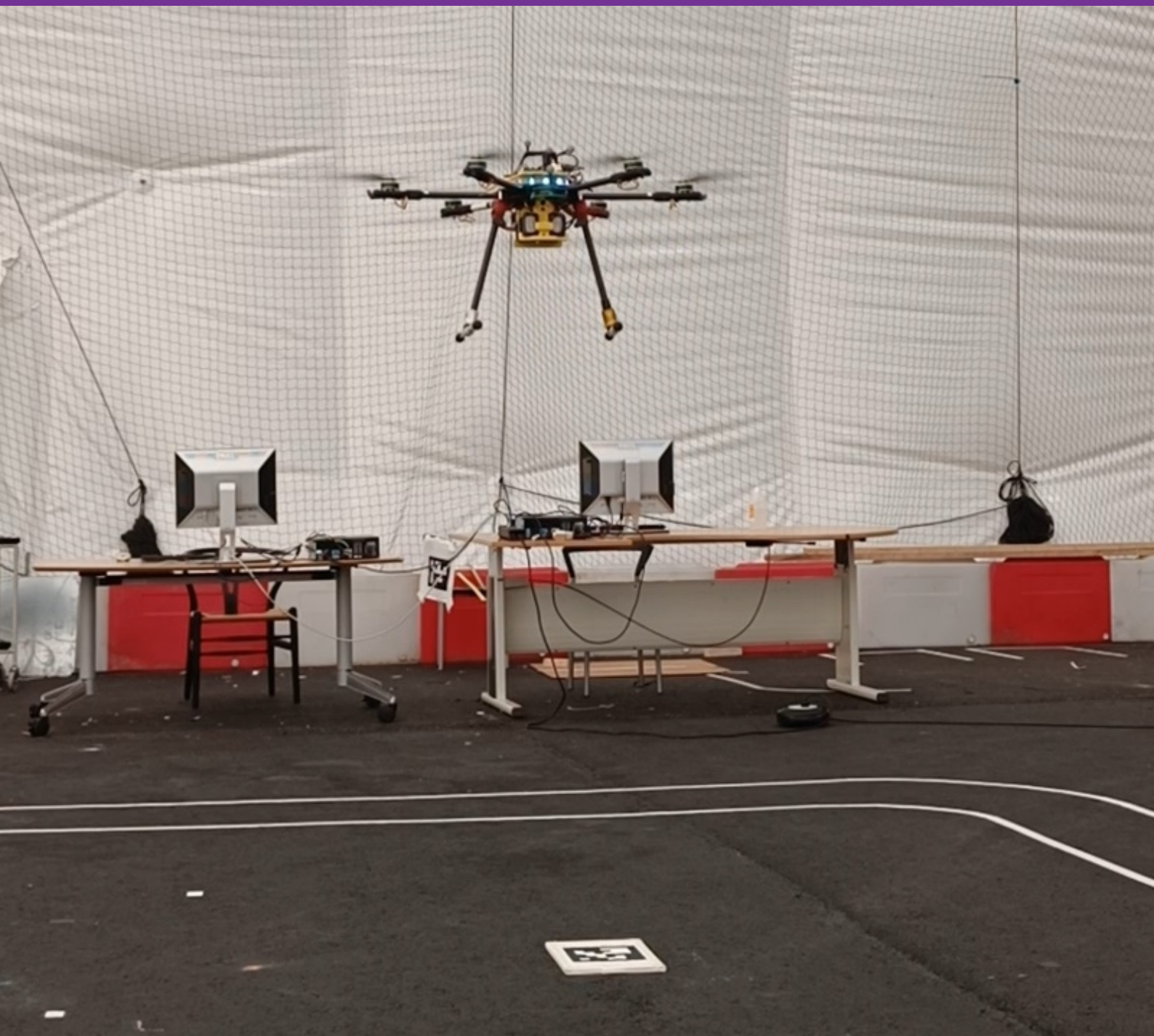
DTU ELECTRO
Department of Electrical and Photonics Engineering

Flight Controller for Autonomous Drones

Master Thesis

Marcus Christiansen

2023



Flight Controller for Autonomous Drones

By
(s213424) Marcus Christiansen

Advisor(s)
Søren Hansen, Associate Professor at the Electrical Engineering and Photonics Department of DTU
Nils Axel Andersen, Associate Professor at the Electrical Engineering and Photonics Department of DTU

Education	Master Thesis
Programme	Autonomous systems
Period	30 January 2023 to 30 June 2023
ECTS	30
Copyright:	Marcus Christiansen 2023
	Reproduction of this publication in whole or in part must include the customary bibliographic citation, including author attribution, report title, etc.
Cover photo:	Marcus Christiansen, 2023
Published by:	DTU, Department of Electrical and Photonics Engineering, Elektrovej 326, 2800 Kgs. Lyngby Denmark https://www.electro.dtu.dk
Remark	This report is submitted as partial fulfillment of the requirements for graduation in the above education at the Technical University of Denmark.

Abstract

This thesis aims to develop a versatile autonomous flight controller for a hexacopter, capable of precise pose estimation and performing tasks such as inspection or pickup tasks in predefined areas.

The thesis consists of several key chapters. In the Simulation chapter, the necessary equations to model a hexacopter are derived, and controllers for the X, Y, Z, and Yaw are designed based on predefined criteria. The controllers are then evaluated, based on their simulated behavior to see if they live up to the criteria.

The Real-Time System Implementation chapter focuses on the practical implementation of the controllers and two tracking methods. The controllers was successfully implemented using bilinear transform. Known issues with the drone were also addressed in this chapter, to ensure its readiness for flight.

In the Real-Time System Results chapter, the controllers are evaluated on how they act on a real system. Two height controllers are tested, which are a PI-Lead design and a Cascading P P-Lead design. The cascading design controller performed well with minor oscillations. The X-Y positional system demonstrated an accuracy within the allowed overshoot limits, defined at the beginning of the report. The Yaw controller maintained a current heading with a maximum deviation of 0.18 degrees. The Aruco tracking system showed good overall performance but had limitations related to accuracy decreasing the further away the drone was from the marker, possibly due to camera calibration, resolution, or misalignment.

Overall, the simulation and implemented controllers exhibited similar characteristics, suggesting a resemblance between the simulated and real-time systems. The results indicated that the drone met the specifications discussed in the introduction. However, fine-tuning the height control was recommended for improved landing stability.

Contents

1

Introduction

The general objective of this thesis is to create a versatile autonomous flight controller for multi-rotor-based aircraft such as quadcopters. In the case of this thesis, it will be a hexacopter configuration. The drone should be able to autonomously go to a destination, then do some task such as inspection or pickup task in a predefined area, with high precision. Therefore the drone has to be able to estimate its pose accurately when it reaches the desired destination.
What is wanted is therefore in general a framework that allows for basic functionality such as position control for quad or higher numbered rotor drones.

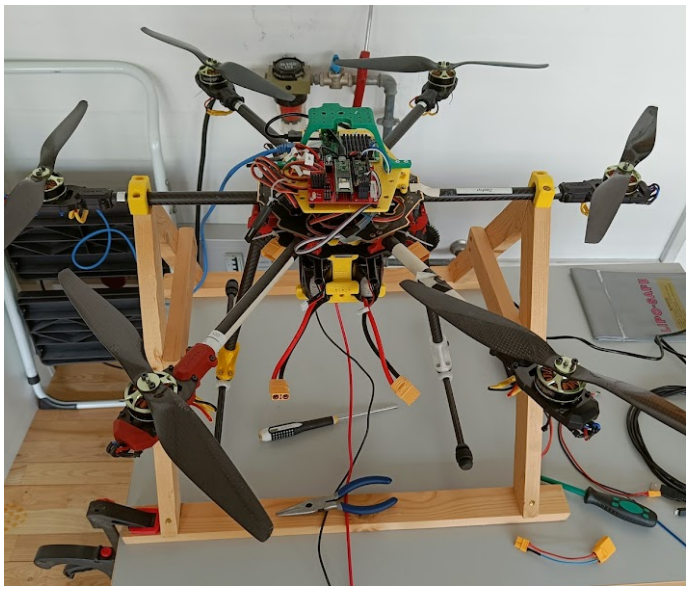


Figure 1.1: Zephyr Hexacopter

Mission
For the sake of having something more concrete, and also to have some requirements, this thesis will work on solving a hypothetical task. This task is as follows:

Deliver a parcel to a house

Fly to the house. The house is located 1 km from the takeoff location. When the drone is near the house, locate the landing pad which is 20×20 cm. The landing pad is located in the back garden and is close to a tree, which allows for an 8 cm margin of error in position. Place the parcel on the pad and return to base. When approaching a house, the drone should limit its speed to 1 m/s due to safety concerns of the occupants of the house. The low speed would ensure that if something happened and the drone was approaching a person, that person would be able to get out of the way.

For such a task the drone should be able to navigate far away from its original starting position, yet still be accurate down to a few centimeters, to accurately place the parcel.

1.1 Objectives & Requirements

Based on the mission, the following objectives and requirements have been derived. Note that these objectives are slightly different from the ones described in the project plan. However, the overall goal stays the same. The main goal is to have a proof of concept, which means that some assumptions will be made through the report.

1.1.1 Objectives

These are the objectives that this thesis will tackle. These objectives are based on what would be required to develop a flight controller capable of solving the mission.

1. Simulate a Hexacopter in Simulink.
2. Use the simulation to design a positional controller for X, Y, Z, and Yaw.
3. Implement controllers on a real-time system.
4. Implement a global positioning system for tracking over large distances.
5. Implement an accurate positioning system for precision tasks.
6. Evaluate if the drone would be able to do the parcel task.

1.1.2 Requirements

The requirements are set up to be, what the system should be capable of, but are not a necessity for a working product, but mainly to solve the mission. The simplicity requirement is mainly to ensure that if someone chooses to, they would be able to continue work on the project.

1. Autonomous navigation over large distances where the battery is the limiting factor.
2. High precision pose estimation in the task area (8 cm accuracy).
3. Keep it simple.

1.2 Project Limitations & Initial Conditions

The initial project is based on a previous project done by Jens Christian Andersen, as will be elaborated in more detail in Chapter 2. The initial project contains a codebase that should allow for manual control, although currently not working, together with a Simulink model of a Hexacopter, and the actual Hexacopter. The pre-built drone platform contains a Teensy, which handles attitude control, IMU data input, and RC control inputs. The drone also contains Raspberry Pi which handles position control and height control by sending commands to the Teensy. In short, this project will mainly be working with the Raspberry Pi. The reason for this setup is to ensure that it is always possible to take manual control of the drone, and in the case of a program crash caused by the autonomous logic running on the Raspberry Pi, the drone will continue to operate via RC remote.

1.2.1 Limitations

The limitations of the project can be summarized into the following points:

1. All testing has to be done indoors due to Danish drone regulations and safety.
2. The project builds on top of an existing project, which limits the project. For instance in regards to hardware. The software has to be able to run on a Raspberry Pi.
3. The time frame of the project is approximately 5 months.

1.3 Thesis Outline

This thesis is separated into six chapters. Intro which sets up what this thesis is trying to accomplice. The Analysis, where different approaches are discussed, so that they can be evaluated and one can be chosen to move forward with. Next up is the Simulation chapter, which will look at the dynamics of a quadcopter, how it can be simulated, and most importantly, how this can be used to design a controller. Next is the Real-Time System Implementation, which explains how the drone is setup and how things from the previous chapters such as controllers are implemented on the actual drone. Then it's the Real-Time System Results chapter that compares the simulated results to real-world results, measured from the drone. This chapter also discusses some of the issues encountered. Next is the Conclusion, which will draw an overall conclusion based on all the findings in the previous chapters. The final chapter is future works, which discuss potential further development of the project, were someone to continue it.

2

Analysis

This chapter will look at known flight controllers and look at some of their advantages and disadvantages of them, together with some of the methodologies required to build a flight controller, and what is required to fulfill the objectives mentioned in Section 1.1.

It is important to state that there are multiple approaches to solving the objectives. This chapter will look into the different approaches for the objectives and argue why a specific approach was chosen. The approaches chosen in this chapter might not be the absolute best but will serve as a great foundation. It's important to state that the goal is not to find the best solution, but just to find a good solution that will work to solve the hypothetical task mentioned in Chapter 1.

2.1 Flight Control

When it comes to drone control software (Also known as flight controllers), there are typically two commercial options: DJI or Pixhawk. DJI's flight controllers are robust and have a great ecosystem of sensors and good documentation. They are however fairly limited in some aspects. The framework for instance is not open source. Even though the flight controller is great, you are still somewhat limited to whatever functionality DJI's interface allows. For instance, as it stands right now, their GEO-fencing system cannot be disabled without contacting DJI [1]. In regards to Pixhawk, it is open-source and widely used. It works for multiple types of drones and is also very well documented. But this versatility comes at a cost, which is that the project is very large. A third and somewhat different option is to work with the Flight Controller developed by Jens Christian Andersen. This controller is completely open-source, which would be ideal for this project. However, as the controller is right now, it currently has some bugs which means that it currently does not work.

2.1.1 Decision

All three solutions described above would be suitable for the task at hand, and all three solutions have their pros and cons. Nevertheless, for this project, Jens Christian Andersen's flight controller has been chosen. It removes the restrictions set by DJI's flight controller API, while also reducing the overall codebase due to the fact that it was designed only to work on multi-rotor aircraft such as quadcopters. Even though this controller will require

more work, it also gives the opportunity to later expand the project, if someone were to wish to continue the project. Thus enabling continuous workflow.

2.2 Choice of Controller Type

Designing feedback control for a dynamic system can be done in a multitude of manners. Some of the commonly used approaches are *Proportional Integral Derivative* (PID), *Linear Quadratic Regulators* (LQR), and *Model Predictive Control* (MPC).
Most drones and flight software such as Pixhawk use classical PID controllers [2]. When it comes to LQR, the general consensus is that they are often better when it comes to steady-state error than PID, but typically at the cost of fast response [3]. Furthermore, both LQR and MPC are optimization approaches to the control design, whereas PID's are more about designing and a bit of trial and error. This means that it can be more difficult to tune LQR's and MPC's, compared to PID's in case the simulated model does not match the real system. The article "Design and Experimental Comparison of PID, LQR, and MPC Stabilizing Controllers for Parrot Mambo Mini-Drone" [4], compares the three different approaches and concludes that in general all three work well, but that LQR and MPC slightly outperform PID. However, the article also acknowledges that it was easier to implement and tune a PID controller afterward.

2.2.1 Decision

For this project, the choice of controller will be PID, mainly because of the following reasons:

- As the article above mentions, they found them easier to implement and fine-tune.
- The codebase made by Jens Christian Andersen uses PID controllers to control the attitude of the drone. It would therefore make sense to keep the approach the same, to not add an extra layer of complexity on top.

2.3 Global Tracking

In the context of this thesis, global tracking refers to some system that can track the drone in a large outdoor area.

Global tracking is typically done using the Global Navigational Satellite System (GNSS). The accuracy of such a system generally depends on which satellite constellation used, and the type of receiver carried by the drone. In general, it can be expected to have a precision of 0.6 meters [5].

A different approach could be to use a vision-based SLAM. This however is a lot more compute-intensive, and would also require a lot more work to get up and running. Even so, the drone would only be able to navigate around known areas.

2.3.1 Decision

Vision-based slam is an interesting approach and could be something implemented in the future. However, as a proof of concept, GNSS will suffice to ensure that the drone reaches the area where it's supposed to operate. However, due to the fact that all testing will be done indoors due to the size of the drone and Danish regulations, OptiTrack's indoor position estimation system [6] will be used. This system works a bit like GNSS, but has far greater accuracy than a GNSS system and could also be used as the internal precision tracking system, but for the sake of simulating a real-world scenario, it will only be used to guide the drone to a specified operating area.

