\mathrm{rad}}/_{\mathrm{s}} \\ \Omega_4 &= 437.14 \, {}^{\mathrm{rad}}/_{\mathrm{s}} \end{aligned} \tag{2.20}$$

Note how motor 2 and 3 spins faster than motor 1 and 4; thus, the quadcopter was spinning clockwise. This can also be seen from the positive angular velocity. The reason why the motor pairs do not spin at exactly the same angular velocity is due to the fact that the motors also need to maintain the roll and pitch angle of the quadcopter.

By inserting the calculated values into 2.16 the drag coefficient can be calculated:

$$\begin{aligned} d &= \frac{\alpha_z I_{zz}}{\Omega_2^2 + \Omega_3^2 - \Omega_1^2 - \Omega_4^2} \\ &= \frac{0.4695\ 22.923 \times 10^{-3}}{595.33^2 + 508.79^2 - 449.38^2 - 437.14^2} \\ &= 48.864 \times 10^{-9 \mathrm{kg\ m^2/rad}} \end{aligned} \tag{2.21}$$

2.7 Space equations

By combining 2.10 and 2.14 the torque around all axes is given as:

$$\tau = \begin{bmatrix} Lb(\Omega_3^2 + \Omega_4^2 - \Omega_1^2 - \Omega_2^2) \\ Lb(\Omega_1^2 + \Omega_3^2 - \Omega_2^2 - \Omega_4^2) \\ d(\Omega_2^2 + \Omega_3^2 - \Omega_1^2 - \Omega_4^2) \end{bmatrix} \tag{2.22}$$

According to Euler's rotation equations, the system can be written as:

$$\mathbf{I}\dot{\omega} + \omega \times (\mathbf{I}\omega) = \tau \tag{2.23}$$

Where $\dot{\omega}$ and ω is the angular acceleration and velocity, respectively, in the inertial frame.

By isolating the angular acceleration $\dot{\omega}$ I get:

$$\begin{aligned} \dot{\omega_x} &= \frac{\tau_x + (I_{yy} - I_{zz})\omega_y\omega_z}{I_{xx}} \\ \dot{\omega_y} &= \frac{\tau_y + (I_{zz} - I_{xx})\omega_x\omega_z}{I_{yy}} \\ \dot{\omega_z} &= \frac{\tau_z + (I_{xx} - I_{yy})\omega_x\omega_y}{I_{zz}} \end{aligned} \tag{2.24}$$

By inserting 2.22 into 2.24 I get the following space equations for the system:

$$\dot{\omega} = \begin{bmatrix} \frac{Lb\left(\Omega_3^2+\Omega_4^2-\Omega_1^2-\Omega_2^2\right)+(I_{yy}-I_{zz})\omega_y\omega_z}{I_{xx}} \\ \frac{Lb\left(\Omega_1^2+\Omega_3^2-\Omega_2^2-\Omega_4^2\right)+(I_{zz}-I_{xx})\omega_x\omega_z}{I_{yy}} \\ \frac{d\left(\Omega_2^2+\Omega_3^2-\Omega_1^2-\Omega_4^2\right)+(I_{xx}-I_{yy})\omega_x\omega_y}{I_{zz}} \end{bmatrix} \tag{2.25}$$

The angular velocity in the inertial frame and the body frame can be related by the transformation matrix $\mathbf{H}(\phi, \theta, \psi)$:

$$\omega = \mathbf{H}(\phi, \theta, \psi) \begin{bmatrix} \dot{\phi} \\ \dot{\theta} \\ \dot{\psi} \end{bmatrix} \tag{2.26}$$

Where $\mathbf{H}(\phi, \theta, \psi)$ is given by:

$$\mathbf{H}(\phi, \theta, \psi) = \begin{bmatrix} 1 & 0 & -sin(\theta) \\ 0 & cos(\phi) & sin(\phi)cos(\theta) \\ 0 & -sin(\phi) & cos(\phi)cos(\theta) \end{bmatrix} \tag{2.27}$$

Thus, the angular velocity in the body frame can be calculated from the angular velocity in the inertial frame in the following way:

$$\begin{bmatrix} \dot{\phi} \\ \dot{\theta} \\ \dot{\psi} \end{bmatrix} = \mathbf{H}(\phi, \theta, \psi)^{-1}\omega \tag{2.28}$$

Chapter 3

Estimation of the orientation

In order to calculate the attitude i.e. roll, pitch, and yaw of the quadcopter, an IMU (Inertial Measurement Unit) is used. In this project it consists of a 3-axis gyroscope, a 3-axis accelerometer, and a 3-axis magnetometer. They measure the angular velocity, the acceleration, and the earth magnetic field, respectively. This chapter will go into detail on how the orientation of the quadcopter can be estimated from these measurements. Furthermore, it will describe how the altitude can be estimated with the ultrasonic sensor and barometer.

3.1 Rotation matrices

In order to rotate the coordinate axis of a coordinate system with the angles ϕ, θ and ψ the following rotation matrices can be used:

$$\mathbf{R_x}(\phi) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos(\phi) & \sin(\phi) \\ 0 & -\sin(\phi) & \cos(\phi) \end{bmatrix} \tag{3.1}$$

$$\mathbf{R_y}(\theta) = \begin{bmatrix} \cos(\theta) & 0 & -\sin(\theta) \\ 0 & 1 & 0 \\ \sin(\theta) & 0 & \cos(\theta) \end{bmatrix} \tag{3.2}$$

$$\mathbf{R_z}(\psi) = \begin{bmatrix} \cos(\psi) & \sin(\psi) & 0 \\ -\sin(\psi) & \cos(\psi) & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{3.3}$$

The matrix product of 3.3, 3.2 and 3.1 respectively is given by:

$$\begin{aligned} \mathbf{R}(\phi,\theta,\psi) &= \mathbf{R_z}(\psi) \cdot \mathbf{R_y}(\theta) \cdot \mathbf{R_x}(\phi) \\ &= \begin{bmatrix} \cos(\psi)\cos(\theta) & \sin(\psi)\cos(\phi) + \cos(\psi)\sin(\theta)\sin(\phi) & \sin(\psi)\sin(\phi) - \cos(\psi)\sin(\theta)\cos(\phi) \\ -\sin(\psi)\cos(\theta) & \cos(\psi)\cos(\phi) - \sin(\psi)\sin(\theta)\sin(\phi) & \cos(\psi)\sin(\phi) + \sin(\psi)\sin(\theta)\cos(\phi) \\ \sin(\theta) & -\cos(\theta)\sin(\phi) & \cos(\theta)\cos(\phi) \end{bmatrix} \end{aligned} \tag{3.4}$$

Thus, the rotation matrix 3.4 will rotate the three coordinate axis by the angles ϕ, θ, and ψ [2][7, p. 134-144].

3.2 Attitude estimation

In order to estimate the attitude, the values are first sampled from the gyroscope, accelerometer, and magnetometer and converted into appropriate units. The accelerometer readings represent the magnitude of the acceleration along the body frame. Thus, if the quadcopter is laying still and is perfectly horizontal, both the x- and y-values correspond to 0 g, while the z-value will correspond to -1 g.

To estimate roll and pitch the accelerometer readings $\mathbf{a_{acc}}$ at time k are first low-pass filtered:

$$\mathbf{a_{acc-lpf}}(k) = \beta \mathbf{a_{acc-lpf}}(k-1) + (1-\beta)\mathbf{a_{acc}}(k), \quad \beta > 0.5 \tag{3.5}$$

The low-pass filter is used to reduce the high frequency noise that affects the readings caused by linear acceleration and vibrations.

The delta angles are obtained by multiplying the angular velocity measured using the gyroscope with the time since the last measurement:

$$\begin{aligned} \Delta\phi &= \dot{\phi}\Delta t \\ \Delta\theta &= \dot{\theta}\Delta t \\ \Delta\psi &= \dot{\psi}\Delta t \end{aligned} \tag{3.6}$$

The previous estimate of the magnitude of acceleration in the body frame $\mathbf{a}(k-1)$ is then rotated using the delta angle found in 3.6:

$$\mathbf{a}(k) = \mathbf{R}(\Delta\phi, \Delta\theta, \Delta\psi)\mathbf{a}(k-1) \tag{3.7}$$

The problem, however, is that this will cause the estimate to drift over time due to the fact that delta angle obtained from the gyroscope corresponds to an integration; thus, the error will also be integrated. This is solved by using a so-called complimentary filter, which applies a high-pass filter on the rotated body frame and then adds and further applies a low-pass filter to the accelerometer data:

$$\mathbf{a}(k) = \gamma\mathbf{a}(k) + (1-\gamma)\mathbf{a}_{\mathbf{acc-lpf}}(k), \quad \gamma > 0.5 \tag{3.8}$$

In practice the filter coefficient γ is close to one.

The complimentary filter is skipped if the magnitude of the acceleration data vector exceeds 0.85 g and 1.15 g. Thus, the acceleration from the accelerometer is not used to update the estimate in extreme manoeuvres.

Each axis represents the acceleration in the body frame **a**. By using eqn. 28 and 29 from [17] the roll and pitch angles can be calculated:

$$\phi = atan2(a_y, \sqrt{a_x^2 + a_z^2}) \tag{3.9}$$

$$\theta = atan2(-a_x, a_z) \tag{3.10}$$

Atan2 is a software implementation of the inverse trigonometric functions atan, but takes the signs of both arguments into account in order to estimate the quadrant[3]. Note that atan2 is used in 3.10 as well in order to prevent possible division by 0.

The magnetometer readings represent the strength of the magnetic field along all three coordinate axis of the body frame. Similar to the roll and pitch estimate, the previous estimate in the body frame $\mathbf{M}(k-1)$ is first rotated using the delta angles from the gyroscope:

$$\mathbf{M}(k) = \mathbf{R}(\Delta\phi, \Delta\theta, \Delta\psi)\mathbf{M}(k-1) \tag{3.11}$$

This is then combined with the new magnetometer readings using a complimentary filter, which acts as a high-pass filter on the rotated frame and a low-pass filter on the magnetometer readings $\mathbf{M_{mag}}$:

$$\mathbf{M}(k) = \delta\mathbf{M}(k) + (1-\delta)\mathbf{M_{mag}}(k), \quad \delta > 0.5 \tag{3.12}$$

Note that the filter coefficient δ is in practice very close to one.

The yaw angle is then calculated according to [15] eqn. 22:

$$\begin{aligned} B_y &= M_z sin(\phi) - M_y cos(\phi) \\ B_x &= M_x cos(\theta) + M_y sin(\theta) sin(\phi) + M_z sin(\theta) cos(\phi) \\ \psi &= atan2(B_y, B_x) + M_{declin} \end{aligned} \tag{3.13}$$

M_{declin} represents the local magnetic declination and varies depending on the geographical location[14].

The yaw angle is then inverted so it complies with the left-hand rotation as defined in figure 2.2:

$$\psi = -\psi \tag{3.14}$$

Furthermore, the range [-180°:180°] is converted to [0°:360°], which is normally used for compass heading:

$$\psi = \begin{cases} \psi + 360^\circ & \text{for } \psi < 0 \\ \psi & \text{otherwise} \end{cases} \tag{3.15}$$

Figure 3.1 shows a flowchart of each step in the attitude estimation.

