Technical University of Denmark

André B.Ø. Bertelsen

Mikkel Wessel Thomsen

Modelling and L1 adaptive control of a quadcopter

Master's thesis, february 2014

DTU Electrical Engineering
Department of Electrical Engineering

English Title: **Modelling and L1 adaptive control of a quadcopter**

Danish Title: **Modellering og L1 adaptiv regulering af en quadcopter**

Report written by:
André B.Ø. Bertelsen
Mikkel Wessel Thomsen

Advisors:
Søren Hansen

DTU Eletrical Engineering
Automation and Control
Technical University of Denmark
Elektrovej
Building 326
2800 Kgs. Lyngby
Denmark
Tel: +45 4525 3576
studieadministration@elektro.dtu.dk

Project periode:	2013/09/30 - 2014/02/28.
ECTS:	30.
Education:	MSc.
Field:	Electrical Engineering.
Class:	After written agreement.
Remarks:	This report is submitted as partial fulfilment of the requirements for graduation in the above education at the Technical University of Denmark.
Number of pages:	133 (last page is p. 125).
Copyrights:	

Preface

This report has been composed by André B. Ø. Bertelsen (s112426) and Mikkel Wessel Thomsen (s072087) at the Department of Electrical Engineering at the Technical University of Denmark, in the period from 30/9-2013 to 28/2-2014. The report is made as a master thesis at the Department of Electrical Engineering, Automation & Control Group.

The target group for this report is university students and teachers, this report is made for educational purposes. The report is divided into 4 parts: "Project Specification", gives a short introduction to the project, including basic quadcopter theory and project specifications. "System Description" contains details about the hardware and software used, and explains how the parameters were found, so the construction of the quadcopter model could be done in Simulink. "L1 Control" gives a theoretical introduction to L1 adaptive control, followed by the design and simulation of an L1 controller, ending with the L1 controller being implemented on the controller board. "Assessment & Conclusion" is the final part, where the system is tested on a physical model, and a conclusion is made, followed by a short description of future work.

The report also contains appendices, that allows for a deeper explanation of some parts, and there will be references to the appendices throughout the report. If the report is read electronically it is possible to jump to chapters, figures, tables, etc. by clicking on their reference, Which is marked by [*number*], where the number corresponds to a source in "References", found in the back of this report. It is also possible to zoom in on many of the figures, since many of them are stored as vector graphics.

For this project, the group members have cooperated to make the documentation, Matlab code, figures, ect. Therefore no main contributor can be assigned for any section of the report.

Along with the report, a CD has been made, containing Matlab scripts, Simulink models, and other relevant files. The CD also contains a multimedia folder where all the pictures and video clips made during the project can be found. The reader is advised to take a look.

André B. Ø. Bertelsen

Mikkel Wessel Thomsen

Contents

CHAPTER 1

Introduction

In the recent number of years, unmanned aerial vehicles (UAV) have played an increasing role in both civilian and military operations. The capabilities of these autonomous UAVs (drones) have given them much attention in the medias, and they have been the topic of many debates, technical as well as ethical. In the new Defence Agreement for 2013-2017, the Danish Ministry of Defence has been granted funds earmarked for experiments with drones for surveillance in Arctic [1]. The focus on drones in the new defence agreement is part of a general trend, where both planes, trains and automobiles are being enhanced with autonomous control. Even the fire department of Copenhagen is currently looking to add multi-rotor drones as a tools that will improve their overview of a situation much faster and safer [2].

In June 2011 U.S. State Nevada passed a law, which allowed autonomous cars to drive on public road, and in 2012 Florida and California followed. Placing the responsibility in case of an accident is still an open question which must be answered, and as a result, legislation limits the use of autonomous vehicles. The legislation related to autonomous flight in Denmark is still a barrier which slows the development of drones larger than 25kg.

The idea of using drones as couriers [3] or as camera dollies, are examples of a uses for drones in public space where the consequence of a failure could spell injury for bystanders. For this reason, reasearch into the robustness of the drones is very important. The emerging field of L1 adaptive control shows promising results as a way of maintaining nominal performance in spite of a fault.

While drones traditionally have been linked with military property this picture is currently changing.

DIY Drones

The current trend of consumer electronics becoming better and cheaper each year, have led to online communities developing open source hardware and software platforms for use of UAVs. An example of such a community is DIY Drones [4], which develops the APM framework which is used for both aircrafts, helicopters and rovers. The open source nature of the project makes modification to existing software and hardware possible, although the documentation is scarce.

DIY Drones has grown a lot over the last few years, starting out using 8-bit processors (atmega2560) and now using 32-bit ARM processors on their new board, the Pixhawk. The evolution of boards can be seen on figure 1.1 on the following page.

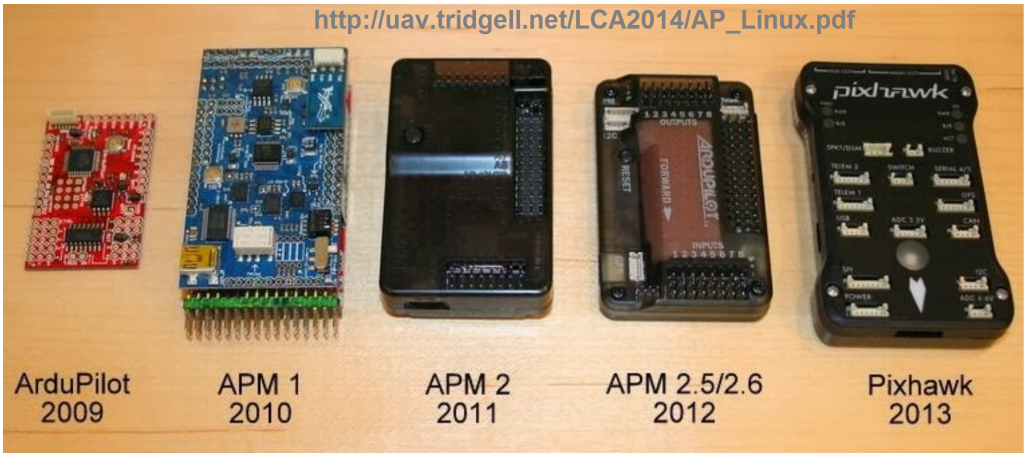


Figure 1.1: *Shows the change the APM project has gone through, from the ArduPilot to the Pixhawk [5]*

DIY Drones are currently working on porting the APM autopilot to Linux, so that it will run on boards such as the BeagleBone and Raspberry Pi. [6]

The DIY Drones community have developed the Arduino-based ArduPilot Mega 2.0 sensor board (APM), which will be the autopilot used in this project, due to a good experience with it, in an earlier project [7]. In the earlier project a radio controlled, fixed-wing aircraft was made autonomous using the APM. This project will instead focus on designing an L1 adaptive controller for a multi rotor, rotary-wing aircraft, specifically a quadcopter.

Part I

Project Specification

CHAPTER 2

Preliminary Analysis

2.1. Quadcopter Theory

In this section the quadcopter will be described from a both practical and mathematical point of view. The term *multicopter* refers to rotary-wing arial vehicles, usually small and unmanned, which are controlled by manipulating the speed, and in some cases the pitch, of the propellers. The term quadcopter describe a subclass of multicopters, which contain four propellers and are widespread because they are relatively cheap and simple to build. The quadcopter used in this project is approximately symmetrical with a body in the center, and four arms emerging perpendicular from it. At the end of each arm, a brushless DC motor (BLDC) is mounted with a propeller on top. The propellers point straight up into the air and thus exerts a force in the opposite direction of gravity when level. Yaw movement is controlled through the torque from the propellers, two of which spin clockwise, two of which spin counter-clockwise, see figure 2.1 on the next page. The front of the quadcopter is the arm with motor no. 3, and it has been colored red for easy identification.

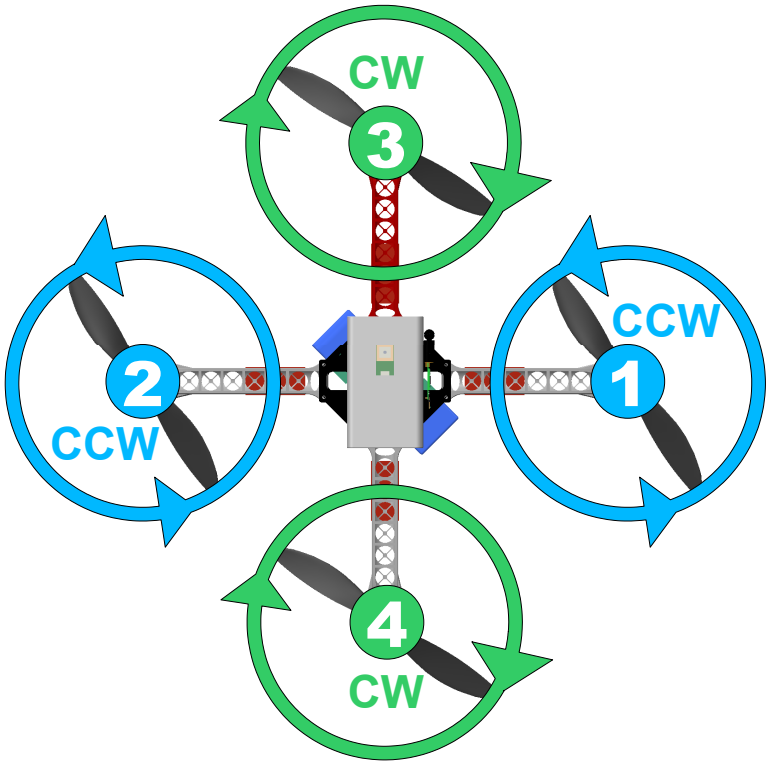


Figure 2.1: *The direction of the four propellers on the quadcopter, CW is for Clockwise and CCW is for Counter Clockwise*

Maneuver (positive direction)	Input
Adjust altitude	Increase the angular velocity of all motors simultaneously
Adjust pitch	Increase the angular velocity of motor 3 relative to that of motor 4
Adjust yaw	Increase the angular velocity of motors 1 and 2 relative to that of motors 3 and 4
Adjust roll	Increase the angular velocity of engine 1 relative to that of motor 2

Table 2.1: *The quadcopter can rotate along the pitch, yaw and roll axis if the angular velocity of the motors are adjusted in the pattern described in this table. To move in the negative direction, reverse the pattern.*

Table 2.1 lists the three axis that the quadcopter can rotate around. The altitude change is in fact movement along the yaw axis, which during non-inverted flight results in an altitude change as described. The quadcopter lacks control-surfaces as on fixed-wing aircrafts, and can only be controlled through manipulating the speed of the motors. The position of the motors give the benefit, that the roll and pitch is controlled by idependent motor sets, and this reduce model complexity. Motors 3 and 4 governs pitch, and are collectively described as the pitch-pair, likewise motors 1 and 2 are called the roll-pair.

2.1.1 Equations of Motion

The quadcopter is described using two three dimensional euclidean spaces, namely the body frame and the earth frame, as seen on figure 2.2 on the next page. The body-fixed

frame has its origin at the location of the APM, with the x,y,z-axis pointing forward, starboard, down, respectively. The earth-fixed frame assumes flat-earth, with the x,y,z-axis pointing north, east, down, respectively. To avoid confusion between the coordinate systems, a right superscript will denote the frame for a variable, when necessary. The body frame will be denoted by "b" and the earth frame will be denoted by "e". Velocity in the body frame will thus be referred to as $v^b = \begin{bmatrix} u & v & w \end{bmatrix}^T$.

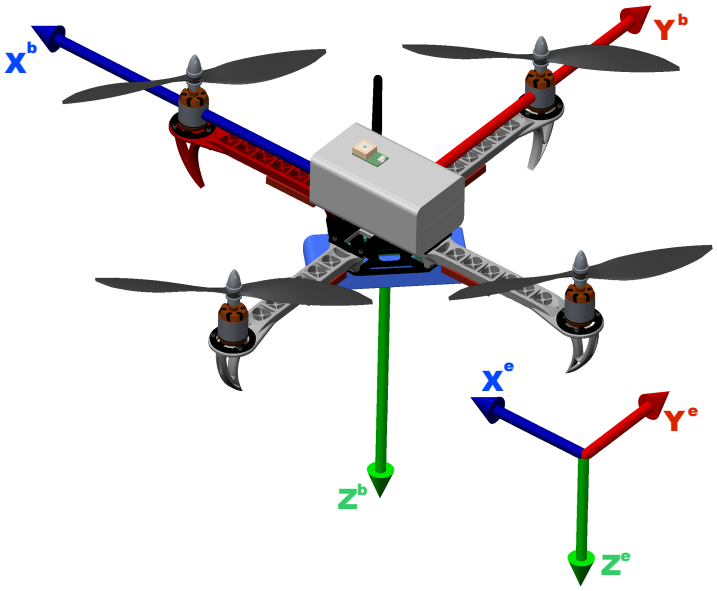


Figure 2.2: *The body frame and earth frame coordinate systems, with arrows representing positive direction. The direction of moments around the axis follow the "rigth-hand-rule".*

To describe the attitude of the body frame relative to the earth frame, Euler angles denoted, $\mathbf{\Phi} = \begin{bmatrix} \psi & \theta & \phi \end{bmatrix}^T$, are used.

Parametrization using Euler angles is subject to singularities and gimbal lock, but these occur outside the flight conditions of this thesis, and for this reason solutions such as quaternion parametrization will not be examined.

The table 2.2 presents the notation used in this section.

Axis	Velocity	Angle	Angular Rate	Moment of Inertia
x	u	ψ	p	I_x
y	v	θ	q	I_y
z	w	ϕ	r	I_z

Table 2.2: *The noation used in this model.*

2.1.2 Direction Cosine Matrix

The transformation from earth frame to body frame is performed as three consecutive right-handed rotations around the z-axis, the new y-axis and the new x-axis [8]. The resulting transformation matrix is denoted $C_{b/e}(\Phi)$, with the subscript b/e meaning body frame from earth frame.

$$C_{b/e}(\Phi) = R_x(\phi)R_y(\theta)R_z(\psi) = \tag{2.1}$$

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & cos(\phi) & sin(\phi) \\ 0 & -sin(\phi) & cos(\phi) \end{bmatrix} \begin{bmatrix} cos(\theta) & 0 & -sin(\theta) \\ 0 & 1 & 0 \\ sin(\theta) & 0 & cos(\theta) \end{bmatrix} \begin{bmatrix} cos(\psi) & sin(\psi) & 0 \\ -sin(\psi) & cos(\psi) & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

where ϕ is the roll angle, θ is the pitch angle, ϕ is the yaw angle and R_n denotes rotation around the n-axis. This yields:

$$C_{b/e}(\Phi) = \tag{2.2}$$
$$\begin{bmatrix} cos(\theta)cos(\psi) & cos(\theta)sin(\psi) & -sin(\theta) \\ sin(\phi)sin(\theta)cos(\psi) - cos(\phi)sin(\psi) & sin(\phi)sin(\theta)sin(\psi) + cos(\phi)cos(\psi) & sin(\phi)cos(\theta) \\ cos(\phi)sin(\theta)cos(\psi) + sin(\phi)sin(\psi) & cos(\phi)sin(\theta)sin(\psi) - sin(\phi)cos(\psi) & cos(\phi)cos(\theta) \end{bmatrix}$$

The transformation matrix from body frame to earth frame, is denoted $C_{e/b}(\Phi)$ and is the inverse of $C_{b/e}(\Phi)$. Since $C_{b/e}(\Phi)$ is orthonormal, the inverse matrix is ${C_{b/e}}^T(\Phi)$.

2.1.3 Euler Kinematical Equation

The angular velocity in body frame relative to earth frame, $\boldsymbol{\omega}^b_{b/e}$, can be calculated using the transformation matrix, $\mathbf{H}_{b/e}(\boldsymbol{\Phi})$.

$$\boldsymbol{\omega}^b_{b/e} \equiv \begin{bmatrix} p \\ q \\ r \end{bmatrix} = \mathbf{H}_{b/e}(\boldsymbol{\Phi})\,\dot{\boldsymbol{\Phi}} \tag{2.3}$$

where $\mathbf{H}_{b/e}(\boldsymbol{\Phi})$ is given as [8]:

$$\mathbf{H}_{b/e}(\boldsymbol{\Phi}) = \begin{bmatrix} 1 & 0 & -sin(\theta) \\ 0 & cos(\phi) & sin(\phi)cos(\theta) \\ 0 & -sin(\phi) & cos(\phi)cos(\theta) \end{bmatrix} \tag{2.4}$$

2.1.4 Translational Motion of the Quadcopter

The two main forces acing on the quadcopter is the gravitational force, $\mathbf{F}^e_g$, and the thrust $\mathbf{F}^b_T$. The gravitational force vector in the body frame can be calculated using the direction cosine matrix:

$$\mathbf{F}^b_g = \mathbf{C}_{b/e}\mathbf{F}^e_g = \mathbf{C}_{b/e} m \begin{bmatrix} 0 \\ 0 \\ g \end{bmatrix} \tag{2.5}$$

where m is the mass of the quadcopter, and g is the gravity of earth.

The thrust is the sum of the force from the four motors. Since thrust is proportional to the angular velocity of the propellers, the force can be modeled as [9]:

$$\mathbf{F}^b_t = b\left({\Omega_1}^2 + {\Omega_2}^2 + {\Omega_3}^2 + {\Omega_4}^2\right) \tag{2.6}$$

where b is the thrust coefficient calculated in 4.2.4 on page 28.

The kinematics can then be described from Newton's second law:

$$\mathbf{F}^e = m\mathbf{a}^e = m\dot{\mathbf{v}}^e \tag{2.7}$$

where $\mathbf{F}^e$ is the sum of forces in earth frame, $\mathbf{a}^e$ is the acceleration in earth frame and $\mathbf{v}^e$ is the velocity in earth frame. The velocity can be derived from the body frame as:

$$\mathbf{v}^e = \mathbf{C}_{e/b}\mathbf{v}^b \tag{2.8}$$

with its derivative being:

$$\dot{\mathbf{v}}^e = \mathbf{C}_{e/b}\dot{\mathbf{v}}^b + \dot{\mathbf{C}}_{e/b}\mathbf{v}^b \tag{2.9}$$

The derivative of the direct cosine matrix times the velocity in body frame is given in: "Autonomous Control of a Miniature Quadrotor Following Fast Trajectories" [10] as:

$$\dot{\mathbf{C}}_{e/b}\mathbf{v}^b = \mathbf{C}_{e/b}\left(\boldsymbol{\omega}^b \times \mathbf{v}^b\right) \tag{2.10}$$

so that 2.9 becomes:

$$\dot{\mathbf{v}}^e = \mathbf{C}_{e/b}\dot{\mathbf{v}}^b + \mathbf{C}_{e/b}\left(\boldsymbol{\omega}^b \times \mathbf{v}^b\right) \tag{2.11}$$

Multiplying by the mass yields 2.7 on the preceding page:

$$\mathbf{F}^e = m\left(\mathbf{C}_{e/b}\dot{\mathbf{v}}^b + \mathbf{C}_{e/b}\left(\boldsymbol{\omega}^b \times \mathbf{v}^b\right)\right) \tag{2.12}$$

Multiplying by the direction cosine matrix on both sides, transforms the force to body frame:

$$\mathbf{C}_{b/e}\mathbf{F}^e = m\mathbf{C}_{b/e}\left(\mathbf{C}_{e/b}\dot{\mathbf{v}}^b + \mathbf{C}_{e/b}\left(\boldsymbol{\omega}^b \times \mathbf{v}^b\right)\right) \tag{2.13}$$

$$\sum \mathbf{F}^b = m\left(\dot{\mathbf{v}}^b + \left(\boldsymbol{\omega}^b \times \mathbf{v}^b\right)\right) \tag{2.14}$$

Gravity and thurst equals the sum of forces:

$$\mathbf{F}_g{}^b + \mathbf{F}_t{}^b = m\left(\dot{\mathbf{v}}^b + \left(\boldsymbol{\omega}^b \times \mathbf{v}^b\right)\right) \tag{2.15}$$

inserting 2.5 on the facing page and 2.6 on the preceding page calculating the right hand side yields:

$$\mathbf{C}_{b/e}m\begin{bmatrix}0\\0\\g\end{bmatrix} + \begin{bmatrix}0\\0\\b\left(\Omega_1^2+\Omega_2^2+\Omega_3^2+\Omega_4^2\right)\end{bmatrix} = m\begin{bmatrix}\dot{u}+qw-rv\\\dot{v}+ru-pw\\\dot{w}+pv-qu\end{bmatrix} \tag{2.16}$$

Isolating the acceleration yields the three following state equations:

$$\dot{u} = rv - qw - gsin(\theta) \tag{2.17}$$

$$\dot{v} = pw - ru + gcos(\theta)sin(\phi) \tag{2.18}$$

$$\dot{w} = qr - pv + gcos(\phi)cos(\theta) - \frac{b}{m}\left(\Omega_1^2+\Omega_2^2+\Omega_3^2+\Omega_4^2\right) \tag{2.19}$$

2.1.5 Angular Motion

The rotational dynamics of the quadcopter can be isolated from the translational dynamics, by using the center of mass as a reference point. For simplicity's sake, the center of mass will be assumed to be the center of the body frame, that is, the APM.

Defining the torque vector of the body axis as $\mathbf{M}^b = \begin{bmatrix}M_x & M_y & M_z\end{bmatrix}^T$ allows Euler's equations of motion to be written as follows, according to [8]:

$$\dot{\mathbf{p}} = \frac{(I_y - I_z)\, qr}{I_x} + \frac{M_x}{I_x} \tag{2.20}$$

$$\dot{\mathbf{q}} = \frac{(I_z - I_x)\, pr}{I_y} + \frac{M_y}{I_y} \tag{2.21}$$

$$\dot{\mathbf{r}} = \frac{(I_x - I_y)\, pq}{I_z} + \frac{M_z}{I_z} \tag{2.22}$$

where I_n is the inertia around the n-axis. The symmetrical quadcopter design minimizes the inertia coupling in yaw, when it is assumed that $I_x \approx I_y$.

$$\dot{\mathbf{r}} = \frac{M_z}{I_z} \tag{2.23}$$

Torque is defined as:

$$\tau = \mathbf{r} \times \mathbf{F}, \tag{2.24}$$

where τ is the torque vector, $\mathbf{r}$ is the displacement vector and $\mathbf{F}$ is the force vector. The displacement vector represents the distance from the point where torque is measured to the point where force is applied, i.e. from the quadcopters center of mass to the DC motors. The torque from the thrust is expresed in 2.25.

$$\mathbf{M}_{thurst} = \sum_{i=1}^{4} \mathbf{r} \times \mathbf{F}_i = \sum_{i=1}^{4} \mathbf{r} \times \mathbf{b}\Omega_i^2 \tag{2.25}$$

where $\mathbf{b} = \begin{bmatrix} 0 \\ 0 \\ -b \end{bmatrix}$, and b is the thrust coefficient from 2.6 on page 8. Defining the length from the center of mass to the motors as l, and inserting the position of the motors yields:

$$\mathbf{M}_{thurst} = \begin{bmatrix} -lb \\ 0 \\ 0 \end{bmatrix} \Omega_1^2 + \begin{bmatrix} lb \\ 0 \\ 0 \end{bmatrix} \Omega_2^2 + \begin{bmatrix} 0 \\ lb \\ 0 \end{bmatrix} \Omega_3^2 + \begin{bmatrix} 0 \\ -lb \\ 0 \end{bmatrix} \Omega_4^2 \tag{2.26}$$

The thrust only affects the torque on the x and y axis, thus:

$$M_x = lb\left(\Omega_1^2 - \Omega_2^2\right) \tag{2.27}$$

$$M_y = lb\left(\Omega_3^2 - \Omega_4^2\right) \tag{2.28}$$

2.1.6 Gyroscopic Moments

The rotation of the propellers and the motors give rise to gyroscopic forces which affect the moment equations. The contribution is given in [11] as:

$$M_{gyro,x} = \dot{H}_x + H_z q - H_y r \tag{2.29}$$

$$M_{gyro,y} = \dot{H}_y + H_x r - H_z p \tag{2.30}$$

$$M_{gyro,z} = \dot{H}_z + H_y p - H_x q \tag{2.31}$$

where H_x, H_y and H_z is the total angular momentum of the four motors, given as [9]:

$$H_z = \sum_{i=1}^{4} I_z \omega_{i,z} = I_p \left(\Omega_1 + \Omega_2 + \Omega_3 + \Omega_4\right) \tag{2.32}$$

where I_p is the combined inertia of both the motor and propeller in the z-axis, and the derivative, $\dot{H}_z$, being the torque around the z-axis. The torque is proportional to the sum of the angular velocities of the engies squared, by the drag coefficient, d. Only yaw is affected by this drag.

$$M_{gyro,z} = d\left(\Omega_1^2 + \Omega_2^2 - \Omega_3^2 - \Omega_4^2\right) + H_y p - H_x q \tag{2.33}$$

Since the motors rotational axis is parallel to the z body axis, only the gyroscopic from rotation around the z axis contributes, and $H_x = H_y = 0$.

Combining the equation from the previous section, yields the final state equations:

$$\dot{\mathbf{p}} = \frac{\left(I_y - I_z\right) qr + lb\left(\Omega_1^2 - \Omega_2^2\right)}{I_x} + I_p \left(\Omega_1 + \Omega_2 + \Omega_3 + \Omega_4\right) q \tag{2.34}$$

$$\dot{\mathbf{q}} = \frac{\left(I_z - I_x\right) pr + lb\left(\Omega_3^2 - \Omega_4^2\right)}{I_y} + I_p \left(\Omega_1 + \Omega_2 + \Omega_3 + \Omega_4\right) p \tag{2.35}$$

$$\dot{\mathbf{r}} = \frac{d\left(\Omega_1^2 + \Omega_2^2 - \Omega_3^2 - \Omega_4^2\right)}{I_z} \tag{2.36}$$

CHAPTER 3

Problem Formulation

This report will deal with a practical implementation of a attitude/altitude-autopilot on a quadcopter. The hardware consists of an APM with its on-board sensors, the airframe, motors, speed controllers (ESC), propellers etc. The autopilot will be implemented on a ArduPilot Mega 2.0 (APM), A broader analysis of the sensors been done in an earlier project [7], a section of which can be seen in appendix B on page 116. an open source sensor board, containing a number of sensors and two microprocessor. The selection, implementation and mounting of the hardware in the airframe has all been made by the group as part of the project. A lot of thoughts and effort has been put into this implementation, the final result and the main thoughts and ideas will be explained in this the report. The quadcopter will be tuned by hand, when tuning the quadcopter the main interest is in the pitch and roll angles, the yaw, and climb rate in the flight mode called stabilize. Tuning the yaw and climb rate to track the references exactly is will not be necessary, since these will not will be experienced to the same degree as a failure to track the reference in pitch or roll angle.

To be able to gain a better knowledge of the physics involved in the system the theory of a quadcopter have be looked at in section 2.1 on page 5. So to be able to derive a model of the system and implement it in Simulink, where it will be verified to behave like the real system. Then the model will used for simulations for nominal performance and fault cases

To improve the systems performance in the present of faults "L1 adaptive control" will be examined. This will be implemented on the Simulink system to verify its behaviours. Then later implemented on the APM board, and tested on the real system.

Part II

System Description

CHAPTER 4

System Description

In this section a description of the quadcopter setup and the eletronics will be given, since it has been assembled by the group as part of the project. The hardware, including the frame and the motors will be covered. A few electronic components that are very helpful, but not strictly necessary for flight, will also be covred. The APM including its on-board sensors will be described in section 5 on page 33, its mount will be covered in this section, however. The thrust coefficient, drag coefficient, inertia and motor dynamics will be calculated in this chapter.

4.1. The Quadcopter

The quadcopter build for this project is the result of considerations related to stability, flight time, reliability and price. The frame, motors, propellers and motor controllers (ESC) has been chosen as a result of our own knowledge and research of similar setups[1]. The quadcopter was measured and drawn in a 3D CAD program[2], and the resulting model is featured in many illustrations in this report, for more picures of the quadcopter see appendix D on page 122. By weighing various parts and adding this information to the 3D model, the inertia could be calculated by the progam. The frame layout was chosen so that pitch and roll would be decoupled, and the "Hobbyking SK450 Glass Fiber Quadcopter Fram" fitted this specification. This frame was also chosen due to it size and simplicity. The assembled frame can be seen on figure 4.1 on the following page.

[1] The choice of motor and propeller was aided using an online tool that calculates various motor parameters, called eCalc at: http://www.ecalc.ch/xcoptercalc.php?ecalc&lang=en

[2] PTC CREO 2.0

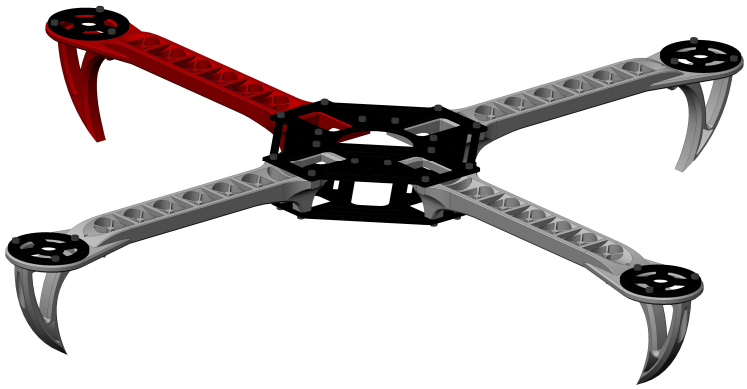


Figure 4.1: *The bare frame with no eletronics mounted*

The motors are four "Turnigy D2836/11 750KV Brushless Outrunner", they get their power from a "Turnigy 5000mAh 3S (11.1V) 20C Lipo" battery. To control the motors, an ESC (Eletrical Speed Controller) is needed, here four "HobbyKing Red Brick 30A ESC" was chosen. The propellers are of the size[3] 12"x4.5". The ESC can aside from controlling the motors also deliver 5V (2A), and this 5V line is used to powers the APM, sensors and telemetry. All four of the ESCs have this feature, but only the 5V lines from one ESC is used.

Other eletronics mounted:

- GPS - MediaTek-3329
- Telemetry - 3D Radio 433MHz (from 3D Robotics)
- Voltage sensor

For more information on these other elements, please refer to our previous report "Integration and Testing of Avionics - 2nd Edition" [7].

A cross section of the assembled quadcopter can be seen on figure 4.2 on the next page. The APM has been put in a box primarily to protect it, in the event of a crash. Another reason is that the altimeter is a barometric sensor, and it is sensitive to fluctuations in wind pressure. The rotating propellers are expected to cause such fluctuating, so by putting the APM in the box, the altimeter is subjected to fewer disturbances. Notice that the APM is fitted to a vibration damping mount. This mount was featured on a blog post of on the DIY-drones homepage[12], the STL-files were downloaded and it has been 3D printed in DTU's fablab.

[3]Diameter x pitch, in inches. The pitch is the distance the propeller would cover during one rotation, assuming it would carve its way forward.

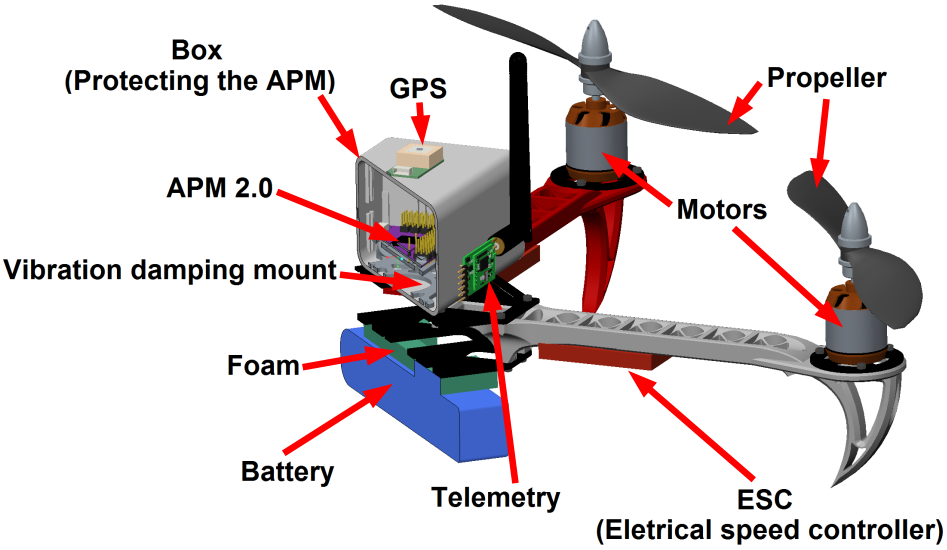


Figure 4.2: *Cross section of the quadcopter - the wires are not shown in this figure.*

The vibrations have been measured in hover with and without the mount. Prior to adding the mount, the APM was resting on a pad of foam. As seen on figure 4.3 and figure 4.4 on the next page, the vibration mount greatly dampens the vibrations, and improves the measurements. The recommended vibration noise levels are found on the arducopters homepage[13], and are marked by the black, horizontal lines on the figures.