2.4 Precision Tracking

The Precision Tracking needs to be precise down to 8cm, since this would allow the drone to do the task described above, but preferably less.

A potential approach could be using a depth camera and image classification. This would allow the drone to find the landing pad and estimate its position relative to it.
A simpler approach is to use Aruco markers on the landing pad. Aruco markers allow the drone to not only estimate its position relative to the marker but also estimate the marker's orientation in relation to itself.
A third approach would be to equip the landing area with a base station such as the *Here 3+* [7]. This system guarantees a precision down to 6 cm.

2.4.1 Decision

For this project, Aruco markers will be used as the Precision Tracking. The reason for this is its simplicity and low cost of implementation. A system such as using a depth camera could be ideal, but most of depth cameras are fairly limited when used outside, due to them using infrared to estimate depth, which would get interference from sunlight. The *Here 3+* system is also great but requires a base station at each landing site. Furthermore, it is not a cheap system, and the simplicity and low requirements for maintenance of Aruco markers can't be beaten by such a system.

2.5 Summary

For flight controllers, this thesis will further develop Jens Christian Andersen's framework. The main reason behind this, is that it allows for a great deal of flexibility, eventhough it requires more work. For the positional feedback control, classical PID controllers will be used. As mentioned above, classical PID controllers should be up for the tasks. Furthermore, flight controllers such as Pixhawk use them, which proves that they are fit for the task. For external/global tracking of the drone, OptiTrack will be used. The main reason for this is that there aren't a lot of other options, without adding a layer of complexity. And finally, for precision pose estimation, the drone will use Aruco markers.

3

Simulation

This section will derive the equations needed to simulate the drone in Matlab. Note that all the constants have been substituted with variables for easier readability, but also due to the fact that most of the variables such as inertia, and specifics needed to simulate the motors, were already found via testing by Jens Christian Andersen. The primary purpose of this chapter is to give the foundation that lies behind the Simulink model done by Jens Christian Andersen, such that it is easier to understand the changes made to it and why they were made.
The equations are heavily inspired by the work done in the article ”Dynamics modeling and linear control of quadcopter” [8], together with the course *Unmanned Autonomous Vehicles*, held at DTU.

3.1 Dynamics

Drones such as Zephyr have 6 degrees of freedom, *pitch*, *yaw*, *roll*, *x*, *y*, and *z*. The way the drone controls these is by varying the speed of each of the motors. Each motor generates torque and lift based on its rotational speed and the drag of the propeller. To easier understand how the angular force and linear force act on each other, the dynamics will be split into three: Motor dynamics, linear dynamics, and rotational dynamics. The

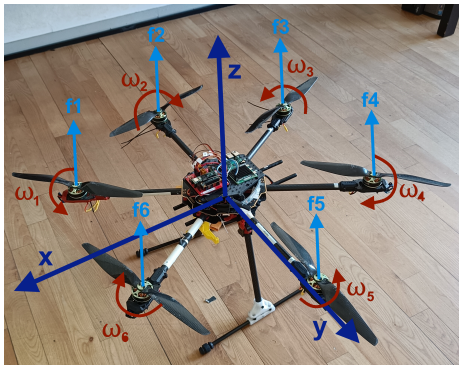


Figure 3.1: Image showing frame of drone and rotational direction of propellers

drone will be modeled as a rigid body, with its own inertial frame shown in dark blue in the

figure above. $Pitch : \theta$ is rotation around the y-axis, $Roll : \phi$ is rotation around the x-axis, and $Yaw : \psi$ is rotation around the z-axis.

3.1.1 Symbols

This section is meant as a reference guide for the symbols used in the equations below. Most of the time the definition will also be described where it is used.

- ω_i : Rotational velocity of motor i.
- m : Mass of the drone.
- τ : Torque
- K : Relationship between the rotational velocity of a propeller and the lift it generates
- K_d : Viscosity of air
- L_x : The distance along the x-axis for a common point of lift, for f1 + f6 and f3 + f4.
- L_y : The distance along the y-axis for a common point of lift, for f1 + f2 + f3 and f4 + f5 + f6 (See Figure 3.1).
- b : Relation between the aerodynamic drag of the propellers and their velocity.
- $R(\theta)$: Rotation matrix.
- $\vec{\omega}$: Vector describing angular velocity around x, y and z.
- K_{motor}: Motor constant.
- J: Moment of inertia for the rotor.
- K_{prop}: Propeller drag related to angular velocity.

3.1.2 Motor Dynamics

The motor dynamics will be used to describe the relationship between the input voltage and angular velocity of the motor's output shaft. All the constants such as resistance, propeller drag, and so on, were calculated previously by Jens Christian Andersen and can be found on rsewiki [9]. Even though the drone uses BLDC motors, it can still be modeled as a DC motor, where the only change is that the input to the motor is a PWM signal. This means that the motor's input will be a signal between $1 - 1024$ multiplied with a constant dependant on the battery pack voltage $\frac{1024}{BatteryMaxVoltage}$. The first step is

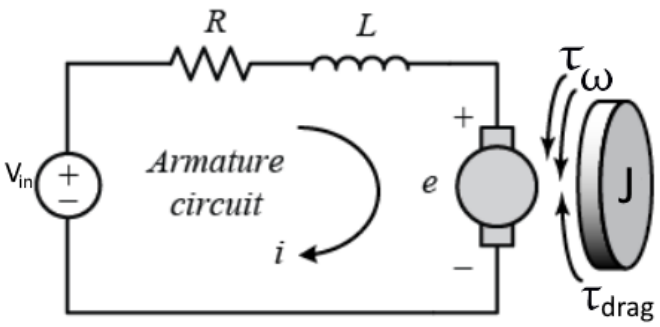


Figure 3.2: DC motor model. Image source Control Tutorials for Matlab and Simulink [10]

to apply Kirchofs voltage law, which states that the sum of voltage differences around any

closed loop in a circuit must be zero.

$$V_{in} - R \cdot i - L \cdot \dot{i} - e = 0 \tag{3.1}$$

An assumption can be made that the inductance is negligible and will therefore be ignored from this point on.

The propeller generates some torque in the opposite direction of the spinning rotor, which is related to the rotors' rotational speed.

$$\tau_{drag} = \omega^3 \cdot K_{prop} \tag{3.2}$$

This is important since it affects the amount of torque the motor is able to generate. The constant K_{prop}, was found through testing done by Jens Christian Andersen [9]. With this equation for torque in the opposite direction, the motor's torque can be described by the following equation.

$$\tau_{motor} = K_{motor} \cdot i - \tau_{drag} \tag{3.3}$$

The back emf e from the motor is derived by the following equation:

$$e = K_{motor} \cdot \omega \tag{3.4}$$

Putting this together with Kirchofs voltage law, can give an equation for i.

$$i = \frac{V_{in} - K_{motor} \cdot \omega}{R} \tag{3.5}$$

The mechanical equation for the motor's torque can be described as follows.

$$\tau_{motor} = J\dot{\omega} \tag{3.6}$$

$$\frac{\tau_{motor}}{J} = \dot{\omega} \tag{3.7}$$

Now τ_{motor} can be substituted with its electrical definition

$$(K_{motor} \cdot i - \tau_{drag}) \cdot \frac{1}{J} = \dot{\omega} \tag{3.8}$$

Now i can be substituted into Equation (3.1).

$$(K_{motor} \cdot (V_{in} - K_{motor} \cdot \omega) \cdot \frac{1}{R} - \tau_{drag} \cdot) \frac{1}{J} = \dot{\omega} \tag{3.9}$$

The final step is just to integrate $\dot{\omega}$ to get the rotational velocity ω. Now that the relationship between voltage and rotational velocity is established, it can be modeled in Simulink. As mentioned this system is implemented using Simscape blocks in Simulink. The only difference is that the equation required is the motor torque, since Simscape takes care of the calculation from torque to rotational velocity.

3.1.3 Linear Dynamics

Now that the relation between propeller speed and voltage has been derived, it can be converted into an upwards force. The first step is to get the force, that the propellers generate in the drone's reference frame. The fact that the drone has fixed motors only, means that all force generated by the propellers gets applied perpendicular to the z-axis (See turquoise arrows in Figure 3.1). This can be expressed by the following equation:

$$F_{prop} = \begin{bmatrix} 0 \\ 0 \\ \Sigma_{i=1}^{6} \omega_i^2 \end{bmatrix} * K \tag{3.10}$$

K is the relationship between the rotational velocity of a propeller and the lift it generates, and ω_i is the rotational velocity of a specific propeller. The F_{prop} force is relative to the drone's frame.

The next step is to derive the full linear dynamics for the drone. These dynamics will take gravity and the viscosity of the air into account. These dynamics will not take things such as the ground effect into account. The linear dynamics are given by the following equation:

$$\begin{bmatrix} m & 0 & 0 \\ 0 & m & 0 \\ 0 & 0 & m \end{bmatrix} \cdot \begin{bmatrix} \ddot{x} \\ \ddot{y} \\ \ddot{z} \end{bmatrix} = R(\theta) \cdot F_{prop} + \begin{bmatrix} -K_d \cdot \dot{x} \\ -K_d \cdot \dot{y} \\ -K_d \cdot \dot{z} \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ -m \cdot g \end{bmatrix} \tag{3.11}$$

Where m is the mass of the drone, K_d is the viscus damping coefficient of the air that the drone is flying through and g is gravity. As noted the propeller force is on the z-axis in the drone's inertial frame. To get it to the global frame, this force is multiplied with a rotational matrix that converts it from inertial to global.

3.1.4 Rotational Dynamics

Rotation of a hexacopter is done by varying the speed of the motors. For instance, if it is desired to rotate the drone around its x-axis, the drone should increase the propeller speed on motors 1, 2, and 3, while decreasing it on 4, 5, and 6. This operation shifts the placement of the force vector away from the drone's center of mass, thus inducing a rotation. The same can be done for the y-axis, although the motors that have to increase and decrease thrust is different.
Rotation around the z-axis is done a bit differently. Here the drone relies on the fact that the propellers due to drag, generate torque in the opposite direction of their rotation. By making all the even-numbered motors rotate the opposite way of the odd-numbered ones, it is possible to generate a higher torque in one direction by setting the rotational speed of the even-numbered motors higher or lower than the odd-numbered ones. Shown below are the equations describing the torque around x, y, and z respectively.

$$Roll : \tau_\phi = L_y \cdot K \cdot ((\omega_4^2 + \omega_5^2 + \omega_6^2) - (\omega_1^2 + \omega_2^2 + \omega_3^2)) \tag{3.12}$$

$$Pitch : \tau_\theta = L_x \cdot K \cdot ((\omega_3^2 + \omega_4^2) - (\omega_1^2 + \omega_6^2)) \tag{3.13}$$

$$Yaw : \tau_\psi = b \cdot (\omega_6^2 - \omega_5^2 + \omega_4^2 - \omega_3^2 + \omega_2^2 - \omega_1^2) \tag{3.14}$$

With the torque around each axis, it is now possible to calculate the velocity around each axis. Since angular velocity is $Torque/Inertia$.

$$\dot{\vec{\omega}} = \begin{bmatrix} \tau_\phi * I_{xx}^{-1} \\ \tau_\theta * I_{yy}^{-1} \\ \tau_\psi * I_{zz}^{-1} \end{bmatrix} - \begin{bmatrix} \frac{I_{yy} - I_{zz}}{I_{xx}} \cdot \omega_y \cdot \omega_z \\ \frac{I_{zz} - I_{xx}}{I_{yy}} \cdot \omega_x \cdot \omega_z \\ \frac{I_{xx} - I_{yy}}{I_{zz}} \cdot \omega_x \cdot \omega_y \end{bmatrix} \tag{3.15}$$

With these equations, we can give a rotational velocity on a given motor and simulate how the drone will behave.

3.1.5 Motor Mixing Algorithm

Before designing the model, it would be wise to look at designing a way such that instead of having 6 motor inputs, the model will have *Pitch*, *Yaw*, *Roll*, and *Throttle* as inputs. This can be achieved by implementing a motor mixing algorithm [11].

$$Thrust - Pitch + Yaw - Roll = \omega_1 \tag{3.16}$$
$$Thrust - Yaw - Roll = \omega_2 \tag{3.17}$$
$$Thrust + Pitch + Yaw - Roll = \omega_3 \tag{3.18}$$
$$Thrust + Pitch - Yaw + Roll = \omega_4 \tag{3.19}$$
$$Thrust + Yaw + Roll = \omega_5 \tag{3.20}$$
$$Thrust - Pitch - Yaw + Roll = \omega_6 \tag{3.21}$$

A Matlab implementation of this controller would look as follows:

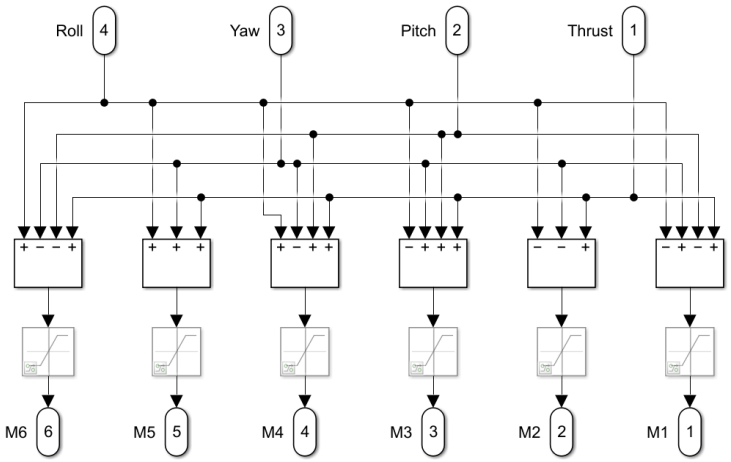
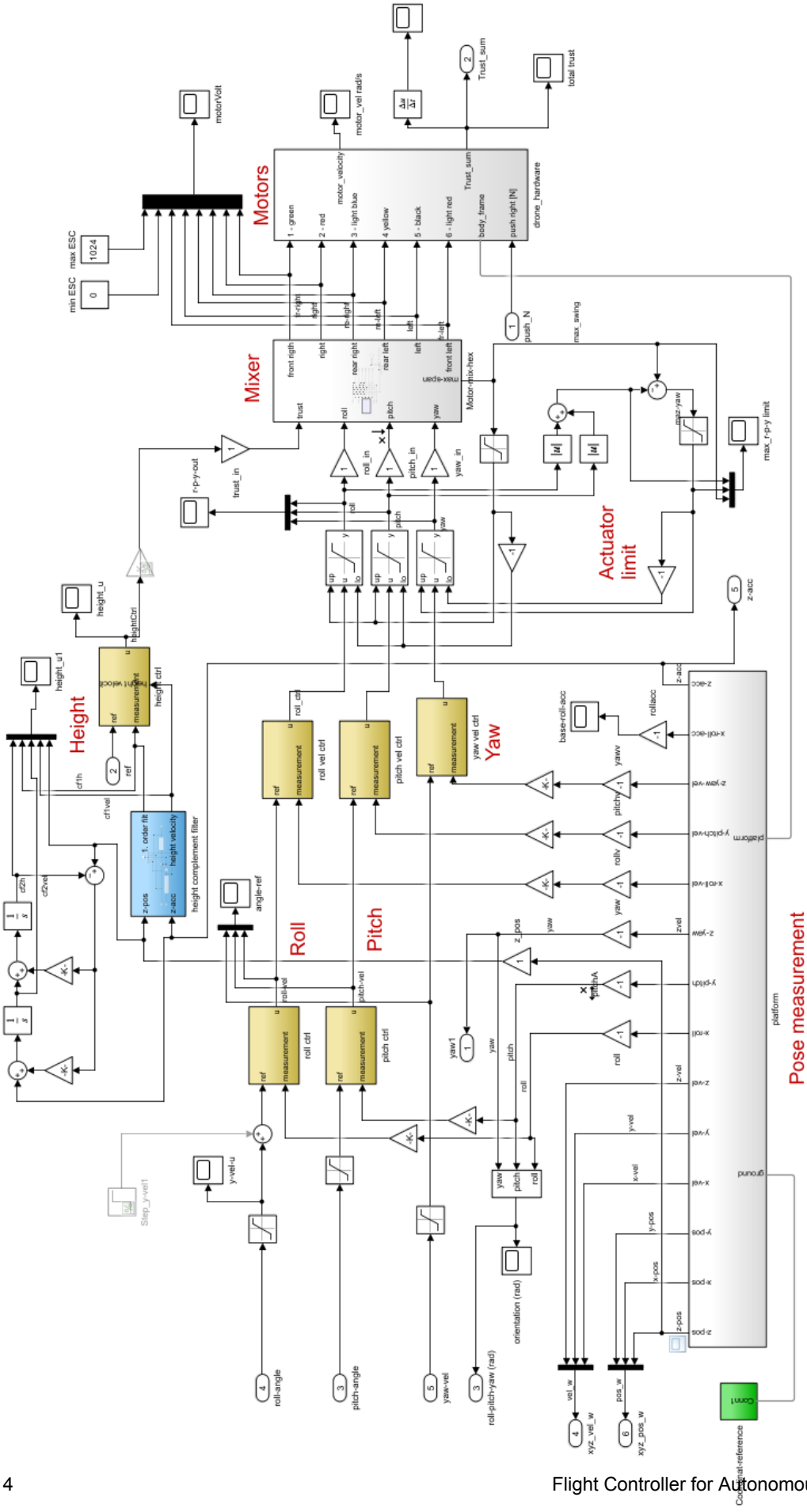


Figure 3.3: Motor Mixing Algorithm Matlab implementation

3.2 Simulink Model

The base Simulink model is shown in Section 3.2. It looks fairly complicated but in general it's actually pretty simple. All the equations from Section 3.1 are inside the block called *Motors*, *Mixer* and *Pose Estimation*. The dynamics for the motors, together with the linear dynamics of the drone and the rotational dynamics are all done using Simscape, which is an extension to Matlab that simplifies a lot of the calculations previously done, down to simple blocks.


