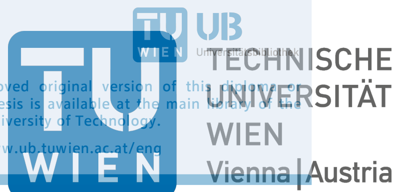
Die approbierte Originalversion dieser Diplom-/Masterarbeit ist in der Hauptbibliothek der Technischen Universität Wien aufgestellt und zugänglich.
http://www.ub.tuwien.ac.at

The approved original version of this diploma or master thesis is available at the main library of the Vienna University of Technology.
http://www.ub.tuwien.ac.at/eng



UNIVERSITAT POLITÈCNICA DE CATALUNYA BARCELONATECH

Master Thesis

Attitude control algorithm for Quadcopter

Design and implementation for indoor tasks

Florian Schebesta

Master Thesis

Attitude control algorithm for Quadcopter

Design and implementation for indoor tasks

carried out for the purpose of obtaining the degree of Master of Science (MSc or Dipl.-Ing. or DI), submitted at TU Wien, Faculty of Mechanical and Industrial Engineering, by

Florian SCHEBESTA

Mat.Nr.: 1225737

Hauptstraße 211

2723 Muthmannsdorf

Austria

under the supervision of

Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Martin Kozek

Institute of Mechanics and Mechatronics, E325

Technical University of Vienna

Dr. Bernardo Morcego Seix

Department of Automatic Control, ESAII

Universitat Politècnica de Catalunya

Dr. Ramon Peréz Magrané

Department of Automatic Control, ESAII

Universitat Politècnica de Catalunya

I confirm, that going to press needs the confirmation of the examination committee.

Affidavit

I declare in lieu of oath, that I wrote this thesis and performed the associated research myself, using only literature cited in this volume. If text passages from sources are used literally, they are marked as such.

I confirm that this work is original and has not been submitted elsewhere for any examination, nor is it currently under consideration for a thesis elsewhere.

Vienna, April 2018 ___________________________

Florian Schebesta

Acknowledgement

First and foremost I would like to thank my family, especially my mother for supporting me through all these years of my studies by always showing me that there is a light at the end of the tunnel during tough times. With all of you I learnt to see that it's the difficulties in life that make you walk easier later on if you overcame them. Thank you *Mama*, for never putting an obstacle in my way when I wanted to go my own. You encouraged me to dare the step and leave home for a while and you did not think of yourself but of me and that it will be good for my evolution.

My appreciation also extends to Dr. Martin Kozek for accepting the leadership for this thesis bearing in mind possible complications that can occur when researching about a project a big amount of time at a different university abroad. He also helped with his great knowledge and managing skills to organize my thesis properly and giving me valuable inputs when uncertainties occurred.

Furthermore, I would like to express my deep gratitude to Dr. Bernardo Morcego Seix and Dr. Ramon Peréz Magrané for introducing me into the world of ROS, showing me that Quadcopters are more than just toys but rather arousing interest in specializing me more in this topic. You also gave me the chance to develop my project at your university taking into account the short amount of time available for work. Thank you for supporting me after I left by providing all your help possible being about 1500 km away. *Moltes Gràcies!*

I want to thank the Vienna Technical University and all of its professors, lecturers, assistants and other employees for guiding my colleagues and me through a period of hard but fair exams, studying uncountable hours and works that helped to prepare for the future professional life. Also, I am very thankful that the opportunity to study abroad was given to me by participating in the ERASMUS+ program.

Last but not least, I am very grateful, that the Universitat Politènica de Catalunya, especially the ESEIAAT gave me the chance to be part of it for a semester and get to know their workflow and their university life which is, although not that far away, different. I would recommend everyone who has the chance to do such an exchange semester to take it because it is an unforgettable experience as you are able to see the world "with different eyes" thereafter.

Kurzfassung

Um den Studenten die Möglichkeit zu bieten, in der Praxis ihre erlangten Kenntnisse über Regelungstechnik, Programmierung und der Interaktion zwischen Computer und automatisierter Maschine oder Roboter umzusetzen, wird an der Universitat Politècnica de Catalunya ein unbemanntes Luftfahrzeug (UAV…unmanned aerial vehicle, oft: Drohne) eingesetzt. Es handelt sich dabei um das Modell „Hummingbird" des Herstellers intel (früher Ascending Technologies), der sich auf die Herstellung von Drohnen für Forschungszwecke spezialisiert hat.

Das Ziel der vorliegenden Arbeit ist es, eine Lageregelung für die Drohne zu entwerfen, die es ermöglicht die Position zu halten. Im Unterschied zu bereits entwickelten Algorithmen, die größtenteils auf dem Einsatz von GPS als Hilfsmittel zur Positionsregelung basieren, kann diese Technologie hier nicht genutzt werden, da die Aufgabe in den Laborräumlichkeiten der UPC zu erledigen ist. Zusätzlich zu den bereits vorhandenen Prozessoren, wird noch ein weiterer Computer montiert, auf dem das Betriebssystem Linux läuft. Weiters ist die Bewältigung der Aufgabe mittels ROS, einem in der Robotertechnik viel verbreiteten Betriebssystem, erfolgen. Das auszuführende Programm wird in der Programmiersprache C++ geschrieben und in ROS ausgeführt. Während des Fluges kann eine drahtlose Verbindung via WLAN zwischen dem ortsfesten Computer und der Drohne hergestellt werden, um Änderungen vorzunehmen. Der Algorithmus wurde zuerst mittels Simulink simuliert und anschließend als C++ Code am Prozessor der Drohne implementiert. Die Reglerparameter wurden mittels numerischer Optimierung des Simulink Modells ermittelt. Durchgeführte Experimente am realen Objekt bestätigten die aus der Simulation gewonnenen Parameter der Regelung.

Diese Masterarbeit wurde im Rahmen eines ERASMUS+ Auslandssemesters an der Universitat Politècnica de Catalunya durchgeführt. Der Auslandsaufenthalt erstreckte sich von September 2017 bis Jänner 2018.

Abstract

For research reasons, the Universitat Politècnica de Catalunya (UPC) uses an unmanned aerial vehicle(UAV) for the students to obtain knowledge about control algorithms, programming and the interaction between user interfaces, that is, a computer and a robot or an independent machine acting according to the implemented algorithm. The UAV is the model “Hummingbird”, manufactured by the Company intel (former Ascending Technologies), which developed an own series of UAVs especially designed for researching reasons.

The target of the project described in this thesis is to implement an attitude control algorithm so that the UAV can maintain its position. Apart from existing control algorithms already implemented in commercial drones which use GPS to obtain data of the actual position, there has to be found another solution as the realization of this project is to be made in the laboratories of the UPC, i.e. indoors. A processor running the operating system Linux will be mounted on the UAV, in addition to the existing two. The whole system will also run ROS, a special operating system and base for operating robots and other autonomous vehicles. The program is written in the programming language C++ and executed in ROS. Through the computer on the vehicle, it can operate autonomously. Nevertheless, a connection between the stationary main computer and the flying device can be established via Wi-Fi to send correcting commands in case of failure. The control algorithm was developed using a simulation software called Simulink where the parameters of the model were obtained. The parameters of the controllers were then optimized using numerical optimization based on the Simulink model. Experiments with the code implemented on the processor of the vehicle proved the parameters found in the simulation were right.
This work was developed during an ERASMUS+ semester abroad at the Universitat Politècnica de Catalunya. The stay lasted from September 2017 until January 2018.

Table of Contents

Acknowledgement .... **IV**

Kurzfassung .... **V**

Abstract .... **VI**

Table of abbreviations **9**

1. Introduction **10**
- **1.1. Motivation** **10**
- **1.2. State of the art** **11**
- **1.3. Previous works** **13**
- **1.4. Objective and range** **14**
  - 1.4.1. Assumptions 15
  - 1.4.2. Range of work 15

2. Description of the system **16**
- **2.1. General** **16**
- **2.2. Hummingbird** **17**
  - 2.2.1. General configuration 17
  - 2.2.2. Modifications 20
- **2.3. Odroid XU4** **22**
- **2.4. ROS** **24**
  - 2.4.1. General description 24
  - 2.4.2. Concept of ROS 26

3. Modeling **30**
- **3.1. General** **30**
- **3.2. Quadcopter model** **31**

4. Simulation **36**
- **4.1. General** **36**
- **4.2. Simulink** **37**
- **4.3. Quad-Sim** **38**

5. Experiments **46**
- **5.1. Experiments on the Simulator** **46**
- **5.2. Empirical controller tuning** **54**
- **5.3. Numerical optimization for tuning** **58**
- **5.4. Programming** **66**
  - 5.4.1. Data acquisition 67
  - 5.4.2. Control computation 70
  - 5.4.3. Command publication 74
- **5.5. Experiments on the drone** **78**

6. Conclusion **80**
- **6.1. Findings** **80**
- **6.2. Outlook** **81**

7. Table of Figures **82**

8. Bibliography **84**

Annex I **87**

Annex II **89**

Annex III **92**

Table of abbreviations

AscTec	Ascending Technologies
DAC	Digital-to-analog converter
DOF	Degrees of Freedom
GPS	Global Positioning System
GLONASS	Global Navigation Satellite System
HLP	High Level Processor
IMU	Inertial Measurement Unit
LLP	Low Level Processor
NED	North East Down
OS	Operating System
ROS	Robot Operating System
UAV	Unmanned Aerial Vehicle
VTOL	Vertical Take-Off and Landing

1. Introduction

1.1. Motivation

In the past, automation was the determining term in various fields of mechanical engineering as people could be replaced by robots to complete dangerous tasks as well as reducing the risk of committing errors induced by monotony of a certain task. This motto has also reached the highly developed branch of aviation looking for solutions to add even more security to the most secure means of transportation by eliminating the possibility of human faults. Although technologies in this sector are already designed to minimize the real interaction of persons by adding intelligent systems to maneuver through difficult weather or avoid such dangerous zones or even fly in autopilot mode most of the time, there is still at least one human being on the plane. In case of any malfunction the possibility of a person losing a live is very high in aviation, although the number of accidents is very low. To remove the risk of a person get hurt, engineers aimed to eliminate the necessity of people on a flying object. Therefore, if it is necessary to fly in dangerous areas such as war zones, regions of difficult weather situations or low visibility the maximum damage that is possible will affect only the material and no person. Of course, there is also a chance that someone on the ground will be hurt by the plane crashing down, but that can sometimes neither be avoided by a human pilot nor by artificial intelligence.

