Norwegian University of
Science and Technology

Autonomous Drone with Object Pickup Capabilities

Bjarne Kvæstad

Master of Science in Cybernetics and Robotics
Submission date: June 2016
Supervisor: Tor Engebret Onshus, ITK

Norwegian University of Science and Technology
Department of Engineering Cybernetics

Project Formulation

The goal is to design and build a drone with pickup and delivery capabilities. It should be possible to preprogram a pickup and delivery route based on GPS coordinates. The drone should be able to pick up payloads of at least 400 grams. For safety reasons it should be possible to override the preprogrammed routine with a regular RC remote. It should also be possible to manually control the drone with the RC remote, with reference to global position (GPS) or roll, pitch and yaw. This project will continue the progress from the fall project. The objectives are listed under

1. Improve the orientation controller
2. Implement a position controller with necessary sensors
3. Implement a method for the drone to recognize objects to be picked up, and lock on to the object
4. Implement a method for picking up an object
5. Do an overall test of all the objectives mentioned above
6. If there is enough time a collision detector system should be implemented

Preface

This is a master's thesis for Cybernetics and Robotics at Norwegian University of Science and Technology (NTNU). This master thesis was executed during the spring semester of 2016.

Since I started to study Computer Science and Industrial Automation at HiT (Telemark Univercity Collage) I've had deep interest for programming and electronics. This thesis is an opportunity to combine microcontrollers and cybernetics in one project. With powerful electronics cheaply available, it makes it easy to create really awesome stuff. I wanted to create a drone with the capabilities to pick up an object from one location, and deliver it to another, this is what this thesis is all about. Because of the price of electronics I am able to finance this project out of my own pocket, and can keep developing it after I have delivered the thesis. This project has been a learning experience where I can use theory in a practical environment. There have indeed been some challenges along the way, but without this the project would be too simple, and maybe boring.

This master's thesis as a continuation of the pre-project executed during the fall semester of 2015, it is therefore recommended that the reader has also read the pre-project report for this master, referred to in apendix B. An insight in electronics and microcontrollers, mathematics, programming, Simulink and controller design is suggested as prerequisite for understanding the contents of this report.

Trondheim, 05.06.16

Bjarne Kvæstad

Acknowledgment

I would like to thank my supervisor, Tor Onshus, for his help and guidance in this project. I would also like to thank the guys in the electronics department for letting me use their equipment to create parts for the drone, building the drone would have been much harder without them. I would also give huge thanks to Pål Jacob Nessjøen for selling me used motors, when the old ones was broken, this literally saved this thesis. A thanks to Erlend Ese for letting me borrow a IR sensor. And finally a thanks to my good friends, Egil Gundersen, Martin Kostveit and Ole Edvart Husbråten, for helping me with testing the drone.

B.K.

Summary and Conclusions

The objective with this project was to create an autonomous drone with the capability to pick up, and deliver an object, without any human interference. This is an continuation of the work done during the pre-project.

The obectives during this project were:
1. Improve the orientation controller
2. Implement a position controller with necessary sensors
3. Implement a method for the drone to recognize objects to be picked up, and lock on to the object
4. Implement a method for picking up an object
5. Do an overall test of all the objectives mentioned above
6. If there is enough time a collision detector system should be implemented.

Before the orientation controller could be improved it was necessary to revise the orientation data processing algorithms. After implementing a calibration routine, motor throttle compensation, rollover compensation and acceptance test for the compass, and a tilt compensation function for the gyroscope, the results were satisfying.

An LQR controller and cascade controller was tested in order to find the best alternative for controlling the drone's orientation. Both controllers were tested and tuned against the drone model in Simulink before implemented on the drone's MCU for further testing and tuning. The cascade configuration was precise and responsive, and proved to be the superior controller design for this use. With this, the first objective was met.

In order for the drone to be implemented with a position controller the following new hardware was installed.

- GNSS receiver, for measuring position and velocity
- Pressure sensor, for accurately measuring altitude
- Vision sensor, for detecting objects

- IR sensor, for detecting the ground

In order to get a visual feedback of the current status of the drone, LED's were installed. Appropriate drivers were developed for the new peripheral devices. Because of the new hardware a new PCB design was necessary, the new design made the electronics more concealed, and the drone more compact. A hook connected to a servo was suggested as a method for picking up objects, but was not implemented due to controller issues.

In order for the visual representation of the drone's position and -velocity to be easy to comprehend the GNSS data was converted to ENU coordinates, and pressure sensor data was recalculated to altitude. Sensor fusion with kalman filtering was implemented to filter the velocity data, and a second order low-pass filter was used to filter the altitude data, this gave noise reduced measurements.

The controller design used for positioning was a cascade controller, and was tuned and tested in the same manner as the orientation controller. The tuning of the position controller had to be done outside, but due to harsh weather conditions, not enough time was spent on tuning the controller, the parameters were therefore not optimal. The overall controller design and implementation worked as planned, with this, the second objective was met.

The raw data from the vision sensor, object x- and y position and object width, was recalculated to a three dimensional position vector, so the object position data could be used in the position controller. To make testing easier, the vision sensor was also modeled in Simulink.

The flight controller combines all the controller and data processing modules plus necessary logic for the modules to cooperate. This was designed in Simulink because of the possibility to simulate against the drone model. Four flight modes were implemented in the flight controller, 1. Manual mode, 2. Manual GNSS mode, 3. Autonomous mode, and 4. Autonomous object mode. A battery monitoring system was also implemented for warning the user of low battery via the LED's.

A real-time operating system was implemented due the need for doing several operations at the same time, and some modules having a strict execution deadline. There was implemented a total of seven tasks and one interrupt routine. The operating system worked flawlessly.

All modes except "autonomous object mode" were tested on the physical drone, though the functionality of all three modes worked as expected there was definitively room for improve-

ments. When testing "manual mode" an issue arose, when given a roll- or pitch input the drone was not able to hold its attitude over time, and would flatten out as it picked up speed. This affected the drone's top speed, which was noticeable on the performance in "autonomous mode" and "manual GNSS mode", this issue affected the drone's ability to hold its geographic position, and move to another position. The root cause of this issue was tough to be poor controller parameters. Because of this issue there was no point in implementing the "autonomous object mode" on the drone, this was rather simulated in Simulink, and the result was satisfying enough to suggest the mode to be implemented on a later moment.

Due to the issues mentioned above, the "autonomous object mode" was never implemented on the drone, there was no point in implementing this mode as the position controller was not precise enough. With this, the third and fourth objective was not met, although much of the work has been done. A test of all the implemented objectives was done, the fifth objective was therefore met. There was no time to complete objective six.

Contents

Project Formulation i

Preface ii

Acknowledgment iii

Summary and Conclusions iv

1 Introduction 2

1.1 Background 2

1.2 Objectives 3

1.3 Approach 4

1.4 Structure of the Report 4

2 Orientation Data Processing 6

2.1 Electronic Compass 7

2.2 Tilt Compensated Gyroscope Data 15

3 Orientation Controller 17

3.1 Controller Implementation 17

3.2 LQR Controller 19

3.3 Cascade Controller 21

4 Hardware 24

4.1 GNSS Receiver 25

4.2 Digital Pressure Sensor 27

4.3 Vision Sensor 27

4.4 LED's 29

4.5 PCB Design 30
4.6 IR sensor 32
4.7 Object Pickup 34

5 Position Data Processing **35**
5.1 Raw Data Processing 36
5.2 Filtering 38

6 Position Controller **43**
6.1 Implementation 43
6.2 Controller 44

7 Object Tracking **48**
7.1 Data Processing 48
7.2 Modeling 51

8 Flight Controller **54**
8.1 Modes 55
8.2 Battery Monitoring 61

9 Implementing a RTOS **62**
9.1 GNSS Receive Interrupt Routine 64
9.2 Hardware Initiate Task 66
9.3 Flight Controller Task 67
9.4 Altitude Acquisition Task 68
9.5 Serial Transmit Task 69
9.6 Serial Receive Task 70
9.7 Battery Monitoring Task 71
9.8 LED Control Task 72

10 Full-scale Testing **73**
10.1 Manual mode 73
10.2 Manual GNSS mode 77

10.3 Autonomous mode 80
10.4 Autonomous object mode 84

11 Discussion and Recommendations for Further Work 87
11.1 Discussion 87
11.2 Recommendations for Further Work 88

A Acronyms 89

B Project Files 91

C Simulink Screenshots 94
C.1 Orientation Data Processing 94
C.2 Orientation Controller 98
C.3 Position Data Processing 101
C.4 Position Controller 104
C.5 Object Tracking 106
C.6 Flight Controller 110

D Schematics 115

E IR sensor output characteristics 117

F List of commands 119

G List of LED sequences 120

Bibliography 121

Chapter 1

Introduction

This chapter gives an introduction to the contents and objective of this master thesis.

1.1 Background

The main objective for this thesis is to create a drone with the capabilities to pick up an object from a predetermined location, and deliver the object to another predetermined location without any human interference.

This might not only be of interest for post offices, online stores, hospitals, etc, but also if there is some object that is needed to be retrieved from a hazardous area. This project involves lots of disciplines, for example cybernetics, electronics and programming, which also makes it an interesting challenge for me to solve.

Raffaello D'Andrea's work on Flight Assembled Architecture is about using quadrotors for building large structures, where the quad rotor is picking up and placing building block to form a structure, without any human interference. But in D'Andrea's work, he uses a motion capture system to obtain a very accurate indoor positioning system [12]. In Randal Beard's paper on Quadrotor Dynamics and Control, he uses a vision sensor to hold a quadrotor over a ground based target [9].

In this paper we will combine the work of D'Andrea and Beard to design a drone that is able to pick up and move a object outdoors, instead of using a motion capture system we will use a GPS (global positioning system) receiver and vision sensor mounted on the drone.

1.2 Objectives

This thesis is a continuation of the work done during the pre-project [17], the following objectives were met during the pre-project

1. Design and build a drone
2. Design and build an orientation controller
3. Make the drone airborne by manual control

The objectives above were completed in such way that they gave a solid foundation for further development of the drone's pickup and delivery capabilities. A model of the drone was also developed during the pre-project which is often used in this thesis.

The objectives during this thesis are

1. Improve the orientation controller
2. Implement a position controller with necessary sensors
3. Implement a method for the drone to recognize objects to be picked up, and lock on to the object
4. Implement a method for picking up an object
5. Do an overall test of all the objectives mentioned above
6. If there is enough time a collision detector system should be implemented

1.3 Approach

Before we start with the objectives mentioned in section 1.2, there is some improvements that should be done, these are mentioned in the summary of the pre-project, and are listed under.

1. Different thrust from motors, find a way to improve this issue
2. Fix the electronic compass
3. New PCB (Printed Circuit Board) design with status LED (Light Emitting Diode).

The flight controller should be designed in Simulink, then it can be simulated against the drone model derived during the pre-project. This is a self financed project, so a crash because of a software bug can be fatal for this project.

When testing the physical drone it is necessary to implement different flight modes, where more and more functionality is tested, this ensures that the testing is divided into less comprehensive objectives.

1.4 Structure of the Report

The rest of this report is organized as follows

Chapter 2 Presents issues with the orientation data processing implementation done during the pre-project, and improves these issues.

Chapter 3 Tests different orientation controller designs in order to find the most precise and efficient design.

Chapter 4 Presents the hardware upgrades necessary for meeting the objectives mentioned in section 1.2.

Chapter 5 Presents the necessary algorithms for processing position and velocity data.

Chapter 6 Describes how the position controller was implemented, along with the design and tuning of the controller.

Chapter 7 Presents the necessary algorithms for calculating the position of an object using the vision sensor, along with modeling of the vision sensor for testing purposes.

Chapter 8 Presents the complete hierarchy of the flight controller along with the different flight modes.

Chapter 9 Explains how the functionality was implemented using a real-time operating system on the microcontroller unit.

Chapter 10 Shows the test conditions and test results of the different modes.

Chapter 11 Discusses the results discovered in this report.

Chapter 2

Orientation Data Processing

As mentioned in the summary of the pre-project the compass did not work as it should, and later it was uncovered that there were a few more issues with the implementation of the orientation data processing from the pre-project, these issues will be presented and solved in this chapter.

Figure 2.1 shows each element in the orientation data process and in which order they are executed, each element in this block diagram solves a particular issue, and will also be explained in this chapter.

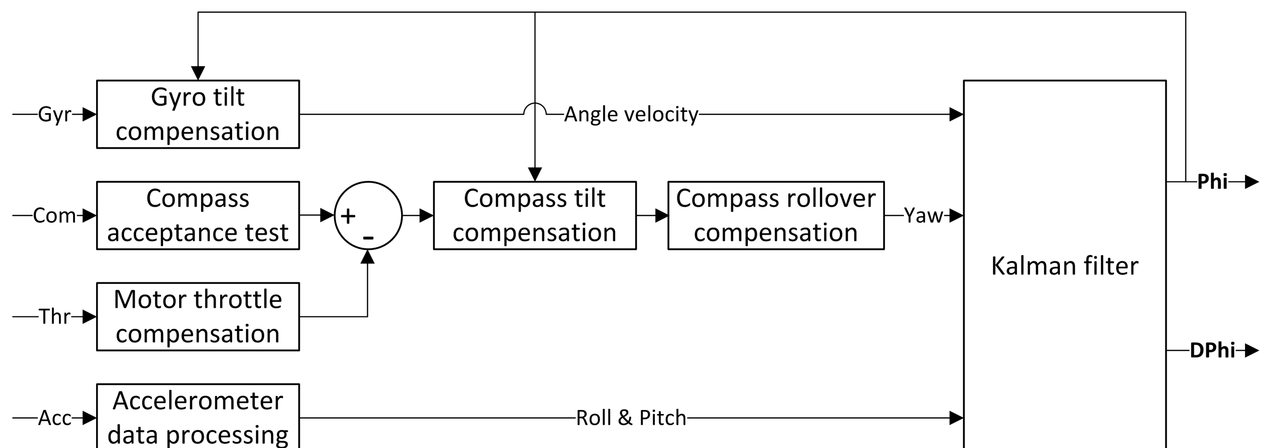


Figure 2.1: Data processing overview

The derivation of the accelerometer data processing, compass tilt compensation and kalman filter can be found in the pre-project [17].

Screenshots of the Simulink implementation can be found in appendix C.

2.1 Electronic Compass

The issues with the compass was traced to the data being offset due to a hard iron interference, caused by one of the machine screws used to mount the PCB to the frame, the compass needed to be re-calibrated.

Along with recalibration there were also some other issues, 1. motor interference, 2. the compass jumps between π and $-\pi$ when the compass point towards south. These issues are also solved in this section.

Calibration

In order to test the magnitude the compass data offset, the MCU (Micro Controller Unit) was set to log the x, y and z values measured from the compass, while the sensor was rotated around its z axis. The data before and after calibration is presented in figure 2.2.

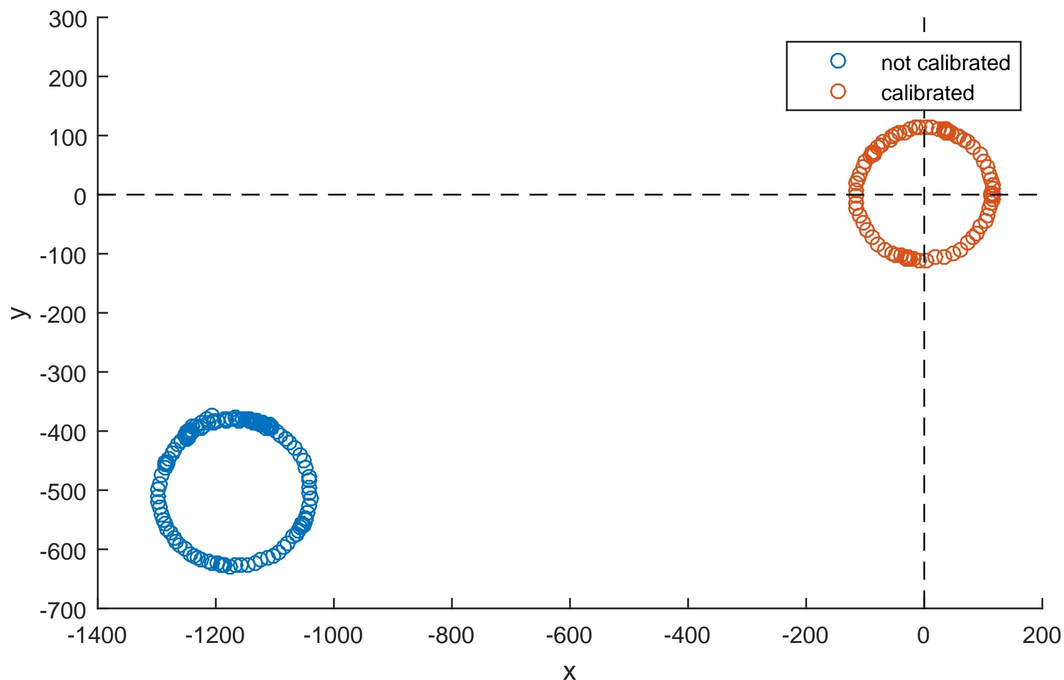


Figure 2.2: Compass data before and after calibration

It is necessary to have the measured data points in a circle around the origin of the XY plot, with this we can use the arctan function to calculate the heading. In this case we had an offset of $x = -1190$ and $y = -510$. The calibration routine was also done for the y and z data (rotation around x axis) in order to calibrate the z axis.

The electronic compass is not only extremely sensitive to other iron objects, but also its geographic location, as the direction of the earths magnetic field vector is not the same everywhere. An automated calibration algorithm was therefore developed to make the compass calibration easier. The user have to rotate the drone around its z axis, and x or y axis in order for the compass to measure the minimum and maximum values, and from this the algorithm will calculate and implement the offset parameters, see algorithm 2.1.

```
1  void calComBias()
2      while |intGyrZ| < 2π do
3          delay(15ms)
4
5          getGyroData(zGyr)
6          intGyrZ = zGyr*15ms
7
8          getCompassData(xCom, yCom)
9          if xCom > xComMax then
10             xComMax = xCom
11         end
12         if yCom > yComMax then
13             yComMax = yCom
14         end
15         if xCom < xComMin then
16             xComMin = xCom
17         end
18         if yCom < yComMin then
19             yComMin = yCom
20         end
21     end
22
23     while (|intGyrX| < 2π) && (|intGyrY| < 2π) do
24         delay(15ms)
25
26         getGyroData(xGyr, yGyr)
27         intGyrX = xGyr*15ms
28         intGyrY = yGyr*15ms
29
30         getCompassData(zCom)
31         if zCom > zComMax then
32             zComMax = zCom
33         end
34         if zCom < zComMin then
35             zComMin = zCom
36         end
37     end
38
39     xComOffset = (xComMax - xComMin)/2 - xComMax
40     yComOffset = (yComMax - yComMin)/2 - yComMax
41     zComOffset = (zComMax - zComMin)/2 - zComMax
```

Algorithm 2.1: Compass calibration algorithm

Motor throttle compensation

According to Ampère's law, the integrated magnetic field around a closed electric loop, in this case battery, wires and motors, relates to the current passing through the loop [21]. The electronic compass is affected by this magnetic field, more power to the motors have a greater impact on the measured compass data. The magnitude of the disturbance with regards to motor power can be seen in figure 2.3.

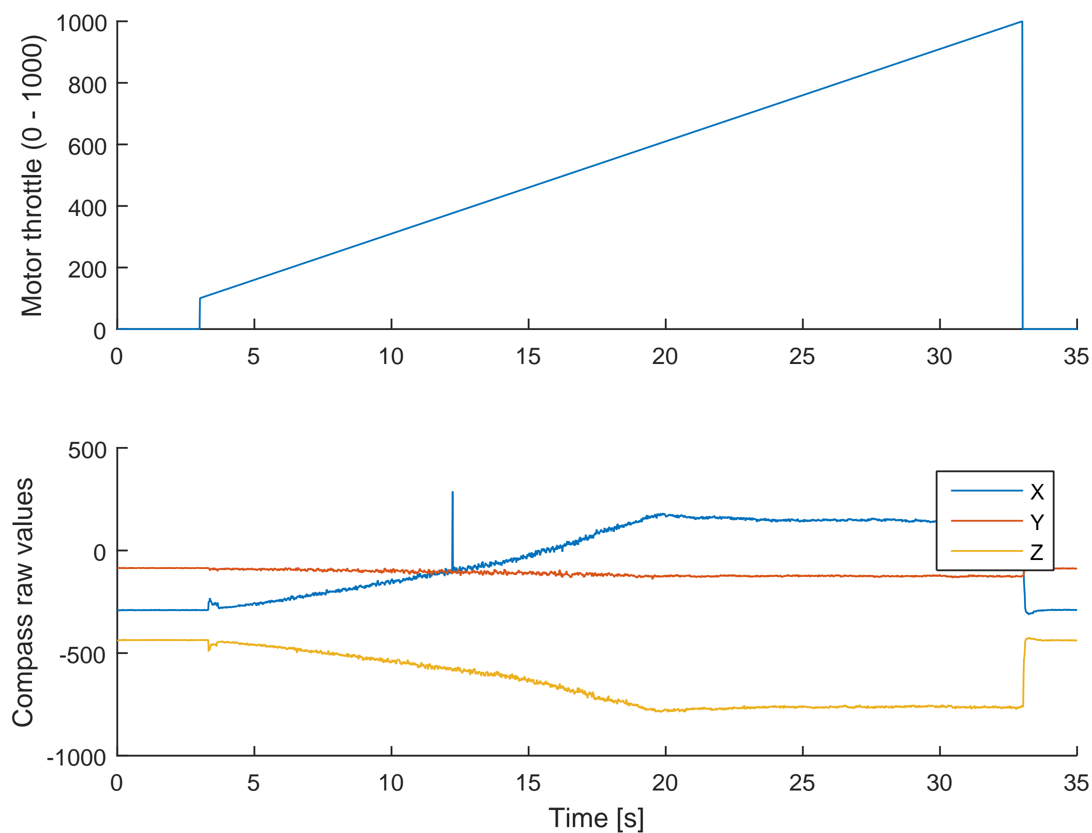


Figure 2.3: Compass disturbance test with motor ramp from 100 to 1000

During the acquisition of the compass sensor data the drone was completely stationary, and in an ideal scenario, the measured compass data should stay the same during the whole test. Instead there is at the most an offset of approximately 500, -50 and -400 on the x, y and z measurements, respectively.

To solve this issue the least square method was used to create a second order polynomial

function with regards to the throttle. With this it is possible to use the throttle to compensate the bias, as shown in function 2.1.

$$c_i[k] = \begin{cases} c_i[k] & \textbf{if } u[k] < 130 \\ c_i[k] - f_i(u[k]) & \textbf{if } u[k] \geqslant 130 \\ c_i[k] - f_i(600) & \textbf{if } u[k] \geqslant 600 \end{cases} \tag{2.1}$$

Where c_i is the compass value of axis i, $u[k]$ is the motor throttle, and $f_i(u[k])$ is the second order polynomial for axis i, shown in equation 2.2.

$$f_i(u[k]) = a_0 + a_1 u[k] + a_2 u[k]^2 \tag{2.2}$$

The coefficients, a_0, a_1 and a_2, were found by using the matrix formalized least square method, explained in Kristensen's lecture notes [16]. See equation 2.3.

$$\begin{bmatrix} a_0 \\ a_1 \\ a_2 \end{bmatrix} = \left(\mathbf{A}^T\mathbf{A}\right)^{-1}\left(\mathbf{A}^T\mathbf{Y}\right) \tag{2.3}$$

Where

$$\mathbf{A} = \begin{bmatrix} 1 & u[1] & u[1]^2 \\ 1 & u[2] & u[2]^2 \\ \vdots & \vdots & \vdots \\ 1 & u[k] & u[k]^2 \end{bmatrix} \qquad \mathbf{Y} = \begin{bmatrix} c_i[1] - c_i[1] \\ c_i[2] - c_i[1] \\ \vdots \\ c_i[k] - c_i[1] \end{bmatrix}$$

This was calculated in MATLAB, the final functions can be seen in 2.4 trough 2.6.

$$f_x(u) = -13.93 + 0.14u + 0.001u^2 \tag{2.4}$$

$$f_y(u) = -0.44 - 0.02u - 0.0001u^2 \tag{2.5}$$

$$f_z(u) = 7.40 - 0.08u - 0.0008u^2 \tag{2.6}$$

A plot of $f_x(u)$, $f_y(u)$ and $f_z(u)$ with the compass data can be seen in figure 2.4.

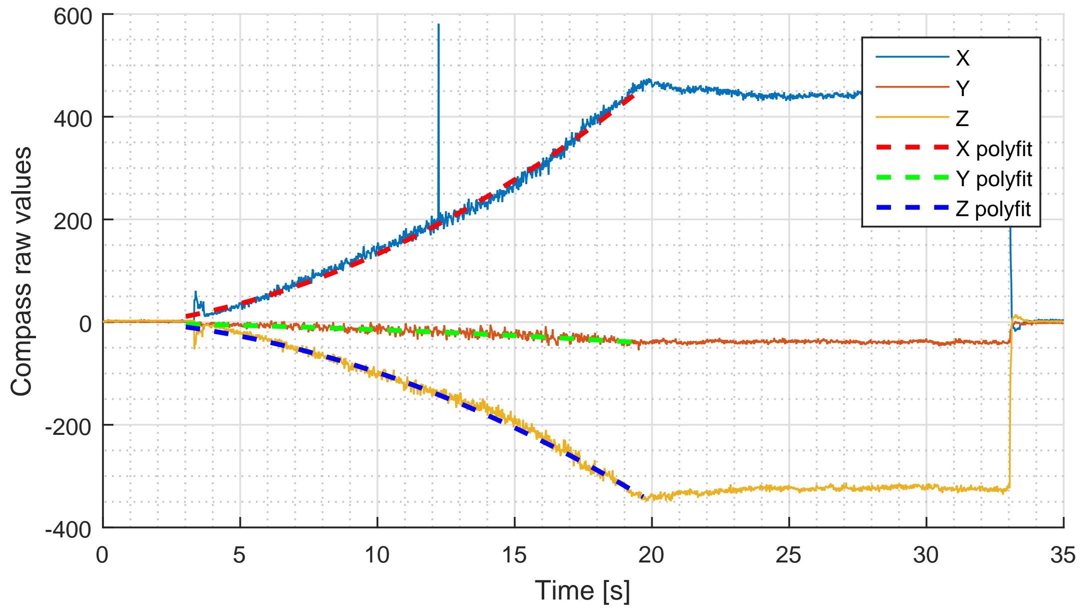


Figure 2.4: Compass data with compensation function

The results after implementing function 2.1 is shown in figure 2.5.

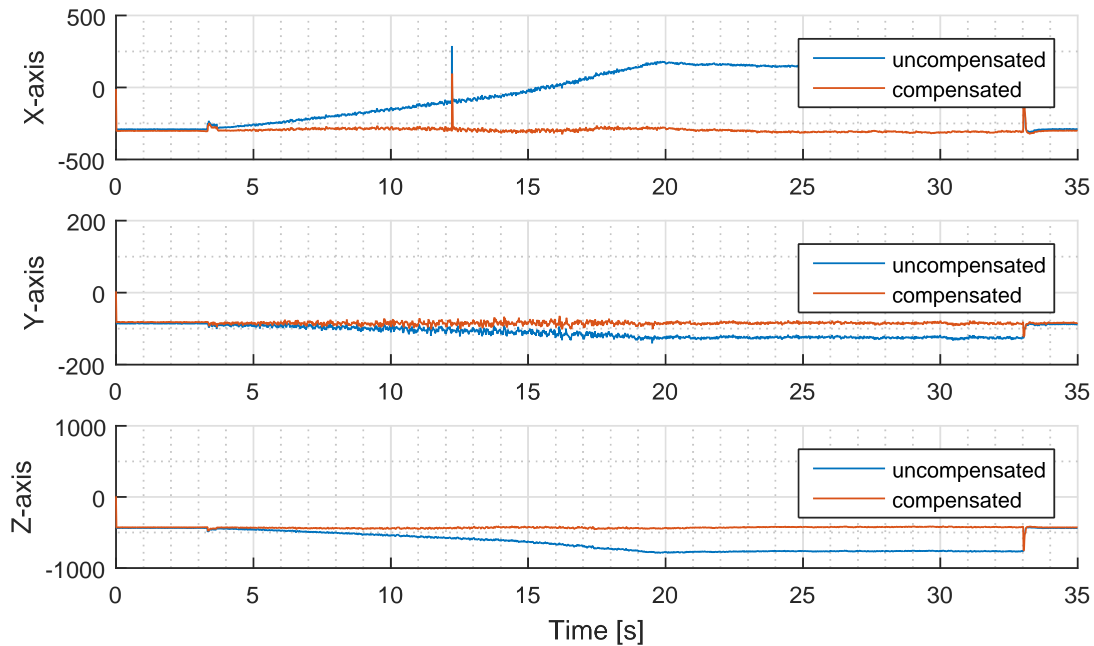


Figure 2.5: Throttle compensated compass data

As seen in figure 2.5 the bias is gone, but the small spikes at $t = 3$ is a result of the current in-

creasing to spin up the motors to the correct speed, this throttle rate of change, $\frac{du}{dt}$, is not taken into account in the function for simplicity.

The spikes in the transitions from $u[k] = 600$ to $u[k] = 0$, at $t = 33$, will be filtered out by the kalman filter.

Rollover compensation

When the compass points towards south, its processed heading is either $-\pi$ or π, this is of course normal for a compass to do. This, however, makes it difficult to process in the kalman filter as the filter would estimate the heading too slow between the jumps from $-\pi$ to π. But the greatest challenge would be to make the controller efficient and intelligent enough to take the shortest way to the desired setpoint. E.g, the current heading and setpoint is $-\pi$ or π (south). Because of noise the actual heading value would jump rapidly between $-\pi$ to π, the derivative part in the controller would go out of control giving major inputs to the motors, and make the drone unstable.

Therefore the best choice is to make the heading value count the total radians moved, with this, there will be no sudden jumps in the heading value when the drone is pointing towards north. The algorithm for recalculating the heading to an incremented heading is presented in algorithm 2.2.

```
1  double getHeading()
2      heading = atan(x_com, y_com)
3
4      if isEmpty(prevHeading) then
5          prevHeading = heading
6          rounds = 0
7      end
8
9      if prevHeading - heading > π then
10         rounds = rounds + 1
11     else if heading - prevHeading > π then
12         rounds = rounds - 1
13     end
14
15     prevHeading = heading;
16     return heading + rounds*2π
```

Algorithm 2.2: Absolute heading algoritm

Figure 2.6 shows a plot where the drone was rotated around the z-axis (yaw) two times clockwise, and then three times counter clockwise. The plot compares the data before and after the absolute heading algorithm.

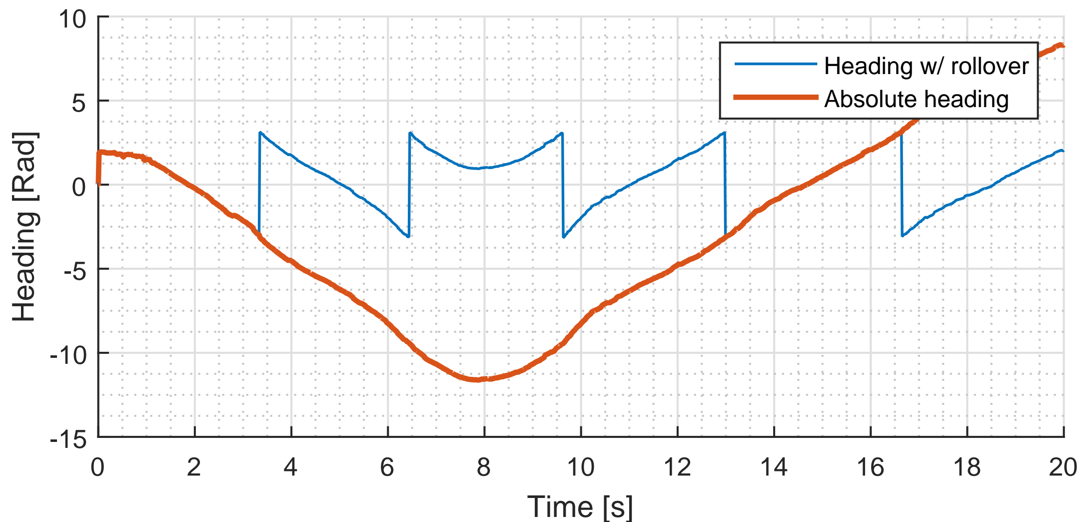


Figure 2.6: Heading with rollover versus absolute heading

Acceptance test

As seen in the x axis plot in figure 2.5, due to noise, there is a pretty significant spike at $t = 12$. This is not a problem for the controllers because it is filtered away in the kalman filter, it is however a problem for the rollover compensation, because this uses the unfiltered compass values. After some testing the drone did some unexpected 360's along the yaw axis, this was caused by spikes in the compass data that caused the rollover compensations to count an extra round. To fix this a dynamic reasonable test was implemented on all compass axes [20], see 2.7.

$$c_i[k] = \begin{cases} c_i[k-1] & \textbf{if } |c_i[k] - c_i[k-1]| \geqslant \delta \\ c_i[k] & \textbf{else} \end{cases} \tag{2.7}$$

Where $c_i[k]$ is the compass data for axis i, and the rate of change limit, δ, was found by trial and error, this was found to be 30.

This did remove most of the unexpected 360's along the yaw axis, but can still happen if the throttle increases rapidly, due to increased current to the motors, mentioned in the motor throttle compensation subsection. Though this happens only under stress tests, and does not seem to happen under normal flight.

2.2 Tilt Compensated Gyroscope Data

After thorough testing of the compass, and its function in the kalman filter a problem with the gyroscope arose; when the drone is given a small negative pitch (to go forward) and then a input on yaw (to turn while going forward), the gyroscope measures a rotation along the z axis and y axis. This becomes an issue in the kalman filter because the angle velocity inputs is greatly weighted due to its low noise variance. So in this scenario the kalman filter will estimate a significant roll as well, this could make the drone unstable and unpredictable during flight.