Motors
Mixer
Actuator limit
Height
Roll
Pitch
Yaw
Pose measurement
motorVolt
motor_vel rad/s
Trust_sum
total trust
drone_hardware
Motor-mix-hex
max_r-p-y limit
height complement filter
roll ctrl
pitch ctrl
yaw vel ctrl
platform
ground
orientation (rad)
roll-pitch-yaw (rad)
roll-angle
pitch-angle
yaw-vel
xyz_vel_w
xyz_pos_w
Comm1
Comand-reference

The orange blocks are controllers for pitch angle, pitch angular velocity, roll angle, roll angular velocity, yaw velocity, and height. All of these controllers were already made and implemented by Jens Christian Andersen, with the exception of the height controller due to some issues which are described in Section 4.2.3. The blue block is a complementary filter fusing together barometer and accelerometer data to estimate height. To summarize, with the model described above, it is possible to give it a specific pitch or roll angle and it will then hold it.

3.2.1 Global Frame to Local Frame Pitch & Roll

The drone can take pitch, roll, yaw, and throttle commands. However, as stated earlier, it is desired that the drone should be able to be given some x and y coordinates, and then go to that location. If the drone aligns perfectly with the global frame, this task is trivial, since pitch would control the rotation of the drone around the y-axis and roll would control the rotation of the drone around the x-axis. However, this would limit the drone's capabilities. It is therefore desired to create a Matlab subsystem that can transform x and y coordinates to pitch and yaw. A simple approach would be to apply a rotation matrix as follows:

$$\begin{bmatrix}\phi\\ \theta\end{bmatrix} = \begin{bmatrix}0 & 1 & 0\\ 1 & 0 & 0\end{bmatrix} \cdot R(\theta)_{zyz} \cdot \begin{bmatrix}x_{angle}\\ y_{angle}\\ 0\end{bmatrix} \tag{3.22}$$

Which can be expanded to:

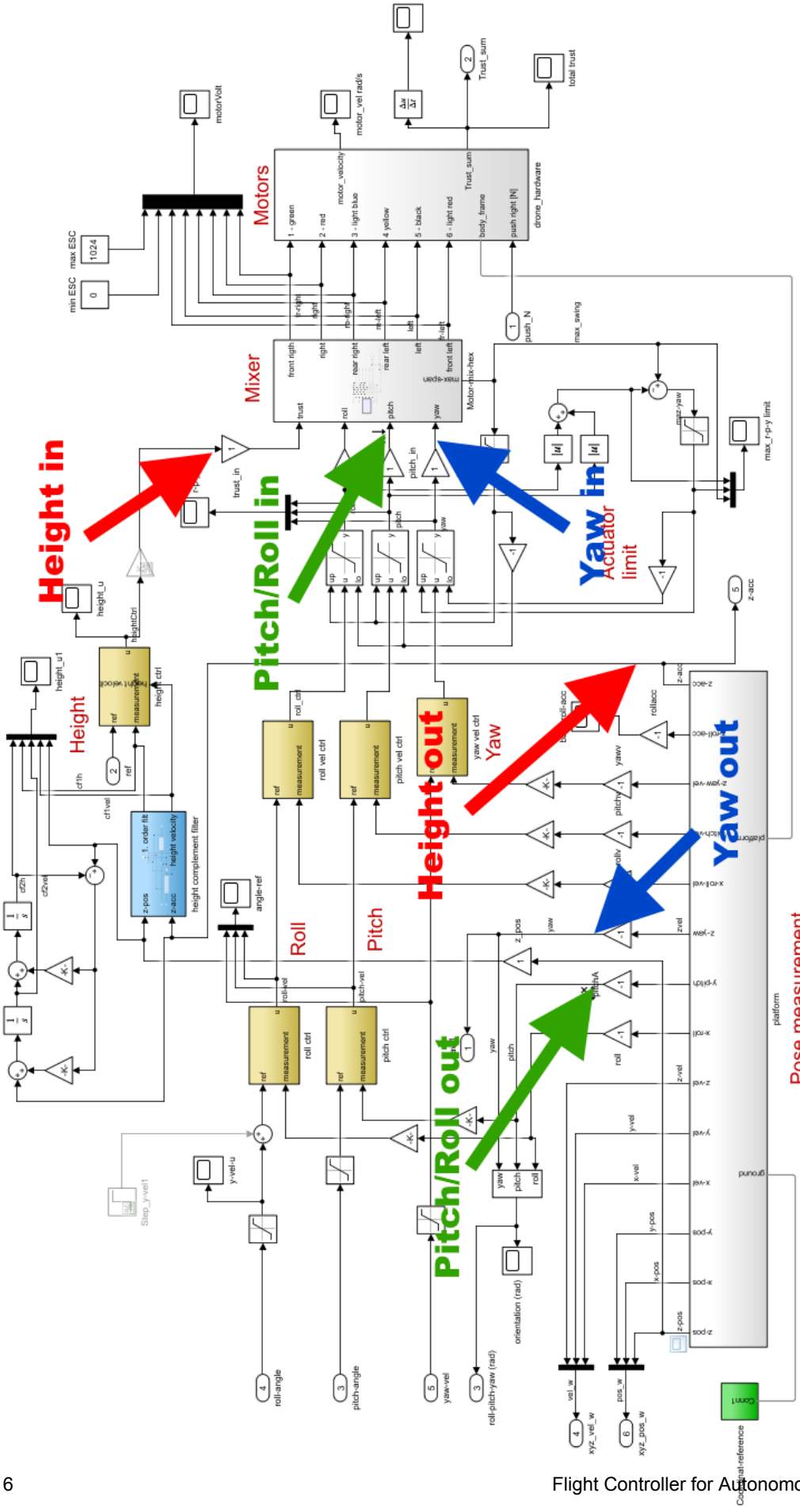
$$\begin{bmatrix}\phi\\ \theta\end{bmatrix} = \begin{bmatrix} x_{angle} \cdot (\cos\phi \cdot \sin\psi + \cos\psi \cdot \cos\theta \cdot \sin\phi) + y_{angle} \cdot (\cos\phi \cdot \cos\psi - \cos\theta \cdot \sin\phi \cdot \sin\psi)\\ -x_{angle} \cdot (\sin\phi \cdot \sin\psi - \cos\phi \cdot \cos\psi \cdot \cos\theta) - y_{angle} \cdot (\cos\psi \cdot \sin\phi + \cos\phi \cdot \cos\theta \cdot \sin\psi)\end{bmatrix} \tag{3.23}$$

Now it is possible to give the drone rotation commands around the global x and y axis instead of its local axis. However, for safety reasons, the drone will be limited to 10 degrees of bank around its x and y-axis. It is therefore possible to make the assumption that the drone will never reach an x and y bank angle where it is necessary to take this into account. The equation above can therefore be simplified down to a simple rotation around the z-axis as shown in Equation (3.24). The only scenario where it would make sense, is if the drone was configured as a race drone, that should be able to fly inverted.

$$\begin{bmatrix}\phi\\ \theta\end{bmatrix} = \begin{bmatrix}x_{angle} \cdot \sin\psi + \cos\psi \cdot y_{angle}\\ x_{angle} \cdot \cos\psi - \sin\psi \cdot y_{angle}\end{bmatrix} \tag{3.24}$$

3.3 Transfer Functions for System

To design the controllers described in Section 3.5, it is necessary to gather the transfer functions that describe the movement of the drone given some input, such that the controller can be tuned. For instance, to tune and design the controller responsible for x velocity, the transfer function (TF) wanted, is the one that describes the relation between the drone's angle and its velocity.


Height in
Pitch/Roll in
Yaw in
Actuator limit
Height out
Yaw out
Pitch/Roll out
Pose measurement
Motors
Mixer
Height
Roll
Pitch
Yaw

As stated in the introduction this project builds on top of an already existing project done by Jens Christian Andersen. Section 3.3 shows the transfer functions which were already given and what they actually describe. In the subsections below, the known controllers shown in the image will be added on top, such that the end product is a transfer function that can be used to tune the new controllers, which will be designed in Section 3.5. The most important part about these transfer functions is that they capture the time constants from the dc motors, which tells how the poles from the motor act.

3.3.1 Pitch & Roll Transfer Function

The drone is fairly symmetrical, which entails that it should only be necessary to design a controller for pitch, and then copy that configuration and parameters over to the controller for roll

Linearizing the drone in a hover gives the following transfer function:

$$pitch_{vel}(s) = \frac{878.7}{3.142 \cdot s^2 + 47.44 \cdot s} \tag{3.25}$$

As stated earlier the controller settings for pitch velocity and pitch position are already known, so to get the transfer function required to design the X-Y position controllers, the previous controllers need to be added to the transfer function. Both Pitch velocity and Pitch angle position are P-Lead controllers.

$$\text{P-Lead Controller} : C(s) = kp \cdot \frac{\tau_d \cdot s + 1}{\alpha \cdot \tau_d \cdot s + 1} \tag{3.26}$$

$$\text{Velocity controller} : C_{vel}(s) = 1.0378 \cdot \frac{0.0801 \cdot s + 1}{0.5000 \cdot 0.0801 \cdot s + 1} \tag{3.27}$$

$$\text{Angle controller} : C_{pos}(s) = 7.57 \tag{3.28}$$

$$\tag{3.29}$$

The Equation (3.26) shows the default controller values. What is wanted is a transfer function that describes the approximate relationship between the pitch angle of the drone and its speed. With all the parameters for the angular velocity controller, it is possible to calculate the closed-loop transfer function for that system.

$$Gvel_{cl}(s) = \frac{C_{vel}(s) \cdot pitch_{vel}(s)}{1 + C_{vel}(s) \cdot pitch_{vel}(s)} \tag{3.30}$$

The next step is to add the angular position controller and close the feedback loop.

$$Gpos_{cl} = \frac{C_{pos}(s) \cdot Gvel_{cl}(s) \cdot \frac{1}{s}}{1 + C_{pos}(s) \cdot Gvel_{cl}(s) \cdot \frac{1}{s}} \tag{3.31}$$

The end product from Equation (3.31) is a transfer function where the input is a desired angle in radians and the output is the drone's angle in radians.

For the next step, two assumptions will be made to simplify the system. The first one is that, since the drone will never bank more than 0.175 radians, it is fair to assume that the force required for hover will be almost constant. The second assumption is that $\sin angle \approx angle$ as long as $angle < 0.175rad$ holds. The reason for this is that it makes

the calculations simpler, with the only downside being a minuscule decrease in the accuracy of the model.
With the previous assumptions, the transfer function for the system can be approximated to the following:

$$Gvel(s) = \frac{Gpos \cdot C_{pos}(s)}{1 + Gpos \cdot C_{pos}(s)} \cdot F_{hover} \cdot \frac{1}{s} \tag{3.32}$$

Which gives the following transfer function.

$$Gvel(s) = \frac{6e04 \cdot s^2 + 2.981e06 \cdot s + 3.703e07}{s^6 + 64.79 \cdot s^5 + 1948 \cdot s^4 + 3.954e04 \cdot s^3 + 4.371e05 \cdot s^2 + 1.064e06 \cdot s} \tag{3.33}$$

3.3.2 Height Transfer Function

The height controller transfer function was already given previously by Jens Christian Andersen as the following:

$$\frac{0.4718}{s^3 + 15.16 \cdot s^2 + 0.8523 \cdot s} \tag{3.34}$$

What this transfer function describes can be seen in Section 3.3.

3.3.3 Yaw Transfer Function

The transfer function shown in Section 3.3, together with its already known P-controller is given as the following:

$$Gyaw\_vel(s) = \frac{9.398}{0.03927 \cdot s^3 + 3.735 \cdot s^2 + 47.44 \cdot s} \cdot 10.0 \tag{3.35}$$

3.4 Discretization

The next step before designing the controllers is to discretize, both the transfer function for a PI-Lead controller and also the transfer functions for the drone. The discretization takes the transfer functions from the s-domain to the z-domain. There are a few methods to do an approximation of a s-domain transfer function to a z-domain transfer function. The two approaches used in this case will be *Zero Order Hold* (ZOH) for the transfer functions for the drone, and then a *Bilinear Transform* (Also known as Tustin transform) for the PI-Lead controllers. Figure 3.4 shows the difference between the two approaches. As shown, the billinear is overall closer to the actual value, compared to ZOH.

3.4.1 Drone Transfer Functions Discretization

The reason why ZOH is used to discretize the drone's transfer functions is that it mimics how the sensors that are responsible for estimating the drone's pose act. The OptiTrack system which will be used to estimate the pose always gives the latest measurement as its output which is exactly the same behavior that ZOH displays.

The transfer functions describing the dynamics will all be discretized using the built-in Matlab function *cd2(<tf>, <sampletime>, <type>)*.

3.5 Controller Design

As stated earlier in Section 3.2, the drone already contained controllers that allowed angular positional control of pitch and roll, together with yaw velocity control. This section will therefore look at designing, implementing, and tuning, of a X-Y position controller, a height controller, and a yaw position controller.