Using so-called Unmanned Aerial Vehicles (UAV) accomplishes this task perfectly as they have by definition no person on board. Their degree of automation can vary from a necessary ground control station to full autonomy by control on board. Many of these vehicles are using a positioning system like GPS or GLONASS to navigate to desired points and to obtain information about the current position. However, there are also regions where positioning systems are not available as they are not covered or zones where receiving is limited because the signal is blocked by buildings, e.g. in urban areas with narrow streets and high buildings. Yet the most popular area where other systems for positioning have to be used is indoors as there is almost always just a weak or no signal. As in various technological projects, the key to find a solution is to copy behavior from the nature because there are a lot of highly developed systems made very robust where failure may be deadly. Thus, for orientation systems the aim is to copy human senses as people orientate themselves mostly using tactile, audible or visual detection. Using sound systems may protect an autonomously flying drone from unwanted touching of objects but their system is more often used to keep a distance than to determine an absolute position.

In contrary, visual systems work like an “eye” of the drone, enabling it to determine its position or finding a certain goal which has to be searched in order to fulfill a task. Furthermore, rotorcrafts are in contrast to most fixed wing aircrafts not able to realize vertical starting or landing maneuvers (VTOL). These may be also very important because the field of application for UAVs are often areas with little space in the plane where only a vertical take-off is possible. Another difference to the fixed-wing aircrafts is the possibility of keep a position in the air in so-called hovering mode.
This may be of interest if the vehicle has to deliver anything by slowly rappelling something or because something has to be supervised or any other thinkable duty where holding a defined position for a certain amount of time is necessary. Additionally, because flying objects unlike cars or trains have no preferred trajectory, a great number of obstacle can block the way as it may change each time. For the above-mentioned reasons, a very exact control of the UAV is indispensable to keep positions as precisely as possible and to avoid crashes when space is limited and through deviations in the attitude, the vehicle does not take-off vertically but with a slight drift in one direction.

1.2. State of the art

Research on UAVs is a relatively recent topic, as the problems occurring and therefore needed development increases steadily with the higher demand of domination of such vehicles because of the rising interest and demand both in military and civil applications. Nevertheless, the interest in autonomy and thus control of such vehicles has started very early, before the use of drones became commonly known. Altuğ et al. [1] were one of the first to tackle the problem of making quadrotor vehicles autonomous by control algorithms although the movement was limited to altitude and yaw angle control. They also implemented a visual system to determine the actual position of the helicopter by observing the vehicle with a ground camera. [2] used a camera mounted on the vehicle serving as base for position and orientation control. Although cameras were used to fly indoor substituting GPS systems for their poor performance, another task was to avoid obstacles as the flying environment may not be known formerly. In [3] there is a solution presented for avoiding obstacles by using optical flow. This shows the disadvantages of heavy computation, whereas other solutions, e.g. in [4] and [5] used radar systems to detect the environment. Bloesch et al. [6] combined the former mentioned tasks of path planning by avoiding collisions as well as the actual controlling task in unknown, GPS-denied environments by using an onboard down looking camera which tracks the movement but also builds a map of the surrounding region.

Errors occurring in [6] were treated in [7] where the slow pose update of the onboard camera with respect to the high agility of the UAV was tackled by filtering the visual input with data from inertia sensors integrated on the vehicle. The drone used in [7] is very similar to the drone used in this thesis as it is the previous model of the same company. Furthermore, the implementation of the system via ROS is another connection and helpful for this work.

For research purposes, systems for the position and orientation feedback change from onboard cameras to external cameras which track the UAV in its movement, as they were used in [1], [8] and [9]. Additionally, the environment in case of this project is known as it is well defined and covered by the cameras. Work on pre-defined regions is presented in [10] using two concentric circles as markers for the estimation of the position and a single camera on board.

1.3. Previous works

Previous research projects on this topic were already conducted at the Department of Automatic Control. Cristian Martínez Céspedes developed in [11] for his master's thesis the image recognition system in the laboratory. He uses four infrared cameras which observe the pre-defined space where the cameras are mounted. The task was to implement an algorithm so that the cameras can observe markers, e.g. mounted on a flying helicopter, which give information about the current position and attitude of the vehicle with respect to the fixed coordinate system of the laboratory. With so-called blobs, which are groups of pixels with same properties (such as the same value of color on a grey scale), the markers are detected and distinguished from the rest of the image. The markers are 3 balls and 1 cube made of Styrofoam mounted onto a cross-shaped symmetrical structure (Figure 1-1). To recognize the different markers, it was necessary to vary properties of the markers so that the cameras know definitely which they are observing. The property of choice was the size of the balls as they give the blobs a distinguishable characteristic. Each camera is connected with a ground computer running the operating system Ubuntu and having ROS installed. One single node in ROS gathers the images of all cameras and calculates the current position and attitude from the obtained pictures.

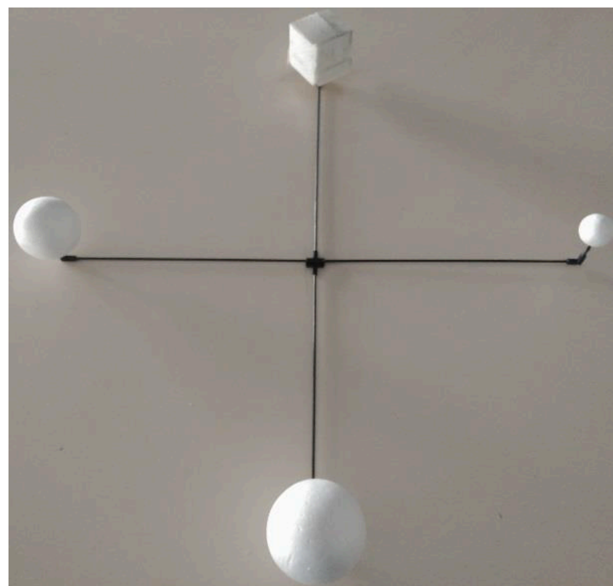


Figure 1-1 Markers used for image recognition

As successor to the above-mentioned project, Carlos Eduardo Barrionuevo Sánchez used in [12] the image recognition developed in [11] to implement the control of a coaxial helicopter. Various changes were made in the work for this thesis with respect to the former work of Cristian Martínez.

The image recognition system was made more robust to changes in the environment which may be a different illumination that changes the distinguishability of the grey scale picture the cameras are observing. For better detection of the markers the cube marker was substituted by another ball with a different size compared to the others.
Due to the lack of a processing unit on board the helicopter, the calculation of the control algorithm is made on a ground computer using Matlab software. The remote control for sending the control commands to the vehicle is connected via DAC cards to the ground computer and sends signals via this connection.

1.4. Objective and range

The goal of the present thesis is a step to autonomous flying of a quadcopter or commonly known as "drone" by implementing an attitude control algorithm. As the vehicle used in this project already has a processing unit, the goal is to make the control independent from a ground station as all of the calculation necessary for the control should be executed on board. The system of choice for communication with the drone is the so-called Robot Operating System (ROS) which is an open-source software often used to communicate with and control of robots. A difficulty for the implementation of autonomous control systems is the high agility of quadcopters which stands in contrast to the limitation of speed of control algorithms which may be immanent to the type of control or can as well be a result of expensive calculations and constrained hardware configuration. Furthermore, the task of controlling a UAV is typically executed outdoors as most of the tasks that should be done by these vehicles have to be executed in that environment. Thus, the system relies on information received from GPS data as it is an easy way to obtain the current position due to its high availability. Nevertheless, the control made in this project has to be done in an indoor environment, where GPS is either not available or only a weak signal can be received which makes it unreliable. However, this assumption is often used for research purposes and is also legit as tasks may be done in a region where GPS is not available or for situations when the GPS system fails.
Based on the previous works presented in the section above, this work is combining the knowledge obtained by the predecessors as the image recognition system as well as the control of a helicopter is used. The substitution of a coaxial helicopter by a quadcopter and the adaption of the recognition system for this vehicle are the obvious tasks that have to be done. Nevertheless, the system of control changes radically as the behavior of a quadcopter is very different from a coaxial helicopter.

Another drastic change is, that the calculation is no longer done on a ground computer but should be done on board. For this reason, the executable programs must obey the limitations of computation power on small processors. The programs should also be robust and matching the agility of the UAV in its speed of reaction.

1.4.1. Assumptions

For the above described reasons, following assumptions have been made before the work was started:

- A quadcopter is object to control
- Existing control should be replaced, but is a back-up for failure
- Modifications to the vehicle are possible
- Communication between the parts of the systems with middleware ROS
- Attitude control is in the focus
- Development of control algorithm is for indoor use
- Laboratory with pre-defined environment for flying
- GPS is not available to obtain information about state or position
- Image recognition system can be used as existing

1.4.2. Range of work

The range of the present work was defined by the following tasks to be completed:

- Mounting of the markers for image recognition on the quadcopter
- Learning about ROS
- Creation of a useful model of the system for control
- Creation of a control algorithm
- Simulation of the control on computer
- Implementation of control on the vehicle
- Combination of image recognition and control
- Tests of control in defined environment

The image recognition system and its already developed markers as described above turned out to be difficult to mount on the existing vehicle as their system is built for a helicopter with a skid landing gear. The implementation of such a system may be a task for further research on this project.

2. Description of the system

2.1. General

The term used in this context is Unmanned Aerial Vehicle (UAV), commonly also known as drones. The topic covers different levels of automation beginning with the use of a ground control station to a full autonomous vehicle. Generally, the name does not specify the configuration of the vehicle, thus, various types of aerial vehicles, so-called fixed-wing aircrafts as well as rotorcraft vehicles are included. However, the focus lies in the field of rotary wings due to their most distinctive features with respect to fixed wing aircrafts: their ability to take-off and land vertically and the possibility of hovering. These features are often asked for in applications because a task may have to be fulfilled at a certain position by staying there for a given time. Limited space for take-off and landing mostly make so-called VTOL (Vertical Take-Off and Landing) operations obligatory. In contrast to typically used rotorcrafts used in manned aerial vehicles, the UAVs often are so-called multicopters, meaning that their thrust is not provided by one central rotor wing but distributed by four (Quadcopter), six (Hexacopter) or eight (Octocopter) rotors. However, [13]also states the possibility of using coaxial helicopter configurations to fulfill the demands of VTOL. The advantage of multicopters is that there is no need for the typical rear rotor of helicopters which is needed apply a torque against the helicopter body's tendency to rotate around its own axis in reaction to the main rotor moment. For the case of a quadrotor, a pair of rotors is turned in another direction as the other one and therefore equals the urge to turn around its height axis. If it is intended to turn around this axis (yaw angle), the rotating speeds have to be modified in order to unbalance the quadcopter in this direction. The rotors are distributed in symmetrical order in one axis, but not necessarily in both axes of the horizontal plane as one can see in Figure 2-1 below, where the V-formed arrangement of the motors is due to the purpose of having a free view field for the mounted camera. [14]

Figure 2-1 Intel Falcon 8+ drone [14]

Due to a higher number of propulsion units, the rotors have smaller dimensions and each of the motors has to deliver less power, therefore serving the aims of lightweight and small construction.