There is one fundamental problem with using Euler angles, as it is affected by what is known as gimbal lock, which happens when two axes align. The result is that one degrees of freedom is lost. For instance this happens every time the pitch angle is ±90°. In this project it is prevented by limiting the roll and pitch angles of the quadcopter to 50° when self-level mode is activated, as will be discussed in 5.

3.3 Altitude estimation using sonar

An ultrasonic sensor, also known as a sonar, is used to measure the distance at up to 3 m. In order for the sensor to start taking measurements it needs a trigger pulse at 2-5 μs [16, p. 1]. It will then respond with a so-called echo signal where the width w is equal to the time it took for the sound to travel to the object and back again. Thus, by measuring the width of the pulse one can determine the distance to the object.

Figure 3.2 shows an example output from the sensor. The yellow wave is the trigger signal sent from the flight controller and the blue wave is the echo signal sent back from the sonar. Note that the difference in the two amplitudes is due to the operating voltage of the microcontroller and the sonar being 3.3 V and 5 V respectively.

According to the specification the minimum and maximum width of the echo signal is 115 μs and 18.5 ms, respectively [16, p. 2]. Thus, the widths that exceed these limits should be discarded.

The distance to the object can then be converted into mm as follows:

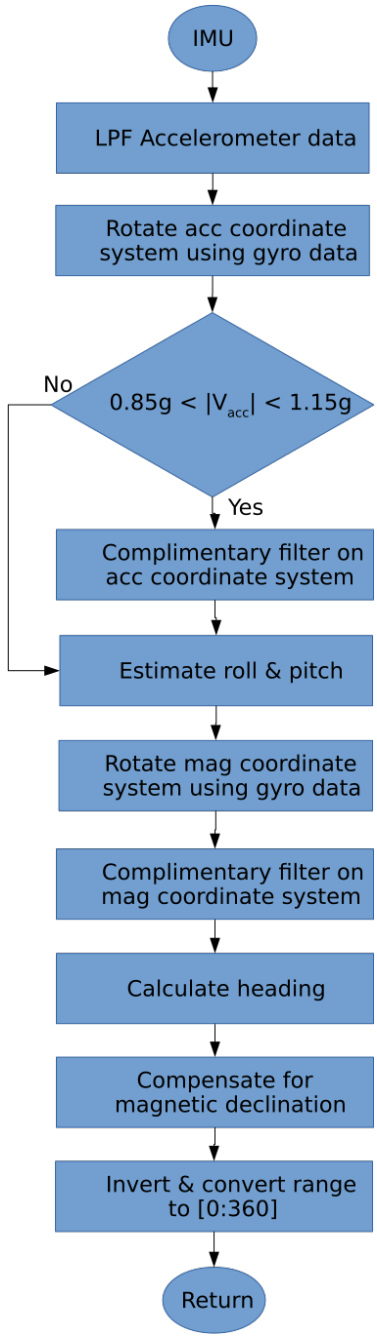


Figure 3.1: Flowchart for attitude estimation

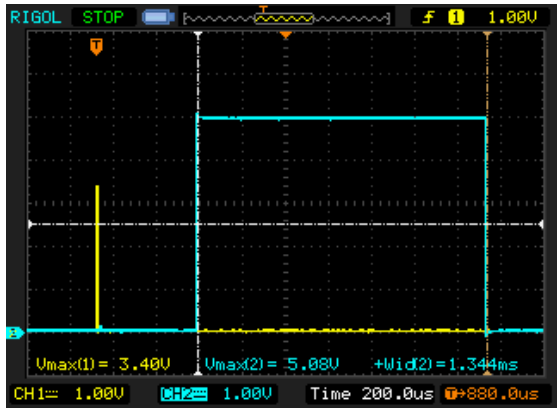


Figure 3.2: Example output from sonar sensor

$$\begin{aligned} d_{sonar} &= \frac{w}{(\frac{2\times10^{-3}}{c_{air}})}\,\text{mm} \\ &= \frac{w \times c_{air}}{2 \times 10^{-3}}\,\text{mm} \end{aligned} \tag{3.16}$$

To be noticed is that the value is divided by two as the width of the pulse represents the time it takes for the sound wave to travel back and forth. The factor 10^{-3} is used to convert the reading into mm.

The speed of sound in air depends on the temperature and is given by the following relationship [16, p. 3]:

$$c_{air} = 331.5 + (0.6T_c)\,\text{m/s} \tag{3.17}$$

This can be approximated to $343.5\,\text{m/s}$ assuming $T_c = 20℃$.

Thus, by inserting the width in figure 3.2 into 3.16 and assuming $c_{air} = 343.5\,\text{m/s}$ the distance can be calculated to be:

$$d_{sonar} = \frac{1.344\,\text{ms} \times 343.5\,\text{m/s}}{2 \times 10^{-3}} = 230.83\,\text{mm} \tag{3.18}$$

In this case I use the temperature reading from the barometer which has a resolution of 0.1℃[19, p. 6]; thus, 3.17 can be written as:

$$c_{air} = 3315 + (0.6T_{0.1c})\,{}^{\mathrm{dm}}/_{\mathrm{s}} \tag{3.19}$$

By inserting 3.19 into 3.16 I obtain:

$$d_{sonar} = \frac{w \times (3315 + (0.6T_{0.1c}))}{2 \times 10^{-2}}\,\mathrm{mm} \tag{3.20}$$

This distance does not represent the height above ground, but depends on the roll and pitch angle of the quadcopter. Since these are both known, the height above ground can be calculated by finding the adjacent side in 3D-space according to the formula:

$$h_{sonar} = d_{sonar} \cos\phi \cos\theta\,\mathrm{mm} \tag{3.21}$$

The final height estimated using 3.21 can then be used to stay at a fixed height above ground, as explained in chapter 7. However, in order to prevent the angle of becoming too large, the roll and pitch angles are limited to 25° when running in this mode.

3.4 Altitude estimation using barometer

At time k the pressure p is measured using the barometer. Using [19, p. 16] the altitude above sea level h_{sea} can be calculated:

$$h_{sea}(k) = 44330\left(1 - \left(\frac{p(k)}{p_0}\right)^{\frac{1}{5.255}}\right) \tag{3.22}$$

The height of ground level h_{ground} above the sea is found by taking an initial measurement when the flight controller is first turned on:

$$h_{ground} = h_{sea}(0) \tag{3.23}$$

By subtracting 3.23 from any future measurements an estimate of the height above ground h_{baro} can be calculated:

$$h_{baro}(k) = h_{sea}(k) - h_{ground} \tag{3.24}$$

By dividing the current height estimate with the previous one and dividing this by the time between the two measurements Δt, the velocity v_{baro} can be found:

$$v_{baro}(k) = \frac{h_{baro}(k) - h_{baro}(k-1)}{\Delta t} \tag{3.25}$$

Likewise the velocity can also be estimated by integrating the z-acceleration with respect to the inertial frame $\tilde{a}_z$:

$$v_{acc}(k) = v_z(k-1) + \tilde{\mathbf{a}}_{\mathbf{z}}(k)\Delta t \tag{3.26}$$

Where the acceleration in the inertial frame is given by:

$$\tilde{\mathbf{a}}(k) = \mathbf{R}(-\phi, -\theta, \psi)\mathbf{a}(k) \tag{3.27}$$

Note that the angles are inverted in order to rotate the body frame into the inertial frame. However the yaw angle is already inverted in 3.14, so that is why it is not needed in 3.27.

A better estimate of the velocity is found by using a complimentary filter on the two velocity estimates 3.25 and 3.26:

$$v_z(k) = \epsilon v_{acc}(k) + (1-\epsilon)v_{baro}(k), \quad \epsilon > 0.5 \tag{3.28}$$

The filter coefficient ϵ should be close to one, so the velocity estimated from the acceleration and barometer is high-pass and low-pass filtered, respectively. The data is filtered this way because the barometer will give the true height over time, but over a short time span it will be very noisy due to weather condition. On the other hand, the estimated velocity from integration of the acceleration will drift over time.

Since the velocity in the z-axis v_z is now known, an estimate of the height can be found from the z-acceleration in the inertial frame:

$$h_{acc}(k) = h(k-1) + v_z(k-1)\Delta t + \frac{1}{2}\tilde{a}_z(k)\Delta t^2 \tag{3.29}$$

The current height is then estimated using a complimentary filter, which acts as a high-pass filter on 3.29 and a low-pass filter on 3.24. Hence, the filter coefficient ζ should be close to one.

$$h(k) = \zeta h_{acc} + (1-\zeta)h_{baro}, \quad \zeta > 0.5 \tag{3.30}$$

Part II

Part 2 - Design & implementation

Chapter 4

Hardware components

In this chapter the different hardware components will be presented.

Figure 4.1 shows an overview of the current setup. Four ESCs (Electronics Speed Controllers) are mounted on each arm and are used to drive the four sensor-less brushless DC motors, which are commonly used in applications like these.

Furthermore, a serial Bluetooth module is connected, which allows the quadcopter to communicate with a dedicated Android application. The Bluetooth connection can also be used to log different variables in real time from the quadcopter while it is flying, as it has been done in chapter 9.

A list of all hardware components used in this project can be found in appendix D.

4.1 The microcontroller

The core of this project is the TI Tiva EK-TM4C123GXL Launchpad development board featuring the TM4C123GH6PM 32-bit ARM Cortex-M4 micro-

Figure 4.1: Quadcopter overview

controller with 256 KB flash, 32 KB single-cycle SRAM, 2 KB EEPROM and is running at a clock frequency of 80 MHz [10].

This microcontroller has the peripherals needed including I2C, UART, PWM, Timers etc. The microcontroller is also 5 V tolerant on most pins, this is important as the sensors like the ultrasonic sensor is operating at 5 V. Furthermore, it has an FPU (Floating Point Unit), which makes it ideal for such an application where a lot of floating point math will be used.

Figure 4.2 shows the schematic for the PCB used for the flight controller. It is designed to plug into the headers on the development board.

An on-board RGB LED is used to indicate the current state of the quadcopter. Green means that it is unarmed, red means armed. The blue LED is used as a status LED for various purposes.

A buzzer is connected for audio feedback to let the user know when it has finished initializing all the sensors. The buzzer is turned on constantly in case an error occurs, e.g. if the signal to the receiver is lost.