To fix this problem, the angular velocity needs to be transformed from body frame to inertial frame. See equation 2.9.

$$\omega_{ib}^{i} = \mathbf{R}_x(\phi)\mathbf{R}_y(\theta)\begin{bmatrix}0\\0\\\dot{\psi}_{gyr}\end{bmatrix} + \mathbf{R}_x(\phi)\begin{bmatrix}0\\\dot{\theta}_{gyr}\\0\end{bmatrix} + \begin{bmatrix}\dot{\phi}_{gyr}\\0\\0\end{bmatrix} \tag{2.8}$$

$$= \begin{bmatrix}1 & 0 & \sin\theta\\0 & \cos\phi & -\sin\phi\cos\theta\\0 & \sin\phi & \cos\phi\cos\theta\end{bmatrix}\begin{bmatrix}\dot{\phi}_{gyr}\\\dot{\theta}_{gyr}\\\dot{\psi}_{gyr}\end{bmatrix} \tag{2.9}$$

Where $\mathbf{R}_x(\phi)$ and $\mathbf{R}_y(\theta)$ are simple rotation matrices from the body system to the inertial system, defined in the pre-project [17].

Chapter 3

Orientation Controller

In this chapter we will try out two different orientation controller designs in order to find the best controller for this drone. The two controllers are cascade controller with PID (Proportional, Integral, Derivative) controllers and a LQR (Linear Quadratic Regulator) controller. Both the controllers were designed, tested and tuned in Simulink using the drone model designed during the pre-project. After this the C code for the controller was auto generated, and implemented on the drone's MCU for further tuning. When tuning the physical drone the control parameters were altered with a computer via Bluetooth, explained in section 9.6.

But first we need to revisit the controller implementation to solve the issue about different thrust from the motors, mentioned in the pre-project [17].

Screenshots of the Simulink code can be found in appendix C.

3.1 Controller Implementation

As mentioned from the pre-project there were some issues regarding the motor thrust. The thrust was measured on each motor, and the results are presented in table 3.1.

Table 3.1: Motor thrust

Motor	Throttle	Thrust
u_1	350	282 g
u_2	350	281 g
u_3	350	302 g
u_4	350	304 g

This shows that there is some significant difference in the thrust from each motor, further testing uncovered that there were small deviations in each propeller, which gave different thrust. To fix this issue the integral effect should be increased on the orientation controller, this should give the controller a bit more adaptiveness to the propeller deviations.

Because of this the controller implementation should also be changed, the setup proposed in the pre-project is suboptimal as that design would compensate all four propellers just to compensate the deviations in, for example, one propeller. The new proposal is presented in equation 3.1 trough 3.4.

$$u_1 = u_t + u_\psi + u_A \tag{3.1}$$

$$u_2 = u_t - u_\psi - u_B \tag{3.2}$$

$$u_3 = u_t - u_\psi + u_B \tag{3.3}$$

$$u_4 = u_t + u_\psi - u_A \tag{3.4}$$

Where u_n is the throttle for motor n, u_t is the ascend power, u_ψ is the yaw controller output, and u_A and u_B is the controller output for each pair of diagonal motors. Since the front of the drone is still defined between motor one and two ($\psi = \frac{\pi}{4}$, so called X-configuration), the new controller setpoints are presented in function 3.7.

$$\begin{bmatrix} r_A \\ r_B \\ r_\psi \end{bmatrix} = \mathbf{R}_z^T\left(\frac{\pi}{4}\right)\begin{bmatrix} r_\phi \\ r_\theta \\ r_\psi \end{bmatrix} \tag{3.5}$$

$$= \begin{bmatrix} \frac{\sqrt{2}}{2} & \frac{\sqrt{2}}{2} & 0 \\ -\frac{\sqrt{2}}{2} & \frac{\sqrt{2}}{2} & 0 \\ 0 & 0 & 1 \end{bmatrix}\begin{bmatrix} r_\phi \\ r_\theta \\ r_\psi \end{bmatrix} \tag{3.6}$$

$$= \begin{bmatrix} \frac{\sqrt{2}}{2}\left(r_\theta + r_\phi\right) \\ \frac{\sqrt{2}}{2}\left(r_\theta - r_\phi\right) \\ r_\psi \end{bmatrix} \tag{3.7}$$

Where $\mathbf{R}_z^T\left(\frac{\pi}{4}\right)$ is the rotation matrix around the z axis from body to inertia system. The same

functions are used for the controller input, y_A, y_B and y_ψ.

3.2 LQR Controller

In order to implement the LQR controller, two main things are needed, a complete model of the drone, and all states of the model are known measured values [8]. In this case both requirements are fulfilled, a complete model of the drone was made during the pre-project [17], and both angle and angular velocity is measured for all attitudes. Due to the issues mentioned in section 3.1, three more states were added, and that was the integrated angle of each attitude, γ, ζ and ξ. The non-linear differential equations of the drone is presented in 3.8.

$$\dot{\mathbf{x}} = \mathbf{h}(\mathbf{x}, \mathbf{u}) \tag{3.8}$$

$$\begin{bmatrix} \dot{\gamma} \\ \dot{\phi} \\ \ddot{\phi} \\ \dot{\zeta} \\ \dot{\theta} \\ \ddot{\theta} \\ \dot{\xi} \\ \dot{\psi} \\ \ddot{\psi} \end{bmatrix} = \begin{bmatrix} \phi \\ \cos\theta\dot{\phi} + \sin\theta\dot{\psi} \\ -\frac{1}{I_x}\left(\dot{\theta}\dot{\psi}I_z - \dot{\theta}\dot{\psi}I_y\right) + \frac{F_c L}{I_x\sqrt{2}}u_A \\ \theta \\ \frac{\sin\phi\sin\theta}{\cos\phi}\dot{\phi} + \dot{\theta} - \frac{\cos\theta\sin\phi}{\cos\phi}\dot{\psi} \\ -\frac{1}{I_y}\left(\dot{\phi}\dot{\psi}I_x - \dot{\phi}\dot{\psi}I_z\right) + \frac{F_c L}{I_y\sqrt{2}}u_B \\ \psi \\ -\frac{\sin\theta}{\cos\phi}\dot{\phi} + \frac{\cos\theta}{\cos\phi}\dot{\psi} \\ \frac{1}{I_z}\left(\dot{\phi}\dot{\theta}I_y - \dot{\phi}\dot{\theta}I_x\right) + T_c u_\psi \end{bmatrix} \tag{3.9}$$

To solve the Riccati equation the non linear state space model needs to be linearized around $\mathbf{\Phi} = \dot{\mathbf{\Phi}} = [0 \quad 0 \quad 0]^T$. This can be done by finding the jacobian of the matrices, this was done by using MATLAB "jacobian" command, see 3.11.

$$\dot{\mathbf{x}} = \mathbf{A}\mathbf{x} + \mathbf{B}\mathbf{u} \tag{3.10}$$

$$\begin{bmatrix} \dot{\gamma} \\ \dot{\phi} \\ \ddot{\phi} \\ \dot{\zeta} \\ \dot{\theta} \\ \ddot{\theta} \\ \dot{\xi} \\ \dot{\psi} \\ \ddot{\psi} \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} \gamma \\ \phi \\ \dot{\phi} \\ \zeta \\ \theta \\ \dot{\theta} \\ \xi \\ \psi \\ \dot{\psi} \end{bmatrix} + \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ \frac{F_c L}{I_x \sqrt{2}} & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & \frac{F_c L}{I_y \sqrt{2}} & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & T_c \end{bmatrix} \begin{bmatrix} u_A \\ u_B \\ u_\psi \end{bmatrix} \tag{3.11}$$

To calculate the K matrix we need to define the weighed matrices, Q and R, see 3.12 and 3.13.

$$Q = \text{diag}\left([0.2 \quad 85 \quad 60 \quad 0.2 \quad 85 \quad 60 \quad 0.01 \quad 50 \quad 0.1]\right) \tag{3.12}$$

$$R = \text{diag}\left([0.5 \quad 0.5 \quad 0.4]\right) \tag{3.13}$$

The weighted matrices were found by simulating the controller towards the drone model developed in the pre-project, and optimized by doing several test flights on the drone. With this we can use the MATLAB "lqr" command to find the 3x9 **K** matrix that is used in the feedback loop, as shown in 3.14.

$$\mathbf{u} = \mathbf{K}\left(\mathbf{r} - \mathbf{x}\right) \tag{3.14}$$

When testing the drone in manual mode with a RC (radio control) remote the drone reacted slowly and was quite wobbly, not as responsive as demanded if it shall be able to pick up objects. This is also reflected on the data logged via Bluetooth during flight, see figure 3.1.

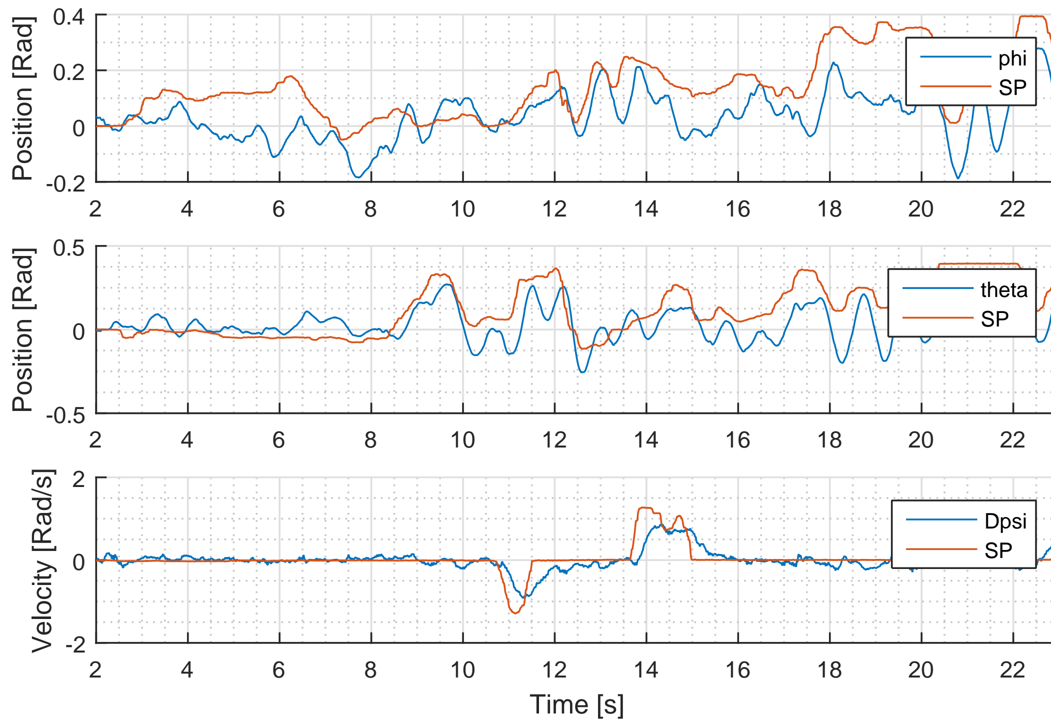


Figure 3.1: Flight log from drone with LQR controller

The poor results should not come as a surprise as the LQR controller is quite similar to a standard PID controller, which gave poor performance during the pre-project. The differences are the method for tuning, and the "derivative" part of the LQR controller uses the actual angular velocity measurements, and not the derivative position.

3.3 Cascade Controller

The cascade controller consists of two PID controllers, one controlling the angular velocity, and one controlling the angular position. The controller regulating the angular position is connected to the input of the controller who controls the angular velocity, this configuration is called a cascade controller [15]. See figure 3.2.

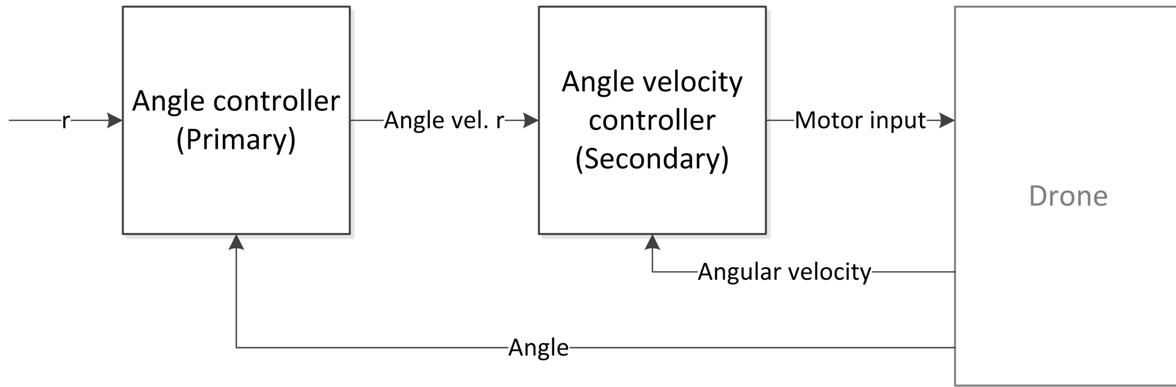


Figure 3.2: Cascade controller block diagram

The algorithm for the controllers are presented in 3.15 trough 3.19.

$$e[k] = r[k] - y[k] \tag{3.15}$$

$$u_P[k] = e[k] \tag{3.16}$$

$$u_I[k] = u_I[k-1] + \Delta t\, e[k] \tag{3.17}$$

$$u_D[k] = \frac{e[k] - e[k-1]}{\Delta t} \tag{3.18}$$

$$u[k] = K_P\, u_P[k] + K_I\, u_I[k] + K_D\, u_D[k] \tag{3.19}$$

Where $y[k]$ is the measured angular velocity or angular position, $r[k]$ is the setpoint, and $u[k]$ is the controller output. The controller tuning was done as recommended in Haugen's "Reguleringsteknikk", where the secondary controller was tuned first, then the primary controller [15]. The optimal PID parameters are presented in table 3.2.

Table 3.2: PID parameters

Attitude	K_P	K_I	K_D
Pitch position, ϕ	7.5	0	0
Roll position, θ	7.5	0	0
Yaw position, ψ	4	0	0
Pitch velocity, $\dot{\phi}$	12	1	2.3
Roll velocity, $\dot{\theta}$	12	1	2.3
Yaw velocity, $\dot{\psi}$	55	0.5	0.1

When testing the drone in manual mode with a RC remote the drone felt precise and accurate, the drone moved quickly to the desired attitude setpoint. This is also reflected on the data logged via Bluetooth during flight, see figure 3.3.

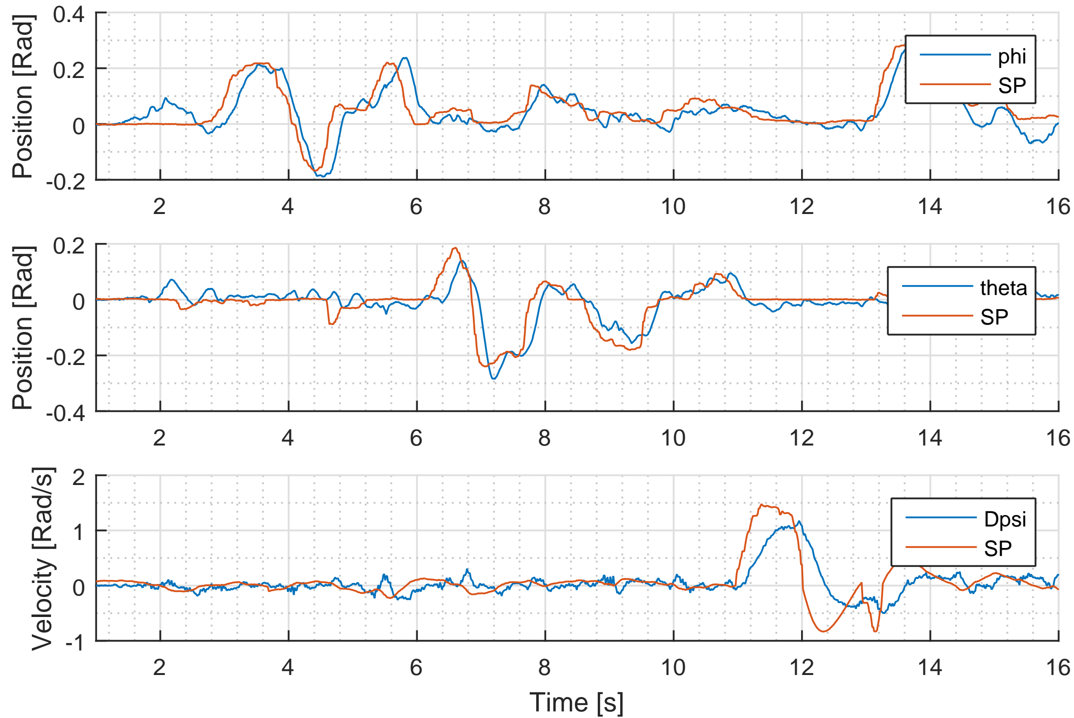


Figure 3.3: Flight log from drone with cascade controller

As seen from the plot the cascade controller performs best of both controllers and is the one that will be used during further development in this project.

Chapter 4

Hardware

During the pre-project there was installed everything needed for the drone to be controlled by manual control, this includes accelerometer, gyroscope, compass (though the compass was not used), MCU, ESC (Electronic Speed Controller), RC signal receiver and motors with propellers. In order for the drone to be able to hold its geographic position, and pick up an object at a specified coordinate, a few hardware upgrades is needed, these upgrades are presented in figure 4.1 and 4.2. In this chapter we will explain how and why these new components were installed.

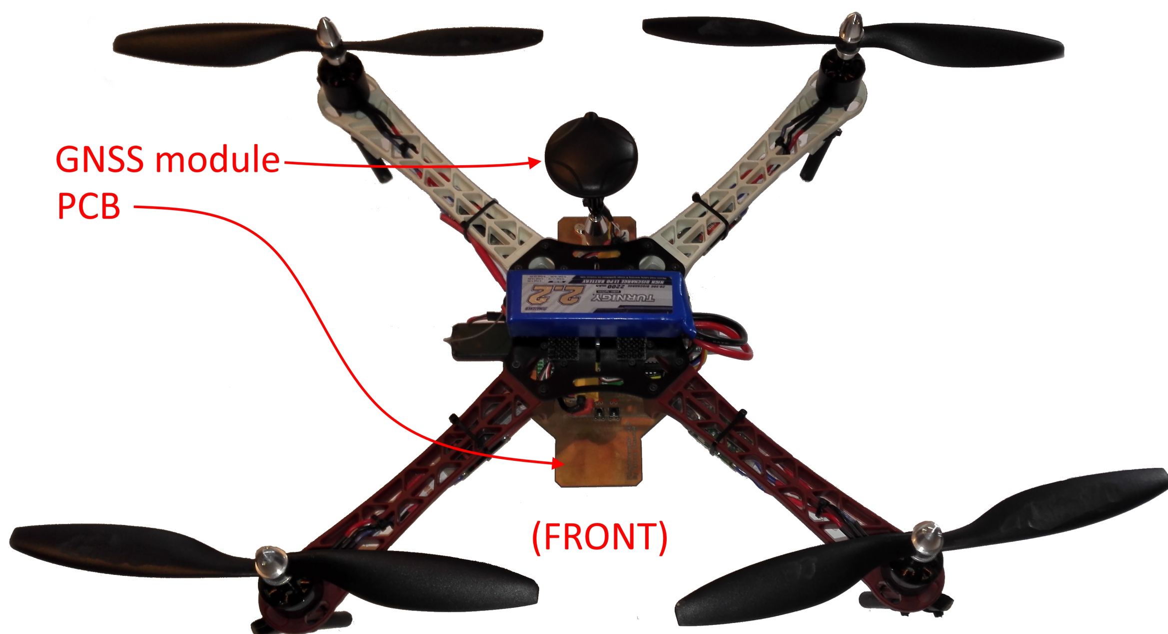


Figure 4.1: Drone, top

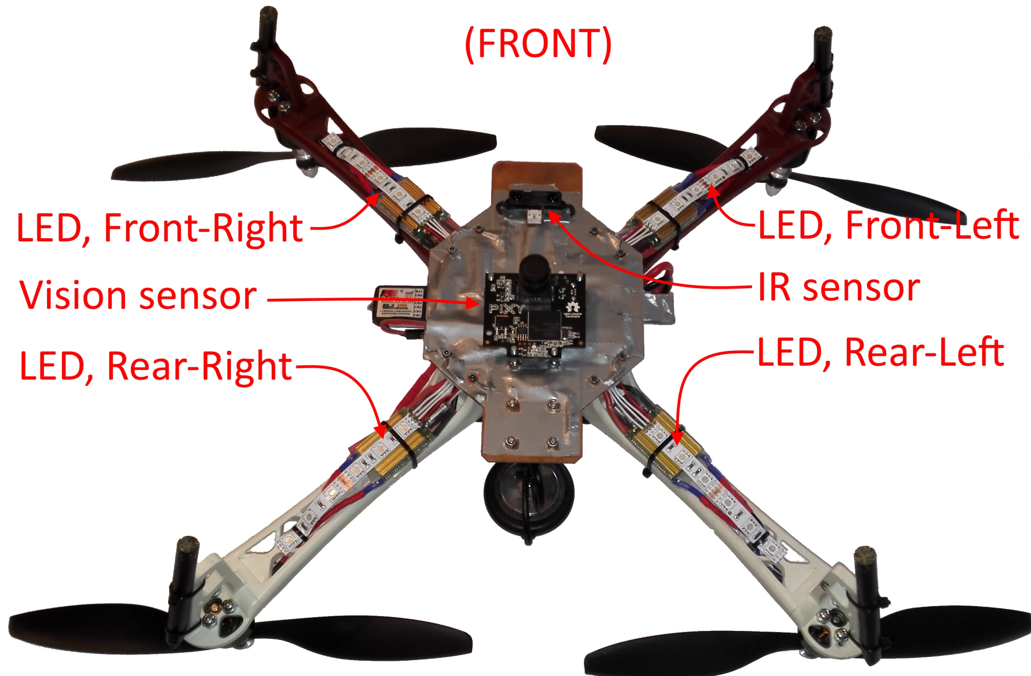


Figure 4.2: Drone, bottom

4.1 GNSS Receiver

In order for the drone to be able to fly autonomously a GNSS (Global Navigation Satellite System) receiver is needed. The GNSS module used consists of a Neo 7M GNSS receiver, by Ublox, and a built in electronic compass, though this is not used because this is already implemented in the sensor module. The GNSS receiver is also capable of measuring the velocity [4]. Figure 4.3 shows the GNSS module.

Figure 4.3: GNSS module

The GNSS receiver communicates with the MCU via UART (Universal Asynchronous Receiver-Transmitter), using either the NMEA 0183 protocol (The National Marine Electronics Association) or UBX (Ublox) protocol.

The NMEA protocol is a standard protocol for GNSS receivers and other marine electronics, the protocol frame consists of the data presented as alphanumeric text, separated by commas.

The UBX protocol was created by Ublox, the protocol frame consists of data sent in raw data bytes, which needs to be bit-shifted to create the required data formats, e.g. doubles and integers. This protocol is therefore used because it takes less CPU (Central Processing Unit) cycles to bit-shift four bytes together to make a double data type, than parsing a double from a char array. The UBX protocol is also more compact than the NMEA protocol, which means less IO interactions for the CPU[4].

The GNSS module is configured by sending commands via UART at MCU start-up, the settings used for this module is presented in table 4.1.

Table 4.1: GNSS receiver setup

Setting	Value	Description
Baudrate	115200 b/s	UART transmit/receive speed
Sample rate	90 ms	GNSS receiver position update rate
Message transmit	NAV-SOL	The NAV-SOL (Navigation Solution) data frame consists of ECEF (Earth Centred Earth Fixed) position coordinates, ECEF velocity, GPS status, position- and velocity accuracy
Position accuracy mask	10 meters	Minimum accuracy for GNSS status to be OK

The length of the NAV-SOL message is 60 bytes, with this the MCU can be set up to generate an interrupt when 60 bytes are received on the UART port, and the CPU utilization is more efficient.

The GNSS receiver is mounted directly on the PCB, and raised 10 centimeters for better satellite reception, see figure 4.1.

The driver for the GNSS receiver, written in C, is referred to in appendix B.

4.2 Digital Pressure Sensor

The GNSS-module does not meet the accuracy demands that allow the drone to fly and operate safely a few meters above the ground, a secure and accurate measurement of the altitude is therefore needed. The digital pressure sensor, BMP085 by Bosch, was mentioned and implemented during the pre-project, but was never used because it was not needed at that time. The sensor can do approximately 40 measurements per second at "Ultra High Resolution"-mode, and is accurate down to 0.25 m [5]. The sensor is soldered to the "GY-80" sensor module together with the digital gyroscope, accelerometer and compass. A picture of this module is presented in figure 4.4.

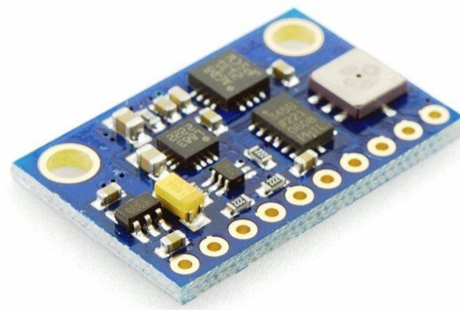


Figure 4.4: GY-80 sensor module, pressure sensor to the right (gray module)

For the barometer to be usable the data needs to be recalculated from its raw data, fetched via the I^2C interface. The recalculation also takes temperature in to account, the sensor is therefore also equipped with a temperature sensor.

The driver for the pressure sensor can be found with the driver for the rest of the drivers for the sensor on the GY-80 module, see appendix B.

4.3 Vision Sensor

For the drone to be able to detect an object, some sort of external sensor is necessary. The GNSS receiver is not accurate enough to be used for flying close enough to an object for it to be picked up.

The vision sensor used is a Pixy CMUcam5, which is fast, able to track objects at a rate of 50 Hz. The vision sensor can be programmed to track objects with color codes to display the object

width, height, x-position and y-position in pixels. The sensor can also detect the rotation of the object, and return its angle in degrees. The available communication interfaces are SPI (Serial Peripheral Interface), UART and I^2C (Inter-Integrated Circuit) [11]. A picture of the vision sensor module can be seen in figure 4.5.

Figure 4.5: Vision sensor

The camera used on the vision sensor has a resolution of 1280x800, but only half of this resolution is used, the actual resolution is therefore 640x400. The vision sensor can detect color coded objects as small as four pixels, given a measured resolution of 320x200 across the field of view. The lens used has a 75 degrees horizontal, and 47 degrees vertical field of view [11].

The vision sensor is connected to the MCU with ten 2.54 mm pin headers to pixy's communication port. SPI was chosen for the communication protocol due to its fast and full-duplex interface. The vision sensor is secured underneath the drone with four machine screws, making the vision sensor always pointing towards the ground, see figure 4.2.

The drivers for the vision sensor is referred to in appendix B.

4.4 LED's

LED strips were mounted to the drone to be able to get a visual feedback about the status of the drone, for example battery status and flight modes. The strips was mounted under each leg of the drone as shown in figure 4.2. Figure 4.6 shows a picture of the LED strips used.

Figure 4.6: LED strips

The LED strip consists of RGB (Red, Green, Blue) LEDs connected in parallel (three LED's are connected in series), when all colors are illuminated each LED uses about 60 mA [2]. The drone has mounted 6 LEDs on each leg, where three and three LED's are connected in series. The total power consumption if LED's are illuminated is shown in 4.1.

$$P = 0.06\ \text{A} \cdot 11.1\ \text{V} \cdot 2\ \text{sets} \cdot 4\ \text{legs} \tag{4.1}$$

$$P = 5.3\ \text{W} \tag{4.2}$$

In order to control the color on each leg, 12 digital outputs are needed. The MCU's GPIO (general purpose input output) are not powerful enough to power the LEDs, a Darlington transistor array is therefore needed, the ULA2803A by Texas Instruments was used for this task. The ULA2803A is a eight channel Darlington transistor array, each channel capable of 500 mA at 50 V [7].

See section C7 and D7 in the schematics on how the LEDs and transistor arrays were con-

nected. The schematics is found in appendix D.

The software driver for the LED's is referred to in appendix B.

4.5 PCB Design

The PCB designed in the pre-project was a prototype and not an optimal design as the electronics was too exposed to weather and a possible drone crash. It was also hard to mount the battery due to the PCB mounting solution.

The new PCB design removes these issues be replacing the bottom mounting plate on the F450 frame with a PCB. This will hide the electronics from the environment inside the frame, and also gives free access to mount the batteries with Velcro on top. This solution also gives the possibility to make the mounting holes for the GNSS-receiver and vision sensor directly on the PCB, making the drone design compact. The new PCB design has the following upgrades.

- LED drivers and outputs for visual confirmation and error signaling
- Easy access to MCU debug/programming port
- Holes for mounting the RC receiver with zip ties.
- One 330μF decoupling capacitor

The PCB layout can be seen in figure 4.7 and the corresponding schematics can be found in appendix D.

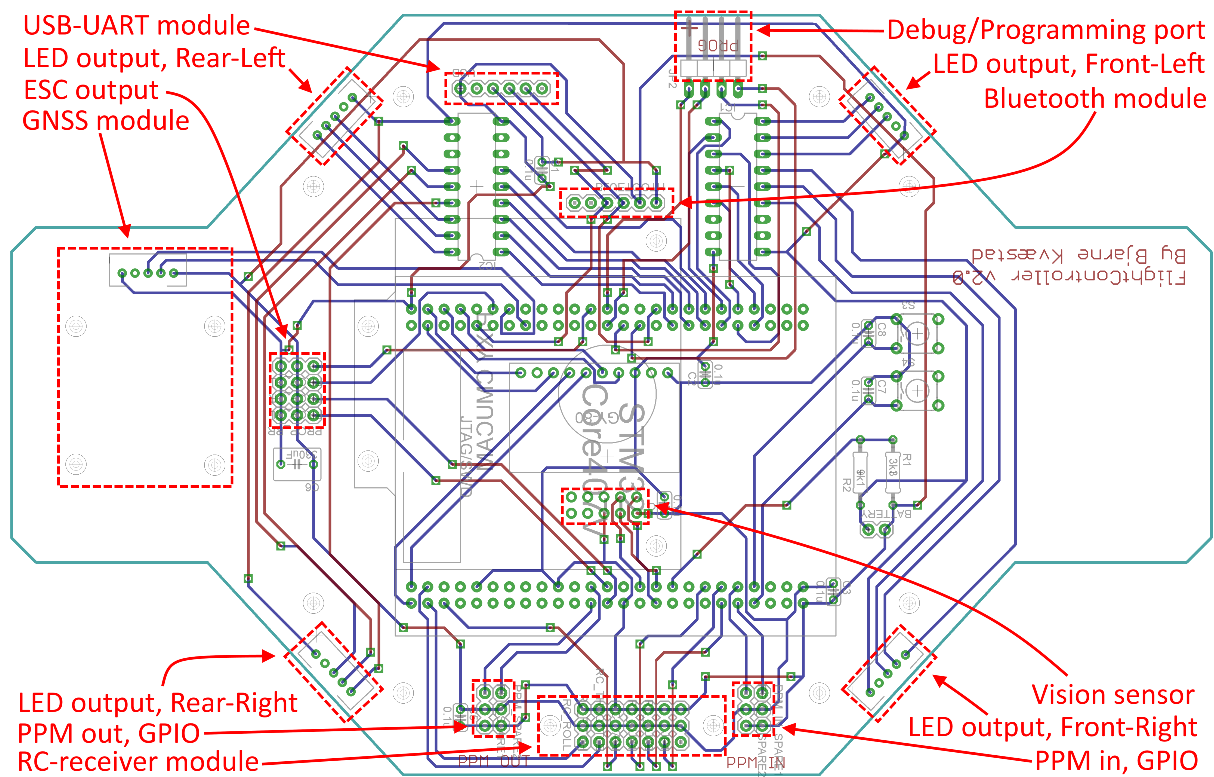


Figure 4.7: PCB layout with port labels

The schematics and PCB was developed in Cadsoft Eagle, and milled with NTNU's milling machine. The finished PCB with all the components soldered on can be seen in figure 4.8.

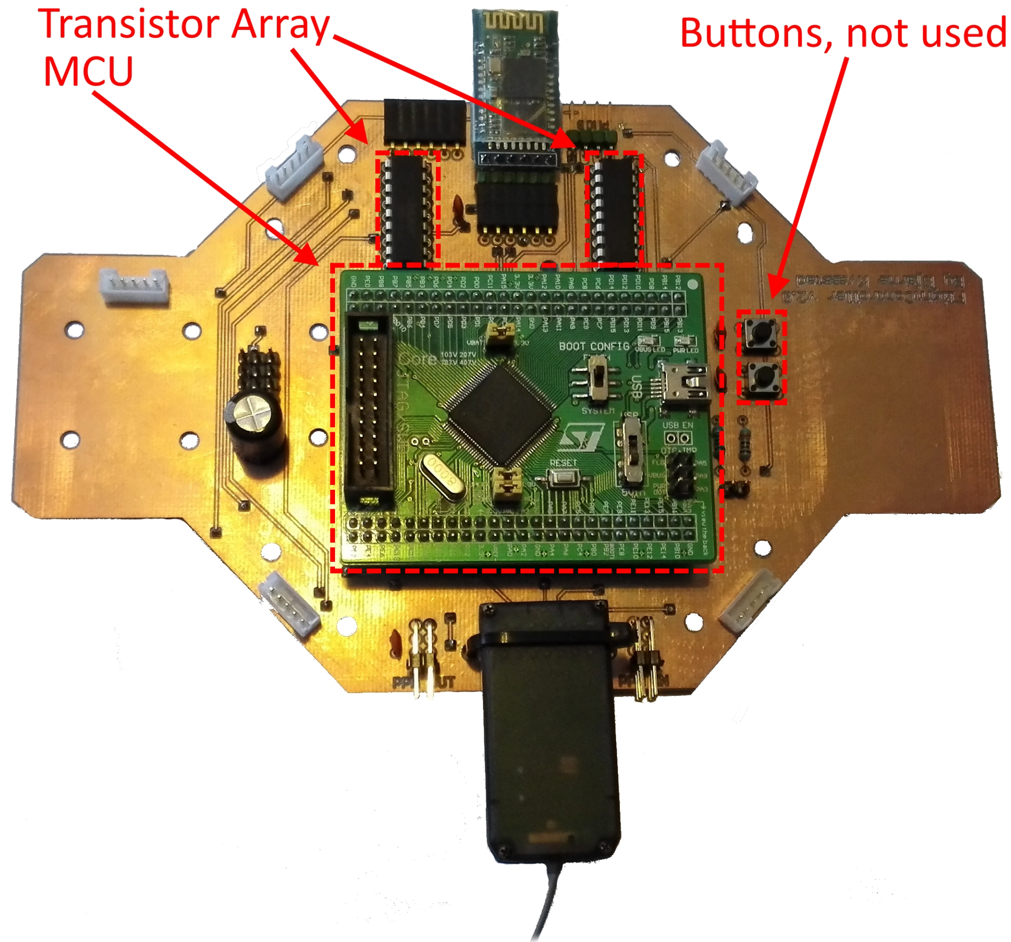


Figure 4.8: Finished PCB

4.6 IR sensor

When the drone is flying in autonomous mode it is preferred that the drone automatically disarms itself after landing, but due to varying terrain the altitude acquired from the pressure sensor is not optimal to use when the drone is landing. The initial altitude is sampled where the drone is at startup, and is considered as $z = 0$ m, it is not realistic to assume that the drone will always land at the same height as its initial position.
For this reason an IR sensor from Sharp is used, this can measure a range from 10 to 80 cm, using a analog output interface [3], a picture of the IR (infrared) sensor can be seen in figure 4.9.