2.2. Hummingbird

2.2.1. General configuration

The UAV used in this project is called “Hummingbird” and was designed and assembled by the German company “intel Deutschland GmbH”, formerly known as “Ascending Technologies” (AscTec). It is a drone with 4 motors and therefore typically called Quadrotor or Quadcopter. These 4 motors are distributed in a symmetrical manner on the body of the vehicle, so that they form an angle of 90° to each other. The motors are electrical powered brushless motors manufactured by Hacker and have a Power of 80W each. Every motor is connected with its own control unit which has the duty to convert the input voltage into the desired motor speed. By default, the motors come with flexible plastic rotors mounted on the motors.

Figure 2-2 Hummingbird quadrotor

As seen in Figure 2-2 above, the main body of this UAV has the shape of a cross with 4 arms with the same length to ensure the symmetry. It is made from carbon fiber reinforced plastic in order to stick to the premises of lightweight construction. Decreasing the weight of a flight vehicle always means to improve the flight time possible with the same amount of energy available. Figure 2-3 below shows a frame made of aluminum (1), that leads to the possibility of guarding a battery in the middle (2).

Furthermore, as this frame is below the plane where the motors are mounted, it allows the vehicle a safe stand when starting or landing (3). Additionally, rubber rings were added for damping the landing as well as guarding the surface below from scratches by the metal.

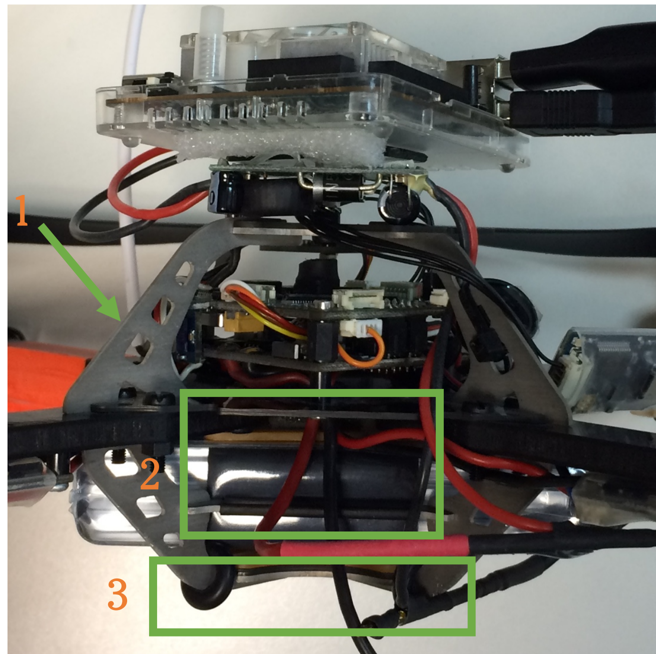


Figure 2-3 Aluminum structure of the main body

Above the battery compartment the main processing unit of the Hummingbird is located. It contains two CPUs which on one hand run the program that controls the drone by default and on the other hand can be modified by the user in order to realize its own control by programming it and flashing it to the so-called High Level Processor (HLP). Furthermore, the vehicle is equipped with an antenna to establish a connection with the radio control (by default the FUTABA F77 transmitter, seen in Figure 2-4) and a GPS module on top of the body to guarantee reachability.

Figure 2-4 Futaba F77 radio control

The control of the Hummingbird uses a 5-channel control by default.

These five channels are:

- thrust of the motors
- roll angle
- pitch angle
- yaw angle
- additional switch to activate automatic control

Motors can be turned on by moving the left stick (1) to its right or left endpoint and holding it there for a few seconds, until the motors have started and are turning on idle speed, if the stick is in its lowest position. Thrust is controlled by the left stick by pushing it up or pulling it down to increase the rotational speed or decrease it, respectively. By default, the position of this stick is in the lowest position and it is internally equipped with a mechanism that keeps its vertical position wherever it is. Furthermore, the yaw angle can be changed by moving the same stick to its left or right for clockwise or counterclockwise turning around the height axis. The default position in this direction is the middle one, where the quadcopter would keep its actual state and stop rotating, the mechanism of the stick contains springs that force the stick to its neutral position when not touching it.
The right stick (2) controls the pitch and roll angles when moving up- and downward or sideways, respectively.

The behavior of the stick is the same in each direction as it is for the yaw angle, so that the neutral position of the stick is in the middle of both axes, where the pitch as well as the roll command would be 0.
The channel for regulation of the thrust regulates primarily all motors on the same level of thrust. In combination with an additional angle command, the rpm of rotors can change in order to achieve the desired attitude. By default, this is controlled via the built-in control algorithm. If the switch for manual or automatic control (3) is set to automatic by flipping it towards the user of the radio control, the control implemented on the HLP takes over the operation of the vehicle.
However, by switching back to the manual mode, the Hummingbird could at any time be maneuvered by using the radio control. For this reason, it is necessary that each time when testing new programs on the vehicle, an experienced pilot is holding the radio control. This pilot should pay attention to the movements of the quadrotor and be able to steer the vehicle from any given position into a stable one and to land thereafter if the drone does not behave as expected before.

2.2.2. Modifications

In order to accomplish the goal of the project, some modifications have been made to the vehicle.
As the task is the position control indoors where GPS is not available or very weak, the GPS sensor (Figure 2-5) was taken off the Hummingbird. With this modification, the quadcopter is left without information of its actual position as there is no additional system implemented by default. Nevertheless, all of the control algorithms, including the default on the LLP need position state data of the vehicle so that its control will work properly.

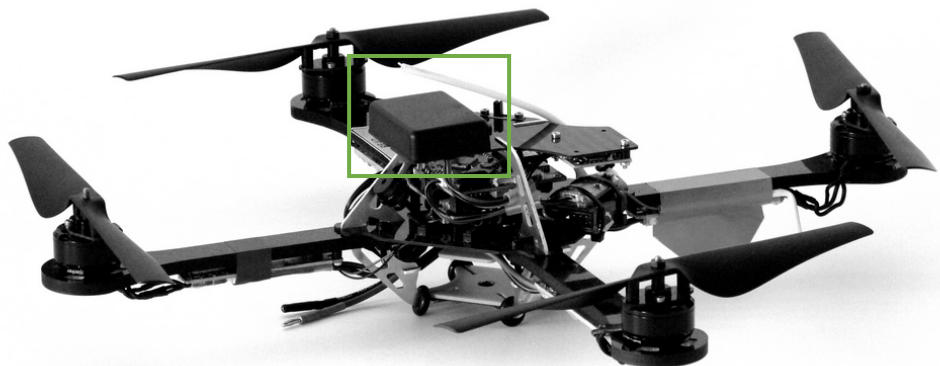


Figure 2-5 GPS Module on Hummingbird quadcopter [27]

To interact with the whole system via ROS, an additional computer was needed which runs this software. AscTec offers its own options to cover this task, called the AscTec Atomboard. However, another solution was chosen due to the limitations in memory and its lower speed compared to the chosen model “Odroid”(described in section 2.3).

Figure 2-6 Odroid mounted on Hummingbird

As can be seen in Figure 2-6 above, this computing unit was mounted on top of the drone with a single screw in the middle making it a simple and stable solution at a time.
The data connection of odroid to the default installed processing unit was solved with a self-made connection similar to that offered by AscTec, converting from a USB 3.0 interface to a serial port.
Due to all these changes, the weight and the inertia matrix of the vehicle are changing and these modifications have to be taken into consideration when developing the control algorithm.

2.3. Odroid XU4

Odroid is a computing device manufactured by Hardkernel mounted onto the Hummingbird quadrotor. It is a so-called Single Board Computer (SBC) and differs from the typically known PCs such as OSX or Windows machines. It uses an ARM processor instead of an Intel processor where the latter has higher efficiency due to its architecture. The OS running on these SBCs are also highly optimized for running on these systems. Open source software makes it easy to adapt to needs for a special project. Furthermore, Solid State technology is used for storage which is another crucial factor regarding speed compared to hard disk devices. The low energy consumption of about 10 to 20 W (instead of up to 1000 W for standard personal computers) allows to use this computing device to be powered by a solar panel or even a powerful battery and therefore it can fulfill tasks with high grades of autonomy. This makes it especially usable for this project, as it makes no sense to use a processing unit that must be connected to power line in a flying object.

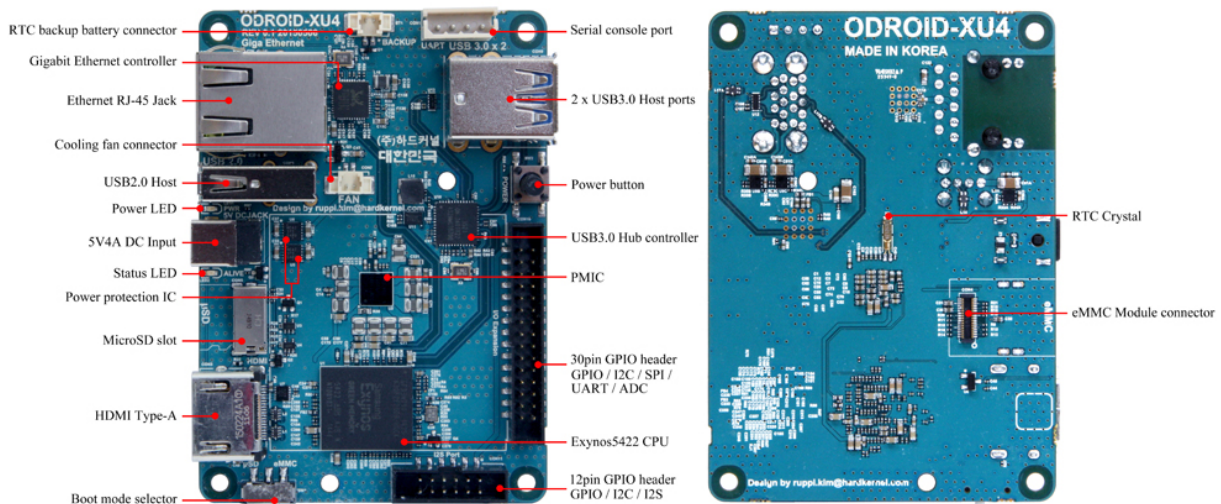


Figure 2-7 Main board ODROID [28]

As seen in the picture above (Figure 2-7), the odroid platform has multiple connection options that all use standardized protocols and ports such as USB, Ethernet or HDMI. However, the connection with the network only with Ethernet interface does not serve in this project, as it is a cable connection and again limits the autonomy of the UAV. Therefore, a Wi-Fi antenna provided by Hardkernel (can be seen in Figure 2-6) is used and plugged into one of the USB ports. To establish connection with the processors of the Hummingbird, another USB port is used to connect with a converter cable that connects with the processing unit using a serial port.