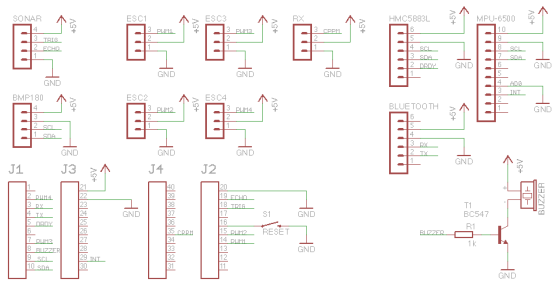


Figure 4.2: Schematic

4.2 Sensors

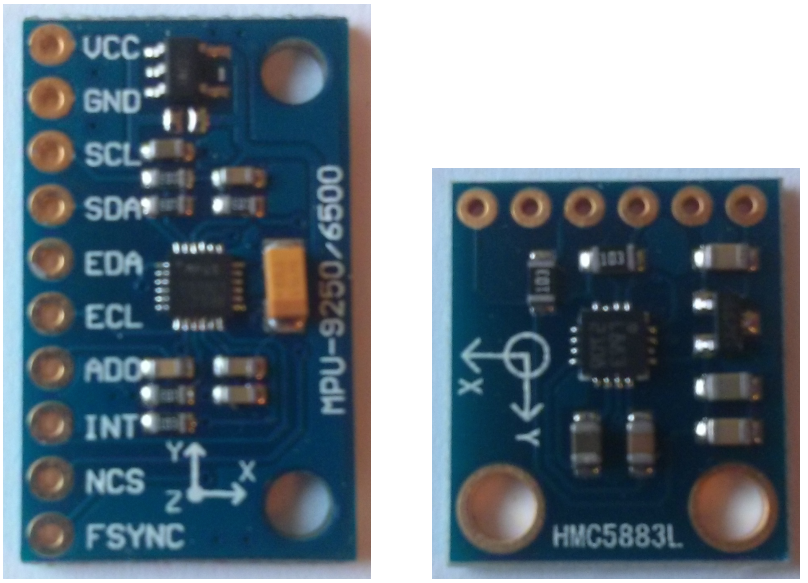
In order to stabilize the quadcopter the flight controller needs to take input from several sensors. These allows the flight controller to estimate the attitude and altitude of the quadcopter using the equations found in part 1.

The microcontroller uses I2C running at a clock frequency of 400 kHz for communicating with all sensor despite the ultrasonic sensor.

4.2.1 Gyroscope & accelerometer

In this project I am using the MPU-6500, which features a 3-axis gyroscope and 3-axis accelerometer in one package. It has built-in digital filters and several other useful features. One convenient feature is that it is highly configurable e.g. the range of the sensors can be adjusted depending on the application i.e. the gyroscope range can be set to either ± 250, ± 500, ± 1000 or ± 2000 degrees and the range of the accelerometer can be set to either $\pm 2\,\mathrm{g}$, $\pm 4\,\mathrm{g}$, $\pm 8\,\mathrm{g}$ or $\pm 16\,\mathrm{g}$ by writing to the appropriate register. In this application a range of ± 2000 deg/s and $\pm 8\,\mathrm{g}$ was used, respectively. I know from experience that a high range is needed, as the sensor might otherwise be saturated at extreme manoeuvres [9].

The sensor was configured to have an update rate of 1 kHz. Furthermore, it was programmed to set a GPIO pin (denoted as the INT pin) at logic level high every time new measurements are ready to be read.



(a) MPU-6500 gyroscope and accelerometer breakout

(b) HMC5883L magnetometer breakout

Figure 4.3

4.2.2 Magnetometer

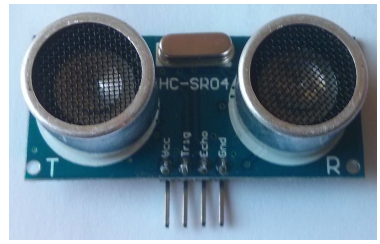
As the estimate of the yaw angle will also drift, due to the integration of the gyroscope data, a 3-axis magnetometer is used. By using the magnetometer the compass heading can be estimated from the earth magnetic field, as explained in chapter 3.2.

In this project I will use the HMC5883L, which has an accuracy of 1-2°. Like the MPU-6500 it also has a GPIO pin (denoted as the DRDY pin) that is used to indicate when new measurements are ready to be processed. The update rate for the HMC5883L is relative slow at only 15 Hz [8].

4.2.3 Ultrasonic sensor

An ultrasonic sensor of the type HC-SR04 with a range of 2 cm to 3 m is used to measure the distance above the ground every 25 ms [16]. This allows the quadcopter to stay at a fixed height above ground. This ultrasonic sensor is not the best in its class, but it is cheap and widely available, which makes it ideal

(a) BMP180 barometer breakout



(b) HC-SR04 ultrasonic sensor

Figure 4.4

for a project like this.

4.2.4 Barometer

A BMP180 barometer is used to measure the current pressure level at an update rate of 33 Hz. The pressure can be used to give a rough estimate of the height above ground. This is due to the fact that the pressure change with respect to the height is not directly proportional, as the pressure will be affected by the wind, temperature and other weather conditions. According to the datasheet the sensor has a relative accuracy of ± 1 m[19, p. 6], which is within my goal of staying at fixed height with no more than ± 2 m drift.

4.3 Motors

The motors used are 3-phase sensor-less BLDC (Brushless DC) motors. Each motor is controlled by a dedicated ESC (Electronic Speed Controllers) that takes care of controlling the 3-phases. The microcontroller can control the speed of the motor by varying the width of a PPM (Pulse Position Modulation) signal connected to the ESC. The ESCs used in this application are running an optimized firmware meant for multirotors [13], which is a general term for a helicopter with multiple rotors. With this firmware the ESCs has a much higher

update frequency and almost instant input-to-output response.

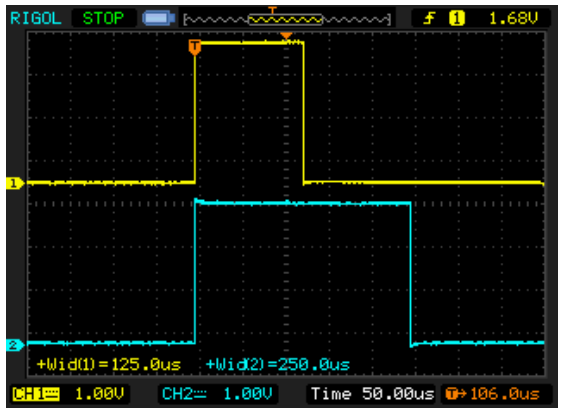


Figure 4.5: Minimum (125 μs) and the maximum (250 μs) motor output

In the code the motor value is internally in the range [-100:100] and is then converted to a PPM signal sent to the ESCs. The PPM signal is generated using dedicated PWM hardware inside the microcontroller. The width of the PPM signal determines the speed. In this case a pulse of 125 μs is the minimum speed while a pulse of 250 μs is the maximum speed, as shown on figure 4.5. Note that the microcontroller will send out the signal immediately once it is calculated ignoring the period. Thus, the PPM signal will be synchronous with respect to the rest of the flight controller. This is also known as OneShot125.

4.4 RC transmitter

Figure 4.6 shows the RC (Remote Control) transmitter used for this project. This allows the operator to control the quadcopter using the sticks and enable and disable different flight modes using the switches.

The throttle and rudder are controller by the y- and x-position of the left stick input respectively. While the right stick controls the aileron and elevator.

The aileron, elevator, and rudder inputs control the roll, pitch, and yaw, respectively. Thus, by moving the rudder to the right the quadcopter will start to yaw clockwise.

The values sent from the transmitter are normally referred to as channels. In total six channels are used in this project. Four are used for the stick inputs

and two are used for the auxiliary switches noted as AUX1 and AUX2 to enable and disable different flight modes.

Figure 4.6: The transmitter used for this project

4.5 RC receiver

A RC receiver is used to convert the RF signal sent from the transmitter into pulses corresponding to the different channels. The output is sent using CPPM (Coherent Pulse Position Modulation), as shown in figure 4.7. The distance between two rising edges represents the input from each channel.

In order to measure the time between two rising edges a *timer input capture interrupt* is used. It is configured to call an interrupt routine on every rising edge. The interrupt routine then simply subtracts the previous timer value from the current one. This value is then converted into μs. The value of each channel can be read from the main-loop using a function, which outputs the value of the channel in the range [-100:100].

A second timer is operating as a periodic timer in order to detect if the connection to the receiver breaks, which could happen in a crash. The timer is

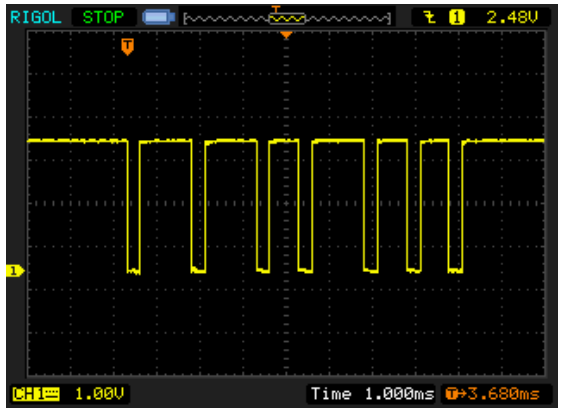


Figure 4.7: 6-channel CPPM output sent from the receiver

configured to overflow every 100 ms and is reset in the *timer input capture interrupt* routine. Thus, if no valid signal is received at least every 100 ms all motors are shut off and the flight controller is disarmed.

4.6 Safety

It is important to take safety into account when working on a system like this, as the propellers are spinning at several thousands RPM. In worst case they could do severe damage to both property and people. The propellers should therefore not be mounted until one has made certain that everything is working properly.

It is highly recommended to run the ESC's calibration routine without the propellers on as if something goes wrong the motors might spin up at its maximum revolution for several seconds. For that reason the ESC's calibration routine is disabled in the code by default and the user will need to activate, compile and upload it manually.

I would recommend doing an initial test without the propellers with a bit of throttle applied. The quadcopter's functionality can then be confirmed by tilting the quadcopter in all three axis and observing that it behaves as intended.

Furthermore one must also consider what happens if the connection to the transmitter is lost. In this case my receiver simply outputs the minimum values for all channels. This is commonly known as fail safe.

The on-board LEDs are also used to indicate whenever the flight controller is armed or not i.e. it lights up the green LED if everything is running correctly and switches to red once the flight controller is armed. This gives some useful visual indication to the operator and potential spectators.

Chapter 5

Flight modes

In total their are four different flight modes supported by the flight controller. The first one is rate mode and uses only the gyroscope to stabilize the quadcopter. This mode is mainly used for advanced pilots and acrobatic manoeuvres. In this mode the aileron and elevator stick inputs indicate the desired rotation rate of the quadcopter. Thus, if the user wants the quadcopter to rotate fast clockwise along its roll axis the aileron input can be put all the way to the right.

5.1 Self-level mode

The second mode is self-level mode, which is activated when the AUX1 channel is more than -10. In this mode the aileron and elevator stick inputs control the roll and pitch angle, respectively. The quadcopter will go back to horizontal once the user centers the sticks. This is especially useful for beginners and if autonomous flight is to be implemented in the future.

5.2 Heading hold mode

The yaw angle, also know as heading, is used to keep the quadcopter fixed at the same heading if no rudder input is applied. This mode uses the gyroscope and magnetometer in order to estimate the yaw angle. It can not use the gyroscope readings alone, as they will drift over time due to integration of the error explained in chapter 3.2.

Heading hold is activated if the AUX1 channel is above 50.

5.3 Altitude hold mode

Altitude hold is activated by setting the AUX2 channel to more than 0. Note that self-level mode needs to be activated as well for it to activate. It is recommended to use heading hold mode in this mode as well, so the user does not have to apply any throttle or rudder input.