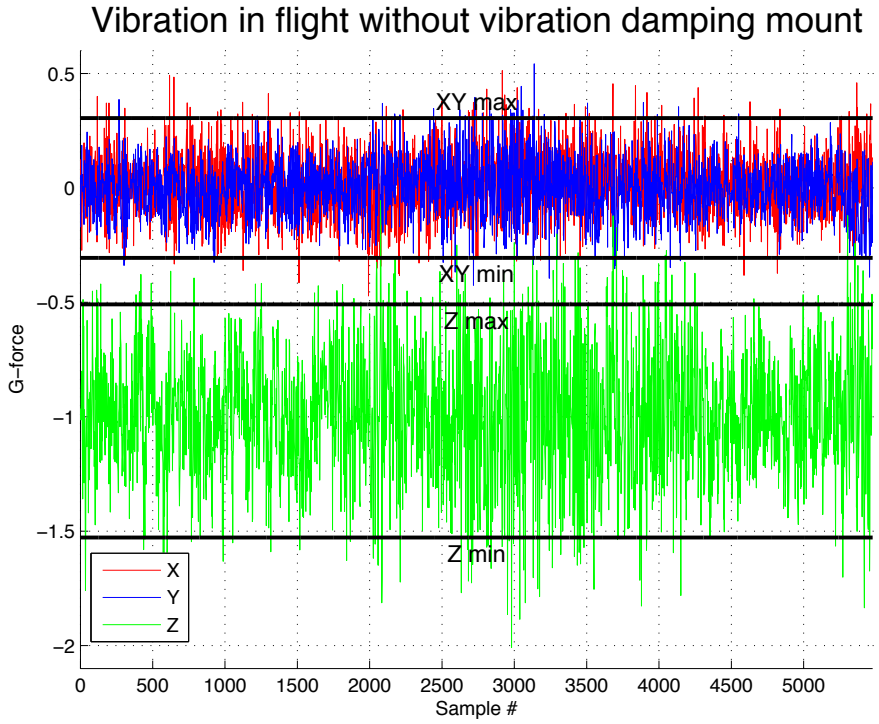


Figure 4.3: *Vibrations during hover with out vibration damping mount, note that the vibratios are outside the recommended levels (The horizontal lines)*

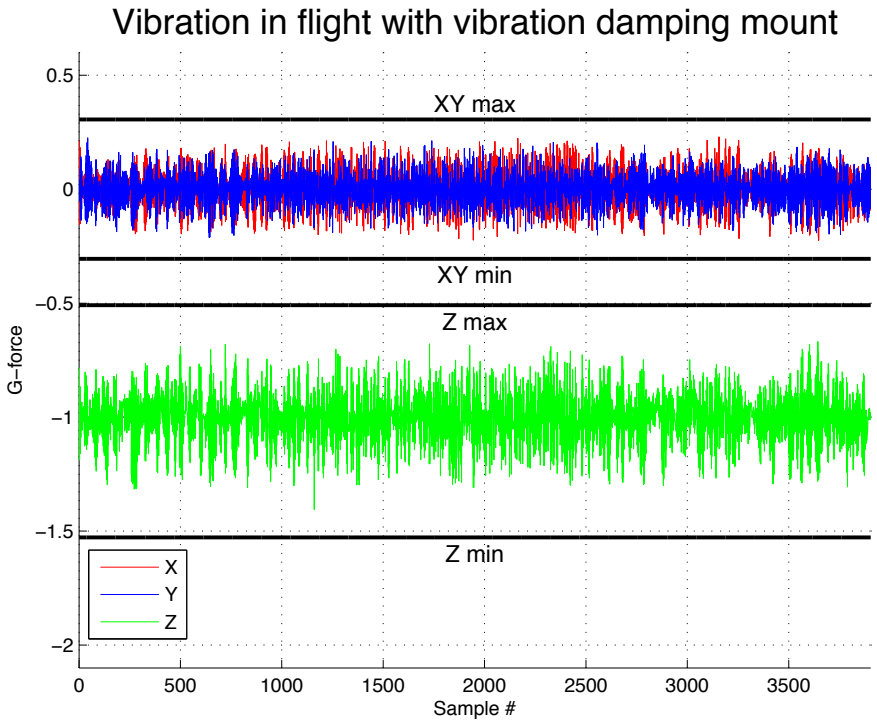


Figure 4.4: *Vibrations during hover with out vibration damping mount, note that the vibratios are outside the recommended levels (The horizontal lines)*

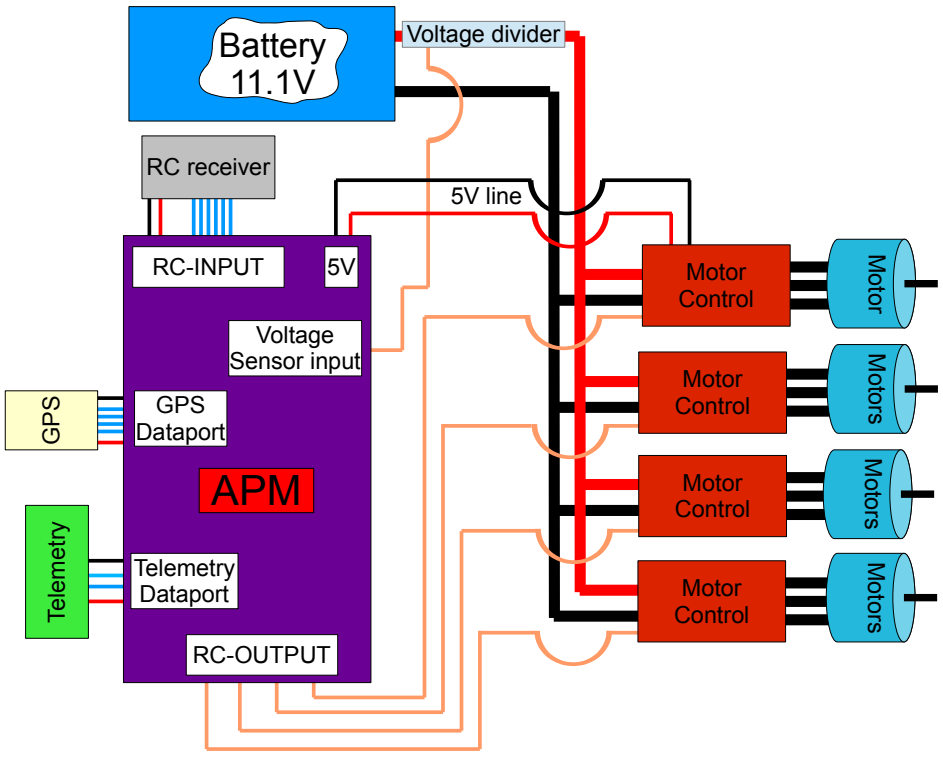


Figure 4.5: *An overview of how the electronics are connected.*

The figure 4.5 on the facing page give an overview of how the different electronics are connected, this is a very simple schematic, but it gives the user an idea how the parts mentioned in this section is connected. The sensors on the APM and software is described in section 5 on page 33.

RC Transmitter and Receiver

On figure 4.5 on the facing page an RC receiver is shown. This is what receives the pilot inputs sent via the RC transmitter. The transmitter/receiver setup has been changed halfway the project, the reason being that the old transmitter, a Spektrum Dx6i, had been sold. The new transmitter is a "Taranis X9D" from FrSKY, some key features is listed below, and a picture can be seen on figure 4.6 on the next page

- Transmitter frequency 2.4GHz
- 16 channels (more when combined with external module)
- Audio Speech outputs
- 8 programmable switches
- 4 programmable knobs/sliders
- Super low latency for ultra-quick response (9ms)
- Processor: STM32 ARM Cortex M3 60MHz
- RSSI alarms (warns you of reception problems before it becomes an accident)
- Runs OPEN TX, an open source RC transmitter OS

Figure 4.6: *The transmitter used for this project*

The receiver is an X8R from FrSKY, it runs the same 2.4GHz protocol as the transmitter, it has 8 channels so it will fit this project well, with some channels to spart even.

For controlling the quadcopter, 4 channels are needed, pitch, roll, yaw and throttle. The APM needs a 5th channel to able to switch between different flight modes. This means that there still is 3 channesl that are unused, but the will be uitlized later for things like activating L1 controller and tuning the L1 gain.

4.2. Identification of the Quadcopter

This section will focus on the identification of parameters of the quadcopter, such as the inertia, thrust and drag coefficient etc. These are used in the simulink model described in 6.1 on page 45

4.2.1 Propeller Inertia

The propeller inertia is needed to be able to calculate the gyroscopic moment explained in section 2.1.6 on page 10. To determine the moment of inertia of the propeller, one propeller blade had been cut into six pieces, see figure 4.7 on the facing page.

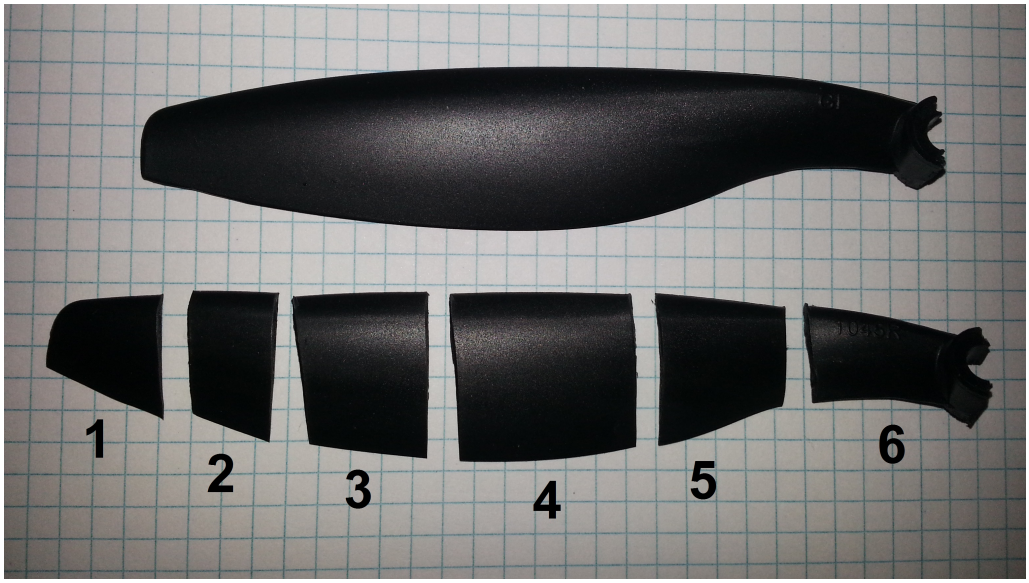


Figure 4.7: *The propeller blade it cut in to six pieces*

Each segment was then weighed and had its distance from its mass midpoint to the rotational axis measured, see the table 4.1. The mass midpoint is estimated to be in the centre of the segment.

	1	2	3	4	5	6	unit
Weight	0.24	0.34	0.7	1.13	0.78	1.08	10^{-3}[kg]
Distance form rotational axis	11.860	10.335	8.670	6.155	3.515	1.200	10^{-2}[m]

Table 4.1: *Propeller data used in calculations below*

The moment of inertia is then calculated using equation 4.1.

$$J_{\text{Segment}} = (L^2 \cdot m_{\text{Segment}}) \tag{4.1}$$

where:

L	is the distance from the rotational axis to the mass midpoint	[m]
m_{Segment}	is the mass of each segment	[Kg]
J_{Segment}	is the inertia of each segment	[Kg $\cdot$ m^2]

	1	2	3	4	5	6	unit
Segment inertia	3.3758	3.6316	5.2618	4.2809	0.9637	0.1555	10^{-6} [Kg $\cdot$ m^2]

Table 4.2: *Moment of inertia of each element*

The inertia of each element can be seen in table 4.2, when the inertia of each segment is know the total propeller inertia J_{Prop} calculated as twice the sum of all segments (there are two blades on one propeller).

$$J_{\text{Prop}} = 2 \cdot \sum J_{\text{Segment}} = 35.339 \cdot 10^{-6} \tag{4.2}$$

where:

J_{Prop}	is the distance from the rotational axis to the mass midpoint	[Kg $\cdot$ m^2]
J_{Segment}	is the inertia of each segment	[Kg $\cdot$ m^2]

Motor Inertia

Another thing that adds to the "Gyroscopic Moments" is the inertia of the motor's rotor. On the motor used the rotor sits on the outside of the motor as shown on figure 4.8

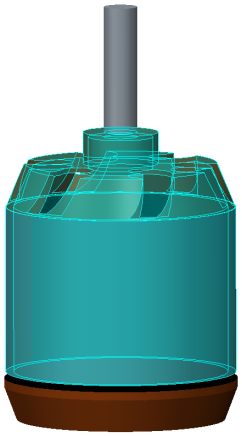


Figure 4.8: *The motors external rotor*

The inertia of this part is calculated as a cylindrical tube with hight 0, since we only need the inertia around the z-axis. Equation 4.3 explains the relations.

$$J_{\mathrm{m}} = 0.5 \cdot m \cdot ({r_2}^2 + {r_1}^2) \tag{4.3}$$

where:

J_{m}	is the inertia of the motors rotor	[Kg · m^2]
m	is the mass of the rotor	[Kg]
r_1	is the inner radius	[m]
r_2	is the outer radius	[m]

Inserting the values into equation 4.3, gives:

$$J_{\mathrm{m}} = 0.5 \cdot 0.0321 \cdot (0.0140^2 + 0.0128^2) = 5.7513 \cdot 10^{-6} \tag{4.4}$$

The total moment of inertia that is used for the simulink model is calculated in equation 4.5.

$$J_{\mathrm{tot}} = 5.7513425 \cdot 10^{-6} + 35.339 \cdot 10^{-6} = 41.090 \cdot 10^{-6} \tag{4.5}$$

4.2.2 PWM to Motor Angular Velocity

The APM sends a PWM signal to the speed controllers (ESC), they convert the PWM signal in to a 3-phase signal to drive the motor. This relationship is for simplicity estimated as linear. To find the relationship, the throttle percentage was measured from flight data at hover to be 40%. To find the angular velocity at this percentage, 3 motors was disabled on the quadcopter and the sound of one propeller running at 30%, 40% and 50% throttle was recorded.

An illustration of the setup can be seen on figure 4.9.

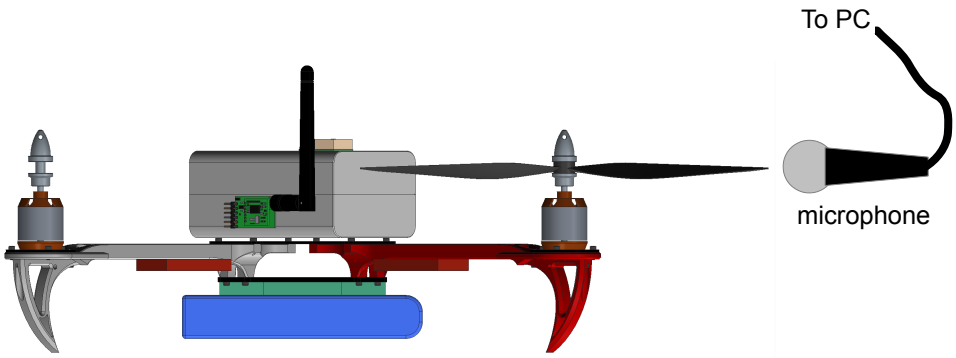


Figure 4.9: *Micropone placement during the sound measuring.*

The microphone was held parallel to the propeller disk to avoid wind noise. The sound was captured using "Audacity" [4], the file was then processed in a Matlab script to find the frequency, half this frequency is the angular velocity of the motor, it is half because the propeller have two blades. The results can be seen in table 4.3

Throttle [%]	PWM [-]	RPS [1/s]	Angular Velocity [rad/s]
30	1380	66.669	418.900
40	1467	83.774	526.342
50	1556	104.259	653.451

Table 4.3: *Moment of inertia of each element*

The pwm signal corresponding to a percentage have been found from the log file. A linear function that fitted the 3 points best was then made, the function is displayed on figure 4.10 on the following page, the function is: $f(pwm) = 0.2122 \cdot pwm - 226.568$. The angular velocity can be multiplied by 2π to transform it from Hz to rad/s.

[4] Audacity® is free, open source, cross-platform software for recording and editing sounds.

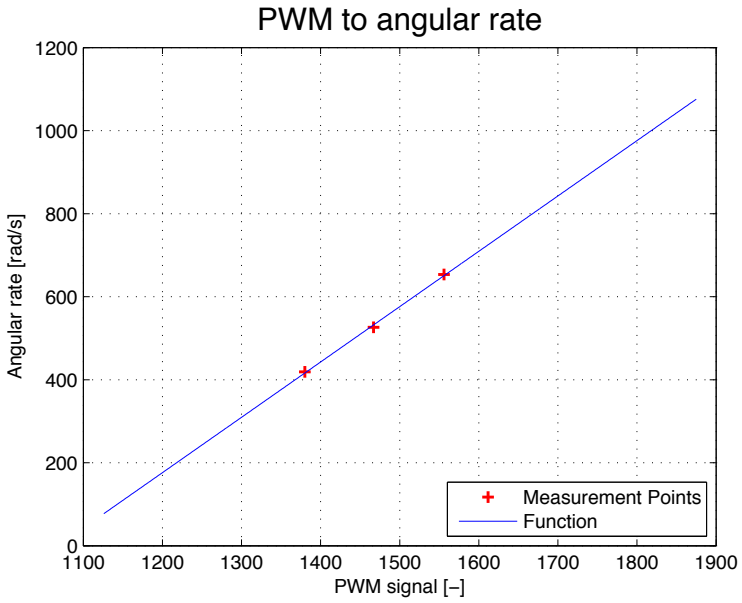


Figure 4.10: *The points measured and the function to calculate the angular velocity.*

The angular velocity of the motores at hover is found to be $\Omega_0 = 2\pi \cdot 83.774 = 526.342$ rad/s.

4.2.3 Drag coefficient

The drag coefficient affects how the quadcopter turns around the z-axis, which is seen in equation 2.36 on page 11. Rearrange this equation to get the drag coefficient gives:

$$d = \frac{\dot{\mathbf{r}} \cdot I_z}{\Omega_1^2 + \Omega_2^2 - \Omega_3^2 - \Omega_4^2} \tag{4.6}$$

where:

d	is the drag coefficient	[Kg · m^2/rad]
$\dot{\mathbf{r}}$	is the change in yaw rate	[rad/s^2]
I_z	is the inertia around the Z-axis	[Kg · m^2]
Ω_i	is angular velocity of the each motors	[rad/s]

The yaw rate r can be obtained from flight data, this data is measured by the IMU, the angular velocity of the motors is obtained via the logged PWM signal to the motors, this is then passed through the "PWM to angular velocity function" described 4.2.2 on page 25 to generate the angular velocity.

The data used comes from a flight where the quadcopter is spinning around its z-axis, and the mean values during a spin can be seen in table 4.4. It is seen that motor 1 and motor 2 has a higher angular velocity than motor 3 and motor 4. This shows that the quadcopter is rotating counter clock wise. The reason that there is a difference in the angular velocity on the motor pairs (1-2 and 3-4) is because they also work on holding the quadcopter horizontal.

data	mean value [rad/s]
r	1.5972
Ω_1	493.5594
Ω_2	531.4842
Ω_3	337.7124
Ω_4	328.2312

Table 4.4: *Flight data for calculating the drag coefficient.*

The mean value of $\dot{\mathbf{r}}$ is found to be $\dot{\mathbf{r}} = 2.0061$. The last part needed is the moment of inertia around the Z-axis, this has been found in section 4.2.5 on page 29, to be: $I_\mathrm{Z} \approx 0.0432$ [Kg · m^2]. Substituting these values into equation 4.6, yields:

$$d = \frac{0.0432 \cdot 2.0061}{493.5594^2 + 531.4842^2 - 337.7124^2 - 328.2312^2} \tag{4.7}$$

From equation 4.7 the drag coefficient is found to be: $d = 285.0826 \cdot 10^{-9}$ kg · m^2/rad.

4.2.4 Thrust coefficient

In the section 4.2.2 on page 25 the pwm-to-rps relationship was determined, and so was the angular velocity at hover, which is one of the elements needed to find the thrust coefficient. The thrust coefficient b can be found in equation 4.8 [9].

$$T = b \cdot \Omega^2 \tag{4.8}$$

where:

T	is the thrust	[N]
Ω	angular velocity	[rad/s]
b	is the thrust coefficient	[kg · m]

By rearrange equation 4.8 and substituting T for the force from gravity, it is found that the thrust coefficient b is calculated as:

$$b = \frac{mg}{4\Omega_0^2} \tag{4.9}$$

$$b = \frac{1.35 \cdot 9.82}{4 \cdot 83.774^2} = 11.962 \cdot 10^{-6} \tag{4.10}$$

where:

m	is the mass of the quadcopter	[kg]
g	is the gravitational acceleration	[m/s^2]
Ω_0	angular velocity at hover	[rad/s]
b	is the thrust coefficient	[kg·m]

4.2.5 Quadcopter inertia

The inertia of the quadcopter is used in the equations describing rotation around the body axis. As previously mentioned, the entire quadcopter has been created in a CAD program, as precise as possible. Therefore it is possible to let the CAD program calculate the inertia around all the axis. The inertia $J_{\text{quadcopter}}$ is found using the CAD program and is show in equation 4.11.

$$J_{\text{quadcopter}} = \begin{bmatrix} 26.108437 & -4.7476919 & -0.13589257 \\ -4.7476919 & 23.505235 & 0.57411620 \\ -0.13589257 & -0.57411620 & 43.243138 \end{bmatrix} \cdot 10^{-3}[\mathtt{kg} \cdot \mathtt{m}^2] \qquad (4.11)$$

To make sure that the inertia is around the values that can be expected a simple calculation of the inertia have been made with the values in table 4.5. These assume that each object is a point mass at the distance from the centre. The remaining mass is put in to a box in the centre of the quadcopter, and its inertia is calculated.

Object	Distance from centre [m]	Weight [kg]	Inertia [X,Y,Z] [kg·m^2]
Arm	0.1118	0.0410	[0.0010, 0.0010, 0.0020]
Motor + foot + propdrive	0.2235	0.0910	[0.0091, 0.0091, 0.0182]
ESC	0.1000	0.0230	[0.46, 0.46, 0.92] $\cdot$ 10^{-3}
	size (h,d,w) [m]		
Centerbox	0.1, 0.125, 0.08	0.7300	[9.9767e-4, 0.0016, 0.0013]
Total inertia			[11.57, 12.13, 22.49] $\cdot$10^{-3}

Table 4.5: *Inertia of the different parts*

It is seen that the inertia found using the two methods are different, as expected. The difference is about a factor 2. These "back-of-the-envelope" calculations show that the orders of magnitude from the CAD program is around what can be expected. Therefore the the inertia from the CAD program is used. It is also noted that the CAD program calculates the cross connections in the inertia matrix, this describes how a rotation around the x axis effects the rotation in the y and z axis, so in the simulations this effect will be present.

4.2.6 Motor Dynamics

It is obvious that the motor can not change its angular velocity instantaneous, so to find the time constant, a special jig has been made, a photo of this can be seen on figure 4.11

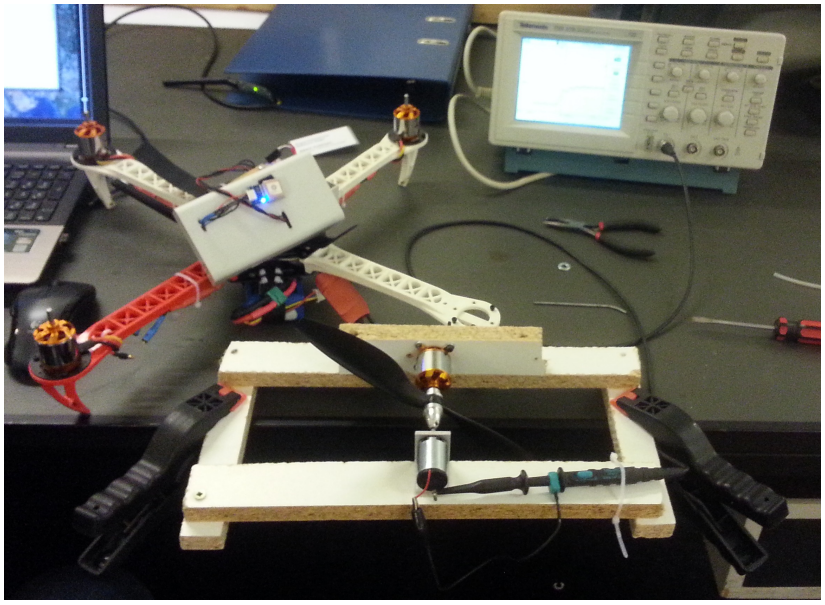


Figure 4.11: *The jig setup to measure the time constant of the motor*

In this jig one of the brushless motors have been connected to another small DC motor, a maxon 110124, with a mechanical time constant of 20.1ms [14]. It is expected that the time constant of the motor from the quadcopter has a considerable larger time constant than the small motor.

The axis of the two motors are connected using a piece of silicon tubing, this allows the motors axis to stay connected even if they are not exactly centred. To be able to attach the silicon tube to the prop-driver on the quadcopter a 2mm hole has been made and a length of 2mm piano wire has been pressed in and glued in place. To make the hole the prop driver was put in the a drill bit and turned, rather than turning the drill it self. This will help the prop driver to centre on the drill. The connection of the two motors can be seen on figure 4.12

Figure 4.12: *The connection between the two axis.*

Then stepping on the angular velocity on the brushless motors, the maxon motor

will generate a voltage across its terminals. This voltage is measured during a using an oscilloscope, and one of the results can be seen on figure 4.13. A relative small step has been made, since the quadcopter is modeled in hovew, were it will make small steps to remain horizontal.

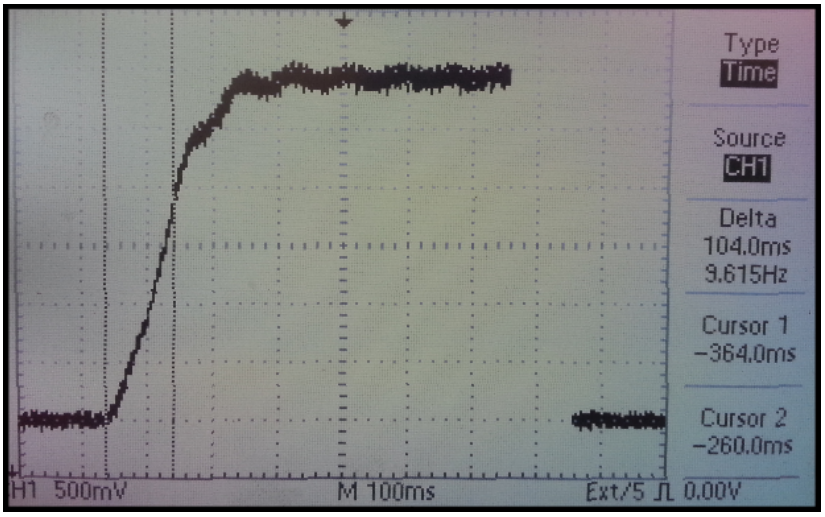


Figure 4.13: *One of the steps made on the motor.*

It is clear that the system is not a first order, but for simplicity's sake, the system is modelled as a first order. The results of the steps can be seen in table 4.6.

measurement	delta V	delta V at 63.2%	delta t [ms]
1	2.94	1.86	104
2	2.94	1.86	104
3	2.90	1.83	108
4	2.90	1.83	108
5	2.90	1.83	108
average			106.4

Table 4.6: *Results of steps on motor with propeller*

The average is found to be $\tau_{\text{motor}} = 106.4$ [ms]. The DC gain of the system is irrelevant for exercise, because the correlation between the PWM and the angular velocity has been found in 4.2.2 on page 25, only the time constant is needed.

CHAPTER 5

ArduPilot Mega

This chapter deals with the hardware and software of the APM. The hardware has already been described in a previous report [7], but the most relevant results will be presented here. The software of the APM is very versatile and customizable, but only the core components related to flight will be described.

5.0.7 Gyroscope and Accelerometer

A combined gyroscope and accelerometer is implemented in the MPU-6000 sensor chip [15]. The MPU-6000 is set to measure the attitude and attitude rates at 100Hz.

The attitude can be obtained by calling *ahrs.roll_sensor*, *ahrs.pitch_sensor* or *ahrs.yaw_sensor*. These are returned in centi-degrees, and assume values from -18,000 to 18,000, except for yaw which assumes values between 0 and 36,000. When the APM board is inverted, the value of roll will shift by 360 degrees, inverted flight is not within the scope of this project, though.

The attitude rates for roll, pitch and yaw are obtained by calling *omega.x*, *omega.y* or *omega.z*, respectively. Unlinke the attitude, this is given in radians per second, but can be converted to centi-degrees by multiplying with the constant *DEGX100*.

5.0.8 Telemetry and Ground Station

The telemetry module is mounted on the starboard side of the box, the antenna being visible in figure 4.2 on page 19. The *3DR Radio* is compatible with the APM code, supports USB connection for the PC. Flight tests during the previous project showed that the signal could be maintained for up to 500 meters, which is more than enough for this project. The telemetry module supports two way communication based on the MAVLink protocol.

The ground station consists of a PC with the telemetry link, running the Mission Planner software [1]. The mission planner is used to configure parameters, extract logfiles and view live telemetry. The mission planer is also needed to *arm* the quadcopter prior to take-off. The quadcopter enters an unarmed state upon initializing, which prevents it from providing throttle to the motors. The status of this safety feature can be viewed and toggled from the mission planner.

For more information on the telemetry and mission planner, please refer to our previous report [7].

[1] Available from http://copter.ardupilot.com/wiki/initial-setup/common-install-mission-planner/

5.0.9 Dataflash

While the telemetry link to the mission planner can be used to log data, this is both slow, 10Hz maximum, and unreliable. The on-board 16MB dataflash can be configured to log a variety of parameters at different frequencies, which can be extracted and exportet to Matlab by the mission planner software.

The standard logfiles include the *PM* which logs performance related to the workload of the APM, and *ATT* that logs the attitude at either 10Hz or 50Hz. It is also possible to modify the code and create a custom log.

While many logs are implemented by default, they are not necessarily enabled. Please refer to our previous report for details on how to use the dataflash [7], a section of which can be found in appendix A on page 114.

5.1. Software implementation

The autopilot firmware used on the APM is written primarily in a mixture of C++ and the Arduino language (based on the wiring framework [2]). At the core is the ArduPilot, and from this base, the APM branches out into ArduPlane, for fixed-wing aircrafts, ArduRover for land vehicles and ArduCopter for rotary-wing aircrafts, which is used for this project.

The ArduCopter project encompasses many features that is beyond the scope of this report. Although the code is open-source, a brief description of the features most critical to this project, are presented in this chapter.

5.1.1 Main loop, Scheduler and UserCode

During flight, the core of the autopilot is the main loop, loop(), in ArduCopter.ino. It is responsible for calling periodic methods at the right frequency. Using the instance scheduler from the AP_Scheduler class, different methods are called at the rate specified in the scheduler table in ArduCopter.ino. The rates are given in 10ms units, since the scheduler is updated at 100Hz. In addition to updating the scheduler, loop() directly calls fast_loop(), also at 100Hz.

The scheduler calls slower loops e.g. fifty_hz_loop at 50Hz, slow_loop at 10Hz and super_slow_loop at 1Hz. The loops are of particular interest, since they call the methods in UserCode.ino, at appropriate intervals. To keep the code neat and organized, users are encouraged to put as much custom code as possible into the methods in UserCode.ino. The file is organized with userhook_init(), a method for initializing variables, along with five methods, called at 100Hz, 50Hz, 10Hz, 3.3Hz and 1Hz.

5.1.2 Fast loop

The method fast_loop() in ArduCopter.ino is responsible for all the necessary tasks to control the quadcopter. During each iteration, the following important tasks are performed:

read_AHRS()[3] – Reads the values from the IMU and calculates both the DCM[4] in addition to the angles and rates in Euler angles. The angles in radians are stored in ahrs.roll_sensor, ahrs.pitch_sensor and ahrs.yaw_sensor, and the corresponding rates in radians per second are stored in omega.x, omega.y and omega.z, respectively.

[2] Wiring is an open-source programming framework for microcontrollers, http://wiring.org.co/
[3] Attitude and Heading Reference System
[4] Direction Cosine Matrix

update_trig() – Calculates a number of parameters derived from the DCM for later use.

run_rate_controllers()[5] – Calls the rate PID controllers for roll, pitch and yaw, with the desired rate from the higher level controllers, and saves the result for each channel.

set_servos_4()[6] – Calls a chain of methods ending in output_armed() in AP_MotorsMatrix.cpp. This is the mixer function, which mixes the desired output for each channel into an appropriate signal for each motor. This is explained in section 5.1.6 on page 38.

read_radio()[7] – Reads the input from the RC transmitter.

read_control_switch()[8] – Updates the flight mode.

update_yaw_mode() – Run the appropriate yaw angle PID controller based on the mode. These include the yaw-navigation controllers, so the reference can be based on either stick input, GPS data or a mix. The output from the controller is used as the target rate for the lower level controller, and is stored for later use.

update_roll_pitch_mode() – Run the appropriate roll and pitch angle controller, in a similar fashion to update_yaw_mode().

update_rate_contoller_targets()[9] – Converts the target rates to body frame using the results from update_trig(), if they have been calculated in earth frame. This will depend on the control mode.

Finally the fast loop also calls user loops, which will be used for implementing the L1 controller later on.

Notice that the loop takes the minimum number of steps necessary from reading the IMU to calculating the rate controller outputs to setting the motors signals. The attitude controllers are run at the end of the loop, with the same IMU data, but their output is not used until the next loop, 10ms later. This implementation has likely been chosen, since the IMU readings of the rate becomes irrelevant faster than the attitude readings.

It is also worth noting that the throttle controller is updated in fifty_hz_loop(), and for this reason runs at half the rate.

5.1.3 Flight Mode Implementation

Each flight mode is divided into a roll/pitch control scheme, a yaw control scheme and a throttle control scheme. Each mode has a corresponding roll/pitch, yaw and throttle controller, and these can be shared amongst flight modes, e.g. the “altitude hold” and “circle”-mode, share the same throttle scheme, while the roll/pitch and yaw controllers are different.

The controllers used for each mode is defined in config.h, and the setup when changing modes is done in set_mode(mode) in system.ino.

5.1.4 AC_PID & APM_PI

The library used by ArduCopter contains two very similar controller implementations. AC_PID is a PID controller with a low-pass filter with a cut-off frequency at 20Hz. The derivative contribution is calculated as follows:

[5] In Attitude.ino
[6] In motors.ino
[7] In radio.ino
[8] In control_modes.ino
[9] In Attitude.ino

```
...
    derivative = (input - _last_input) / dt;
...
    derivative = _last_derivative + (dt / ( _filter + dt)) * (derivative -
        _last_derivative);
...
    _last_input     = input;
    _last_derivative    = derivative;
...
    return _kd * derivative;
...
```

Figure 5.1: *snippet from AC_PID.cpp*

where "input" is the error and "dt" is the sampling time (0.01). The filter constant is hardcoded to $7.9577 \cdot 10^{-3}$. The integration is calculated as:

```
...
    _integrator += ((float)error * _ki) * dt;
    if (_integrator < -_imax) {
        _integrator = -_imax;
    } else if (_integrator > _imax) {
        _integrator = _imax;
    }
    return _integrator;
...
```

Figure 5.2: *snippet from AC_PID.cpp*

notice the integrator limit "_imax".

The controller given in APM_PI[10] is practically identical, with the D-contribution removed. The PID-controller implementation in simulink is shown in figure 5.3.

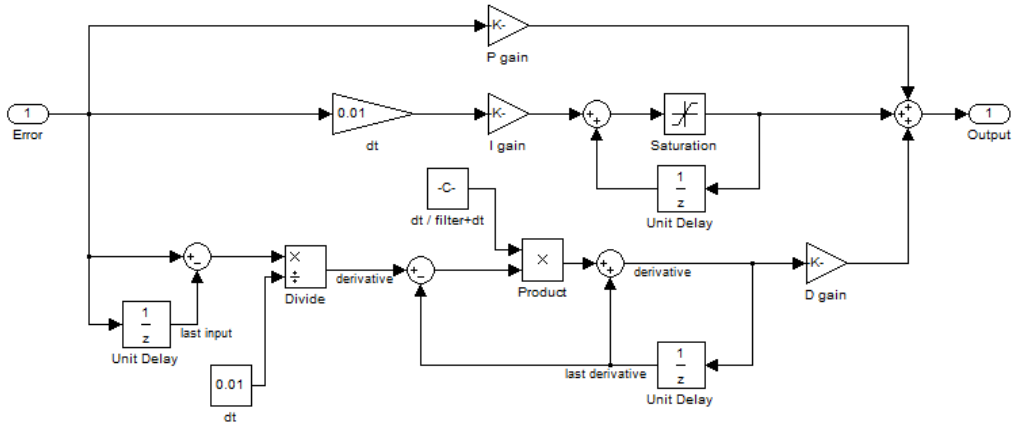


Figure 5.3: *Simulink implementation of the AC_PID controller. The APM_PI controller is identical, except that the D term is removed.*

The table 5.1 on the next page shows what controllers are used in the ArduCopter:

[10] APM_PI.cpp

Roll	APM_PI
Pitch	APM_PI
Yaw	APM_PI
Alt hold	APM_PI
Roll rate	AC_PID
Pitch rate	AC_PID
Yaw rate	AC_PID
Throttle rate	AC_PID
Throttle acceleration	AC_PID

Table 5.1: *What controller uses which control scheme.*

5.1.5 Frame Configuration

Each motor has a predefined contribution to roll, pitch and yaw, depending on the multirotor configuration that has been selected. The configuration used in this project is known as a *plus frame*, and it is important to distinguish it from the *X frame*, since they both consists of four motors set in increments of 90 degrees. The plus frame has a pair of motors along each axis, whereas the X frame is rotated 45 degrees around the Z-axis, and thus each motors absolute contribution to pitch and roll is equal.