The operating system (OS) that runs on odroid is a Linux-based version of Ubuntu 14.04. This open source OS is based on a Command Line Interface, although many distributions also offer Graphical User Interfaces (GUI). However, the main operation tasks in this project are made using the CLI, as there is no monitor available. To operate the odroid it is connected via Wi-Fi to another computer. On this other computer, the shell command ssh (secure shell) is used to take over control of the OS of odroid. ROS is also installed on the odroid, being the connection of the Hummingbird to the ROS system.

2.4. ROS

2.4.1. General description

To establish a standardized way of communication between the UAV and the base computer, the operating system ROS is used. ROS is an acronym for Robot Operating System which describes itself as a "flexible framework for writing robot software" [15]. It was developed with the aim of delivering powerful tools that are suitable for various tasks in different types of robots. Because that is the crucial point in creating robot software: one might imagine that simple tasks for humans are nearly as straightforward when writing a program for a robot to do the same exercise. And one who has experience on programming software for a special type of robot should easily be able to develop the commands necessary for another type as well. However, due to the different evolution paths and development goals targeted by the creators there is no universal path to dominate all robots. Various types of robots have their own peculiarities which should be considered. Although they are known and it is possible to write a program for a special robot, stability is another topic. Developing robots more with possibilities also makes them more vulnerable to little errors that can cause malfunction in unexpected moments. Therefore, it is more than ever necessary to use a robust and multi-purpose software as it is provided by ROS. In order not to stop developing it is an open-source system that relies on input by its community which provides software elements as well as help with the so-called ROSWiki, an encyclopedia to inform about the use of the system itself or its parts. Thus, the knowledge, distributed all over the planet can be concentrated in one place, as different research teams are specialized in various topics and can share their developments with the community and receive help in subjects they are not as focused onto.

ROS is characterized by following principles:

- Distribution of computation

  Through the above-mentioned topics, it is possible for an (theoretically) infinite number of computers to communicate with one another over the same topic. This means, that it is not only a communication path as usually, where data is sent from one point to another with the possibility of the second unit responding. In contrary, with ROS topics, data can be sent and received (in ROS terms: published and subscribed, respectively) from every element of the network without the necessity of any computer knowing where it is sending his data or where it is receiving it from.

This allows to distribute the calculation of the robot's action and therefore adapts perfectly to the often-practiced distribution of tasks in the robot itself. And even if the system consists only of a single computer, it can be helpful to divide the functions into smaller parts, a method called "complexity via composition".

- Teamwork via open-source

A community developing the software and share it with others makes it possible that everyone can access the knowledge of specialists in their field by using their programs published and commented in the ROS-Wiki, the user community web page. Therefore, not everyone has to study in detail several parts of the problem he is facing, as others may already have done it and can share their expertise.

- Rapid testing

  One should implement the system correct with the two levels of control, i.e. the low-level control which is in direct interaction with the hardware and the high-level stage which duty is to process programs and make decisions based on the data from the low-level processor or from an external computer. Then it is possible to replace the low-level operations with a simulation and therefore try the programs without physically applying it to the robot which may not be available all the time.

To prevent others from making mistakes using ROS, it is also necessary to clarify doubts about the functionality and the range of usability of this system.

- ROS is no programming language

Although ROS uses its own commands and is executed like a program, one should not make the mistake like a lot of people do when they first get in touch with ROS to treat it like a programming code. In fact, ROS is more like an operating system or like an extension to it as it cannot be used as a standalone solution. If you use it with Ubuntu, which is highly recommended (although there are some OSX versions, but they are more or less experimental), ROS is installed like an expansion pack to make it work. However, to fulfill the purpose of creating programs you will need additional software or at least an editor to write programs in the supported programming languages C++ or Python as ROS is also no IDE. The aim of ROS is to execute them via its nodes (see later).

- ROS is not only a library

Although the above described missing functionalities, one must not see the system only as a tool to organize your program structure, because it has its own command-line tools, graphical tools and an included building service.

- ROS does not have a Graphical User Interface

It is not possible to use ROS via a so-called GUI which most computer users are used to. In fact, it can only be accessed via the terminal and there one can use its total functionality. But one should know the commands for every single step as there is no possibility to click on anything nor use a dropdown menu. However, it includes graphic solutions to display movements etc. which depends on the package.
[16]

2.4.2. Concept of ROS

Technically, ROS is a Linux-based operating system that is an extension package to the operating system of the computer. As above mentioned the control and operation of the systems does not work with a graphical user interface, but via the terminal of the operating system. This means that the main tool for entering commands is the keyboard of the computer.

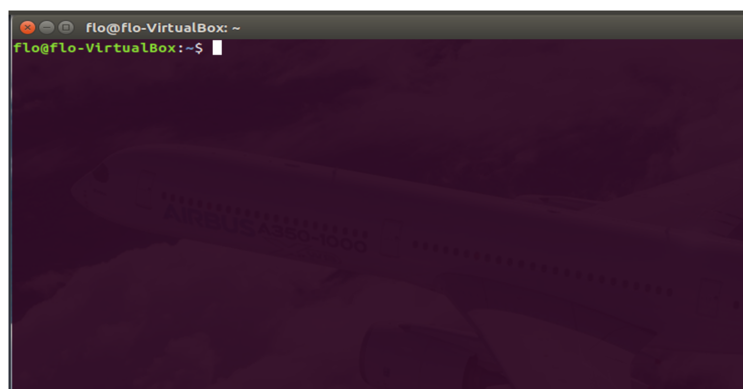


Figure 2-8 Terminal window

Furthermore, the developing company Willow Garage organized ROS in a very structured way so that every action which happens in its environment can be distinguished by both the system and the user and easily assigned to an activity or whatever happens at a certain time. The basic principles and functions of ROS are explained in the following paragraphs.

Nevertheless, it is always recommended to visit ROS's own Wiki page at wiki.ros.org to gain further information or at least for informing about the latest changes or new distributions as developing of programs never stops at a certain point, but rather will be developed even further.
ROS is organized in a three-level structure, based on the tasks fulfilled by the parts in each level.

1. Filesystem Level

This is the organization of ROS on the machine itself. The following list does just mention the main parts of this level.

Workspace

The ROS *workspace* is the environment where all the actions have their root. If one wants the system to work well, it is inevitable to create a workspace and define the path where it is located in the local data system. Otherwise ROS won't find its data and therefore not work properly.

Packages

Packages are the core part of the ROS environment and must be built in order to get the system working. As mentioned above, ROS includes its own building abilities, so that a package can be built via the corresponding ROS command. Packages contain different things depending on their functionality. But among them can be so-called *node,* which are the executable program files, as well as a ROS-independent library or any further software that should be coupled to the function of the package. With the organization in packages it is easier to provide the whole software needed for a special task. Furthermore, it allows to work with different packages at the same time without them disturbing each other. The goal in building packages is to not put too much information or data in one package, but just as much as is needed for it to work appropriately. Therefore, each package is different but everyone contains the two manifests *package.xml*, an explanation what the package is for and who maintains it, i.e. who should be contacted in case of malfunction, and *CMakeLists.txt*, describing how to build the code and where to install it to.

2. Computation Graph Level

In this stage, computing and executing of programs is done. Basic elements of this level are shown below.

Nodes

Nodes are the running instances of ROS program. Every node is supposed to accomplish a different task for the robot, however the real mechanical system may be distributed. As mentioned above, this is a typical strategy in programming robots. A package can contain many nodes to realize a certain task of the system.

Topics

ROS Nodes do not communicate directly, but via *Messages*. These messages are sent from one node by *publishing* it. This task is also done by above mentioned nodes. At the point of publishing, the *Publisher* does not know which node will be receiving the message it sent. Therefore, to identify the message it is publishing and to find the correct receiver, the publisher gives its message a unique identification by publishing it to a certain *Topic*. Another node which wants to receive the message, must *subscribe* to a topic. Then it only receives the messages published on this topic, while others, which refer to another topic may be unseen. With this system of communication, it is unnecessary to know who sends commands and who reads them. Thus, different nodes can publish messages at the same time without the possibility of a communication way being blocked. Also, there can be more receivers, as it could be possible, that more tasks depend on a certain message. The message is the structure of the data, with many standard types like integer, floating point and Boolean predefined.

Services

It is also possible to create messages in the style of request and response. With these so-called *Services*, a *Client* can send a request to the service and wait for its response.

Master

To facilitate the all the communication on topics via messages, the ROS system needs a so-called *Master*. Every system can only have one master which should be started before all the work can be done.

This is just an initial step with makes no changes, but allows the system to work. Therefore, the Master should not be stopped until the whole task is finished and the system can be shut down.

Launch

It is often necessary to start the master as well as a lot of nodes by typing in many different commands in the Terminal. Fortunately, ROS makes this process easier by providing a tool call *Launch*, which allows to a defined sequence of commands all with one single launch command. Furthermore, it can also establish parameters on the ROS parameter server.

3. Community Level

In this level are included all of the ROS distributions that are available for free as an open source software and can be downloaded on the ROS homepage *www.ros.org*. Furthermore, a ROS wiki (*wiki.ros.org*) is established to explain all the details of the different packages that can be downloaded. For beginners, it is also a useful first approach to get familiar with this system as it includes descriptions as well as tutorials to extend from only theoretical explanation into practical application. This makes the first contact easier as it may seem very abstract when first getting in touch with it. As above-mentioned ROS is open-source and therefore reliable on its community which includes users as well as developers who help extend the usability in various regions. With their own projects uploaded, developers help others so that they can use made packages for their tasks thus saving time. Furthermore, there are researchers who are specified in different branches and have an extended knowledge which others may not have. By providing their software, they may help others and get in turn help in branches there are no experts. There is also the possibility to exchange knowledge in the forum established on *answers.ros.org*.
[17] [18]

3. Modeling

3.1. General

In controlling, the base model of the control system always consists of the controller itself and the system to be controlled as well as a feedback loop as indicated in Figure 3-1.