Figure 4.9: IR sensor

The IR sensor is mounted under the PCB, pointing downwards, as shown in figure 4.2. The sensor is connected to one of the spare GPIO ports, shown in figure 4.7.

Some recalculation is needed to get the data measured in the correct format, in this case in centimeters. By studying the output characteristics of the IR sensors, presented in figure 5 in the data sheet, see appendix E, a function can be derived with regards of the measured voltage on the ADC-port (analog to digital converter), see equation 4.3 trough 4.5.

$$V - V_0 = \frac{V_1 - V_0}{x_1 - x_0}(x - x_0) \tag{4.3}$$

$$V - 0.42 = \frac{2.12 - 0.42}{0.081 - 0.013}(x - 0.013) \tag{4.4}$$

$$V = 25x + 0.095 \tag{4.5}$$

Where V is the measured voltage calculated from the 12-bit ADC input, y_{ADC}. x is the inverse number of distance, and L is the distance in centimeters, see equation 4.6 and 4.7.

$$V = \frac{3.3\,\text{V}}{2^{12}} y_{ADC} \tag{4.6}$$

$$x = \frac{1}{L + 0.42} \tag{4.7}$$

By inserting 4.6 and 4.7 in 4.5 and solving the equation with regards to the distance L, we get

$$L = \frac{25}{\frac{3.3}{2^{12}} y_{ADC} - 0.095} - 0.42 \tag{4.8}$$

This sensor can also, on a later moment, be used for a collision detection system.

4.7 Object Pickup

For picking up an object a retractable hook should to be used, this design is easy and good enough for this prototype. The idea is to mount a 3D-printed hook on a servo, the hook will be lifted in an upwards position during takeoff and landing, so it does not come in conflict with the ground. When an object is in range the hook will swing down so the object can be hooked on, the object must be equipped with a loop for the hook to grab. When the object has been moved to the correct location the hook will be lifted, and the object will fall off. See figure 4.10.

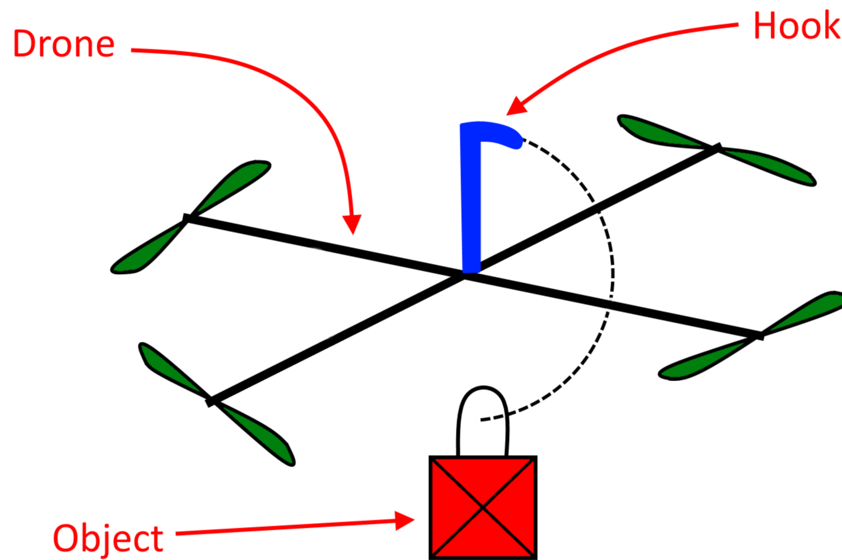


Figure 4.10: Hook setup

After doing some testing with the vision sensor the minimal range from the drone to the object is approximately 20 cm, where the whole object is still in the field of view of the camera. The hook should therefore be minimum 20 cm long, and the servo should have enough torque to counteract the inertia of the hook and object.

This hook was not implemented due to the issues explained in chapter 10.

Chapter 5

Position Data Processing

In order for the drone to control its position, a barometer and GNSS receiver is used. Mentioned in section 4.1 the GNSS-module takes measurements every 90 ms, this is also going to be the refresh rate for the position data processing algorithms. In this chapter we will process the measured data from the sensors to a noise reduced signal that can be used as reference for the position controllers, designed in chapter 6.

Figure 5.1 shows the different elements in the position data processing sequence, and the order they are executed in. Each element in this block diagram will be derived and explained in this chapter.

Screenshots of the Simulink implementation can be found in appendix C.

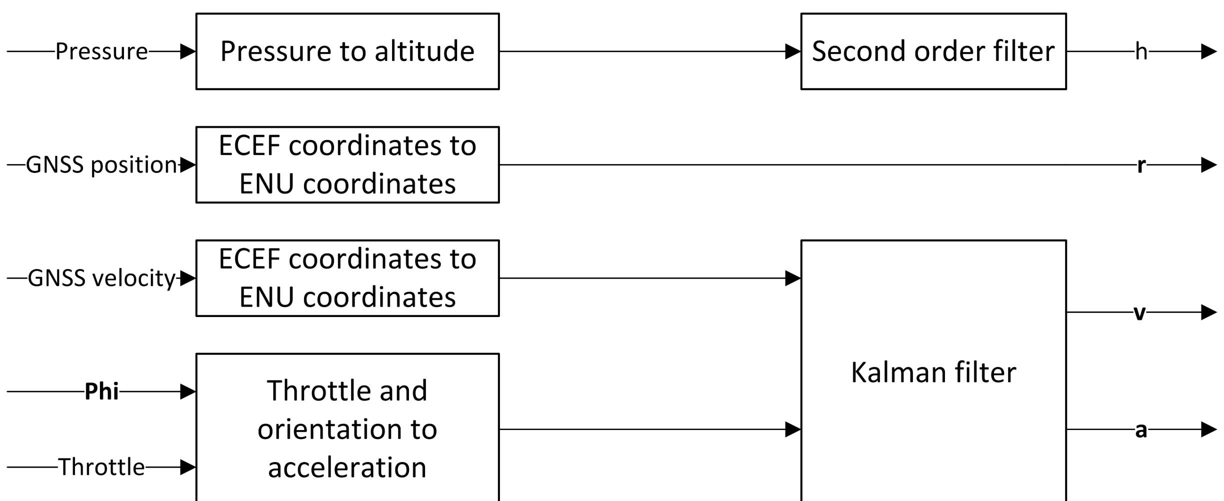


Figure 5.1: Position data processing overview

5.1 Raw Data Processing

In this section we will recalculate the raw sensor data to a format that is more suited to this purpose.

GNSS data

As mentioned in section 4.1, the data received from the GNSS module is in ECEF format, which is a coordinate system with its origin fixed at the center of the earth [19]. Controlling the drone with ECEF coordinates as reference is not practical as the z-axis points towards the North Pole. To make it easier to read the position of the drone in real time, ENU (east north up) coordinates are used.

The ENU format creates a local coordinate system where the drone takeoff point is the origin, x-axis points East, y-axis points North and z points up. The coordinate system is therefore a local tangent plane, measured in meters. This would also make testing easier because we will deal with smaller numbers. Function 5.2 shows the ECEF to ENU position transformation [13].

$$\begin{bmatrix} r_E \\ r_N \\ r_U \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\left(\frac{\pi}{2} - \mu_0\right) & \sin\left(\frac{\pi}{2} - \mu_0\right) \\ 0 & -\sin\left(\frac{\pi}{2} - \mu_0\right) & \cos\left(\frac{\pi}{2} - \mu_0\right) \end{bmatrix} \begin{bmatrix} \cos\left(\frac{\pi}{2} + l_0\right) & \sin\left(\frac{\pi}{2} + l_0\right) & 0 \\ -\sin\left(\frac{\pi}{2} + l_0\right) & \cos\left(\frac{\pi}{2} + l_0\right) & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_e - x_0 \\ y_e - y_0 \\ z_e - z_0 \end{bmatrix} \tag{5.1}$$

$$= \begin{bmatrix} -\sin l_0 & \cos l_0 & 0 \\ -\cos l_0 \sin\mu_0 & -\sin l_0 \sin\mu_0 & \cos\mu_0 \\ \cos l_0 \cos\mu_0 & \sin l_0 \cos\mu_0 & \sin\mu_0 \end{bmatrix} \begin{bmatrix} x_e - x_0 \\ y_e - y_0 \\ z_e - z_0 \end{bmatrix} \tag{5.2}$$

Where x_e, y_e and z_e are measured ECEF coordinates, and x_0, y_0 and z_0 are ECEF coordinates measured at the drone takeoff point. The function for transforming the ECEF velocity data to ENU coordinates is done by multiplying the velocity vector with the rotation matrix in function 5.2. See function 5.3.

$$\begin{bmatrix} v_E \\ v_N \\ v_U \end{bmatrix} = \begin{bmatrix} -\sin l_0 & \cos l_0 & 0 \\ -\cos l_0 \sin \mu_0 & -\sin l_0 \sin \mu_0 & \cos \mu_0 \\ \cos l_0 \cos \mu_0 & \sin l_0 \cos \mu_0 & \sin \mu_0 \end{bmatrix} \begin{bmatrix} \dot{x}_e \\ \dot{y}_e \\ \dot{z}_e \end{bmatrix} \tag{5.3}$$

l_0 and μ_0 are longitude and latitude of the takeoff point, the rotation matrix is therefore only calculated once. The ECEF to longitude transformation equation are presented in formula 5.4.

$$l = \text{atan2}\left(\frac{y_e}{x_e}\right) \tag{5.4}$$

And the algorithm for calculating the latitude from ECEF coordinates is presented in algorithm 5.1.

1. $p = \sqrt{x_e^2 + y_e^2}$
2. $\mu_0 = \dfrac{z_e}{p\left(1 - e^2\right)}$
3. $N_0 = \dfrac{r_e^2}{\sqrt{r_e^2 \cos^2 \mu_0 + r_p^2 \sin^2 \mu_0}}$
4. $h = \dfrac{p}{\cos \mu_0} - N_0$
5. $\mu = \tan^{-1}\left(\dfrac{z_e}{p}\left(1 - e^2 \dfrac{N_0}{N_0 + h}\right)^{-1}\right)$
6. If $\mu_0 \neq \mu$ then set $\mu_0 = \mu$ and continue from step 3. If $\mu_0 = \mu$ then μ is the final latitude.

Algorithm 5.1: Latitude algorithm [19]

Where r_e, r_p and e is the WGS-84 (World Geodetic System 1984) parameters, and are presented in table 5.1.

Table 5.1: WGS-84 paramteters [19]

Parameter	Value	Comments
r_e	6378137 m	Equatorial radius of ellipsoid
r_p	6356752 m	Polar axis radius of ellipsoid
e	8.181919e^{-2}	Eccentricity of ellipsoid

Digital pressure sensor data

The altitude is a function of a differential atmospheric pressure from a initial pressure, usually the ground pressure, see equation 5.5 [5].

$$h = 44330\left(1 - \left(\frac{p}{p_0}\right)^{\frac{1}{5.255}}\right) \tag{5.5}$$

Where p is the measured pressure in Pascal and p_0 is the initial pressure, measured at ground level. h is the altitude measured in meters.

5.2 Filtering

In chapter 3 it was concluded that the cascade controller design was the best choice, this design is particularly sensitive to a noisy input as it is two controllers connected in series. It is therefore very important that the position and velocity data are filtered properly, so the sensor noise is not amplified and cascading trough all the controllers, giving a noisy signal to the ESC's. This can cause a sluggish motor response.

In figure 5.2 the position and velocity data along the y-axis (North) is presented along with the barometer data, z-axis position (Up). The negative value between $t = 3$ and $t = 7$ in the first subplot is caused by the propellers creating a small over pressure at takeoff.

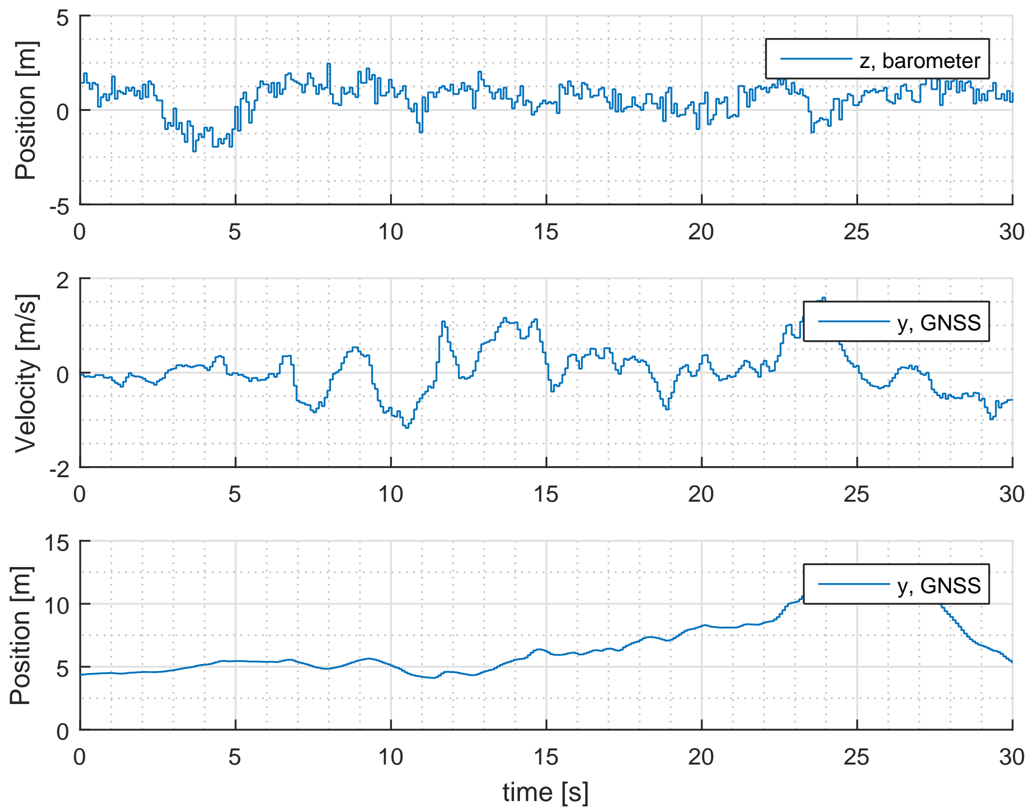


Figure 5.2: Position and velocity measurements

As seen from figure 5.2 there is only need to filter the pressure sensor data and the velocity data.

Velocity filtering

The velocity data was filtered using the same method as in the pre-project for the drone orientation data [17]. By using a kalman filter where velocity, drone orientation and motor throttle as input, we can create a noise reduced estimate of the velocity. The algorithm for the kalman filter is presented in equation 5.6 trough 5.10 [10].

$$\mathbf{K}_k = \mathbf{P}_k^- \mathbf{H}^T \left(\mathbf{H} \mathbf{P}_k^- \mathbf{H}^T + \mathbf{R} \right)^{-1} \tag{5.6}$$

$$\hat{\mathbf{x}}_k = \hat{\mathbf{x}}_k^- + \mathbf{K}_k \left(\mathbf{z}_k - \mathbf{H} \hat{\mathbf{x}}_k^- \right) \tag{5.7}$$

$$\mathbf{P}_k = \left(\mathbf{I} - \mathbf{K}_k \mathbf{H}_k \right) \mathbf{P}_k^- \tag{5.8}$$

$$\hat{\mathbf{x}}_{k+1}^- = \mathbf{A} \hat{\mathbf{x}}_k \tag{5.9}$$

$$\mathbf{P}_{k+1}^- = \mathbf{A} \mathbf{P}_k \mathbf{A}^T + \mathbf{Q} \tag{5.10}$$

The drone will use one kalman filter for each position component, x, y and z. The state space model of the correlations of the drone measurements are shown in equation 5.11 and 5.12.

$$\mathbf{x}_{k+1} = \mathbf{A} \mathbf{x}_k \tag{5.11}$$

$$\mathbf{z}_k = \mathbf{H} \mathbf{x}_k \tag{5.12}$$

Where

$$\mathbf{x}_k = \begin{bmatrix} \dot{\tilde{x}} \\ \ddot{\tilde{x}} \end{bmatrix} \qquad \mathbf{A} = \begin{bmatrix} 1 & \Delta t \\ 0 & 1 \end{bmatrix}$$

$$\mathbf{Z}_k = \begin{bmatrix} \dot{\tilde{z}} \\ \ddot{\tilde{z}} \end{bmatrix} \qquad \mathbf{H} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

Where Δt is the sample rate, 0.09 s, and $\tilde{z}$ is the measured data x, y or z, and $\tilde{x}$ is their estimates $\hat{x}$, $\hat{y}$ or $\hat{z}$. The acceleration, $\ddot{\tilde{z}}$, is calculated in function 5.13.

$$\begin{bmatrix} a_x \\ a_y \\ a_z \end{bmatrix} = \mathbf{R}_x^T(\phi) \mathbf{R}_y^T(\theta) \mathbf{R}_z^T(\psi) \begin{bmatrix} 0 \\ 0 \\ u \frac{4F_c}{m} \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ -g \end{bmatrix} \tag{5.13}$$

Where F_c is the force constant, found in the pre-project [17], m is the mass of the drone, g is the gravitational acceleration and u is the motor throttle.

The system- and noise variance matrices, **Q** and **R**, were found by using the noise variance of

the velocity and acceleration as base, a bit more tuning was needed. The matrices are presented in 5.14 and 5.15.

$$\mathbf{Q}_x = \begin{bmatrix} 0.001 & 0 \\ 0 & 3.113e^{-4} \end{bmatrix} \quad \mathbf{Q}_y = \begin{bmatrix} 0.001 & 0 \\ 0 & 3.488e^{-4} \end{bmatrix} \quad \mathbf{Q}_z = \begin{bmatrix} 0.117 & 0 \\ 0 & 6.844e^{-7} \end{bmatrix} \tag{5.14}$$

$$\mathbf{R}_x = \begin{bmatrix} 0.027 & 0 \\ 0 & 1.563e^{-4} \end{bmatrix} \quad \mathbf{R}_y = \begin{bmatrix} 0.005 & 0 \\ 0 & 1.508e^{-4} \end{bmatrix} \quad \mathbf{R}_z = \begin{bmatrix} 0.006 & 0 \\ 0 & 9.438e^{-5} \end{bmatrix} \tag{5.15}$$

Figure 5.3 shows the filtered velocity data from figure 5.2 using the kalman filter.

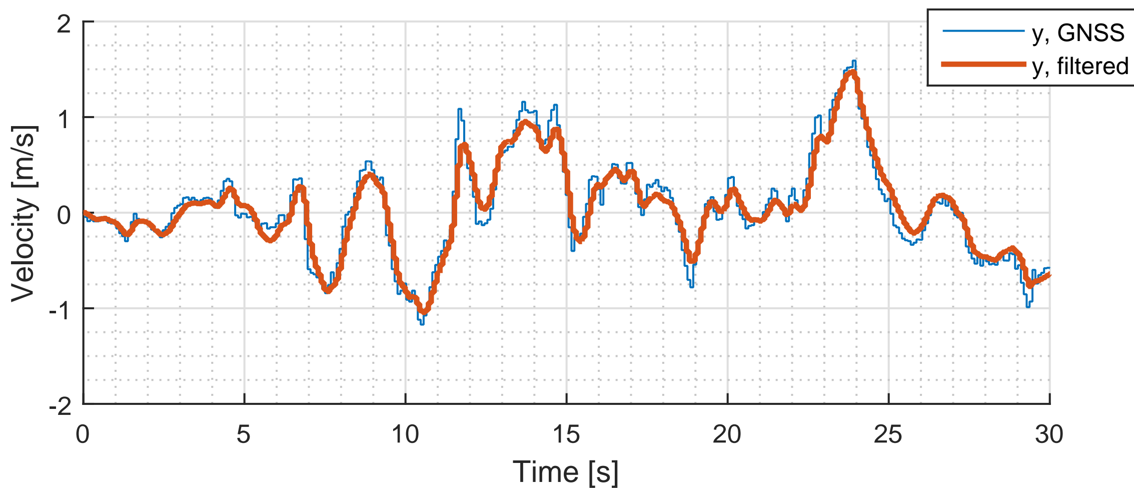


Figure 5.3: Filtered velocity measurements

The results show there are some small deviations of approximately ±0.1 m/s, which is good enough for our use. The velocity data is definitively noise reduced, and ready to be used in the position controller.

Altitude filtering

As previously mentioned the altitude, or the z-component, is measured with a digital pressure sensor, these measurements needs to be filtered. For a simple and efficient filtering, a second degree low-pass filter is used. The filter transfer function is presented in 5.16.

$$H(s) = \frac{Y(s)}{U(s)} = \frac{\omega^2}{s^2 + 2\zeta\omega s + \omega^2} \tag{5.16}$$

Where ω is the cut off frequency, and ζ is the damping factor. The filter differential equation is found by taking Laplace inverse on the transfer function, see 5.17 trough 5.19.

$$\mathscr{L}^{-1}\left\{s^2Y(s) + sY(s)2\zeta\omega + Y(s)\omega^2 = U(s)\omega^2\right\} \tag{5.17}$$

$$\ddot{y}(t) + \dot{y}(t)2\zeta\omega + y(t)\omega^2 = u(t)\omega^2 \tag{5.18}$$

$$\ddot{y}(t) = u(t)\omega^2 - \dot{y}(t)2\zeta\omega - y(t)\omega^2 \tag{5.19}$$

The differential equation 5.19 was implemented in Simulink, and simulated by inputting data previously recorded during a flight, the parameters, ω and ζ, were found by trial and error. The optimal parameter values are

$$\omega = 2.3 \qquad\qquad \zeta = 0.6$$

The output of this second degree low-pass filter can be seen in figure 5.4, where the barometer data from figure 5.2 was used as input.

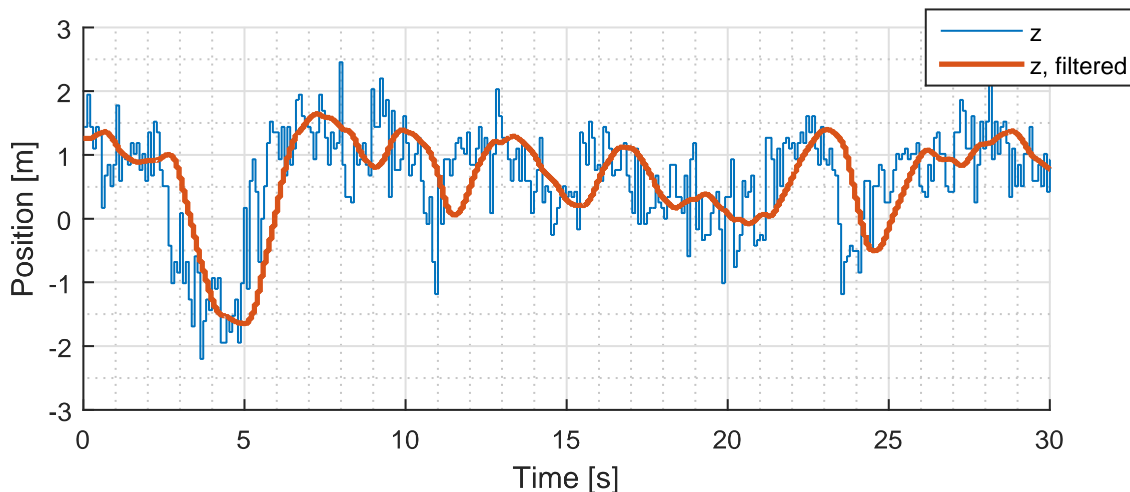


Figure 5.4: Filtered altitude measurements

As seen from the plot above the filtered altitude is delayed with about 0.5 seconds, this should not be a problem as long as the altitude controller parameters are not too aggressive.

Chapter 6

Position Controller

In order to make the drone autonomous the drone needs a position controller which uses the sensor data processed in chapter 5. As it is not straight forward to implement the position controller, this chapter will describe how to controller was implemented, along with the design and tuning of the position controller.

Screenshots of the Simulink implementation can be found in appendix C.

6.1 Implementation

When implementing the position controller the outputs (u) will be connected to the setpoint inputs on the orientation controller (r), but first the position controller outputs needs to be converted so that if the drone's heading is 0 (north) the position controller output gives the correct input to the orientation controller. E.g. a forward motion along the y-axis (north), would mean that the orientation controller must get a negative pitch (ϕ) setpoint, which is the first element in the orientation vector, $\Phi = \begin{bmatrix}\phi & \theta & \psi\end{bmatrix}^T$. By using this method of thinking and the right-hand rule we can derive a transformation matrix, see function 6.1.

$$\begin{bmatrix} r'_\phi \\ r'_\theta \\ r'_\psi \end{bmatrix} = \begin{bmatrix} 0 & -1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} u_x \\ u_y \\ r_\psi \end{bmatrix} \tag{6.1}$$

Where u_x and u_y is controller output for the x- and y-axis, and r is the setpoint for each attitude. This setup will only work if the drone is heading north, the amount of roll and pitch for reaching the setpoint depends on the current heading, we need to take this into account. We must therefore transform the controller output from inertial system to body system, we can do this by using the rotation matrix around the z-axis (yaw), see function 6.2.

$$\begin{bmatrix} r_\phi \\ r_\theta \\ r_\psi \end{bmatrix} = \begin{bmatrix} \cos\psi & \sin\psi & 0 \\ -\sin\psi & \cos\psi & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} r'_\phi \\ r'_\theta \\ r'_\psi \end{bmatrix} \tag{6.2}$$

By inserting 6.1 in 6.2 we get

$$\begin{bmatrix} r_\phi \\ r_\theta \\ r_\psi \end{bmatrix} = \begin{bmatrix} \sin\psi & -\cos\psi & 0 \\ \cos\psi & \sin\psi & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} u_x \\ u_y \\ r_\psi \end{bmatrix} \tag{6.3}$$

The controller outputs u_x and u_y are limited to -0.35 and 0.35 radians in order to avoid that the drone does a too steep roll or pitch.

The controller output for altitude, u_z, is connected directly to the propeller output together with the orientation controller outputs, see function 6.4.

$$u_t = u_z + u_o \tag{6.4}$$

Where u_t is the motor throttle, represented in equation 3.1 trough 3.4, and u_o is the take off throttle value, currently set to 285.

6.2 Controller

The main purpose of the position controller is to navigate using the filtered GNSS receiver- and altitude data as feedback. The position controller was designed in the same manner as the orientation controller, by designing and tuning the controller in Simulink, and auto generating C code for implementation in the MCU.

Two controller designs were tested, one PID controller for each axis, and two PID controllers for each axis in cascade configuration. The single PID controller performed so bad during simulation that no more time was used on this controller configuration. The bad performance makes sense since the drone will not slow itself down before it has passed the setpoint, making the drone oscillate around the setpoint. This could of coarse be damped enough by using a high derivative parameter, but that would also mean that the sensor noise would be amplified, and cascading trough the orientation controllers.

The cascade PID controller proved to give the best results, and is the configuration used in this project. The cascade controller for positioning works in the same manner as the orientation controller, explained in section 3.3. Where the secondary PID controller is controlling the drone's velocity and the primary PID controller is controlling the drone's position. See figure 6.1.

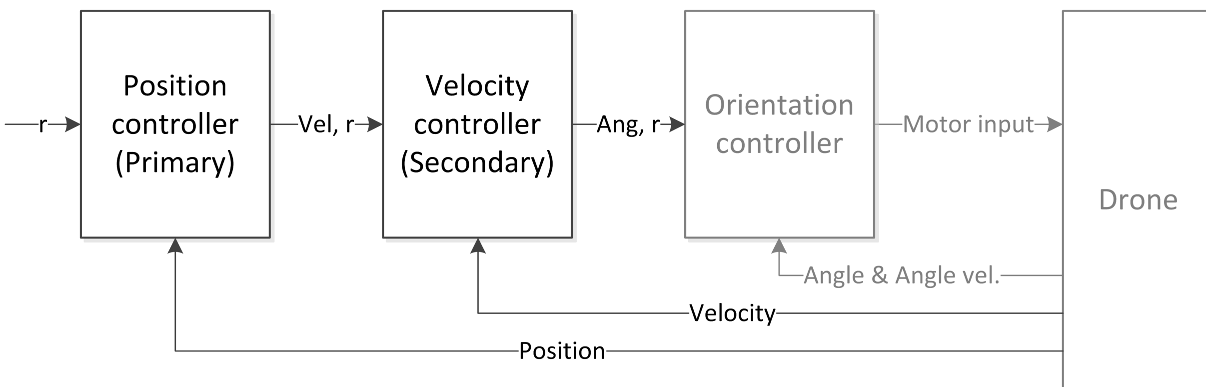


Figure 6.1: Cascade controller block diagram

The controller algorithms are the same used in the orientation controller, presented in 3.15 trough 3.19.

The position controller was tuned towards the drone model derived in the pre-project [17], the controller parameters found during this simulation were a rough estimate, and were a bit too aggressive when implemented on the drone. This due to the fact that the measurement delay in the GNSS-module and pressure sensor was not modeled, after a bit more tuning by trial and error, better parameters were found, see table 6.1.

Table 6.1: PID parameters

Axis	K_P	K_I	K_D
x position	0.45	0	0
y position	0.45	0	0
z position	0.28	0	0
x velocity	0.06	0.2	0.025
y velocity	0.06	0.2	0.025
z velocity	58	6	2

The control parameters were altered with a computer via Bluetooth when tuning the drone, explained in section 9.6.

Figure 6.2 shows a flight where the setpoint was set to 10 meters north ($y = 10$m), and 4 meters up ($z = 4$m). During this test there was a light breeze from west ($-x$), explaining the positive bias on the x-axis.

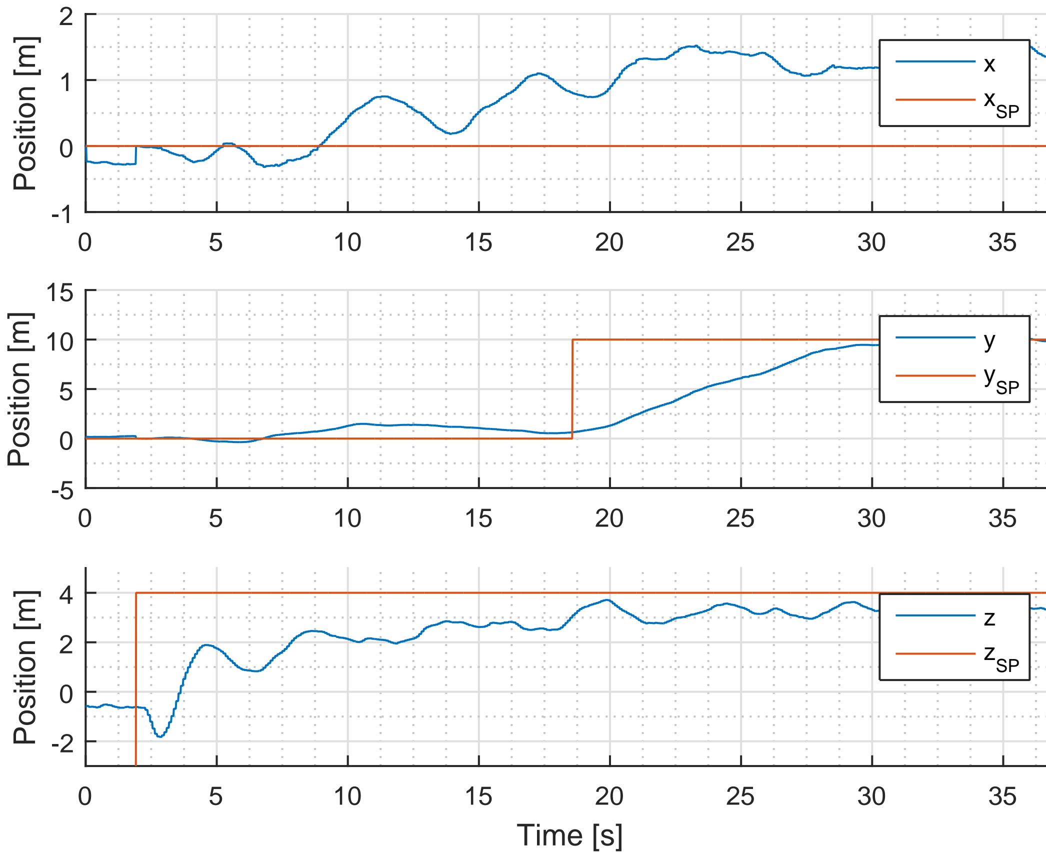


Figure 6.2: Drone position

As seen in the second subplot (y) in figure 6.3, the drone has trouble flying faster than 1 m/s, despite the high integrator parameter on the velocity controller. This issue is discussed in chapter 10.