The plus frame is defined in setup_motors() as:

```
add_motor(AP_MOTORS_MOT_1,  90, AP_MOTORS_MATRIX_YAW_FACTOR_CCW, 2);
add_motor(AP_MOTORS_MOT_2, -90, AP_MOTORS_MATRIX_YAW_FACTOR_CCW, 4);
add_motor(AP_MOTORS_MOT_3,   0, AP_MOTORS_MATRIX_YAW_FACTOR_CW,  1);
add_motor(AP_MOTORS_MOT_4, 180, AP_MOTORS_MATRIX_YAW_FACTOR_CW,  3);
```

Figure 5.4: *snippet from AP_MotorsQuad.cpp*

where

```
AP_MOTORS_MATRIX_YAW_FACTOR_CW = -1
AP_MOTORS_MATRIX_YAW_FACTOR_CCW = 1
```

and add_motor(...) is the following method from AP_MotorsMatrix.cpp:

```
void AP_MotorsMatrix::add_motor(int8_t motor_num, float angle_degrees, float
    yaw_factor, uint8_t testing_order)
{
    // call raw motor set-up method
    add_motor_raw(
        motor_num,
        cosf(radians(angle_degrees + 90)),              // roll factor
        cosf(radians(angle_degrees)),                   // pitch factor
        yaw_factor,                                     // yaw factor
        testing_order);
}
```

Figure 5.5: *snippet from AP_MotorsMatrix.cpp*

The method add_motor_raw(...) is from the same file, and used to store the contributions to roll, pitch and yaw for a motor as _roll_factor, _pitch_factor and _yaw_factor.

Notice from setup_motors() and add_motor(...), that the pitch and roll contribution is a value between -1 and 1 depending on the the motors location on a circle in the x,y-plane. The roll, pitch and yaw factors represents a single motors contribution relative to the other motors, and are used in the mixer, which will be explained next. For this reason the yaw factor is simply determined by the direction of the rotor.

For a custom multirotor, the contributions should be specified using add_motor_raw(), see AP_MotorsOcta.cpp, for an example.

5.1.6 Mixer

Since each motor affects altitude, yaw and either pitch or roll, it is necessary to implement mixing of the controller signals. The mixer used in ArduCopter is very versatile and supports a great number of multicopter configurations, but this adds to the complexity. The mixer also accounts for different flight scenarios such as take off and perform a number of checks unrelated to normal flight. To reduce the complexity only the key features of the mixer is taken into account when building the simulink model.

The mixer is implemented in the *output_armed()* method in AP_MotorsMatrix.cpp, and uses the controller outputs stored in _rc_roll, _rc_pitch, _rc_yaw and _rc_throttle to calculate the signals. The following code explains how the control signals from the roll and pitch controllers affect a given motor:

```
rpy_out[i] = _rc_roll->pwm_out * _roll_factor[i] +
                             _rc_pitch->pwm_out * _pitch_factor[i];
```

Figure 5.6: *snippet from AP_MotorsMatrix.cpp*

where rpy_out[i] is the motor signal for motor i. The control signal is multiplied with the appropriate factor to determine the total signal. This factor was specified in section 5.1.5 on the preceding page.

The following code adds the control signal for yaw in a similar fashion:

```
rpy_out[i] =    rpy_out[i] +
                                  yaw_allowed * _yaw_factor[i];
```

Figure 5.7: *caption here*

The variable "yaw_allowed" is a saturated version of the yaw rate controller signal, based on the RC throttle input, the extrema of the current motor signals and various throttle-related constants. As explained previously, the value of _yaw_factor will be either -1 or 1, depending on the rotor's direction.

The final part of the mixer adds the throttle contribution, and this is an equal amount for all motors. The throttle signal that is added to the motor signals have been modified based on the size of the other control signals, as well as the throttle at hover, among others.

If a multicopter is working close to its limits, or during a rapid maneuver, the motors might reach a saturation point. When a motor signal is saturated, attitude control is diminished, or even lost in extreme cases, and to mitigate this, a scaling is introduced. Should any motor signal exceed either the upper or lower limit of the motor signals, the entire lot of motor signals are scaled by a factor *rpy_scale*. This factor is calculated based on the rpy value that exceeded the limit, along with the values *thr_adj* and

out_best_thr_pwm. These are both calculated based mainly on the roll, pitch and throttle controller inputs.

The following diagram explains the mixer in broad strokes. For our simulink implementation, please refer to appendix C on page 120.

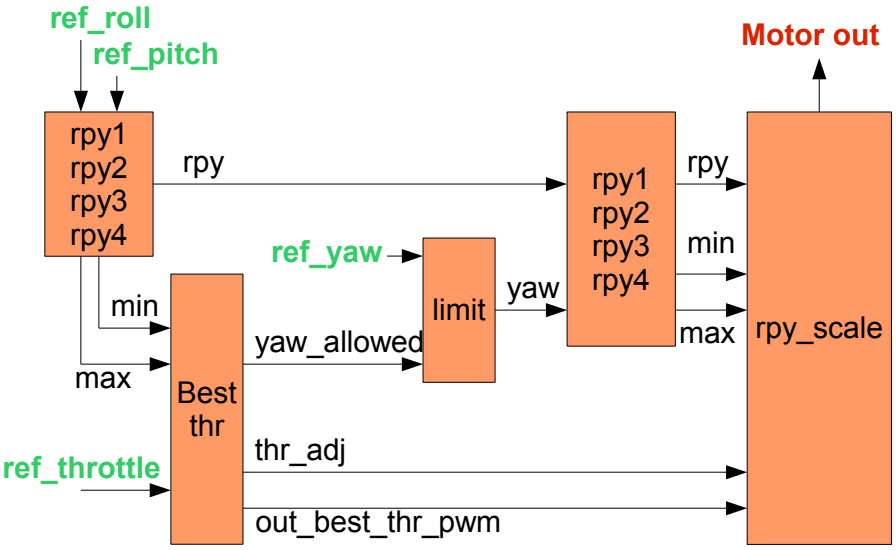


Figure 5.8: *A simplified flowchart explaining the mixer. The inputs are in green and the output is in red.*

5.1.7 APM Simulink Implementation

The simulink implementation consists of an APM subsystem and a quadcopter subsystem, as seen in figure 5.9. The quadcopter subsystem is based on the state equations derived in 2.1 on page 5 and its simulink implementation will be described in 6.1 on page 45. The APM subsystem is based on the code described in this chapter and will be described here.

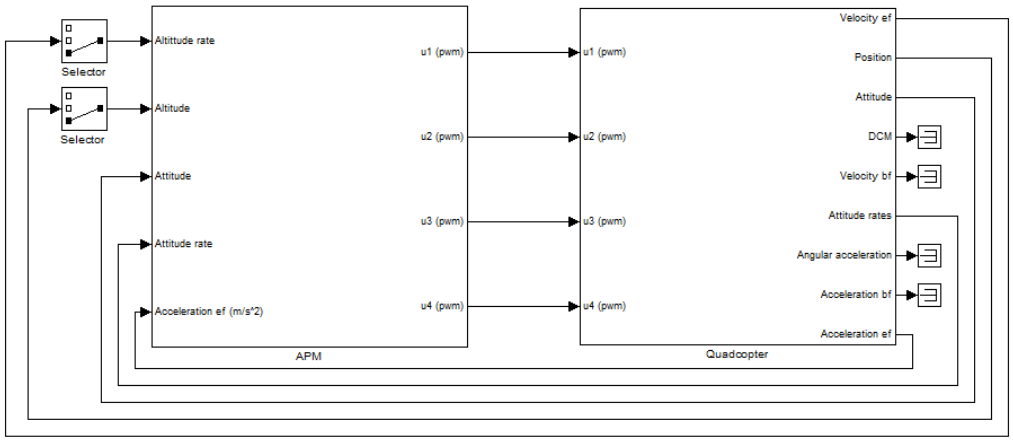


Figure 5.9: *The top level of the simulink implementation. The APM part is contained in the left block, and the physical representation of the quadcopter in the right.*

The APM block contains five important subsystems, as seen on 5.10 on the next page.

"References" manages the output to the system, which must be specified in a matlab script. Notice that the controllers either recieve the reference or the L1 control signal, based on a switch.

"L1 Controller" is the L1 adaptive controller implementation, which will be described in 9.1 on page 75.

"Converter" converts the output from the quadcopter system to the values that the APM would use.

"Controllers" contain the pitch, roll, yaw and throttle controllers as previously described in this chapter.

"Mixer advanced" is the mixer, also described in this chapter.

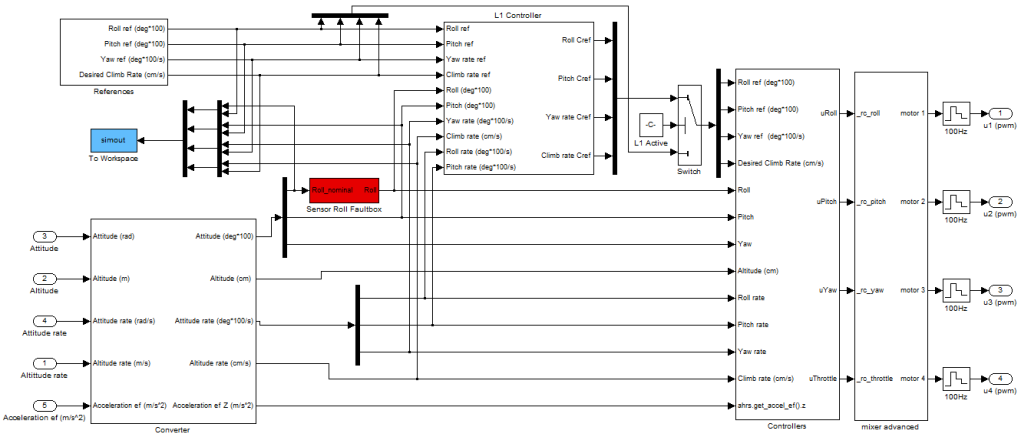


Figure 5.10: *The APM implementation in simulink. The red box is used to implement a sensor fault, which will be described later.*

Please refer to the simulink model on the CD, for a detailed view of each subsystem.

5.2. Controller Implementation

The altitude hold flight mode implementation in the APM software consists of four different PID controller loops. The roll and pitch controllers maintain a roll and pitch angle based on a roll and pitch reference. The yaw and altitude controller maintains the current heading/altitude, while receiving the yaw rate and climb rate as references.

All the methods in this section are found in *Attitude.ino*, unless otherwise stated.

This section aims to give a brief explanation of the APM controller implementation used in this project, but the reader is encouraged to explore the simulink model featured on the CD, for additional details.

5.2.1 Roll/Pitch Controller

Because of the symmetry of the roll and pitch, this section will only focus on the roll controller, since the pitch controller is implemented identically.

The roll controller consists of three steps, as explained in figure 5.11 on the next page. The first is a PI controller that calculates a roll rate reference, based on the RC or L1 roll reference input. The next is a transformation of the roll rate reference from earth frame to body frame. The last step is to calculate a control signal based on the roll rate reference, using a PID controller.

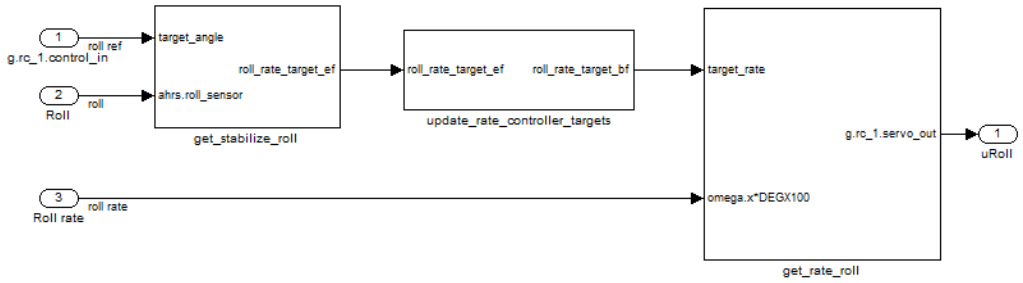


Figure 5.11: *Overview of the pitch controller implementation.*

In order to find the shortest path from the roll reference and the roll angle, the error between then is wrapped into a value between]-180,180]. This signal is then limited to an interval of [-4500, 4500], and this signal is send into the PI controller. The output from the controller is a signal called *roll_rate_target_ef* and it represents the target roll rate in earth frame. The method *update_rate_contoller_targets(...)*[sic] converts this to a body frame roll rate, based on the current attitude. Since this project focus stabilized flight, with limitations to the roll and pitch angles, the roll rate in earth frame is assumed to be equal to the roll rate in body frame, and so this implementation of the method simply passes the signal directly through.

The rate roll controller given in the method *get_rate_roll(...)*, calculates the target rate minus the current rate as an error to the PID-controller *pid_rate_roll*[11] and the output from the PID controller is limited to an interval of [-5000,5000].

5.2.2 Yaw controller

The yaw controller differs from the roll and pitch controller in that it receives a rate, rather than an angle and as a result, it must calculate an angle reference. A Simulink model of the angle controller, implemented in *get_yaw_rate_stabilized_ef(...)*, is shown on figure 5.12.

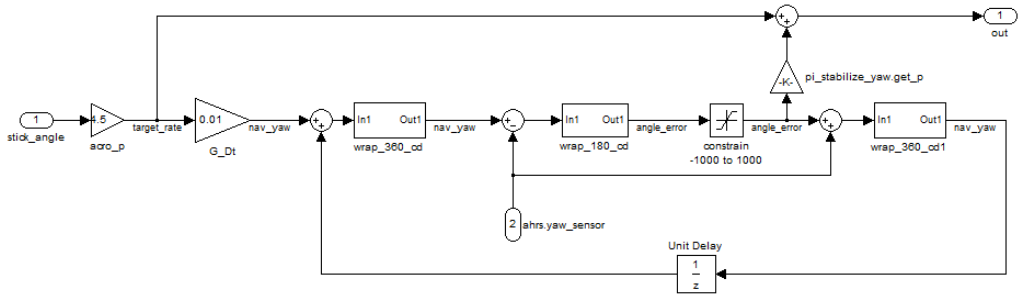


Figure 5.12: *The yaw heading controller. The "nav_yaw"-loop updates with the stick input and is used to maintain the yaw heading.*

The variable *nav_yaw* depends on both the stick (or L1) input and the current heading. This variable is in essence the yaw angle reference, which is used to calculate the yaw angle error, *angle_error*. Due to the constraint, the angle error can only assume values between [-1000, 1000] (centi-degrees), and in turn, the biggest difference between the current heading and the desired heading, nav_yaw, is an absolute value of 1000. The value of nav_yaw will remain constant, when the stick input is 0, as long as the aircraft is able to maintain the desired yaw angle.

[11] The pitch PID controller is pid_rate_pitch

The output from the angle controller becomes a sum of the desired rate from the stick input and the desired rate from the PI-controller *pi_stabilize_yaw*.

As with the roll/pitch-controller, the desired rate is in earth frame and is converted to body frame using *update_rate_contoller_targets(...)*, and for the same reasons as explained before, this implementation assumes the values to be equal.

The rate controller is run in *get_rate_yaw*, which is very similar to the roll/pitch rate controller implementation. The current rate is subtracted from the target rate from the angle controller, and the error is fed into the PID controller *pid_rate_yaw*. The output is constrained between [-4500, 4500]. The point where the yaw controller differs is that the output from the rate controller is constrained between [-RC rate reference-2200, RC rate reference+2200]. This dampens the output if the stick is released, and allows the pilot to override the controller.

The simulink implemtation is shown in figure 5.13.

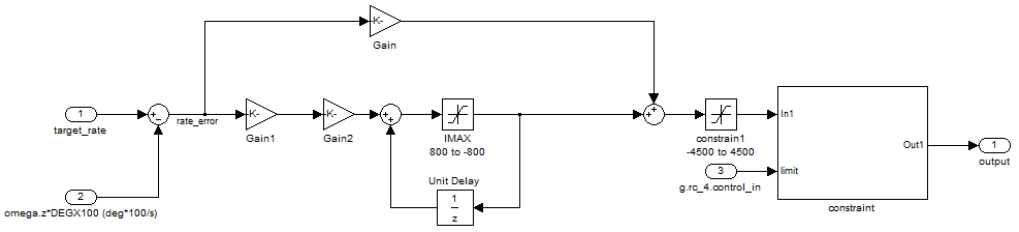


Figure 5.13: *The yaw rate controller implementation. Notice the variable constraint block which allows the pilot to override the controller.*

5.2.3 Altitude controller

The throttle RC (or L1) input controls the climb rate of the quadcopter. The neutral position consists of a dead zone from 40%-60%, in which the quadcopter will maintain its current altitude. The method *get_pilot_desired_climb_rate(...)* manages the dead zone, and pass the output as the desired climb rate. The method *get_throttle_rate_stabilized(...)* works in the same way as the yaw angle controller, in that the desired altitude is incremented based on the desired climb rate, with a constraint that limits the difference between the desired and current altitude to an absolute value of 750, which corresponds to 7.5 meters.

The desired altitude is used in *get_throttle_althold(...)* to calculate a desired rate, using the *pi_alt_hold* PI controller, and constraining the output to [-500, 500]. The desired rate is used in *get_throttle_rate(...)*, shown in the figure 5.14 on the facing page. This method calculates the desired acceleration needed to reach the desired climb rate, using two filters and the *pid_throttle* PID controller. The output is constrained to [-32000, 32000].

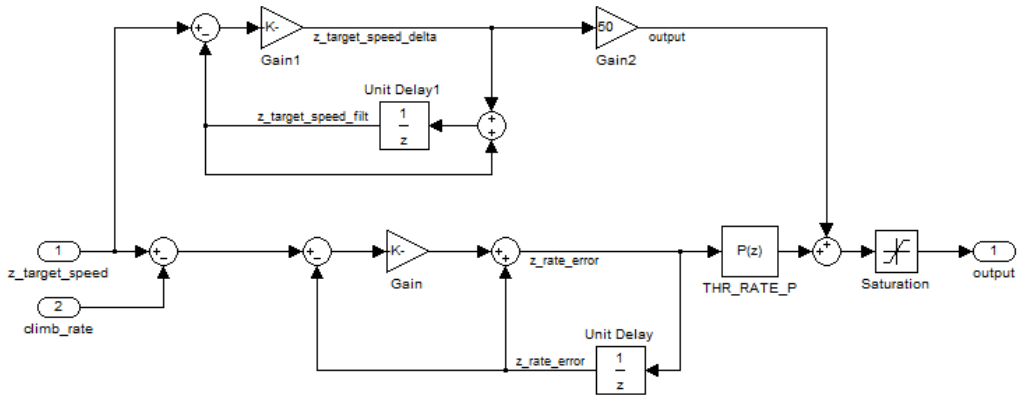


Figure 5.14: *Climb rate controller implementation.*

Although the altitude and climb rate controllers run at 50Hz, the final part of the altitude controller runs at 100Hz. The method *get_throttle_accel(...)* is called in *run_rate_controllers(...)* at 100Hz, together with the other controllers, as previously mentioned. As seen in the figure 5.15, the acceleration error, measured in cm/s^2, is constrained to [-32000, 32000], and is then put through a low pass filter with a cross frequency of 2Hz.

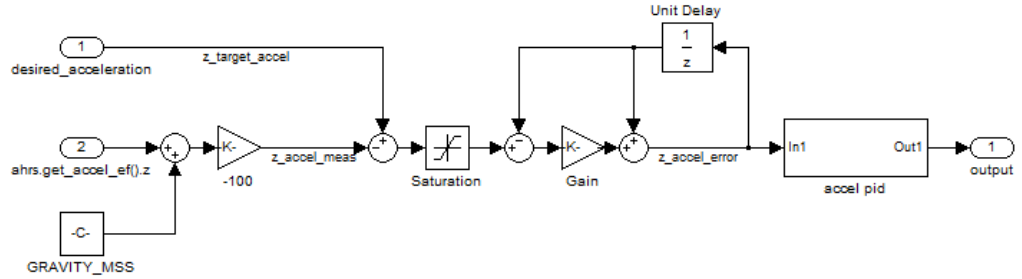


Figure 5.15: *Climb acceleration controller implementation.*

The final tasks is to run the error through the PID controller *pid_throttle_accel* and constrain the output to a value between [0, 1000].

5.2.4 Tuning

The quadcopter is tuned with a trial and error method, based on the tuning tutorial available on the DIY Drones website [16]. This method is easy and fast[12], and has been chosen, since PID tuning is not the focus of this project. The most important downside to the trail and error approach is the risk of crashing the quadcopter. To prevent accidents, the quadcopter was guyed to the ground, leaving little room for vertical movement, and limiting the attitude. When the quadcopter was able to maintain its attitude, the wires were removed, and the controllers were fine tuned.

The following controller gains were found to provide satisfactory performance:

[12]The quadcopter was tuned in less than 10 minutes

name	value	name	value
Pitch rate controller		**Throttle rate controller**	
RATE_PIT_P	0.09	THR_RATE_P	6
RATE_PIT_I	0.5	THR_RATE_I	0.0
RATE_PIT_D	0.005	THR_RATE_D	0.0
RATE_PIT_IMAX	500	THR_RATE_IMAX	300
Roll rate controller		**Throttle accel controller**	
RATE_RLL_P	0.09	THR_ACCEL_P	0.75
RATE_RLL_I	0.5	THR_ACCEL_I	1.5
RATE_RLL_D	0.005	THR_ACCEL_D	0.0
RATE_RLL_IMAX	500	THR_ACCEL_IMAX	500
Yaw rate controller		**Alt hold controller**	
RATE_YAW_P	0.2	THR_ALT_P	1
RATE_YAW_I	0.02	THR_ALT_I	0.0
RATE_YAW_D	0.0	THR_ALT_IMAX	300
RATE_YAW_IMAX	500		
Pitch angle controller		**Roll angle controller**	
STB_PIT_P	3	STB_RLL_P	3
STB_PIT_I	0.0	STB_RLL_I	0.0
STB_PIT_IMAX	800	STB_RLL_IMAX	800
Yaw angle controller			
STB_YAW_P	1		
STB_YAW_I	0.0		
STB_YAW_IMAX	800		

Table 5.2: *The controller gains in the APM*

CHAPTER 6

Theoretical Model

6.1. Building a Model in Simulink

The math behind the model derived in 2.1 on page 5 and combined with the constants found in 4.2 on page 22, it will be implemented in Matlab Simulink, which allow simulations of the system to be made. The physical part of the Simulation is continous, while the APM part is discrete. This section will focus on the implementation of the physical system in Simulink, which is illustrated in figure 6.1.

The model is initialized by running *prepareSimulink.m*, available on the CD.

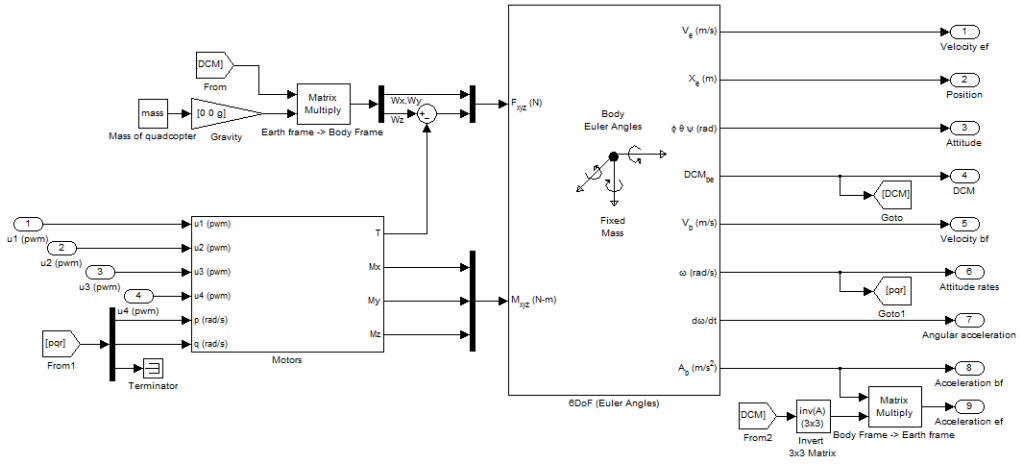


Figure 6.1: *The physical part of the quadcopter implemented in simulink.*

The four inputs u1 through u4 are the controller signals from the APM. The main parts of this model is the Euler block, used to calculate movement, and the Motors block, which calculates the thrust and moment from the motors. The small system in the top-left is the gravity and thrust system.

6.1.1 Gravity and Thrust

Gravity acts solely on the vertical axis in earth frame, but the force vector used in the Euler block requires the input to be in body frame, so the DCM is used to transform the

force. The thrust from the quadcopter is added to the force vector, before it is fed to the Euler block.

Notice that only the forces of gravity and thrust affect the quadcopter in this model. Forces such as wind and wind resistance are ignored, since they mainly affect the longitudinal and lateral position of the quadcopter, and this is beyond the scope of this project. As a result the quadcopter does not have a maximum speed.

6.1.2 Euler 6DoF Block

The Euler 6 degrees of freedom block calculates position and velocity in earth frame, the attitude, the attitude rates in body frame and the DCM, among others[1]. The initial position, velocity, attitude and attitude rates must be specified, together with the intertia matrix and the mass of the object. The inputs to the block is a vector of the sum of forces acting on the system in body frame and a vector of the applied moments in body frame. The Euler block merely calculates the movement based on the inputs, and does not provide any physical entities such as gravity or wind.

6.1.3 Motor System

The motor system receives a PWM signal for each motor from the PWM. This is used to calculate the thrust and applied moments around the x,y and z axis in body frame. The PWM signal is converted into an angular velocity in *Calculate angular rate* and a time delay is applied in *Motor dynamics*. The four angular rates *Omega1* through *Omega4* are used to calculate the thrust in *Calculate thrust*, and the moments in *Calculate Mx*, *Calculate My* and *Calculate Mz*. The gyroscopic moment of the propellers is calculated in *Calculate Hz* and is scaled with the quadcopters pitch and roll rate. The result is added to the roll and pitch moment, respectively. The entire motor system is visible in the figure 6.2.

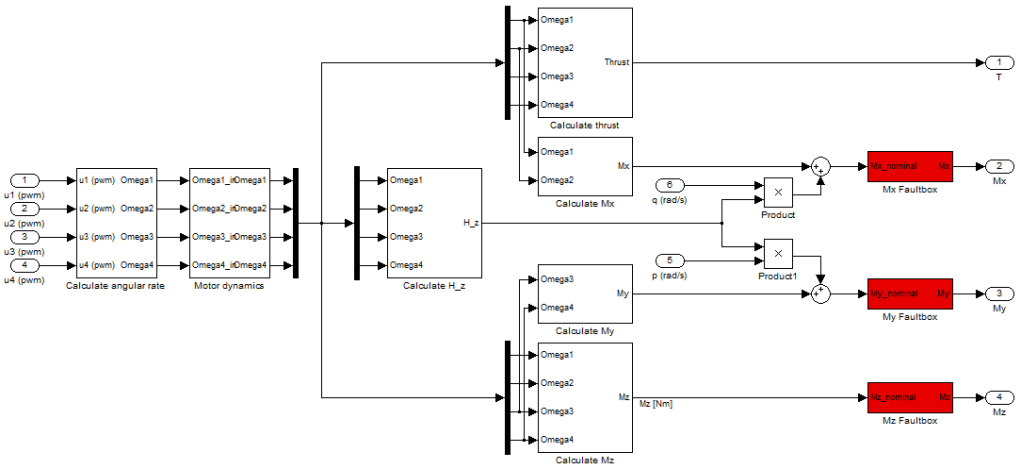


Figure 6.2: *Implementation of the motor system.*

The red blocks are used for implementing faults into the system. A list of faults is available in *L1_errors_init.m* on the CD. The subsystems are based on the formulas in section 2.1 on page 5, and a sample of the implementation of *Calculate Mx* can be seen in the figure 6.3 on the next page.

[1] Please refer to the Simulink help file for a full list of outputs.

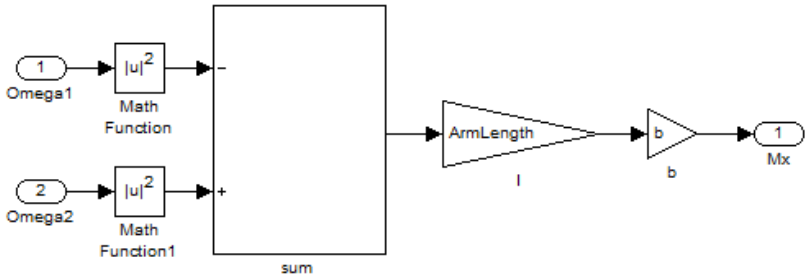


Figure 6.3: *Implementation of the calculation of applied moment around the x axis.*

6.1.4 PWM to Angular Velocity

The PWM signal each motor receives is converted to an angular velocity in two steps, the first being a conversion and the next being a transfer function. The conversion uses the steady state relationship between PWM and angular velocity, which was derived in section 4.2.2 on page 25.

The PWM conversion is seen in figure 6.4.

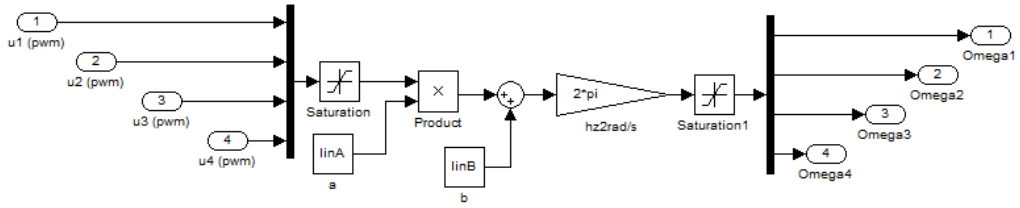


Figure 6.4: *Conversion of the PWM signal from the APM to an angular velocity.*

The second part, calculating the time delay, was done using an oscilloscope. The motor dynamics derived in section 4.2.6 on page 30.

The motor dynamics have been implemented as four identical state space system blocks, seen in figure 6.5.

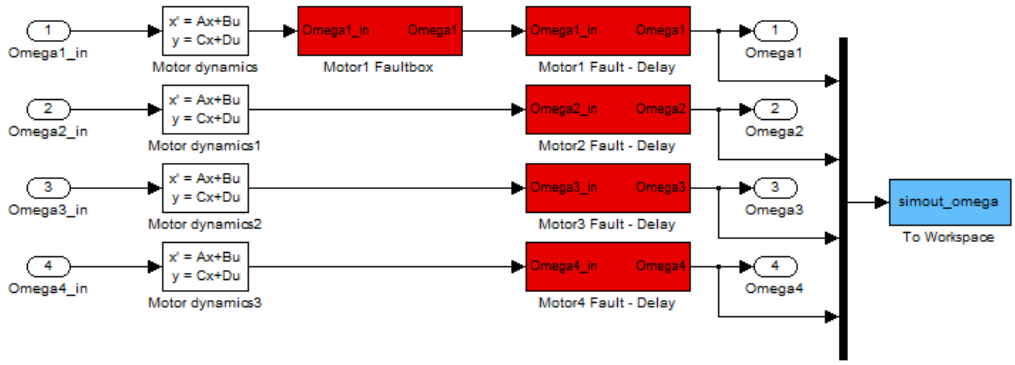


Figure 6.5: *Conversion of the PWM signal from the APM to an angular velocity.*

Once again the red blocks are used to introduce faults to the system.

6.2. Simulink Model Verification

In the previous section 6.1 on page 45 a SIMULINK model of the quadcopter with controllers has been created. In this section this model will be tested and verified, with input take from the real system in flight.

Comparison to Flight Data

The first idea was to use the input to the motors (the PWM-signal), but since the quadcopter is unstable without controllers even the smallest disturbances and model variations will result in an unstable output, so this option will not work. Instead it was decided that the pilot input was used as a reference, and the simulink output was compared to the flight data. The pilot input was fed into the APM controllers, previously implemented in simulink, and the controller output was sent to the quadcopter implementation. This method assured stability of the model through the controllers, and allowed the quadcopter and APM model to be verified at the same time.

The quadcopter was flown in altitude-hold mode, in this mode the pilot controls the desired angle on roll and pitch, on yaw and altitude the rate is controlled, when the rate is zero the system will maintain the current heading or altitude by itself. Three switches had been programmed on the transmitter to make a step the pitch angle, the yaw rate and the climb rate. The manoeuvres were executed one at a time. Roll have not been examined, due to the fact that is very similar to pitch, the only real difference of these two systems are in the inertia, and that difference is relatively small.

The result can be seen on figures 6.6, 6.8 on page 50, 6.7 on the next page, *To repeat the simulation, please run TEST1_Model_Verification_....m* (The end of the file name specifies the motion simulated).

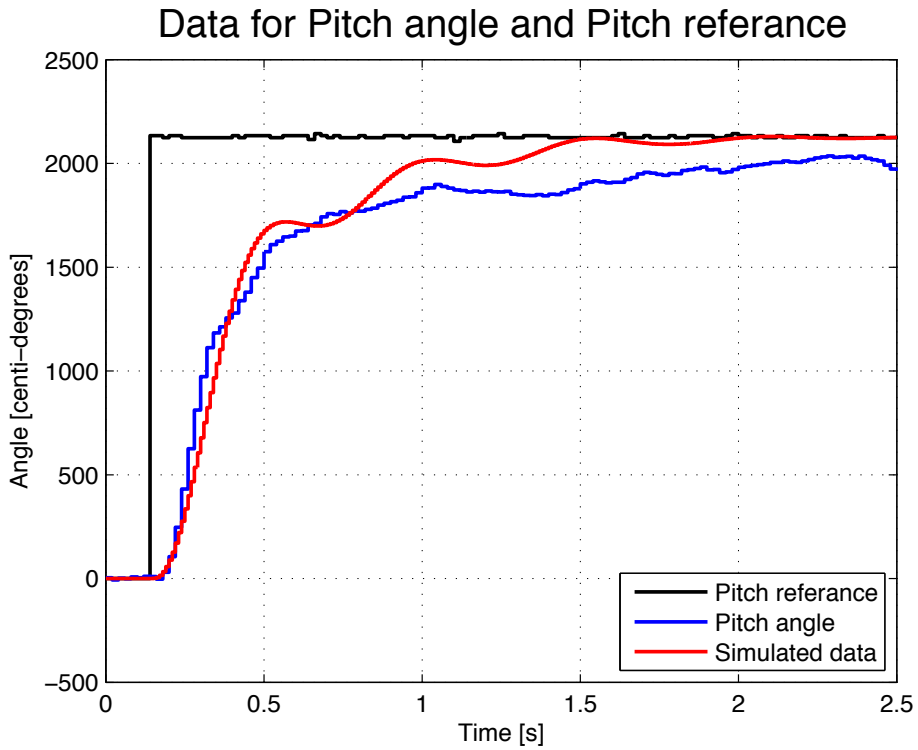


Figure 6.6: *Pitch input and output of the real system and the modelled system.*

It is seen that pitch fit very well, the simulated system behaves as if it is a higher order then the real system, but the responses are still very similar. It is also worth noticing that the system is far away from its working point (at 0°), and still the model preforms very well.