In this mode the position of the throttle stick corresponds to the absolute height of the quadcopter i.e. if the throttle stick is all the way up, which makes it stay at a height of approximately 1.5 m while it will stay at a height of 5 cm if the throttle is all the way at the bottom. For now only the height estimated using the ultrasonic sensor is used, as due to time constraints I never got to implement altitude hold based on the barometer.

I limited the maximum height to 1.5 m as this proved to be the practical limitation of the sensor even though the specification for the ultrasonic sensor said that the maximum distance is 3 m. The sensor starter to have several false readings. This is especially critical due to the relative slow update rate of only 40 Hz.

Chapter 6

Software implementation

The firmware for the flight controller is written in C and is compiled using GNU Tools for ARM Embedded Processors (gcc-arm-none-eabi) version 4.8.4. A bundled Makefile allows one to easily compile and flash the firmware.

The Tiva Firmware Development Package revision 1.1 written for the TI Tiva EK-TM4C123GXL Launchpad is used throughout the code. This allows me to easily use the different peripherals without the need of writing directly to the registers [11].

Due to the extend of the code it can be found on the attached CD-ROM or at the following page: `https://github.com/Lauszus/LaunchPadFlightController`.

In the following section the behaviour of the main-loop will be described. A more low-level explanation of the different aspects of the code will not be presented, but the reader is encouraged to study the code.

6.1 Main-loop

Figure 6.1 shows a flowchart of the main-loop. The first step is to initialize the PLL (Phase Locked Loop) so the 16 MHz external crystal frequency is multiplied up to 80 MHz. This is then used as the main clock frequency for the microcontroller.

After that it initializes all the different peripherals and the 24-bit System Timer (SysTick) is set to increment every 1 μs. This is used for basic time measurements. The sensors are then initialized and the gyroscope is calibrated. This is done by taking several measurements and then finding the average of these values; thus, this will correspond to the values of the gyroscope at rest. Since the user might move the flight controller, the range between the measurements is checked. If the range is too large the routine is run again until the quadcopter is kept still during the calibration routine.

The magnetometer self-test is then run in order to calculate the gain used to compensate for differences in the sensor along the three axes [8, p. 19].

Next the buzzer is beeped briefly in order to indicate that the initialization is done.

The flight controller then checks whenever the user has armed it or not. The user can arm the flight controller by applying minimum throttle and positioning the rudder all the way to the right. The flight controller can be disarmed again at minimum throttle and if the rudder is at the leftmost position. If it is armed it turns on the red LED, if not it turns on the green LED.

It then checks if new accelerometer and gyroscope measurements are available by reading the current state of the INT pin. If this is the case it reads these values and checks if the magnetometer has new measurements ready as well by reading the state of the DRDY pin. The gyroscope, accelerometer, and magnetometer values are then used in order to estimate roll, pitch, and yaw.

The current altitude is then estimated. This is done by first triggering the sonar if it is more than 25 ms since it was last triggered. It then checks if a new measurement has been read from the sonar. If this is the case it will convert it into mm . Next the barometer is read in case new measurements are available and the altitude estimate is then updated according to chapter 3.4. This estimate can be used instead of the one measured using the sonar if the quadcopter gets above the range of the sonar sensor i.e. 3 m.

If the flight controller is disarmed or if throttle is at minimum it reads any

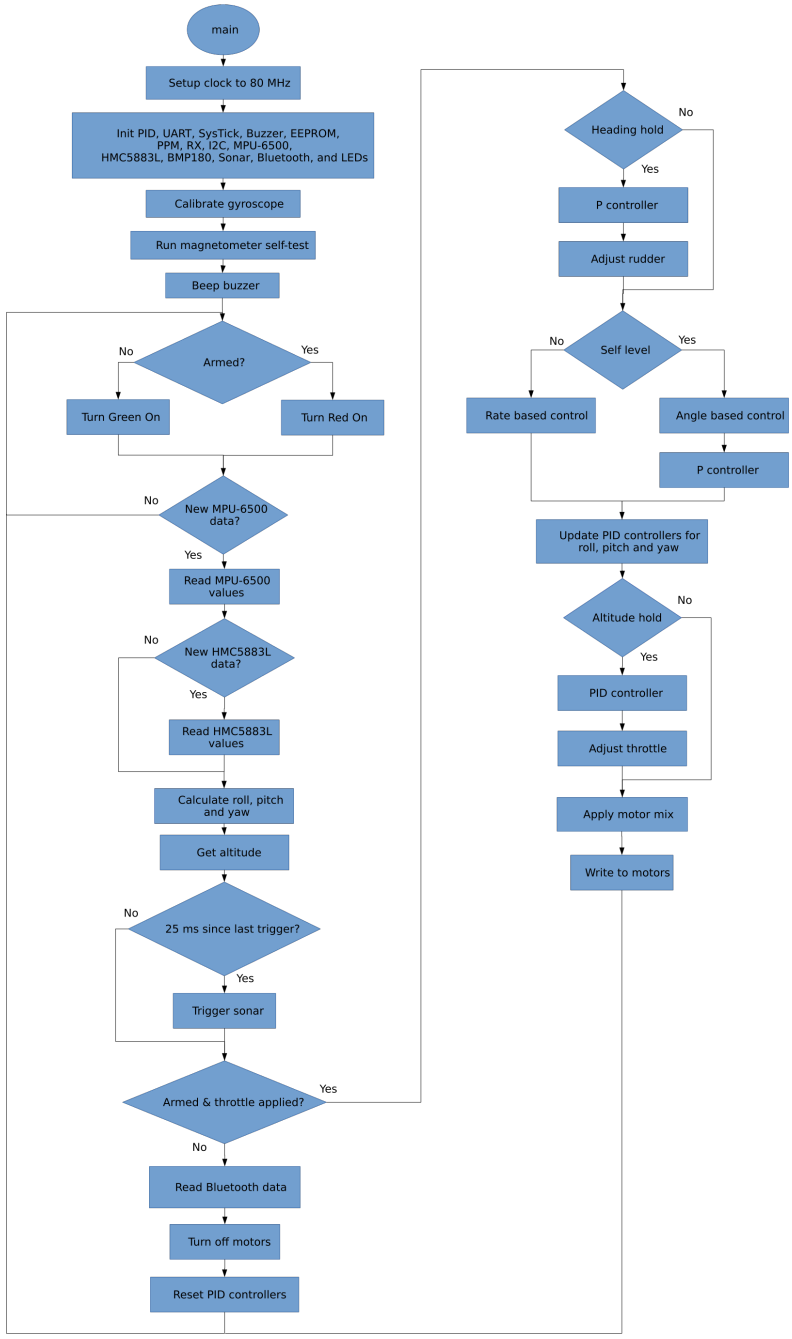


Figure 6.1: Flowchart for main-loop

new data sent via Bluetooth, turns off all motors and resets the different PID controllers. Thus, the Bluetooth data is only read while the motors are not running. This is to prevent the limited speed of the Bluetooth transfer to affect the performance of the flight controller.

It then checks whenever heading hold is activated, if so the rudder output is adjusted, which makes the quadcopter hold its heading.

The next step is to check whenever self-level mode is activated. If it is activated the flight controller will use angle based control i.e. the position of the aileron and elevator sticks determines the roll and pitch angles of the quadcopter, respectively.

If self-level mode is not activated it is in so-called rate mode. In this mode the aileron and elevator sticks determines the rotation rate of the quadcopter.

It then checks if altitude hold is activated, if it is the case it adjusts the throttle value accordingly in order for the quadcopter to stay at a fixed height.

Finally, the output is written to the four motors and the main-loop is run again.

Chapter 7

PID controllers

In total, four PID controllers and three P controllers are used throughout the code. All the coefficients are saved in the EEPROM. This allows the controllers to be easily tuned without the need for the code to be re-uploaded. In order to tune the parameters an Android application was written, which will be discussed in chapter 8.

Figure 7.1 shows a block diagram of the controllers implemented in Simulink. All the different inputs to the controllers including the two channels AUX1 and AUX2 are shown. The two channels AUX1 and AUX2 are used to activate and deactivate the different flight modes.

Figure 7.2 shows the inside of the P controllers for the self-level and heading hold. As it can be seen it is essentially just three P controllers, which will pass through the set point if they are deactivated. Figure 7.3 shows the diagram for the P controller used for self-level along roll.

7.1 Stabilization in rate mode

Three of the PID controllers are used to stabilize the quadcopter along its three axis i.e. roll, pitch, and yaw. The input is the output from the gyroscope

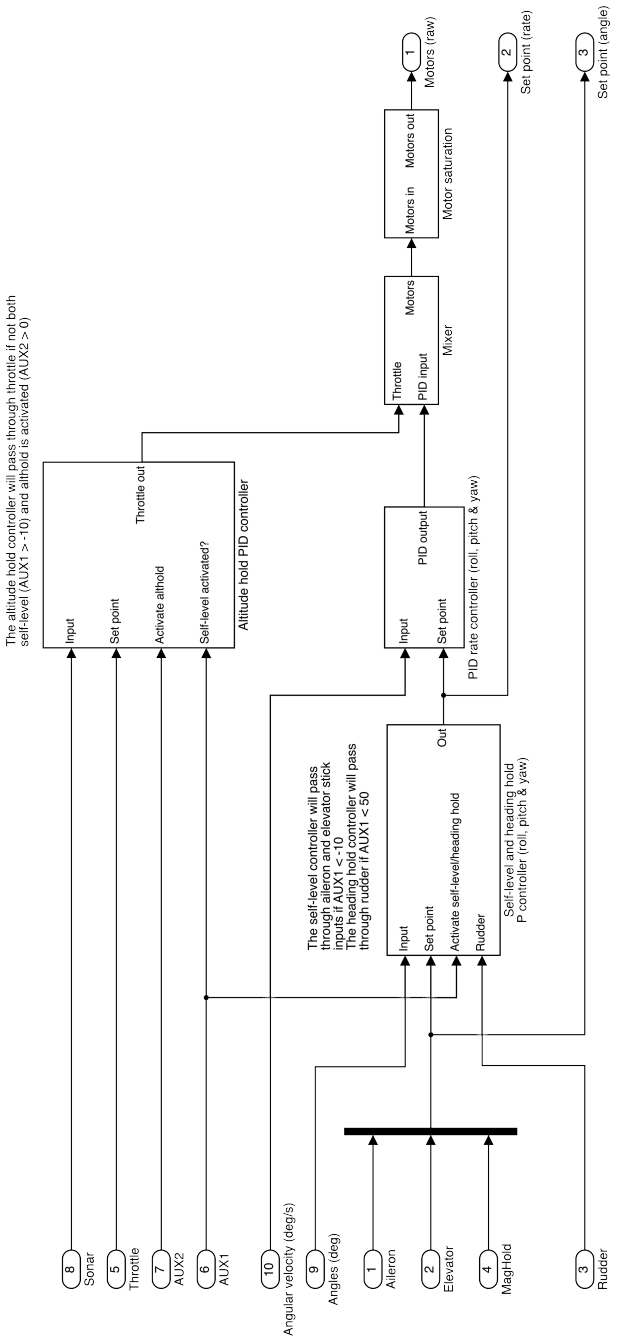


Figure 7.1: Block diagram of the controllers

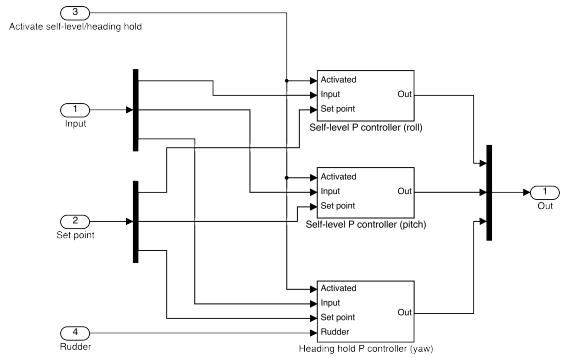


Figure 7.2: Self-level and heading hold P controller (roll, pitch & yaw)

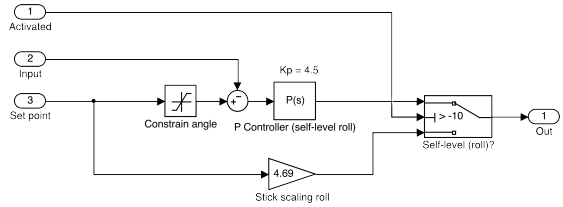


Figure 7.3: Self-level P controller (roll)

in deg/s while the set point in rate mode is the scaled stick input. As the quadcopter is almost symmetric, the same values are used for both the roll and pitch PID coefficients.