All controllers follow the design shown in Figure 3.5. The controller is a standard PI-Lead setup, with the little change that the lead term has been moved to be a part of the feedback

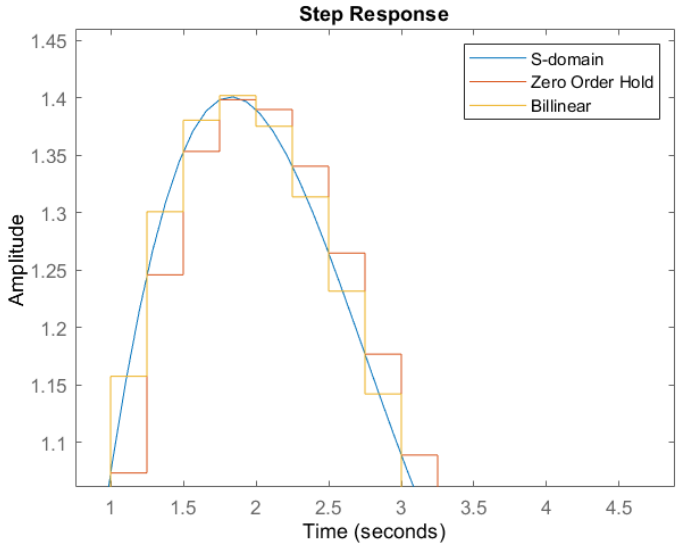


Figure 3.4: Step response of a generic transfer function and its z-domain counterparts, with a sample rate of 0.25s

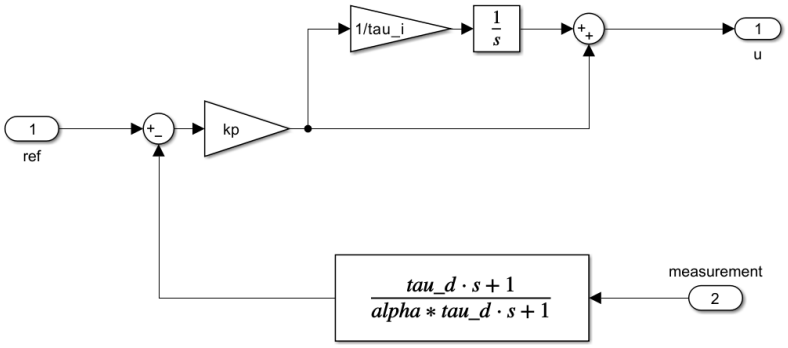


Figure 3.5: Base controller design

path. By doing this, it is possible to reduce the impact that an error in measurement will have. The downside is that it gives a slower response [12].

All controllers will be designed and tuned based on the following criteria:

- **High Phase and Gain Margin:**
  The reason for this is that even if the Matlab model does not model the drone perfectly (Which is most definitely the case), the controllers should be robust enough, to a point where this imperfection won't lead to an unstable controller. There isn't really a right or wrong number to aim for here, but in general, a phase margin above 35 is preferred

- **Realistic Constant Gain:**
  The P term of the controller is its gain. What is meant by a realistic gain is that given some realistic input, the controller's output signal should not saturate the actuator (I.E ensure that none of the motors go below or above their min and max thrust). A smaller thing to also consider is that with a low gain the drone will act less aggressively, which would be preferred as well mainly due to safety.

- **Avoid I Term:**
  Only if possible and the steady state error is minuscule. The I term adds a layer of complexity to the drone, due to the fact that it is possible to run into a windup issue. There is ways around this, but it is generally preferred to avoid just for simplicity's sake.
- **Avoid fast rise time:**
  This seems a bit counterintuitive but the idea is that a slow rise time should make the drone less aggressive which for testing and safety is a desirable trait. Of course, the drone should not be incredibly slow. A Lead term has a dampening effect, which means it can give a slower rise time, which in turn, of course, makes the drone less erratic, especially because it is placed in the feedback path of the controller. Furthermore, a slower rise time can also contribute to less overshoot. When you have something like a drone, it is preferred to not have a lot of overshoot since this could lead to the drone crashing into things.

3.5.1 Controller Discretization

Since the discretized controller has to be implemented in C++, it will be discretized by hand. The main reason is that this allows for a function that has a variable as the sample time, which will be important when the drone has to switch between its external tracking system with one sample time, to its internal tracking system with another sample time.

The bilinear transform approximates the s-domain variable to the following z-domain expression:

$$s \approx \frac{2}{T} \cdot \frac{z-1}{z+1} \tag{3.36}$$

All that is needed to be done at this stage is take the I and Lead term of the PI-Lead controller.

$$C_i(s) = \frac{\tau_i + 1}{\tau_i \cdot s} \tag{3.37}$$

$$C_d(s) = \frac{\tau_d \cdot s + 1}{\alpha \cdot \tau_d \cdot s + 1} \tag{3.38}$$

And then simply just replace the s-domain variable with the z-domain equivalent approximation.

$$C_i(z) = \frac{T \cdot z + T + \tau_i}{2 \cdot \tau_i \cdot z - 2 \cdot \tau_i} \tag{3.39}$$

$$C_d(z) = \frac{\tau_d \cdot 2 \cdot z - 2 \cdot \tau_d + T \cdot z + T}{2 \cdot \alpha \cdot z \cdot \tau_d - 2 \cdot \alpha \cdot \tau_d + T \cdot z + T} \tag{3.40}$$

This will look like the following in Matlab Simulink:

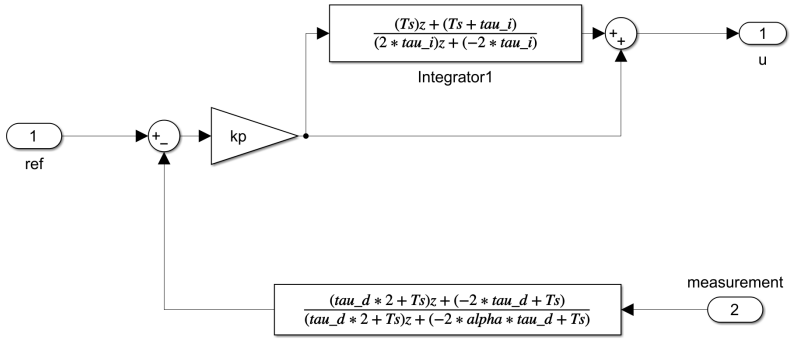


Figure 3.6: Matlab Simulink implementation of a discretized PI-Lead controller

3.5.2 Height Control

To make things simpler, it will be assumed that it is possible to give the drone a default thrust value allowing for hover and that this value will be added to the input after the controller. This means that if the height controller's output is 0, then the drone should just stand still in a hover. Of course, this will lead to the drone having some steady-state error that cannot be ignored, because of disturbances and inconsistencies in the model and the real system. This can be fixed by including an I term.

For the height controller, initially, two approaches were investigated. The first one was implementing a PI-Lead height position controller, just as what was initially planned based on the initial model. The second approach was to implement a cascading design consisting of a positional controller feeding into a velocity controller for height. The second one is more complicated but has the added benefit of in theory having a natural integrator, which should mean that it should not require an I term to remove any steady-state error. And as a bonus, it is possible to limit the maximum velocity along the z-axis.

By analyzing the bode plot of the transfer function previously mentioned in Section 3.3.2, it shows that the function has a starting phase shift of 180 degrees, together with its gain crossover frequency where the phase is also 180 degrees. This means that the system is unstable and requires a lead term to pull the system's phase up [12]. This is the reason why both methods will have a lead term.

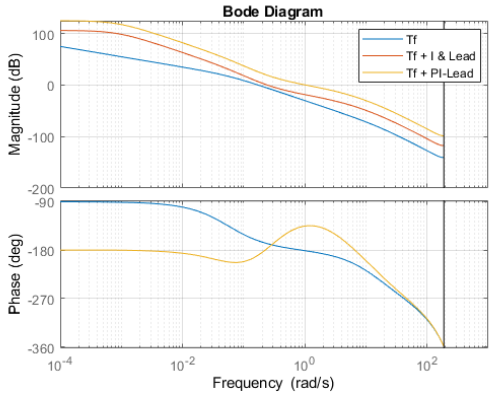


Figure 3.7: The discrete transfer function for height in a bode plot

NOTE: In Figure 3.7 both TF's should have a phase shift of 180. The inconsistency at the start is suspected to come from the linearization of the Simscape components of the Simulink model.

PI-Lead Controller is designed based on the TF shown in Figure 3.7, the lead term needs to come before the phase shift that pulls the TF even further down. After a bit of trial and error, it seems that placing the lead term at around 1 rad/s fulfills the criteria listed in Section 3.5 fairly well. It gives a gain margin of 20 dB and a phase margin of 45 degrees, which is decent.

When a frequency has been chosen, the next step is to calculate the time constants for the lead and I term. Before this can be done, it is necessary to select a lead factor and an I factor. The lead factor α were chosen to be a fairly aggressive value of 0.07 since this allows for a maximum theoretical phase shift up to 60 degrees$\approx$. The next step is the I factor Ni. This factor decides how far the I term "lags" behind the operation point. A good

starting value is said to be 4 [12].

$$\tau_d = \frac{1}{\sqrt{\alpha} \cdot \omega_c} = 3.77964473 \tag{3.41}$$

$$\tau_i = Ni \cdot \frac{1}{\omega_c} = 4 \tag{3.42}$$

The next step is to find the kp value. This can be found by analyzing the bode plot of the Tf + the I and Lead term. Since 1 rad/s was chosen as the working point, all needed to be done now is to ensure that this is also the gain crossover frequency. At that point, the function is -18≈ dB away from 0 dB. With this information, the kp can be calculated as the following

$$kp = 10^{-dB/20} = 10^{18/20} = 7.9433 \tag{3.43}$$

This gives the transfer function plotted in Figure 3.7, with a step response as shown in Figure 3.8. This has an overshoot of 1.39≈ and a settling time of around 7 seconds, which is not the best performance.

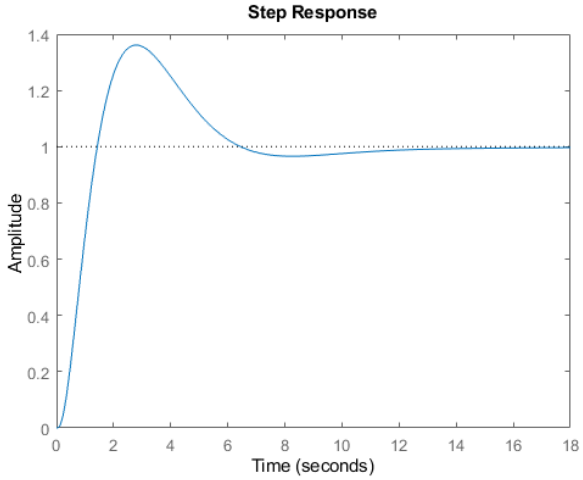


Figure 3.8: Step of the height TF + First Controller

Cascading Controller consists of a P controller and a P-Lead controller. The first controller that will be tuned is the P-Lead controller, responsible for regulating the velocity. This setup allows for the introduction of a limiter between the two controllers, which will allow for a max velocity of the drone. To tune it, the same TF as the previous one will be used, but with a small difference. The previous TF output was the drone's position. For tuning the velocity controller, the output should be velocity. To fix this, all needed to be done is to apply differentiation, which can be done by multiplying the TF with s. This gives the TF shown in Figure 3.9.

Due to the differentiator applied before, the phase shift is no longer 180 at the beginning. However, when it comes to tuning the P position controller which comes after, the TF function will have to be converted back to output position, which will be done by adding an integrator, which will pull the phase down again. This is the reason why the velocity controller will be a P-Lead controller.

Again like before using trial and error and analyzing the response, it seems that a good working point for the velocity controller is 25 rad/s. And like before, a fairly aggressive

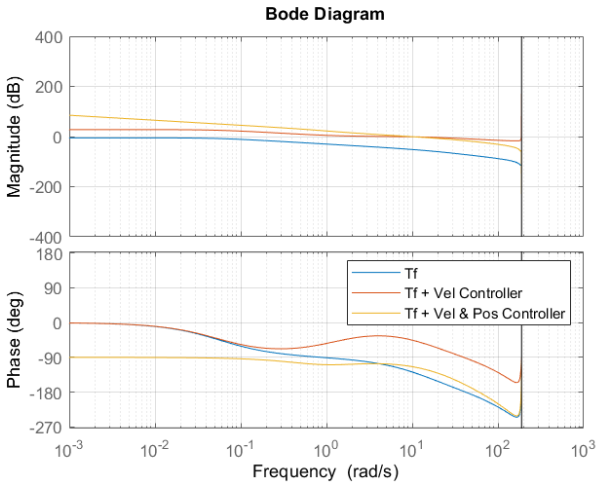


Figure 3.9: The discrete transfer function for height in a bode plot, used for the Second Cascading Controller design

lead factor of $\alpha = 0.07$ is chosen. This gives us a τ_d time constant of:

$$\tau_d = \frac{1}{\sqrt{\alpha \cdot \omega_c}} = 0.755928946 \tag{3.44}$$

After applying the lead term to the TF, the TF is around -33 dB away from the gain crossover where the frequency is 25 rad/s.

$$kp = 10^{-dB/20} = 10^{33/20} = 44.6683592 \tag{3.45}$$

The designed velocity controller has a phase margin of 143, which is excellent and should ensure that it is fairly robust to disturbances and imperfections in the model [12].

Now that the velocity controller has been tuned, the next step is to apply it to the TF, then integrate it so that we have a TF that can be used to design the position controller. Integration in the s-domain is $\frac{1}{s}$. Since this is a P controller, all that has to be done is choose a frequency to become the new crossover frequency. For this controller, the aim is a high phase and gain margin. By placing the new crossover frequency at 10 rad/s, the controller gets a gain margin of 22.1 dB and a phase margin of 66.8 which is excellent, while also providing a decent gain.

The step response of the controller is shown in Figure 3.10. It has an overshoot of around 0.1 and a settling time of around 0.3 which are great results. The settling time will probably be higher in the simulation, due to the velocity limit talked about earlier.

3.5.3 X & Y Position Control

This will be designed as a cascading controller consisting of a P positional controller and a P-Lead velocity controller. The reason for this is that it gives a way of limiting the drone's maximum speed, by putting a saturation limit on the output of the positional P controller, which makes the drone less aggressive in its movements.

Velocity Control

Like previously, the first step is to find a lead factor. Here the initial phase shift is 90, which means that the system already has a decent phase margin, which allows for a less aggressive lead factor. For this controller, the lead factor will be $\alpha = 0.3$, since this will

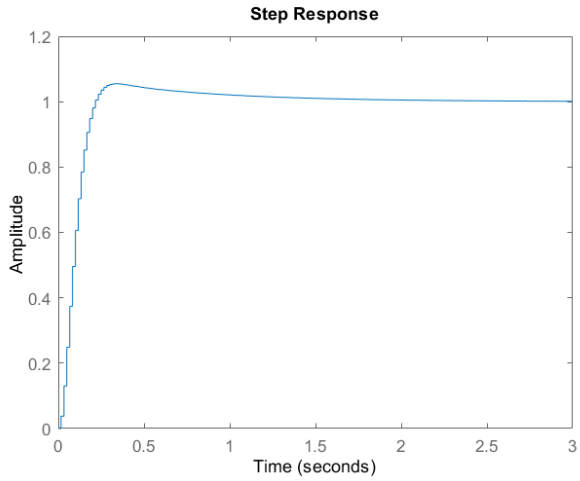


Figure 3.10: Step response of the cascading height controller

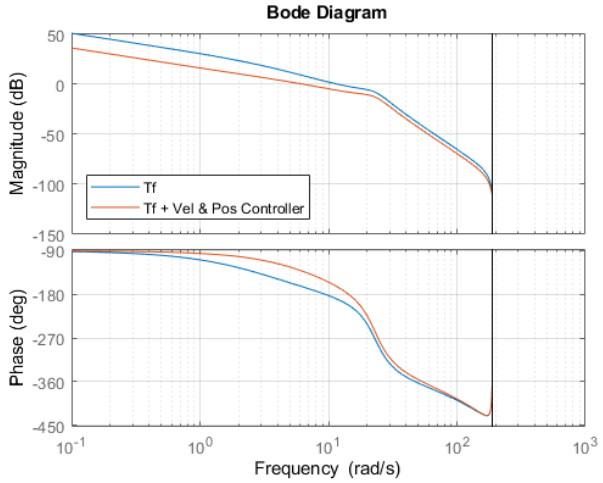


Figure 3.11: Tf of the discrete transfer function for position

have less of an impact on the magnitude. Next step is to choose the working point, which again is a bit of trial and error to see what gives the best lives up to the criteria previously set. For the velocity controller part, 5 rad/s was chosen.

$$\tau_d = \frac{1}{\sqrt{0.3} \cdot 5} = 0.365148372 \tag{3.46}$$

The next step is again to make 5 rad/s the new gain crossover.

$$kp = 10^{-14/20} = 0.1995 \tag{3.47}$$

This gives a phase margin of 47. and a gain margin of 8 dB.