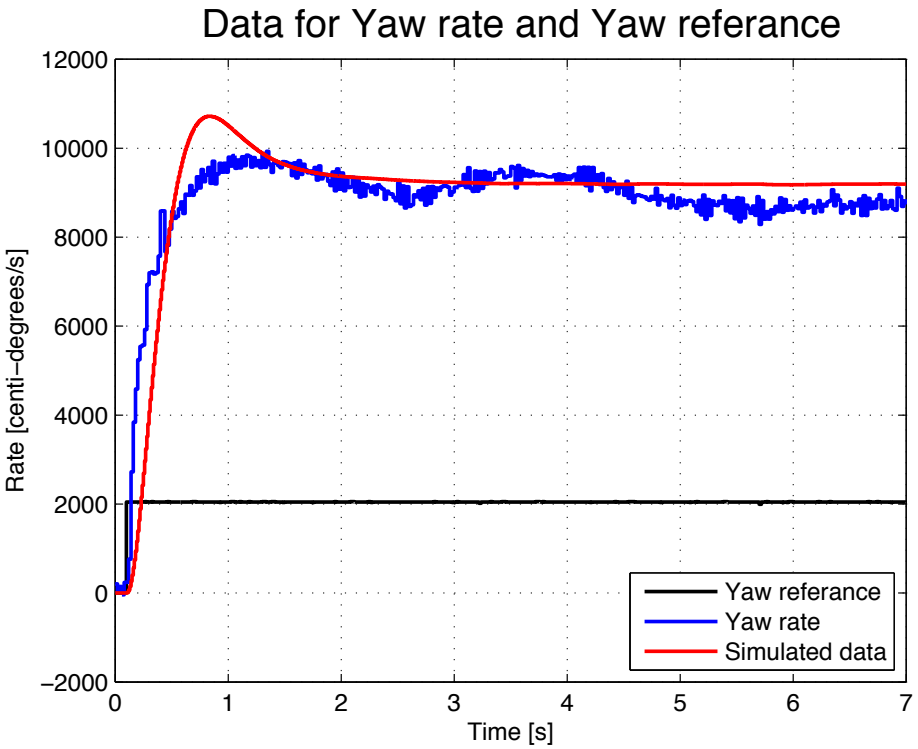


Figure 6.7: *Yaw input and output of the real system and the modelled system.*

The yaw movement also fits well with the real system, here it looks like the real system is of a higher order then the simulated one, due to oscillations, this might also come from wind noise and other disturbances. Apart from a overshoot the yaw motion in simulink tracks the real system well.

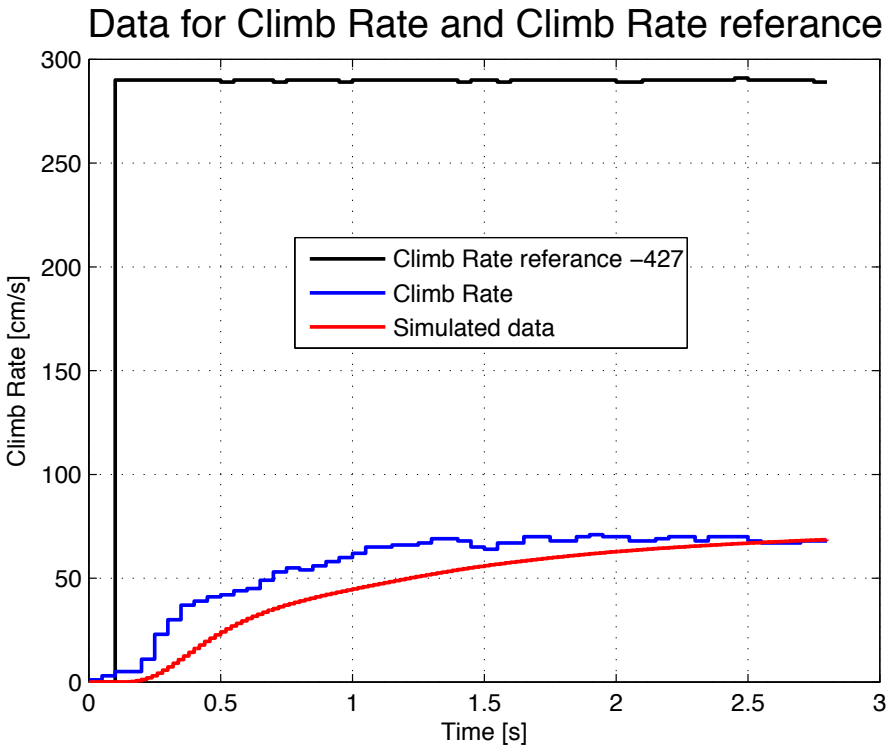


Figure 6.8: *Throttle input and climb rate output of the real system and the modelled system.*

The climb rate works is a similar fashion to yaw, the pilot does not control the altitude directly, but rather the climb rate, so when the rate is zero the altitude controller will maintain the altitude it is in. The simulated system is slower then the real system, this might come from the fact that the "PWM-to-RPS" function is not linear as simulated. The fact that the system is slower at changing altitude will not affect the simulated model much.

The previous figures shows the transition of the system (a step response), the next figures will show how well the model tracks the real system over a longer period of time. The two figures 6.9 on the facing page and 6.11 on page 52 show such behaviour, *To repeat the simulation, please run TEST2_Model_Verification_....m* (The end of the file name specifies the motion simulated).

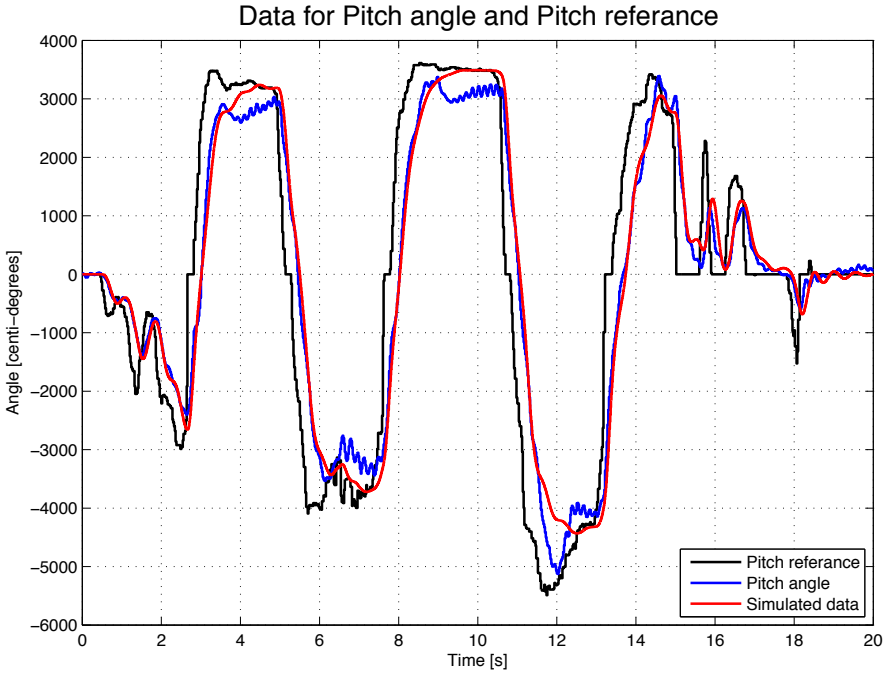


Figure 6.9: *Pitch changes over a period of 20sec.*

Note that figure 6.9 shows the system far away form its working point, but the simulated system still tracks well, notices the start where the input is smaller, this part has been magnified on figure 6.10.

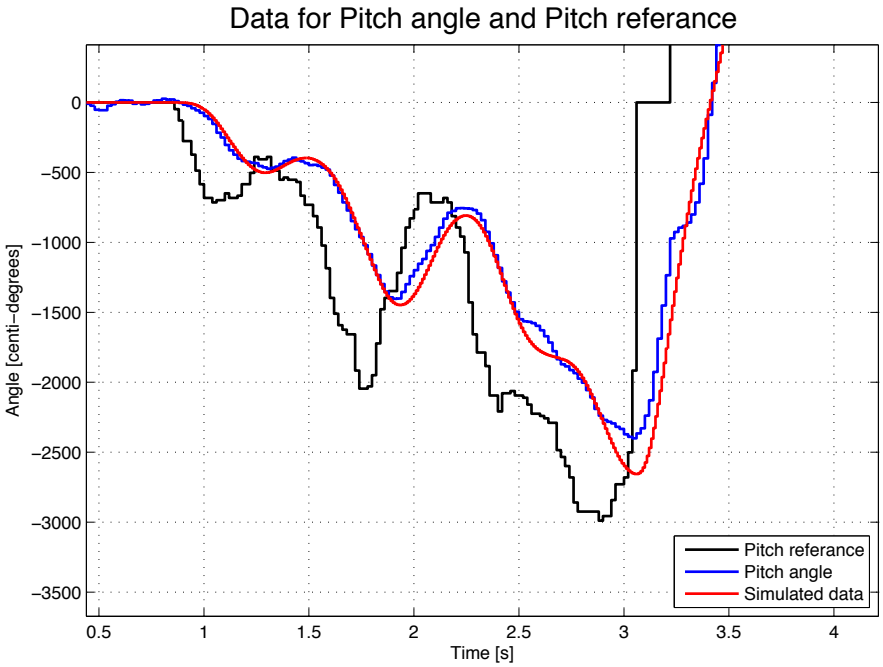


Figure 6.10: *Zoomed in on the start of figure 6.9*

This shows that the simulated system tracks very well for small changes, which is expected since this would be very close the the working point.

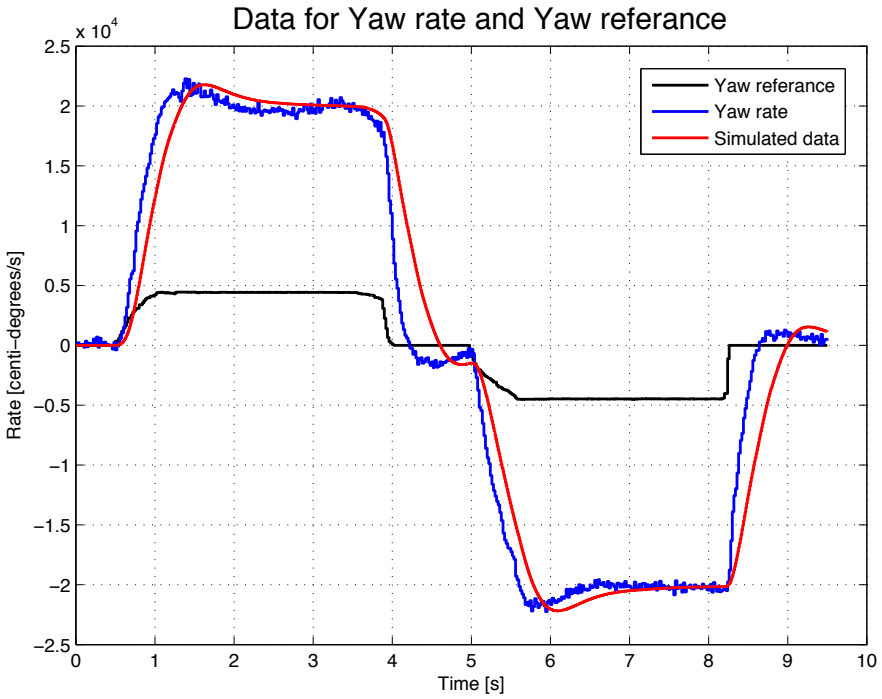


Figure 6.11: *Test of yaw rate*

The simulated yaw rate follows the real system very nice, figure 6.11 show the quadcopter turning at 200 deg/s, this is very fast compared to the working point, but the simulated system still tracks and preformed very well.

For the purpose of getting a better simulated system, the PID gains could have been tuned to better match the real system. This was not done, because changing the gains might solve for this particular case, but it is unknown how it will affect the flight envelope of the simulated system and how well it compares to the real system. One might also have made a black box for the system, but a downside to this is that a lot of the system parameters are lost in a model derived like that, so it would be unclear how e.g. a change in inertia would affect the system performance.

In general for all the plots it is seen that the simulated system is little slower then the real system, but still tracks the system good enough to be used to estimate the systems behaviour in simulations.

6.3. Error Handeling

To repeat the simulation below, please run "TEST" and "TEST2" scripts in "System with Errors (No L1)" on the CD In the previous sections a simulink model has been derived, it has been tested to show that the simulations fit well with the real system. So now it can be used to take a look at how the systems behaves when there is faults present. In the following only one reference will be stepped at a time.

6.3.1 Multiplicative Fault

First a multiplicative fault of 0.65 have been placed on motor 1 (the starboard motor, see figure 2.1 on page 6) and the responses of roll angle, yaw rate and climb rate have been plotted.

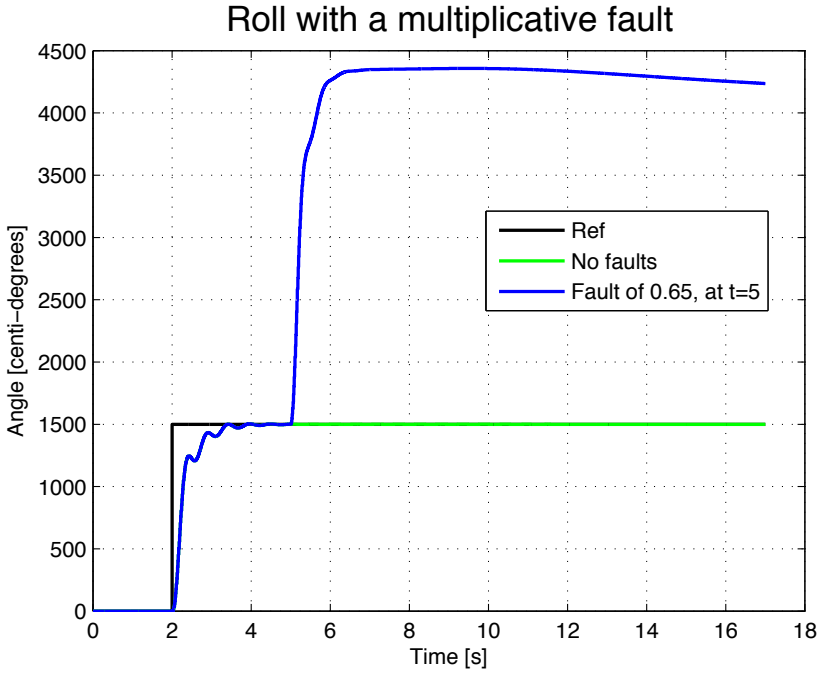


Figure 6.12: *The roll angle with a fault of 0.65 at time 5*

If one motor losses performance, could be a damaged propeller this is the response one might expect. As seen on figure 6.12, the system finds a new "steady state", at almost 43 degrees. This is not a real steady state since the system will go to the reference eventually, but it will take a long time to do so, and at such a steep roll angle the quadcopter will move very fast in the earth frame.

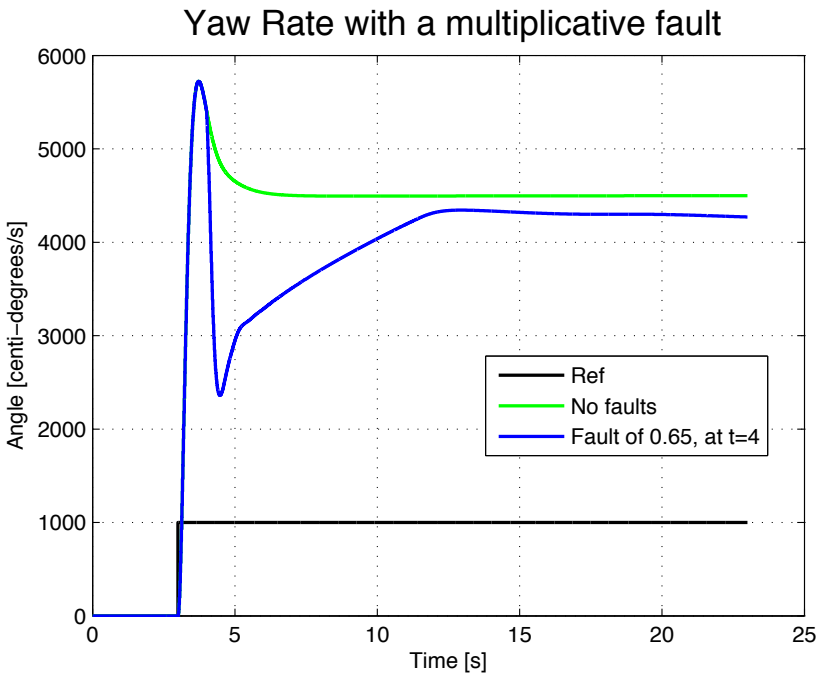


Figure 6.13: *The yaw rate with a fault of 0.65 at time 5*

The impact of the fault on figure 6.13 is not as bad as on the roll angle, there is some transition where the rate is far from the nominal system. But after about 6 sec the rate is maintained at almost the same level as the nominal system

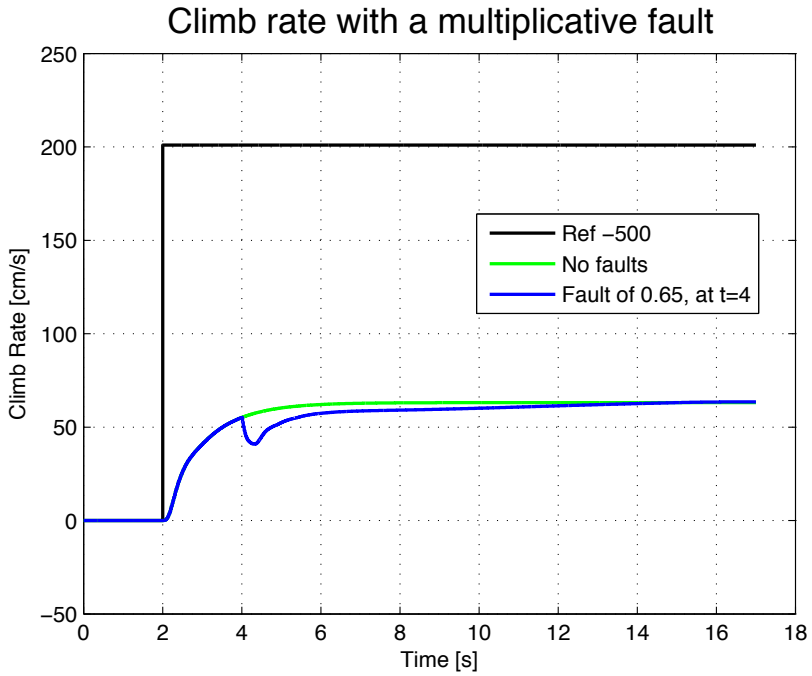


Figure 6.14: *The climb rate with a fault of 0.65 at time 5*

The impact seen on figure 6.14 is not every large, the faulty system preforms very

much like the nominal system.

6.3.2 Additive Fault

Next an additive fault of -150 have been placed on motor 1 and the responses of roll angle, yaw rate and climb rate have been plotted.

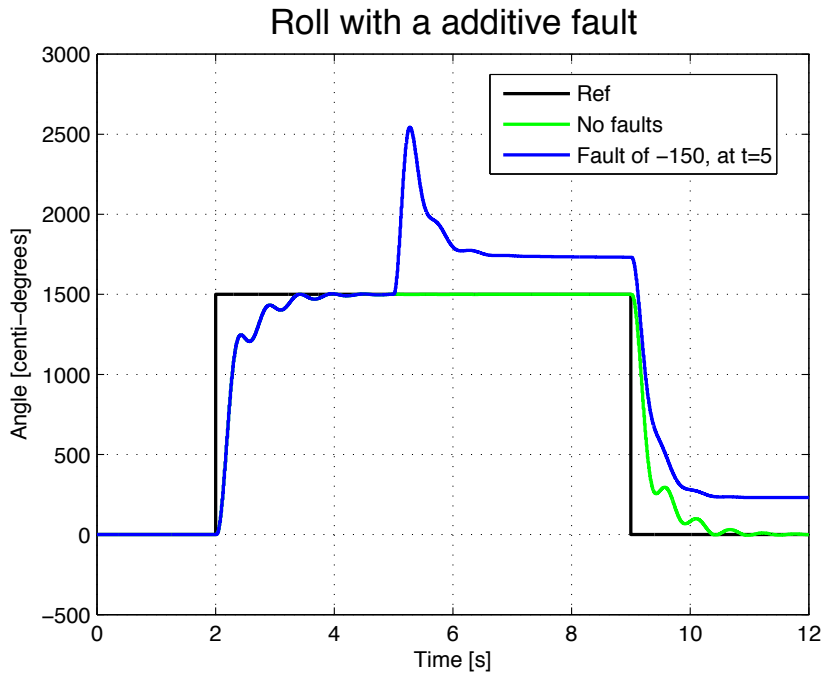


Figure 6.15: *The roll angle with a fault of -150 at time 5*

As seem from figure 6.15, during this fault the system does not differ much from the nominal system, here is a steady state error of about 2 degrees.

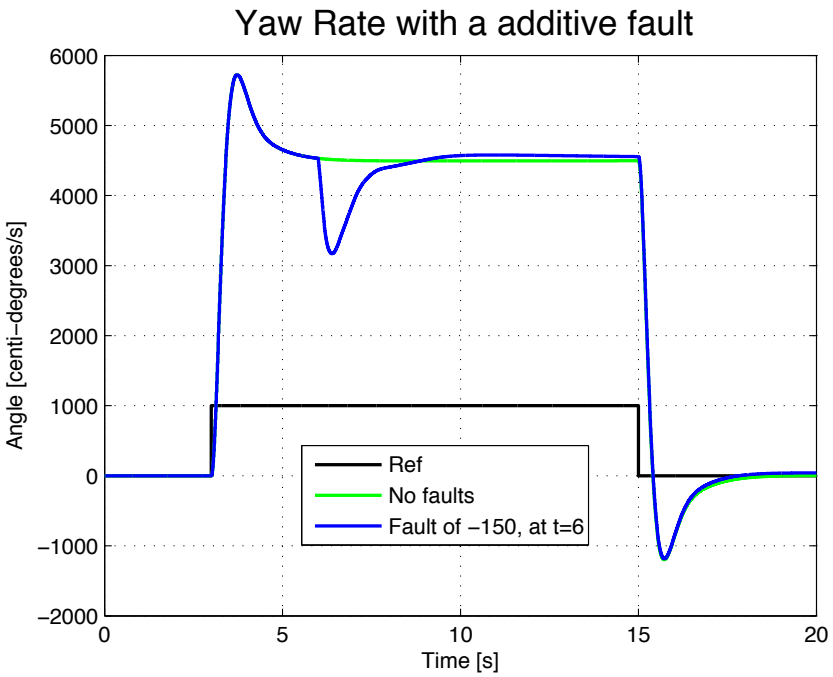


Figure 6.16: *The yaw rate with a fault of -150 at time 5*

The story is the same with the yaw rate on figure 6.16, there is a spike when the fault happens, but the system can cope with it fine. Even then the reference goes back to 0 the system almost follows the nominal system.

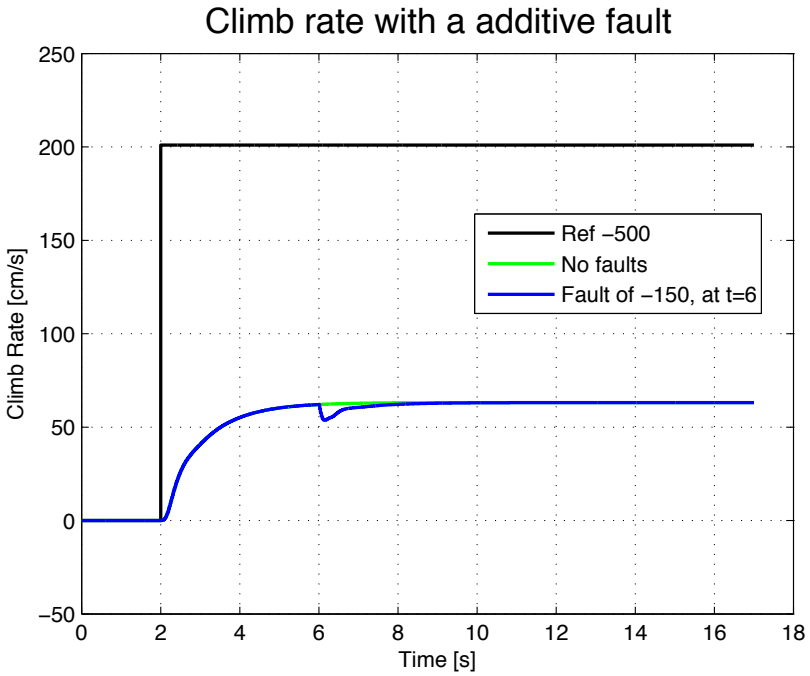


Figure 6.17: *The climbrate rate with a fault of -150 at time 5*

The fault does not affect the climb rate much, the faulty system is able to obtain

close to nominal preformance as seen on figure 6.17 on the preceding page.

CHAPTER 7

Part Conclusion

Up to now a frame, motor, battery and ECS combination has been chosen. This setup have been selected as a result of knowledge gained from an earlier project [7], research and have been aided by an online tool for calculating various motor parameters[1]. Each part has been recreated in a 3D CAD program[2], this allows to calculated the inertia matrix for the quadcopter better then what can be done by hand. This also enables the creation of more illustrative figures, seen through out this report, more photos and 3D renders can be seen in appendix D on page 122.

The parts has been assembled to a functional quadcopter, where the placement and mounting of the APM 2.0 (the sensor board) has been taken in to consideration. It has been placed on a vibration damping mount the differenced with and without mount can be seen on see figure 4.3 on page 19 and 4.4 on page 20. The APM and mount has been placed inside a box, seen on figure 4.2 on page 19 to protect it in case of a crash but also to shield the barometric sensor from the pressure fluctuations created by the propellers. The quadcopter has been tuned, by hand using an online guide [16], the gains found can be seen in table 5.2 on page 44. With the quadcopter fully functional various tests have been made to determine the parameters described in section 4.2 on page 22. The open source software of the APM, especially the controllers and the mixer, have been read through, and then recreated as close as possible in simulink. Some of the more advanced functions have been cut away in order to keep the system fairly simple.

The theory have been used to implement a model of the quadcopter in simulink, see section 6.1 on page 45. The main part in this model is the 6DOF block, but also the motor dynamics and effects such as the gyroscopic moment from the rotating propellers have been implemented as well.

This model have been verified in section 6.2 on page 48, and it functions very well. This means that the model can be used to estimate the real system when faults occurs, this has been done in section 6.3 on page 52. This shows that the system does not preform well when there are faults present, some faults is more significant then others.

The next part of this project will explain, implement and test an L1 adaptive control scheme to test if this can help suppress the faults. If the results are satisfactory the L1 controller will be implemented in the APM software and tested on the real system.

[1] eCalc at: http://www.ecalc.ch/xcoptercalc.php?ecalc&lang=en
[2] PTC CREO 2.0

Part III

L1 Control

CHAPTER 8

Introduction to L1 Adaptive Control

Adaptive control can be used to maintain nominal performance despite modeling uncertainties, initialization errors, non-linear dynamics and time-varying dynamics. The L1 adaptive controller has been developed from the model reference adaptive-control (MRAC), but the key difference is, that the L1 architecture focuses on compensating the low-frequency content of the uncertainties, as opposed to the entire frequency range[17]. The problem with MRAC is that even though the tracking error can be arbitrarily reduced, by increasing the adaption gain, the result is a high-gain feedback control, leading to high-frequency oscillations in the control signal and reduced tolerance to time delays. Efficient tuning of an MRAC system is difficult because of this tradeoff, which has been removed in L1. Since the adaption and robustness is decoupled, the adaption rate should be increased as much as the computer allows.

Most L1 adaptive control designs are build around a control system consisting of a plant and a controller. The controller is based on the assumption that the plant is linear and time-invariant, and the nominal performance of the control system is achieved when these assumptions hold. The purpose of the L1 controller is to maintain the nominal performance in the face of varying plant dynamics, and this is done by sending an appropriate control signal as a reference to the nominal controller of the plant. The L1 controller is added to a stable system to increase robustness and maintain nominal performance in the face of model variations, as opposed to increasing the performance of the system.

The L1 controller consists of two parts, disturbance estimation and control. The disturbance estimation is done through a state predictor and an adaption law together in a loop. Disturbances should be seen in a broader definition as deviations from the nominal dynamics due to the various events discussed above. Some L1 designs estimate changes in state, input and external disturbances separately, but the variations they cause can also be estimated together as an adaption estimate.

The state predictor is comparable to an observer of the control system, in that it estimates the control system states, using the same input. The observer gain has been replaced by adaption laws, and the controller gain has been augmented with the adaption estimates and various filters.

The difference between the measure states and the predicted states are used to update the adaption estimates periodically. The estimation loop is designed in a way that ensures stability, and avoid parameter drift, and this part of the L1-controller should be

run as frequent as possible. It is not necessary to update the control signals and the measurements as often as the estimation loop. In fact, the adaption laws can run at a much higher frequency, and should generally be as high as possible.

8.1. State Feedback L1 Controller

A MIMO state feedback L1 controller is presented in "L1 Adaptive Control Theory"[18], page 159. Consider the system:

$$\dot{\mathbf{x}}(t) = \mathbf{A}_m \mathbf{x}(t) + \mathbf{B}_m \boldsymbol{\omega} \mathbf{u}(t) + f(t, \mathbf{x}(t), \mathbf{z}(t))$$

$$\mathbf{y}(t) = \mathbf{C}_m \mathbf{x}(t)$$
$$\mathbf{x}(0) = \mathbf{x}_0$$

where $\mathbf{x}(t) \in \mathbb{R}^n$ is the measured system state vector, $\mathbf{u}(t) \in \mathbb{R}^m$ is the control signal, $\mathbf{y}(t) \in \mathbb{R}^m$ is the regulated output. The state matrix, $\mathbf{A}_m \in \mathbb{R}^{n \times n}$, the input matrix, $\mathbf{B}_m \in \mathbb{R}^{n \times m}$ and the output matrix $\mathbf{C}_m \in \mathbb{R}^{m \times n}$ describes the desired dynamics for the nominal closed loop system, including the controllers. All poles in $\mathbf{A}_m$ must have a negative real part ($\mathbf{A}_m$ is a Hurwitz matrix), and the system must be both controllable $(\mathbf{A}_m, \mathbf{B}_m)$ and observable $(\mathbf{A}_m, \mathbf{C}_m)$. $\boldsymbol{\omega} \in \mathbb{R}^{m \times m}$ is the uncertain system input gain matrix, which is replaced by $\mathbb{I}^{m \times m}$ in the nominal case.

The unmoddel non-linear dynamics are given as:

$$\dot{\mathbf{x}}_z(t) = g(t, \mathbf{x}_z(t), \mathbf{x}(t))$$
$$\mathbf{z}(t) = g_0(t, \mathbf{x}_z(t))$$
$$\mathbf{x}_z(0) = \mathbf{x}_{z0}$$

where $\mathbf{x}_z(t)$ is the state vector of the internal unmodeled dynamics, and $z(t)$ is the output vector. $g(t, \mathbf{x}_z(t), \mathbf{x}(t))$ and $g_0(t, \mathbf{x}_z(t))$ are unknown linear functions.

The state equation can be rewritten as:

$$\dot{\mathbf{x}}(t) = \mathbf{A}_m \mathbf{x}(t) + \mathbf{B}_m (\boldsymbol{\omega} \mathbf{u}(t) + f_1(t, \mathbf{x}(t), \mathbf{z}(t))) + \mathbf{B}_{um} f_2(t, \mathbf{x}(t), \mathbf{z}(t)) \tag{8.1}$$

where $\mathbf{B}_{um}$ is the unmatched uncertainty input and

$$\mathbf{B}^{-1} f(t, \mathbf{x}(t), \mathbf{z}(t)) = \begin{bmatrix} f_1(t, \mathbf{x}(t), \mathbf{z}(t)) \\ f_2(t, \mathbf{x}(t), \mathbf{z}(t)) \end{bmatrix}, \mathbf{B} = \begin{bmatrix} \mathbf{B}_m & \mathbf{B}_{um} \end{bmatrix} \tag{8.2}$$

The state predictor is defined as

$$\dot{\hat{\mathbf{x}}}(t) = \mathbf{A}_m \hat{\mathbf{x}}(t) + \mathbf{B}_m (\boldsymbol{\omega}_0 \mathbf{u}(t) + \hat{\boldsymbol{\sigma}}_1(t)) + \mathbf{B}_{um} \hat{\boldsymbol{\sigma}}_2(t) \hat{\mathbf{y}}(t) = \mathbf{C}_m \hat{\mathbf{x}}(t)$$

where $\hat{\mathbf{x}}(0) = \mathbf{x}_0$. The coloum vectors $\hat{\boldsymbol{\sigma}}_1(t) \in \mathbb{R}^m$ and $\hat{\boldsymbol{\sigma}}_2(t) \in \mathbb{R}^{n-m}$ are the adaptive estimates. $\hat{\boldsymbol{\sigma}}_1(t)$ and $\hat{\boldsymbol{\sigma}}_2(t)$ are calculated according to the adaption law:

$$\begin{bmatrix} \hat{\boldsymbol{\sigma}}_1(t) \\ \hat{\boldsymbol{\sigma}}_2(t) \end{bmatrix} = \begin{bmatrix} \hat{\boldsymbol{\sigma}}_1(kT_s) \\ \hat{\boldsymbol{\sigma}}_2(kT_s) \end{bmatrix}$$

$$t \in [kT_s, (k+1)\,T_s[$$

$$\begin{bmatrix} \hat{\boldsymbol{\sigma}}_1(kT_s) \\ \hat{\boldsymbol{\sigma}}_2(kT_s) \end{bmatrix} = -\begin{bmatrix} \mathbb{I}_m & \mathbf{0} \\ \mathbf{0} & \mathbb{I}_{n-m} \end{bmatrix} \mathbf{B}^{-1} \boldsymbol{\Phi}(T_s)\, e^{\mathbf{A}_m T_s} \tilde{\mathbf{x}}(kT_s)$$

where k is the sample number. The last part of the L1 controller is the control law, given as:

$$\mathbf{u}(s) = -\mathbf{K}\mathbf{D}(s)\,\hat{\boldsymbol{\eta}}(s)$$

where $\hat{\boldsymbol{\eta}}(s)$ is the Laplace transform of

$$\hat{\boldsymbol{\eta}}(t) = \boldsymbol{\omega}_0 \mathbf{u}(t) + \hat{\boldsymbol{\eta}}_1(t) + \hat{\boldsymbol{\eta}}_{2m}(t) - \mathbf{r}_g(t)$$

where $\hat{\boldsymbol{\eta}}_1(t) = \hat{\boldsymbol{\sigma}}_1(t)$, $\hat{\boldsymbol{\eta}}_2(t) = \hat{\boldsymbol{\sigma}}_2(t)$, $\hat{\boldsymbol{\eta}}_{2m}(s) = \mathbf{H}_m^{-1}(s)\,\mathbf{H}_{um}(s)\,\hat{\boldsymbol{\sigma}}_2(s)$ and $\mathbf{r}_g(s) = \mathbf{K}_g(s)\,\mathbf{r}(s)$, with

$$\begin{aligned} \mathbf{H}_{xm}(s) &= (\mathbb{I}_n s - \mathbf{A}_m)^{-1}\mathbf{B}_m \\ \mathbf{H}_{xum}(s) &= (\mathbb{I}_n s - \mathbf{A}_m)^{-1}\mathbf{B}_{um} \\ \mathbf{H}_m(s) &= \mathbf{C}_m \mathbf{H}_{xm}(s) \\ \mathbf{H}_{um}(s) &= \mathbf{C}_m \mathbf{H}_{xum}(s) \end{aligned}$$

Figure 8.1 shows a diagram of the L1 state feedback controller.