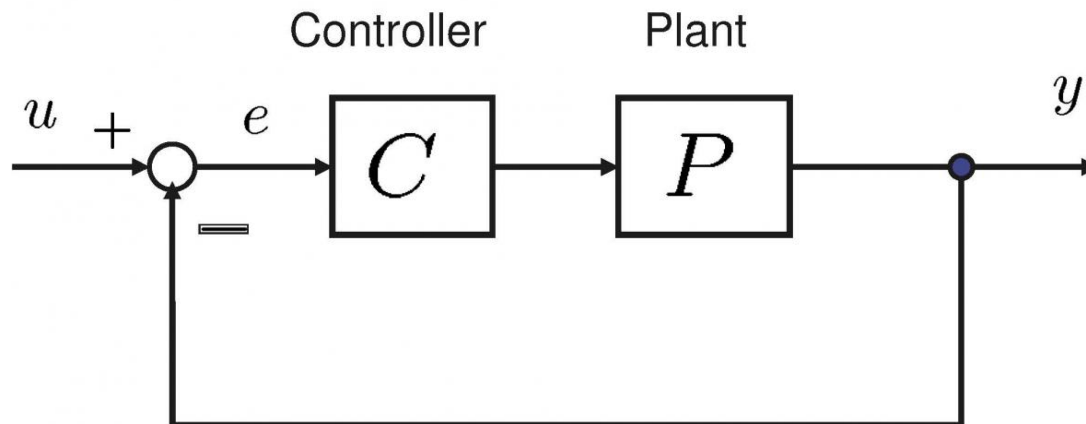


Figure 3-1 Unity feedback system [19]

The plant block P represents the system to be controlled by the controller C, where the controller is reacting to the feedback from the plant. Therefore, control systems are very complex because the whole system has to be considered and the controller and the plant are highly dependent on each other, e.g. the choice of the controller depends on the properties of the plant. And before the design of the controller, often only physical properties of the system to control are known whereas the dynamic response of the system is of interest.

To obtain this knowledge, it is necessary to depict the real mechanical system in a very accurate way so that the behavior is nearly the same. There will always be a difference between the systems as simplifying the mathematical equations is always useful to make the terms tractable. Since the result of the modeling process for mechanical systems will be a set of differential equations it is recommendable to look for assumptions that can be made to reduce the complexity of the system without changing the comportment of the system radically. In many cases, linearization around a work point will help to handle highly non-linear equations. After the modeling process, it is important to validate the new created model by showing that its behavior is as expected and especially comparable to the real physical system. The system will be translated to a form called state- variable form which means that the dependencies on the state variables are perceivable. [20]

3.2. Quadcopter model

Modelling of quadcopters is a complex task as the resulting equations are highly non-linear and the system quadcopter is also a underactuated system as it has 6 degrees of freedom(DOF) and only 4 actuators represented by the speed of each motor. [21] [22] Furthermore, a quadcopter is a so-called Multi Input Multi Output system (MIMO). [20] [22] To create the model, some assumptions were made [22] :

- The body of the quadcopter is a rigid body
- Model for quasi-stationary hover condition
- Cross configuration of the quadcopter
- Center of the body coincides with the center of gravity
- Aerodynamic effects are neglected

To define a model and write its equations correctly, a coordinate system is obligatory. For the model in this project we use two different coordinate systems in order to describe the motion of the vehicle.

First, there is the inertial coordinate system of the earth denominated with the index I with its axes:

$$[x_I, y_I, z_I]^T \tag{3.1}$$

where the z_I axis points upwards.

The second coordinate system is attached to the body frame (index B) in the above-mentioned cross configuration, which means, that the axes x_B and y_B are connected with two perpendicular bars of the cross-like structure of the main body. The z_B axis is again pointing upwards in the direction of the vehicle taking off. The vector of the body systems follows:

$$[x_B, y_B, z_B]^T \tag{3.2}$$

To transform vectors from one coordinate system to another, Euler angles are used to perform a rotation. The order of the rotation follows the scheme Z-Y-X, which means, that the system is rotated first about the z_I-axis by the yaw angle ψ, later rotated around the y_I-axis by the pitch angle θ and finally rotated around the x_I-axis by the roll angle ϕ.

This rotation describes the transformation from the inertial system to the body frame, but the reverse transformation matrix is obtained by simply forming the transposed matrix. The result is obtained by multiplying the three elemental rotations described above.

$$\boldsymbol{R}_{BI} = \boldsymbol{R}_{\phi} * \boldsymbol{R}_{\theta} * \boldsymbol{R}_{\psi} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & c\phi & s\phi \\ 0 & -s\phi & c\phi \end{bmatrix} * \begin{bmatrix} c\theta & 0 & -s\theta \\ 0 & 1 & 0 \\ s\theta & 0 & c\theta \end{bmatrix} * \begin{bmatrix} c\psi & s\psi & 0 \\ -s\psi & c\psi & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{3.3}$$

where
s…sin
c…cos

The final result of the rotation matrix can be seen below:

$$\boldsymbol{R}_{BI} = \begin{bmatrix} c\theta c\psi & c\theta s\psi & -s\theta \\ s\phi s\theta c\psi - c\phi s\psi & c\phi c\psi + s\phi s\theta s\psi & s\phi c\theta \\ s\phi s\psi + c\phi s\theta c\psi & c\phi s\theta s\psi - s\phi c\psi & c\phi c\theta \end{bmatrix} \tag{3.4}$$

The forces acting on the quadcopter are the gravity in the negative z-axis of the earth coordinate system $-z_I$ and the four forces of each of the rotors being assumed positive in the z_B direction upwards as they create a thrust lifting the device.
Naming a vector **r** that denotes the position of the center of mass of the quadcopter in the world frame, the second law of Newton can be written as follows:

$$m\ddot{\boldsymbol{r}} = \begin{bmatrix} 0 \\ 0 \\ -mg \end{bmatrix} + \boldsymbol{R}_{BI} \begin{bmatrix} 0 \\ 0 \\ \Sigma F_i \end{bmatrix} \tag{3.5}$$

where
g…gravity
m…mass of the vehicle
F_i…forces of each rotor

The angular velocities of the vehicle in the body coordinate system are:

$$\boldsymbol{\omega}_{BI} = \begin{bmatrix} p \\ q \\ r \end{bmatrix} \tag{3.6}$$

As the goal is to control the attitude and therefore we are using the roll, pitch and yaw angles, it is useful to describe the angular velocities in terms of the derivatives of the three angles.

As each of the angular velocities $[\dot{\phi}\ \dot{\theta}\ \dot{\psi}]$ are described in different stages of the rotation they have to be transformed correctly to the body coordinate system. Each stage of the three rotations is described by the following turning scheme and its intermediate systems are named in the following manner: I→1→2→B.
Therefore, the angular velocity can be described by:

$$\boldsymbol{\omega}_{BI} = \boldsymbol{R}_{B1} * \begin{bmatrix} 0 \\ 0 \\ \dot{\psi} \end{bmatrix} + \boldsymbol{R}_{B2} * \begin{bmatrix} 0 \\ \dot{\theta} \\ 0 \end{bmatrix} + \begin{bmatrix} \dot{\phi} \\ 0 \\ 0 \end{bmatrix} \tag{3.7}$$

The resulting description after some simple mathematical remodeling is:

$$\begin{bmatrix} p \\ q \\ r \end{bmatrix} = \begin{bmatrix} \dot{\phi} - \dot{\psi} s\theta \\ \dot{\psi} s\phi c\theta + \dot{\theta} c\phi \\ \dot{\psi} c\phi c\theta - \dot{\theta} s\phi \end{bmatrix} \tag{3.8}$$

Every rotor produces an additional moment to the thrust force it is creating, whereas there can be distinguished two pairs of rotors. One of them (1,3) rotates in negative sense according to the right-hand rule applied on the z_B-axis or counter-clockwise when observing the vehicle from above, while the other pair of rotors (2,4) rotates in the other direction. Therefore, they are creating positive and negative contributions to the equation of moments, respectively. Letting l be the length of each of the arms of the cross-like body structure of the quadrotor and the inertia tensor as follows:

$$\boldsymbol{I} = \begin{bmatrix} I_x & -J_{xy} & -J_{xz} \\ -J_{yx} & I_y & -J_{yz} \\ -J_{zx} & -J_{zy} & I_z \end{bmatrix} \tag{3.9}$$

where
I…moments of inertia
J…deviation moments

The values of the inertia tensor can be found on [23].

The Euler equations can be written as follows:

$$\begin{bmatrix} (F_2 - F_4)l \\ (F_3 - F_1)l \\ M_2 - M_1 + M_4 - M_3 \end{bmatrix} = \boldsymbol{I} * \begin{bmatrix} \dot{p} \\ \dot{q} \\ \dot{r} \end{bmatrix} + \begin{bmatrix} p \\ q \\ r \end{bmatrix} \times \boldsymbol{I} * \begin{bmatrix} p \\ q \\ r \end{bmatrix} \tag{3.10}$$

The forces and moments created by the rotors are dependent on their rotational speed and can be modeled like in [24] with a quadratic dependency and some proportional factors as seen below:

$$F_i = k_F {\omega_i}^2 \tag{3.11}$$

$$M_i = k_M {\omega_i}^2 \tag{3.12}$$

where the factors obtained in experiments in [24]are the following:

$$k_F \approx 6.11 * 10^{-8} \frac{N}{rpm^2} \tag{3.13}$$

$$k_M \approx 1.5 * 10^{-9} \frac{Nm}{rpm^2} \tag{3.14}$$

Similar to Figure 3-1, the control whole control system for this thesis could also be described in a block diagram. Figure 3-2 below shows the corresponding diagram for the attitude control loop of the Hummingbird.

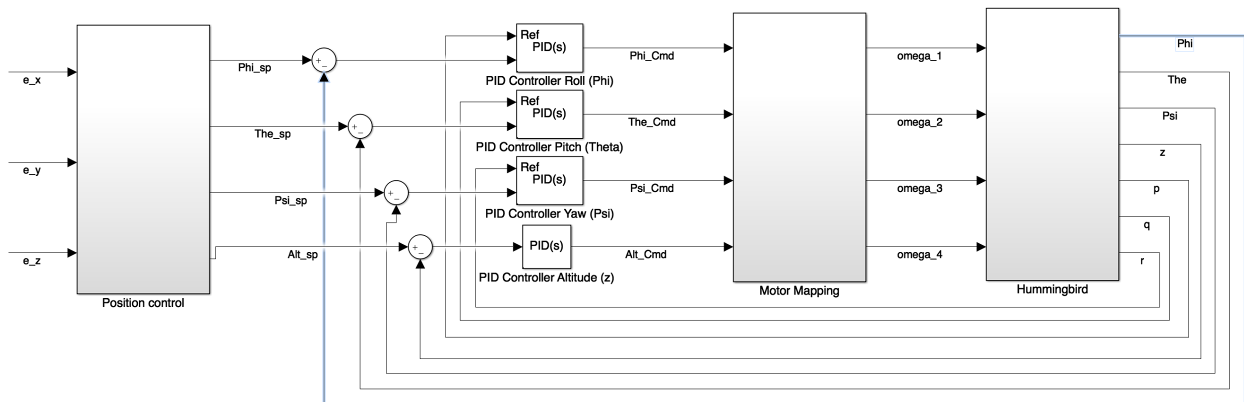


Figure 3-2 Block diagram of the control loop

The figure above is concentrated on the attitude control and therefore just a part of the whole control system of the Hummingbird which is important for this thesis.
As already mentioned, the control of attitude does not work well as a stand-alone solution without a position control, as little deviations will lead to a drift of the vehicle away from its original point if position is not controlled. For this reason, the primary input of the system are the errors in each direction of the position e_x, e_y and e_z. How these errors are obtained is described later in chapter 4.3. The output of this block are then the set points needed for the attitude control.