Position Control

For the positional P controller, a Kp value of 1 was chosen, due to the fact that the phase margin and gain margin were already decent.

The step response of the controller is shown in Figure 3.12. Again an acceptable result. The overshoot seems to be no more than 0.1 and the settling time seems to be 5 seconds.

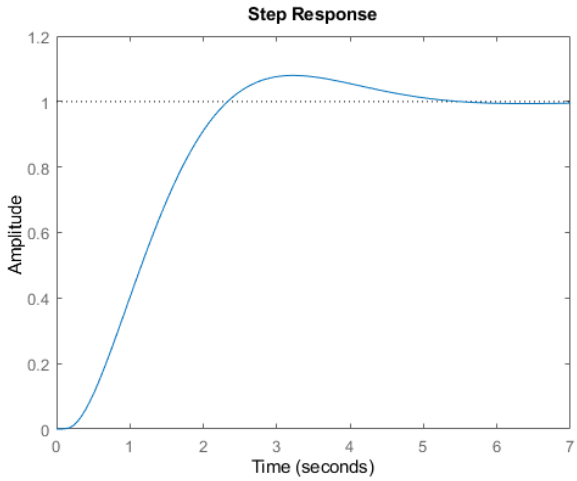


Figure 3.12: Step response of the cascading position controller

3.5.4 Yaw Position Control

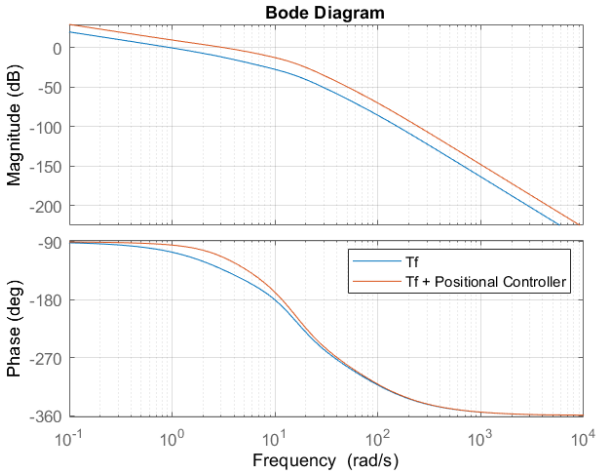


Figure 3.13: Tf of the discrete transfer function for yaw

Since Yaw already contains a velocity controller, it should be possible to make a positional controller with a P controller, but due to some weird results which will be discussed a bit more in Section 5.6.3, it ended up being a P-Lead controller. The main suspicion was that the IMU data were a bit noisy and that the Lead worked as a filter, even though the IMU already had a filter applied to it.

Again due to a fairly good phase margin, a less aggressive lead factor of $\alpha = 0.5$ was chosen. Which meant that the gain from the lead term was minuscule. If a more aggressive factor, the gain crossover frequency would be moved further to the right in the bode plot, resulting in it being closer to where the phase is 180. And as stated earlier, if the phase shift is 180 degrees and the gain is 0, the system is unstable [12]. The working point was chosen to be 3.3 since this would lead to a phase margin of 65 and a gain margin of 27, which is excellent.

The step response of the controller shown in Figure 3.14, shows promising results. It has an overshoot of less than 0.1 radians and a settling time of around 5 seconds.

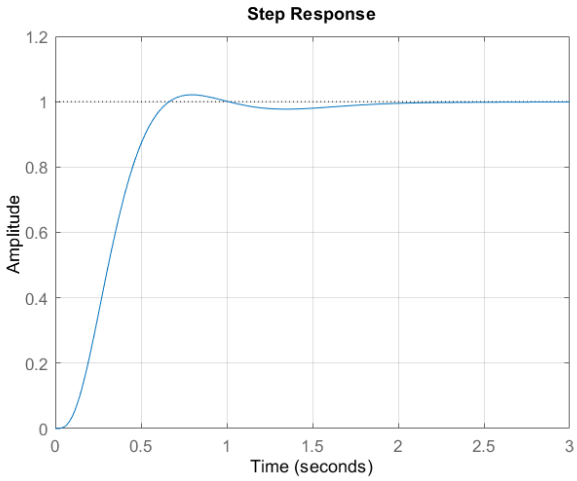


Figure 3.14: Step response of the yaw controller

3.6 Simulation Results

Now that all the controllers have been designed and tuned, it is time to evaluate their performance on the Simulink model. Figure 3.15, shows the full Simulink model used to simulate the drone. The controllers that have been designed in this chapter are the orange blocks.

Back in Chapter 1, a simple example of a mission was presented, together with some requirements. To summarize the mission called for an accuracy of 8cm, which means that the controllers should never overshoot with more than 8 cm, preferably a lot less. It also stated that the speed should be 1 m/s due to safety concerns

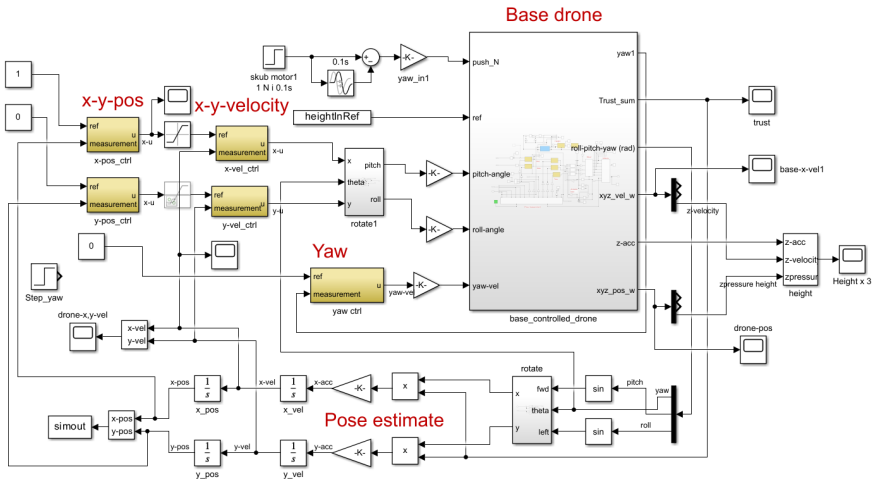


Figure 3.15: Full model of the drone. The inside of the block Drone is shown in Figure 3.15

3.6.1 Height Controller

To test the two height controllers, they will both initially start at 0 meters, stay for 3 seconds, and then be given a command to go to 4 meters.

PI-Lead

- **Rise time:** 6.16 seconds

- **Settle time:** 13.42 seconds
- **Max velocity:** 1.05 m/s
- **Max overshoot:** 0.13 m

Cascading Design
This controller is limited to 0.5 m/s so rise time is a little misleading.

- **Rise time:** 8.148 seconds
- **Settle time:** around 16.46 seconds
- **Max velocity:** 0.55 m/s
- **Max overshoot:** 0.485 m

Both seem to have negligible steady-state error lower than 3 cm. The PI-Lead seems to have the best performance, although its speed is 0.05 above a fix for this could be to just approach in incremental steps. The ability to limit the velocity for the cascading controller is a nice feature, it only exceeded its set speed by 0.05 m/s. Both controllers will be tested on the actual system since they both performed well in regards to the criteria set in Chapter 1.

3.6.2 X-Y Controller
The test will be done where the drone goes 1 m up and allow it to settle, then it will be issued a command to move 4 m along the x-axis. Again because this is a cascading controller, it will be limited to a max speed of 1 m/s.

- **Rise time:** 5.23 seconds
- **Settle time:** around 2.9 seconds
- **Max velocity:** 1 m/s
- **Max overshoot:** 3.4 cm
- **Stead-state error:** around 3 cm

Overall great performance. It never exceeded its set speed, it has a low settle time, and decent rise time (Mainly limited by the max velocity).

3.6.3 Yaw Controller
There aren't any specific requirements set when it comes to yaw. The test case is that the drone flies up and stabilizes, then the drone will be issued a command to rotate 180 degrees.

- **Rise time:** 4.59 seconds
- **Settle time:** almost 0 seconds
- **Max velocity:** 0.890 rad/s
- **Max overshoot:** almost 0
- **Stead-state error:** almost 0

Not much to comment on in regard to performance, it is both recently fast and close to no overshoot or steady-state error.

3.7 Summary

In total 8 controllers were designed and tuned. One controller for x and one for y position, one for x velocity and one for y velocity, and one for yaw. Height ended up having two different controllers made, which both had their advantages and drawbacks. Overall all the controllers perform close to or within the specifications specified in Chapter 1, in the mission section.

4

Real-Time System Implementation

4.1 Setup

Figure 4.1 shows the initial communication setup, done by Jens Christian Andersen. The additions made to this diagram is that the Raspberry is connected to the OptiTrack system and a webcam. As mentioned previously, the Teensy works as an attitude controller and

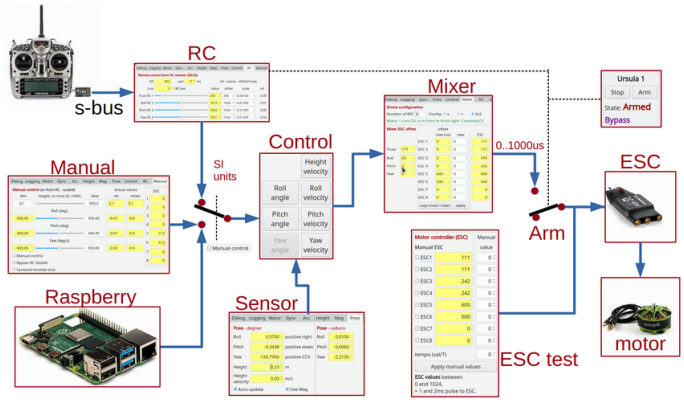


Figure 4.1: Diagram of the communication of the drone [9]

is responsible for all the controllers shown in Section 3.2. The Teensy has access to the IMU data. As it is set up right now the Raspberry does not have access to the IMU data, and will therefore use the OptiTrack system to also measure pitch, roll, and yaw. The Raspberry will run all the controllers designed in Chapter 3.

4.2 Initial Problems with the Drone

Before implementing any autonomous control software, it was necessary to ensure that the drone could actually fly manually. After some initial testing done by Søren Hansen, two issues were discovered which meant that the drone was not safe to fly.

4.2.1 Unscheduled Landing Issue

The first issue was that the drone had a tendency to at random times decide to land, by simply reducing the throttle to 0 and then turning it up slowly. Which would cause the drone to hit the ground hard.

After looking at the codebase it was discovered that the drone had an auto-landing feature, which would kick if the following statement were true:

```
1    if(hgt.height < 0.5 and hgt.heightVelocity < -0.1)
```

The drone consists of 3 state machines, that decide the overall behavior of it. One for if the drone is *armed* or *disarmed*, this is just sets if the motors are allowed to spin up or not. Then one for determining if the drone is in *autonomous mode* or *manual mode*. Finally, it has one which consists of the following states, *OnGround*, *Landing*, *InFlight*, and *Starting*. If the statement shown above is true it will jump from any state except *OnGround* and *Starting*, to *Landing*. No matter if its in *autonomous mode* or *manual mode*.

On paper, this looks like a neat solution. If the drone is less than 0.5 m above ground and is moving less than 0.1 m/s, then it should land. The issue though is that this runs on the Teensy and therefore bases all of its height and velocity measurements on the IMU, which relies on accelerometer and barometer data to estimate its height. In general, when it comes to height estimation, it seems to be less and less accurate over time. From some simple logging, it seemed to be off up to 1.2 meters after 30 seconds. Another issue with this is that the *Landing* state takes control and supersedes any inputs from the RC control, which should never happen. Furthermore, the auto-landing feature seems to cut power to the engines temporarily, and then increase the throttle again, which of course causes the drone to crash.

The current fix for this is simply to disable the auto-landing feature, and also ensure that if the drone is in *manual mode*, it will always presume that it's *InFlight*. This ensures that the pilot always can take control in case anything unexpected happens.

In the future, it would be preferred to fix the auto-landing feature, and then equip the drone with a laser-based height estimator.

4.2.2 Refusing Pitch & Roll commands

It was discovered that for some reason the drone would refuse to pitch forward, but would pitch backward. The drone has a max and min limit set on its angle position controllers for pitch and roll, which by default is set to 5 degrees, which should allow it to pitch both ways. The idea with this max and min is to ensure that the drone does not do anything too drastic

The way that the drone estimates its orientation and heading, is by using an IMU. The IMU has 10 DOF [13] and measures attitude and heading by fusing all the sensor data with a Madgwick filter. It turned out that this filter was a few degrees off, and that the drone was manually trimmed to have an actual pitch of around 0. This led to the drone actually thinking that it was already tilted a bit and therefore not allowing its pitch angle to increase.

The fix for this was to increase the max and min limit from 5 degrees to 20 degrees, and then put a relative limit on the reference commands sent from the Raspberry to the Teensy. This allowed to drone to move freely, while also limiting the maximum angle it was allowed to bank when it was flying autonomously.

4.2.3 Internal Height Controller

As stated in Section 3.2, the drone already contained a height controller. This controller runs on the Teensy, and as discussed in Section 4.1 and therefor does not have access to the OptiTrack data, which means that the Teensy height controller would have to rely on

IMU height estimation. Furthermore, as shown in Figure 3.15, if this controller would have been implemented, it would have meant that instead of the RC remote controlling throttle, it would actually control the set height, which is not ideal when the height estimation is off.

The height controller was therefore converted to a P controller with a feed foward as shown here, since this allowed for manual throttle control. The initial settings were as shown in Figure 4.2.

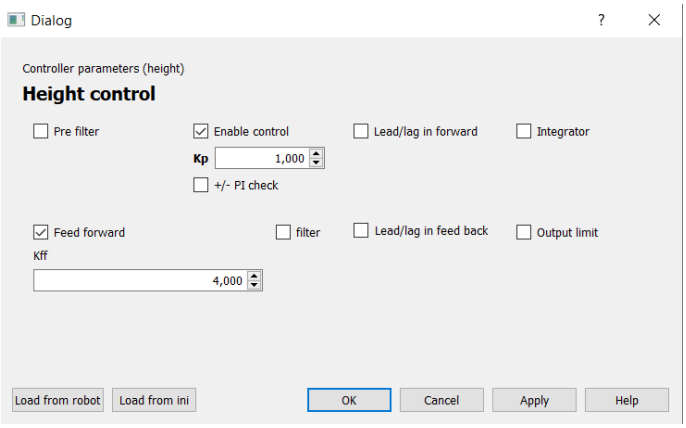


Figure 4.2: Height tuning software made by Jens Christian Andersen [9]

Disabling the controller and effectively turning it into a P controller with a gain of 0 and a feed-forward of 1, also seemed to not be an option, since issuing the drone a 100% thrust command barely spun up the propellers. It was therefore initially decided to ignore the controller, but it was later discovered that the feedback was causing some interesting behavior.