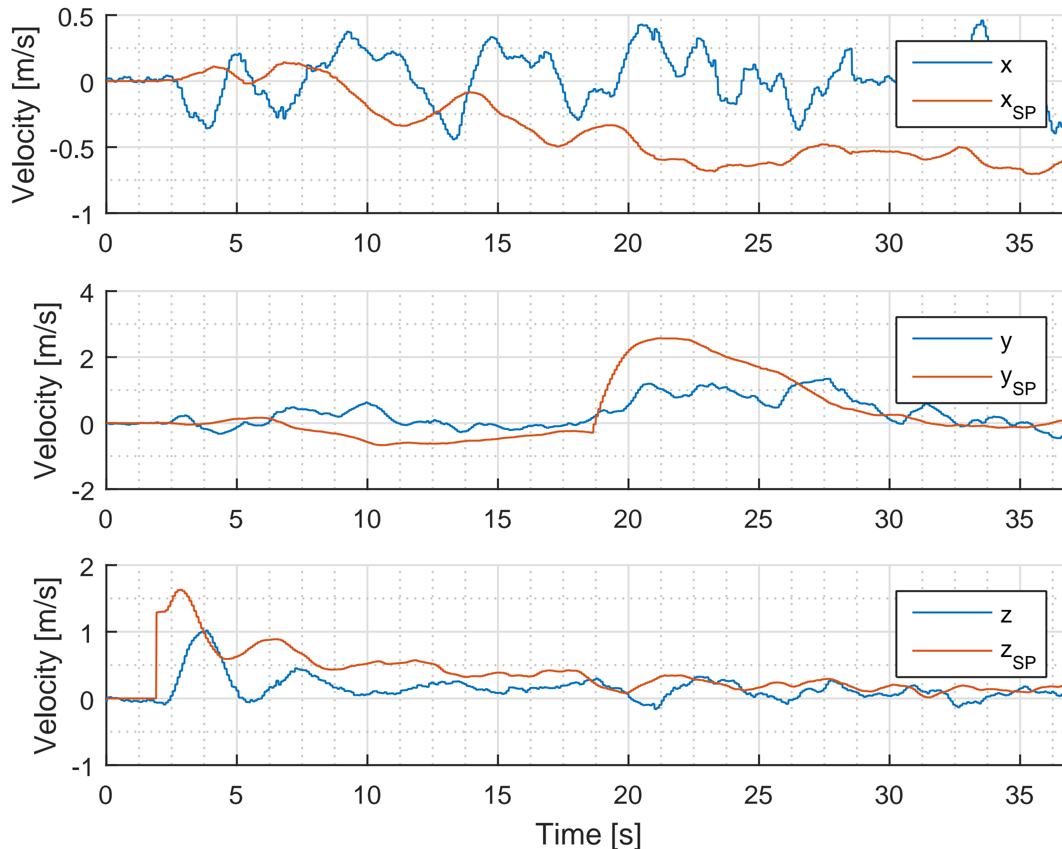


Figure 6.3: Drone velocity

Because of this issue the testing- and tuning conditions are quite straight as the testing must be outside for optimal GNSS reception, with little to no wind. When the weather conditions are good in Trondheim, which is rare, there is a small flying window of about 30 minutes before the batteries needs to be recharged. The controller parameters are therefore not optimal as there simply has not been enough time to tune the controller. The plots show however that the controller setup works as it should.

Chapter 7

Object Tracking

As mentioned in section 4.3 a Pixi CMUcam5 was chosen as the vision sensor for object detection, from this we get the height, width, x position and y position of the object in pixels, along with the current object angle in degrees. To make it easier to design and test the necessary algorithm for converting this data to x, y and z coordinates, the vision sensor is also going to be modeled in Simulink.

Screenshots of the Simulink code can be found in appendix C.

7.1 Data Processing

An optimal approach is to convert the data from the vision sensor to the same format as the position data processing algorithms, that way we can use the same position controller for the vision sensor data.

Object Position

We start by creating a simple sketch of the current setup with the drone and vision sensor, see figure 7.1.

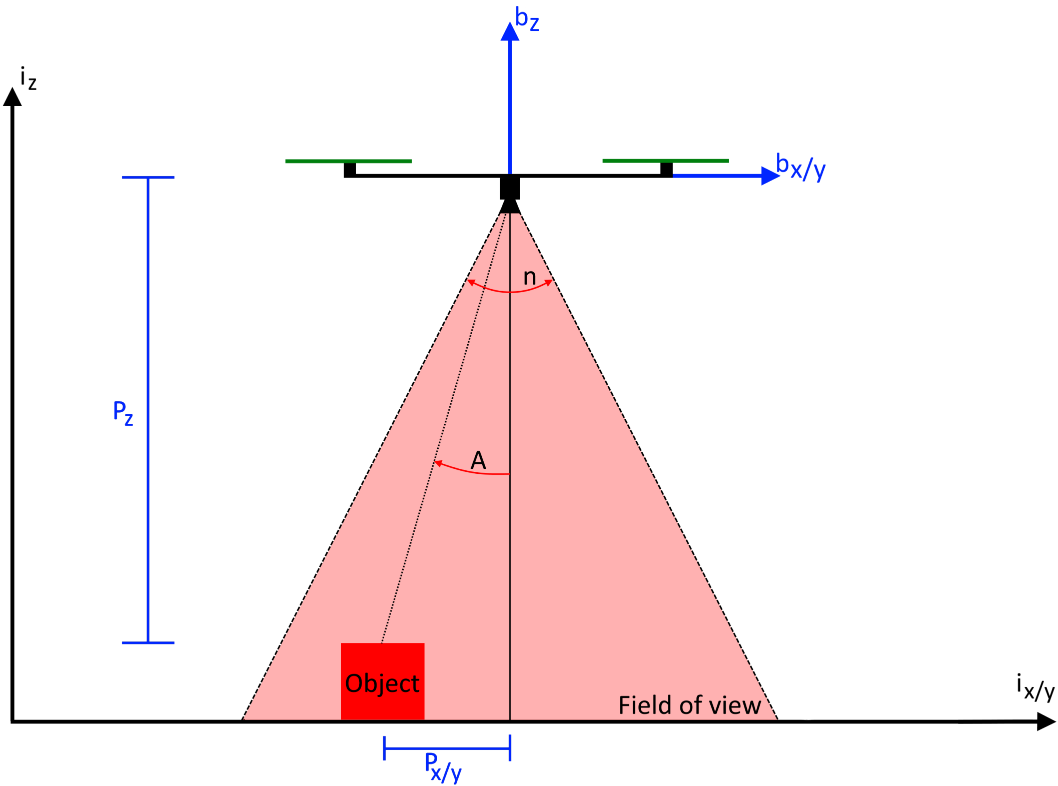


Figure 7.1: Vision sensor scetch

Where n is the field of view for the x- or y-axis, and P_x, P_y and P_z is the position of the object in body frame. The object angle, A, is calculated from the functions inspired from Randal Beard's paper on Quadrotor Dynamics and Control [9]. See function 7.1 and 7.2

$$A_x = \epsilon_x \frac{n_x}{\frac{M_x}{2}} \tag{7.1}$$

$$A_y = \epsilon_y \frac{n_y}{\frac{M_y}{2}} \tag{7.2}$$

Where M_x and M_y is the vision sensor resolution for the x- and y-axis, ϵ_x and ϵ_y is the object position in pixels. As seen in figure 7.2, ϵ_x and ϵ_y needs to be recalculated to body frame from the raw values ρ_x and ρ_y, where the origin (0,0) is in the top left corner in the field of view. See equation 7.3 and 7.4.

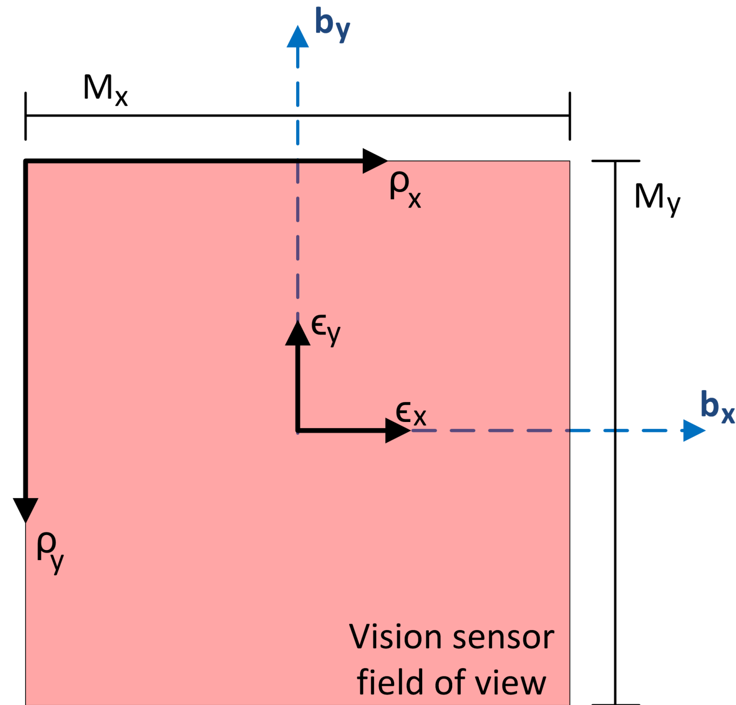


Figure 7.2: Vision sensor field of view

$$\epsilon_x = \rho_x - \frac{M_x}{2} \tag{7.3}$$

$$\epsilon_y = \frac{M_y}{2} - \rho_y \tag{7.4}$$

With this we can use basic trigonometry to calculate P_x and P_y, see function 7.5 and 7.6.

$$P_{x_b} = \tan(A_x) P_{z_b} \tag{7.5}$$

$$P_{y_b} = \tan(A_y) P_{z_b} \tag{7.6}$$

Where P_z can be calculated in the same manner as P_x and P_y, by using the vision sensors ability to measure the width of the object we can calculate how far away the object is if we know the true width of the object, see functions 7.7 and 7.8.

$$A_w = \rho_w \frac{n_x}{M_x} \tag{7.7}$$

$$P_{z_b} = \frac{P_w}{\tan A_w} \tag{7.8}$$

Where ρ_w is the object width in pixels. P_w is the real object width in meters, for this to work it is important that the object has a square surface, as the measured object width depends on the yaw ,ψ, of the drone.

Finally we can combine all the equations in this section and correct for the drone orientation with the rotation matrix, see function 7.9.

$$\mathbf{r}^i_{ob} = \mathbf{R}^i_b(\Phi)\mathbf{r}^b_{ob} = \mathbf{R}^i_b(\Phi)\begin{bmatrix} P_{x_b} \\ P_{y_b} \\ P_{z_b} \end{bmatrix} = \mathbf{R}^i_b(\Phi)\begin{bmatrix} \tan\left(\rho_x \frac{2n_x}{M_x} - n_x\right) \\ \tan\left(\rho_y \frac{2n_y}{M_y} - n_y\right) \\ 1 \end{bmatrix} \frac{P_w}{\tan\left(\rho_w \frac{n_x}{M_x}\right)} \tag{7.9}$$

As mentioned in section 4.3 the vision sensor resolution, M_x and M_y, is 320 and 200 respectively. The field of view, n_x and n_y, is $75\frac{\pi}{180}$ radians and $47\frac{\pi}{180}$ radians, respectively.

Object Angle

Since the drone is going to use a hook to pick up an object the angle difference between the drone and object (ψ_{ob}) matters, or else the hook will miss the handle mounted to the object. A small recalculation to the object angle acquired from the vision sensor needs to be done before it can be used in the controller, see function 7.10.

$$\psi_{bo} = -\psi_{ob}\frac{\pi}{180} \tag{7.10}$$

7.2 Modeling

In order to make testing easier the vision sensor is also going to be modeled, a block diagram was made in order to give an overview of the inputs and outputs of this model, see 7.3.

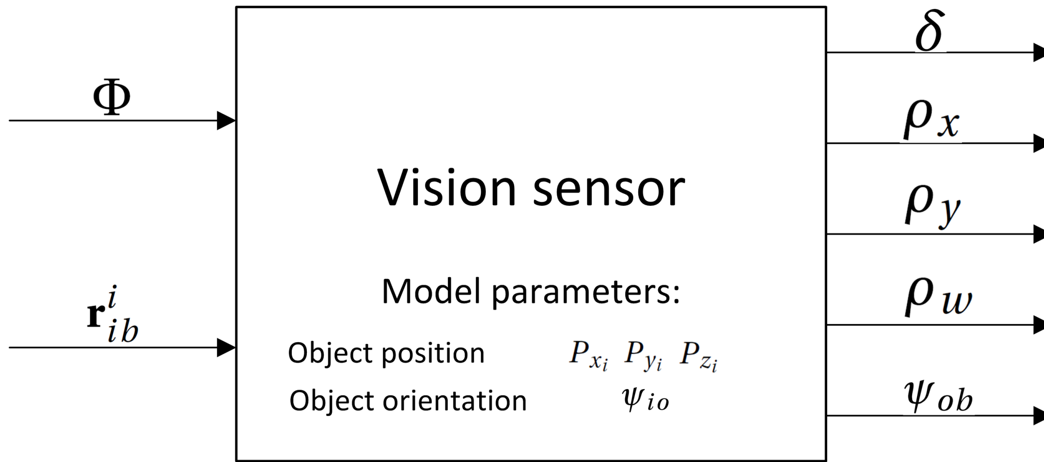


Figure 7.3: Vision sensor model block diagram

The notation of the symbols in this block diagram can be found in table 7.1.

Table 7.1: Notation

Symbol	Description
$\Phi = \begin{bmatrix}\phi_{ib} & \theta_{ib} & \psi_{ib}\end{bmatrix}^T$	Drone orientation vector
$\mathbf{r}^i_{ib} = \begin{bmatrix}x & y & z\end{bmatrix}^T$	Drone position vector
δ	Vision sensor object detected output
ρ_x	Vision sensor object position in x-axis in pixels.
ρ_y	Vision sensor object position in y-axis in pixels.
ρ_w	Vision sensor object width in pixels.
ψ_{io}	Vision sensor object orientation in degrees

Dynamics

For the model dynamics we can reverse the data processing algorithms 7.9 and 7.10 and add a floor function, see 7.11 trough 7.13.

$$\rho_x = \left\lfloor \frac{M_x}{2n_x}\left(\tan^{-1}\left(\frac{P_{x_b}}{P_{z_b}}\right) + n_x\right)\right\rfloor \tag{7.11}$$

$$\rho_y = \left\lfloor \frac{M_y}{2n_y}\left(\tan^{-1}\left(\frac{P_{y_b}}{P_{z_b}}\right) + n_y\right)\right\rfloor \tag{7.12}$$

$$\rho_w = \left\lfloor \frac{M_x}{n_x}\tan^{-1}\left(\frac{P_w}{P_{z_b}}\right)\right\rfloor \tag{7.13}$$

Where

$$\mathbf{r}^b_{ob} = \mathbf{R}^b_i(\Phi)\mathbf{r}^i_{ob} \tag{7.14}$$

$$\begin{bmatrix} P_{x_b} \\ P_{y_b} \\ P_{z_b} \end{bmatrix} = \mathbf{R}^b_i(\Phi)\left(\begin{bmatrix} P_{x_i} \\ P_{y_i} \\ P_{z_i} \end{bmatrix} - \mathbf{r}^i_{ib}\right) \tag{7.15}$$

The modeled angle, ψ^b_{ob}, is presented in function 7.17.

$$\psi_{ob} = \left\lfloor -\psi_{bo}\frac{180}{\pi} \right\rfloor \tag{7.16}$$

$$\psi_{ob} = \left\lfloor (\psi_{io} - \psi_{ib})\frac{180}{\pi} \right\rfloor \tag{7.17}$$

Logic

Obviously the vision sensor will not be able to track the object if it is out of line of sight, we therefore need some logic in our model to set the "object detected" output, δ. See function 7.18.

$$\delta = \begin{cases} 0, & \textbf{if } \rho_x < 0 \textbf{ or } \rho_x > M_x \\ 0, & \textbf{if } \rho_y < 0 \textbf{ or } \rho_y > M_y \\ 0, & \textbf{if } \rho_w \leq 1 \textbf{ or } \rho_w > M_x \\ 1, & \textbf{else} \end{cases} \tag{7.18}$$

Chapter 8

Flight Controller

The flight controller is one Simulink subsystem that includes the orientation controller, position controller, orientation data processing, position data processing and object data processing, all five modules are already mentioned in previous chapters, see figure 8.1.

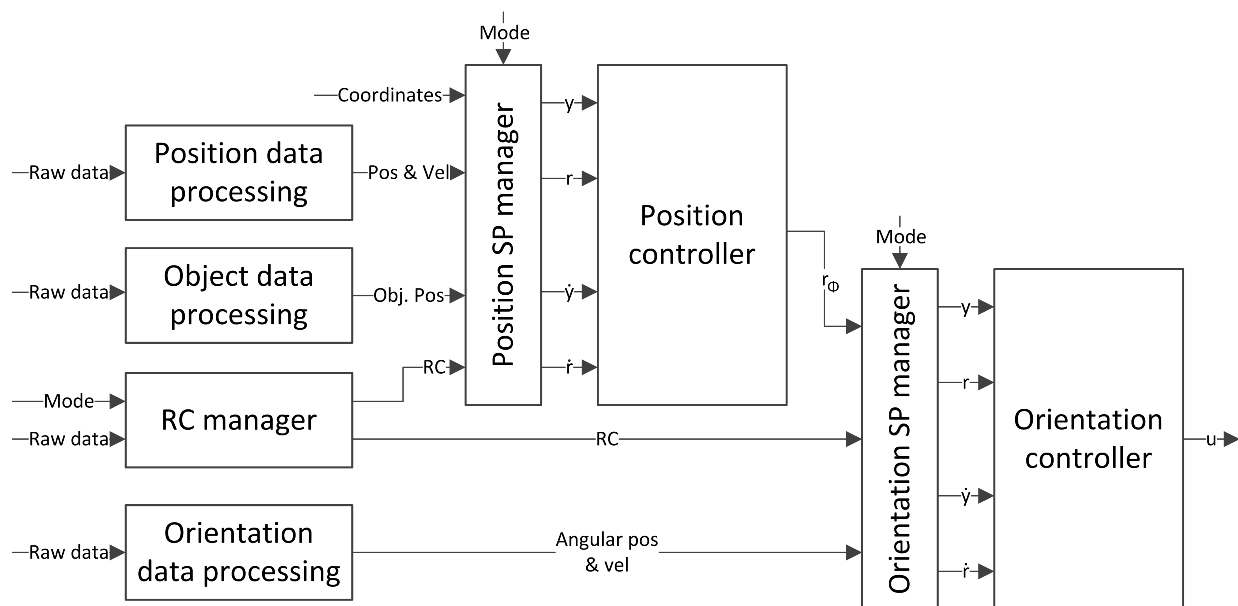


Figure 8.1: Flight controller block diagram

The subsystem also implements flight modes and all the necessary logic to support the modules under one Simulink block, this makes it easier to implement on the drone by using the Simulink code generator feature. This also gives an advantage of testing towards the drone model before implementing the flight controller on the actual drone, reducing the chance of unwanted

situations during flight, like bugs and programming errors.

In this chapter we will talk about the functionality of each flight mode, and also the battery monitoring system, which is not a part of the same subsystem as the rest of modules, but implemented directly on the MCU.

Screenshots of the Simulink implementation can be found in appendix C.

8.1 Modes

Different flight modes were implemented to make testing easier and safer. When testing an autonomous drone it is important to have the possibility to switch to manual mode during flight if the drone where to fly away. The flight controller was programmed with four different flight modes, by implementing more features on each mode will a full scale test be more manageable. It is possible to change between modes by using the auxiliary switches on the RC remote. The four modes and its functions are presented in table 8.1.

Table 8.1: Flight modes

No.	Control source	Name	Functions
0	RC remote	Manual mode	• Control angular position, Roll (θ), Pitch (ϕ) • Control angular velocity, yaw ($\dot{\psi}$) • Throttle is controlled directly
1	RC remote	Manual GNSS-mode	• Control angular position, Roll (θ), Pitch (ϕ) • Control angular velocity, yaw ($\dot{\psi}$) • Control altitude velocity ($\dot{z}$) • Position hold
2	Bluetooth	Autonomous mode	• A flight plan is loaded to drone pre-flight • The drone will fly to loaded waypoints • Automatically takeoff/land
3	Bluetooth	Autonomous object mode	• Object coordinate is loaded to drone pre-flight • The drone will fly to object and pick it up • Deliver object at takeoff point

Manual mode

When using manual mode the position controller is disabled, and the drone's orientation is controlled from a RC remote, where roll and pitch is controlled directly, and yaw is controlled by setting the angular velocity. The throttle is controlled directly from the RC remote. In this mode the drone will never be able to hover stable in one geographic position when $\psi = \theta = 0$ because of wind and small deviations in sensors, motors and propellers, due to no geographic position feedback. It is possible to switch to this mode from all the other modes during flight. The flow chart in figure 8.2 shows the control sequence for this mode.

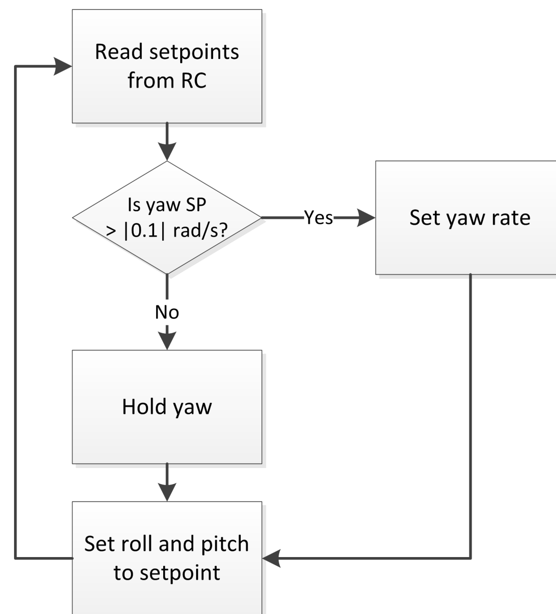


Figure 8.2: Manual mode flow chart

Manual GNSS-mode

In this mode the position controller is partly enabled, and using the RC remote to control the drone's orientation, roll, pitch and yaw, in the same manner as "manual mode". Unlike the "manual mode" throttle is now controlled by adjusting the ascend/descend rate, $\dot{z}$. When $r_\phi = r_\theta = 0$ the flight controller will lock on the current x and y position, and when $r_{\dot{z}} = 0$ the flight controller will lock on to the current z position. With this there is no need to do manual altitude

corrections when altering roll or pitch, as needed in "manual mode". It is possible to switch to this mode from "autonomous mode" and "autonomous object mode" during flight. The flow chart in figure 8.3 shows the control sequence for this mode.

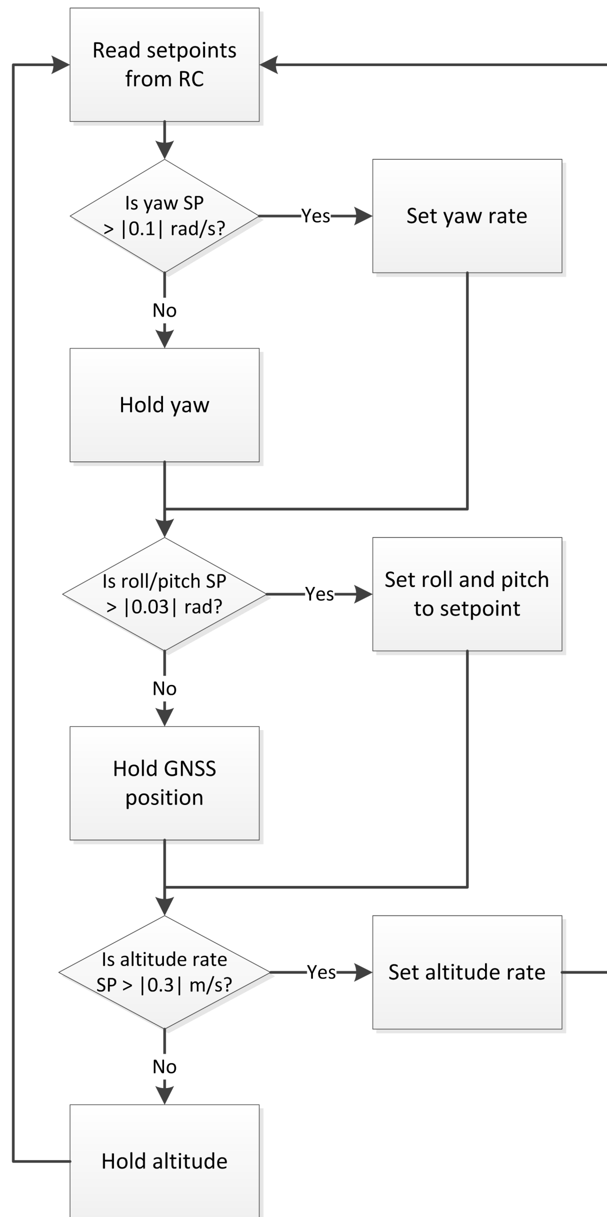


Figure 8.3: Manual GNSS-mode flow chart

Autonomous mode

In this mode the position controller is fully enabled. The flight plan, in ENU format, is uploaded to the drone pre-flight via Bluetooth, this mode can only be activated via a computer or a smart phone. The drone will automatically calculate the correct heading between coordinates, see function 8.1.

$$\psi'_{SP} = \text{atan2}\left(\frac{r_x - r_{x_{SP}}}{r_y - r_{y_{SP}}}\right) \tag{8.1}$$

In order for the drone to always correct its heading by taking the shortest path to ψ'_{SP}, we need to correct for the fact that drone heading can be more than π or less than $-\pi$, because of the rollover compensation, explained in section 2.1. See function 8.2.

$$\psi_{SP} = \begin{cases} 2\pi + n \cdot 2\pi + \psi'_{SP}, & \textbf{if } \left|\psi - n2\pi + \psi'_{SP}\right| > \pi \\ n \cdot 2\pi + \psi'_{SP}, & \textbf{else} \end{cases} \tag{8.2}$$

Where n is total rotations the drone has done around its z-axis, see 8.3.

$$n = \left\lfloor \frac{\psi}{2\pi} \right\rfloor \tag{8.3}$$

It is possible to cancel a flight mission during execution by changing to "manual mode" or "manual GNSS-mode". The flow chart in figure 8.4 shows the control sequence for this mode.

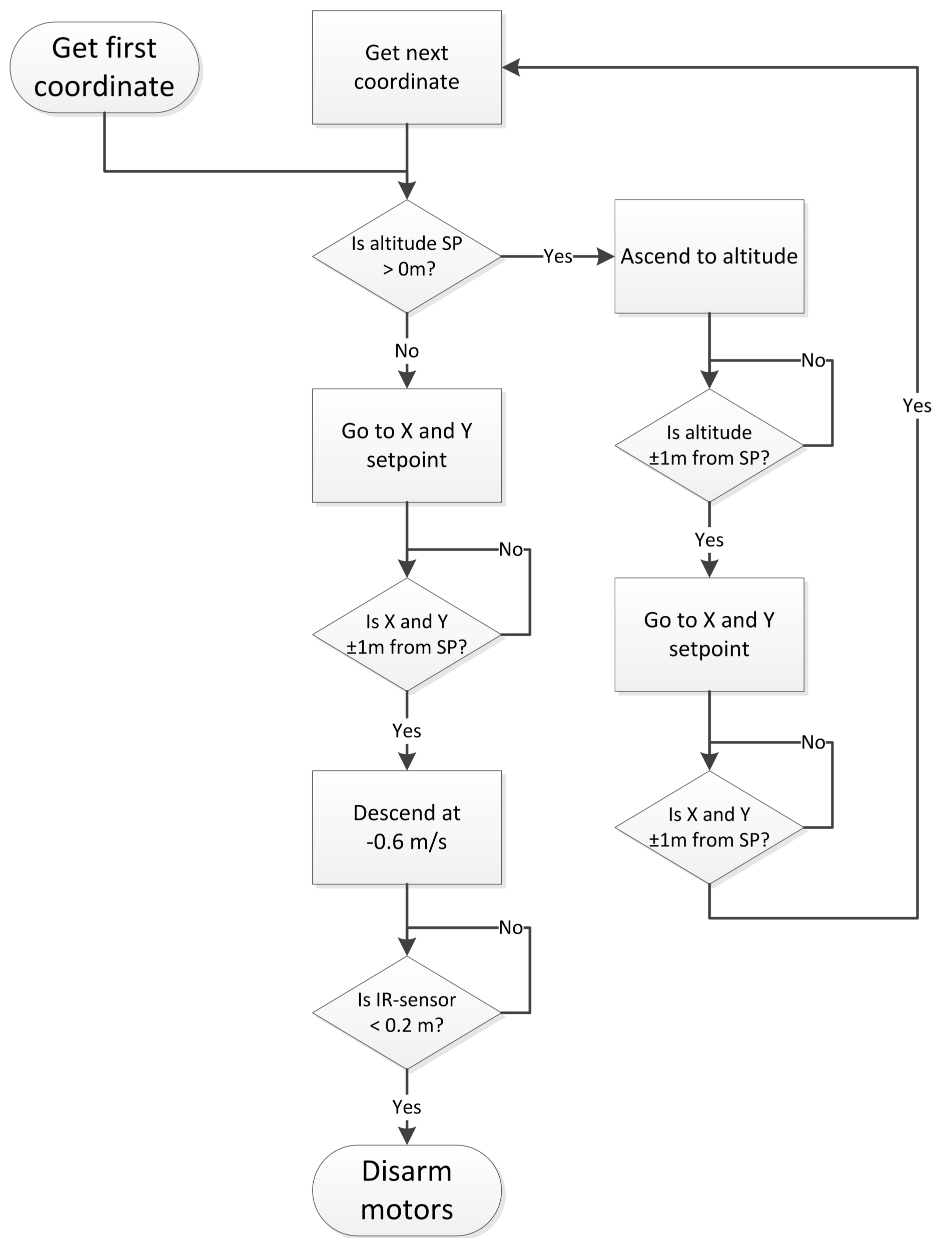


Figure 8.4: Autonomous mode flow chart

Autonomous object mode

In this mode the drone will be fully Autonomous, and use the GNSS-receiver to fly to a pre-loaded coordinate to search for an object using the vision sensor. When an object is detected the flight controller will automatically lock on to the position of the object and fly close enough for the drone to pick it up using the retractable hook, the correct yaw angle between coordinates

will be automatically calculated. When the object is picked up the drone will fly to the takeoff point to drop off the object. The flow chart in figure 8.5 shows the control sequence for this mode.

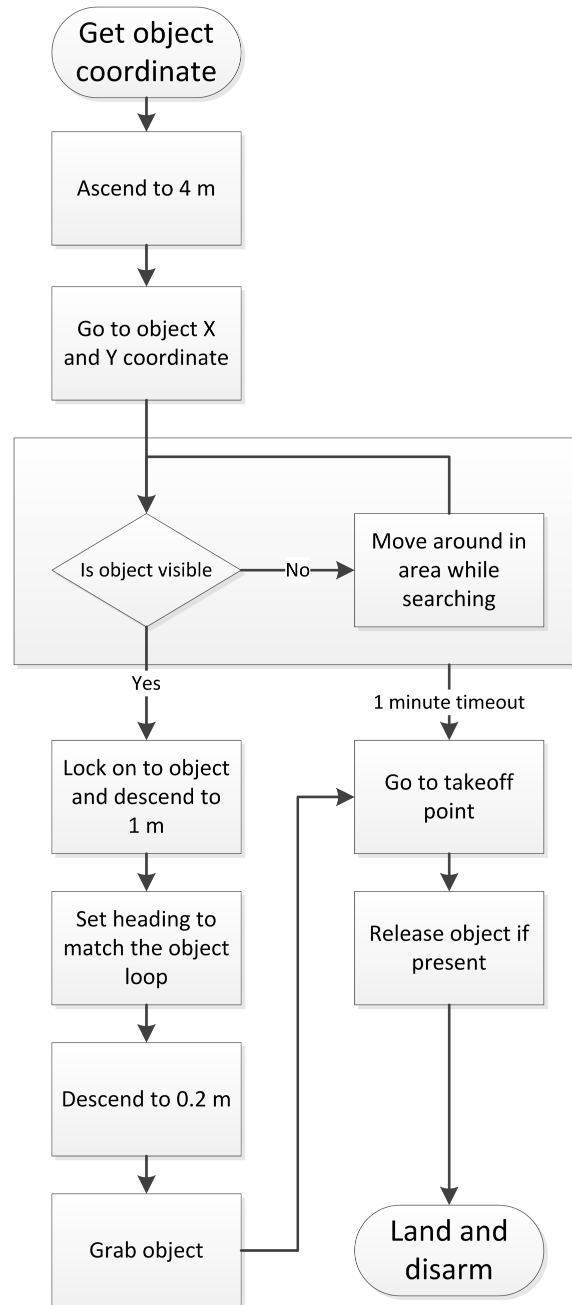


Figure 8.5: Autonomous object mode flow chart

It is always possible to switch to "manual mode" or "manual GNSS-mode" during flight.

8.2 Battery Monitoring

A battery monitoring system is important for the drone to be capable to warn the user of a low battery level, and automatically land by itself before the battery level gets too low for a soft landing. The ESC will automatically cut off power to the motors at 2.9 V per LiPo-cell (Lithium Polymer) [1], meaning a cut-off voltage of $3 \times 2.9 = 8.7$ V, because the batteries used has three cells. How the battery voltage is measured is explained in the pre-project [17].

Two battery level limits are defined, "Low" and "Low Low". Ideally we want the drone to land by itself if the battery is too low, a function implemented without much effort, but during this project this feature is not implemented so the battery capacity can be exploited.

Figure 8.6 shows a plot of the battery discharge during a bench-test and the battery level limits.

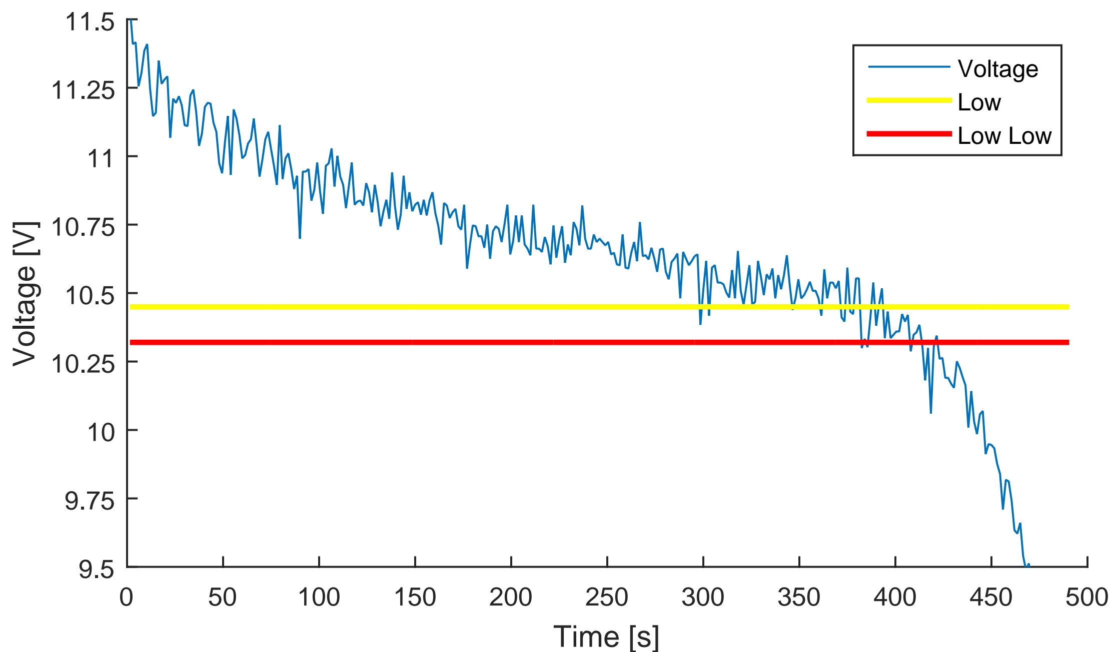


Figure 8.6: Battery discharge test, where $u_1 = u_2 = u_3 = u_4 = 355$

As seen from the figure above the "Low" limit is at about 10.45 V and "Low Low" at 10.32 V, the flight time left from "Low"-limit will be approximate 1.5 minutes and 0.5 minutes from "Low Low". The drone will signal "Low" and "Low Low" limits via the LED's, mentioned in section 4.4.