These set points are like in every control loop used to compute the current error of the system by subtracting the current value of the controlled variable. This controlled variable is obtained by measurements on the plant, i.e. the quadcopter. Furthermore, the structure of the following PID controllers uses the advantage that derivatives of the controlled variable are already available as the measurement unit on the Hummingbird also delivers the actual values of the angular velocities in each of the three axes. Therefore, they are also looped back from the Hummingbird to the PID controllers, except for the altitude controller, which derives directly in the controller. For the above-mentioned reason, the three attitude controllers are strictly speaking no real PID controllers but rather PI+D controllers as they are in fact performing derivative action but the derivation itself is not done from the controlled variable. The inner structure and work principle of the controllers is described in chapter 4 and can be seen in Figure 4-7,e.g. Result of the calculations in the controllers are the command variables for each angle roll, pitch and yaw and for the altitude z. The following block performs the so-called motor mapping of the rotors of the quadcopter. This process converts the commanded angles and the commanded altitude automatically into rotational speed rates for the 4 motors. This mapping is pre-defined and was never changed during the project. Result are the RPM for the 4 motors as manipulated variable of the system. They enter the controlled plant, namely the Hummingbird and change its behavior. Sensors like the IMU already described in this thesis then deliver the current values of the controlled variable as well as other values. Of course the Hummingbird measures more than the seven values depicted in the scheme in Figure 3-2 but they are not represented for better clarity of the picture.

4. Simulation

4.1. General

When designing a new control algorithm for a system, there are always doubts about the success of the new implemented controller as many factors have to be taken into account. One has to depict the real object in a way that it reflects its actual physical behavior and at the same time isn't too complex to be controlled automatically. Furthermore, the mathematical equations have to be set up and there is a lot of potential failure by simply confusing signs or other minor details with tragic effects. Additionally, the control should not only work, but also operate in an appropriate time range that the control reaches its goal according to the physical possibilities.

All these peculiarities are of course also applicable and as those systems are flying, which is per se a very unstable state, they are even more exposed to the danger of malfunction. Those failures can often cause major destruction of the vehicle and therefore must be evited by all means. A useful tool in this context is the use of simulation systems which are basically programs working on a computer which are configured to depict the real system and its controller in the most accurate manner. These systems allow to simulate the process taking place with the new implemented algorithm in a very safe manner as there is no physical movement but only a calculation of what would happen if the control was applied on the real system. Optionally, the output can be in a graphical manner, which often helps the user to see at a glance if the system is working properly or in the other case, what may be going wrong. Another possibility is to just record data during the virtual experiment and analyze this data later by comparing inputs and outputs of the system checking if the comportment is the expected or not and again draw conclusions.

4.2. Simulink

The software used for these simulation tasks is Simulink which is a software package distributed with the program Matlab. Simulink features a so-called block diagram environment which makes it possible to create different blocks with each one representing a part of the system. Each of the blocks can contain a second (or third, …) level of structure consisting of blocks (Figure 4-1).

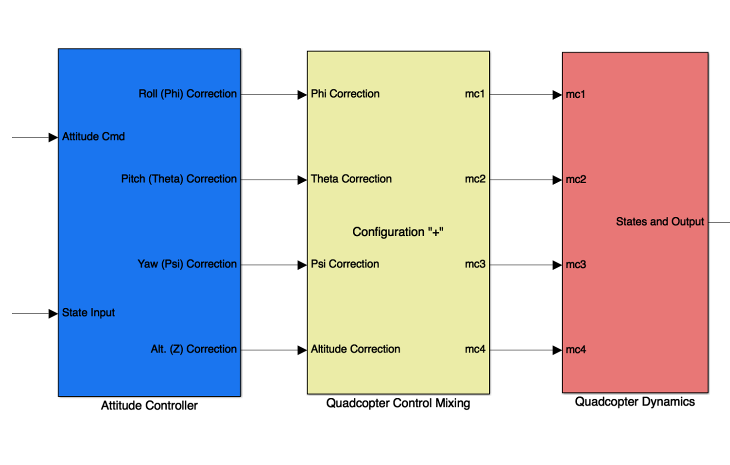


Figure 4-1 Simulink system consisting of different connected blocks

These second level blocks are used to represent the function of the outer block by using pre-defined or self-created functions to calculate the output of the first level block with respect to the corresponding input (Figure 4-2).

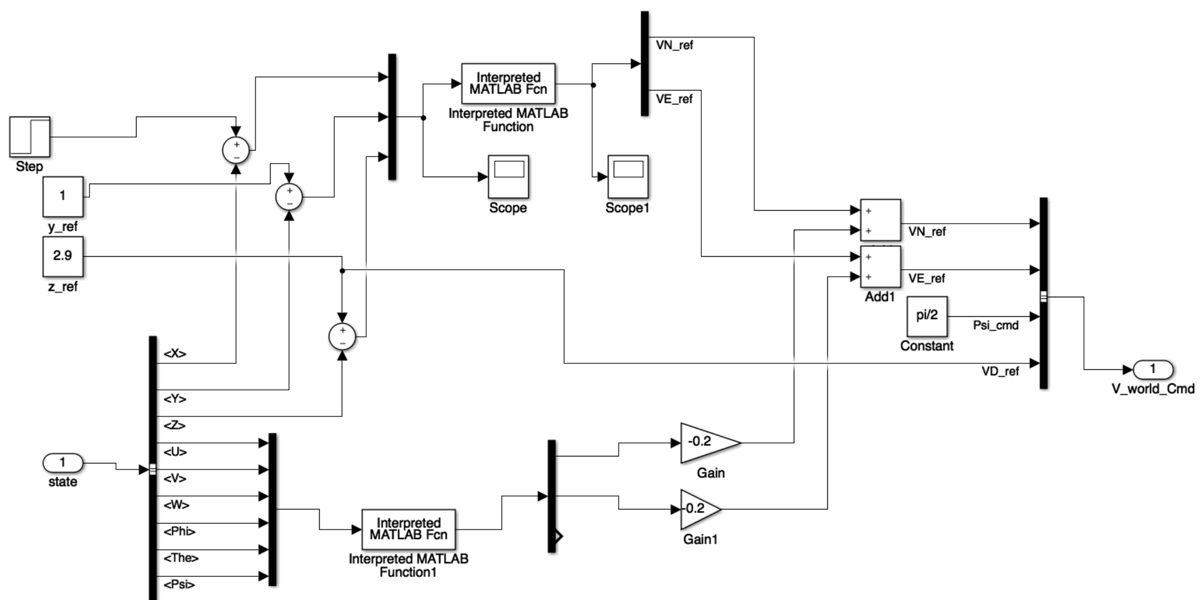


Figure 4-2 Calculation part of a block

To simulate a real system, it has to be depicted in the best possible way in these block diagrams by introducing all necessary variables as well as the calculations that link input and output of each stage. It is also possible to write programs using the programming language of Matlab and introduce that function into a block, e.g. if it is necessary to implement loops or if using pre-defined blocks would be more complicated.
Testing if the behavior of the simulation is similar to that of the real system is often hard, because as the system is as well translated to a simplified model while implying a new control at the same time may lead to a lot of errors. Therefore, it may be of interest to design a test procedure which ensures that the comportment is equal or at least with a high degree of similarity.

4.3. Quad-Sim

To simulate the system in this project, an open-source quadcopter simulator named "Quad-Sim" was used, which was developed in 2014 and can be downloaded via [25].

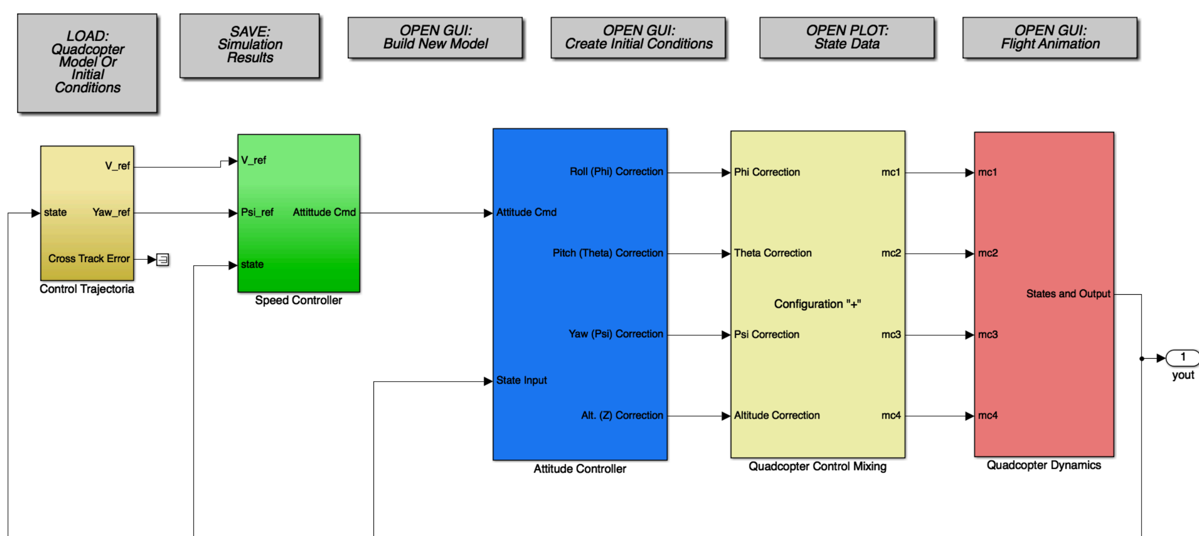


Figure 4-3 Quad-Sim simulator opened in Simulink

shows the block diagram of the simulator created with Simulink. It consists of various blocks that describe the system and its properties as well as the control of the quadcopter. In this simulator, the control consists of three stages:

- Position Control
- Velocity Control
- Attitude Control

which are implemented as follows.