The outputs from the three different PID controller are added or subtracted to the throttle stick input in such a way that it stabilizes the quadcopter, as described in chapter 7.6. For instance, if the quadcopter starts to pitch down, the pitch PID controller will speed up the two front motors and slow down the two back motors with the same amount. The result is that the quadcopter will start to pitch up again. In the same way the speed of the two right motors are increased and the two left motors are decreased if the quadcopter starts to roll clockwise.

The yaw axis is stabilized by the fact that the bottom right and top left motors both spin clockwise while the other two motors spins counter-clockwise. Thus, by applying more speed to any pair will result in a torque along the yaw axis due to Newton's third law. The result will be that the quadcopter will start to rotate along its yaw axis.

7.2 Stabilization in self-level mode

Two cascaded P controllers are used for self-level mode, one for roll and one for pitch. The error between the desired angle, which is determined from the aileron and elevator sticks, and the current angle is found and is then multiplied by a constant (Kp). The output from the two P controllers are used as the set point for the respective PID controllers originally used for rate mode. Since it is only a P controller the steady-state error will not be eliminated. In practice this is not a problem, as the quadcopter is meant to be user operated.

7.3 Stabilization in heading hold mode

A third P controller is used for heading hold mode. It takes the estimated yaw angle using the magnetometer and gyroscope and subtracts it from the desired heading. This value is then normalized, so 0 degrees is forward and ± 180 degrees is backward. The difference is then multiplied by a constant (Kp) and added to the rudder input. The result is that the quadcopter will yaw until the desired heading is obtained.

The desired heading is set equal to the current heading every time the user applies a rudder input of more than ± 5. This ensures that the user still has rudder control despite heading hold being active. The status can be seen on the blue LED, as it will light up when the P controller is running.

7.4 Stabilization in altitude hold mode

The fourth PID controller is used in altitude hold mode, which allows the quadcopter to stay at a fixed height. It uses the distance above ground measured using the ultrasonic sensor as the input. The set point is determined from the throttle position. The output from the PID controller is added to the initial throttle value; that is the throttle value when altitude hold mode was activated.

As the user can move the throttle stick much faster up and down than the system is able to respond, the throttle input is not used directly. Instead, it is first low-pass filtered and then converted to a corresponding height. The result is that even a fast throttle movement leads to a smooth transaction to a different height.

7.5 PID contoller implementation in code

The PID controller update routine is implemented in the code as shown in listing 7.1. There is some aspects that is worth mentioning. The first is the fact that trapezoidal approximation [18] is used to integrate the error for the I-term. The I-term is then constrained in order to prevent integration windup.

Next the difference between the current and last error is found, which is used for the D-term. A moving average is used to reduce the noise, this makes the D-term a little easier to tune and less sensitive to vibrations.

Finally the sum of the P-, I-, and D-terms are returned as the output.

```
1  float updatePID(pid_t *pid, float setPoint, float input, float dt) {
2      float error = setPoint - input;
3
4      // P-term
5      float pTerm = pid->values->Kp * error;
6
7      // I-term
8      // Use Trapezoidal Integration, see: http://ecee.colorado.edu/shalom/
           Emulations.pdf
9      // Multiplication with Ki is done before integration limit, to make it
           independent from integration limit value
10     pid->iTerm += pid->values->Ki * (error + pid->lastError) / 2.0f * dt;
11     // Limit the integrated error - prevents windup
12     pid->iTerm = constrain(pid->iTerm, -pid->values->integrationLimit, pid->
           values->integrationLimit);
13
14     // D-term
15     // Calculate difference and compensate for difference in time by dividing
           by dt
16     float deltaError = (error - pid->lastError) / dt;
17     pid->lastError = error;
18     // Use moving average here to reduce noise
19     float deltaSum = pid->deltaError1 + pid->deltaError2 + deltaError;
20     pid->deltaError2 = pid->deltaError1;
21     pid->deltaError1 = deltaError;
22     float dTerm = pid->values->Kd * deltaSum;
23
24     return pTerm + pid->iTerm + dTerm; // Return sum
25 }
```

Listing 7.1: Function used to update the PID controllers

7.5.1 PID controller tuning

In order to tune the different PID controllers a method inspired by the Ziegler-Nichols method was used [12, p. 292-299]. Initially all coefficients were set to zero. Then K_p was then increased until the system started to oscillate. The

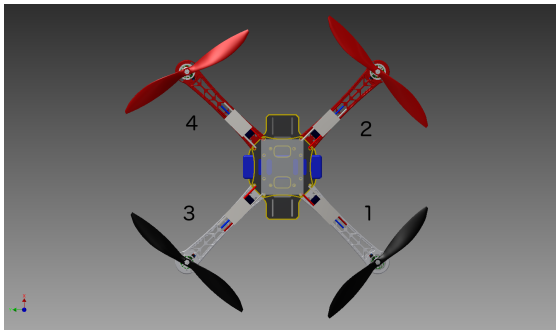


Figure 7.4: 3D rendering of quadcopter. The numbers indicates the corresponding motor number

K_p value was then decreased until the oscillation stopped. Next the K_i value was increased until the system started to oscillate again, but this time it should have a longer period due to the overshoot caused by the I-term. K_i was then decreased again to make it go away. K_d was then increased to dampen the system and the procedure was repeated until the system behaved as intended. The values were then further tuned by looking at the step response of the real system, which will be discussed in chapter 9. The final coefficients used for the different controller can be found in appendix C.

7.6 PID mixer

The output from the PID controllers is written to the motors in such a way that it stabilized the quadcopter. Figure 7.4 shows a 3D rendering of the quadcopter. As it can be seen the x-axis is pointing forward, the y-axis is pointing to the left, and the z-axis is pointing upward. The motors are labeled as well; thus, by increasing the speed of motor 1 and 3 and decreasing the speed of motor 2 and 4, the quadcopter will start to pitch down. Listing 7.2 shows how this is implemented in code and is often referred to as a mixer. Note that the *motor[0]* represents motor 1, as the code starts from index 0.

```
1 float motors[4]; // Motor 0 is bottom right, motor 1 is top right, motor 2 is
      bottom left and motor 3 is top left
2 for (uint8_t i = 0; i < 4; i++)
3     motors[i] = throttle;
4
5 // Apply mix for quadcopter in x-configuration
6 motors[0] -= rollOut;
7 motors[1] -= rollOut;
8 motors[2] += rollOut;
```

```
motors[3] += rollOut;

motors[0] += pitchOut;
motors[1] -= pitchOut;
motors[2] += pitchOut;
motors[3] -= pitchOut;

motors[0] -= yawOut;
motors[1] += yawOut;
motors[2] += yawOut;
motors[3] -= yawOut;

updateMotorsAll(motors);
```

Listing 7.2: Mixer

Furthermore, to ensure that the quadcopter still behaves properly if the motors reach the maximum output value, which could happen during extreme manoeuvres, the maximum value sent to the motors are found. If this value exceeds the maximum output value, all outputs are scaled in such a way that the difference between all outputs is still the same. Listing 7.3 shows how this was implemented in code.

```
void updateMotorsAll(float *values) {
    // Find the maximum motor output
    float maxMotor = values[0];
    for (uint8_t i = 1; i < 4; i++) {
        // If one motor is above the maxthrottle threshold, we reduce the
            value
        // of all motors by the amount of overshoot. That way, only one motor
        // is at max and the relative power of each motor is preserved
        if (values[i] > maxMotor)
            maxMotor = values[i];
    }

    if (maxMotor > MAX_MOTOR_OUT) {
        for (uint8_t i = 0; i < 4; i++)
            values[i] -= maxMotor - MAX_MOTOR_OUT; // This is a way to still
                have good gyro corrections if at least one motor reaches its
                max
    }

    for (uint8_t i = 0; i < 4; i++)
        updateMotor(i, values[i]); // Write output to ESCs
#if ONESHOT125
    syncMotors();
#endif
}
```

Listing 7.3: Mixer

Chapter 8

Android application

An Android application was written in order to adjust the various parameters used for the PID controllers and various other settings. The source code is written in Java and is available under the GPLv2 license at: `https://github.com/Lauszus/LaunchPadFlightControllerAndroid` and on the attached CD-ROM.

The Android application communicates with the flight controller via Bluetooth. This allows the user to adjust the different parameters without the need to reprogram the flight controller, as the different parameters are all saved in the microcontrollers internal EEPROM. This is especially useful when tuning the parameters on a new quadcopter and should allow the user to get it flying within minutes.

Figure 8.1 shows and an overview of the application.