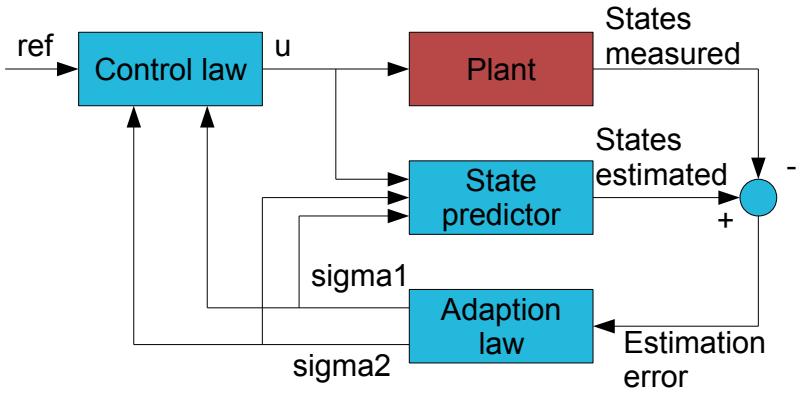


Figure 8.1: *The L1 state feedback adaptive controller in blue. The plant already contains the necessary controllers to make it stable.*

8.1.1 Requirements

The L1 adaptive controller theory contains a number of conditions which must be fulfilled, for the proofs to be valid. The controllability $(\mathbf{A}_m, \mathbf{B}_m)$, and observability $(\mathbf{A}_m, \mathbf{C}_m)$ matrices must have full rank. The uncertainty input matrix must be chosen, so that

$\mathbf{B}_m^T \mathbf{B}_{um} = \mathbf{0}$ and $rank\,(\mathbf{B}) = m, \mathbf{B} = \begin{bmatrix}\mathbf{B}_m & \mathbf{B}_{um}\end{bmatrix}$

The transfer functions in matrix $\mathbf{D}\,(s)$, must be strictly proper. Together with $\mathbf{K}$, $\mathbf{D}\,(s)$ must be chosen so that the transfer functions in matrix

$$\mathbf{C}\,(s) = \boldsymbol{\omega}\mathbf{K}\mathbf{D}\,(s)\,(\mathbb{I}_m + \boldsymbol{\omega}\mathbf{K}\mathbf{D}\,(s))^{-1}$$

is strictly proper stable, with the DC-gain, $\mathbf{C}\,(0) = \mathbb{I}_m$. The choice of $\mathbf{K}$ and $\mathbf{D}\,(s)$ must also ensure that $\mathbf{C}\,(s)\,\mathbf{H}_m^{-1}\,(s)$ is a proper stable transfer matrix.

Five assumptions are made about the system and the uncertainties in "L1 Adaptive Control Theory"[18], page 160. Please refer to this book for additional details.

I There exists $B_{k0} > 0$, so that $\|f_k\,(t,0)\|_\infty \leq B_k$ for all $t \geq 0$ and for $k = 1, 2$.

II For arbitrary $\delta > 0$, there exists a positive $K_{1\delta}$, $K_{2\delta}$ so that

$$\|f_k\,(t, X_1) - f_k\,(t, X_2)\|_\infty \leq K_{k\delta}\,\|X_1 - X_2\|_\infty\,, k = 1, 2$$

where $\|X_j\|_\infty \leq \delta, j = 1, 2$ uniformly in t.

III The x_z dynamics are BIBO stable with respect to initial conditions x_{z0} and input $x(t)$. That is, there exists a $L_z, B_z > 0$ so that for all $t \geq 0$

$$\|z_t\|_{L_\infty} \leq L_z\,\|x_t\|_{L_\infty} + B_z \tag{8.3}$$

The unmodeled dynamics are assumed to be stable, since the quadcopter was able to maintain hover, after it had been tuned.

IV The system input gain matrix ω is assumed to be an unknown strictly row-diagonally dominant matrix with a known sign for each element.

Also, it is assumed that thereexists a known compact convex set $\boldsymbol{\Omega}$ so that $\boldsymbol{\omega} \in \boldsymbol{\Omega} \subset \mathbb{R}^{m \times m}$ and that the nominal system gain $\boldsymbol{\omega}_0 \in \boldsymbol{\Omega}$ is known.

In the quadcopter, $\boldsymbol{\omega}_0 = \mathbb{I}_4$.

V The transmission zeros of the transfer matrix $\mathbf{H}_m\,(s)$ lie in the open left half plane.

Finally the choice of $\mathbf{K}$ and $\mathbf{D}\,(s)$ must ensure that for a given ρ_0 the following inequality must hold:

$$\|\mathbf{G}_m(s)\|_\infty + \|\mathbf{G}_{um}(s)\|_\infty\, l_0 < \frac{\rho_r - \|\mathbf{H}_{xm}(s)\mathbf{C}(s)\mathbf{K}_g\|_\infty - \rho_{in}}{L_{1\rho r}\rho_r + B_0}$$

where

$$\mathbf{G}_m(s) = \mathbf{H}_{xm}(s)(\mathbb{I}_m - \mathbf{C}(s))\mathbf{H}_{xum}(s)$$
$$\mathbf{G}_{um}(s) = (\mathbb{I}_n - \mathbf{H}_{xm}(s)\mathbf{C}(s)\mathbf{H}_m^{-1}(s)\mathbf{C}_m)$$

8.1.2 Design Considerations

The paper "L1 Adaptive Control Augmentation System with Application to the X-29 Lateral/Directional Dynamics: A Multi-Input Multi-Output Approach"[19] discusses a number of different ways to implement an L1 controller on the lateral and directional dynamics in an X-29 aircraft. They experiment with a separate L1 SISO system for the lateral and the directional dynamics, and an L1 MIMO system for the both of them. Their findings are that the MIMO system minimizes undesired responses due to coupling

between states, without losses to the tracking performance. For this reason a MIMO-implementation have been chosen.

The state feedback controller described in the previous section requires that all states in the system are measured, which creates a few issues. The state equations derived in section 2.1 on page 5, relied on the angular velocity of all the motors, and since these are not measured a workaround must be made. Another problem posed by this implementation, is that it is very demanding computation-wise. The state predictor and the adaption law must perform massive matrix-multiplications because of the many states present in the model[1]. The solution to both problems is to consider the system at a higher level. By approximating step responses from the reference to roll, pitch, yaw rate and climb rate, as first or second order systems, the nominal quadcopter, including controllers, can be reduced to four transfer functions, disregarding couplings. This greatly reduces the computation load on the CPU, and at the same time the L1 controller has access to measurements of all the states.

By disregarding the couplings in the approximations, the minimization of undesired couplings between roll, pitch, yaw rate and climb rate is lost. The gain from chosing a MIMO implementation is reduced to the performance boost of roll/roll rate and pitch/pitch rate.

The altitude hold flight mode takes the RC input for yaw rate and climb rate and calculates a navigation reference for yaw and altitude. This poses a problem, since the yaw angle and altitude becomes marginally stable "memory" states, only affected by the input. Because of the non-negative poles from these states, A_m is not Hurwitz and the L1 theory is no longer supported. The solution has been to disregard the states, when designing the L1 controller. This is not a problem, since the yaw angle and altitude are controlled by the pilot or a higher level navigation controller. The dynamics from the yaw angle and altitude controller are still present through the transfer function approximations of the system, previously described.

The higher level approach with estimated transfer functions in place of a theoretical state space approach, reduced the complexity, but initial testing of the system exposed problems related to fault estimation. A sudden additive fault in the angular velocity of a motor, affect the attitude rates directly, however the attitude is only indirectly affect (as evident by the state space model). To improve performance, the L1 system was augmented with the measurements of the roll and pitch rate. Since roll and pitch was directly affected by the L1 control signals, roll and pitch rate would enter the system as unmatched uncertainties. The disturbance estimation immediately improved, and the L1 performance improved dramatically.

The L1 controller uses the adaption estimates to calculate a control signal for the control system. The L1 controller will be unable to maintain the nominal performance, if the disturbance estimation if off.

8.2. Designing an L1 Controller for the Quadcopter

The desired system consisting of $\mathbf{A}_m$, $\mathbf{B}_m$ and $\mathbf{C}_m$ is a linearization of the nominal system in steady state hover. The linearization is based on first and second order approximations of step responses obtained from both the physical model and the simulation.

First order approximations were used from the reference to yaw rate and climb rate and second order approximations were used from reference to roll and pitch. The exact values were found through trail and error.

[1] In addition to two attitude states, three attitude rates states, climb rate and four motor states, states in the PID controllers must also be considered.

The following transfer functions, based on flight data from the real system, were found to be adequate approximations:

$$\frac{r_{ref}}{r} = \frac{53}{s+12}$$
$$\frac{w_{ref}}{w} = \frac{0.43}{s+1.8}$$

The following state space model, based on simulink data, were found to be an adequate approximation:

$$\begin{bmatrix} \dot{p}_x \\ \dot{\phi}_x \end{bmatrix} = \begin{bmatrix} -6 & -3 \\ 5 & 0 \end{bmatrix} \begin{bmatrix} p_x \\ \phi_x \end{bmatrix} + \begin{bmatrix} 1 \\ 0 \end{bmatrix} \phi_{ref} \tag{8.4}$$

$$\begin{bmatrix} p \\ \phi \end{bmatrix} = \begin{bmatrix} 1 & 3 \\ 30 & 0 \end{bmatrix} \begin{bmatrix} p_x \\ \phi_x \end{bmatrix} \tag{8.5}$$

Its important to note, that the states $\dot{p}_x$ and $\dot{\phi}_x$ do not equal the actual roll rate and the roll angle. The approximation was made from reference to the states, so the approximated states are the output of the desired system, rather than the states of the desired system. As a result the state estimator calculates $\dot{p}_x$ and $\dot{\phi}_x$ rather than $\dot{p}$ and $\dot{\phi}$. This is important to consider, when the state estimation error is calculated, since the measured states $\dot{p}$ and $\dot{\phi}$ must be converted. This is done by inverting the output matrix as in equation 8.6.

$$\tilde{\mathbf{x}}(t) = \hat{\mathbf{x}}(t) - \mathbf{C}_{m,aug}{}^{-1}\mathbf{y}(t) \tag{8.6}$$

where $\tilde{\mathbf{x}}$ is the state estimation error, $\hat{\mathbf{x}}$ are the estimated states, $\mathbf{C}_{m,aug}$ is the augmented output matrix and $\mathbf{y}$ are the measurements. The augmented output matrix is the desired output matrix, $\mathbf{C}_m$, augmented with the roll rate and pitch rate, as shown in 8.8 on page 71.

$$\mathbf{C}_{m,aug} = \begin{bmatrix} & & \mathbf{C}_m & & & \\ C_{p/\phi,21} & 0 & 0 & 0 & C_{p/\phi,22} & 0 \\ 0 & C_{q/\theta,21} & 0 & 0 & 0 & C_{q/\theta,22} \end{bmatrix} \tag{8.7}$$

Figure 8.2 on the next page, figure 8.3 on the facing page, figure 8.4 on page 70 and figure 8.5 on page 70 show the approximation compared to the simulink model.

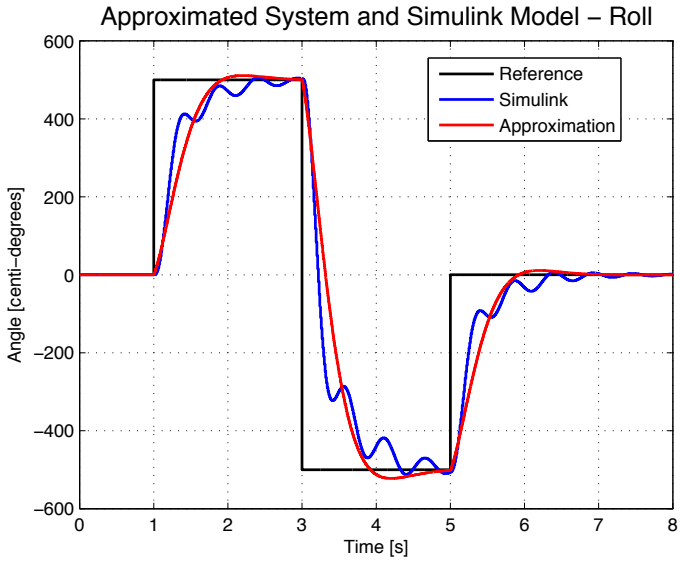


Figure 8.2: *The approximated system compared to the simulink model.*

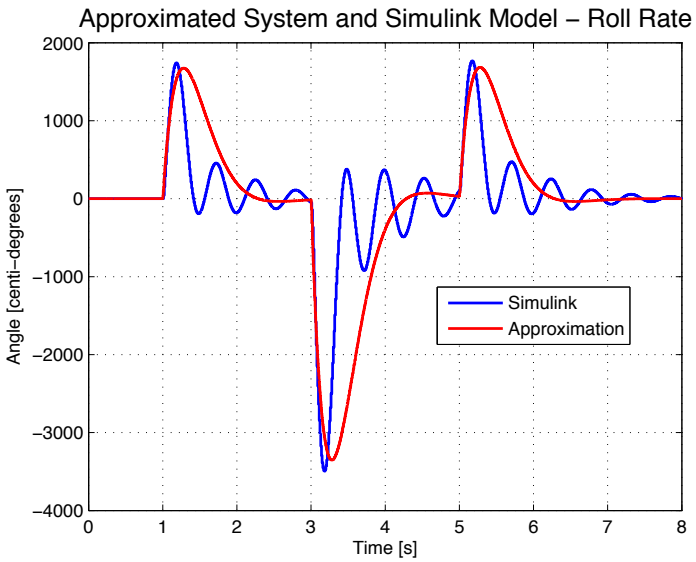


Figure 8.3: *The approximated system compared to the simulink model.*

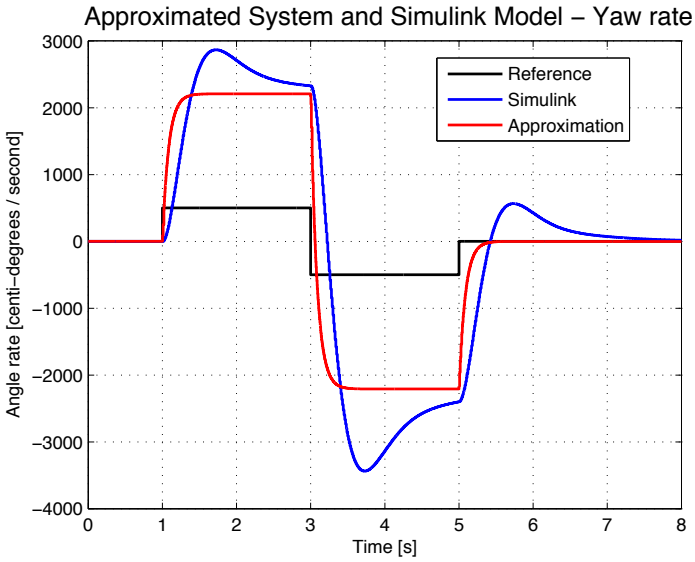


Figure 8.4: *The approximated system compared to the simulink model.*

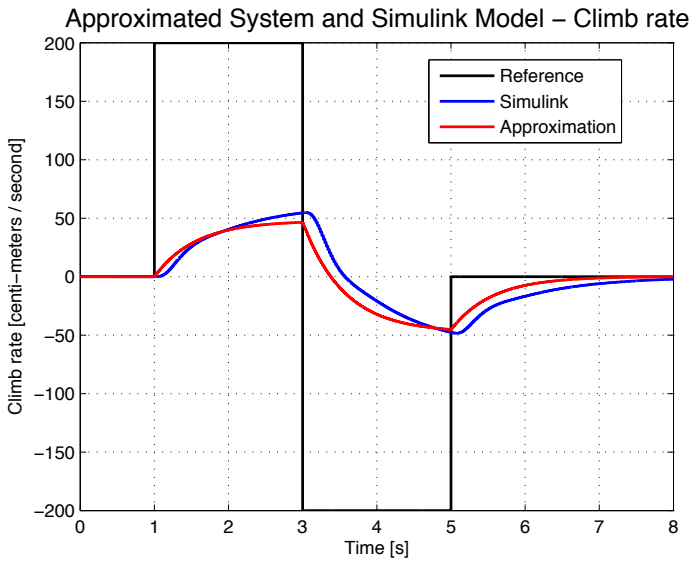


Figure 8.5: *The approximated system compared to the simulink model.*

The state space model of the desired system can then be constructed:

$$\mathbf{A}_m = \begin{bmatrix} A_{p/\phi,11} & 0 & 0 & 0 & A_{p/\phi,12} & 0 \\ 0 & A_{q/\theta,11} & 0 & 0 & 0 & A_{q/\theta,12} \\ 0 & 0 & A_r & 0 & 0 & 0 \\ 0 & 0 & 0 & A_w & 0 & 0 \\ A_{p/\phi,21} & 0 & 0 & 0 & A_{p/\phi,22} & 0 \\ 0 & A_{q/\theta,21} & 0 & 0 & 0 & A_{q/\theta,22} \end{bmatrix}$$

$$\mathbf{B}_m = \begin{bmatrix} B_{p/\phi,11} & 0 & 0 & 0 \\ 0 & B_{q/\theta,11} & 0 & 0 \\ 0 & 0 & B_r & 0 \\ 0 & 0 & 0 & B_w \\ B_{p/\phi,21} & 0 & 0 & 0 \\ 0 & B_{q/\theta,21} & 0 & 0 \end{bmatrix}$$

$$\mathbf{C}_m = \begin{bmatrix} C_{p/\phi,11} & 0 & 0 & 0 & C_{p/\phi,12} & 0 \\ 0 & C_{q/\theta,11} & 0 & 0 & 0 & C_{q/\theta,12} \\ 0 & 0 & C_r & 0 & 0 & 0 \\ 0 & 0 & 0 & C_w & 0 & 0 \end{bmatrix}$$

substituting values:

$$\mathbf{A}_m = \begin{bmatrix} -6 & 0 & 0 & 0 & -3 & 0 \\ 0 & -6 & 0 & 0 & 0 & -3 \\ 0 & 0 & -12 & 0 & 0 & 0 \\ 0 & 0 & 0 & -1.8 & 0 & 0 \\ 5 & 0 & 0 & 0 & 0 & 0 \\ 0 & 5 & 0 & 0 & 0 & 0 \end{bmatrix}$$

$$\mathbf{B}_m = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

$$\mathbf{C}_m = \begin{bmatrix} 1 & 0 & 0 & 0 & 3 & 0 \\ 0 & 1 & 0 & 0 & 0 & 3 \\ 0 & 0 & 53 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0.43 & 0 & 0 \end{bmatrix}$$

$$\mathbf{C}_{m,aug} = \begin{bmatrix} & & \mathbf{C}_m & & & \\ 30 & 0 & 0 & 0 & 0 & 0 \\ 0 & 30 & 0 & 0 & 0 & 0 \end{bmatrix} \tag{8.8}$$

The controllability $(\mathbf{A}_m, \mathbf{B}_m)$, and observability $(\mathbf{A}_m, \mathbf{C}_m)$ matrices are verified to have full rank. The unmatched uncertainty input matrix, $\mathbf{B}_{um}$, must be designed to comply with the requirements presented in section 8.1.1 on page 65. The matrix:

$$\mathbf{B}_{um} = \begin{bmatrix} 0 & 0 \\ 0 & 0 \\ 0 & 0 \\ 0 & 0 \\ 1 & 0 \\ 0 & 1 \end{bmatrix}$$

meets these requirements.

The performance of the L1 controller depends on the adaption sampling time, T_s. For the simulations, a value of 1/100, has been chosen, since 100Hz is the highest rate user code can be run in the APM software, without additional modification. The time-invariant parameters is the piece-wise adaption law, is replaced by the matrix, $\mathbf{G}$

$$\mathbf{G} = \begin{bmatrix} \mathbf{G}_1 \\ \mathbf{G}_2 \end{bmatrix}$$

where $\mathbf{G} \in \mathbb{R}^{n \times n}$, $\mathbf{G}_1 \in \mathbb{R}^{m \times n}$ and $\mathbf{G}_2 \in \mathbb{R}^{(n-m) \times n}$. Then

$$\begin{bmatrix} \hat{\boldsymbol{\sigma}}_1 (kT_s) \\ \hat{\boldsymbol{\sigma}}_2 (kT_s) \end{bmatrix} = - \begin{bmatrix} \mathbb{I}_m & \mathbf{0} \\ \mathbf{0} & \mathbb{I}_{n-m} \end{bmatrix} \mathbf{B}^{-1} \boldsymbol{\Phi} (T_s) \, e^{\mathbf{A}_m T_s} \tilde{\mathbf{x}} (kT_s)$$

becomes

$$\begin{bmatrix} \hat{\boldsymbol{\sigma}}_1 (kT_s) \\ \hat{\boldsymbol{\sigma}}_2 (kT_s) \end{bmatrix} = \begin{bmatrix} \mathbf{G}_1 \\ \mathbf{G}_2 \end{bmatrix} \tilde{\mathbf{x}} (kT_s)$$

where k is the sample number. Inserting numerical values yields:

$$\mathbf{G}_1 = \begin{bmatrix} -97.0175 & 0 & 0 & 0 & 1.4850 & 0 \\ 0 & -97.0175 & 0 & 0 & 0 & 1.4850 \\ 0 & 0 & -94.1200 & 0 & 0 & 0 \\ 0 & 0 & 0 & -99.1027 & 0 & 0 \end{bmatrix}$$

$$\mathbf{G}_2 = \begin{bmatrix} -2.4750 & 0 & 0 & 0 & -99.9875 & 0 \\ 0 & -2.4750 & 0 & 0 & 0 & -99.9875 \end{bmatrix}$$

Parts of the control law can also be calculated beforehand. $\mathbf{H}(s)$ contains only the following non-zero elements:

$$\mathbf{H}_{11} (s) = \frac{3s^7 + 69s^6 + 729s^5 + 4563s^4 + 18225s^3 + 46575s^2 + 70875s + 50625}{s^7 + 33s^6 + 423s^5 + 3051s^4 + 13635s^3 + 38475s^2 + 64125s + 50625}$$

$$\mathbf{H}_{22} (s) = \mathbf{H}_{11} (s)$$

8.2.1 Tuning

The fast adaptation of the L1 controller, results in a high frequency oscillations in $\hat{\boldsymbol{\sigma}}$. To prevent the high frequency oscillations from propagating through the system, a low pass filter is added to the the control law, and this must be tuned appropriately.

The tuning deals with three parameters simultaneously, $\mathbf{K}$, $\mathbf{K}_g$ and $\mathbf{D}(s)$. The pre-filter $\mathbf{K}_g$ is chosen so that the steady state error tracking error becomes minimal. The gain $\mathbf{K}$ and transfer function matrix $\mathbf{D}(s)$ are chosen so that the response to step inputs mimics the nominal system, while oscillations from $\hat{\boldsymbol{\sigma}}$ are minimal.

The parameters must be chosen so that the requirements from section 8.1.1 on page 65 are fulfilled. Parameters that result in good responses have been found through trial and error, and subsequently verified to meet the requirements. The minimum amount of coupling between the states, and the similar nature of the roll and pitch dynamics

reduces the complexity of this task, the the transfer functions in $\mathbf{D}(s)$ are nonzero in the diagonal and can be chosen independently. The same is true for $\mathbf{K}$ and $\mathbf{K}_g$.

The prefilter is set to:

$$\mathbf{K}_g = \begin{bmatrix} 2.000 & 0 & 0 & 0 \\ 0 & 2.000 & 0 & 0 \\ 0 & 0 & 1.015 & 0 \\ 0 & 0 & 0 & 1.200 \end{bmatrix}$$

The control gain and transfer functions are set to:

$$\mathbf{K} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 8 \end{bmatrix}$$

$$\mathbf{D}_{1,1} = \mathbf{D}_{2,2} = \frac{100s + 0.001}{s^2 + 100s}$$

$$\mathbf{D}_{3,3} = \frac{1}{0.01s^2 + 0.5s}$$

$$\mathbf{D}_{4,4} = \frac{1}{0.02s^2 + 4s}$$

The DC-gain of the $\mathbf{C}(s)$ transfer function matrix becomes

$$\mathbf{C}(0) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

and $\mathbf{C}(s)\,\mathbf{H}_m^{-1}(s)$ is a proper stable transfer function matrix, which is in accordance with the requirements.

The theoretical proof of stability based on the L1 norm is omitted, since it requires deeper research into the quadcopter dynamics than presented in this project, but it would be suitable for a theoretical project.

CHAPTER 9

L1 Control Implementation

9.1. L1 Simulink Implementation

In order to test the L1 controller, it has been implemented in the existing simulink model. The implementation consists of three subsystems, the state predictor, the adaption law and the control law. The implementation will follow the theory in the book "L1 Adaptive Control Theory"[18], so it will be a continuous implementation, with zero order holds.

In addition to the simulink model, a script *prepareSimulink.m*, is used to initialize variables, making modifications easier. This is available on the CD. Various parameters in the L1 controller are logged during a simulation, at the light blue blocks shown on figure 9.1 on the next page. These modifications allow the entire simulation to be run from a Matlab script.

The climb rate controller differs from the rest, in that it assumes values between 0-1000, and 500 is neutral, and in addition to this, a dead zone is implemented, so that the reference must be either lower than 400 or larger than 600 to be considered valid. The dead zone can create a difference between the input to the state estimator and the input to the actual system, which will result in an estimation error. To prevent this, a dead zone has been added to the L1 model right before the state estimator (the orange part in figure 9.1 on the following page). The control signal for throttle must exceed an absolute value of 100, otherwise it will be set to zero. The offset of 500 has been implemented at the input and output of the L1 controller (the cyan blobs on figure 9.1 on the next page).

Finally, since the state feedback L1 controller requires measurements of the states estimated by the state predictor, the measurements are converted to states by multiplying with ${C_m}^{-1}$ (the purple part in 9.1 on the following page). The exact reason for this is explained in section 8.2 on page 67, equation 8.6.

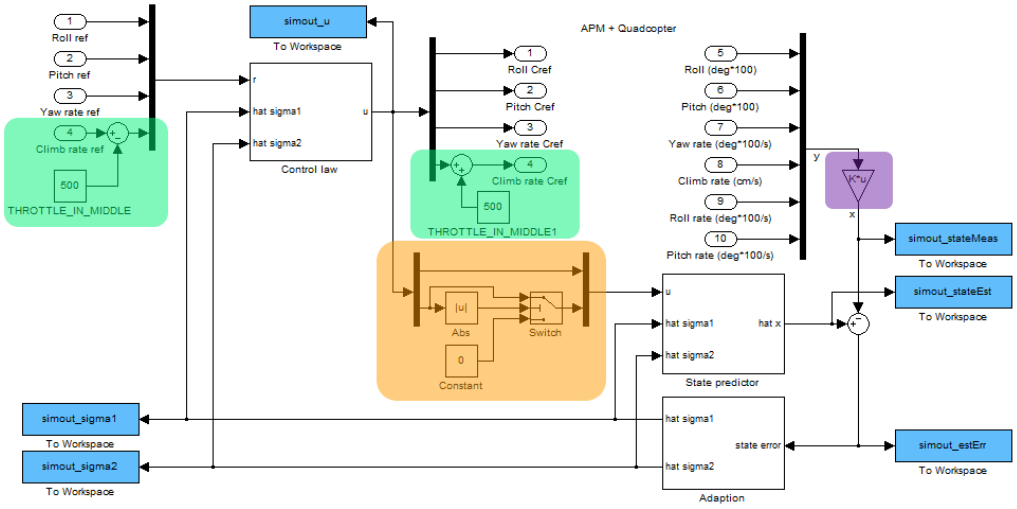


Figure 9.1: *The simulink implementation of the state feedback L1 adaptive controller.*

9.1.1 State Estimator and Adaption Law

The adaption law samples the estimation error based on *Ts*. The rate is set to 100Hz, since this is the fastest rate that user code will run on the APM without any modifications. The adaption law is split into calculating *sigma1* and *sigma2*, using *G1* and *G2*, respectively. These matrices are the time-invariant part of the adaption law, and are further explained in section 8.2 on page 67. The implementation can be seen in figure 9.2

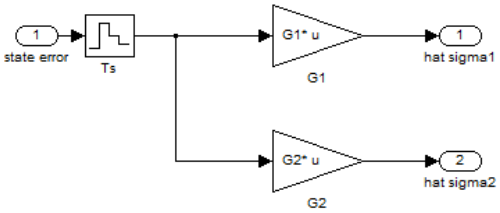


Figure 9.2: *The simulink implementation of the adaption law.*

The state estimator is a straight forward state space implementation, except for the addition of *Bm*, *Bum*, *sigma1*, *sigma2* and *omega0*, which are explained in the theory. The matrix *omega0* is an identity matrix, and could be omitted. Notice that the output is terminated and instead the state is used as an output.

The continuous state predictor is not possible to program into the APM, so it must be converted to discrete time.The simulink model follows the theory of the book [18], describing a continuous system, rather than a discrete one, however, tests has shown the difference between running the simulation with a continuous state predictor, instead of a discrete one, are minimal. The implementation can be seen in figure 9.3 on the facing page.

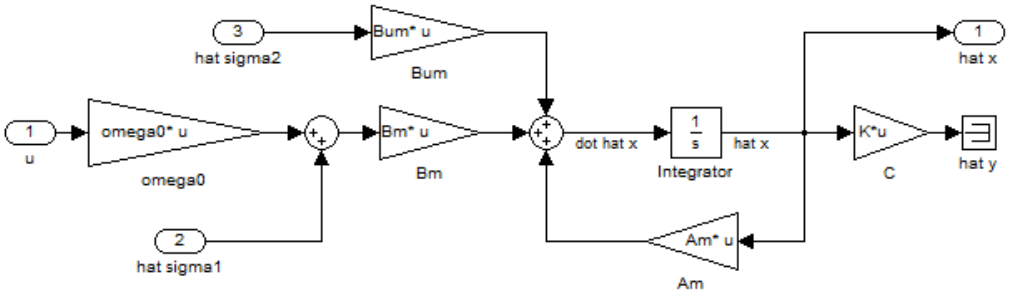


Figure 9.3: *The simulink implementation of the state estimator.*

9.1.2 Control Law

The control law implementation follows the theory as well. The filter, D, has been split into four different transfer functions, and the output is sampled at a rate of Ts. As with the state predictor, the matrix *omega0* could be omitted. The implementation can be seen in figure 9.4.

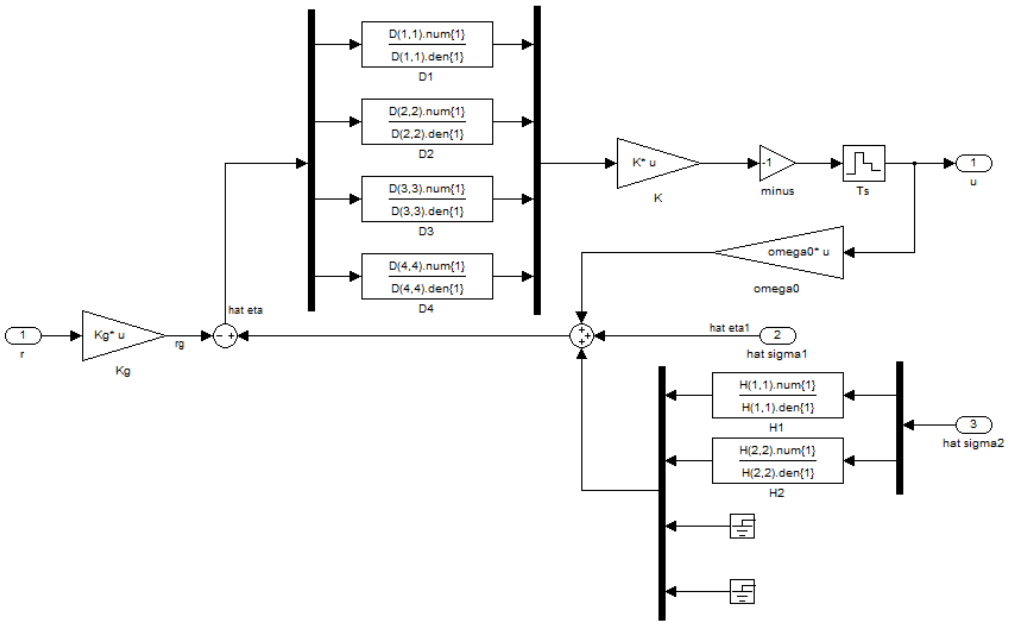


Figure 9.4: *The simulink implementation of the control law as described in.*

The filter, H, has been split into two identical transfer functions, and the zero-elements are represented as ground-signal blocks.

9.2. Test of the L1 Controller

In this section, the L1 adaptive controller will be tested against the nominal system in the presence of faults and model uncertainties. The section will discuss the importance of unmatched uncertainties, modeling approximations and different types of faults.

The Matlab and Simulink files used for the simulations in this section, are available in the "L1 Simulink Test" folder on the CD. Each simulation can be repeated simply by running the appropriate file, which will be specified for each test.