Chapter 9

Implementing a Real-time Operating System

If all the equipment and its functionality is going to be implemented and working together in a predictable manner, a RTOS (real-time operation system) is needed. This was also recommended in the "Future work" section in the pre-project due to some periodic tasks has a strict deadline.

There are many different RTOSs to chose from, but in this case FreeRTOS V. 8.2.1 was chosen due to the available support network online, many functionalities, small footprint and compatibility with the MCU [14]. An overview of each task and its functionality is listed in table 9.1.

Table 9.1: Tasks

Task	Type	Priority	Description
GNSS receive interrupt	Aperiodic	7	Receives GNSS data and controls checksum
Hardware initiate task	Sporadic	6	Initiate hardware
Flight controller task	Periodic, 15 ms	5	Executes all the necessary calculations to keep the drone airborne
Altitude acquisition task	Aperiodic	4	Acquire pressure sensor data and calculates the altitude
Serial receive task	Aperiodic	3	Receives commands sent via Bluetooth and processes it to instructions
Serial transmit task	Aperiodic	2	Sends data via Bluetooth
Battery monitoring task	Periodic, 1000 ms	1	Monitors the battery voltage
LED control task	Periodic, 500 ms	0	Controls the LED's for displaying current status

How these tasks communicates between each other is presented in the UML (unified modeling language) diagram, see figure 9.1.

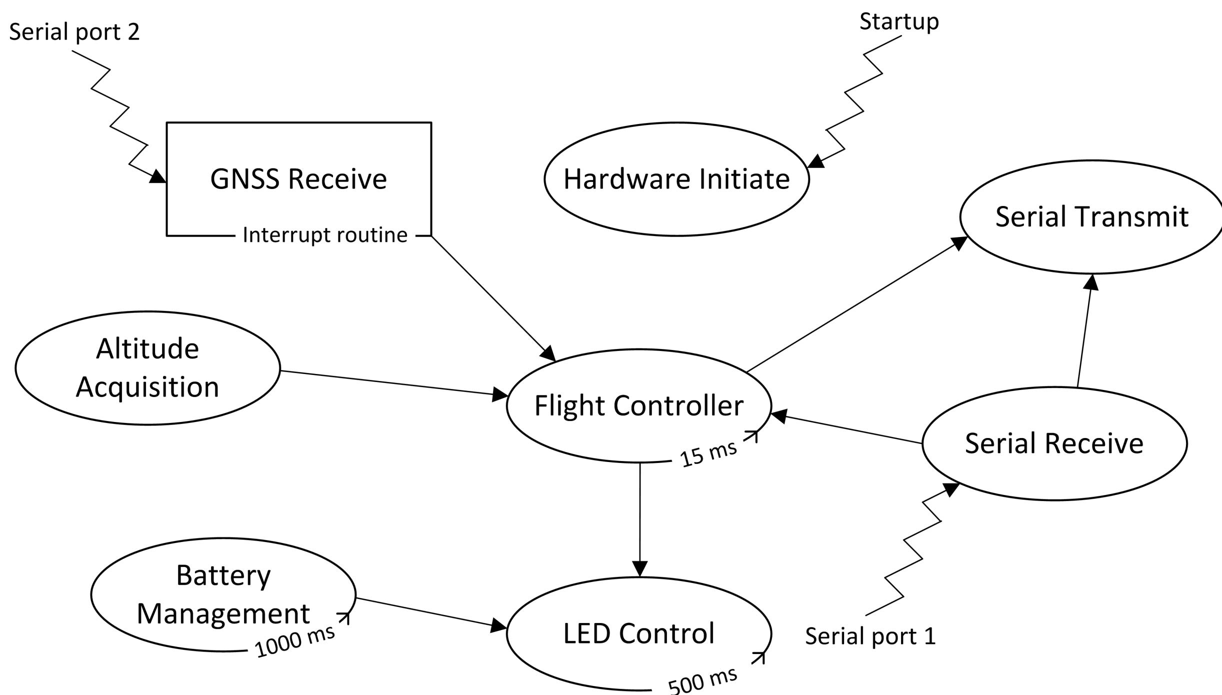


Figure 9.1: UML diagram

In order to synchronize the data flow between the tasks, functionalities such as mutexes, semaphores and queues are used.

The sections below provide a wider overview of how each task works. The software implementation of the tasks, written in C, can be found in appendix B.

9.1 GNSS Receive Interrupt Routine

Though this is not a task it still serves an important purpose. As mentioned in section 4.1 the data sent from the GNSS receiver is 60 bytes long, and is sent every 90 ms. With this knowledge we can create a routine that creates a interrupt when 60 bytes are received on the serial port. When a interrupt has been generated the microcontroller will check the received data and find the current GNSS status, but first the check-sum is controlled, see algorithm 9.1.

```
1  void checksum(data)
2      CKA = data[58]
3      CKB = data[59]
4      crcA = 0
5      crcB = 0
6
7      for i = 0; i < 58; i++ do
8          crcA += data[i]
9          crcB += crcA
10     end
11
12     if (crcA = CKA) && (crcB = CKB) then
13         checksum = OK
14     end
```

Algorithm 9.1: Check-sum controlling algorithm [4]

The process for finding the current GNSS status and conditions for using the received data is presented in the flow chart in figure 9.2.

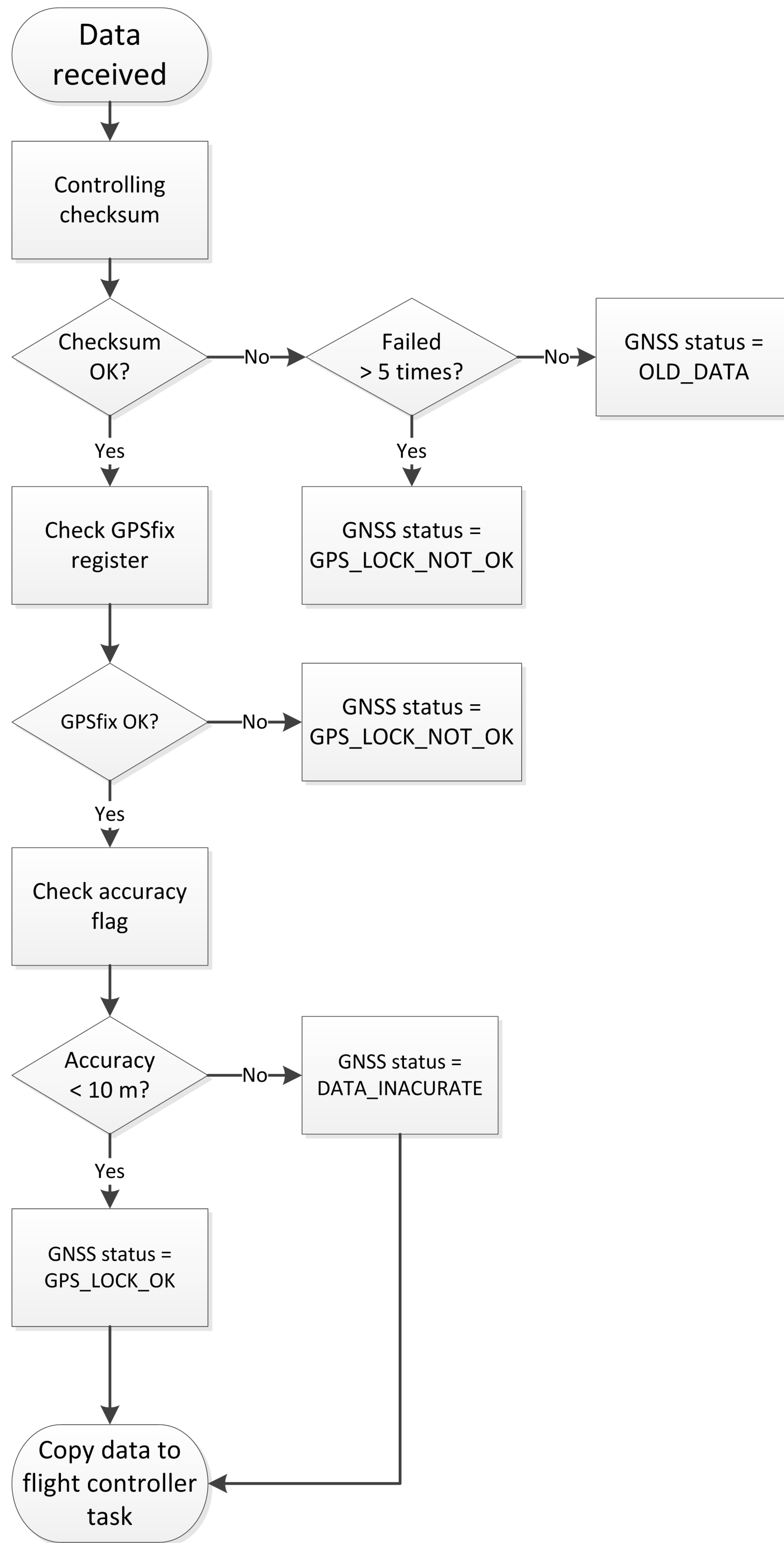


Figure 9.2: Interrupt routine flow chart

9.2 Hardware Initiate Task

The hardware initiate task is used to allocate memory and set the initial values for the flight controller, send the correct settings to the sensors and activate hardware interrupts. This task has the highest priority because no other task is allowed to run before this task is done, after the task is done it will delete itself, and the flight controller task will have the highest priority. A flow chart of this task is can be seen in figure 9.3.

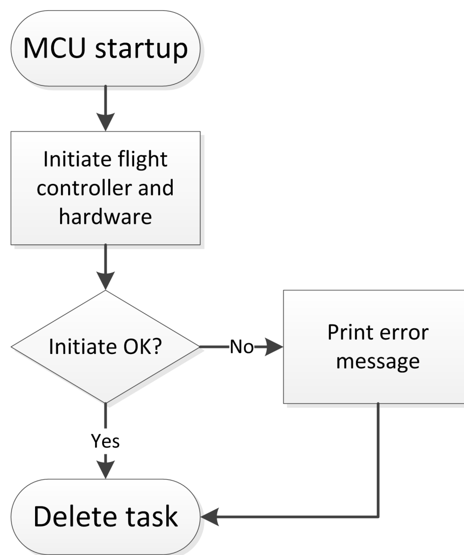


Figure 9.3: Hardware initiate flow chart

9.3 Flight Controller Task

As mentioned in chapter 8 the flight controller algorithm is auto generated code from Simulink, and contains one struct which contains all the input and output data, this struct is used as a parameter in a "step" function, which calculates the correct output from the data on the inputs. In this case the output is the motor throttle for each motor, this is therefore the most important task running, and has the highest priority after the hardware initiate task has been deleted. It is important that this task always runs in such manner that it executes the flight controller algorithm every 15 milliseconds, or else the drone will get unstable, and in worst case, crash. A flow chart of this task is can be seen in figure 9.4.

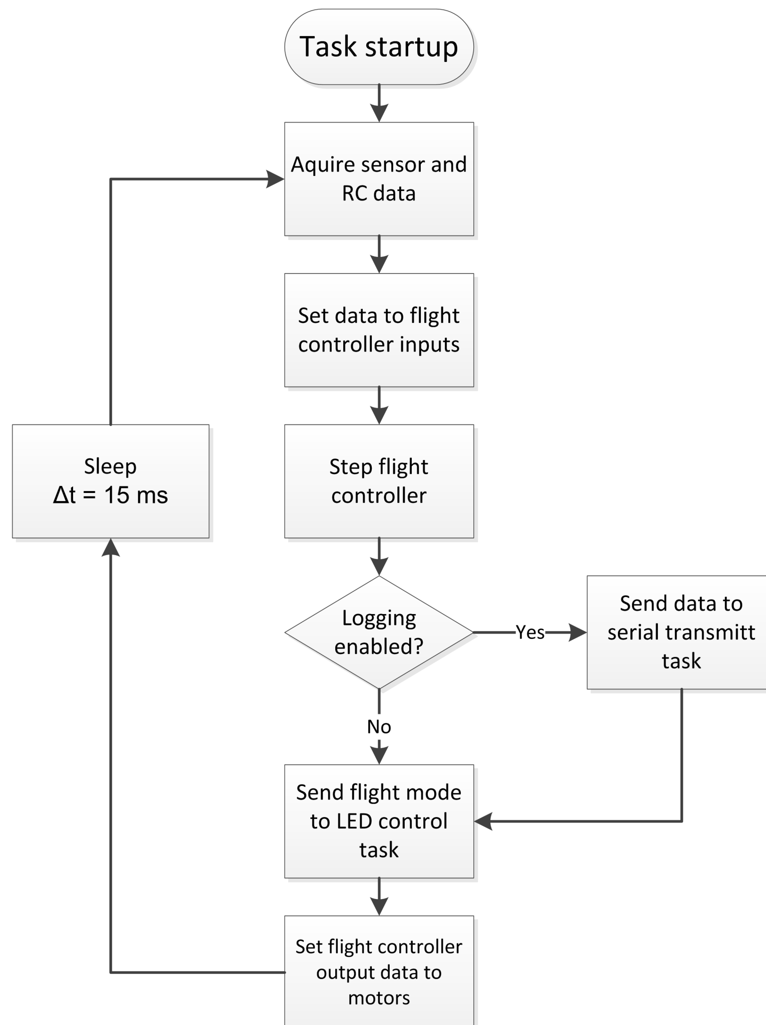


Figure 9.4: Flight controller flow chart

9.4 Altitude Acquisition Task

The altitude acquisition task's purpose is to retrieve and calculate the drone altitude. This is a separate task because it takes about 30 milliseconds to retrieve the pressure data, due to the processing rate in the pressure sensor [5]. The altitude data is needed approximately every 90 ms, because this is the rate the position controller is working at. The task is triggered by the flight controller task trough a semaphore, and the data is sent to the flight controller task trough a queue, see figure 9.5.

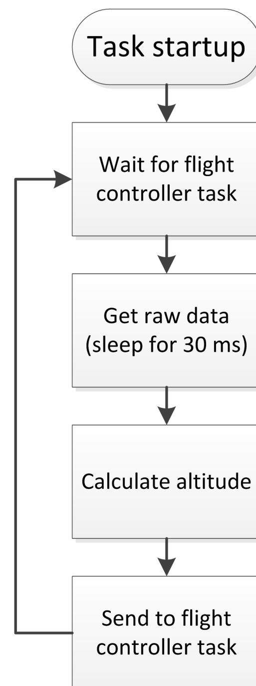


Figure 9.5: Altitude acquisition flow chart

9.5 Serial Transmit Task

The serial transmit task synchronizes the outgoing data traffic that is going to the serial port. This is done by using the FreeRTOS tool, queue, which is using a using a FIFO (first in first out) principle [6]. If the queue is empty the task will wait, utilizing more of the CPU to the other tasks, when some data has been put on the queue the thread will wake up and print the data, see figure 9.6.

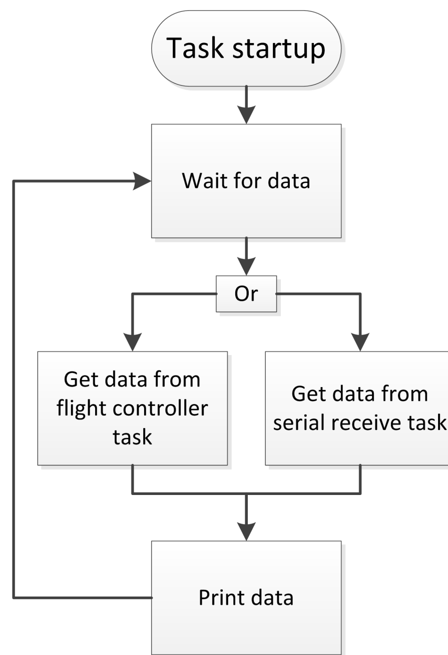


Figure 9.6: Serial transmit flow chart

9.6 Serial Receive Task

The serial receive task receives and processes the commands sent via Bluetooth, this is done by using a CLI (command line interface), a feature integrated in the RTOS [14], the command set is implemented by the user. A table of the commands implemented on the drone can be found in appendix F. The task will wait on a semaphore, this semaphore will be incremented by a serial receive interrupt routine, when a command is received the task will check if it is a known command, and process it accordingly, see figure 9.7.

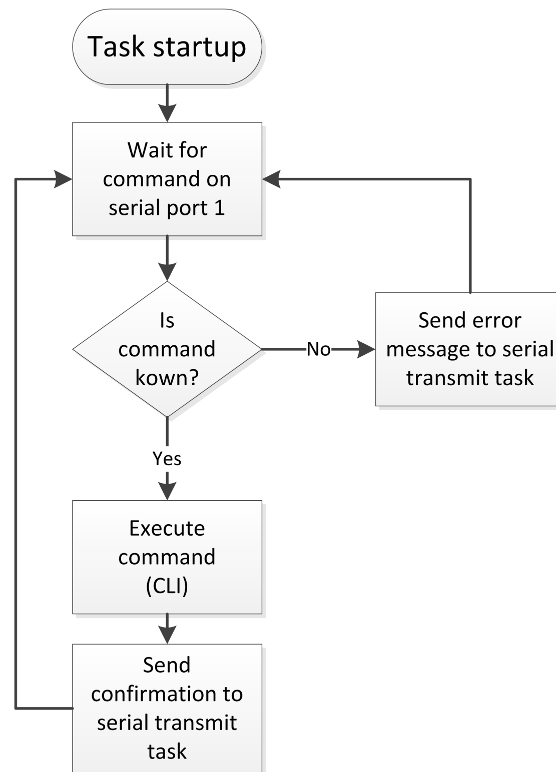


Figure 9.7: Serial receive flow chart

The C code, implementing the CLI is referred to from appendix B.

9.7 Battery Monitoring Task

The battery monitoring task is checking the battery voltage one time per second, if the battery voltage is low the task will send a command to the LED control task, so the user can get a visual representation about the battery status. How the battery voltage is checked is mentioned in the pre-project [17]. The task will define the battery status as "Low" or "Low Low" when five consecutive measurements are below a given battery limit. See figure 9.8.

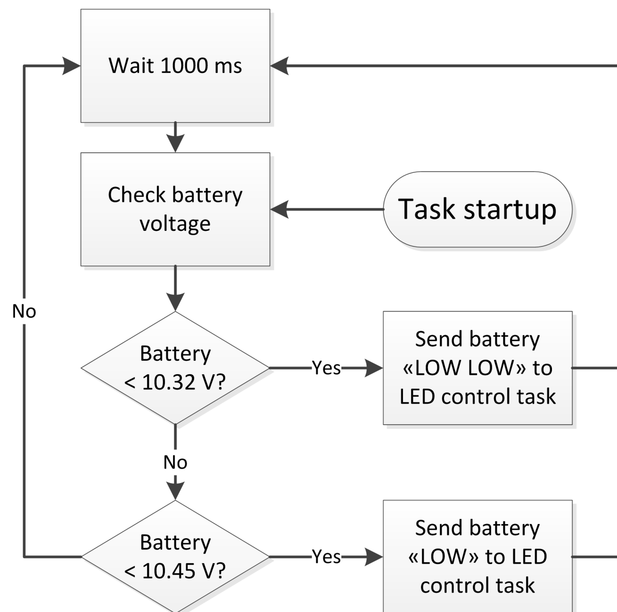


Figure 9.8: Battery monitoring flow chart

9.8 LED Control Task

The LED control task controls the LED strips connected underneath the drone, mentioned in section 4.4. Each flight mode, battery status and setup sequence has a different LED sequence, all the sequences are explained in appendix G. The task will wait for 500 milliseconds for a new LED sequence from the battery monitoring task or flight controller task, if no such sequence is received the task will continue on to maintain the LED blinking of the current LED sequence. See figure 9.9

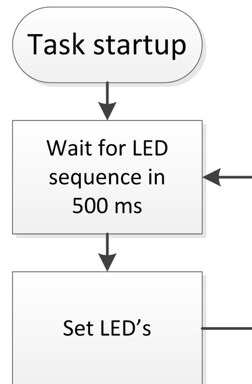


Figure 9.9: LED contror flow chart

Chapter 10

Full-scale Testing

Kristiansten Fortress in Trondheim was chosen to be the testing ground because of if its big open area and good GNSS reception. In this chapter we will test all flight modes, except the autonomous object mode, which is going to be simulated. As mentioned in chapter 6 the weather conditions for testing and flying were strict due to issues with the drone top speed, which is also going to be investigated in this chapter.

When flying the drone, sensor data, setpoints and controller outputs was logged via Bluetooth every 15 ms, by using MATLAB we can create good graphical representation of the data for further analyzing.

10.1 Manual mode

The objective with this mode is to be able to control the drone's orientation manually with the RC remote, the drone should always reach the setpoint efficiently and with minimal deviation, explained in detail in section 8.1.

To test this, the drone was flown normally outside where there is good space for testing hovering and speed flights.

The drone felt responsive and precise, this was also proved in chapter 6, though the drone was not able to hold the pitch or roll setpoint over longer time, as seen in figure 10.1. A video from this flight is referred to in appendix B named "manual mode".

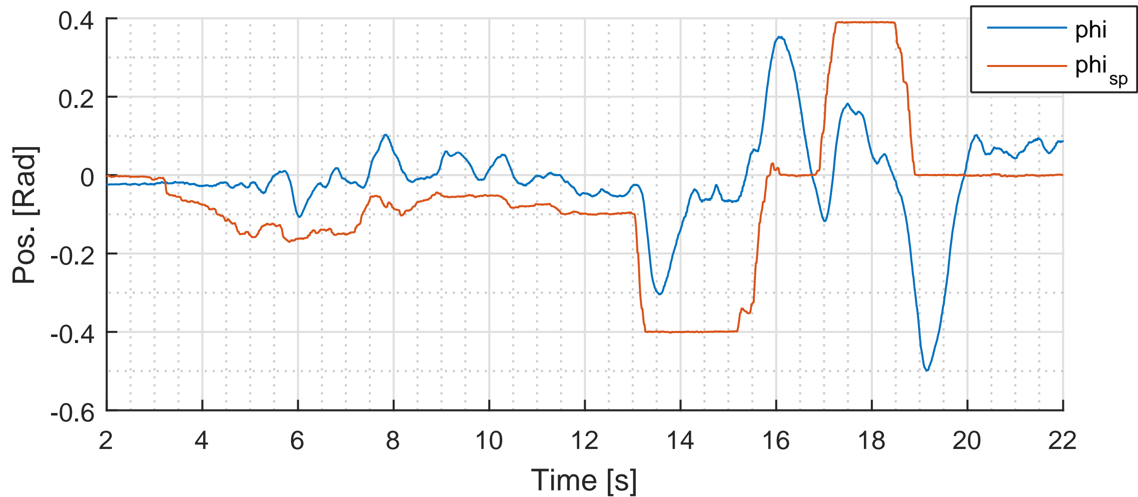


Figure 10.1: Pitch position

For simplicity we are only going to focus on the pitch motion, as the pitch and roll controllers have the same PID parameters.

Seen at $t = 13.5$, $t = 16$ and $t = 19$ the controller is very responsive to rapid setpoint change, it seems that the derivative part (D) of the controller is too big.

As seen at $t = 13$ the pitch setpoint is altered to -0.4 radians, and the drone seems to respond well, but when the drone is starting to get a forward momentum it flattens out, seen at $t = 14$. The reason why can be seen in the angle velocity plot, see figure 10.2.

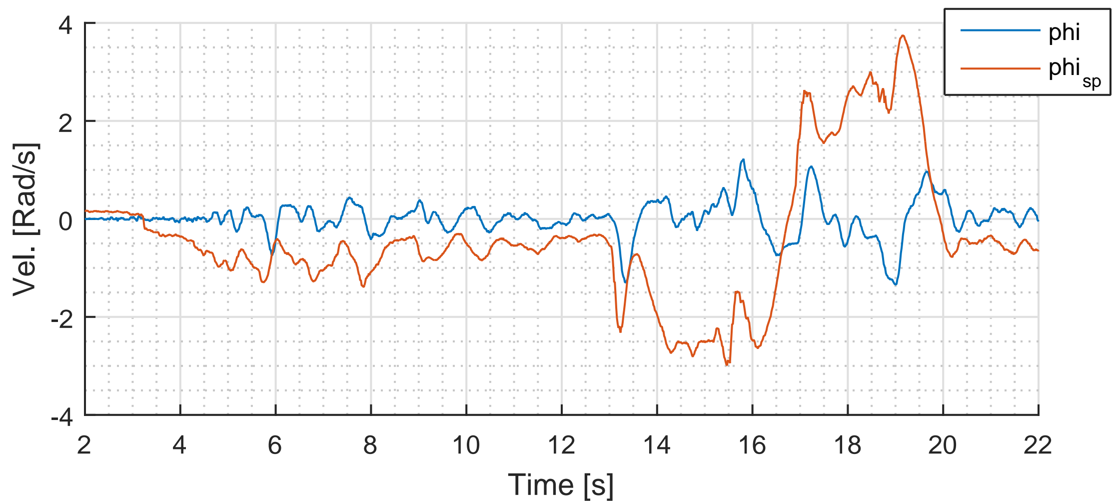


Figure 10.2: Pitch velocity

As mentioned in chapter 3 the angle position controller (primary controller) is controlling the

setpoint of the angle velocity controller (secondary controller). So in this case $i_{SP} = u_i$, where i is ϕ, θ and ψ.

When investigating figure 10.2 we can see that the there is not much response when there is an stationary error $e[k]$. We can track this issue further down to the orientation controller output, u, see figure 10.3.

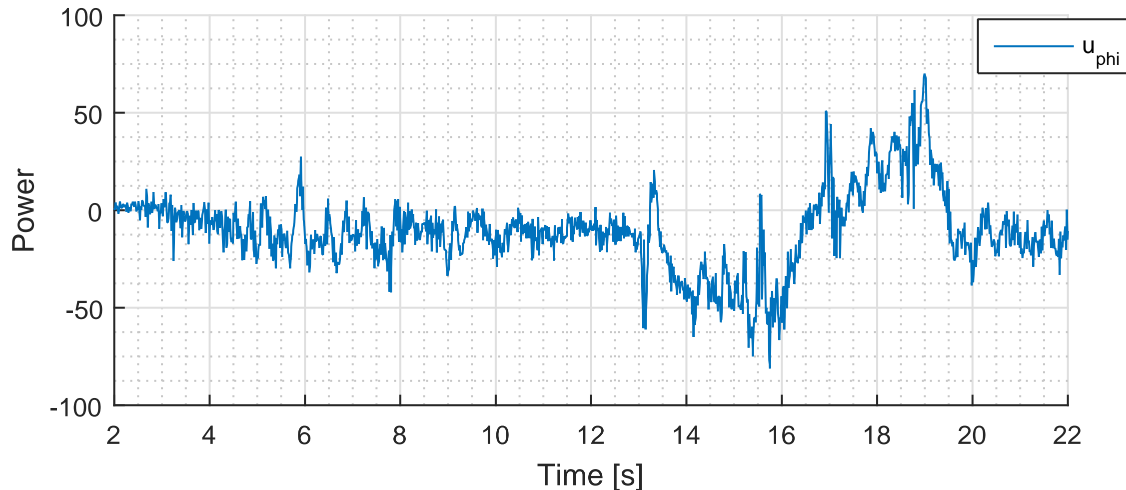


Figure 10.3: Pitch controller output

The setpoint change, $\phi_{SP} = -0.4$, happening at $t = 13$, and as seen in the plot the controller is giving a big negative impulse (due to the derivative part of the PID controller), and another big positive impulse about 250 milliseconds after, due to the negative error rate. As the drone flattens out at $t = 14$, controllers integral part is winding up, making the controller output increasing, see 10.3 from $t = 14$ to $t = 16$. The controller power output is approximately -50 (10% of the whole power area), which according to chapter 3 means -50 on front propellers and +50 on back propellers. In a hover scenario this is more than enough for the drone go to a desired orientation setpoint, but if it is in forward motion the propeller wash from the front propellers (in the direction of motion) might have an effect on the back propellers, making them loose lift, and the drone will flatten out.

In order to test this, the drone was tied down, it could only rotate along its x axis (pitch), removing the chance of propeller wash. A video of this bench test is referred to in appendix B, named "manual mode bench test". See figure 10.4.

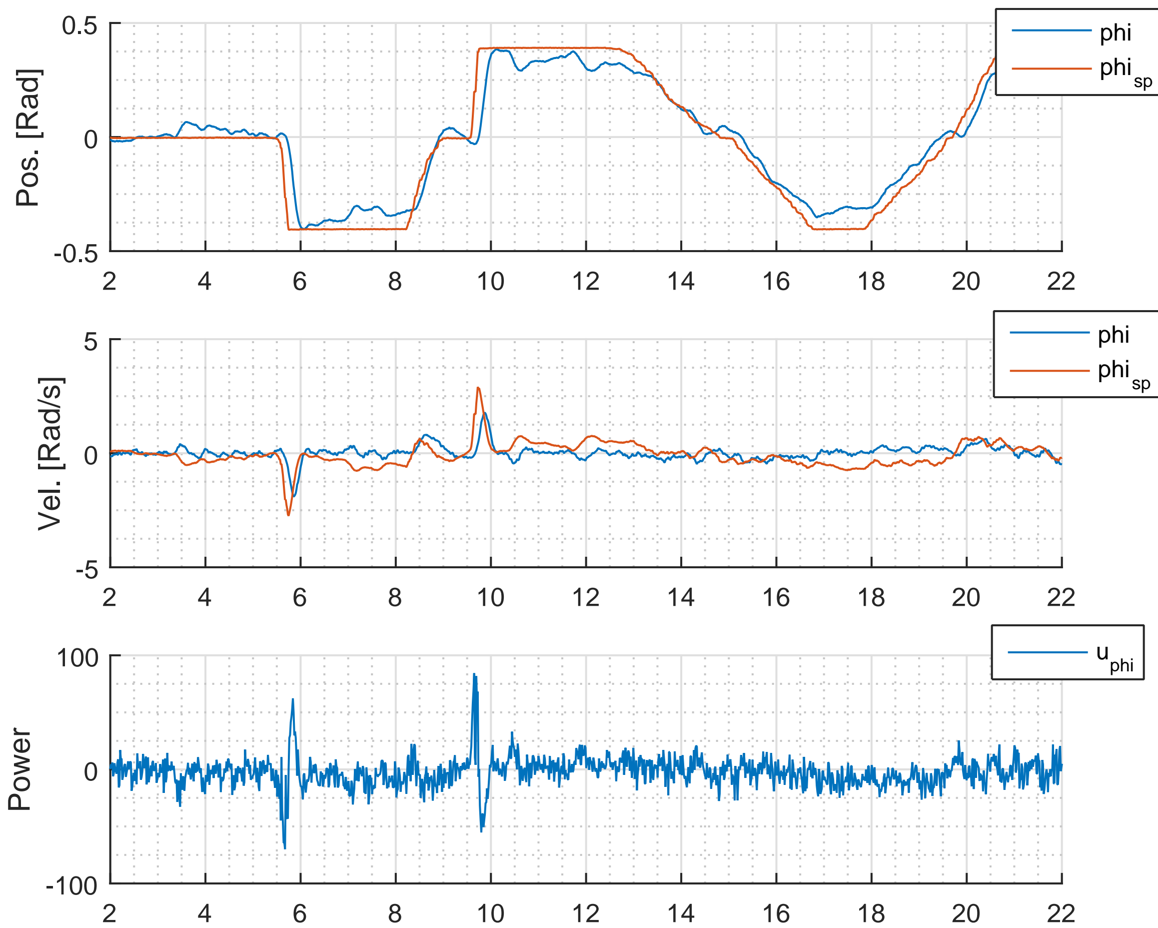


Figure 10.4: Benchtest results

As seen in the figure, when removing the chance of propeller wash disturbing the back propellers the drone and controller performed much better. The controller also performs well during simulations, but this is because aerodynamics is not modeled, for simplicity.

The overall functionality of this mode works as expected, there was good performance in the yaw (ψ) controller, but not roll and pitch when the drone was in motion. It might be worth a try to increase the P and I parameter of the angular velocity controller (secondary controller), and decrease the P parameter on the angular position controller (primary controller).

It is worth to mention that Kristoffer Gryte, a drone expert at NTNU, explained that the ESC's has a lot to do with the drone responsiveness, and using a ESC with SimonK firmware is the best

choice because it is designed specifically for multirotors [18]. The ESC's currently used is quite unknown, and the quality is not guaranteed to be the best, due to the lack of a proper data sheet.

10.2 Manual GNSS mode

The objective with this mode is to be able to control the drone's orientation manually with the RC remote, when there is no input from the RC remote the drone shall hold its current geographic position, explained in detail in section 8.1.

To test this the drone was powered up outside, when the GNSS receiver accuracy was approximately down to 3 meters the testing begun. The drone was flown up to speed, and then the ordered to stop (no RC remote input) to see if the drone held the position. Tests also included to manually pull the drone away from its position and see if it flew back to setpoint, seen in the video referred to in appendix B, named "manual GNSS mode".

During flight the drone did behave as expected, by giving the drone maximum pitch/roll, the drone came up to speed, and when there was no input from the RC remote, the drone did aggressive but precise maneuvers to counteract its linear momentum. See figure 10.5, the drone is controlled by the RC remote when the setpoint equals the measured value.