8.1 Data protocol

A simple protocol inspired by the MultiWii project [4] was implemented in order to send data back and forth reliably. Table 8.1 and 8.2 shows the structure of the

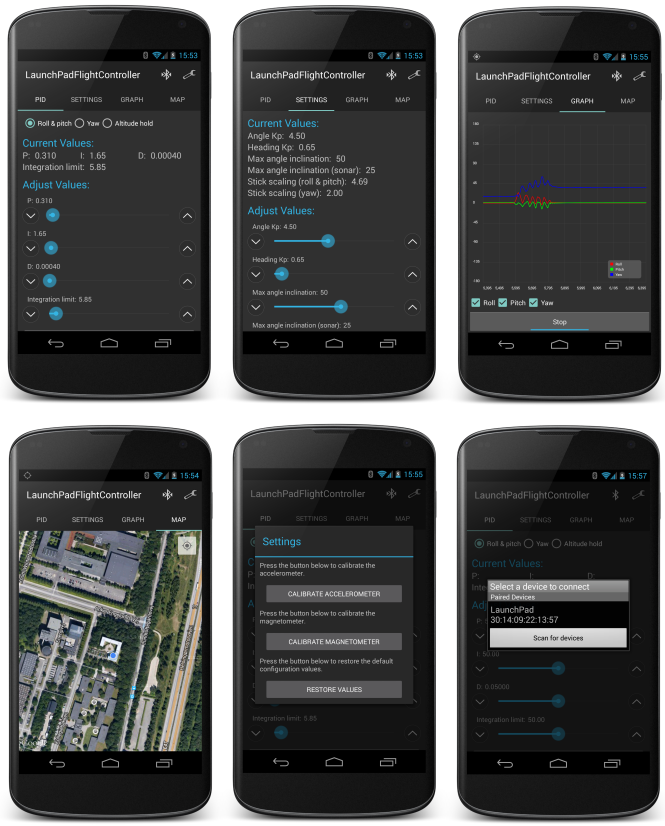


Figure 8.1: Overview of Android application

two Bluetooth commands used to request data and respond to a message. The Bluetooth request command is used by the Android application to set new PID values, get the current PID values etc. The flight controller will then respond with a message containing the relevant data.

The first three bytes contains a fixed heading, which is used by the parser in the Android application and flight controller to determine the beginning of each message. The only difference between the two headings is that byte 2 is different in order to distinguish the two types of messages.

Byte 3 contains the command (cmd) parameter, which sets the information that is requested. In total 12 different command parameters are defined.

Byte[0]	Byte[1]	Byte[2]	Byte[3]	Byte[4]	Byte[5]	Byte[6]	...	Byte[x]	Byte[x + 1]
'$'	'S'	'>'	cmd	length (n)	Data[0]	Data[1]	...	Data[n - 1]	checksum

Table 8.1: Message request command

Byte[0]	Byte[1]	Byte[2]	Byte[3]	Byte[4]	Byte[5]	Byte[6]	...	Byte[x]	Byte[x + 1]	Byte[x + 2]	Byte[x + 3]
'$'	'S'	'<'	cmd	length (n)	Data[0]	Data[1]	...	Data[n - 1]	checksum	'\r'	'\n'

Table 8.2: Message response command

Byte 4 is the length (n) of the data to follow and depends on the cmd parameter.

The next n-bytes will contain the actual data. Note that all floating point numbers are converted to integers before transmission, as this is needed as the floating point numbers might be implemented differently on the two platforms. It also allows the two systems to use bitwise operation, which are useful when splitting larger data types into bytes.

The next byte is a simple LRC (Longitudinal Redundancy Check) checksum and is generated by taking the XOR of the command (cmd), the length, and the data sent. A LRC checksum should be sufficient by an application like this. It has the benefit that it can be generated easily without the need for lengthy computational time or large lookup tables, which is a disadvantage of other algorithms like CRC (Cyclic Redundancy Check).

The flight controller also sends out a carriage return and line feed, which makes it easier to pass the received string in Java.

The first six command parameters are used to set and get the PID parameters for the PID controllers used for roll & pitch, yaw, and altitude hold. As shown on the first tab in the Android application. The roll and pitch PID controllers both share the same parameters do the near-symmetric property of the quadcopter.

The next set of commands is used by the settings tab and is used to adjust the K_p coefficients for the P controllers used in self-level mode and heading hold mode. The maximum inclination angle used in self-level mode and altitude hold mode can also be adjusted. Finally the stick scaling can be modified. The stick scaling determines the maximum rotation rate of the quadcopter in rate mode; thus, a large stick scaling value will allow the quadcopter to turn faster around its axes.

The next command is used by the Android application in order to make the flight controller send out the estimated roll, pitch, and yaw angles of the quadcopter. This is then visualized on a graph in real time on the Android application.

Furthermore it can also start the accelerometer and magnetometer calibration routines. The accelerometer calibration routine should be performed while the quadcopter is horizontal and is used to compensate if the flight controller is tilted relative to the quadcopter frame. The magnetometer calibration runs for 30 s and the user needs to rotate the 360° along all axes i.e. roll, pitch, and yaw. This is needed in order to measure the minimum and maximum value for each axis and, thereby, to find the zero value.

Finally, a command parameter to restore the default configuration parameters is also implemented.

Part III

Part 3 - Evaluation & conclusion

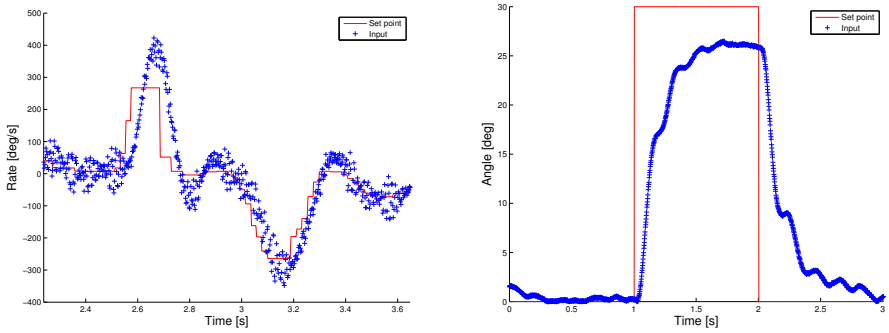
Chapter 9

Performance & measurements

In this chapter the practical response for some of the different controllers will be presented and evaluated. The data was logged by sending out the values in real time via Bluetooth to a computer. The data were then plotted and analysed using Matlab.

Note that the x-axis in the figures in this chapter does not represent the true time, but rather the estimated time used internally by the flight controller.

Figure 9.1 was generated by logging the input and set point of the roll PID rate controller every 1 ms. The input represents the gyroscope roll in deg/s and the set point is the aileron input from the user. As it is observed there is some overshoot and the response lags a little behind the set point. The lag was measured to be about 50 ms. It is also noticeable that there seem to be some vibrations. These might be reduced by balancing the propellers and motors or by implementing a low-pass filter on the gyroscope data.



(a) Log of quadcopter flying in rate mode

(b) Step response for self-level mode

Figure 9.1

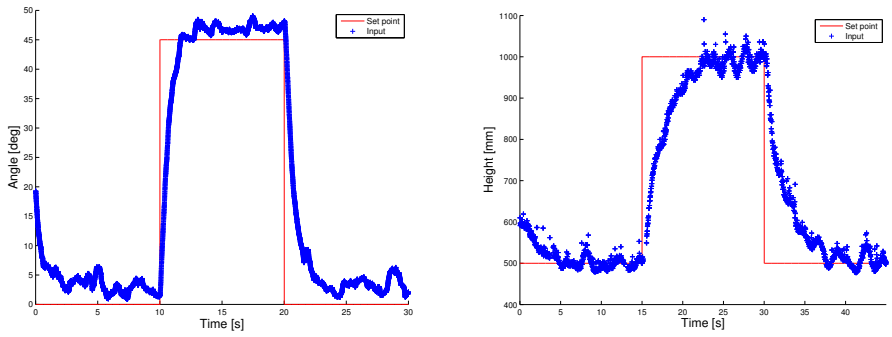
9.1 Practical step response

Figure 9.1 shows a step response for the quadcopter in self-level mode. The step was done in code in order for the transaction to be instant. The set point where first set to 0° for 1 s and then to 30° for 1 s and finally set back to 0°.

As can be seen there is a steady-state error. This makes sense as the self-level mode is only using a P controller, as explained in 7. The steady-state error is measured to be about 4°. Also the step response is not that smooth, which could probably be improved by using a PID controller as well.

Figure 9.2 shows the steps response of heading hold mode when stepping with an angle of 45°, where it can be seen that overshoot is about 4° and the steady-state error is about 2°. Again the step response could probably be improved upon by using a PID controller instead of just a P controller.

Figure 9.2 shows the step response of the altitude hold using the distance measured by the ultrasonic sensor as the input to the altitude hold PID controller. The set point is first set to 500 mm in the code for 15 s, then to 1500 mm and then back to 500 mm after 15 s. As it can be seen it takes about 10 s for the quadcopter to reach the desired height. It then oscillates around that point until it returns back to 500 mm. Again, it lasts about 10 s for it to reach the desired height. I suspect the small oscillation of ±40 mm is due to the slow update rate of the ultrasonic sensor of only 40 Hz as explained in chapter 4.2. Thus, it could be interesting to use a sensor with a higher update rate like a laser. It can also be seen that the sonar sensor sometimes returns an error reading, as it seems to be the case at approximately 23 s in figure 9.2. This may be prevented by using



(a) Step response for heading hold mode **(b)** Step response for altitude hold mode

Figure 9.2

a more high-end distance sensor or by applying a low-pass or median filter to the ultrasonic data.

9.2 Altitude results from barometer

The altitude estimation from the barometer and the accelerometer was implemented according to chapter 3.4, but due to time constrains the implementation of altitude hold based on the barometer was never finished before the submission of this thesis. However the estimated altitude will be analysed in this section and will be the basis for a future implementation.

The estimated altitude were logged every 25 ms as shown in figure 9.3. The green values represent the height found using 3.30, as described in chapter 3.4. However as the output was distorted by noise I decided to low-pass filter these values, which are represented by the blue values. Furthermore, the true ground height were found by finding the difference between the height estimated using the barometer & accelerometer and the height found using the sonar. This offset was then subtracted from any further measurements once the height were above 10 cm. Below 10 cm the height were set equal to the sonar height. It should also be noted that I ended up applying a low-pass filter to the barometer height estimate 3.24, as the estimate was inaccurate due to noise.

In figure 9.3 the quadcopter were raised by hand from 0 m to 1 m, then up to 2 m, then back to 1 m, then up to 2 m again and finally back to 0 m.

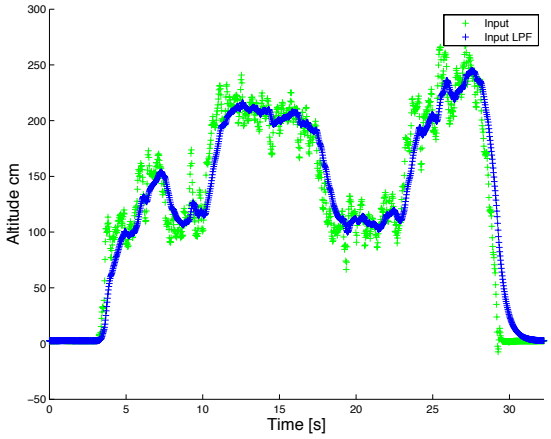


Figure 9.3: Altitude indoor test

Figure 9.4 shows the height logged while flying by an operator. First it was controlled to hover at approximately 2 m, then at 4 m, then at 7 m, then all the way up to 10 m, then back to 5 m, then down to 2 m and finally it was landed. Note that the absolute value should be taken with a grain of salt because the height were just approximated by the operator.