To compare the performance of the L1 controller to nominal performance, most plots in this section will contain the reference (black), the faultless system (green), faulty system (blue) and the faulty system with L1 (red), unless otherwise stated.

Due to the symmetry in the system, faults will only be applied to motor 1.

9.2.1 Maintaining Nominal Performance

To repeat the simulation, please run TEST_Nominal_performance.m

The goal of L1 control is to maintain the nominal performance specified by the desired system, in the spite of faults. The L1 performance is tested with multiplicative faults of different magnitude, that occurs at time 3. The results are shown in figure 9.5, figure 9.7 on page 80, figure 9.8 on page 81 and figure 9.9 on page 82. Only one simulation is performed, with steps on the roll, pitch, yaw rate and climb rate reference.

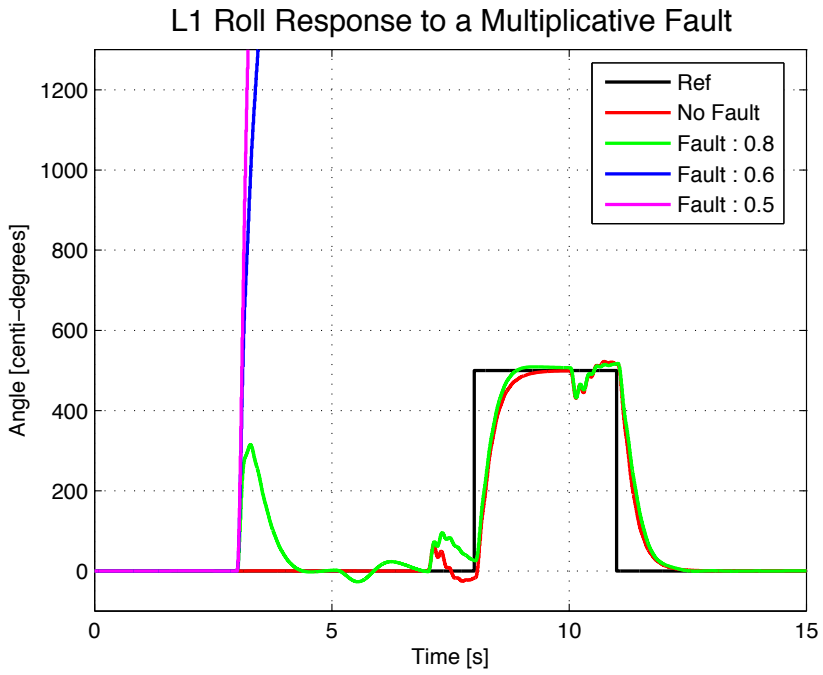


Figure 9.5: *Response of three L1 controllers at different sizes of multiplicative faults on motor 1 at time 3, along with a faultless L1 controller. A step of 5 degrees on the reference is performed from time 8 to 11.*

The roll axis is directly affected by the motor fault, and the magnitude affects the size of the error. The L1 controller is only able to compensate propperly for the smallest fault, and maintain nominal performance. The medium fault results in a huge steady state error (off picture) and the biggest fault renders the system unstable. The L1 controller is in fact able to surpress these faults and maintain nominal performance, however it is limited by the rate PID controllers. The parameters *RATE_RLL_IMAX* and *RATE_PIT_IMAX* specify the integrator limit in the rate controllers, and if the offset caused by the fault is sufficiently large, the controller is unable to compensate. By increasing the default limit on the integrator from 500 to 1400 the system with a medium fault reaches nominal performance, and the system with the biggest fault becomes stable, as seen in figure 9.6 on the facing page.

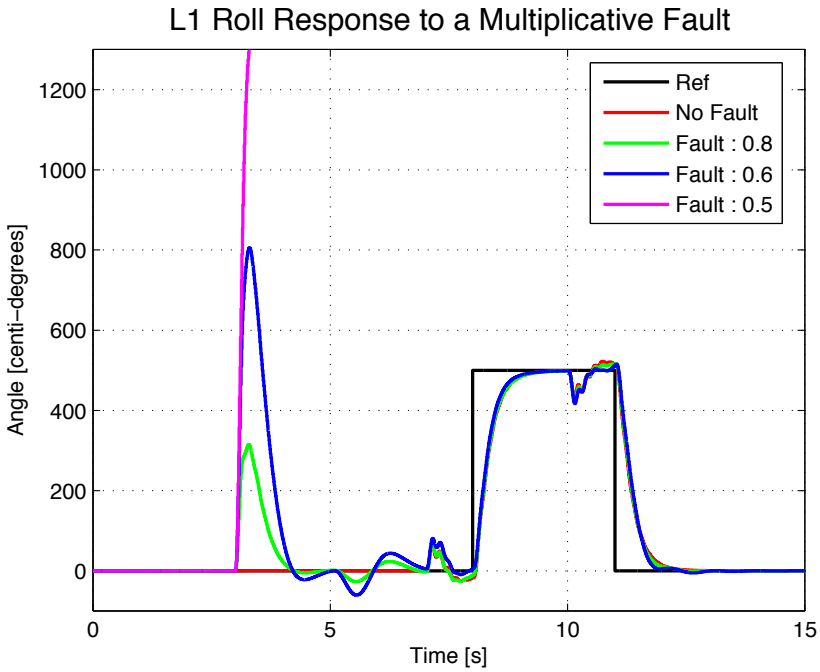


Figure 9.6: *A repetition of the simulation in figure 9.6, but with the integrator limits RATE_RLL_IMAX and RATE_PIT_IMAX increased from 500 to 1400.*

A further increase to 2200 would allow the system with the biggest fault to maintain nominal performance. The expansion of the integrator limits is examined in section 9.2.2 on page 82. The remaining figures in this test are simulated with the original limit of 500.

Notice the dip at time 10 in both 9.5 on the facing page and 9.6, due to the change in pitch and climb rate reference.

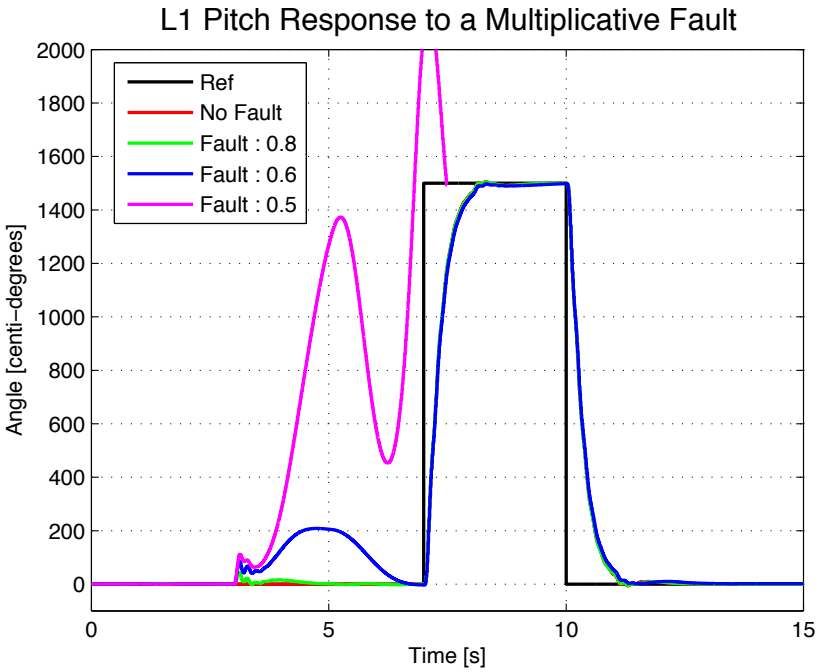


Figure 9.7: *Response of three L1 controllers at different sizes of multiplicative faults on motor 1 at time 3, along with a faultless L1 controller. A step of 15 degrees on the reference is performed from time 7 to 10.*

Since pitch is only indirectly affected by the fault, the resulting error is significantly smaller than for roll. The L1 controller is able to maintain nominal performance for all but the biggest fault, which becomes unstable in the roll state.

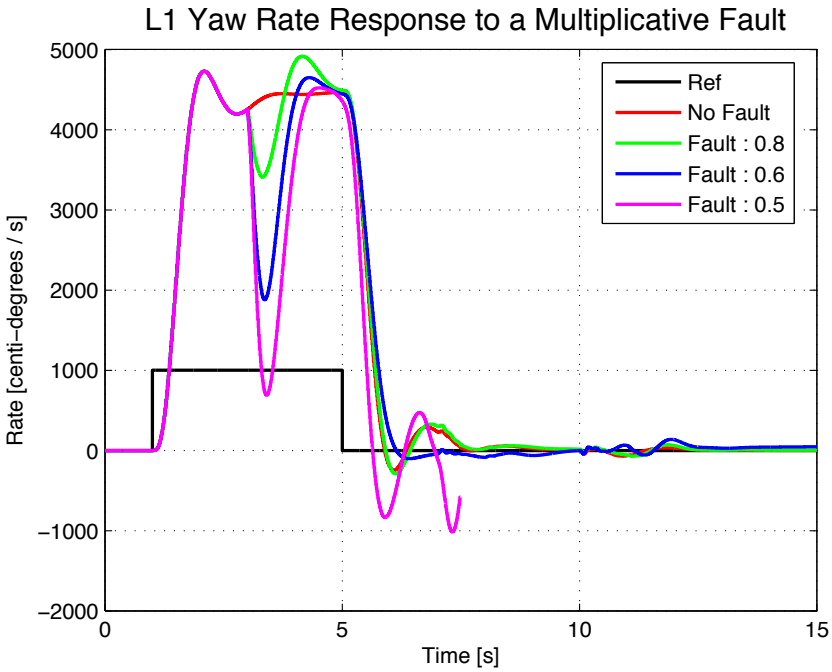


Figure 9.8: *Response of three L1 controllers at different sizes of multiplicative faults on motor 1 at time 3, along with a faultless L1 controller. A step of 10 degrees per second on the reference is performed from time 1 to 5.*

The yaw rate is more susceptible to the fault than pitch. Only the system with the small fault is able to maintain nominal performance, while the system with the medium fault has a minor degradation. Notice that the overshoot at time 6 is in fact reduced in the system with the medium fault, the goal of L1 is, however, to maintain the dynamics of the desired system, and thus the reduction of the overshoot is, ironically, not intended.

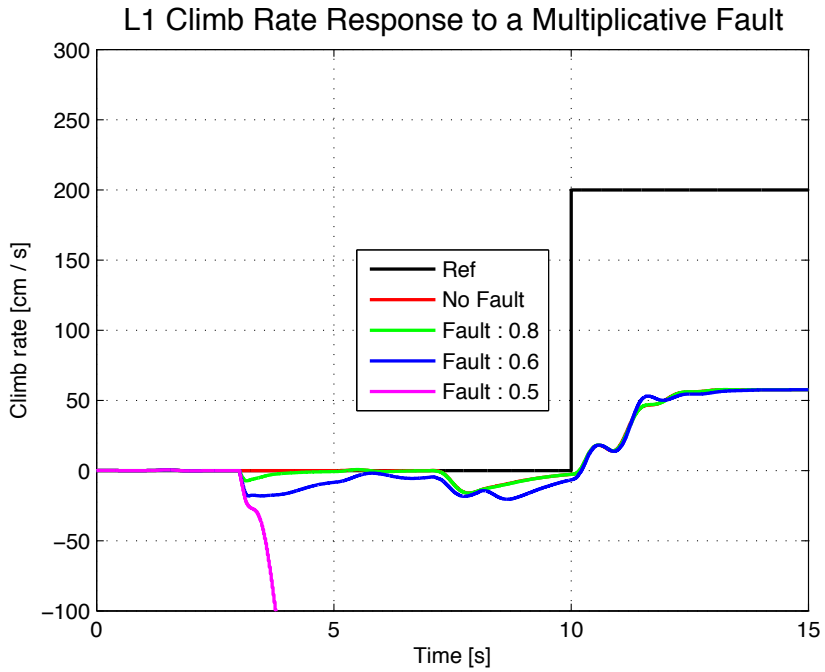


Figure 9.9: *Response of three L1 controllers at different sizes of multiplicative faults on motor 1 at time 3, along with a faultless L1 controller. A step of 200 on the reference is performed at time 10.*

Only the system with the biggest fault loses nominal performance, due to the unstable roll state. The systems with the small and medium faults tracks the nominal performance very good.

This tests verified that the L1 controller is able to maintain nominal performance in the presence of faults. It also showed that unmodeled dynamics in the APM controllers can influence the performancen, which will be examined in the next test.

9.2.2 Integrator Limit

To repeat the simulation, please run TEST_Bonus_I.m

The APM contains numerous methods that influence and shape the control signal that is send to each motor. The desired system dynamics does not take these unlinear dynamics into account, and the L1 controller may for that reason run into unforseen limitations, as seen in the previous test. For this reason, an exact model of the APM is important during testing, since it makes it easier to detect problems and improve performance.

An abrupt additive fault of -200 radians per second is added to the angular velocity of motor 1 at time 3. Figure 9.10 on the facing page shows the consequence of increasing the integrator limit for the roll rate controller, *RATE_RLL_IMAX*, by 100, 200 and 300, from its starting value of 500.

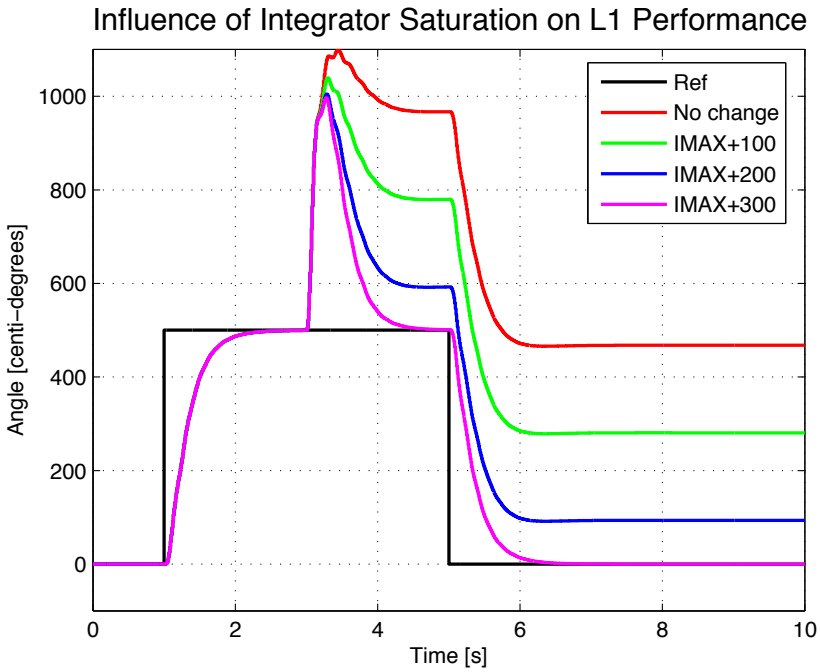


Figure 9.10: *Response of an L1 controller to an additive fault on motor 1 at time 3, with different values of the roll rate integrator limit, RATE_RLL_IMAX. The default value is 500.*

The shape of each response is very similar, although translated on the y-axis. An increase of the integrator limit improves tracking during a fault, but increases the magnitude of a potential integrator wind up. The "offset" a fault can cause in a system exposes limits that are not seen during nominal performance, and a reconfiguration of certain parameters using fault tolerant control, could be necessary in some cases.

In the following tests, the L1 controller will be compared to the nominal, L1-less system. The default integrator limit will be increased by up to 300, to allow the L1 controller to compensate propperly.

9.2.3 Additive Motor Fault

To repeat the simulation, please run TEST_Additive_fault.m

The motor faults in this section will affect the angular velocity of the motor. The calculation of applied moments and thrust is proportionally based on the angular velocity of the motors, so a fault which reduces the angular rate, can also be viewed as a fault which reduces the performance of the motor by other means, e.g. a broken propeller. Since these types of faults affect the actuators, they can be referred to as actuator faults.

The first test is an abrupt fault simulated by a step input of -150 rad/s on motor 1 at time 3. The roll reference is 500 centi-degrees from time 1 to 5.

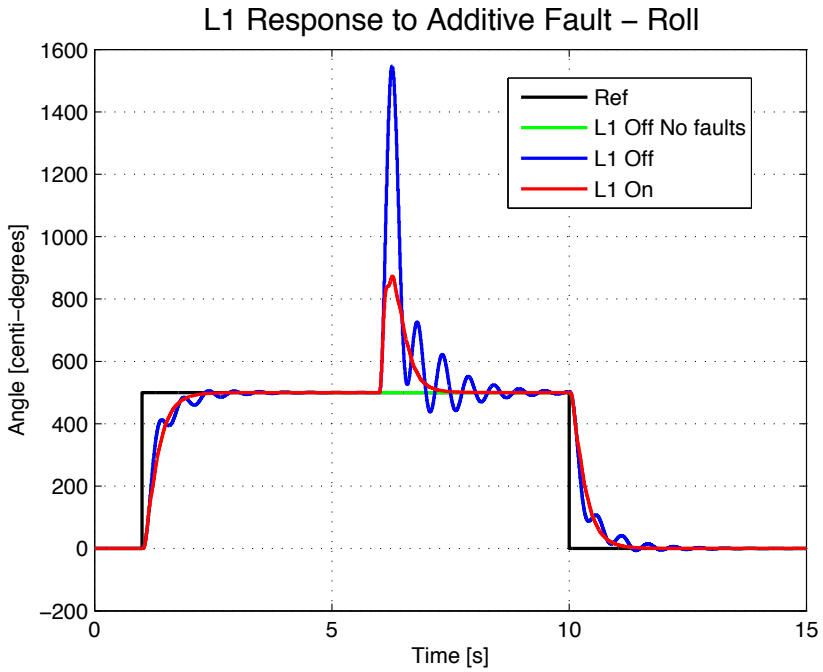


Figure 9.11: *Response of the faultless, faulty and faulty L1 system to a step on the roll reference, and an additive motor fault at time 3.*

The figure 9.11 shows the roll angle for the system with and without faults and with and without L1. The nominal system (green) and the faulty system (blue) perform identical up until the fault. The performance of the faulty system is immediately degraded, and oscillations of ± 1 degrees are introduced. The faulty system is still able to track the reference. The L1 controller is able to suppress the oscillations present in the faulty system, and is able to maintain nominal performance, comparing the steps before and after the fault have occurred. Notice also, that the L1 controller is able to reduce the initial impact of the fault by reducing the angle error to a maximum of 4 degrees, while the system without L1 get an error of 11 degrees.

The sudden dip in output from motor 1, affects the yaw rate, so the quadcopter starts to turn around the z-axis. The resulting yaw rate is shown on the figure 9.12 on the facing page:

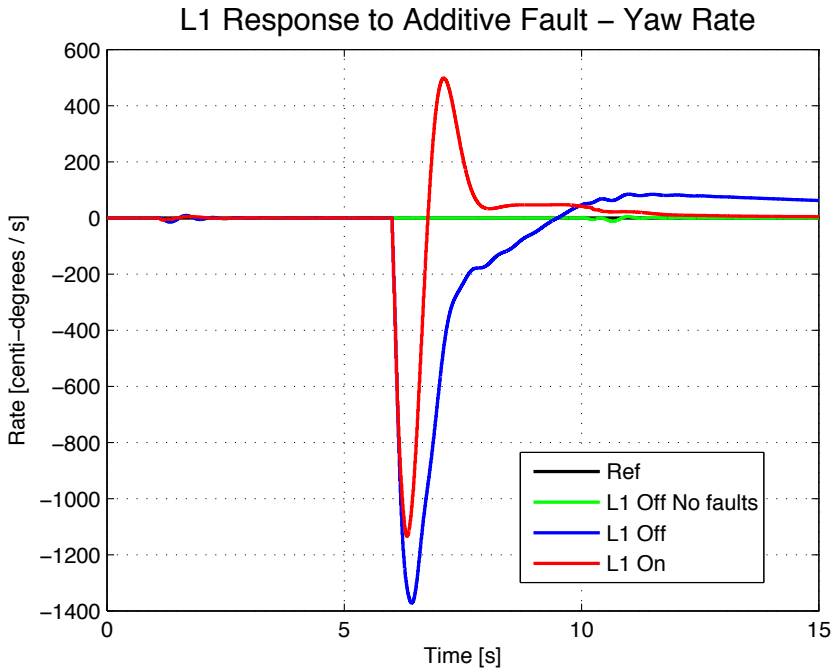


Figure 9.12: *Response of the faultless, faulty and faulty L1 system to a step on the roll reference, and an additive motor fault at time 3.*

The L1 controller compensates for the fault quicker than the nominal system. The initial spike is reduced from -14 degrees to -11 degrees. The L1 system settles roughly three seconds faster than the nominal system, and its steady state error is less that 0.1 degrees. The nominal system is much slower, and has a larger steady state error, relative to the L1. Notice that the L1 controller overshoots the yaw rate, this is due to the design of the L1 controller.

Since the yaw rate dynamics have been approximated as a first order transfer function, the dynamics which leads to oscillations are unaccounted form, and the overshoot is a consequence of this. The same problem was present in the roll and pitch models, until unmatched uncertainties were added. The addition of unmatched uncertainties, increases the likeness between the state predictor and the nominal system, and the estimation error is thus reduced. The addition of unmatched uncertainties requires that the added states are measured, and increases the complexity of the system, which in turn increases the computation load. The performance of the yaw rate controller is deemed adequate for this project, since it is not as critical to stability as pitch and roll.

When the quadcopter rotates around the z-axis, the pitch angle and roll angle will start to switch places. This, combined with the effort to cancel the yaw rotation, leads to disturbances in pitch, even though roll and pitch are directly decoupled. The effect of this, can be seen in figure 9.13 on the next page.

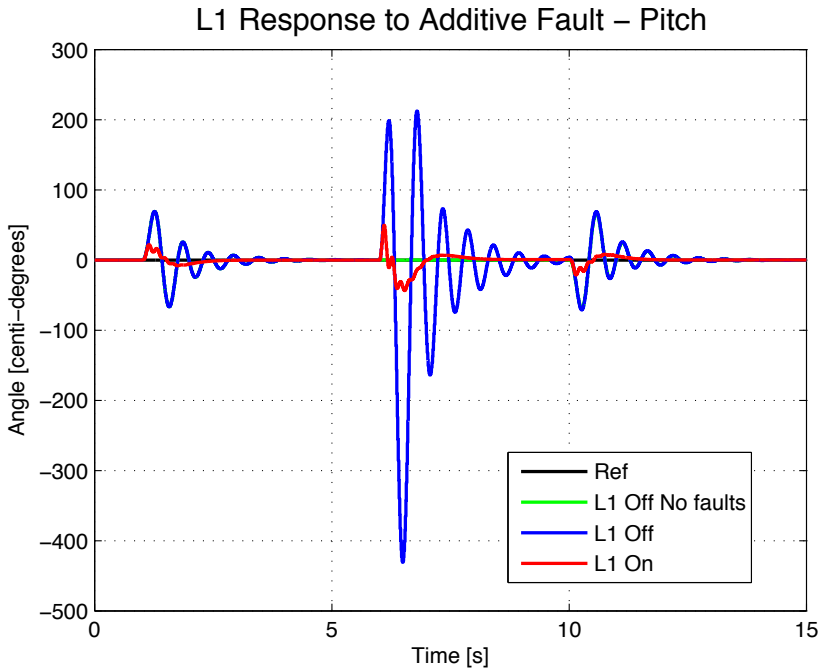


Figure 9.13: *Response of the faultless, faulty and faulty L1 system to a step on the roll reference, and an additive motor fault at time 3.*

Notice that the step on roll at time 1 affects the pitch angle in a greater degree for the nominal system that the L1 system. The oscillations brought on by the fault at time 3 is greatly reduced as well. Since the adaption rate is sufficiently fast, the controller is able to compensate immediately, even though the cross couplings have not been specified by the desired system dynamics.

The state estimation error for roll and roll rate is shown in figure 9.14 on the facing page.

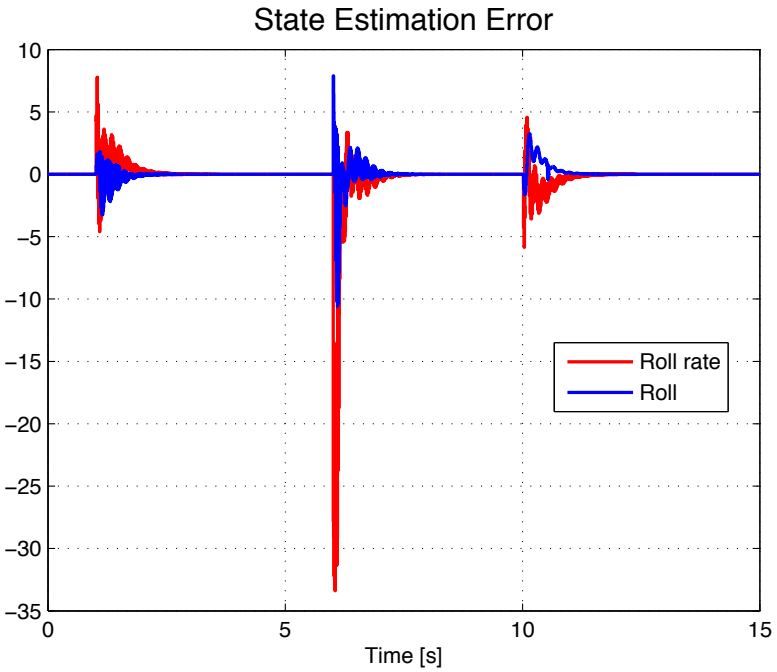


Figure 9.14: *The estimation error of the roll and roll rate states. Readers on an electronic device is adviced to zoom in.*

The step on the roll reference at 1 and 5 excites a response, which is a consequence of using an approximation of the nominal system as a desired system for the L1 controller. The difference between the systems result in an imprecise state predictor and in turn an estimation error. The modeling error is calculated in sigma1, which affects the control signal until the estimation error is zero.

The response when the fault occurs at 3 differs from the response when the reference was changed, in that the estimation error is much greater, due to the sudden change in the dynamics.

9.2.4 Sensor Fault

To repeat the simulation, please run TEST_Additive_sensor_fault.m

While the L1 controller is able to compensate for an actuator fault, it is unable to compensate for sensor faults. The previous simulation is repeated, but the motor fault has been replaced by an additive sensor fault of -500 centi-degrees at time 3. The result, available in figure 9.15 on the next page, show that the L1 controller, along with the L1-less system, compensates for the perceived angle error, by shifting the roll angle 5 degrees. The tracking of the roll reference is thus permanently offset by 5 degrees.

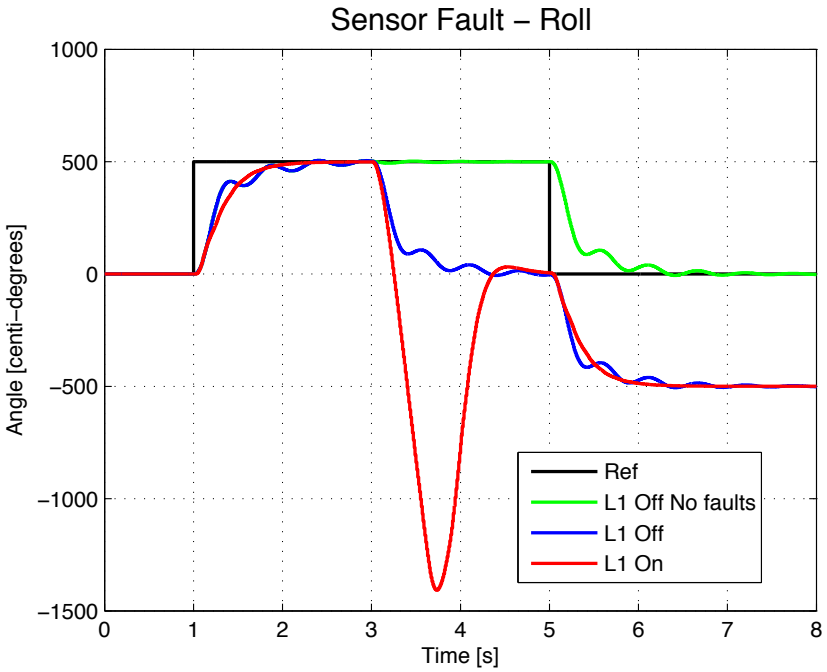


Figure 9.15: *Response of the faultless, faulty and faulty L1 system to a step on the roll reference, and an additive sensor fault at time 3.*

The sudden shift of 5 degrees result in an estimation error, and this result in a huge deflection in sigma, as seen in figure 9.16. This, in turn, affects the control signal resulting in the huge overcompensation.

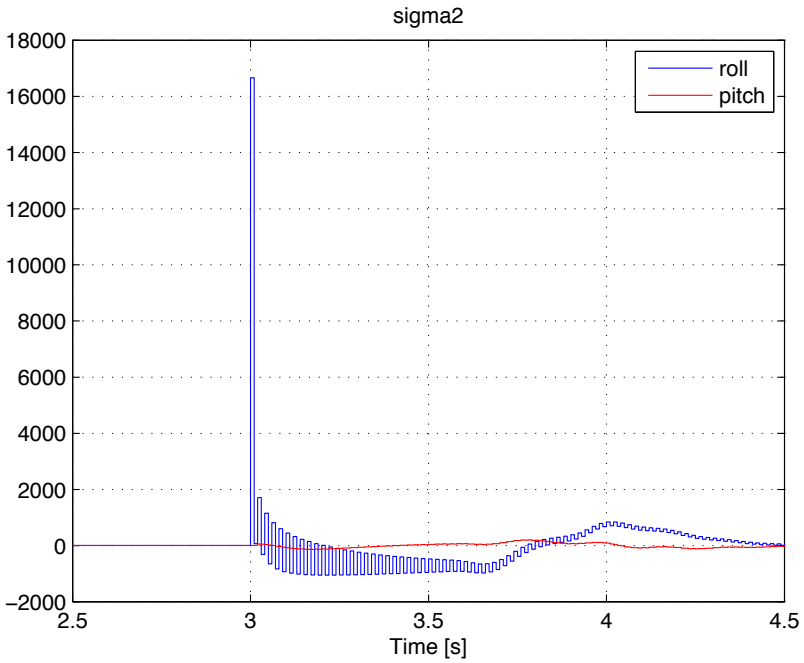


Figure 9.16: *The response of sigma2 to an additive sensor fault at time 3.*

The solution to sensor faults could be the addition of fault tolerant control, which consists of detecting the faults, based on knowledge of the system, and subsequently changing the control strategy to one that does not utilize the faulty parts of the system.

9.2.5 Offset Payload

To repeat the simulation, please run TEST_Additive_moment.m

The addition of a payload placed at the end of one of the arms would add an applied moment of:

$$\tau = l \cdot F = l \cdot g \cdot m \tag{9.1}$$

where l is the distance from center of the quadcopter to the location of the payload, g is the gravitational acceleration and m is the mass of the payload.

For simulation purposes the changes to mass and inertia of the quadcopter, because of the payload, is not taken into account.

A payload of 100g is added at the location of motor 1 at time 1 and removed again at time 5. This will mostly affect the applied moment around the x-axis, which corresponds to the roll angle, plotted in figure 9.17.

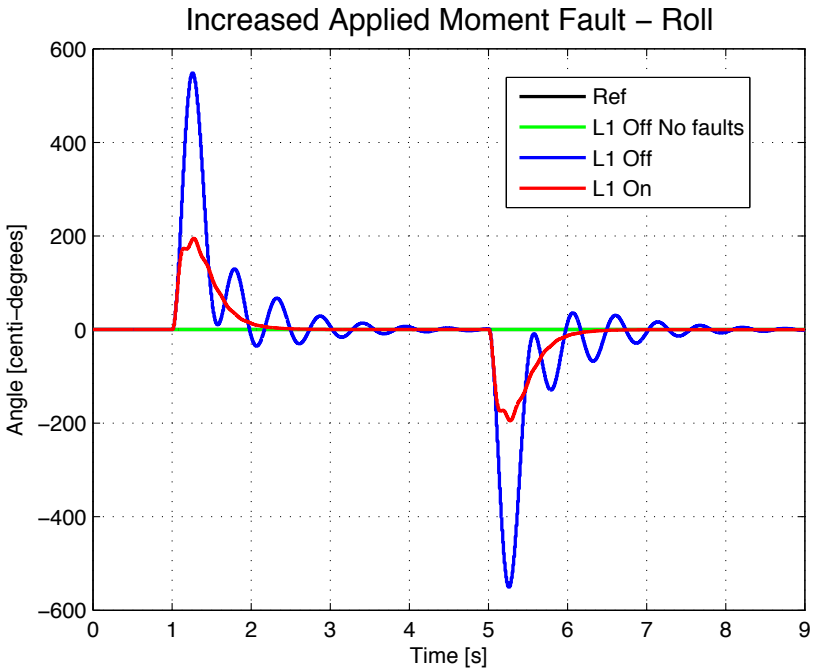


Figure 9.17: *Response of the faultless, faulty and faulty L1 system to a step on the roll reference, and a payload of 100g added to the location of motor 1 from time 1 to 5.*

The uneven weight distribution excites oscillations in the nominal system, while these are suppressed in the L1 system. Both systems are able to track the reference, however only the L1 system maintains nominal performance. As it were the case in the previous tests, the L1 is able to suppress the fault faster than the system without L1.

9.2.6 Inertia from Payload

To repeat the simulation, please run TEST_Wrong_inertia.m

Instead of a payload mounted away from the center of gravity, like in the previous example, this example deals with the consequence of an increased inertia and mass due to a payload. A payload is simulated by increasing the mass of the quadcopter by 40% and the inertia matrix by 50%. A step on the roll reference of 5 degrees is performed at time 4 and a step down to zero degrees is performed at time 7. Figure 9.18 shows the roll angle response.

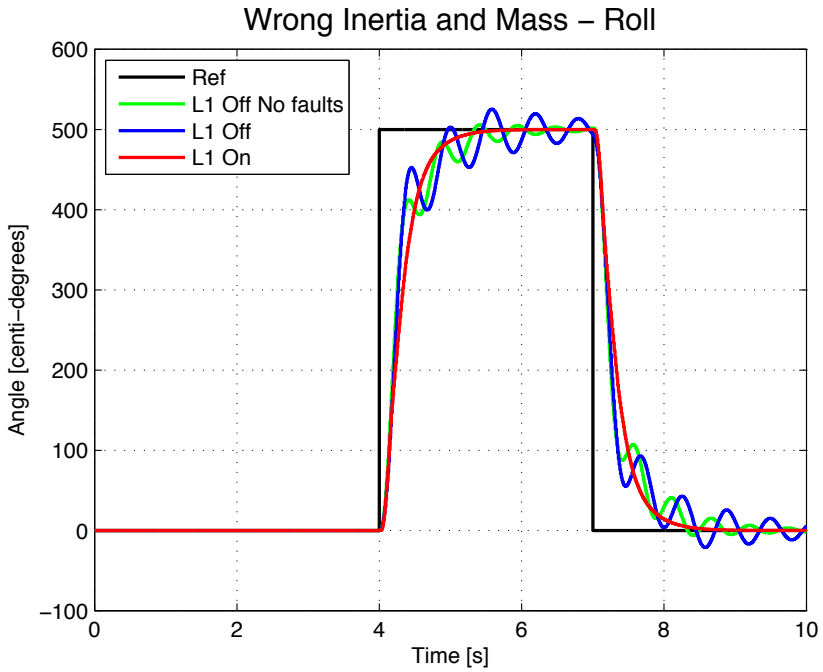


Figure 9.18: *Response of the faultless, faulty and faulty L1 system to a step on the roll reference, and an increase in mass and inertia of 40% and 50%, respectively.*

Notice that while the L1 system maintains nominal performance, the system without L1 oscillates. The increased weight does not affect the attitude, only the climb rate. Figure 9.19 on the next page shows the consequence of the increase in mass. The initial conditions of the model are not sufficient to cancel the gravitational force, and thus the quadcopter drops. A step on the reference of 200 is performed at time 2. The climb rate has a steady state error, which is intended, as and shown in section 6.2 on page 48.