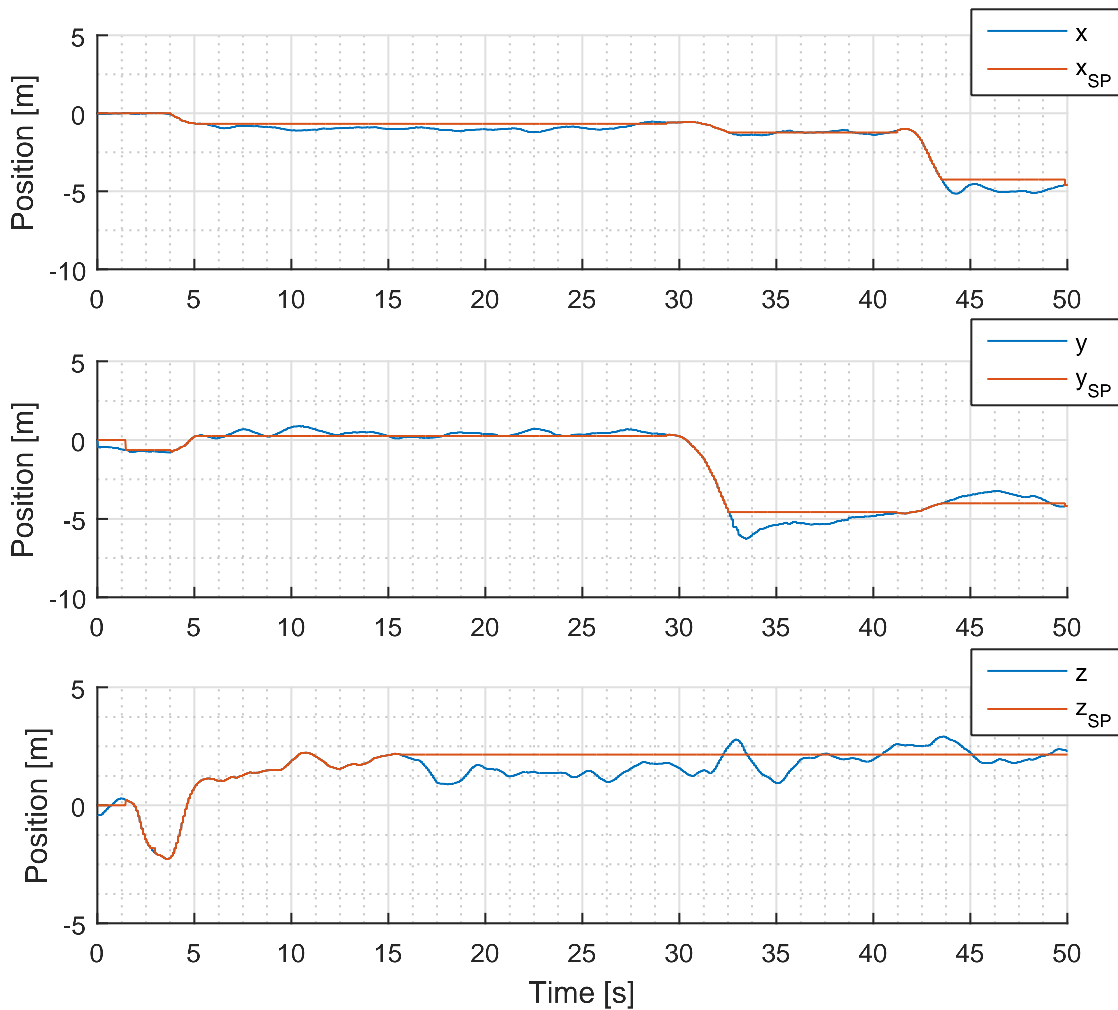


Figure 10.5: Drone position

When studying the controller error in the position plot, from $t = 5$ to $t = 30$, we can see that the controller accuracy is about ±0.5 meters for the horizontal plane, and ±1 meter for the vertical plane. As mentioned in chapter 6 when testing outside the weather conditions need to be close to perfect (no rain, no wind), making the controller tuning opportunity rare, so there should be some accuracy improvements by perfecting the controller parameters.

It is also worth to mention that at $t = 2$ in z-axis plot in the position plot the altitude suddenly goes negative, the reason for this is that the propellers creates a high pressure at takeoff, which is affecting the pressure sensor.

The linear velocity for the drone was also logged during flight, see figure 10.6.

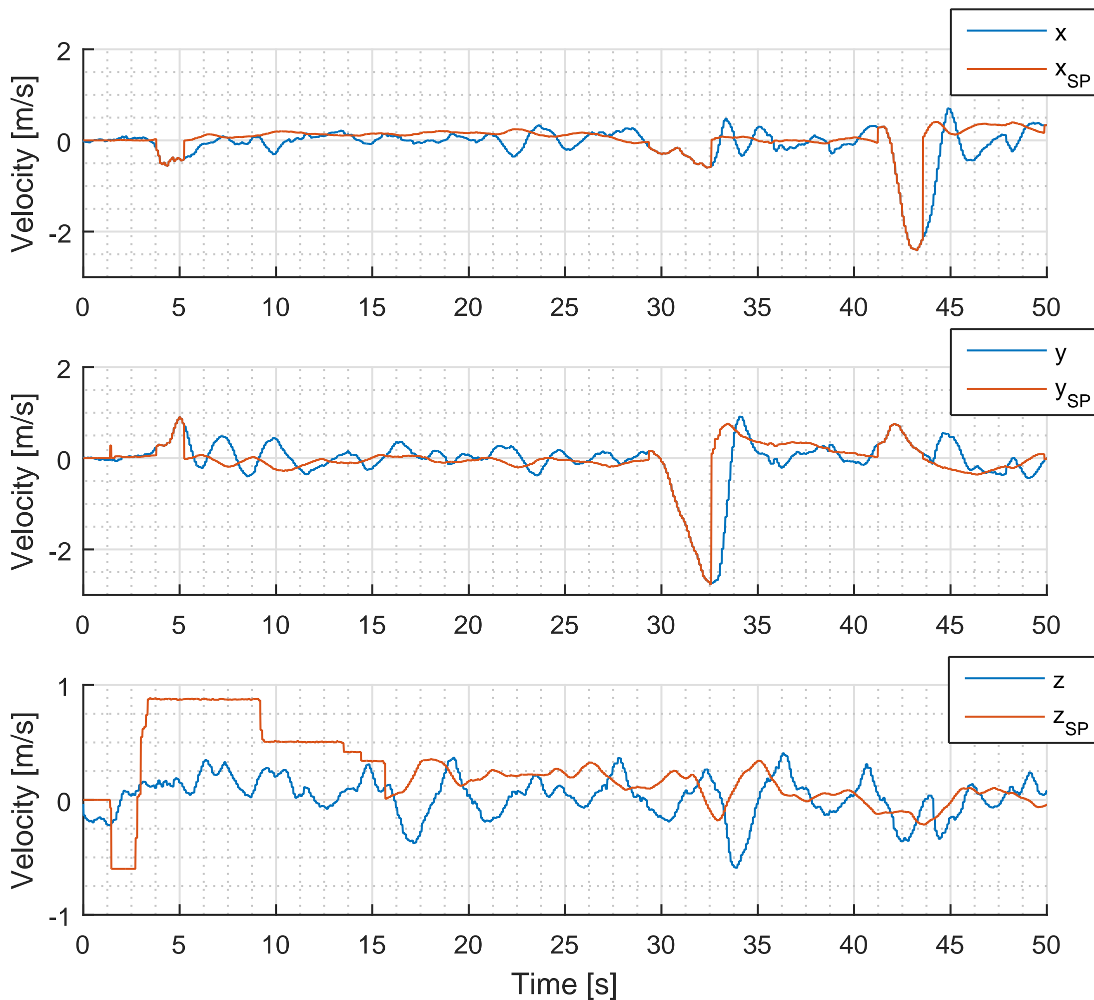


Figure 10.6: Drone velocity

As mentioned in chapter 6 the position controller (primary controller) is controlling the setpoint of the velocity controller (secondary controller). So in this case $i_{SP} = u_i$, where i is x, y and z.

As seen from the second subplot (y) in figure 10.6 the drone used about two seconds to decelerate the speed from about 3 m/s to 0 m/s, a quite good performance. But in the third subplot (z) at $t = 3$ to $t = 15$, the setpoint is controlled from the RC remote, as explained in subsection 8.1. The z-axis velocity controller does not perform so good, this could however be because of the integrator parameter being too low, increasing it should give better results.

The overall functionality of this mode works as expected, but it is also proved that the posi-

tion controllers also need some more tuning. Though the drone deceleration is fast and precise when it is commanded to stop its linear speed, there is definitively room for improvements, especially for the z axis controller.

10.3 Autonomous mode

The objective with this mode is being able to load a flight plan to the drone via Bluetooth before takeoff, when executed the drone will fly to each waypoint with an accuracy of ± 1 meter, the heading between waypoints will be calculated during execution. The drone should also be able to land and disarm the motors by itself. The functionality of this mode is explained in detail in section 8.1.

To test this mode the drone was powered up outside and the following flight plan was loaded

1. (0,10,4), fly 4 meters up then 10 meters north.
2. (10,10,4), fly 10 meters east.
3. (10,0,4), fly 10 meters south.
4. (0,0,0), fly 10 meters west, then land.

The result of this flight plan is that the drone shall fly in a square which is 10×10 meters, and land the same place it started, the heading shall also be altered by $\frac{\pi}{2}$ between waypoints. The waypoints were marked on the ground at the test site in order to see if the drone actually flew the correct distance, see figure 10.7.

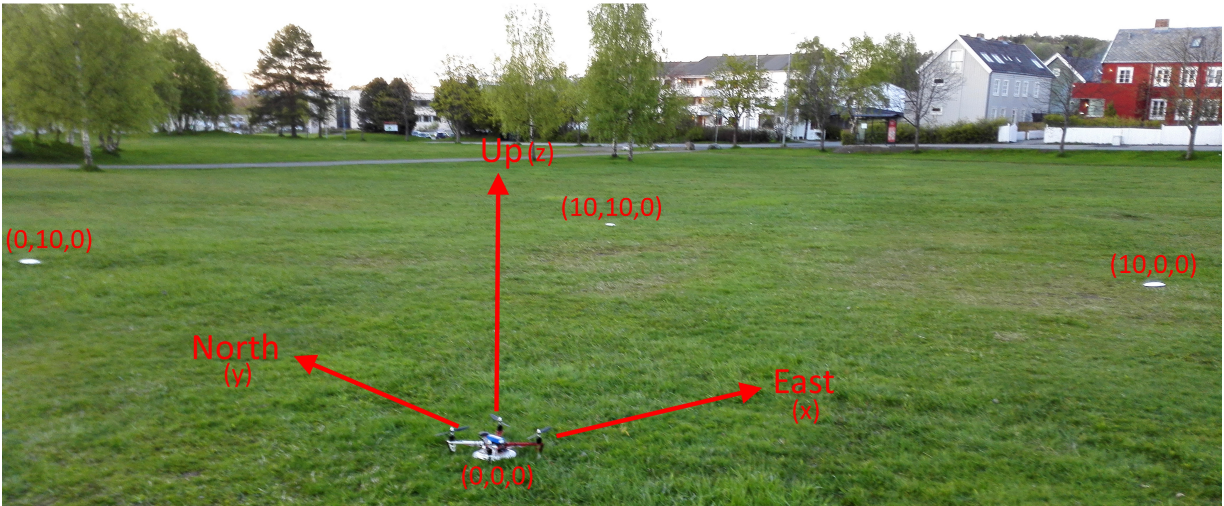


Figure 10.7: Test setup

During flight the drone did behave as expected, which does not come as a surprise as the flight controller was simulated against the model in Simulink before implemented on the drone. But, as before, the drone is a bit slow to move from waypoint to another. A video from this flight is referred to in appendix B, named "autonomous mode". The position plots can be seen in figure 10.8.

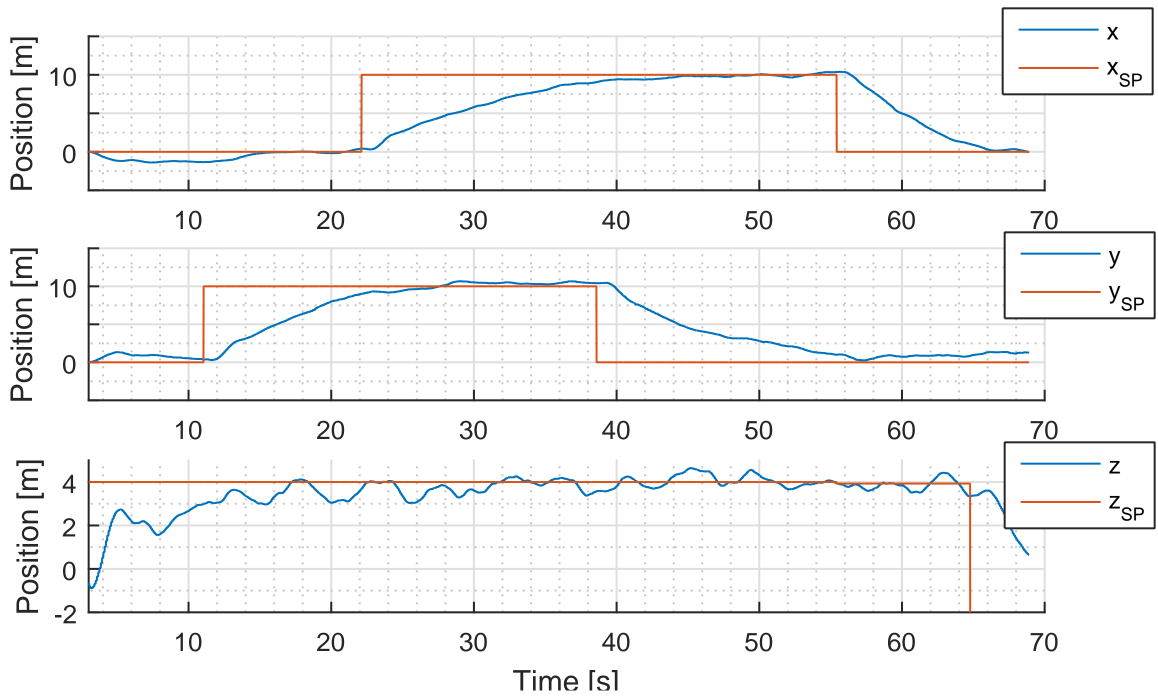


Figure 10.8: Drone position

As seen in the position plot the drone hits the setpoints quite accurately, but it is slow, using about 15 seconds to fly 10 meters. The issue becomes clear by taking a closer look at the velocity plot, see figure 10.9.

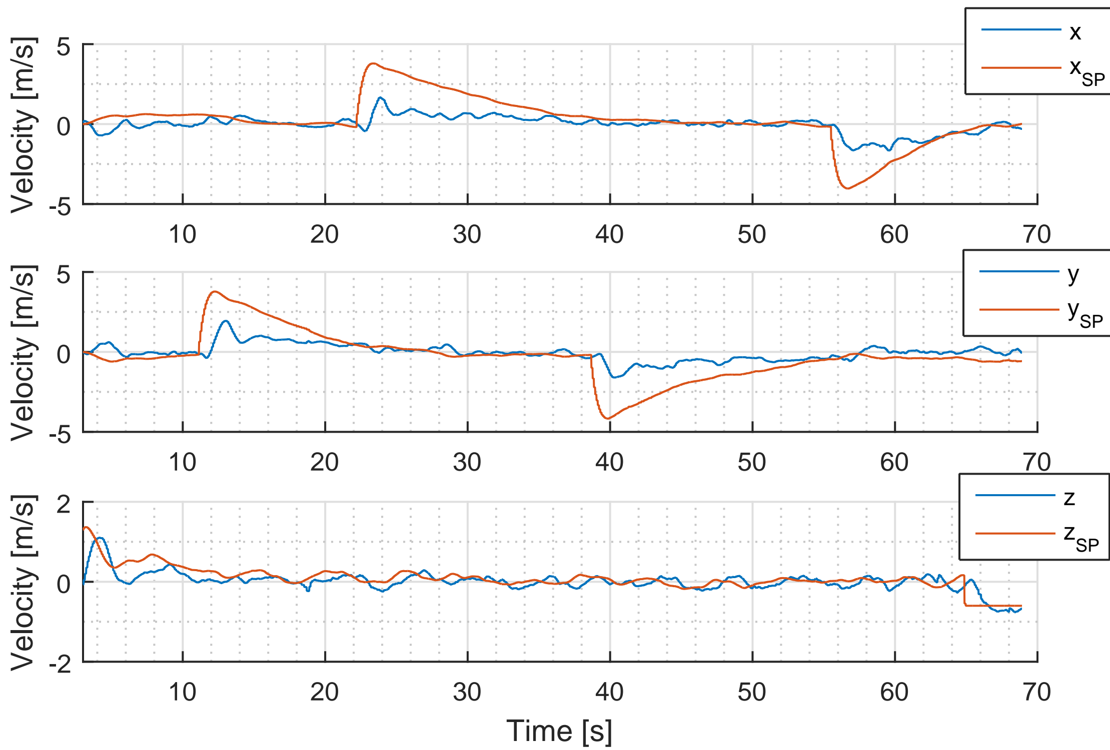


Figure 10.9: Drone velocity

Though the velocity setpoint (primary controller output) is quite high, the drone is not able to fly that fast because of the issue discussed in section 10.1.

The drone did, however, alter its heading automatically between waypoints, see figure 10.10.

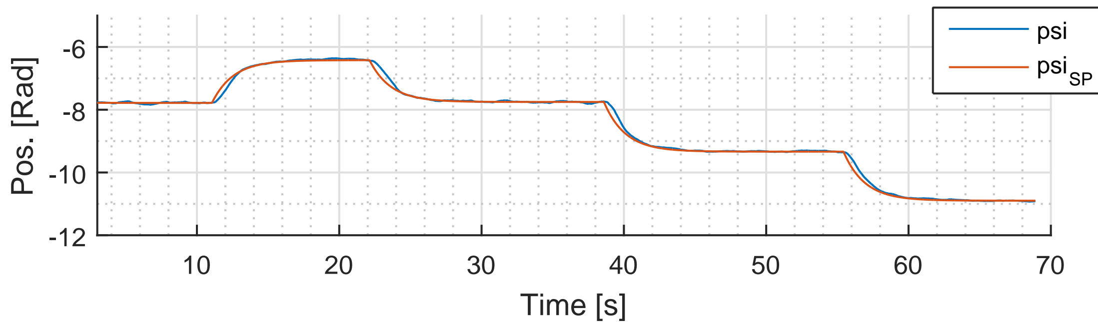


Figure 10.10: Drone heading

The smooth transitions in the heading setpoint, ψ_{SP}, is done by feeding the setpoint trough a low-pass filter, and avoid quick and aggressive motor inputs, which is not needed for heading change. This function is also visible in the video referred to in appendix B.

The functionality of this mode seems to work flawlessly, but, as explained, the drone is very slow. During this test there was almost no wind, but during other tests the drone had some problems holding its position if there was more wind than 1 m/s.

A bird's eye view of this test can be seen in figure 10.11.

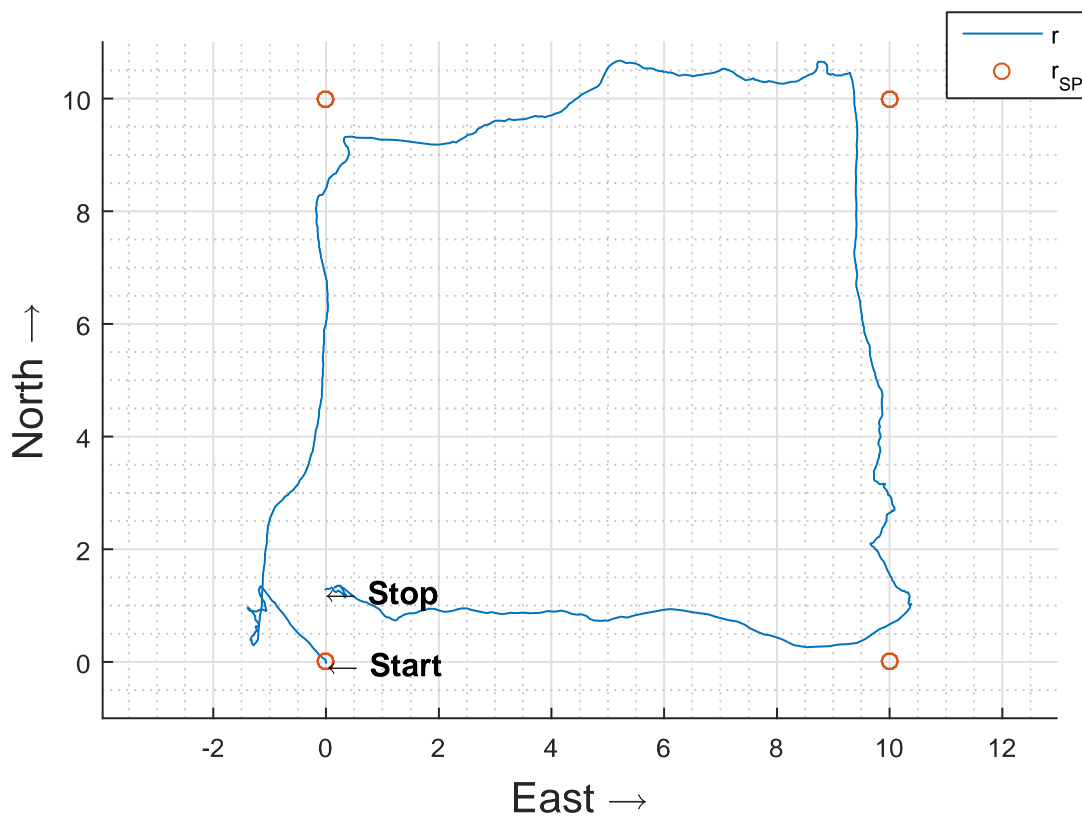


Figure 10.11: XY plot of drone trajetory

As seen from the XY plot the drone does hit the way-points within the minimum of one meter before it moves on to next way-point. The drone landed a bit over 1 meter from where it started, though the setpoint was to land at (0,0,0) it is still quite good.

10.4 Autonomous object mode

The objective with this mode is to load a coordinate of the location of an object to the drone via Bluetooth, the drone will go to this coordinate and search for the object, once found, the drone will lock on to the object and fly close enough to pick up the object. Once the object is picked up the drone will fly back to where it took off and drop the object. The functionality of this mode is explained in detail in section 8.1.

Due to the issues explained in section 10.1 the drone has not been accurate enough to test this mode, this test will therefore be simulated. We will not simulate the whole sequence as explained in section 8.1, but test the equations derived in chapter 7. With this we can test the most uncertain element of this mode; can the drone get close enough to the object using a vision sensor? The closer the object is to the vision sensor the narrower the field of view gets, so position precision is important here. If good results are far off by simulating, it might not be worthwhile implementing on the drone.

We are going to simulate a flight where the drone ascends to 4 meters, and flies towards the same coordinate as the object, in this case 10 meters north from takeoff point (0,10,0). Once the drone has arrived it will descend slowly to 0.2 meters, we are going to check when the object is out of view for the vision sensor. See figure 10.12.

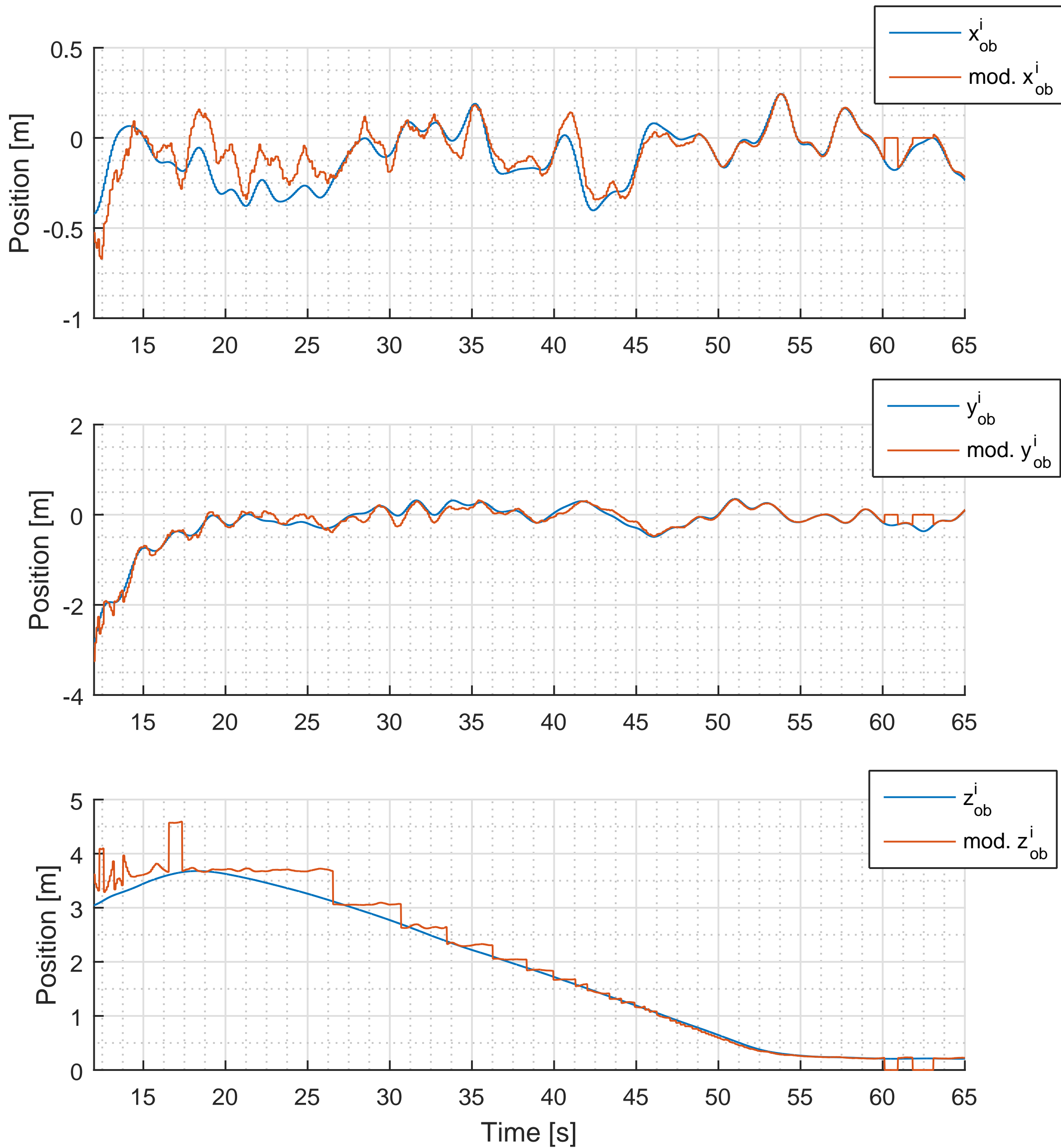


Figure 10.12: Simulation of r^i_{ob}

The simulation went as expected, and as seen in figure 10.12 the drone could get down to 0.2 meters from the object, but at $t = 60$ the object went out of view. This means that at 0.2 m the drone is so close that the object is just barely in view, this could of course be fixed by using a longer hook, mentioned in section 4.7.

Because we are using the object width to measure the distance from the object, we get a poor resolution when the object is far away, see the third subplot in figure (z^i_{ob}) in figure 10.12. This happens because the object covers fewer pixels in the vision sensor, but the resolution gets better as the object gets closer to the drone. Good resolution is important when the drone is close to the object because of the reduced field of view, this is therefore not something to worry about.

When studying the first subplot (x^i_{ob}) in figure 10.12, we can see there is some deviation from modeled and real value. The reason for this can we see in function 7.9, the calculated distance is used to calculate the object position, mentioned in the previous paragraph. As seen from the subplot it gives an inaccuracy of approximately 20 centimeters when the drone is about 3 meters away from the object, but this inaccuracy gets less and less the closer the drone gets to the object, making the inaccuracy negligible.

By analyzing the x- and y data the precision is down to approximately ± 0.2 meters, which in our case is not optimal unless we are using a really big loop on the object. The drone model does, however, not take air resistance in account making it more likely to move faster in the x- or y direction for small deviations in roll or pitch, which is modeled as white noise.

With this we can conclude that the simulation was successful and definitively worth trying to implement on the drone.

Chapter 11

Discussion and Recommendations for Further Work

In this chaper we will discuss the results found during this project at recommendations for further work.

11.1 Discussion

There is room for improvement when it comes to the motor throttle compensation, explained in section 2.1. Instead of creating a function with regards to motor throttle, we could move the electronic compass further away from the wiring and motors. As mentioned in 4.1, the GNSS module has a built in compass, which could yield better results.

The roll and pitch issues mentioned in section 10.1 was thought to be due to propeller wash, but Kristoffer Gryte meant that the propeller wash issue was negligible and should not be the root cause of the problem presented here. During the bench test the drone performed well, so in this case it looks like the propeller wash does have an impact on the performance, but this is probably because of suboptimal orientation controller parameters. The tuning of the orientation controller was done during winter times, which means not ideal flying conditions outside due to the risk of getting snow in the electronics. The tuning was therefore done inside with limited space for flying at high velocity, and the issue was not noticed. Because of this it is

suggested to revisit the controller parameters.

This made the drone very vulnerable to wind, because the position controller was not able to do small attitude corrections to counteract the wind.

Although objective three and four was never met, much of the work has already been done, the vision sensor was installed to the drone and drivers were developed. Data processing algorithms were designed and tested, with good results. The method for picking up an object, along with the control sequence, was also suggested. When the roll- and pitch issue is fixed it should be straight forward to implement the last flight mode.

As explained in section 4.6, the necessary sensor for completing objective six, has also been installed, but is at the moment only used in the landing sequence. This sensor could also be used for a collision detection system.

11.2 Recommendations for Further Work

The following objectives were not met

1. Implement a method for the drone to recognize objects to be picked up, and lock on to the object
2. Implement a method for picking up an object
3. If there is enough time a collision detector system should be implemented.

The correct approach to finish these objectives would be

1. Re-tune the orientation controller.
2. Do a precision test of the position controller, does the position controller track the setpoint with centimeter precision? If not, improve this, maybe use the accelerometer?
3. Implement the object data processing algorithms, found in chapter 7.
4. Test the object tracking ability, is it precise enough?
5. Implement the hook, as explained in section 4.7.

Appendix A

Acronyms

GPS Global Positioning System

PCB Printed Circuit Board

MCU Micro Controller Unit

PID Proportional, Integral, Derivative

LQR Linear Quadratic Regulator

RC Radio Control

ESC Electronic Speed Controller

GNSS Global Navigation Satellite System

UART Universal Asynchronous Receiver-Transmitter

NMEA The National Marine Electronics Association

UBX Ublox

CPU Central Processing Unit

NAV-SOL Navigation Solution

ECEF Earth Centred Earth Fixed

SPI Serial Peripheral Interface

I^2C Inter-Integrated Circuit

LED Light Emitting Diode

RGB Red, Green, Blue

GPIO General Purpose Input Output

IR Infrared

ADC Analog to Digital Converter

LiPo Lithium Polymer

ENU East North Up

WGS-84 World Geodetic System 1984

RTOS Real-Time Operation System

OS Operating System

UML Unified Modeling Language

FIFO First In First Out

CLI Command Line Interface

Appendix B

Project Files

Under is a list of the location of the attached files.

1. Keil uVision project (MCU Project files)

   Project Files/Source/FreeRTOSV8.2.3 FreeRTOS files

   Project Files/Source/Drivers HAL Drivers

   Project Files/Source/Simulink Necessary Simulink code generation files

   Project Files/Source/Src/CLI-commands Command line interface implementations

   Project Files/Source/Src/gps GPS driver

   Project Files/Source/Src/LED LED driver

   Project Files/Source/Src/myTasks All tasks

   Project Files/Source/Src/pixy Vision sensor drivers

   Project Files/Source/Src/PPMcontrol RC remote and ESC driver

   Project Files/Source/Src/sensors Sensor drivers

   Project Files/Source/Src/main main file (mostly hardware setup)

2. MATLAB / Simulink project files

   Project Files/Matlab/Common Block library and parameters file

   Project Files/Matlab/Data Processing Data processing blocks

Project Files/Matlab/Model Quadcopter model

Project Files/Matlab/Orientation Controller Orientation controller blocks

Project Files/Matlab/Flight Controller Flight controller block

Project Files/Matlab/flightController_grt_rtw Auto generated C code

3. Videos

Project Files/Videos/manual mode Test flight in manual mode

Project Files/Videos/manual mode bench test Bench test manual mode

Project Files/Videos/manual GNSS mode Test flight manual GNSS mode

Project Files/Videos/autonomous mode Test flight autonomous mode

4. Eagle project files (Schematics)

Project Files/Schematics/FlightController/FlightController.brd PCB file

Project Files/Schematics/FlightController/FlightController.sch Schematics

Project Files/Schematics/FlightController/board PCB screenshot

Project Files/Schematics/FlightController/board Schematics screenshot

5. Datasheets

Project Files/Datasheets/5050LED LED's

Project Files/Datasheets/Bluetooth module Bluetooth

Project Files/Datasheets/uln2803a Darlington transistor array

Project Files/Datasheets/MCU Microcontroller datasheets

Project Files/Datasheets/ESC Picture of ESC datasheet

Project Files/Datasheets/Sensors/ADXL345 Accelerometer

Project Files/Datasheets/Sensors/BMP085 Pressure sensor

Project Files/Datasheets/Sensors/GP2D12 IR sensor

Project Files/Datasheets/Sensors/HMC5883L Compass

Project Files/Datasheets/Sensors/L3G4200D Gyroscope

Project Files/Datasheets/Sensors/u-blox7-V14 GNSS receiver

6. Pre-project report

   Project Files/pre-project report Fall project report

7. Master thesis

   Project Files/Master Master thesis

Appendix C

Simulink Screenshots

The Simulink project files is referred to in appendix B.