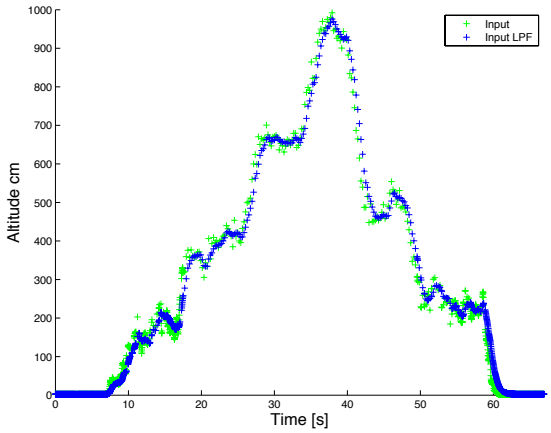


Figure 9.4: Altitude flight test

As it can be seen from figure 9.3 and 9.4 the low-pass filtered values seems to be a good estimation and should be able to fulfill my initial goal to make the quadcopter stay at height with no more than 2 m drift.

The Matlab code used to plot the figures in this chapter can be found in appendix A.

9.3 Theoretical model

Based on the space equations found in part 1 a Simulink model of the system was created, as shown in figure 9.5. By connecting this model to the model of the controller shown in figure 7.1 the entire system can be simulated.

The simulation assumes that the discrete flight controller implementation can be approximated by a continuous system. A picture of the full Simulink model can be seen in appendix E. The full Simulink model can be found on the attached CD-ROM.

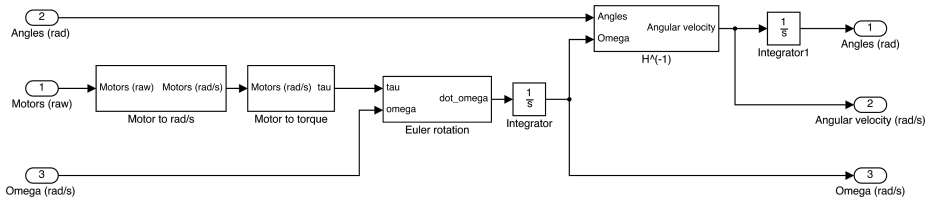
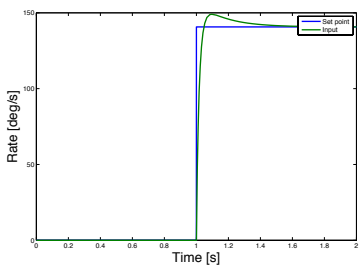


Figure 9.5: Block diagram of the system

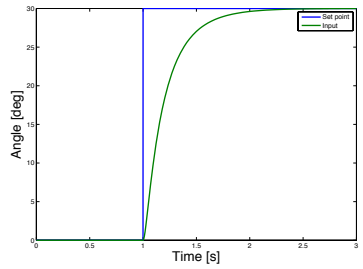
9.3.1 Theoretical step response

Figure 9.6 shows the roll step response while in rate and self-level mode. It can be seen that the theoretical step response of the rate PID controller also overshoots the set point, just as the practical measurements shown. The rise time is very fast in this case it was measured to 50 ms, which is more or less identical to the practical system. A fast rise time is desirable as it allows the pilot to do flips and other manoeuvres without loosing control.

Comparing the step response for self level mode it can be seen that the rise time is about the same. However their is no steady-state error in the theoretical step response. This is possible due to the assumption that the theoretical system is perfect symmetric and the motors and propellers are vibrations free. Furthermore it looks like the real system is a higher order than the simulated system, as it their is clearly ripples on the waveform shown in figure 9.1.



(a) Theoretical step response for roll PID rate controller



(b) Theoretical step response for roll self-level controller

Figure 9.6

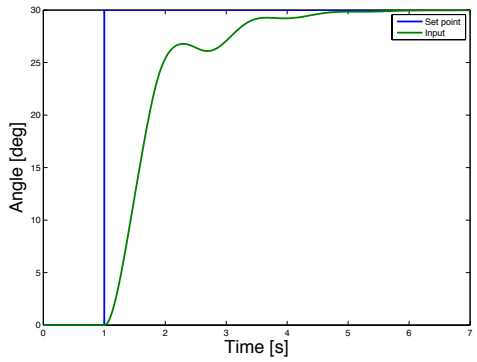


Figure 9.7: Theoretical step response for yaw heading hold controller

Figure 9.7 shows the theoretical step response for heading hold. The rise time is roughly the same as the practical system, but the theoretical response got no steady-state error with no overshoot. Again I believe it is due to the symmetric and vibration assumptions made in the theoretical model.

Chapter 10

Results & discussion

As shown in the previous chapter the response for the different flight modes behaved as intended. Some aspects could be improved, most notably the elimination of the steady-state error in self-level and heading mode by using a PID controller instead of a P controller. However as the quadcopter is intended to be controlled by a human the steady state error is not a problem in practice.

My step response for altitude hold behaved just as intended and fulfilled my initial goal of staying at a height of $\pm 10\,\text{cm}$ when using the ultrasonic sensor, as the oscillations were only $\pm 40\,\text{mm}$. The only problem that I encountered was that the ultrasonic sensor sometimes gives an incorrect reading, especially when the quadcopter gets close to the limit of the sensor at 3 m. This might be improved by applying a low-pass filter or by estimating the velocity by subtracting any new values with the previous one. The estimated velocity could be used as a sanity check, as the height could be discarded if the velocity exceeded a certain value.

The height estimated using the barometer and accelerometer worked very well after I further low-pass filtered the estimated output. I should be able to use this for altitude hold when the quadcopter get out of the ultrasonic sensors limited range of only 3 m. Unfortunately due to time constraints I never got to implement it.

The step response of the theoretical system showed that the model did not behave exactly as the practical responses measured on the actual system. However the rise time was very similar for all responses. I believe that these differences could be eliminated by expanding the theoretical model.

I am content with the final result, as the quadcopter flies just as well, if not better, than other commercially available flight controllers that I have previously used. The fast rise time might be one of the reasons as it makes the quadcopter very responsive. This allows me to make flips and other acrobatic manoeuvres. One of the reasons could be that the Android application enables me to tune and adjust different settings rapidly, so the tedious aspect of tuning the flight controller becomes more enjoyable.

Demonstration videos of the quadcopter flying can be found on the attached CD-ROM.

10.1 Future work

As mentioned earlier I still need to implement altitude hold using the height estimated by the barometer and acceleration. This would mean that I need to implement another PID controller to control the velocity in the z-axis. This is need as the height estimated from the barometer and acceleration is not limited to a certain height as it is the case when using the ultrasonic sensor.

Another fun improvement would be to implement auto flip; thus, the quadcopter would be able to rotate 360°, either along roll or pitch by command.

It would also be interesting to use an optical flow sensor to estimate the change in x- and y-position of the quadcopter. An optical flow sensor, like the one in an optical mouse, could be used to measure the change in position. In combination with altitude hold this would allow the quadcopter to stay in the same altitude and position close above ground without the user needing to adjust the position. This is especially useful for indoors operation where a GPS signal is typical limited.

I have considered designing a custom PCB with the microcontroller and all the required sensors and components on it, so it could be made smaller and more user-friendly. A potential commercial option may lie in a production of an assembly kit.

Chapter 11

Conclusion

This thesis has examined the different aspects needed in order to develop a flight controller used for a quad rotor helicopter in x-configuration.

I conducted a theoretical analysis of the system and described the equation used to estimate the attitude and altitude.

Based on this analysis the system was implemented on a microcontroller using a variety of sensors. This allowed a person to operate and fly the quadcopter in several different modes depending on the desired behaviour. An Android application were also written allowing the different parameters to be adjusted easily.

The flight controller were then analysed by logging the step response for different parts of the system. Based on the theoretical model the system could be simulated in Simulink and compared to the practical system. The simulation did match the real system, but showed some discrepancies that were possible due to simplifications and assumptions made in the theoretical model.

All initial goals were archived except getting altitude hold implemented using the barometer. The final product behaved as designed and I would have no concerns about others using the developed flight controller, as it performs very well.

Bibliography

[1] André B. Ø. Bertelsen and Mikkel Wessel Thomsen. *Modelling and L1 adaptive control of a quadcopter*. February 2014.

[2] Jim Bethel. CE503 Rotation Matrices - Derivation of 2D Rotation Matrix. `https://engineering.purdue.edu/~bethel/rot2.pdf`, 2005. Purdue University - School of Civil Engineering. Visited: 15. June 2015.

[3] cplusplus. atan2 - C++ Reference. `http://www.cplusplus.com/reference/cmath/atan2`, 2015. Visited: 15. June 2015.

[4] Alexandre Dubus. MultiWiiCopter. `http://www.multiwii.com`, 2015. Visited: 1. June 2015.

[5] National Geographic. 5 Surprising Drone Uses (Besides Amazon Delivery). `http://news.nationalgeographic.com/news/2013/12/131202_drone_uav_uas_amazon_octocopter_bezos_science_aircraft_unmanned_robot`, December 2013. Visited: 23. June 2015.

[6] Andrew Gibiansky. Quadcopter Dynamics and Simulation. `http://andrew.gibiansky.com/blog/physics/quadcopter-dynamics`, November 2012. Visited: 20. June 2015.

[7] Herbert Goldstein, Charles P. Poole Jr., and John L. Safko. *Classical Mechanics*. Addison-Wesley, 3rd edition edition, June 2001. ISBN: 02-016-5702-3.

[8] Honeywell International Incorporated. 3-Axis Digital Compass IC HMC5883L. `http://www51.honeywell.com/aero/common/documents/myaerospacecatalog-documents/Defense_Brochures-documents/`

HMC5883L_3-Axis_Digital_Compass_IC.pdf, February 2013. Rev. E, #900405. Visited: 10. June 2015.

[9] InvenSense Incorporated. MPU-6500 Product Specification Revision 1.0. http://store.invensense.com/datasheets/invensense/MPU_6500_Rev1.0.pdf, September 2013. Visited: 10. June 2015.

[10] Texas Instruments Incorporated. Tiva™ TM4C123GH6PM Microcontroller - Data Sheet. http://www.ti.com/lit/ds/symlink/tm4c123gh6pm.pdf, June 2014. Visited: 10. June 2015.

[11] Texas Instruments Incorporated. Tiva™ C Series LaunchPad Evaluation Board Software. http://www.ti.com/tool/sw-ek-tm4c123gxl, 2015. Visited: 10. June 2015.

[12] Ole Jannerup and Paul Haase Sørensen. *Reguleringsteknik*. Polyteknisk Forlag, 4th edition edition, 2009. ISBN: 87-502-0982-5.

[13] Simon Kirby. Open Source Firmware for ATmega-based Brushless ESCs. https://github.com/sim-/tgy, 2015. bs_nfet with OneShot125 and COMP_PWM = 1. Visited: 1. June 2015.

[14] Magnetic-Declination.com. Find the magnetic declination at your location. http://www.magnetic-declination.com, 2015. Visited: 16. June 2015.

[15] Talat Ozyagcilar. Implementing a Tilt-Compensated eCompass using Accelerometer and Magnetometer Sensors. In . Freescale Semiconductor, January 2012. Rev. 3, AN3461. Visited: 10. June 2015.

[16] Parallax. PING))) Ultrasonic Distance Sensor (#28015). http://che126.che.caltech.edu/28015-PING-Sensor-Product-Guide-v2.0.pdf, 2013. Visited: 10. June 2015.