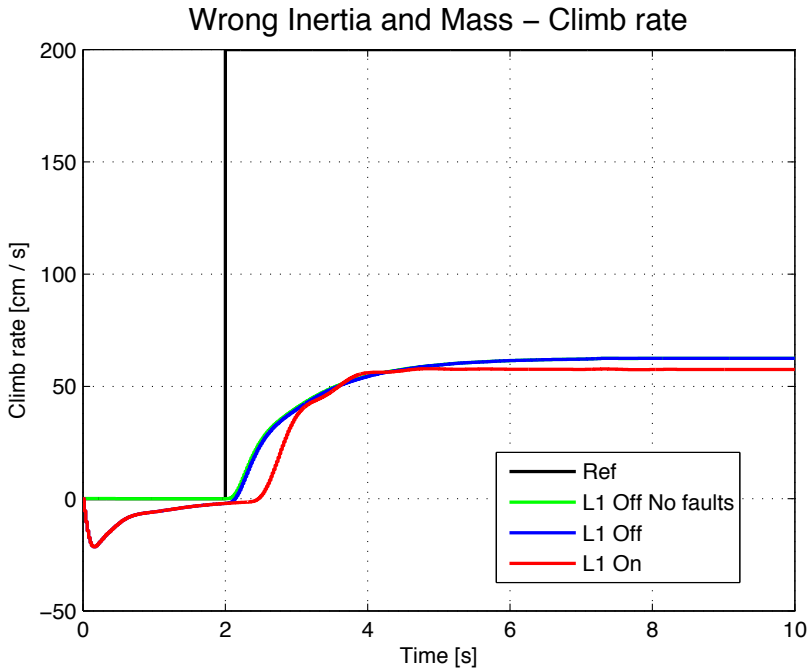


Figure 9.19: *The added mass of 40%, does hardly affect either the nominal or L1 system*

The steady state error of the L1 system is slightly larger than for the nominal system. This can be adjusted by increasing the gain of the prefilter, K_g, for the climb rate reference. The response of both the L1 and L1-less system are very similar. The climb rate is apparently unaffected by the increase in mass, and even the increasing oscillations in pitch, roll and yaw starting from time 4, are decoupled from the climb rate. This would change, however, if the motors were reaching their limits, in which case the mixer would scale the output.

The angular velocity of the motors without L1 is shown in figure 9.20 on the next page and with L1 in figure 9.21 on the following page.

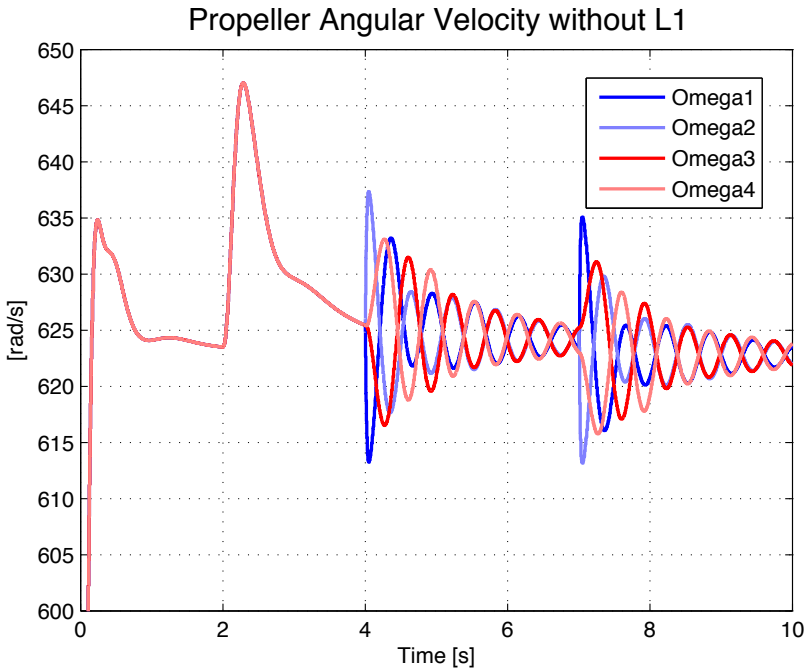


Figure 9.20: *Angular velocity of the motors during a step on the climb rate and roll rate reference. The red and blue colors represent the pitch-pair and roll-pair, respectively. Mass and inertia has been increased, and L1 is off.*

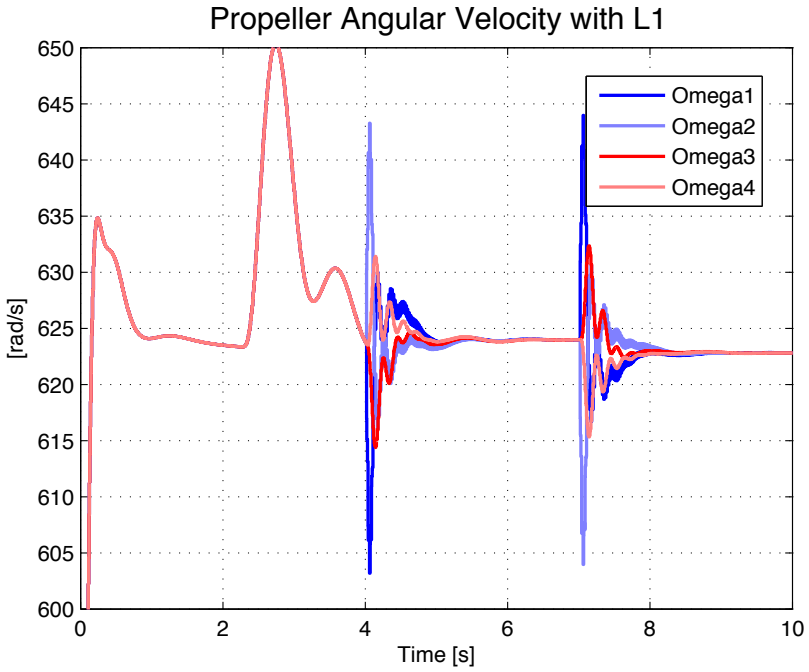


Figure 9.21: *Angular velocity of the motors during a step on the climb rate and roll rate reference. The red and blue colors represent the pitch-pair and roll-pair, respectively. Mass and inertia has been increased, and L1 has been activated.*

The L1 system responds more aggressively to the change in roll reference at time 4

and 7 than the nominal system, however it also subsides quicker. The output of all motors are equal up until the step at 4. This is due to the fact that the simulated quadcopter is completely symmetrical, and there are no disturbances. The upper limit of the motors is 1076 radians per second, which is equal to 171 revolutions per second.

9.2.7 Motor Failure

To repeat the simulation, please run TEST_Complete_failure.m

In the event that a motor should stop functioning completely, the L1 controller will be unable to compensate, and the quadcopter will experience a failure. This is evident in figure 9.22, where the output of motor 1 is set to zero from time 3 an onwards. The simulation is stopped when the roll angle exceeds 120 degrees.

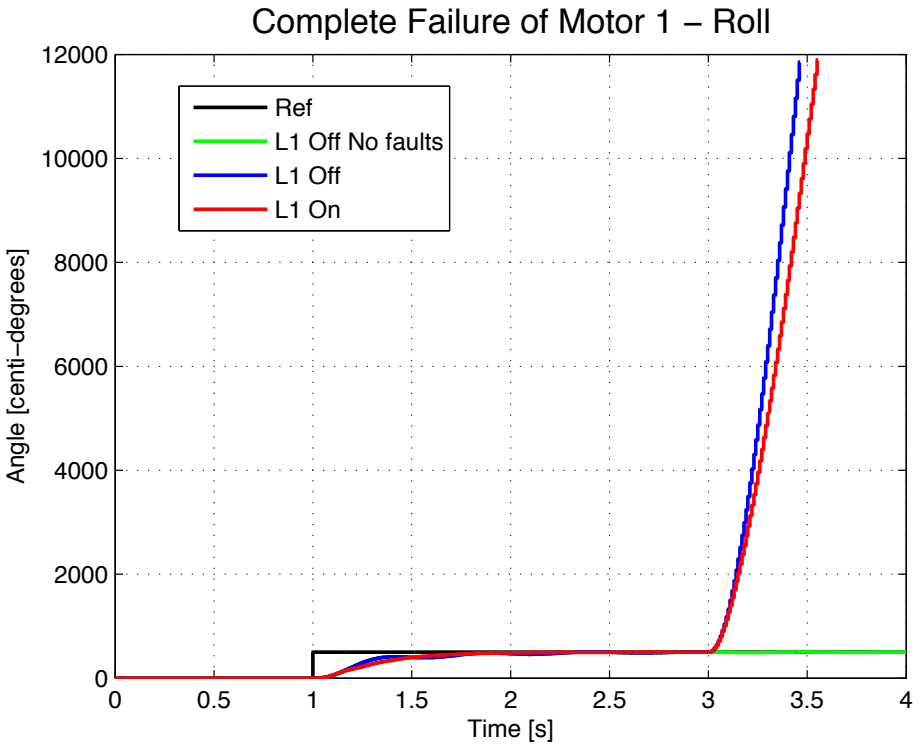


Figure 9.22: *Response of the faultless, faulty and faulty L1 system to a step on the roll reference, and a failure of motor 1 at time 3.*

As with the sensor fault, the L1 controller is unable to compensate, and fault tolerant control must be introduced to avoid failure of the quadcopter.

9.2.8 Multiplicative Motor Fault

To repeat the simulation, please run TEST_Multiplicative_fault.m

The L1 controller is pushed to its limit, by introducing a multiplicative fault of 0.41 on motor 1. The integrator limit is increased by 2200 to allow the controllers room to compensate. The result is shown in figure 9.23 on the following page.

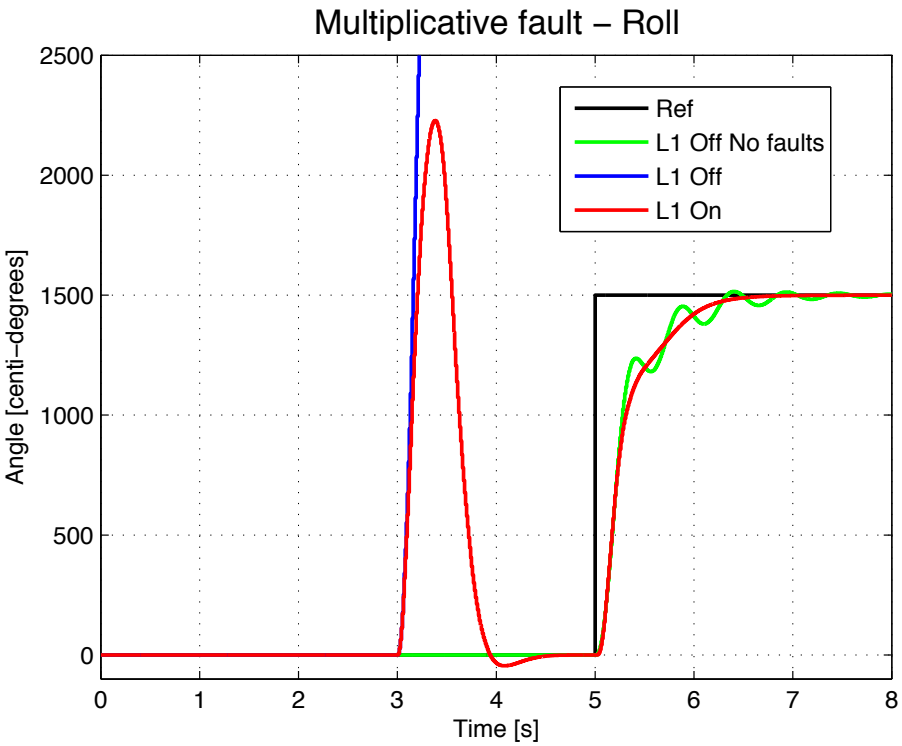


Figure 9.23: *Output of motor 1 reduced to 41% of nominal performance at time 3.*

The L1 controller manages to control pitch and yaw rate, however climb rate suffers. The quadcopter is unable to maintain lift, primarily due to the mixer. When the roll, pitch and yaw rate must remain zero, all motors must deliver the same output, which limits the thrust from each motor to that of the faulty one. A control strategy disregarding yaw rate could improve the climb rate, but this would require a reconfiguration using e.g. fault tolerant control.

CHAPTER 10

Implemention in the APM Software

10.1. L1 APM Implementation

In this chapter, the L1 adaptive controller from section 9.1 on page 75 is implemented in the APM software. A major concern is whether or not the APM is capable of running an L1 controller in addition to the basic software, so this will be tested in the last part of the chapter 10.1.3 on page 99.

10.1.1 Adding Custom Code

The DIY Drones community encourages user to add custom code the APM software. In order to prevent software updates from breaking existing user code, they recommend that as much custom code as possible is placed in the *UserCode.ino* file. UserCode.ino consists of *userhook_init*, which is run once at startup, and five different methods called at different rates during flight[1]. It is also recommended that *UserVariables.h* is used to declare and initialize variables used in the code. Finally *APM_Config.h* must include a definition, so that the user methods are run. The definitions in 10.1 on the next page are commented out by default, and must be uncommented to function.

[1] 1Hz, 3.3Hz, 10Hz, 50Hz and 100Hz

```
// User Hooks : For User Developed code that you wish to run
// Put your variable definitions into the UserVariables.h file (or another file
     name and then change the #define below).
#define USERHOOK_VARIABLES "UserVariables.h"
// Put your custom code into the UserCode.pde with function names matching
    those listed below and ensure the appropriate #define below is uncommented
    below
#define USERHOOK_INIT userhook_init();                      // for code to be
    run once at startup
#define USERHOOK_FASTLOOP userhook_FastLoop();            // for code to be run
     at 100hz
#define USERHOOK_50HZLOOP userhook_50Hz();                  // for code to be
    run at 50hz
#define USERHOOK_MEDIUMLOOP userhook_MediumLoop();        // for code to be run
     at 10hz
#define USERHOOK_SLOWLOOP userhook_SlowLoop();            // for code to be run
     at 3.3hz
#define USERHOOK_SUPERSLOWLOOP userhook_SuperSlowLoop();  // for code to be run
     at 1hz
```

Figure 10.1: *Snippet of APM_Config.h*

In the *fast_loop* method in *ArduCopter.ino*, the following lines are evaluated during compiling:

```
#ifdef USERHOOK_FASTLOOP
    USERHOOK_FASTLOOP
#endif
```

Figure 10.2: *Snippet the fast_loop method in ArduCopter.ino*

If *USERHOOK_FASTLOOP* was never defined, *userhook_FastLoop* will never be called.

The state estimator, adaption law and control law can all be run from within the user code, it is necessary, however, to modify *ArduCopter.ino*, to replace the roll, pitch and yaw rate reference from the RC transmitter with the control signal from the L1 controller. To ensure safety during tests, the pilot should be able to disengage the L1 controller at any time. This is done using the unused RC channel 7. The APM software use channels 6, 7 and 8 as auxiliary inputs and they are read at the same rate as the rest of the channels by default. Their PWM signals are converted to a number between 0 and 1000, so a threshold of 500 was chosen to turn the L1 on or off. The code in figure 10.3 on the facing page, figure 10.4 on the next page and figure 10.5 on the facing page shows all the modifications made to *ArduCopter.ino*.

```
    case ROLL_PITCH_STABLE:
        // apply SIMPLE mode transform
        if(ap.simple_mode && ap_system.new_radio_frame) {
            update_simple_mode();
        }

        control_roll            = g.rc_1.control_in;
        control_pitch           = g.rc_2.control_in;

        /// CUSTOM CODE - START
        if (g.rc_7.control_in > 500) { // L1 ON
            get_stabilize_roll(L1_u_roll);
            get_stabilize_pitch(L1_u_pitch);
        }
        else // L1 OFF
        {
            get_stabilize_roll(control_roll);
            get_stabilize_pitch(control_pitch);
        }
        /// CUSTOM CODE - END

        break;
```

Figure 10.3: *From the update_roll_pitch_mode method in ArduCopter.ino*

```
case YAW_HOLD:
        // heading hold at heading held in nav_yaw but allow input from pilot
        /// CUSTOM CODE - START
        if (g.rc_7.control_in > 500) { // L1 ON
            get_yaw_rate_stabilized_ef(L1_u_yaw);
        }
        else // L1 OFF
        {
            get_yaw_rate_stabilized_ef(g.rc_4.control_in);
        }
        /// CUSTOM CODE - END
        break;
```

Figure 10.4: *From the update_yaw_mode method in ArduCopter.ino*

```
    case THROTTLE_HOLD:
        if(ap.auto_armed) {
            // alt hold plus pilot input of climb rate
            /// CUSTOM CODE - START
            if (g.rc_7.control_in > 500) { // L1 ON
                pilot_climb_rate = get_pilot_desired_climb_rate(L1_u_throttle
                    +500);
            }
            else // L1 OFF
            {
                pilot_climb_rate = get_pilot_desired_climb_rate(g.rc_3.
                    control_in);
            }
            /// CUSTOM CODE - END
            ...
        break;
```

Figure 10.5: *From the update_throttle_mode method in ArduCopter.ino*

Notice that when the channel 7 is less than or equal to 500, the L1 is bypassed, and the quadcopter is directly controller by the RC unit again.

Due to the concerns that user code might slow down the APM to a degree that stability is lost, a similar check was made in *UserCode.ino* to skip the L1 calculations, if the L1 controller is disengaged.

10.1.2 Writing the Code

The L1 controller was based on a desired system with many decoupled states, and as a result many matrices in the final controller contained numerous zero elements. In order to save computational power, the matrix computations needed for the L1 controller were hard coded to omit any zero-element multiplication.

A way to increase the execution speed of the code is to decrease the accuracy of the computation by rounding the coefficients. This trade-off turned out to be needless as a tuning parameter, since the computation ran, even when defining all parameters as 32-bit floats, which will be further explained in the next section 10.1.3 on the next page.

The state estimator from section 9.1.1 on page 76 is continuous, so it is converted to a discrete state space system, using the Matlab command *c2d* with a sample time of 100Hz. The same was done for the low-pass filter D and the transfer function H.

The implementation can be visualized with the flowchart in figure 10.6.

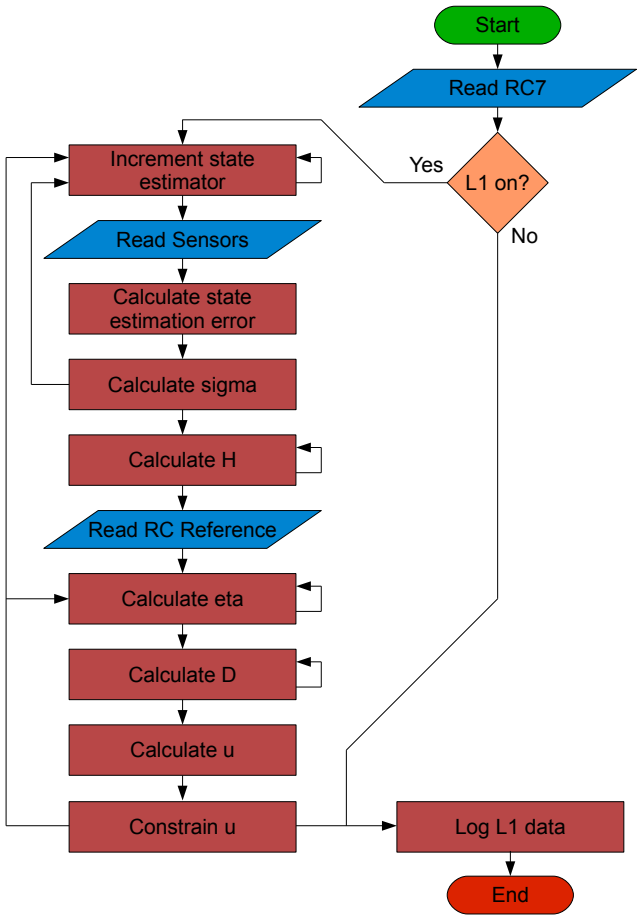


Figure 10.6: *Flowchart of the L1 implementation in the userhook_FastLoop method. The arrows pointing backwards in the system represents that the value from the previous run was used.*

The state estimator, eta and D block stores previous values for future calculations. Technically eta does not depend on its previous values, but the calculation of the output of D relies on the previous inputs, since it is a discrete transfer function.

Initial flight tests showed that the filter coefficients used in the simulations should be re-tuned to work with the quadcopter, so live tuning where implemented. RC channel 8 was used to control the value of K for roll and pitch by adding the code in 10.7 to the beginning of the L1 implementation.

```
L1_K_11 = (float)g.rc_8.control_in/50.0;
L1_K_22 = L1_K_11;
```

Figure 10.7: *Live tuning of K from UserCode.ino*

By dividing the input with 50, K would assume values between 0 and 20. Channel 8 was controlled by a turning dial on the RC transmitter.

In order to verify the performance of the L1 controller, various parameters are logged at the end of the user code. The sample rate and number of parameters that are logged are a trade-off, since the writing speed of the dataflash is limited. This is discussed in the stress test 10.1.3.

The code in its entirety is available from the CD in the folder "APM code".

10.1.3 Stress Test

The addition of L1 control will put an extra load on the APM, so it is important to verify that performance does not suffer. The onboard dataflash is capable of logging a number of parameters, including the so called *performance monitor*, or PM for short[2]. The PM run at 0.1Hz and contains the following three parameters that are important in determining the load on the APM:

NLon The number of main loops that took 5% longer than the 10ms they should, since the last PM was logged.

NLoop The total number of loops since the last PM was logged. This should normally be 1000.

MaxT The maximum time in micro seconds any loop took to complete since the last PM was logged. During initialization this value may exceed 6.000.000.

According to DIY Drones, the ratio between NLon and NLoop should never exceed 15%.

The quadcopter was loaded with the new L1 code, and using RC channel 7, the L1 controller was toggled on and off, without applying the necessary amount of throttle to initiate take-off. It was important that the duration of each state exceeded the sampling time of the PM. The results are available in table 10.1 on the following page.

The PM log proved that the inclusion of L1 computations impacts the performance of the APM. The APM is capable of running the L1 controller in addition to the rest of the code, but there is a severe increase in the number of long running loops. The slowest loop at each entry is increased by 18% when L1 is turned on. The APM is still able to performs 1000 loops per 10 seconds, and the percentage of long running loops is not close to 15%.

From the data available, it can be concluded that the APM is capable of running an L1 adaptive controller at 100Hz. Although the code runs noticeably slower with,

[2] A detailed description of all standard logs are available at http://copter.ardupilot.com/wiki/downloading-and-analyzing-data-logs-in-mission-planner

L1	NLon	NLoop	MaxT [ms]	% slow running
Off	1	1000	10508	0.1
Off	0	1000	10500	0.0
Off	0	1000	10496	0.0
Off	1	1000	10504	0.1
On	59	1001	13012	5.9
On	48	1000	13580	4.8
On	60	1006	11672	6.0
On	47	1000	11392	4.7

Table 10.1: *Performance measurements from the PM log during a L1 stress test of the APM. The amount of slow running loops should never exceed 15%.*

than without L1, performance does not suffer. It is evident, however, that the current implementation brings the APM close to its limits, and a more sophisticated controller may cause problems.

While the results suggested that the APM is capable of running the L1 code without a performance drop, the sampling rate of the attitude log was very inconsistent. Since the L1 logged its performance at 100Hz, and the attitude was logged at 50Hz, it was suspected that the dataflash was the bottleneck. By reducing the sampling rate of the L1 log to 50Hz and the attitude to 10Hz, the dataflash performed without degradation. In summary, the APM is capable of running our L1 controller at 100Hz.

Part IV

Assessment & Conclusion

CHAPTER 11

Acceptance Test

In section 10.1 on page 95 the L1 controller has been implemented in the APM software, a stress test of the system has been made in section 10.1.3 on page 99, showing the APM is capable of running the L1 code.

In this chapter, the flight tests have been made with the and without the L1 controller, so the two systems are compared. *Please note that these flight tests have been made with an L1 implementation with a bug*, this bug has been fixed but there was no time to testfly the new software, the new software is included on the CD. But even with this software bug, the L1 shows better performance then the nominal system. The test that proved easiest to do ad provided the best data was balancing the quadcopter on a box, show on figure 11.1

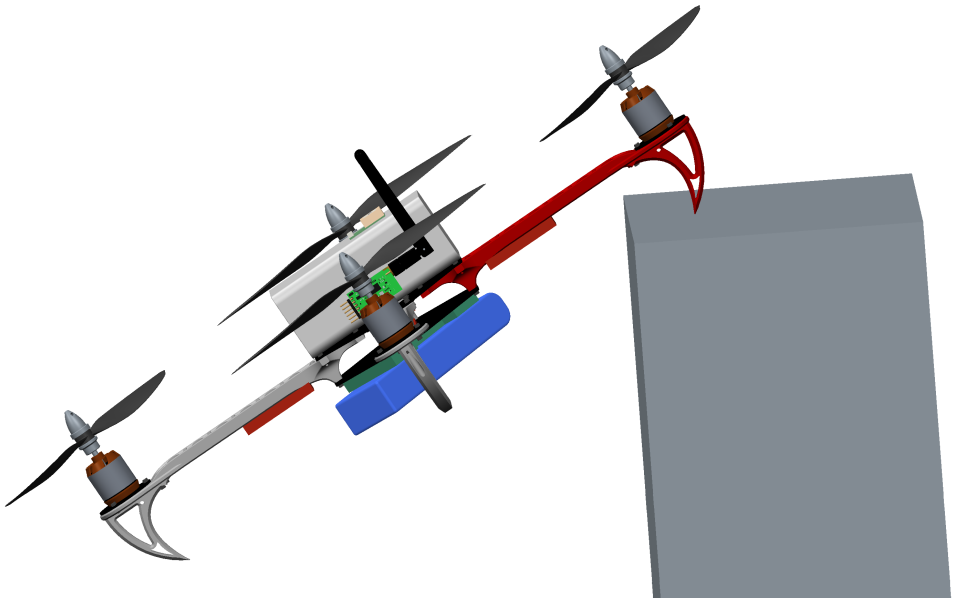


Figure 11.1: *The quadcopter on a box, and an angle to test the L1 controller.*

The flight tests had to be done indoor, due to bad weather, so there was not sufficient room to do steps on the quadcopters angle, and with no position tracking for indoor use ordinary flight tests proved difficult.

The results can be seen on figure 11.2, figure 11.3 on the next page and figure 11.4 on page 106.

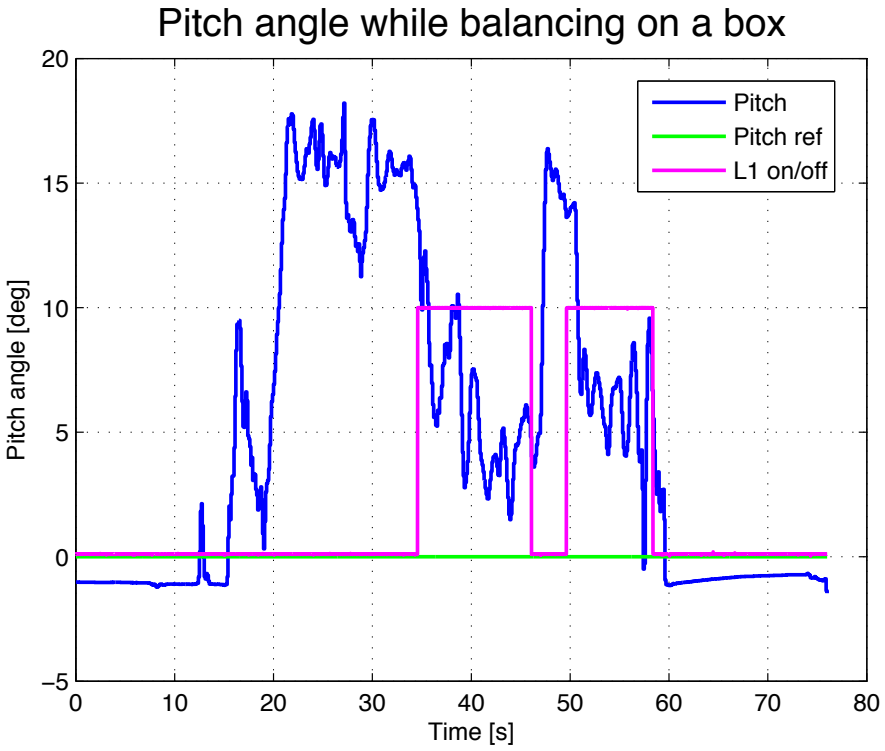


Figure 11.2: *The quadcopter on a box, and an angle to test the L1 controller.*

Figure 11.2 shows that that when the L1 is turn off the system has a pitch angle of about 17 degrees, but the when L1 is turned on (the magenta line, High=on, Low=off), the system goes to a pitch angle of 5 degrees, then when L1 is turned off the system goes back to its old pitch angle again.

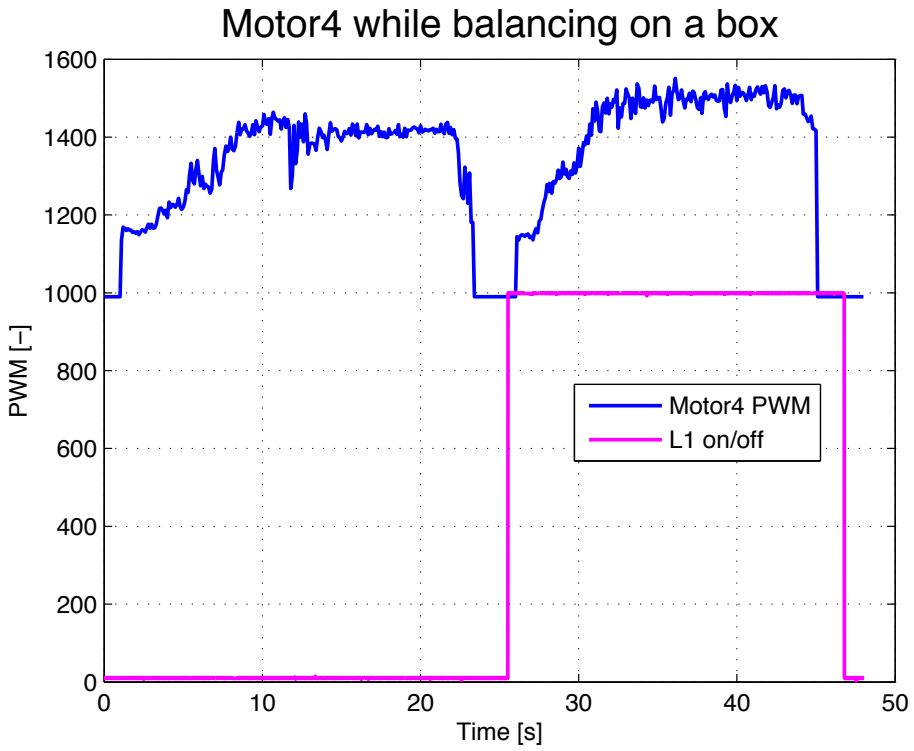


Figure 11.3: *The quadcopter on a box, and an angle to test the L1 controller.*

Figure 11.3 shows the same test, but with a lower throttle setting. This affects the systems ability to obtain a pitch angle of zero, but even with this low throttle setting the L1 controller can change the reference signal to try and compensate. Note that the pitch reference from the pilot is zero through out the test.

The last figure figure 11.4 on the next page, Shows the L1 controller changing the PWM signal sent to motor 4 (the rear motor in the pitch direction), to compensate for the angle error shown on figure 11.3

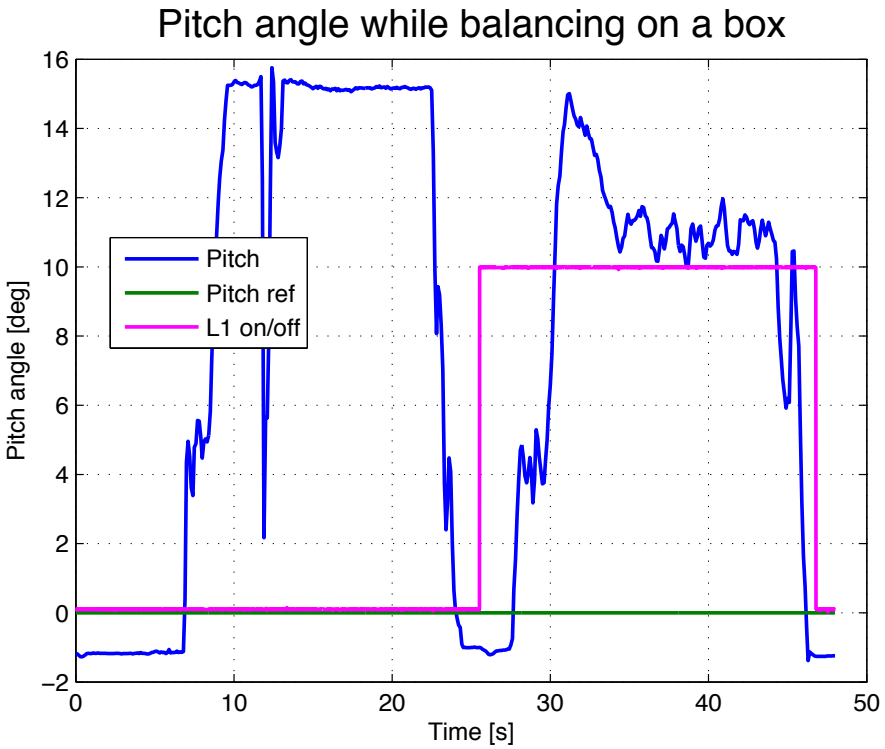


Figure 11.4: *The quadcopter on a box, and an angle to test the L1 controller.*

There is some oscillations in the L1 respond, it is expected that with proper tuning of the L1 these can be removed. These tests shows that the L1 controller works on the APM, there is some room for improvements, better tuning would help greatly on the responses.

CHAPTER 12

Conclusion

A quadcopter was build from scratch with the ArduPilot Mega 2.0 as an autopilot. A vibration dampening mount was 3D-printed, installed and verified to improve performance. Using manual tuning the quadcopter was able to maintain horizontal flight, and its pitch, roll, yaw rate and climb rate could be manipulated with an RC controller.