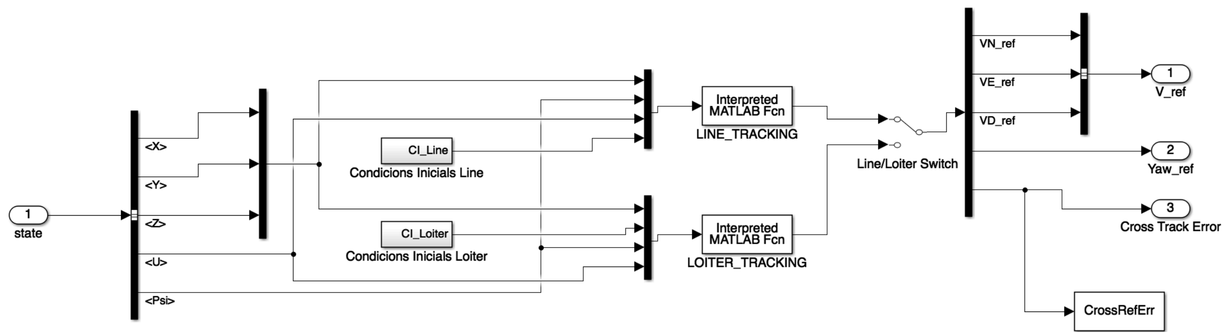


Figure 4-4 Position control block

Figure 4-4 above shows the block implemented for the position control of the vehicle. The input of this block is a bus called state that includes all the variables of the actual state of the drone that it receives from the last block as can be seen with the loop in . Then the needed values that are included in the bus are selected in the black bar, which is a so-called Bus Selector. The variables written in between "< >" under each line represent the value which is obtained by every single line leaving the selector. In this example, the current positions in X, Y and Z are obtained as well as the velocity in x-direction (the vector of the velocity being $\underline{V} = [U, V, W]^T$ as velocities in x, y and z, respectively) and the yaw angle Psi. Right next to the bus selection, some initial conditions are obtained by blocks which read variables that can be loaded via the "LOAD" button on the top left in . Functions programmed in Matlab then calculate the output values, in this case for a loiter and line condition. The output of this block are the set points for the velocities in North East Down (NED) coordinate system and a yaw angle.

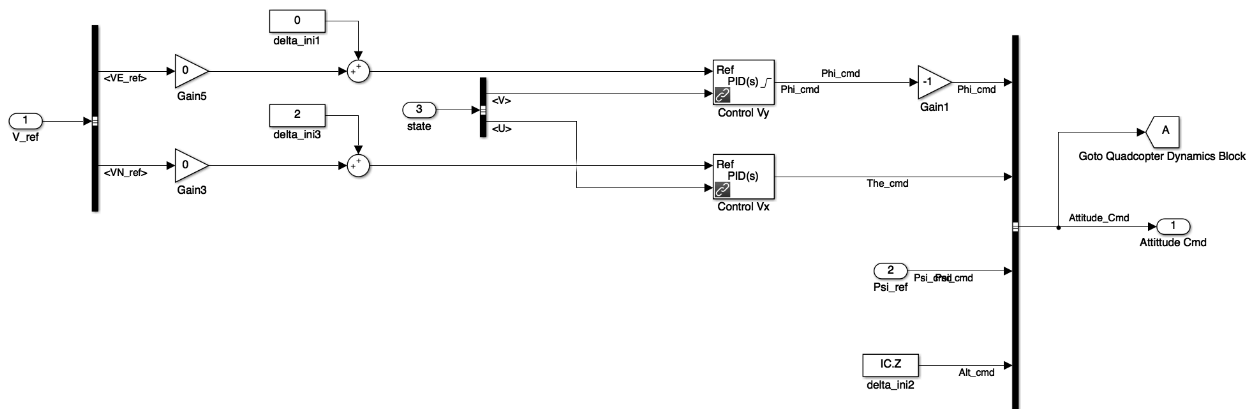


Figure 4-5 Velocity control block

For the velocity control level, the calculation works as indicated in Figure 4-5. It receives the velocities coming from the position control block which are then conditioned with a gain factor and a factor added.

The velocities from the current state are then obtained via a bus selector and both the velocities in x and y are then forwarded into a block that contains a PID controller, where the corresponding factors K_P, K_I and K_D can be chosen in the properties of the block very easily. The variables that are controlled with each of these controllers are the pitch and roll angle (Phi and Theta). Furthermore, the yaw angle Psi is forwarded from the former block and the altitude command from the initial conditions. The output of this block is then forwarded to the attitude command unit.

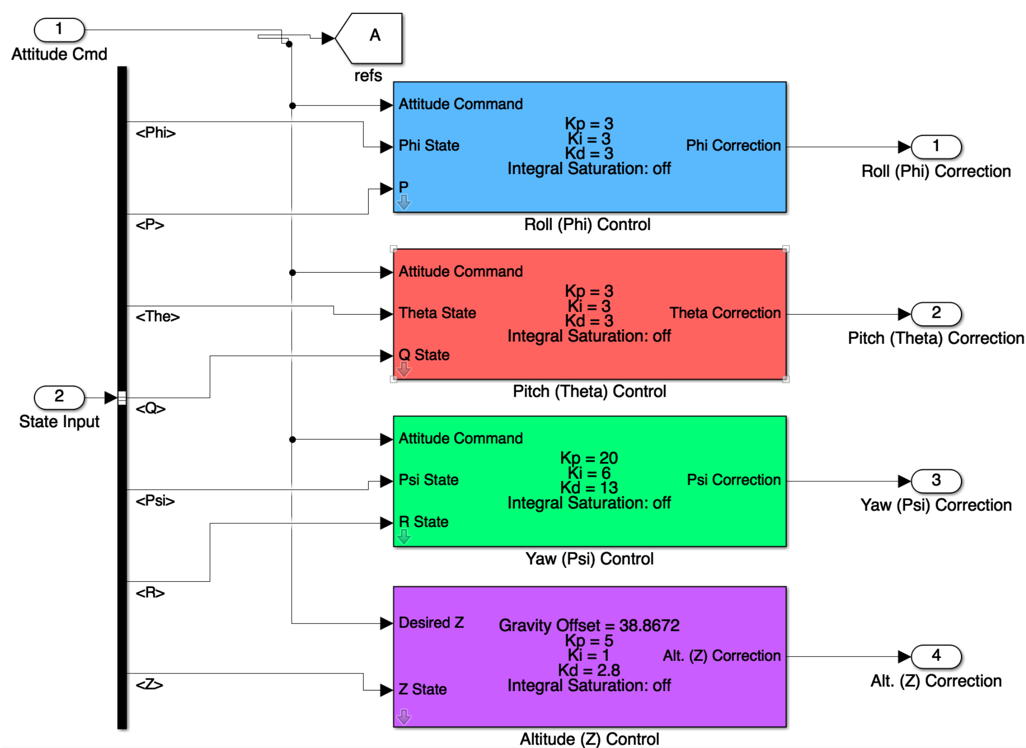


Figure 4-6 Attitude command block

Figure 4-6 shows, that this block contains 4 sublevel blocks with each of them being a controller for the Roll (Phi), Pitch (Theta) and Yaw (Psi) angles as well as for the altitude of the quadcopter. The constants for each term of the PID controller can be edited from this view but the blocks themselves have a different logical structure (Figure 4-7), meaning that there were no pre-defined PID blocks from Simulink used.

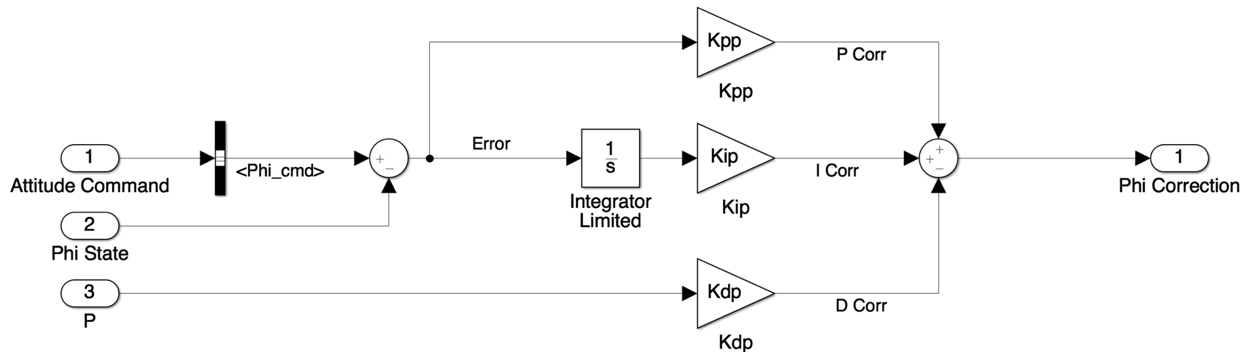


Figure 4-7 Structure of PID block in attitude control block

Because the task of this project is to implement an attitude controller and the system itself is very complex as there are many dependencies, it was decided to omit the velocity control part and the position control part to just focus on the attitude control. However, as in [9] already concluded, if only IMU measurements are used, the vehicle will start to drift away over time caused by little errors. Therefore, it is necessary to also control the position of the vehicle to keep it exactly in the desired attitude as well as position for example when hovering.

As the simulation should represent the system in a very realistic way, the model had to be changed in Simulink as well. Thus, only the velocity control part was omitted by simply deleting the whole green "Speed Control" block from . The resulting block structure can be seen in Figure 4-8 below.

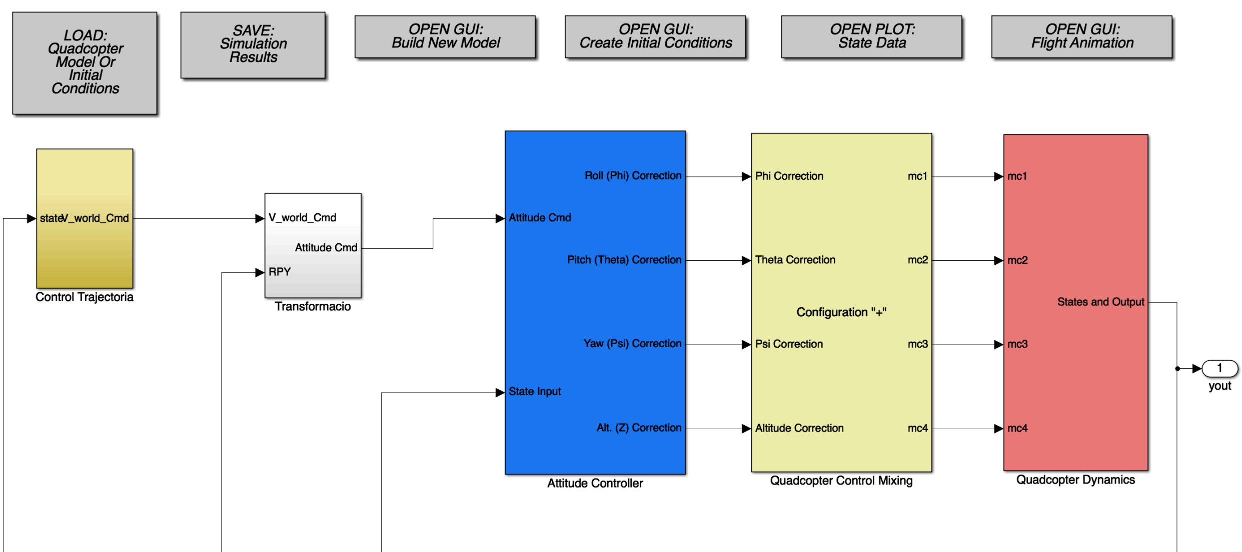


Figure 4-8 Changed Simulink structure

With the velocity control missing, the position control and its internal algorithm had to be changed as the wanted output leaving the position control block has to carry the signal entering the attitude control block and is therefore different from the above described version. Because the focus lies on the attitude control, the intention was to keep the position control algorithm as simple as possible. Nevertheless, as the controllers of the velocity part is absent, the behavior of the whole system changes significantly as the integrated PID blocks have disappeared.
Thus, the “effect” of those blocks on the control has to be compensated.