C.1 Orientation Data Processing

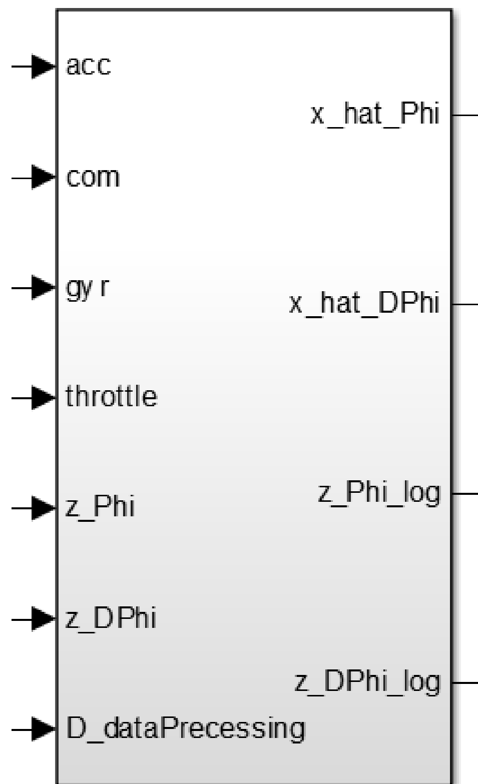


Figure C.1: Orientation data processing block

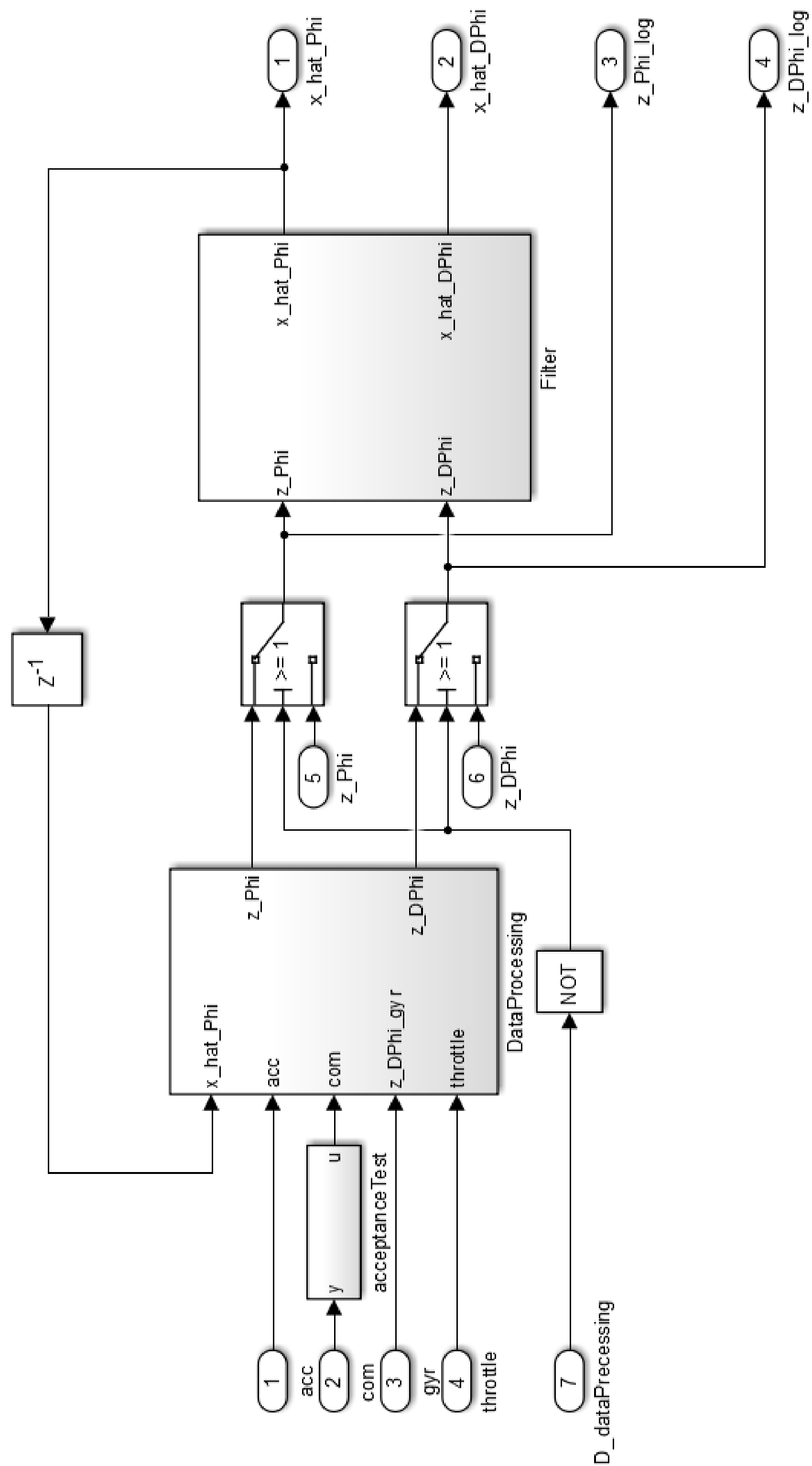


Figure C.2: Inside orientation data processing block

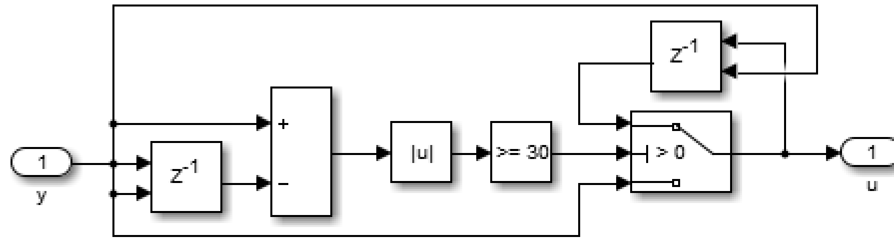


Figure C.3: Inside compass acceptance test block

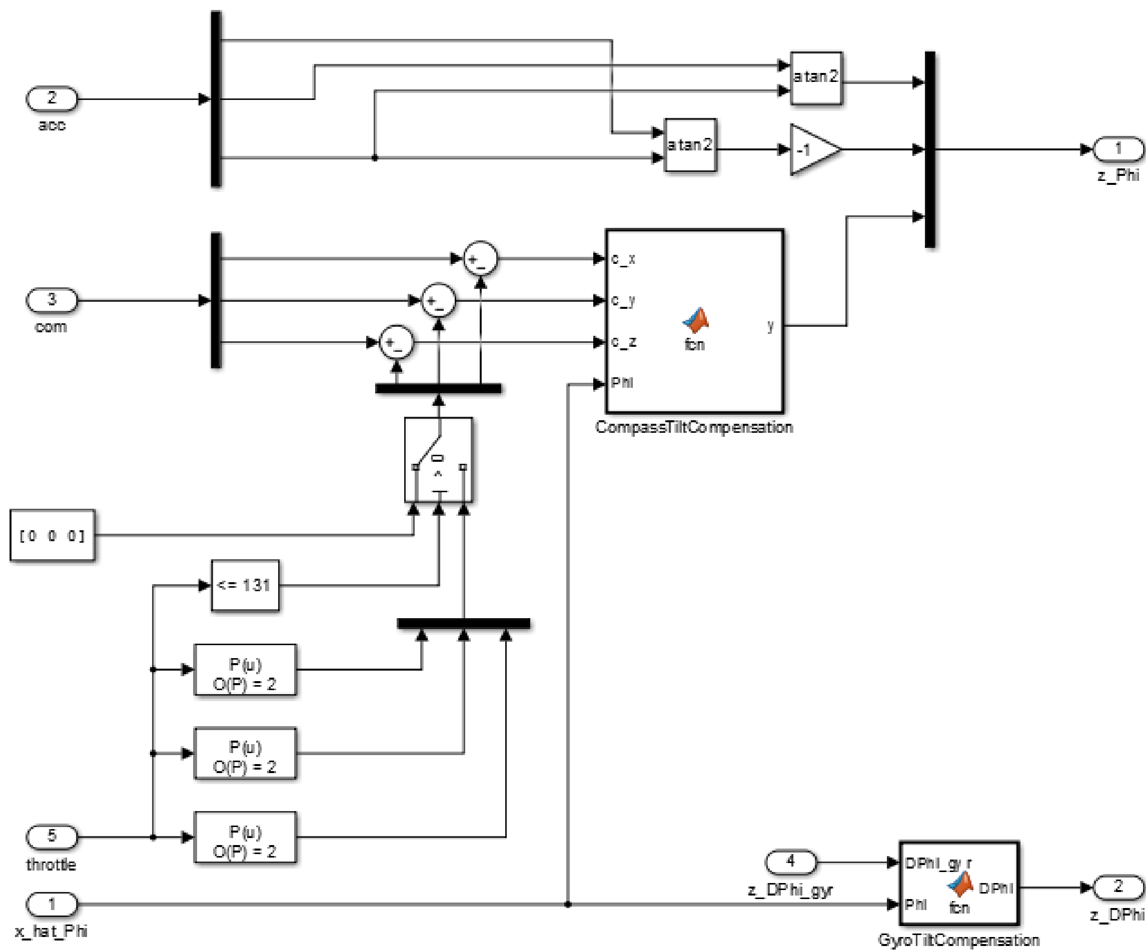


Figure C.4: Inside data processing block

```
function y    = fcn(c_x, c_y, c_z, Phi)

phi = Phi(1);
theta = Phi(2);

persistent rounds prevPsi;

x_psi = c_x*cos(theta) + c_y*sin(theta)*sin(phi) + c_z*sin(theta)*cos(phi);
y_psi = c_y*cos(phi) - c_z*sin(phi);
psi = atan2(x_psi, y_psi);

if isempty(prevPsi)
    prevPsi = psi;
    rounds = 0;
end

if prevPsi - psi > pi
    rounds = rounds + 1;
elseif psi - prevPsi > pi
    rounds = rounds - 1;
end

prevPsi = psi;

y = psi + rounds*2*pi;
```