4.2.4 Thrust Units Scaling

It was discovered that the drone and the Matlab model were using different units for thrust. The drone was using 1-100 to represent thrust, whereas the Matlab model was using 1-1024. This was causing the scaling of the controllers to be wrong. What it meant was, that the controllers actually had a gain of rougly 10, which in the bode plot, would move the gain crossover frequency, and potentially make the controllers unstable. The solution to this was to change the scaling on the drone to fit the Matlab model. An added benefit of this was that it turned out to also somehow be linked to the issue with the Teensy height controller mentioned in Section 4.2.3. After the drone was changed to operate between 1-1024 thrust, it was possible to disable the Teensy height controller, and still fly the drone via RC remote.

4.3 Pose Estimation

OptiTrack is a very accurate indoor tracking system. The configuration used for the drone had a mean error of 1.1mm. It outputs the drone's position and a quaternion for the drone's orientation. Integration with the system is done by using the NatNet SDK [14]. The OptiTrack system has an update rate of 240hz. A general rule of thumb when making discrete controllers is always to connect their refresh time up on the measurement system, or run slower, to not read the same measurement twice. However, 240hz is quite fast and it should be easily possible to run with a refresh rate on the controllers of 60hz or even lower.

Aruco tracking is another method that will be used for pose estimation. It works by having predefined 2d markers with a fixed size (See Figure 4.5 for an example of a marker), and

640x480	320x240
14 fps	30 fps

then detecting the corners of the marker. Then based on the corner's pixel distance from each other, it is possible to estimate which way the marker is pointing in relation to the camera. Furthermore, by taking advantage of the fact that the size of the marker, it is also possible to estimate its x,y, and z position in relation to the camera. It does require knowledge about the camera's intrinsic matrix, and its distortion coefficients. To use this as pose estimation for the drone, it is necessary to transform the Aruco marker from the camera frame to the drone's frame. The Aruco system has a lower refresh rate than the OptiTrack, mainly due to the limiting processing power of the Raspberry Pi. This can be increased at the cost of resolution as shown.

4.3.1 Camera Calibration

It is necessary to know the intrinsic camera matrix, such that the images from the camera can be undistorted. The intrinsic camera matrix can be calculated if the focal length of the camera is known. However, a much easier way is to use the chessboard method. A good test to see if the calculated intrinsic matrix is correct is to use it to undistort an image. Figure 4.3 shows quite clearly how the planks in the background go from curved to straight, indicating that the image was successfully undistorted. What the intrinsic matrix does is it

Figure 4.3: Undistorted vs distorted image

allows one to take a point in the image plane, and project a line into 3d space where said point should lie on. This together with depth information from the Aruco marker which is calculated based on its size in the image plane, it is possible to estimate an Aruco marker's position in 3d space.

As small note that will become important later is how the Field Of View (FOV), gets reduced when the picture gets undistorted. This detail will become important in the following chapter when analyzing how good the Aruco tracking ended up being.

4.3.2 Aruco Transform

As mentioned above, the Aruco tracking system gives the coordinates of the Aruco marker in the camera frame (Which is not the same as the drone frame), and the orientation of the marker. This information cannot be used as is, it needs to be transformed first. What is wanted is either the marker's position relative to the drone in the global frame, or the marker's position in the global frame. For this, the first option will be sufficient enough as proof of concept. If it were to fly on multiple markers, the other approach would make things a bit easier.

The first step is to align the x y and z axes so that they are easier to work with. When the drone is on the ground and its global yaw is 0, then the three camera axes should

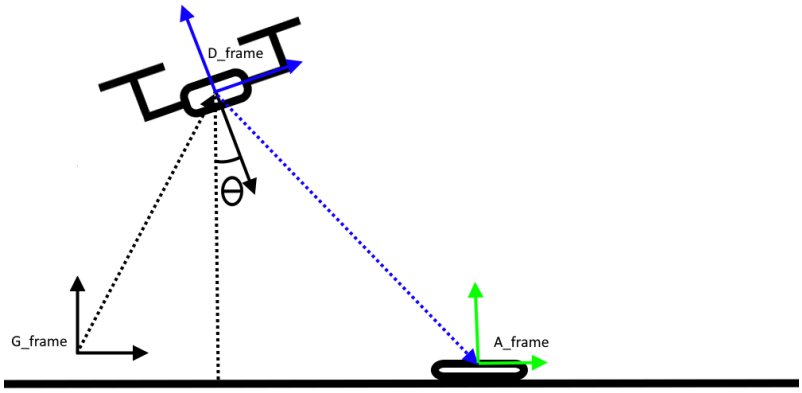


Figure 4.4: Overview over the different frames and how they are related to each other

align with the global frame axes (Which also means that they align with the drone frame). Figure 4.5 shows the camera frame and which way the axes are. To get the axes to align, the positive x direction needs to point up, the positive y direction needs to point right, and the positive z direction should point toward the camera.

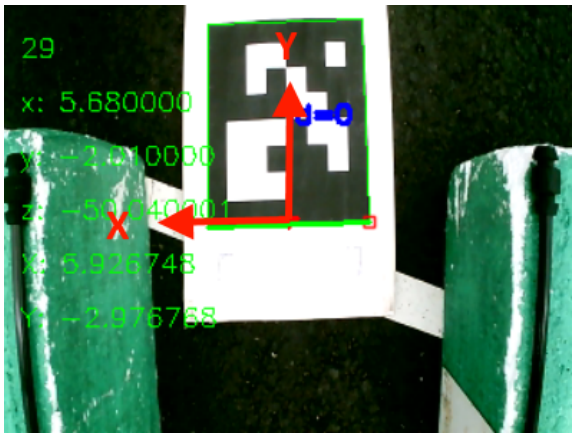


Figure 4.5: Default Aruco camera frame, where Z is away from the camera

$$\begin{bmatrix} X_{drone} \\ Y_{drone} \\ Z_{drone} \end{bmatrix} = \begin{bmatrix} Y_{cam} \\ X_{cam} \\ Z_{cam} \end{bmatrix} \cdot \begin{bmatrix} 1 \\ -1 \\ -1 \end{bmatrix} \tag{4.1}$$

Now that the axes align, the next step is to take the drone's orientation into account. As mentioned, it is wanted to get the marker's position relative to the drone, but keeping the global orientation. This can be achieved by doing a simple rotation. The problem is illustrated in Figure 4.4.

$$\begin{bmatrix} X_{global} \\ Y_{global} \\ Z_{global} \end{bmatrix} = Rot(\phi_{drone}, \theta_{drone}, \psi_{drone}) \cdot \begin{bmatrix} X_{drone} \\ Y_{drone} \\ Z_{drone} \end{bmatrix} \tag{4.2}$$

This gives the coordinates needed for the controller, such that the drone can navigate in relation to the Aruco marker.

4.4 Tracking Abort System

Due to some concerns regarding the stability of the Aruco tracking, a simple system was implemented to ensure that the drone would abort in case it lost track. The system works by checking if the drone has seen an Aruco code in the last few seconds. If the drone hasn't it will abort any mission and try and maintain its current position, refusing to do anything further before the pilot takes manual control.

4.5 Controller Implementation

Back in Section 3.5.1, the z-domain representation of the I term and the Lead term of a PI-Lead controller were defined as the following:

$$C_i(z) = \frac{T \cdot z + T + \tau_i}{2 \cdot \tau_i \cdot z - 2 \cdot \tau_i} \tag{4.3}$$

$$C_d(z) = \frac{\tau_d \cdot 2 \cdot z - 2 \cdot \tau_d + T \cdot z + T}{2 \cdot \alpha \cdot z \cdot \tau_d - 2 \cdot \alpha \cdot \tau_d + T \cdot z + T} \tag{4.4}$$

Where T is the sample time. This could of course be calculated with Matlab, but it is wanted to have the sample time such that it can be changed since, the Aruco tracking and OptiTrack runs a different sampletime. Furhtermore, it allows one to easily do adjustments to τ_i and τ_d

The goal is to create a C++ implementation of the controller shown in Figure 3.5. The first step is to get it on a form that's easier to work with. The z is a time unit, but in its current form is not that useful. It is therefore desired to turn it into time unit delays since the previous values will be known, but future values are not.

$$C_i(z) = \frac{T \cdot z + T + \tau_i}{2 \cdot \tau_i \cdot z - 2 \cdot \tau_i} \cdot \frac{z^{-1}}{z^{-1}} \tag{4.5}$$

$$C_d(z) = \frac{\tau_d \cdot 2 \cdot z - 2 \cdot \tau_d + T \cdot z + T}{2 \cdot \alpha \cdot z \cdot \tau_d - 2 \cdot \alpha \cdot \tau_d + T \cdot z + T} \cdot \frac{z^{-1}}{z^{-1}} \tag{4.6}$$

Which is then written as follows:

$$C_i(z) = \frac{T + T + \tau_i \cdot z^{-1}}{2 \cdot \tau_i - 2 \cdot \tau_i \cdot z^{-1}} \tag{4.7}$$

$$C_d(z) = \frac{\tau_d \cdot 2 - 2 \cdot \tau_d \cdot z^{-1} + T + T \cdot z^{-1}}{2 \cdot \alpha \cdot \tau_d - 2 \cdot \alpha \cdot \tau_d \cdot z^{-1} + T + T \cdot z^{-1}} \tag{4.8}$$

Where z^{-1} represents a one-unit time delay.

Now another representation of transfer functions is the following $\frac{Y(z)}{U(z)}$, where $Y(z)$ is output and $U(z)$ is input. $C_i(z)$ and $C_d(z)$ can therefore be replaced with $\frac{Y(z)}{U(z)}$. To better keep track $C_i(z)$'s $Y(z)$ and $U(z)$ will be refered to as $Y^I(z)$ and $U^I(z)$, and for $C_d(z)$ it will be $Y^L(z)$ and $U^L(z)$.

$$\frac{Y^I(z)}{U^I(z)} = \frac{T + T + \tau_i \cdot z^{-1}}{2 \cdot \tau_i - 2 \cdot \tau_i \cdot z^{-1}} = C_i(z) \tag{4.9}$$

$$\frac{Y^L(z)}{U^L(z)} = \frac{\tau_d \cdot 2 - 2 \cdot \tau_d \cdot z^{-1} + T + T \cdot z^{-1}}{2 \cdot \alpha \cdot \tau_d - 2 \cdot \alpha \cdot \tau_d \cdot z^{-1} + T + T \cdot z^{-1}} = C_d(z) \tag{4.10}$$

The next step is to isolate $Y(z)$ on one side withot $U(z)$ for both.

$$Y^I(z) \cdot 2 \cdot \tau_i = T \cdot U^I(z) + T \cdot z^{-1} \cdot U^I(z) + \tau_i \cdot z^{-1} \cdot UI(z) + 2 \cdot \tau_i \cdot z^{-1} \cdot Y^I(z) \tag{4.11}$$

$$Y^L(z) \cdot 2 \cdot \alpha \cdot \tau_d + T = \tau_d \cdot 2 \cdot U^L(z) - 2 \cdot \tau_d \cdot z^{-1} \cdot U^L(z) + T \cdot z^{-1} \cdot U^L(z) + 2 \cdot \alpha \cdot \tau_d \cdot z^{-1} \cdot Y^L(z) - T \cdot z^{-1} \cdot Y^L(s) \tag{4.12}$$

Then the constants on the left side get moved.

$$Y^I(z) = \frac{T \cdot U^I(z) + T \cdot z^{-1} \cdot U^I(z) + \tau_i \cdot z^{-1} \cdot U^I(z) + 2 \cdot \tau_i \cdot z^{-1} \cdot Y^I(z)}{2 \cdot \tau_i} \tag{4.13}$$

$$Y^L(z) = \frac{\tau_d \cdot 2 \cdot U^L(z) - 2 \cdot \tau_d \cdot z^{-1} \cdot U^L(z) + T \cdot z^{-1} \cdot U^L(z) + 2 \cdot \alpha \cdot \tau_d \cdot z^{-1} \cdot Y^L(z) - T \cdot z^{-1} \cdot Y^L(s)}{2 \cdot \alpha \cdot \tau_d + T} \tag{4.14}$$

The next step is to get rid of the z^-1 time unit delays. When multiplying a constant with a unit delay, nothing happens because its value is constant over time. When multiplying it with either an input or output that changes over time, it can be rewritten to be that input/output at that time unit. For example, if there's an input $U(z)$ and multiply it by z^{-1}, its value will be what $U(z)$ was, one unit time delay ago. Example $U_i(z) \cdot z^{-1} = U_{i-1}$.

$$Y_i^I(z) = \frac{T \cdot U_i^I(z) + T \cdot U_{i-1}^I(z) + \tau_i \cdot U_{i-1}^I(z) + 2 \cdot \tau_i \cdot Y_{i-1}^I(z)}{2 \cdot \tau_i} \tag{4.15}$$

$$Y_i^L(z) = \frac{\tau_d \cdot 2 \cdot U_i^L(z) - 2 \cdot \tau_d \cdot U_{i-1}^L(z) + T \cdot U_{i-1}^L(z) + 2 \cdot \alpha \cdot \tau_d \cdot Y_{i-1}^L(z) - T \cdot Y_{i-1}^L(s)}{2 \cdot \alpha \cdot \tau_d + T} \tag{4.16}$$

Now the controller can be implemented on a discrete system such as a microprocessor or a microcontroller. The implementation is fairly simple. The measurement of the sensor is the input for the lead term. For the I term, the input is the output of the lead term, minus the reference input, multiplied with the Kp constant. The final output is given as follows

$$output = (ref - Y_i^L(z)) \cdot Kp + Y_i^I(z) \tag{4.17}$$

Where $U_i^I(z)$ is set to be $ref - Y_i^L(z)) \cdot Kp$, and $U_i^L(z)$ is set to be the measurement.

The full implementation can be found on GitHub (See Appendix A.1)

4.6 Summary

This chapter set out to derive the necessary systems for the drone to achieve autonomous flight, and calibrate the systems such that they could be used. Together with fixing initial problems with the drone, that made it impossible for it to fly.

5

Real-Time System Results

This chapter will look at some of the data logged from the autonomous flights and try to compare them to the simulated results. Furthermore, it will also be discussed why some of the results might not be what was expected.

5.1 Introduction

The controllers will first be judged on the OptiTrack system since this is the most robust of the two tracking options.

Note that some of the controller parameters have been modified slightly for increased performance. As stated earlier, the model will never be perfect, and it was therefore expected, that the final implemented controllers would need a bit of fine-tuning.

5.2 Height Control

The test case for the two controllers and the parameters used in the test are shown below. This is to mimic the test done in Section 3.6.

PI-Lead Design

- **Height ref:** 3 meters.
- **Max velocity:** Does not allow.
- **Position Controller Settings:** $Kp = 8.907$, $\tau_i = 3.849$, $\tau_d = 3.637$, $\alpha = 0.07$

Cascading Design

- **Height ref:** 3 meters.
- **Max velocity:** 1 m/s
- **Position Controller Settings:** $Kp = 18.4106$
- **Velocity Controller Settings:** $Kp = 44.6684$, $\tau_d = 0.7559$, $\alpha = 0.07$

5.2.1 PI-Lead Design

The PI-Lead controller shown in Figure 5.1 is completely erratic and also unstable. The magnitude of its oscillation is clearly growing so the system is definitely unstable. Furthermore, it seems to go up and down without any clear indication as to why. It turned out

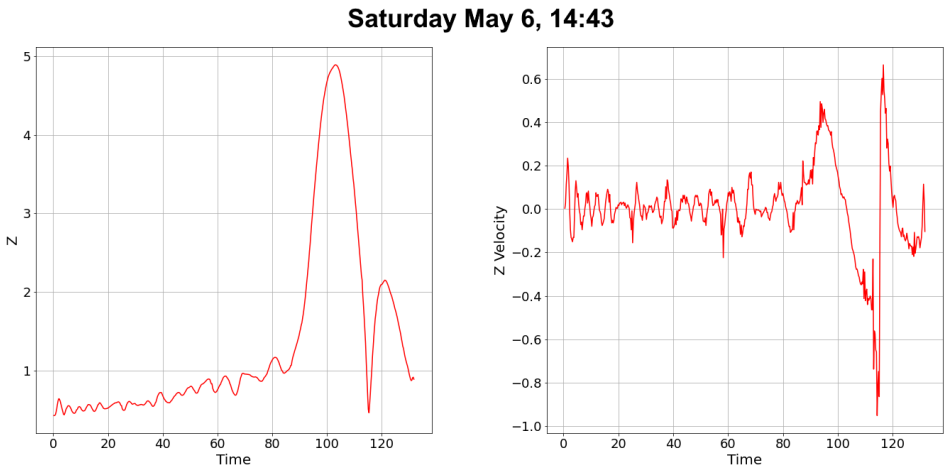


Figure 5.1: PI-Lead controller design without velocity limit. Height over time and height velocity over time, measured with OptiTrack

that there was actually a good reason for this erratic behavior, which is further explained in Section 5.6.1.