[17] Mark Pedley. Tilt Sensing Using a Three-Axis Accelerometer. In . Freescale Semiconductor, March 2013. Rev. 6, AN3461. Visited: 10. June 2015.

[18] H. Peng, George T.-C Chiu, and T-C. Tsao. Chapter 6. Design of Discrete Time Controller – Input/Output Approaches. http://ecee.colorado.edu/shalom/Emulations.pdf, Spring 2008. Department of Electrical, Computer, and Energy Engineering - University of Colorado at Boulder. Visited: 15. June 2015.

[19] Bosch Sensortec. BMP180 Digital pressure sensor. http://ae-bst.resource.bosch.com/media/products/dokumente/bmp180/BST-BMP180-DS000-12~1.pdf, May 2015. Rev. 2.8, BST-BMP180-DS000-12. Visited: 10. June 2015.

Appendix A

Matlab scripts

The Matlab script used to plot the graphs in chapter 9 can be seen below:

```
clear all
close all
clc

hold on
if 0
    data = load('acro/acro_manual.txt');
    ylabel('Rate [deg/s]', 'fontsize', 15);
elseif 0
    data = load('self_level/self_level_30.txt');
    ylabel('Angle [deg]', 'fontsize', 15);
elseif 1
    data = load('heading/heading_45deg_10s.txt');
    for i=1:length(data(:,4)) % Normalize data
        if (data(i,4) < -180)
            data(i,4) = data(i,4) + 360;
        elseif (data(i,4) > 180)
            data(i,4) = data(i,4) - 360;
        end
    end
    ylabel('Angle [deg]', 'fontsize', 15);
else
    convert = 1;
    if convert == 1
        data = load('altitude/altitude_step.txt');
        ylabel('Height [mm]', 'fontsize', 15);

        % Used to convert the values back to height
        MIN_MOTOR_OUT = -100;
```

```
        MAX_MOTOR_OUT = 100;
        SONAR_MIN_DIST = 50;
        SONAR_MAX_DIST = 1500;
        data(:,3) = mapf(data(:,3), MIN_MOTOR_OUT, MAX_MOTOR_OUT,...
                                            SONAR_MIN_DIST, SONAR_MAX_DIST);
        data(:,4) = mapf(data(:,4), MIN_MOTOR_OUT, MAX_MOTOR_OUT,...
                                            SONAR_MIN_DIST, SONAR_MAX_DIST);
    else
        data = load('altitude/altitude_step.txt');
    end
end

time = data(:,2) / 1e6; % Convert to seconds
setPoint = data(:,3);
input = data(:,4);

figure(1)
plot(time, setPoint, 'r');
figure1 = plot(time, input, 'b+');
legend('Set point', 'Input')
xlim([min(time) max(time)])
xlabel('Time [s]', 'fontsize', 15);
hold off

saveas(figure1, strcat(pwd, '/img/figure1'),'epsc') % Save figure
```

My *mapf* function can be found below:

```
function val = mapf(x, in_min, in_max, out_min, out_max)

val = (x - in_min) * (out_max - out_min) / (in_max - in_min) + out_min;
if (val < out_min)
    val = out_min;
elseif (val > out_max)
    val = out_max;
end
```

The script used to log the altitude height can be found below:

```
clear all
close all
clc

if 1
    data = load('altitude_indoor.txt');
else
    data = load('altitude_baro_10m.txt');
end

time = data(:,2) / 1e6; % Convert to seconds
time = time - time(1); % Subtract first value, so it starts at 0s

input_lpf = data(:,3);
input = data(:,4);

figure(1)
hold on
plot(time, input, 'g+');
```

```
figure1 = plot(time, input_lpf, 'b+');
legend('Input', 'Input LPF')
xlim([min(time) max(time)])
xlabel('Time [s]', 'fontsize', 15);
ylabel('Altitude cm', 'fontsize', 15);
hold off

saveas(figure1, strcat(pwd, '/img/figure1'),'epsc') % Save figure
```

Appendix B

Motor output to rad/s

The data used to find the linear relationship between the motor output in the range [-100:100] to the angular velocity in $^{\text{rad}}/_{\text{s}}$ can be seen below in figure B.1.

Value	Motor 1	Motor 2	Motor 3	Motor 4	Average (RPM)	Average (Hz)	Average (rad/s)
-80	1579	1568	1554	1570	1567,75	26,13	164,17
-70	2200	2181	2176	2200	2189,25	36,49	229,26
-60	2853	2820	2791	2843	2826,75	47,11	296,02
-50	3464	3409	3407	3461	3435,25	57,25	359,74
-40	4095	4030	3988	4067	4045,00	67,42	423,59
-30	4706	4643	4568	4702	4654,75	77,58	487,44
-20	5307	5214	5107	5293	5230,25	87,17	547,71
-10	5889	5787	5733	5897	5826,50	97,11	610,15
0	6468	6382	6360	6493	6425,75	107,10	672,90
50	9634	9478	9429	9733	9568,50	159,48	1002,01

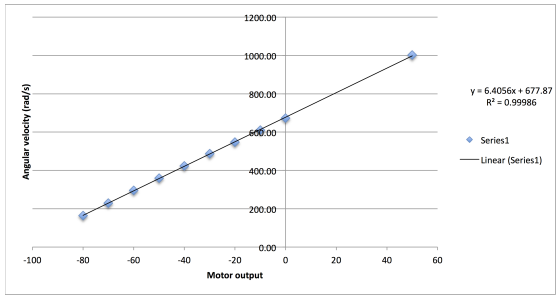


Figure B.1: Motor output to $^{\text{rad}}/_{\text{s}}$ table

Appendix C

PID coefficients

Table C.1 shows the final PID coefficients used for the different controllers.

Controller	**Kp**	**Ki**	**Kd**	**I-term limit**
Roll rate controller	0.310	1.65	0.00040	±5.85
Pitch rate controller	0.310	1.65	0.00040	±5.85
Yaw rate controller	1.000	6.00	0.00040	±10
Altitude hold controller	0.040	0.03	0.00350	±10
Roll angle controller	4.50			
Pitch angle controller	4.50			
Yaw angle controller	0.65			

Table C.1: PID coefficients

Appendix D

Hardware components

A list of all hardware components used in this project can be seen below:

- Tiva C Series TM4C123G LaunchPad
- Gyro & accelerometer (MPU-6500)
- Magnetometer (HMC5883L)
- Barometer (BMP180)
- Ultrasound sensor aka sonar (HC-SR04)
- HC-06 Bluetooth module
- Motors: Dualsky 980kv
- ESC's: Hobby King 20A ESC 3A UBEC (F-20A): `http://hobbyking.com/hobbyking/store/uh_viewItem.asp?idProduct=37253` - flashed with SimonK bs_nfet with OneShot125 and COMP_PWM = 1
- Propellers: 10x4.5
- Frame: Hobby King Q450 V3 Glass Fiber Quadcopter Frame 450mm: `http://hobbyking.com/hobbyking/store/__49725__Q450_V3_Glass_Fiber_Quadcopter_Frame_450mm_Integrated_PCB_Version.html`

- LiPo: Turnigy 3300mAh 3S 30C Lipo Pack: `http://hobbyking.com/hobbyking/store/uh_viewItem.asp?idProduct=35870`
- RX: OrangeRX R615X: `http://www.hobbyking.com/hobbyking/store/__46632__OrangeRx_R615X_Spektrum_JR_DSM2_DSMX_Compatible_6Ch_2_4GHz_Receiver_w_CPPM.html`
- TX: Turnigy 9XR: `http://www.hobbyking.com/hobbyking/store/__31544__Turnigy_9XR_Transmitter_Mode_2_No_Module_.html` - flashed with OpenTX
- TX module: OrangeRX 2.4GHz transmitter module: `http://hobbyking.com/hobbyking/store/__24656__OrangeRX_DSMX_DSM2_2_4Ghz_Transmitter_Module_JR_Turnigy_compatible_.html`

Appendix E

Simulink model

A screenshot of the model can be seen in figure E.1. The manual switches redirect the step input to the different controllers e.g. in the configuration shown, it steps the rate PID roll controller. The full Simulink model can be found on the attached CD-ROM.

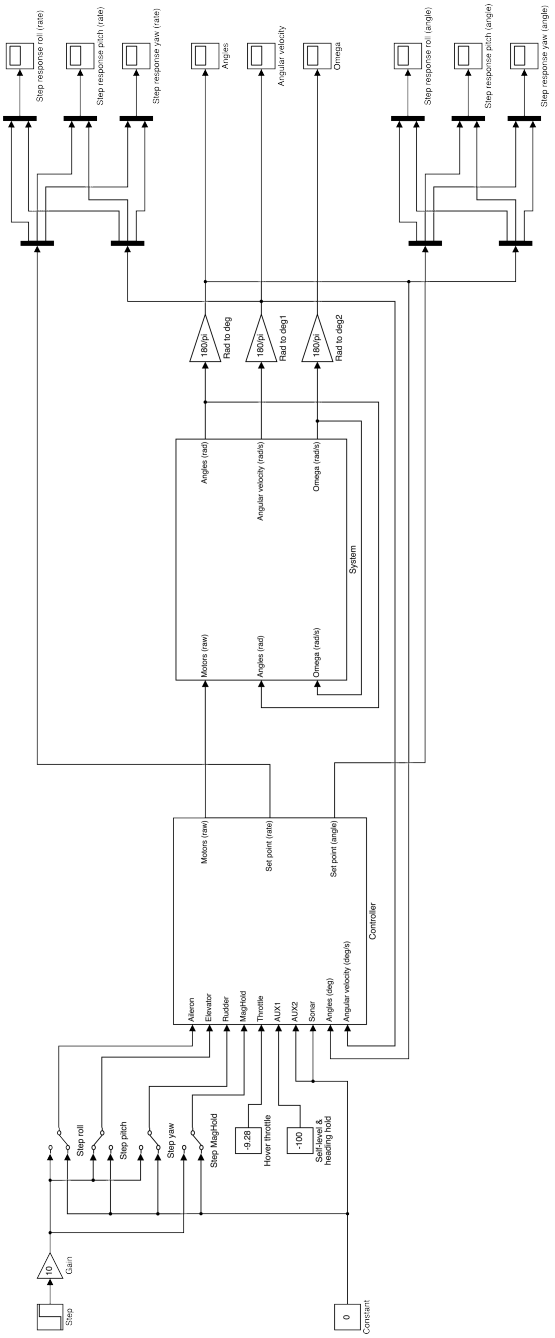


Figure E.1: Overview of the Simulink model

Appendix F

CD-ROM content

The final source code for the flight controller, Android application, the Matlab files used in chapter 9, demonstration videos are located on the CD-ROM attached to this report. An outline of the CD content is depicted below:

- Source code
  - Flight controller firmware
  - Android application
- Matlab files
  - Matlab scripts including log files from quadcopter in flight
  - Simulink model including measurements used to establish the draft coefficient etc.
- 3D model
  - 3D model created in Autodesk Inventor
- Video demonstrations
- Report