Figure C.5: Inside compass tilt compensation block

C.2 Orientation Controller

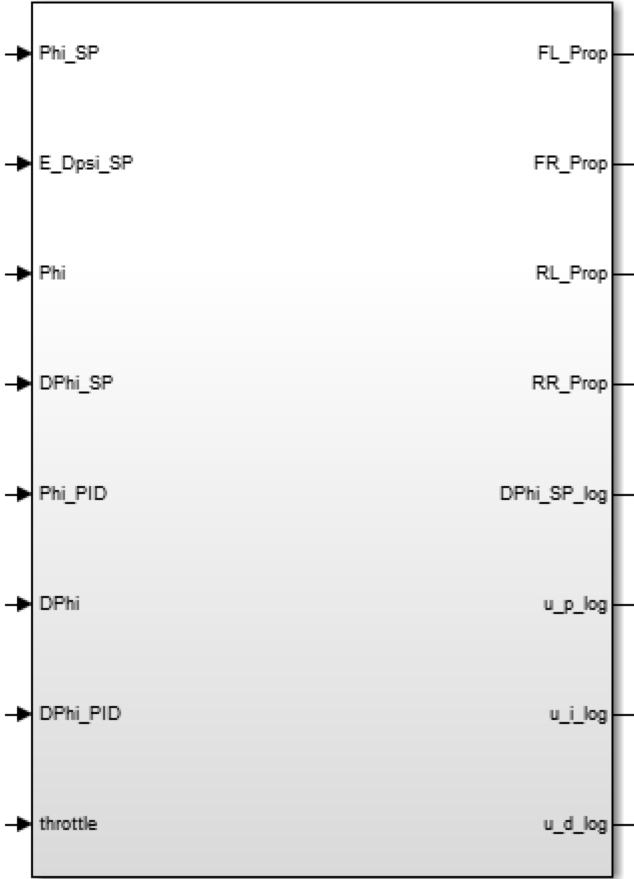


Figure C.6: Orientation controller block

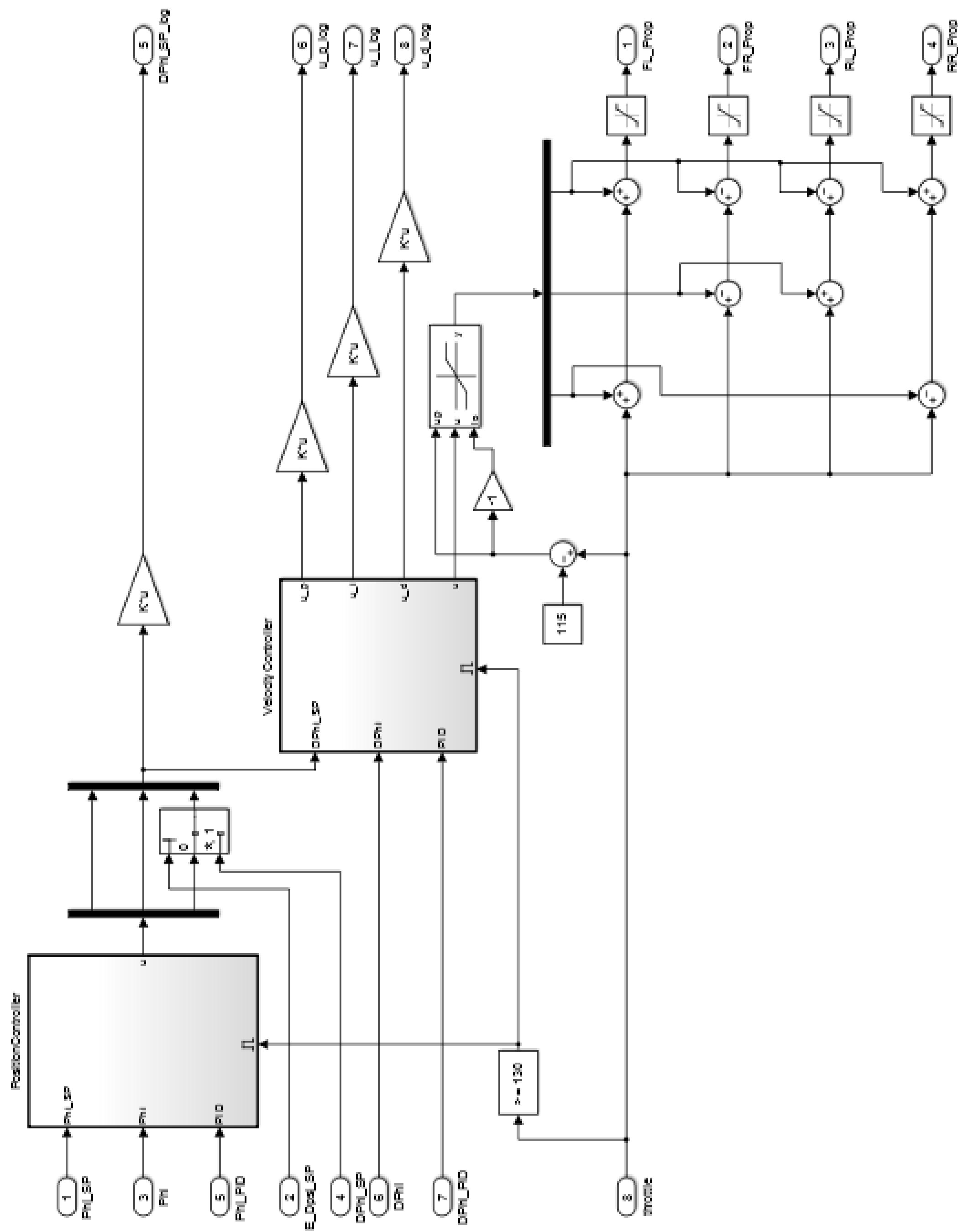


Figure C.7: Inside orientation controller block

Figure C.8: Inside velocity controller block

C.3 Position Data Processing

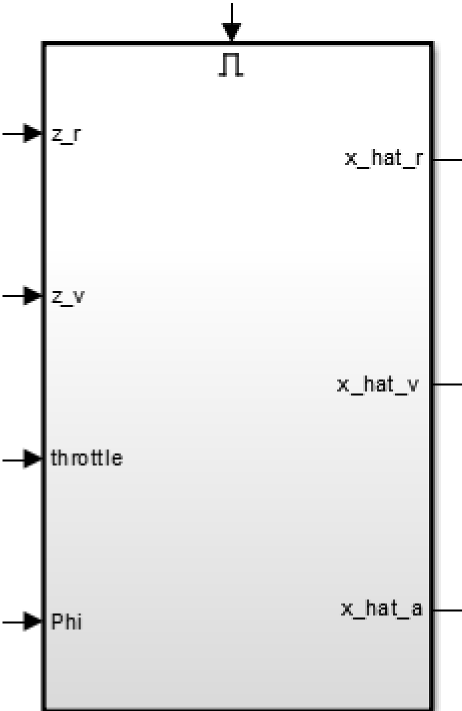


Figure C.9: Position data processing block

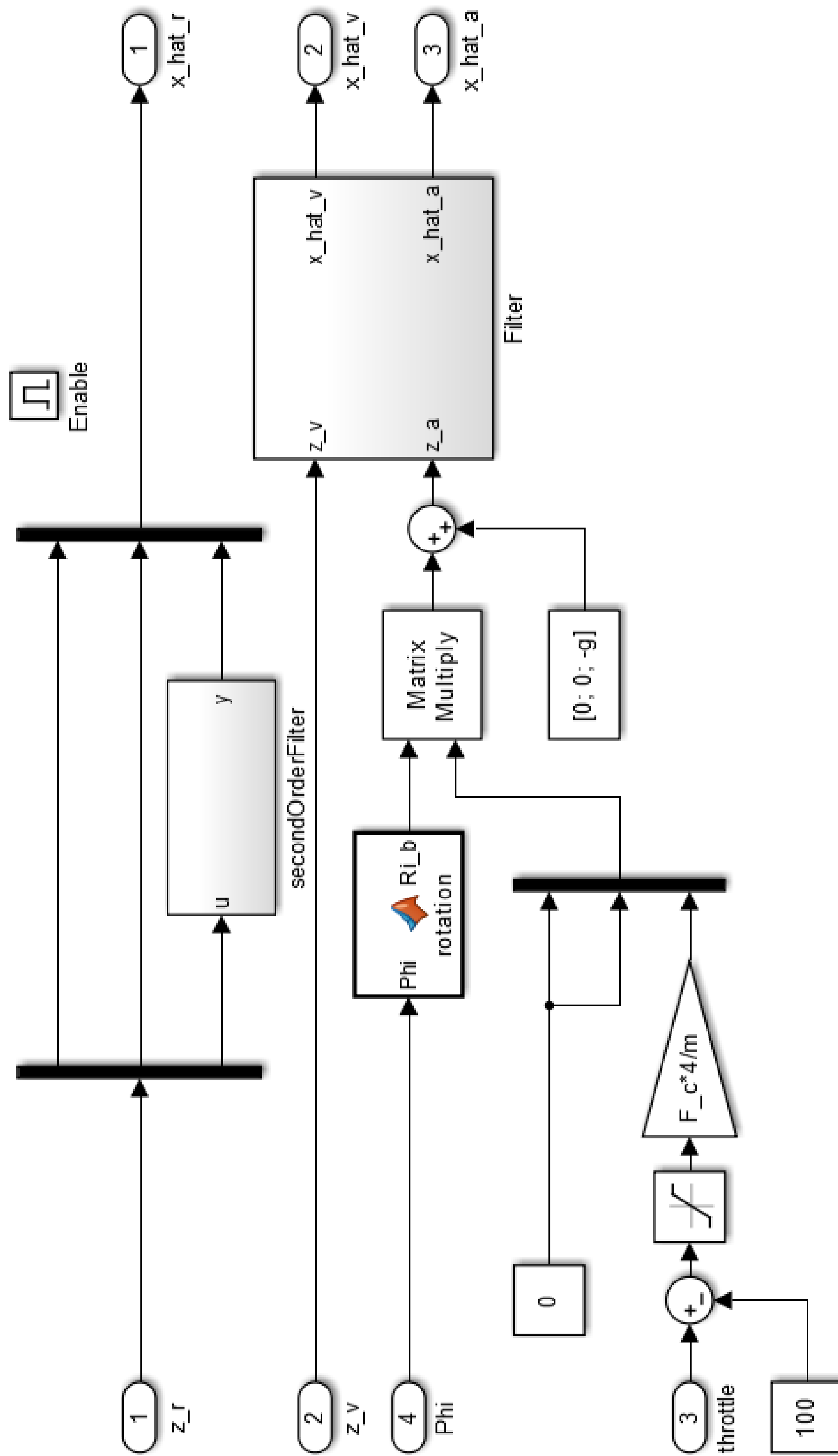


Figure C.10: Inside position data processing block

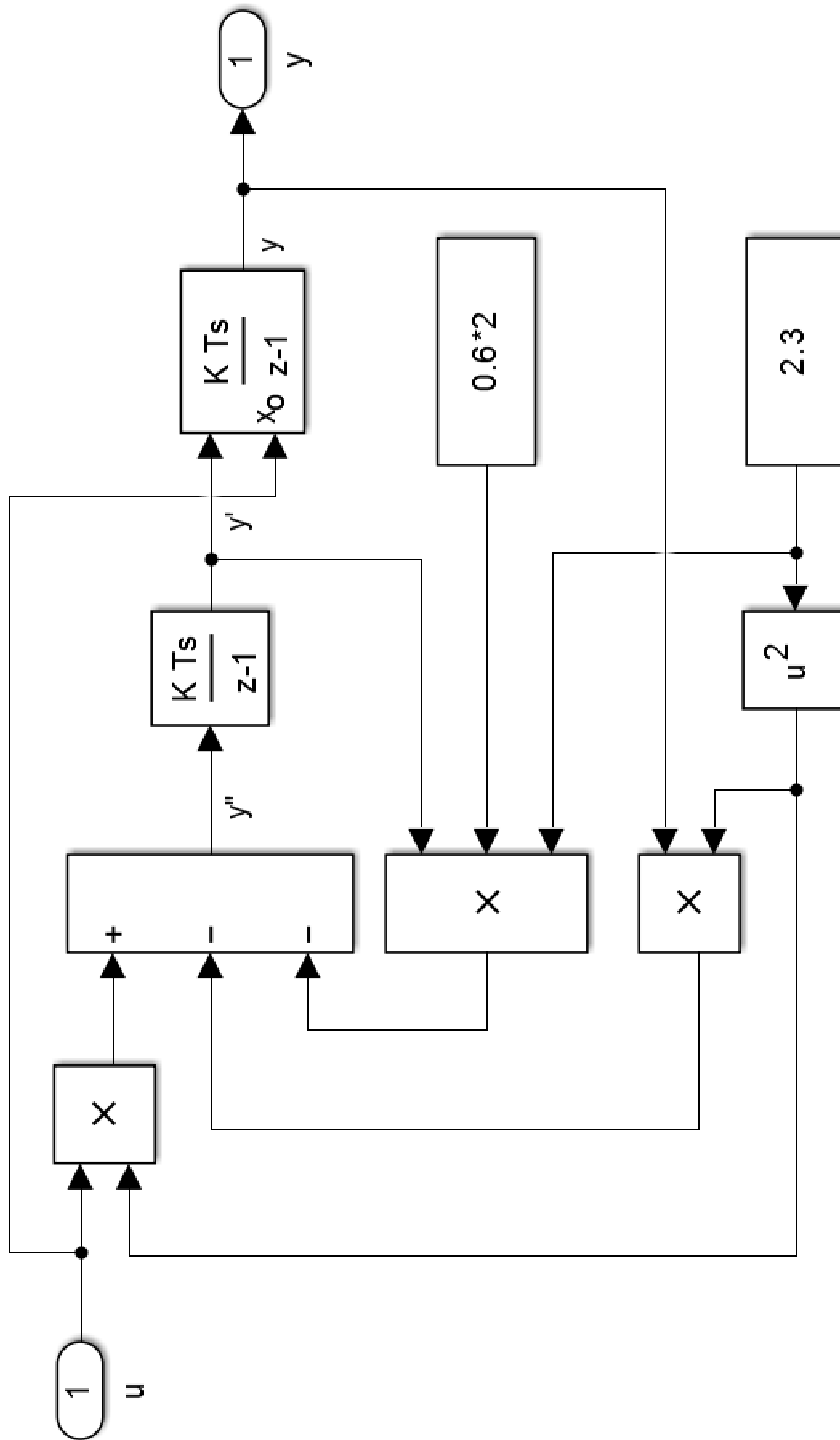


Figure C.11: Inside second order filter block

C.4 Position Controller

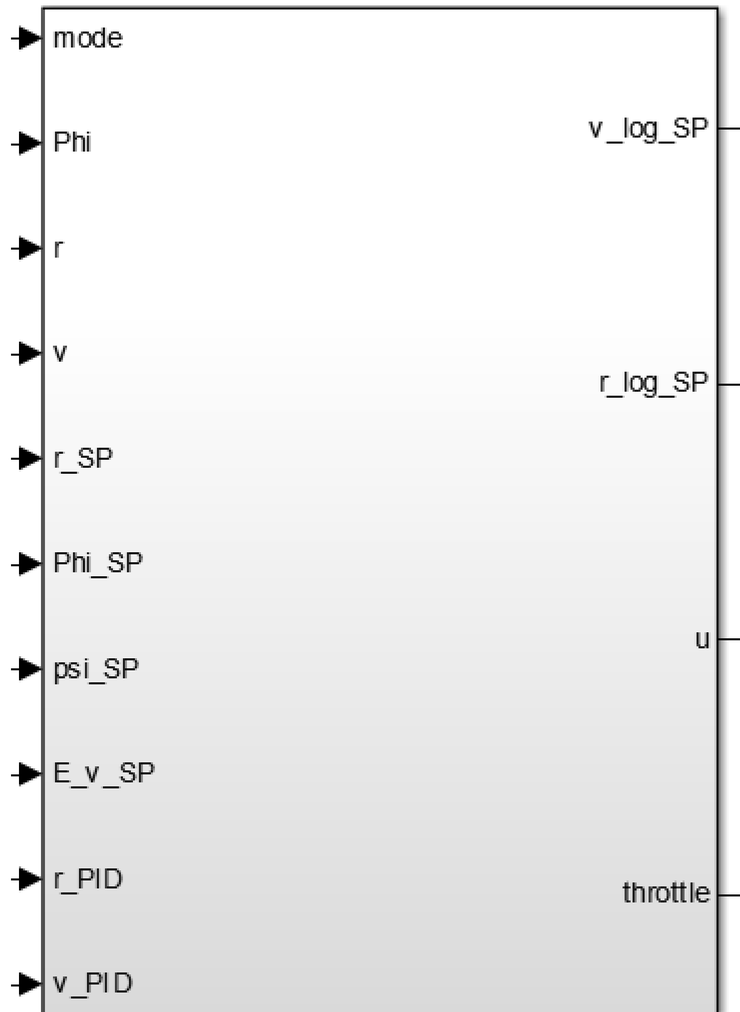


Figure C.12: Position constoller block

Figure C.13: Inside position controller block

C.5 Object Tracking

Data Processing

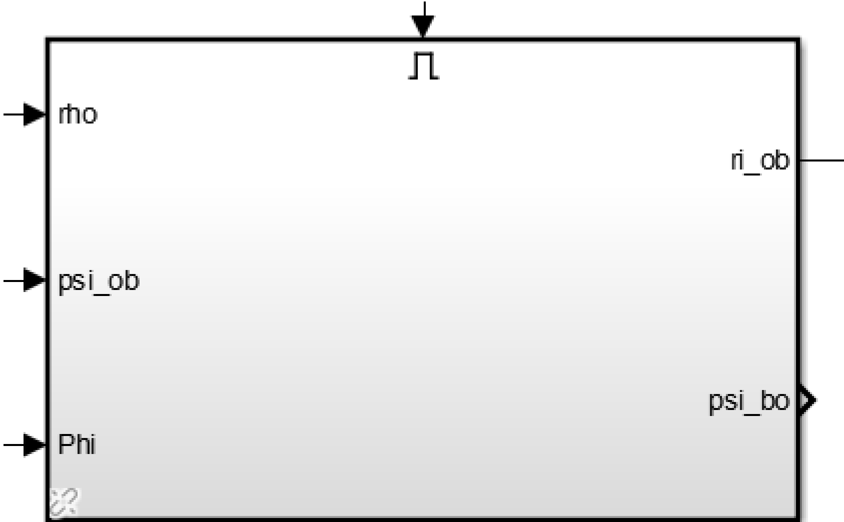


Figure C.14: Object data processing block

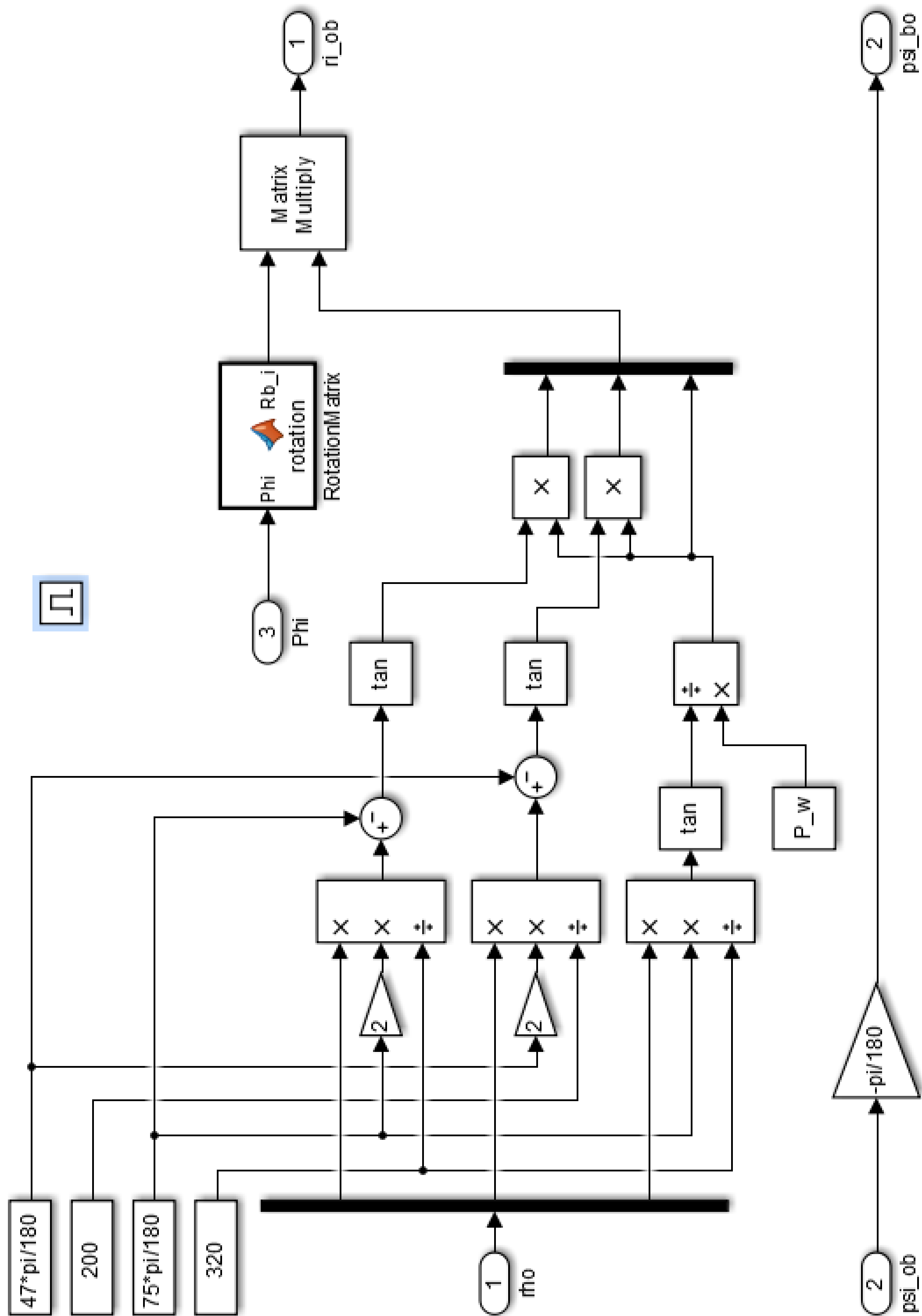


Figure C.15: Inside object data processing block

Model

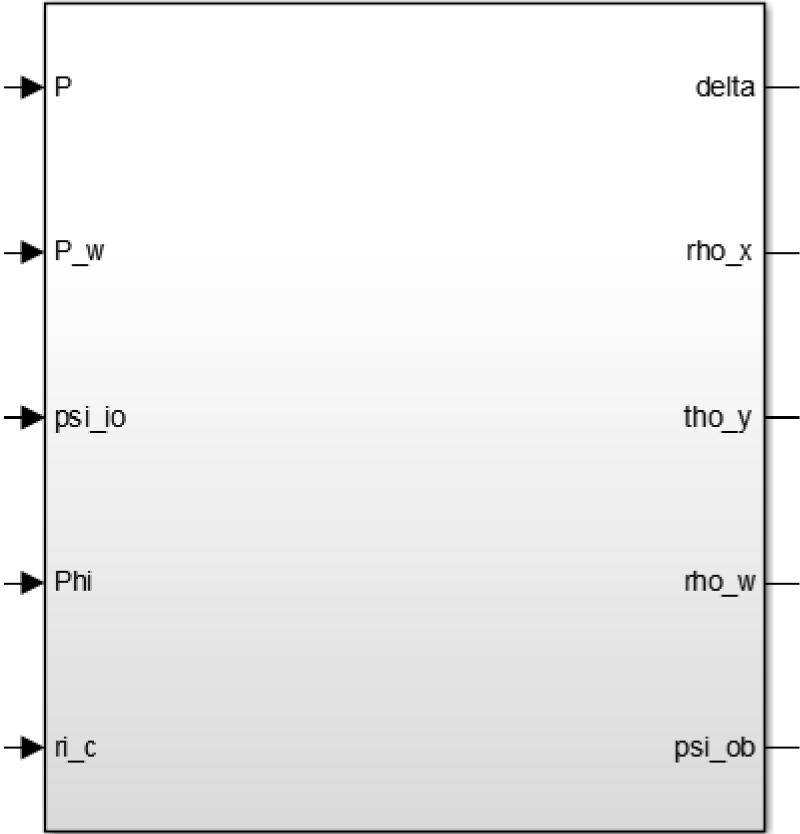


Figure C.16: Object model block

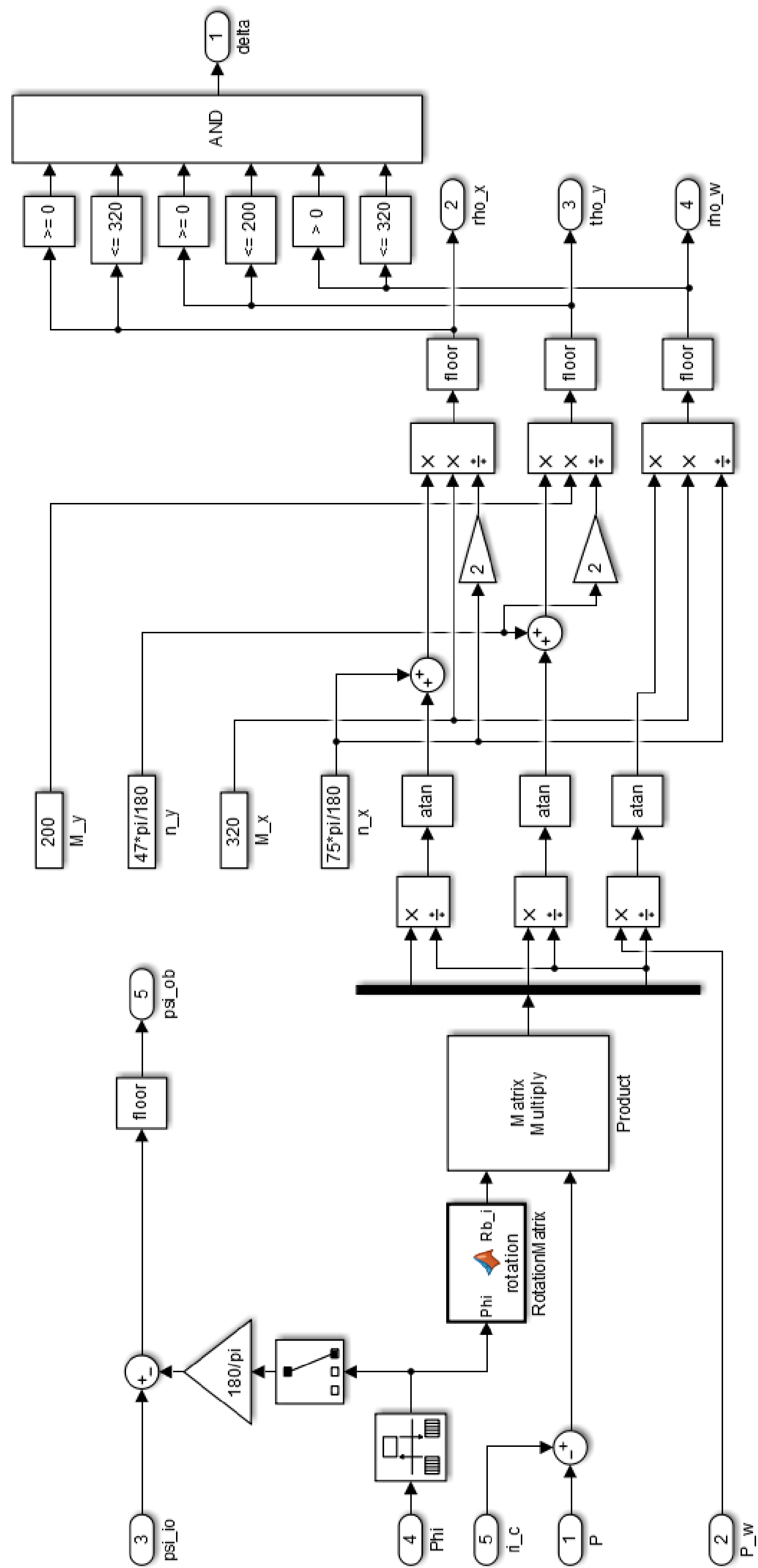


Figure C.17: Inside object model block

C.6 Flight Controller

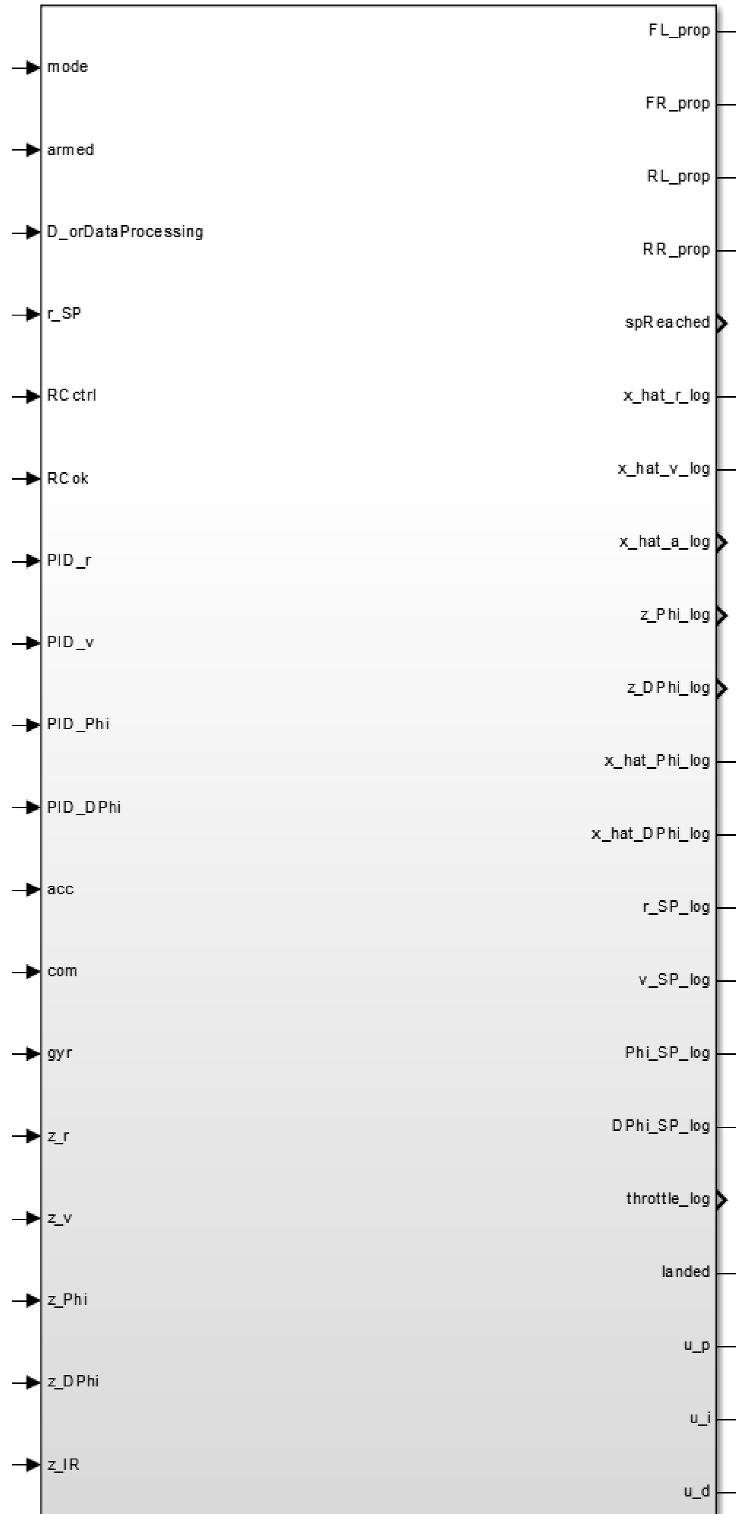


Figure C.18: Flight controller block

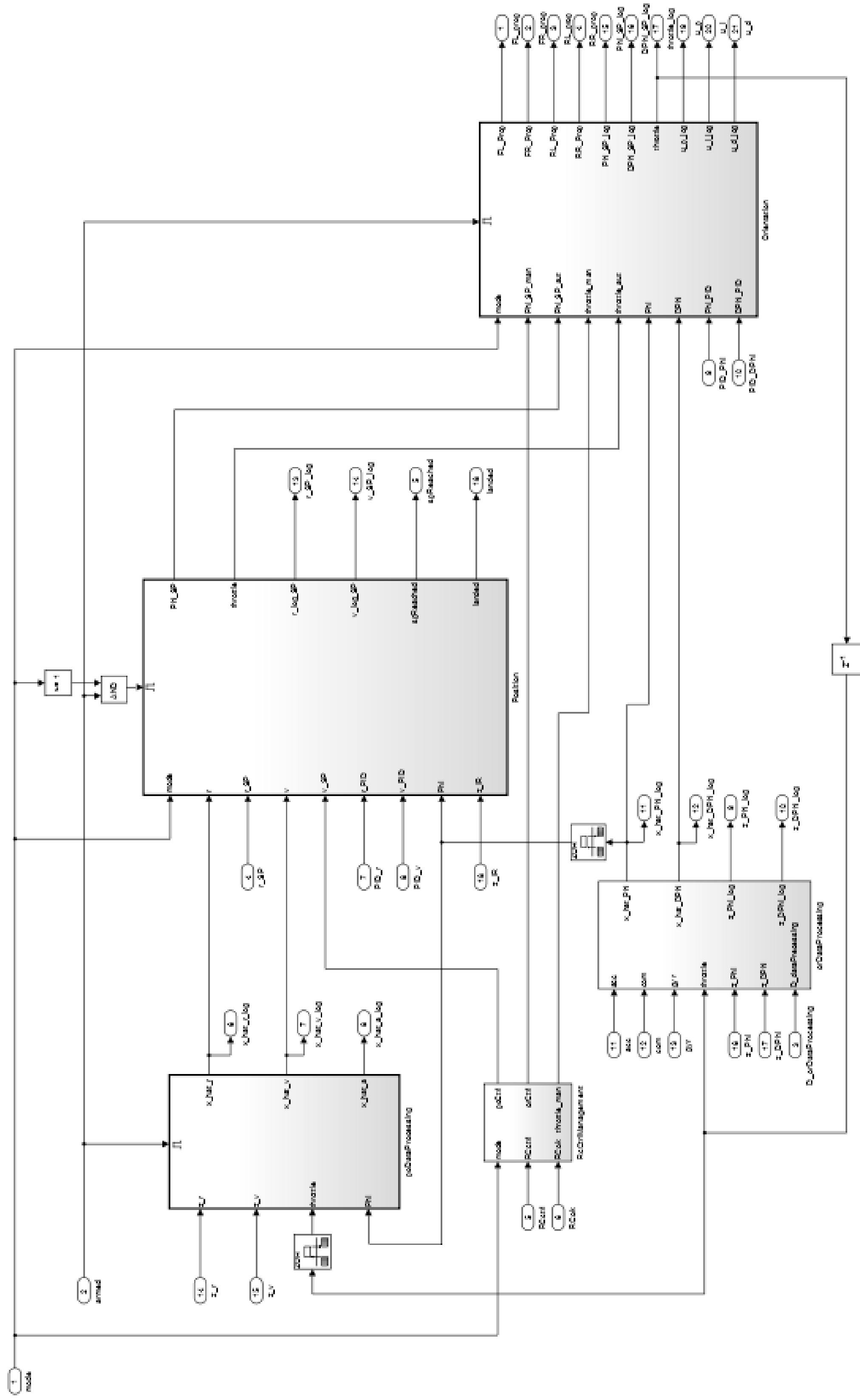


Figure C.19: Inside flight controller block

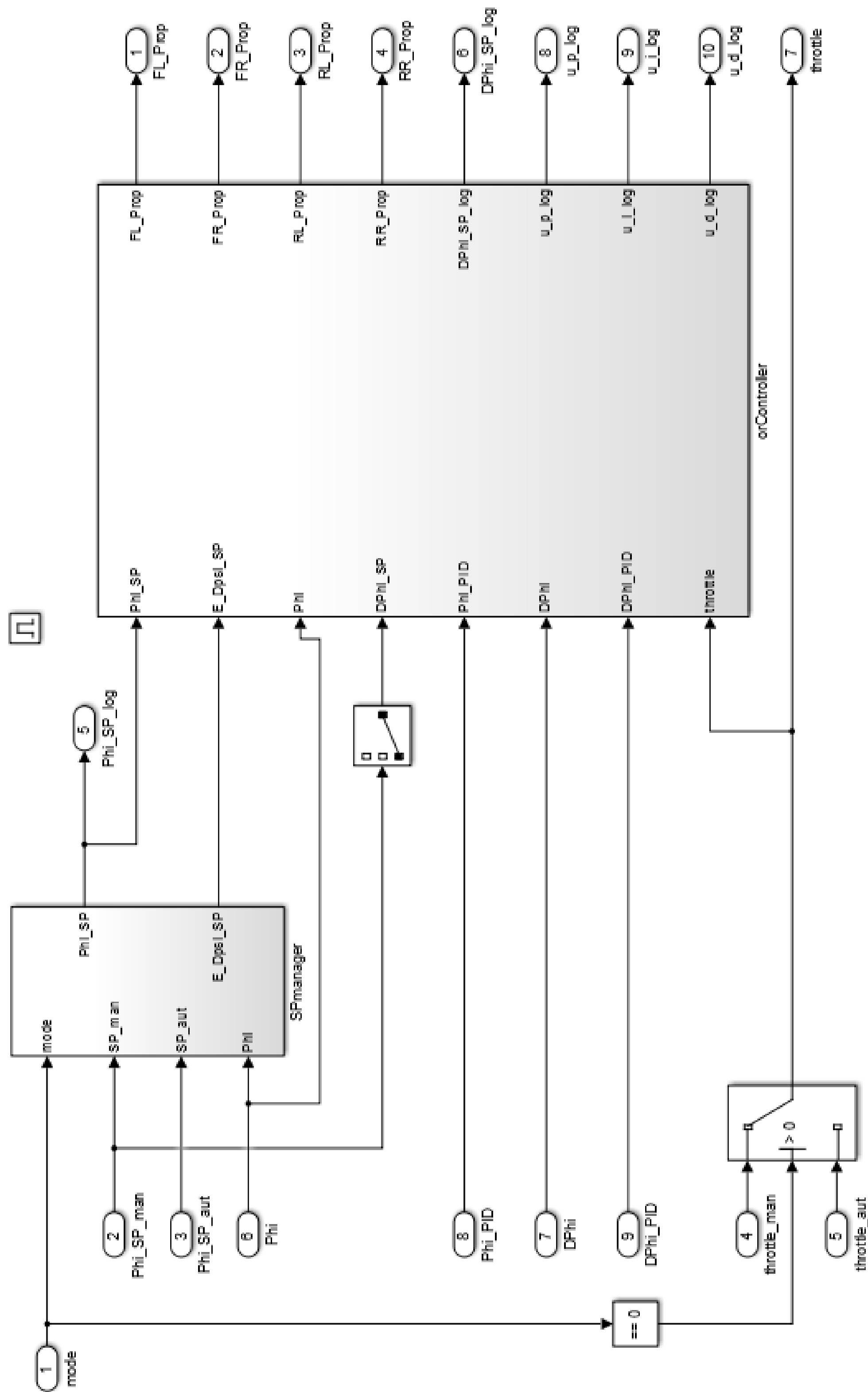


Figure C.20: Inside orientation block

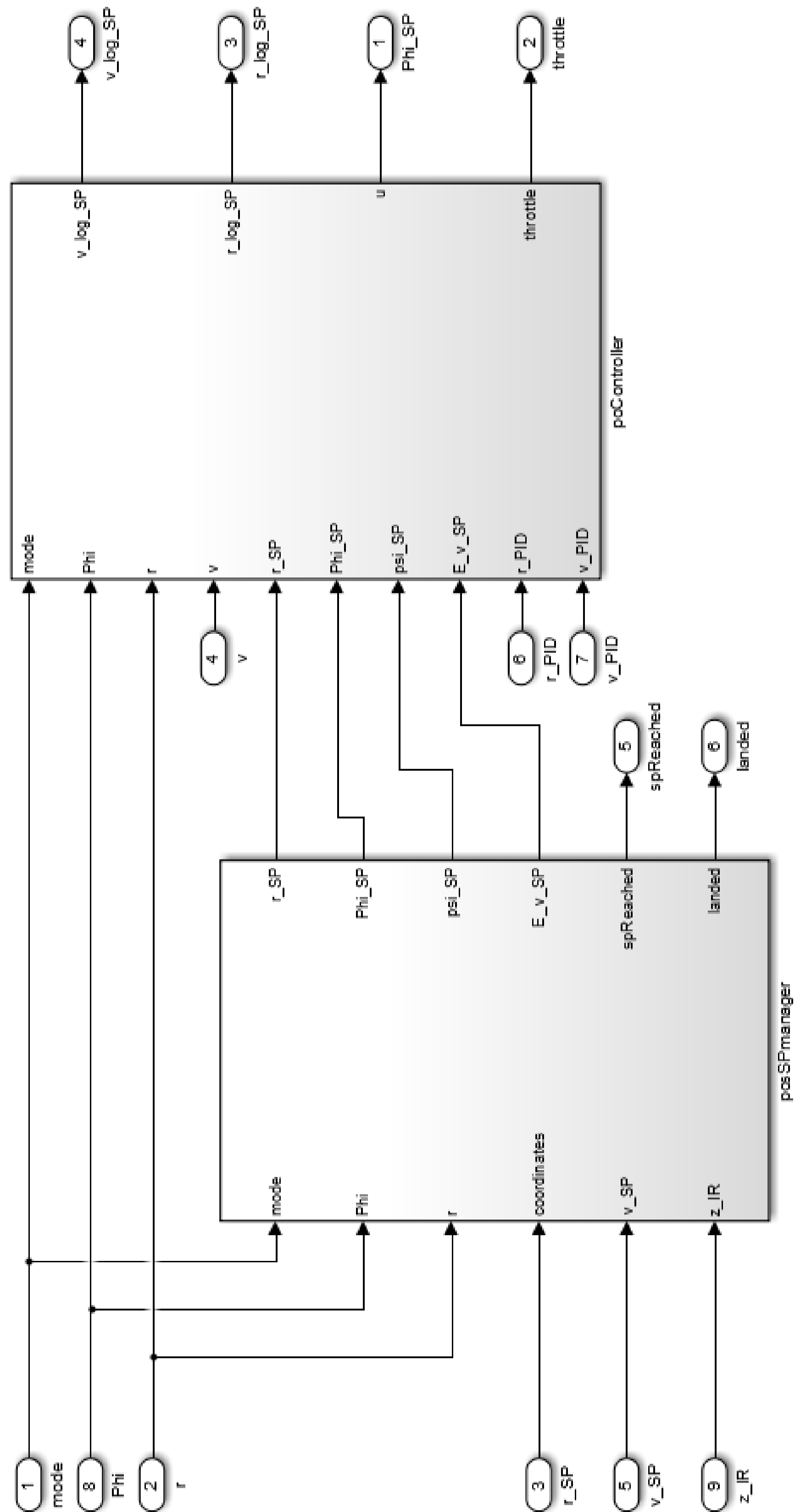


Figure C.21: Inside position block

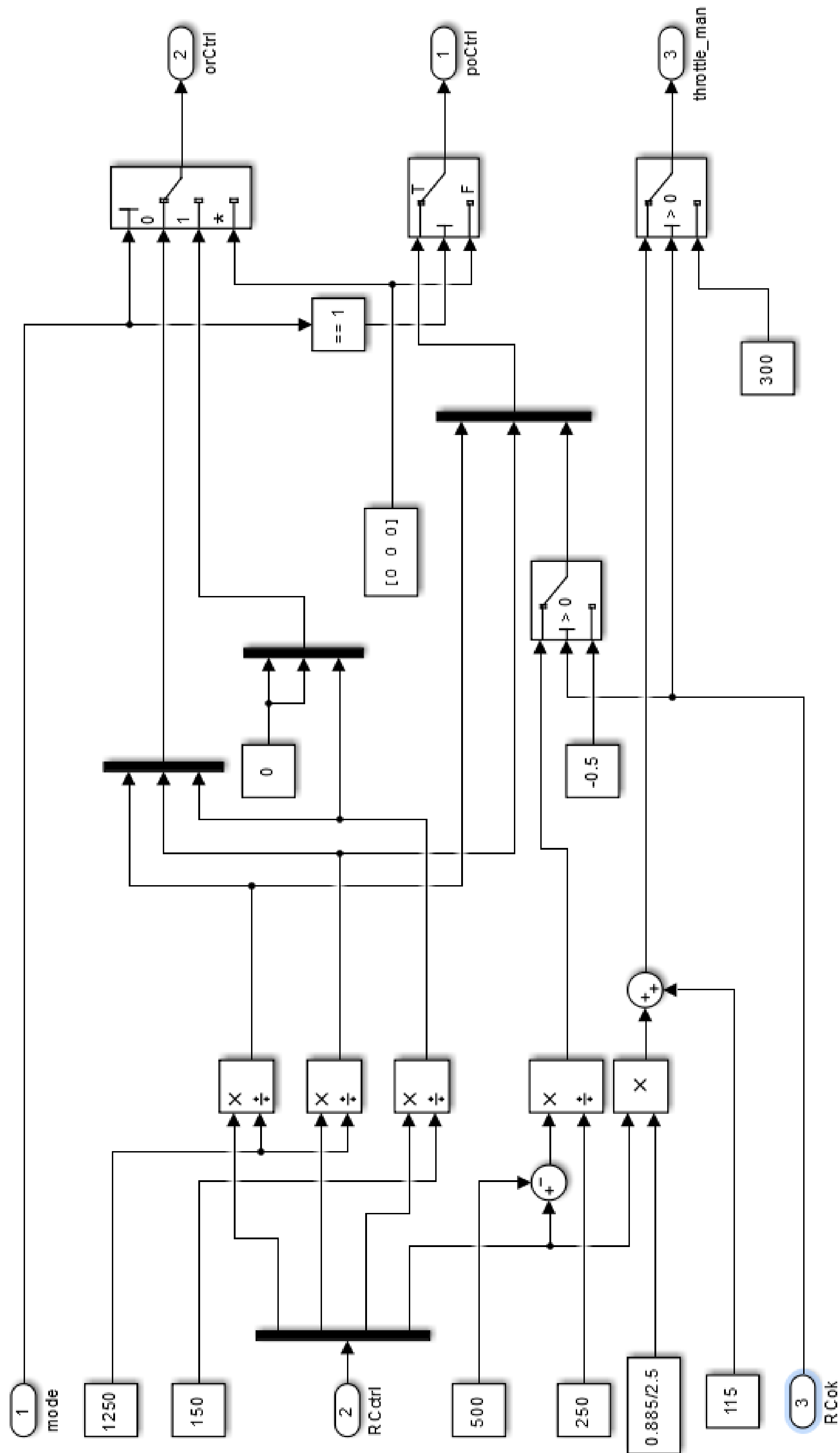
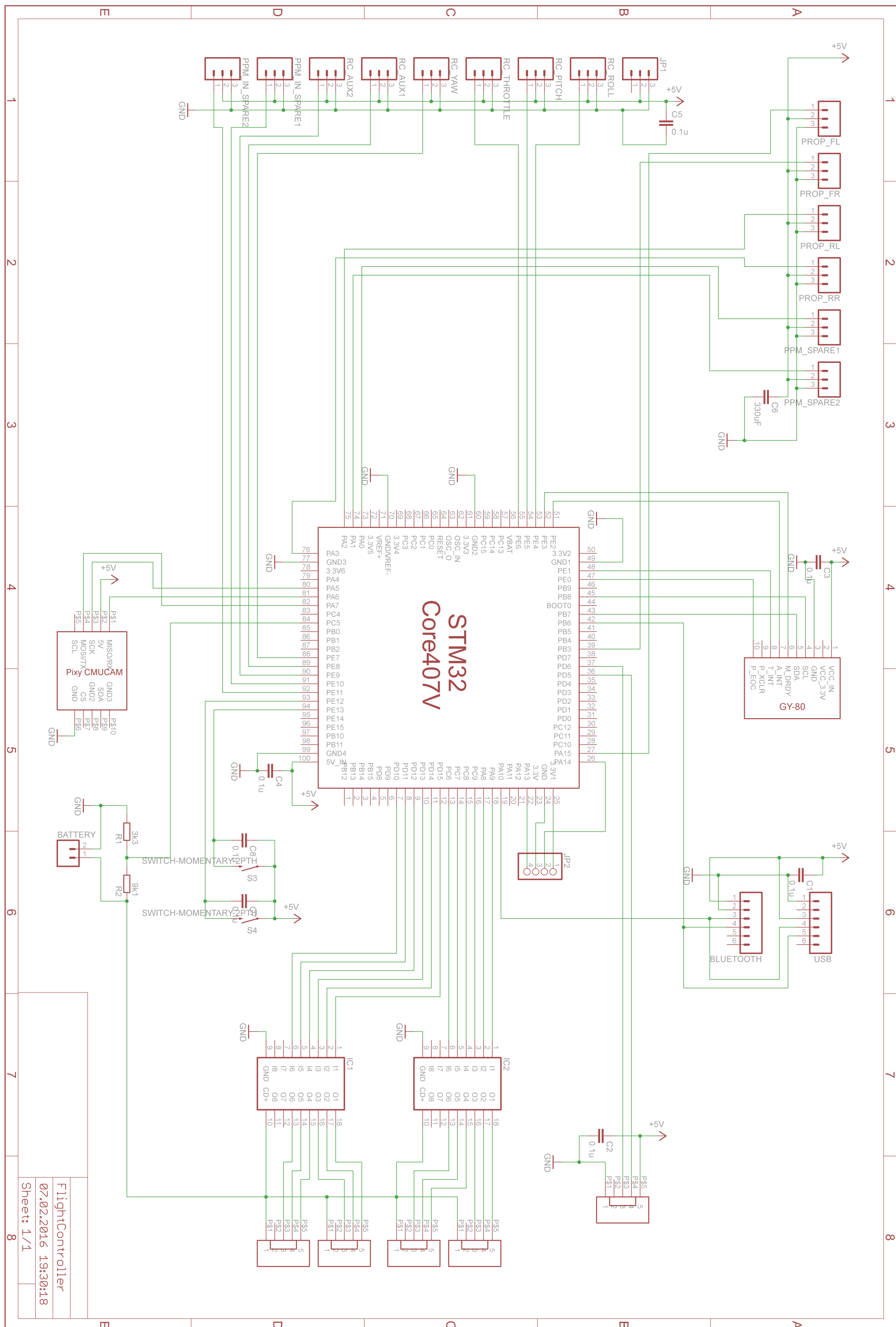


Figure C.22: Inside RC management block

Appendix D

Schematics

A picture of the schematics is presented on the next page. The Eagle project files are also referred to in appendix B.



Appendix E

IR sensor output characteristics

The characteristics are presented on the next page. The full datasheet for the IR sensor is reffered to in appendix B.

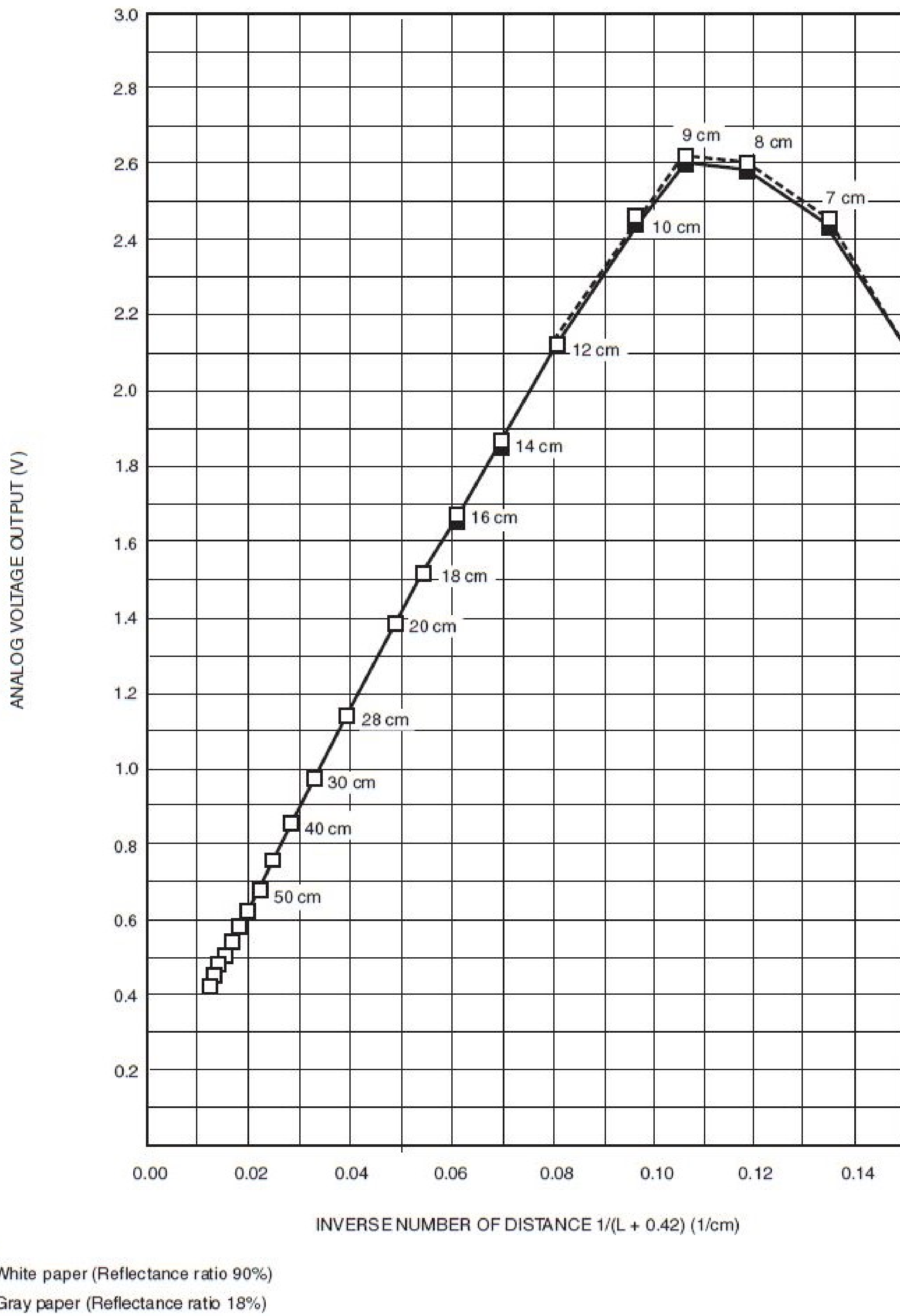


Figure E.1: IR sensor output characteristics

Appendix F

List of commands

Table F.1: Commands

Command	Parameter 1	Parameter 2	Parameter 3	Parameter 4	Description
orctrlparam	’ppitchroll’, ’pyaw’, ’vpitchroll’ or ’vyaw’	’p’, ’i’ or ’d’	<value>	N/A	Set orientation controler PID parameters
listorctrlparam	N/A	N/A	N/A	N/A	Print orientation controler PID parameters
poctrlparam	’pxy’, ’pz’, ’vxy’ or ’vz’	’p’, ’i’ or ’d’	<value>	N/A	Set position controler PID parameters
listorctrlparam	N/A	N/A	N/A	N/A	Print position controler PID parameters
setprop	propeller FL <value>	propeller FR <value>	propeller RL <value>	propeller RR <value>	Manually set propeller power
log	’on’ or ’off’	N/A	N/A	N/A	Turn logging on/off
getbatvolt	N/A	N/A	N/A	N/A	Print battery voltage
loadflight	x coordinate (East)	y coordinate (North)	y coordinate (Up)	N/A	Load flight plan
exeflightplan	N/A	N/A	N/A	N/A	Execute flight plan
clearflightplan	N/A	N/A	N/A	N/A	Delete flight plan
getflightplan	N/A	N/A	N/A	N/A	Print flight plan
getgpsstatus	N/A	N/A	N/A	N/A	Print GPS status

Appendix G

List of LED sequences

Table G.1: LED sequences

Status	LED FL leg	LED FR leg	LED RL leg	LED RR leg	Description
Ilde	Blue	Blue	Blue	Blue	LED's flashes in a ring, from FL leg to RL leg
Idle GPS lock OK	Green	Green	Green	Green	LED's flashes in a ring, from FL leg to RL leg
Armed manual mode	Green	Green	Yellow	Yellow	Front LED constant, back LED flashes two times per second
Armed GNSS mode	Green	Green	Blue	Blue	Front LED constant, back LED flashes two times per second
Armed autonomous mode	White	White	White	White	LED's flashes in a ring, from FL leg to RL leg
Low battery	N/A	N/A	Red	Red	Back LED flashes one time per second
Low low battery	N/A	N/A	Red	Red	Back LED are constant on
Calibrate compass z axis	Green	Off	Green	Off	Constant
Calibrate compass x- and y axis	Green	Off	Off	Green	Constant
Calibrate compass OK	Green	Green	Green	Green	Constant
Calibrate compass RC remote	White	White	White	White	Constant

Bibliography

[1] *Skywing SW-30A Data sheet.*

[2] *SMD 5050 RGB Technical Data Sheet.* Dreamland, www.yuanlei-led.com.

[3] (2005). *Optoelectronic Device Data Sheet.* Sharp, http://sharp-world.com.

[4] (2013). *u-Blox 7 Receiver Description.* U-blox, www.u-blox.com, 14 edition.

[5] (2015). *BMP085 Data Sheet.* Bosch, www.bosch-sensortec.com, 1.2 edition.

[6] (2015). *FreeRTOS Reference Manual.* Real Time Engineers ltd., www.freertos.org, 8.2.1 edition.

[7] (2015). *ULN2803A Darlington Transistor Array Data Sheet.* Texas Instruments, www.ti.com.

[8] Antsaklis, P. J. (2006). *Linear Systems.* Birkhäuser Boston, second edition.

[9] Beard, R. W. (2008). Quadrotor dynamics and control. Brigham Young University.

[10] Brown, R. G. and Hwang, P. Y. (2013). *Introduction to Random Signals and Kalman Filtering.* Courier Westford, fourth edition.

[11] CharmedLabs. Cmucam5 pixy. [Online; Accessed March 02, 2016] `http://www.cmucam.org/projects/cmucam5`.

[12] D'Andrea, R. Flight assembled architecture. [Online; Accessed May 31, 2016] `http://raffaello.name/projects/flight-assembled-architecture/`.

[13] ESA. Transformations between ecef and enu coordinates. [Online; Accessed February 02, 2016]

http://www.navipedia.net/index.php/Transformations_between_ECEF_and_ENU_coordinates.

[14] FreeRTOS. Freertos homepage. [Online; Accessed May 21, 2016]
http://www.freertos.org.

[15] Haugen, F. (2013). *Reguleringsteknikk*. Akademika Forlag, first edition.

[16] Kristensen, K. F. (2015). Matematiske metoder - lineær algebra. Lecture notes for EE3209.

[17] Kvæstad, B. (2015). Autonomous drone. Technical report, Norwegian University of Science and Technology.

[18] TheFancyVoyager. Best esc for quadcopters and multirotors. [Online; Accessed May 29, 2016]
http://thefancyvoyager.com/best-esc-for-quadcopters-and-multirotors.

[19] Vik, B. (2014). Integrated satellite and inertial navigation systems. Department of Engineering Cybernetics Norwegian University of Science and Technology.

[20] Wellings, B. (2009). *Real-Time Systems and Programming Languages: Ada, Real-Time Java and C/Real-Time POSIX, 4/E*. Pearson, fourth edition.

[21] Wolfson, R. (2012). *Essential Univercity Physics*, volume 2. Pearson, second edition.