Results

- **Rise time:** Around 35 seconds but misleading.
- **Settle time:** Unstable.
- **Max velocity:** 0.46 m/s.
- **Max overshoot:** 1.3 meters.

Overall the results are not great. It did however stay below 1 m/s, but that does not mean much, since increasing its ref point, might very likely increase its velocity.

5.2.2 Cascading Design

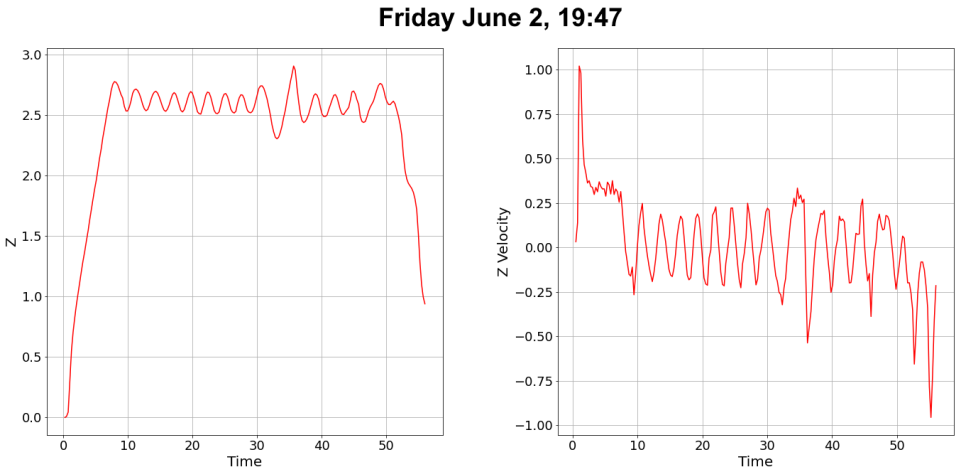


Figure 5.2: Cascading controller design. Height over time and height velocity over time, measured with OptiTrack

From Figure 5.2, it is quite clear that the controller does not reach the specified desired height of 4 meters. It is also quite noticeable that the controller is not unstable per se, but seems to oscillate up and down with a frequency of 0.36 hertz. The peak-to-peak amplitude of the oscillation is around 0.2.

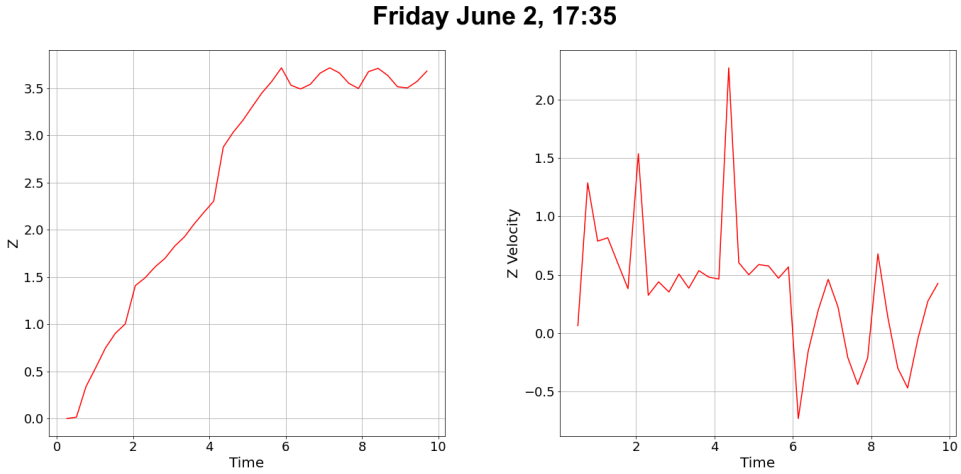


Figure 5.3: Cascading controller design without velocity limit. Height over time and height velocity over time, measured with OptiTrack

Figure 5.3 shows the same controller but without a limiter. It is quite clear that the controller here goes above the speed limit, which indicates that the velocity controller seems to act as it should. This could indicate that the issue lies with the positional controller part of the cascading controller. Again the plot shows that the controller reaches fairly close to the same height as before.

Results:

- **Rise time:** around 8 seconds
- **Settle time:** Less than 3 seconds but settles at the wrong value
- **Max velocity:** 1 m/s
- **Max overshoot:** Miniscule but oscillates

These results are fairly close to the ones gotten from the simulation (See Section 3.6.1) which is a sign that the Simulink model represents the drone fairly decently. It does however settle at the wrong value.

5.2.3 Conclusion

When it comes to the PI-Lead design, it had some erratic behavior. The graph shown above was not the best nor the worst test, but most of them had the same outcome. It was later discovered that there was a good reason for its erratic behavior, which is further described in Section 5.6.1. The other controller also had similar results before said issue was fixed. The plan was to test the controller again, but due to an unscheduled landing, leading to the drone not being able to fly anymore, only the cascading design were tested after the drone were fixed. It is believed that this controller should have worked, since it was proven that the Simulink model represented the real drone and controllers fairly well, at least with the other controller.

Overall it is quite clear that the cascading controller is the best of the two designs. This is of course mainly due to the fact that this controller was tested on the drone without the issue described in Section 5.6.1. With a bit of fine-tuning, it is believed that this design would work very well. Furthermore, it seems to have a similar performance to the simulated controller and drone.

5.3 X & Y Positional Control

For this system, the test case will be done a bit differently. Here the test will be performed where the drone loses its Aruco tracking and goes into its mission abort mode. This is to highlight the abort system but also highlight the controller's capability. The controller will still be evaluated and compared to the results from the simulation shown in Section 3.6.2.

- **Height ref:** 4 meters.
- **Max velocity:** 1 m/s
- **Position Controller Settings:** $Kp = 1$
- **Velocity Controller Settings:** $Kp = 0.2917$, $\tau_d = 0.3468$, $\alpha = 0.3$

Note that these controller values are the same as in Section 3.5.3. The Figure 5.4, shows

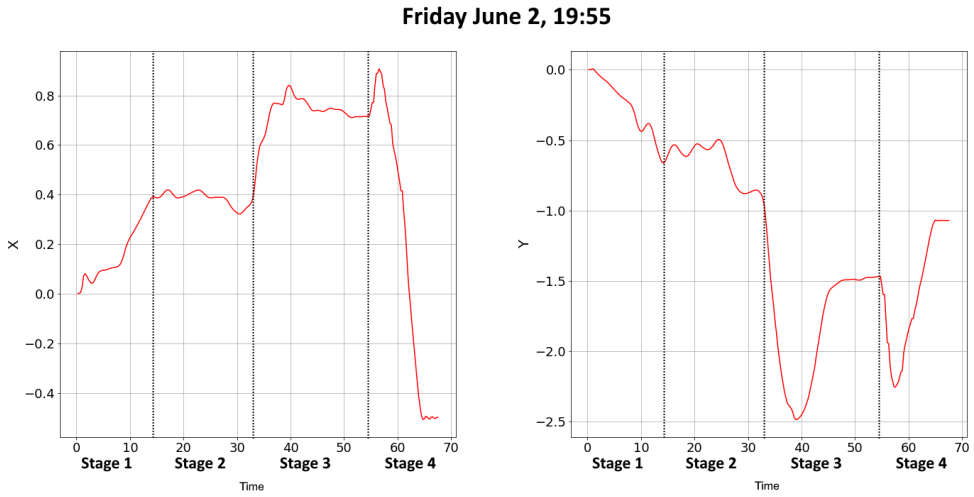


Figure 5.4: X and Y over time

what happens when the *Tracking Abort System* kicks in. What is shown can be split into 4 stages. Stage 1 is where the drone has found an Aruco code and navigates toward it. Stage 2 is where the drone is at the Aruco code. Stage 3 is where the drone loses tracking and starts drifting until the Tracking Abort System kicks in (See Section 4.4) and the drone starts to hold its position. Stage 4 is where the pilot takes manual control of the drone and lands it. The Figure 5.5 shows the drone's x and y velocity. As stated earlier, its max movement speed is set to 1 m/s but it quite clearly goes above this which is not ideal. However, it seems that when it goes above the set speed, is also when it loses tracking. In the other tests, it never exceeds the set speed, so the assumption is that it is due to loss of tracking.

Results:
The idea was to use the data logged from *Friday June 2, 17:35* (See Appendix A.2), where the desired position is $x : 0$ and $y : 0$. However, these results are quite misleading, since

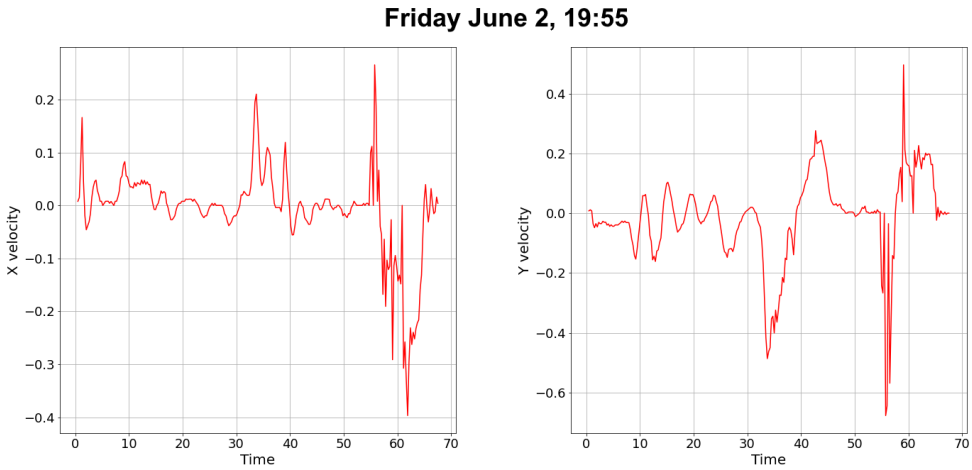


Figure 5.5: X and Y velocity over time

the tuning parameters were not the same as the ones used in the ones calculated in Section 3.5.3.

The results for overshoot, steady-state error and settle time will therefore be based on stage 1 and stage 2 of the X-Y controller shown in Figure 5.4. This is not ideal but does still show some performance of the controller. The main issue is that the different reference values over time were not logged so some assumptions have to be made. Therefore the assumption is that the Aruco location the drone is going to is $(0.4, -0.6)$ in meters, which would give the drone a rise time of 11 seconds. For overshoot, the assumption will be that the reference for stage 2, is around the average. And for steady-state error, there isn't a good approach.

- **Rise time:** 11 seconds
- **Settle time:** around 2.9 seconds
- **Max velocity:** 0.68 m/s
- **Max overshoot:** Around 2-4 cm
- **Stead-state error:** Presumably less than 5 cm

5.3.1 Conclusion

In general, this test is not the greatest to measure the performance of the controller, compared to what was simulated. However, it does show that the controller is at least fairly accurate. Based on the data an, the overall assumption is that its performance is a bit worse than what was simulated, but overall acceptable. The most important factor is the velocity control and the max overshoot, which is below what was required for the imaginary mission.

5.4 Yaw Control

The yaw controller was down-prioritized and therefore was not tested before the drone sadly crashed. A few tests were however done, where the only objective was to keep the drone pointing at the same heading. The Figure 5.6, shows the yaw in degrees over time, from the same flight as shown in Section 5.3. Not that the highest it is ever off from its initial position is 0.18 degrees.

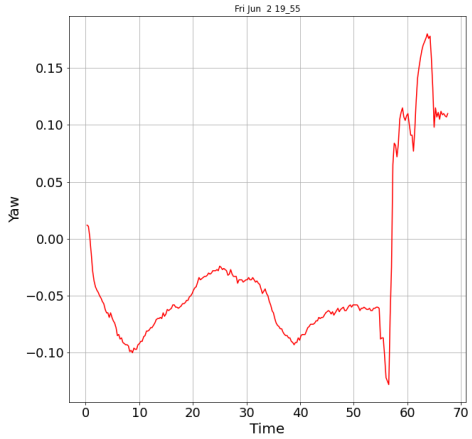


Figure 5.6: Yaw over time, given in degrees

5.5 Tracking using Aruco

The testing done here is quite simple. The drone flies up, finds a marker, and estimates its position in relation to the marker, where the OptiTrack will be used as the ground truth. The goal is to see some correlation between the OptiTrack-tracked position and the Aruco-tracked position of the drone. In short, the more correlated they are, the better. Furthermore, the difference between the two should never exceed the requirements set in the beginning. To make it a bit easier to correlate, the marker was placed just below the drone and the initial position of the drone was subtracted from the OptiTrack data, to make its start position $0, 0, 0$.

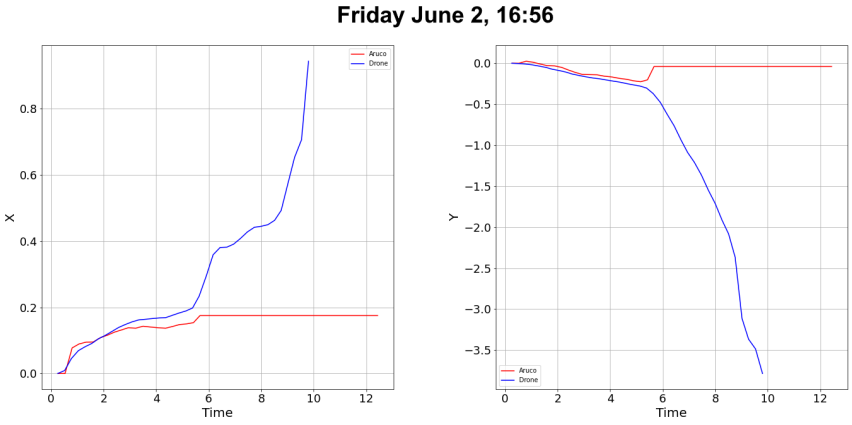


Figure 5.7: Aruco vs OptiTrack

Figure 5.7 shows the OptiTrack and Aruco position data plotted against each other. Both seem to be fairly correlated up to a point where the Aruco tracking flatlines. That is the point where the drone loses sight of the marker.

Figure 5.8 shows the difference between the two tracking methods. Here it is quite clear to see that they are highly correlated and that the Aruco tracking is well below the threshold of the mission requirements, at least when it is fairly close to the marker.