A theoretical model of the quadcopter was constructed, and testing was done on the quadcopter, to obtain the necessary parameters for a mathematical model. The quadcopter was drawn in a 3D CAD program, which was used to calculate the inertia. This knowledge was used to construct a simulink model of the quadcopter.

The ArduCopter code was deciphered in order to construct an accurate simulink model of the autopilot system. The APM model and the quadcopter model were put together, and tested using flight data obtained from the quadcopter. The tests verified both the APM and quadcopter model as being an adequate approximation of the real system.

The L1 adaptive state feedback controller was introduced, as a method of compensating for a fault in the quadcopter. An approximation of the quadcopter dynamics was made in simulink, and this model was used as the desired system for the L1 controller. The adaption rate was set at 100Hz, matching the fastest user loop on the APM, and the tuning was based on this. It was necessary to take unmatched uncertainties in the form of roll rate and pitch rate into account to get satisfactory performance.

The L1 controller was implemented in simulink, and tested against the nominal system in the event of different types of fault.

The L1 was able to surpress cross couplings and compensate for faults much better than the nominal system. It was shown that limits in the APM could prevent the L1 from maintaining nominal performance. The L1 controller was unable to compensate in case of a sensor fault, or a failure of a motor. The L1 controller was able to compensate for a motor only delivering 41% of its expected output, and maintain its attitude, unlike the nominal system, which became unstable.

The L1 controller was converted to discrete time, and written into the user loops in the APM. The performance logs showed that the code was a major load on the APM, but it was able to maintain its nominal performance.

Flight test showed a small improvement compared to the nominal system. The low pass filter consisting of D and K needed to be retuned for the flight test.

CHAPTER 13

Future Work

Small projects

The scope of these projects are limited, and could be considered as a supplement to a project when starting from scratch. They do not require any modification to the source code either.

- Comparison between cross and plus configurations
- Hardware in the Loop tests
- Navigation gain tuning
- Camera with livefeed
- Optical flow sensor or other positioning methods
- Testing rig

Advanced projects

These projects requires knowledge about advance control techniques, and should not be undertaken until a working autopilot has been implemented. These projects may require modification of the source code.

- Detailed theoretical model for L1 norm stability proof
- Implementation on faster autopilot hardware
- L1 controller based on theoretical model
- L1 APM code generated from Matlab script
- Detailed model of couplings
- Examine the pitch/roll stability with constant yaw rate during motor failure
- Fault-tolerant control
- Flying Robot at DTU Robocup

The Sky's the Limit!

References

[1] E. Holm. (2012, December) Nu skal danske droner overflyve arktis. [Online]. Available: http://ing.dk/artikel/134642-nu-skal-danske-droner-overflyve-arktis

[2] J. Møllerhøj. (2014, January) Københavns brandvæsen vil bruge droner under indsatser. [Online]. Available: http://ing.dk/artikel/koebenhavns-brandvaesen-vil-bruge-droner-under-indsatser-165808

[3] Amazon. (2014, February) amazon prime air. [Online]. Available: http://www.amazon.com/b?node=8037720011

[4] D. Drones. (2013, September) Diy drones. [Online]. Available: http://www.diydrones.com/

[5] A. Tridgell, "Apm on linux - porting an 8 bit autopilot to linux," January 2014. [Online]. Available: http://uav.tridgell.net/LCA2014/AP_Linux.pdf

[6] ——, "A peek into the future of ardupilot," January 2014. [Online]. Available: http://diydrones.ning.com/profiles/blogs/a-peek-into-the-future-of-ardupilot

[7] M. W. T. André B.Ø. Bertelsen, "Integration and testing of avionics - 2nd edition," Tech. Rep., 2012, special course at DTU. [Online]. Available: https://dl.dropboxusercontent.com/u/29685684/DTU/rapport_v2.pdf

[8] F. L. L. Brian L. Stevens, *Aircraft Control and Simulation*, 2nd ed. WILEY, 2003.

[9] T. Jiinec, "Stabilization and control of unmanned quadcopter," Master's Thesis, Czech Technical university in Prague, May 2011. [Online]. Available: https://pure.ltu.se/ws/files/33849977/LTU-EX-2011-33770150.pdf

[10] A. F. Sørensen, "Autonomous control of a miniature quadrotor following fast trajectories," Master's Thesis, Aalborg Univercity & U.C. Berkeley, June 2010. [Online]. Available: http://projekter.aau.dk/projekter/files/32313183/report.pdf

[11] M. J. Stepaniak, "A quadrotor sensor platform," PhD Thesis, Ohio University, November 2008. [Online]. Available: http://projekter.aau.dk/projekter/files/32313183/report.pdf

[12] G. McCaldin. (2013, October) 3d printed anti vibration mount. [Online]. Available: http://diydrones.com/profiles/blogs/3d-printed-anti-vibration-mount

[13] D. Drones. (2013, September) Apm:copter. [Online]. Available: http://copter.ardupilot.com/

[14] "A-max 22, Ø22 mm, precious metal brushes cll, 5 watt," October 2013. [Online]. Available: http://www.maxonmotor.com/medias/sys_master/8807016136734/13-121-EN.pdf

[15] InvenSense. (2013, December) Mpu-6000 datasheet. [Online]. Available: http://invensense.com/mems/gyro/documents/PS-MPU-6000A-00v3.4.pdf

[16] D. C, "Arducopter tuning guide," October 2013, note: From AC wiki tuning page: Dave C's AC2.8.1 tuning guide has good information for tuning for rate roll and pitch but altitude hold, Loiter and navigation has changed dramatically since AC2.8.1 so those sections are no longer valid. [Online]. Available: http://diydrones.com/forum/topics/arducopter-tuning-guide

[17] E. K. E. X. I. M. G. Naira Hovakimyan, Chengyu Cao, "L1 adaptive control for safety-critical systems," *IEEE Control Systems Magazine*, vol. 31, no. 5, October 2011.

[18] C. C. Naira Hovakimyan, *L1 Adaptive Control Theory*. SIAM, 2010.

[19] E. X. Brian Griffin, John J. Burken, "L1 adaptive control augmentation system with application to the x-29 lateral/directional dynamics: A multi-input multi-output approach," August 2010. [Online]. Available: http://ntrs.nasa.gov/archive/nasa/casi.ntrs.nasa.gov/20100037212.pdf

Appendices

Appendix A: Using the Dataflash

Disclaimer

This is a copy of a section from the report, "Integration and Testing of Avionics - 2nd Edition"[7]. There for some of the references might not work as intended.

Using the Dataflash

The standard APM 2.0 comes with a 16MB dataflash, useful for logging various data during flights, but it requires a little effort to use it.

There are three steps related to logging data using the dataflash. 1. Set which data to log 2. Log the data 3. Extract the data

After the data has been extracted it should be imported into Matlab for analysis, and the way we have done it is described in appendix **??** on page ??. Set which data to log The file APM_config.h should be modified, to tell the APM what data it should log. The user can specify a number of parameters related to the APM, by adding "#define" statements to APM_config.h. The available parameters can be found in APM_Config.h.reference. The parameters available for logging onto the dataflash, along with a description of each, can be found in the source code, in in the "DEBUGGING" section. Each parameter corresponds to a struct containing a number of different values, and a description of each can be found on ArduPlane [**?**]. It is advised to always log the GPS-parameter, since this contains a time-stamp, which makes the log easier to read.

If you e.g. want to log the GPS and attitude at 50Hz, but not anything else, you should add the following code to APM_Config.h:

```
#define LOG_ATTITUDE_FAST       ENABLED
#define LOG_ATTITUDE_MED        DISABLED
#define LOG_GPS                 ENABLED
#define LOG_PM                  DISABLED
#define LOG_CTUN                DISABLED
#define LOG_NTUN                DISABLED
#define LOG_MODE                DISABLED
#define LOG_RAW                 DISABLED
#define LOG_CMD                 DISABLED
#define LOG_CUR                 DISABLED
```

Notice that LOG_ATTITUDE_FAST logs the attitude at 50Hz, while LOG_ATTITUDE_MED only logs at 10Hz.

After the new code has been uploaded to the APM, the dataflash should be emptied, since this makes it easier to find the new logs. Open Mission Planner, select the Terminal tab and connect to the APM using an USB-cable. You may need to refresh the terminal by clicking the tab again to get a communication going.

When the command prompt is ready, enter "logs" to view available logs stored on the dataflash, and enter "erase" to delete them all. After this step you are ready to disconnect the APM, and log the data.

Log the data

The data logging is in progress whenever the APM is turned on. The datalogging will end when the APM is powered off, without dataloss. Remember that the limited space in the dataflash will result in the dataflash overwriting itself after about 20 minuttes depending on how many parameters are being logged. An overwrite is easy to spot when viewed in the MP in that the log will start on line x, and end on line x-1.

Extract the data

To get the data onto you computer, connect to the APM via the Mission Planner Terminal, as you did when emptying the dataflash. Now click on the "Log Download" button, and click "Download All Logs" in the window that pops up. Reading the dataflash is a slow process, and may take upwards of 15 minutes, with very little indication that it is working. When done, the file will be stored as a ".log" file in the logfiles folder. Each session results in one .log-file containing all the data it was supposed to log. The name of the file will be the time it began to log. The next step is to import the .log file into Matlab for data processing.

Appendix B: Gyroscope and Accelerometer

Disclaimer

This is a copy of a section from the report, "Integration and Testing of Avionics - 2nd Edition"[7]. There for some of the references might not work as intended.

Gyroscope and Accelerometer

The gyroscope and accelerometer is implemented in the MPU-6000 sensor chip. The way the chip has been implemented in the APM allows the attitude to be updated at 50Hz. This sensor is essential for the project, there for a test of how is measures the attitude, when it is implemented on the APM is done. Testing A sensor test has been performed in order to verify the precision of the MPU, as well as determine if the APMs the convention for positive direction of rotation is equal that of mentioned in section 2.1 in the report. The APM has been dismounted for the tests, and is hand-held, which makes testing it easier. A picture of the test rig in figure B.1.

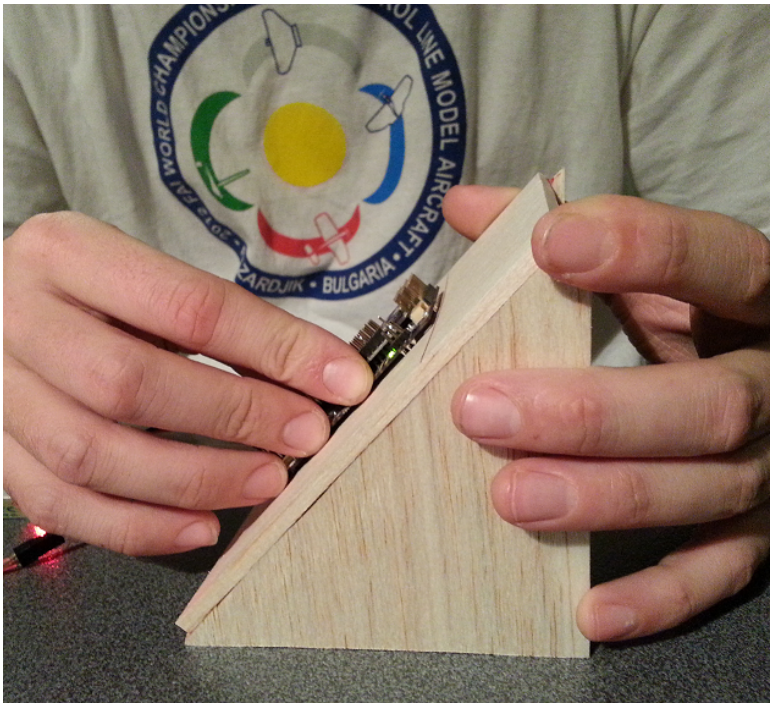


Figure B.1: *The test rig at an angle of 45°*

For the first test, the APM is placed on a table, and afterwards a 45° slope. On the slope the APM is rotated to emulate a clean pitch up and down, as well as a clean roll 45° to each side. The results are shown in table B.1 on the next page and in figure B.2 on the facing page

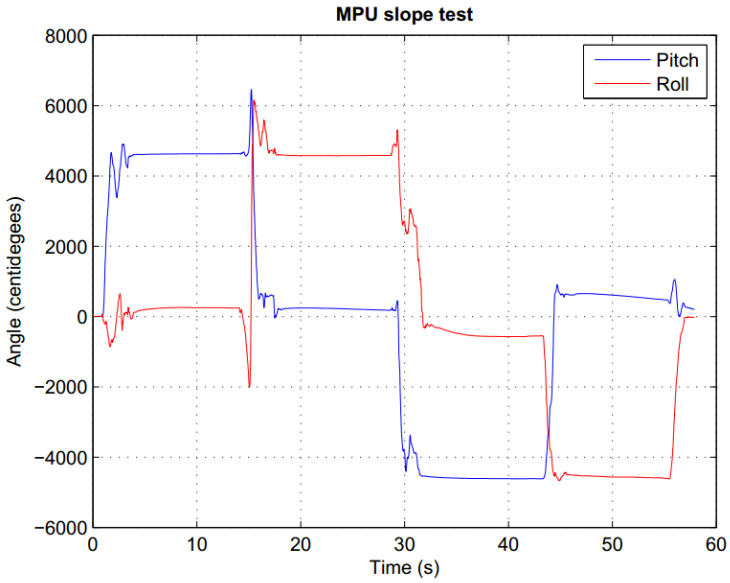


Figure B.2: *Results of the slope test.*

Test	Mean pitch [deg]	Mean Roll [deg]
45 Pitch	46	2
-45 Pitch	-46	-5
45 Roll	2	46
-45 Roll	5	-46

Table B.1: *The result of the slope test.*

The result shows some correlation between pitch and roll, and a 1° offset in both pitch and roll, but this is most likely due to the lack of precision in the test setup. More interesting is the fact that the APM measures angles in centi-degrees instead of degrees, this is most likely done to avoid truncation and rounding errors. The convention for positive direction matches the conventions established in the theory section.

The next test examines the limits of the sensor by flipping the APM around each of its axis, a full rotation in each direction. First the pitch axis is examined, the results can be seen on figure B.3 on the next page.

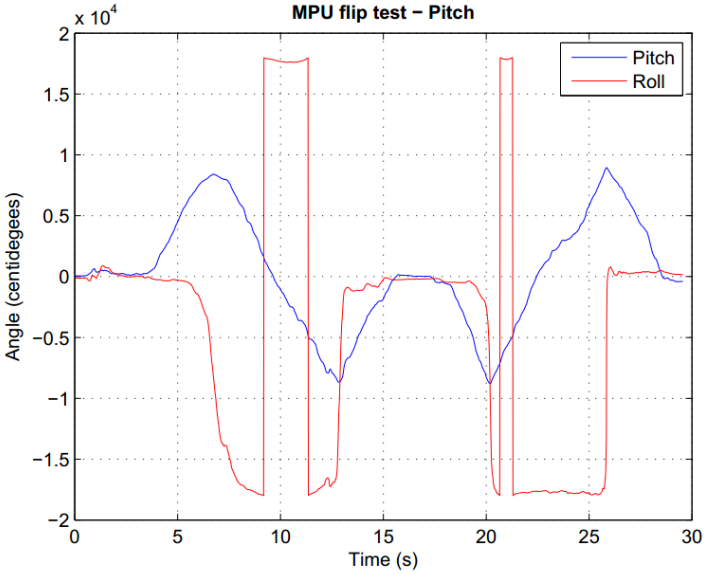


Figure B.3: *Results of the slope test for pitch.*

During the period from 3 to 16 seconds, the APM is flipped in positive pitch direction, but at 7 seconds, the direction seems to change. This is due to the way the APM handles inversions, which means that the pitch will never exceed an absolute of 90°. During the period when it is upside-down (7 to 13 seconds) the roll attitude shifts by 180°, also due to the APMs inversion handling. This will pose a problem in black box modelling and controller design, if the aircraft is to perform loops, but this is not the case in this project. Since the aircraft is not suppose to fly upside-down in this project. Notice that the roll attitude assumes values between -180° and 180° , it will shifts by 360° in the appropriate direction, until it is within that interval.

The next test examines the roll sensor in a similar fashion, the results can be seen on figure B.4.

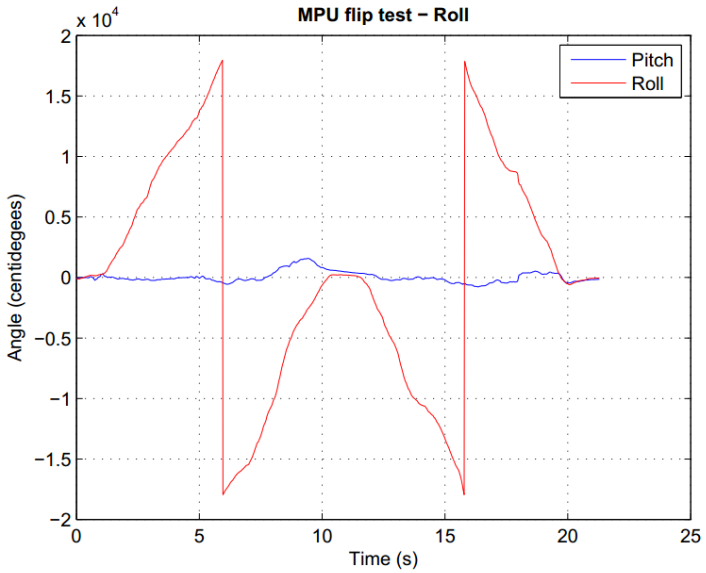


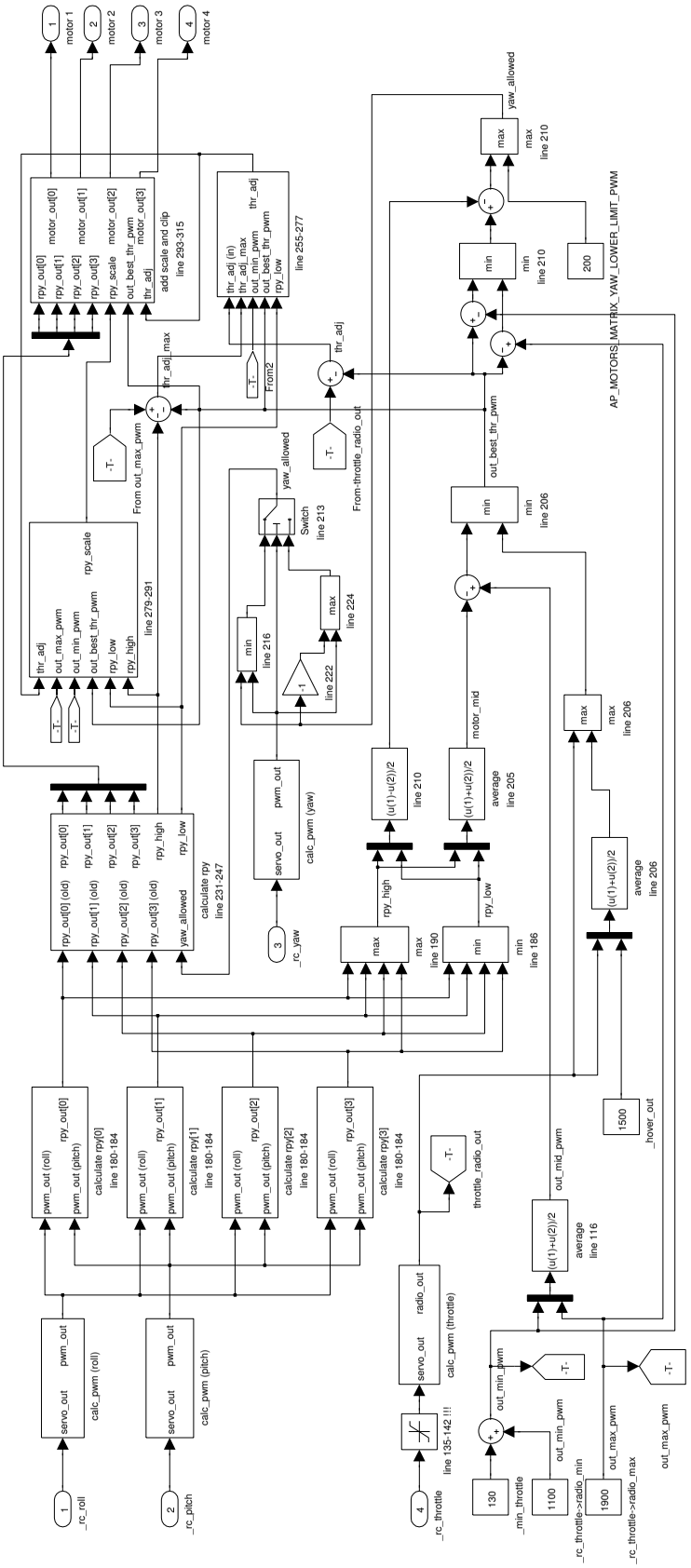
Figure B.4: *Results of the slope test for roll.*

It is easy to see that the roll exceeds 180° at 6 seconds, and promptly shifts to -180°. The pitch remains steady around 0°, even during the inversion. The fluctuations are due to the hand-held nature of the experiment.

Summary The test have been preformed, it is seen that the sensor preforms very well, it is indeed useable for the project, the one thing to look out for is when the sensor is rotated "to much", when the APM thinks it flies upside-down. It was also learned that the output was in centi-degrees.

Appendix C: APM mixer

The figure C on the next page, shows the mixer, an implementation of the function *Output_Armed* from the file AP_MatrixMotors.cpp


motor 1
motor 2
motor 3
motor 4
add scale and clip
line 293-315
line 255-277
line 279-291
From out_max_pwm
From2
From-throttle_radio_out
Switch
line 213
line 216
line 222
line 224
calc_pwm (yaw)
calculate rpy
line 231-247
line 210
average
line 205
line 206
line 190
line 186
calculate rpy[0]
line 180-184
calculate rpy[1]
calculate rpy[2]
calculate rpy[3]
calc_pwm (roll)
calc_pwm (pitch)
line 135-142 !!!
calc_pwm (throttle)
throttle_radio_out
average
line 116
AP_MOTORS_MATRIX_YAW_LOWER_LIMIT_PWM
_rc_roll
_rc_pitch
_rc_yaw
_rc_throttle
_min_throttle
_rc_throttle->radio_min
_rc_throttle->radio_max
_hover_out
out_min_pwm
out_max_pwm
out_mid_pwm
motor_mid
yaw_allowed
thr_adj
thr_adj_max
out_best_thr_pwm
rpy_scale
rpy_low
rpy_high
motor_out[0]
motor_out[1]
motor_out[2]
motor_out[3]
rpy_out[0]
rpy_out[1]
rpy_out[2]
rpy_out[3]
pwm_out
servo_out
radio_out
1100
1900
1500
130
200

Appendix D: Quadcopter photos

This appendix will show photos of the quadcopter and figures rendered in 3D CAD [1]. This is done for two main reasons, to give the reader a sense of the quadcopter and to show the detail of the 3D model.

Figure D.1: *A photo of the real quadcopter.*

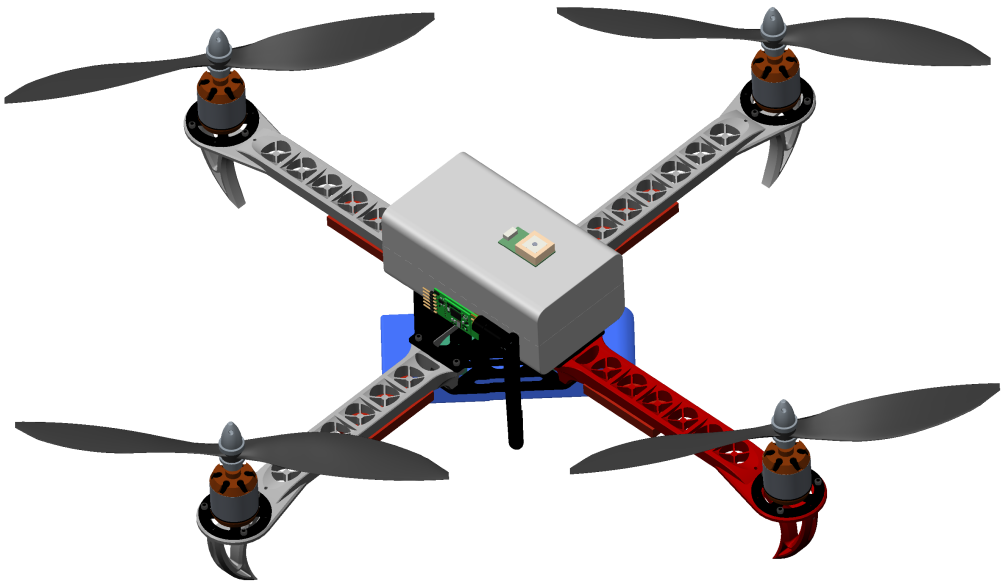


Figure D.2: *A 3D render of figure D.1.*

[1] PTC CREO 2.0

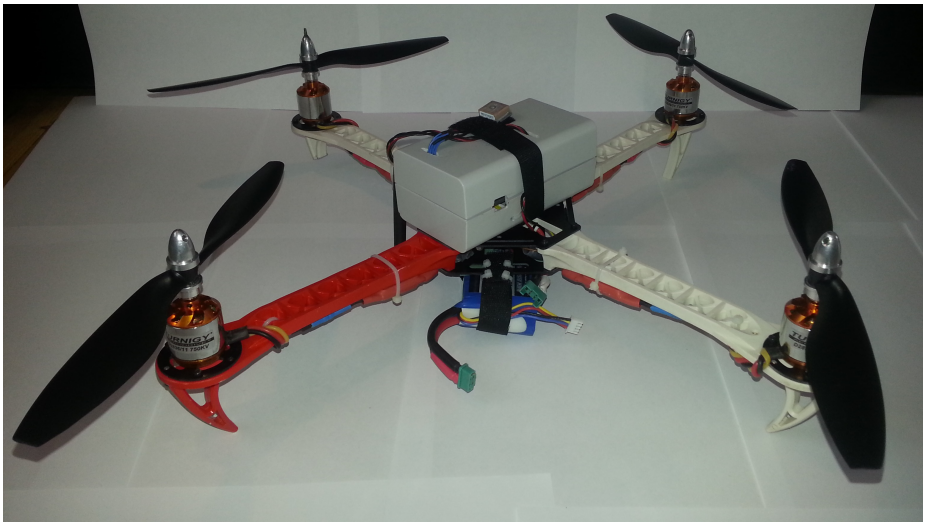


Figure D.3: *A photo of the real quadcopter.*

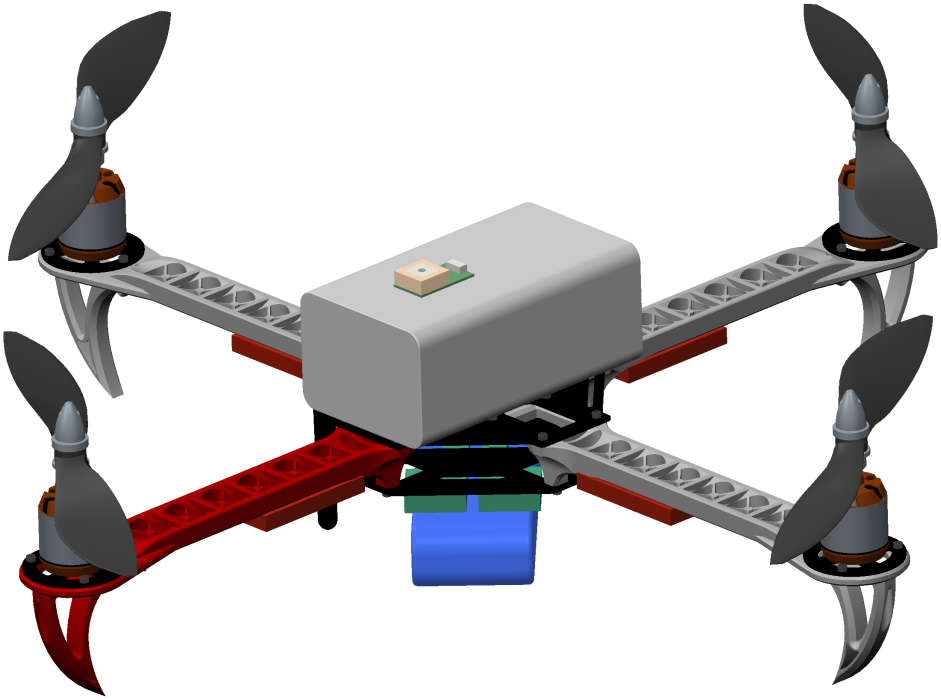


Figure D.4: *A 3D render of figure D.3.*

Figure D.5: *A photo of the real quadcopter.*

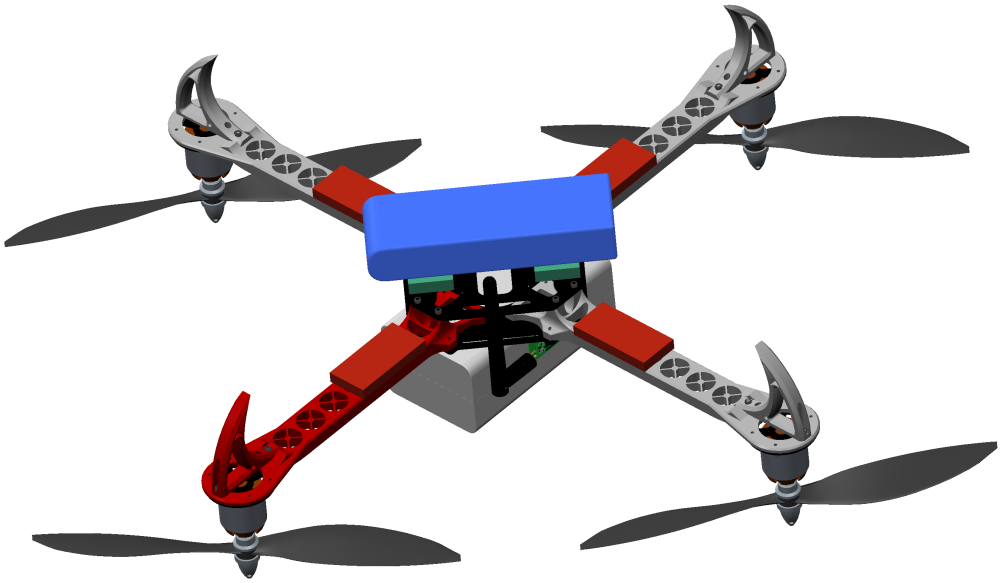


Figure D.6: *A 3D render of figure D.5.*

These two photos is just to the inside of the box, the wires and the mount, it is a snug fit.

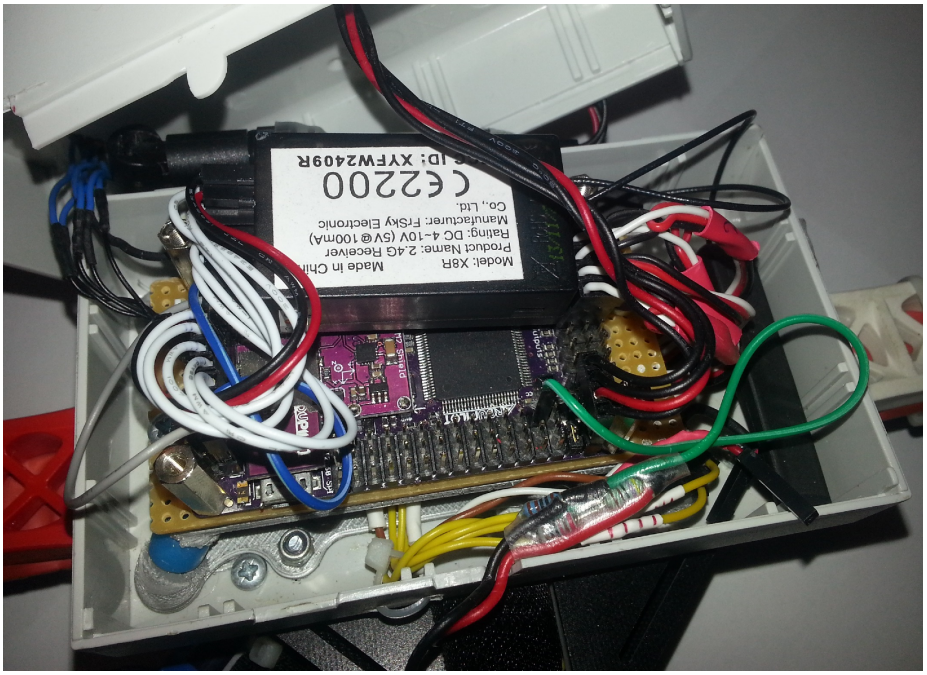


Figure D.7: *A look inside the box where the APM, RC receiver is mounted.*

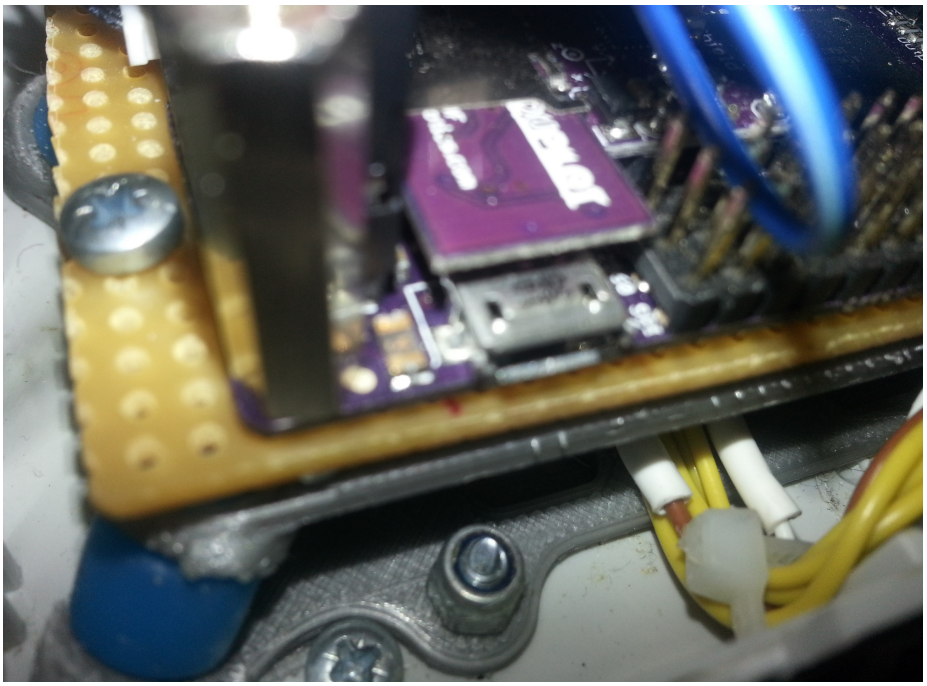


Figure D.8: *Close up of the vibration damping silicone feet (the blue part, lower left conor), and the APM USB connection.*