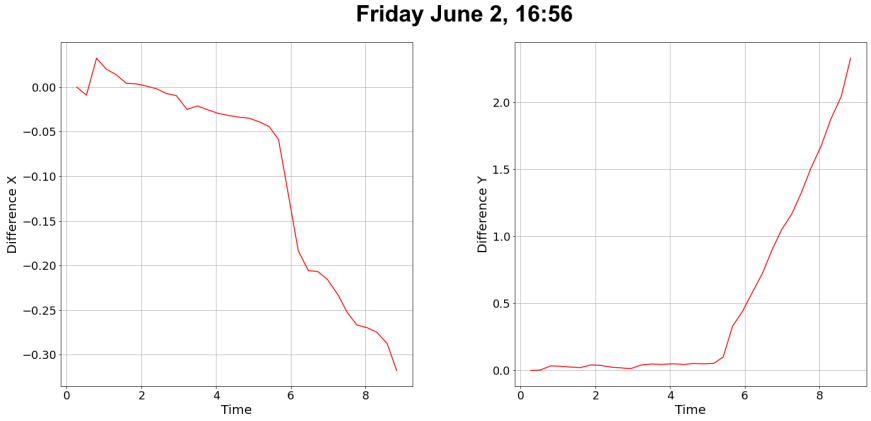


Figure 5.8: Aruco vs OptiTrack difference

5.5.1 Conclusion

Overall the Aruco tracking works well. It is however limited in the sense that due to some unforeseen issues. One issue is that the further away the drone gets from the marker, the less accurate it is. This could indicate that the camera calibration done in Section 4.3.1, might have been a bit imperfect. It could also be an artifact of the low resolution required to run the camera with an acceptable FPS. A third and possible reason could also quite simply be that the camera is not mounted perfectly and actually is tilted a little bit so that the transform becomes more inaccurate the further away the drone is.

5.6 Issues Encountered Under Testing

Under testing the system, a lot of issues were encountered. Some of them were solved but some of them were not solvable in this project and is therefore mentioned here in case someone wishes to continue or rebuild the project.

5.6.1 PI-Lead Erratic Behavior

When testing the PI-Lead controller for height, it was shown that the drone would move up and down erratically for no apparent reason. What turned out to be the issue was that the Teensy had a height controller implemented. This controller were ofc disabled, but as it turns out, it wasn't for some reason. This meant that the PI-Lead height controller designed in this report was fighting against an internal unstable P Feed Forward controller, which was fusing barometric and accelerometer data to estimate height. As stated earlier, the drone's internal height estimation was way off and therefore not used for any of the controllers.

5.6.2 Aruco Tracking

In regards to the Aruco tracking that was discussed earlier, there were some issues that were encountered that could not be fixed. One of those was that the webcam used for tracking had auto exposure enabled. This meant that when the drone took off it would increase the exposure, because the ground below it was black asphalt. This lead, meant that the drone were unable to see the Aruco marker, and therefore lost tracking if it flew higher than 1-2 meters above the ground. A second limitation was that the webcam had fairly limited FOV and because of this needed to fly high to be able to see the marker, which it couldn't because of the auto exposure issue.

5.6.3 Gyro Drift

Something that was discovered quite late is that the internal gyro seems to drift. The Teensy has a 10 degrees of freedom IMU to measure its orientation. It does this by fusing all the measurements from the IMU together with a Madgwick filter. This filter can be tuned such that it should not experience any significant drift. It does however seem to not work properly as the drone's gyro has a tendency to drift over time. This combined with the fact that the drone has a max bank angle has led to a crash because the drone were unable to counteract the drift due to the max bank angle.

A quick fix for this was to increase the max bank angle bank so that the positional controller can counteract the drift and set a max flight time of 120 seconds. This is not ideal but meant that the drone was able to fly, and the controllers could be tuned and tested.

5.6.4 ESC Attachment Point

One issue that led to a catastrophic failure was a loose wire on one of the ESC's. It did happen once when another connecter was disconnected. It was caught on a preflight check that the wire was loose. This connector was therefore squeezed on and it never came loose again. Despite the fact that all the connectors were taped and the drone was

Figure 5.9: Drone after the crash

pre-flight tested before the start of any testing, it happened again with another connecter. This led to the drone trying to do what can only be described as an unsuccessful barrel roll. The crash broke off 1 motor, the DC-DC buck converter for the raspberry, and the raspberry itself. In Figure 5.9 it is quite clear which motor was the culprit behind the crash.

6

Conclusion

This thesis set out to make a flight controller for autonomous operation of the Hexacopter Zephyr. The requirements of this task were that it should be able to deliver a parcel to a house (See Chapter 1).

6.1 Simulation

This chapter set out to derive the equations necessary to model a hexacopter, and then use said model to design controllers for X, Y, Z, and Yaw. The controllers were designed based on a set of criteria as mentioned in Section 3.5. The chapter shows how these controllers were designed, and evaluated that their simulated performance fulfills the criteria set in Chapter 1.

6.2 Real-Time System Implementation

This chapter looked at implementing the controllers, the two tracking methods, and fixing any known issues on the drone such that it was flight ready. The controllers were successfully implemented, by using bilinear transform to get them to the discrete z-domain such that they could be implemented in c++.

6.3 Real-Time System Results

This chapter set out to prove that the drone's positional system works, but also that the simulation model defined in Chapter 3, represents the Real-Time System.

For the height controller, two implementations were made. The PI-Lead design had erratic behavior, but it turned out to be because of an unforeseen issue with an internal controller trying to fight against it. The second controller was tested after the issue was fixed. This controller was a cascading design. It had a tendency to do some minor oscillation, but it is believed that this could be easily fixed with a bit of fine-tuning. Overall the simulated version had not the exact same behavior, but it did give a good starting point for the initial controller parameters.

The X-Y positional system was proven to be fairly accurate, based on the data collected from test flights. It was proven that its overshoot was below the maximum allowed overshoot, which was defined in Chapter 1.

The Yaw controller was not tested extensively, which meant it was never pushed to its limit. However, when given a simple task such as keeping a current heading, it did this to perfection and was never off with more than 0.18 degrees.

The Aruco tracking system performs well overall. However, it has limitations that arise from unforeseen problems. One particular issue is that the accuracy of tracking decreases as the drone moves farther away from the marker. This suggests that the camera calibration may have been slightly imperfect. Another possible factor could be the need for a lower resolution to maintain an acceptable frame rate, which might introduce some inaccuracies. Additionally, it is plausible that the camera itself is not mounted perfectly and is slightly tilted, resulting in increasingly inaccurate transformations as the drone moves further away. Figure 5.8 sums up the result pretty nicely.

6.4 Summary

Overall the simulation seems to match the drone to a certain degree. It was shown that the controllers simulated, and implemented share characteristics that suggest a similarity. With the results achieved, it is believed that the drone in its current state does live up to the specifications discussed in the introduction. However, the height control would need some fine-tuning to ensure a more stable landing.

6.5 Discussion

One of the most tricky parts of this thesis was the actual integration of all the systems. Because there were so many systems that had to interact with each other, meant that it was quite difficult to debug. Most of the actual time was spent on debugging and finding out why the drone didn't act as it was supposed to. However, in the end, it seems that most of the bugs were cleared out and the actual drone mimicked the simulation to a degree.

Overall the project seems to be somewhat successful. The controllers are assumed to need some fine-tuning. It at least gives a good starting foundation to continue work and further develop the drone software.

7

Future works

This chapter will look at improvements and different approaches that could be interesting to look at if someone wished to continue the project.

7.1 Aruco Tracking

The performance of the Aruco tracking was not ideal. There are two approaches to solving this issue, but both would require hardware modifications.

The first solution would be to equip the drone with a higher quality camera, with higher FOV. This would also require the drone to have more processing power, so a good suggestion would be to equip the drone with a Jetson Nano to increase its image processing power.

The second solution would be to equip it with a gimbal. This would allow the drone to track the Aruco code with the gimbal, and therefore not require a higher FOV camera. Furthermore, when a code was found, it should in theory be able to track it and then reduce the FOV of the camera, to process even less information.

7.2 PI-Lead Height Controller Test

As mentioned Section 5.6.1, the PI-Lead controller for height, was tested but not properly. It would therefore be ideal to do an actual test to see if this controller had a bit better performance than the cascading design, as it was in the simulation.

7.3 Gyro Drift

It was discovered that the internal Madgwick filter, running on the Teensy had a tendency to drift over time. This led to the drone not being able to fly for more than 120 seconds before the drift became too much. A Madwgick filter is a more advanced version of a simple sensor fusion filter, but the idea is similar. It is therefore suspected that the main issue is that the linear acceleration part of the filter is acting too slowly. The linear acceleration data is normally used to stabilize and ensure that the filter doesn't drift. Since it can measure the direction of linear acceleration, it can measure earth's gravity, which can be used to calibrate the filter.

7.4 Auto Landing

The auto-landing feature which was disabled would be a great addition to the drone. However, for it to work, the drone should be outfitted with something to better estimate its height. Furthermore, it should be reworked to act a bit more intelligent, than just turning down the throttle slowly.

A proposal would be to use a laser-based range finder to estimate the height of the drone.

7.5 Safety Measures

As mentined in the report, the drone had an unfortunate crash. This crash was mainly caused by a loose connector. The suspected reason for the loose connectors is that they are fairly exposed on the frame. This combined with a fairly flimsy landing gear is not ideal. A solution to this would therefore be to print some sort of guard for them so they do not take a direct impact if the drone. Furthermore, in regard to the landing gear, a better approach would be to attach the gear directly under each motor. This would increase the surface area the landing gear covers, which would make it more resistant to tipping over. An easy solution would be to print some feet and attach them to each of the propeller arms of the drone.

Next up, it would be suggested to make a helmet for the drone in the case of a crash. The helmet's main goal would be to protect the electronics. Putting the Raspberry Pi, Teensy, and IMU on rubber mounts would also help. It might even reduce the amount of noise from the IMU.

Lastly, it would be suggested to implement a recovery feature, that allows the drone to reconfigure in case of power loss on a motor. If this was implemented, it would have saved the drone from a crash.

7.6 Rework Data Logging

The data logging for the software on Raspberry is currently not ideal. The system implemented on the Teensy is much better and makes it easier to debug the drone. It would therefore be ideal to make a similar system for the Raspberry.

7.7 Different Tracking Approach

Something that could be really interesting to investigate would be to use Visual Slam on the drone. This should in theory be able to both estimate the orientation and position of the drone. This would however, require better processing hardware, which might increase battery consumption drastically and therefore not be a good approach.

7.8 Human Detection

The drone is in essence a small flying lawnmower. If it were to fly completely on its own, it should have some way of detecting and avoiding people, and even completely shut down if someone gets too close. The idea would be to equip it with a camera and look for people, and then determine if its safe or not to fly in the area where it is.

Bibliography

[1] *Denmark No Fly Zones*. 2023. URL: https://droner.dk/newsroom/oplas-alle-nfz-zoner-via-qualified-entities-program-qep.

[2] *Px4 - Multicopter PID Tuning Guide*. 2023. URL: https://docs.px4.io/main/en/config_mc/pid_tuning_guide_multicopter.html.

[3] Lucas M. Argentim et al. "PID, LQR and LQR-PID on a quadcopter platform". In: *2013 International Conference on Informatics, Electronics and Vision (ICIEV)*. 2013, pp. 1–6. DOI: 10.1109/ICIEV.2013.6572698.

[4] Mohamed Okasha, Jordan Kralev, and Maidul Islam. "Design and Experimental Comparison of PID, LQR and MPC Stabilizing Controllers for Parrot Mambo Mini-Drone". In: *Aerospace* 9.6 (2022). ISSN: 2226-4310. DOI: 10.3390/aerospace9060298. URL: https://www.mdpi.com/2226-4310/9/6/298.

[5] *GPS Accuracy*. 2023. URL: https://www.gps.gov/systems/gps/performance/accuracy/.

[6] *OptiTrack - Motion Capture Systems*. 2023. URL: https://optitrack.com/.

[7] *Here 3+ - Manual*. 2023. URL: https://docs.cubepilot.org/user-guides/here-3/here-3-manual.

[8] Pengcheng Wang et al. "Dynamics modelling and linear control of quadcopter". In: *2016 International Conference on Advanced Mechatronic Systems (ICAMechS)*. 2016, pp. 498–503. DOI: 10.1109/ICAMechS.2016.7813499.

[9] Jens Christian Andersen. *Elektro - Drone Control*. 2020. URL: http://rsewiki.elektro.dtu.dk/index.php/Drone_control.

[10] *Control Tutorials - For Matlab and Simulink*. 2023. URL: https://ctms.engin.umich.edu/CTMS/index.php?example=MotorSpeed&section=SystemModeling.

[11] MaxP. *Autoquad - Motor Mixing Table Setup*. 2014. URL: http://autoquad.org/wiki/wiki/configuring-autoquad-flightcontroller/frame-motor-mixing-table/.

[12] Christian Andersen & Ilmar Santos & Hans Henrik Niemann & Ole Jannerup. *Feedback Control Techniques - for practical PID design*. 1st Edition. DTU Lyngby: Polyteknisk Boghandel & Forlag, 2021.

[13] *PJRC - Teensy Propshield*. 2023. URL: https://www.pjrc.com/store/prop_shield.html.

[14] *OptiTrack - NatNet SDK*. 2023. URL: https://optitrack.com/software/natnet-sdk/.

A

Appendix

A.1 Codebase

All code for the drone and some of the Matlab files can be found on GitHub. Note that the codebase is up to date, but the Matlab files are not.

https://github.com/Luup850/Zephyr-drone

A.1.1 drone

The folder drone contains a few folders, some used for testing and some for the software running on the drone. The following folders are the important ones:

- **drone_ctrl**: The software running on the Teensy (Made by Jens Christian Andersen).
- **mission**: This is the main part of the autopilot. This is where the controllers and tracking systems are located. All of this code runs on the Raspberry.
- **Python**: Contains script for data analysis and the logged data.
- **matlab**: Contains the modified version of Jens Christian Andersens drone model, together with other scripts.
- **Docs**: Mainly contains plots used in this report.

A.2 Real System measurements

A.2.1 Friday June 2, 17:35

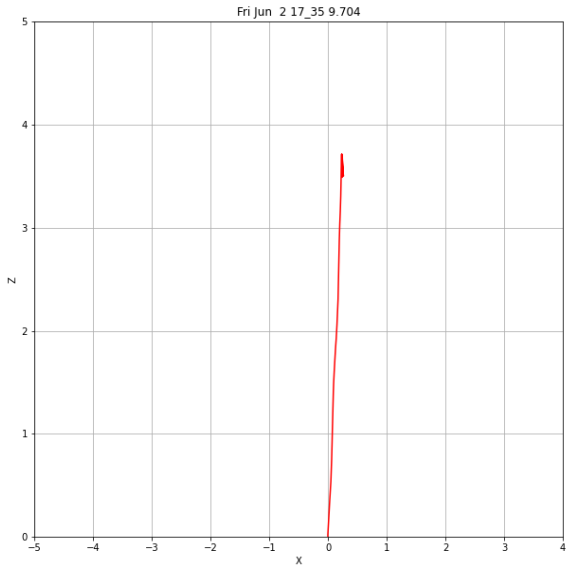


Figure A.1: X position plotted against Z position

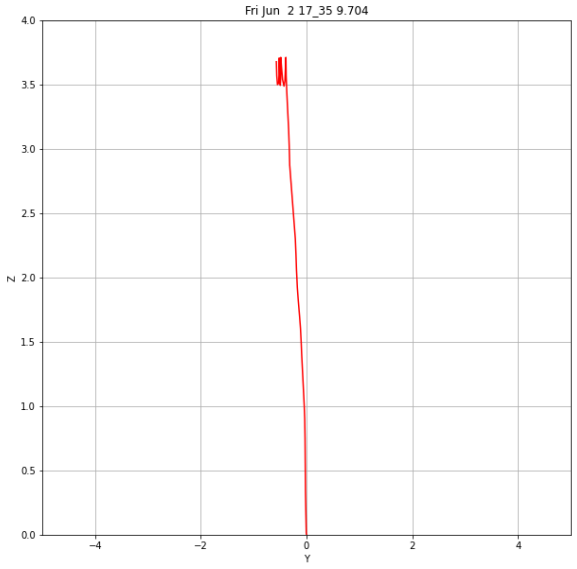


Figure A.2: Y position plotted against Z position