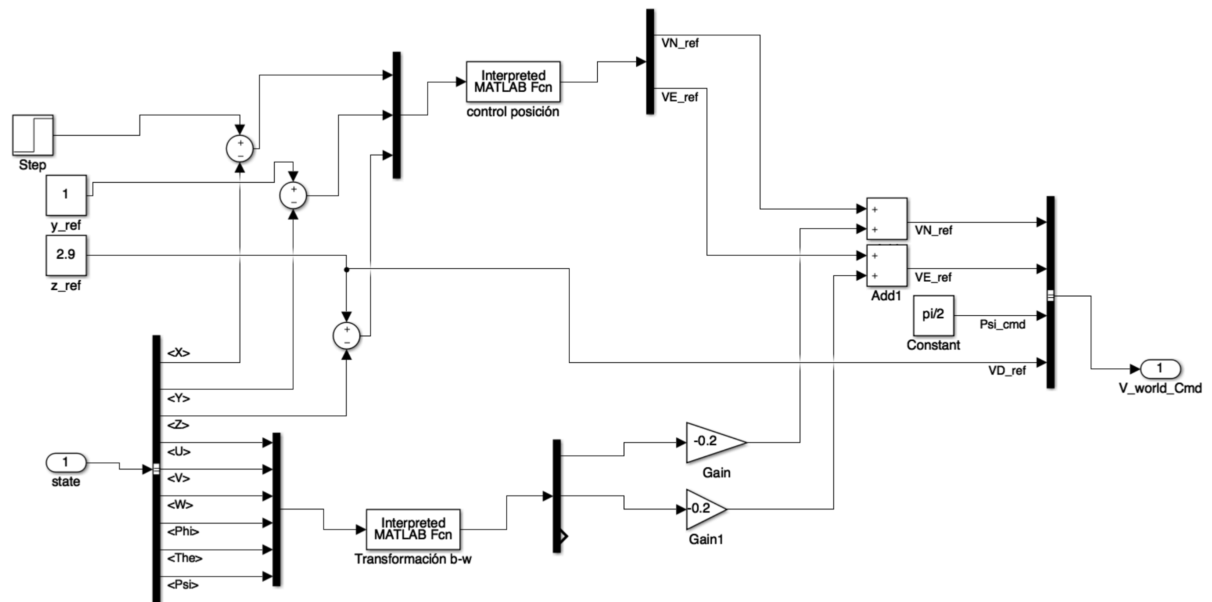


Figure 4-9 Position control block

Figure 4-9 above shows the adapted position control block. The state enters the block as unique input and is then distributed via a bus selector to its different variables. The variables of position are compared to the desired variables by being subtracted which results in the current error. The errors in x, y and z coordinates, respectively, are then multiplexed into one signal input signal for a Matlab function block. The function which is executed in this block is shown in Figure 4-10 below.

```
function [ out ] = control_posicion( in )
eN=in(1);
eE=in(2);
eD=in(3);

pitch=atan2(eD,sqrt(eN^2+eE^2));
yaw=atan2(eN,eE);

R=sqrt(eN^2+eE^2+eD^2);

vel=min(R*0.1,0.5);

VN_ref=vel*cos(pitch)*sin(yaw);
VE_ref=vel*cos(pitch)*cos(yaw);


out=[VN_ref VE_ref];

end
```

Figure 4-10 Matlab function for velocity calculation

The input of the function are the position errors calculated in the position control block as described above. They are called North, East and Down (NED) and are noted in the world reference system. The definition of errors in the NED direction can be seen in Figure 4-11 below. In this figure, the coordinate system marked with NED represents the desired position whereas the coordinate system with unmarked axes symbolizes the current position of the vehicle.

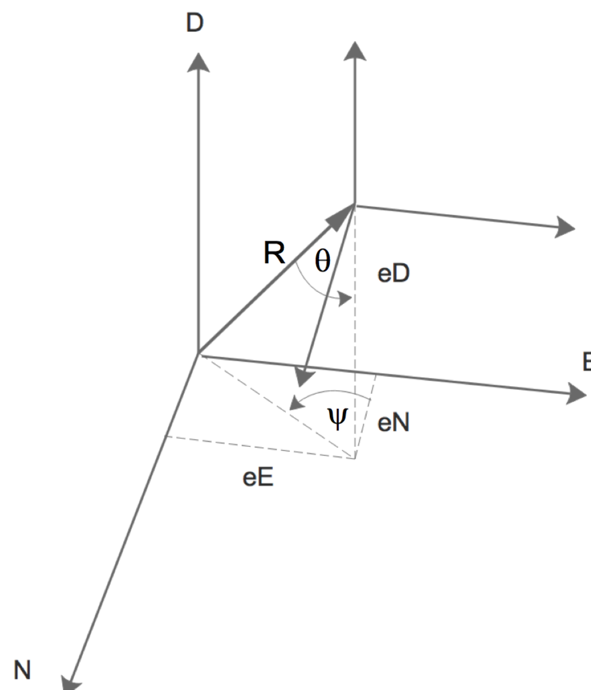


Figure 4-11 Calculation of velocity based on errors

The vector *R* connects the two systems' origins, being therefore the square root of the three errors squared. The angles *pitch* (θ) and *yaw* (ψ) are calculated as indicated in the drawing.

As can be seen in the code above, the outputs of the function are denoted as velocities as they are needed for further working of the model since the velocity control block has been omitted. However, the function shows a behavior similar to a PD-controller which is needed so that the system acts like before even without a "real" PD-controller implemented in a velocity control block.
The magnitude of the velocity is limited by using the minimum function of Matlab in the definition of the variable *vel* above which either multiplies the vector *R* by a constant factor 0.1 or in case this value is bigger than 0.5, uses 0.5 directly. The multiplication by a constant depending on the magnitude of the vector can be seen as proportional part as the velocity will be the higher the larger the distance between the desired position and the actual position is. The velocities are then projected in the axes N and E by simply multiplying them with the corresponding trigonometric functions.
The output only contains the velocities in North and East direction as the vertical velocity is calculated separately.

In the lower part of the diagram that can be seen in Figure 4-9 , there is another Matlab function used to transform the velocities from the current state of the vehicle which are noted in body reference system to the world frame by using a matrix and the angles roll, pitch and yaw like the transformation explained in chapter 3.2. The outputs of the function are the velocities U, V and W noted in the world coordinate system. They are then conditioned by a gain factor obtained in an empirical way. Adding the negative values to the desired velocities obtained from the above described function leads to the set values for the velocities. Because the focus is set on roll and pitch angle, the velocity for the z component is not calculated.
The set point values are then unified in a bus and forwarded to the next block in the diagram.

The following block for transformation contains the elements which can be seen in Figure 4-12.

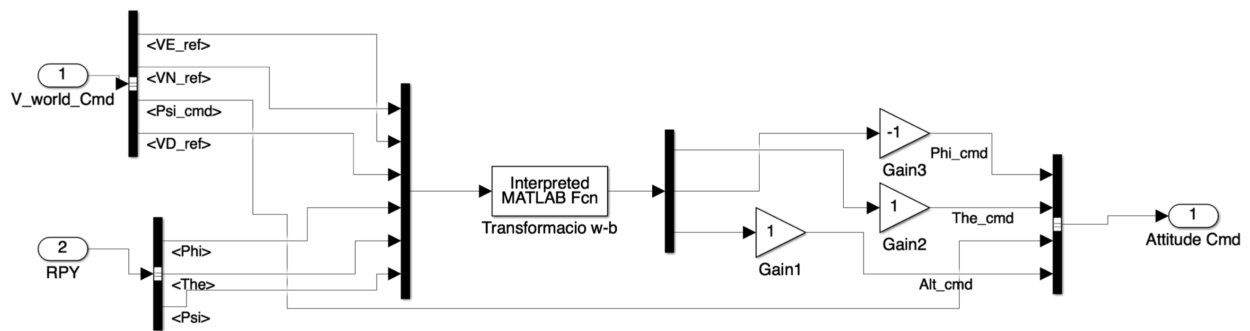


Figure 4-12 Transformation block

The input *V_world_cmd* comes directly from the position control block whereas the input *RPY* is a measurement of the vehicle and therefore contains the actual values of the state. All of the inputs are selected from a bus and then united with a multiplexer to simplify the input of the Matlab function. Thereafter, the function transforms the values to the body reference system using another time the rotational matrix presented in chapter 3.2. The outputs are the set points for the roll, pitch and yaw angles. Because of a different coordinate system used in the following control mixing block, the angle ϕ is defined in the wrong direction and therefore has to be multiplied by -1 to turn in the correct way.

5. Experiments

5.1. Experiments on the Simulator

Working with control systems always includes empirical methods which lead to full understanding of the system and its reaction to certain input. Therefore, there is no “golden rule” for tuning the controllers as they are highly dependent on the system and vice versa. As using the real physical model for first testing of a new control algorithm is mostly no option because the possibility of failure is high and the often expensive systems may get damaged. Thus, as already explained in chapter 4, simulators are a proper alternative for the first trial-and-error experiments as there will be no danger to objects.
Nevertheless, tuning of controllers can only be optimized to a certain level and the final refinements have to be done on the real model. This depends on the quality and accuracy of the model as its behavior may be a little different from the actual performance although the model should depict the reality as exact as possible with the above-mentioned limitations regarding the size of the model and its equations.

Due to changes in the simulator Quad-Sim made for this project, the PID controllers had to be configured differently to the downloaded version. Furthermore, the UAV model simulated is distinct as its mass and its inertia are unlike the model used in Quad-Sim. Thus, the first step was to configure the model so that it matches the Hummingbird. To do this, the developers implemented a function that can be opened by clicking the button *OPEN GUI: Build New Model* above the main block diagram, marked with a green square in Figure 5-1.

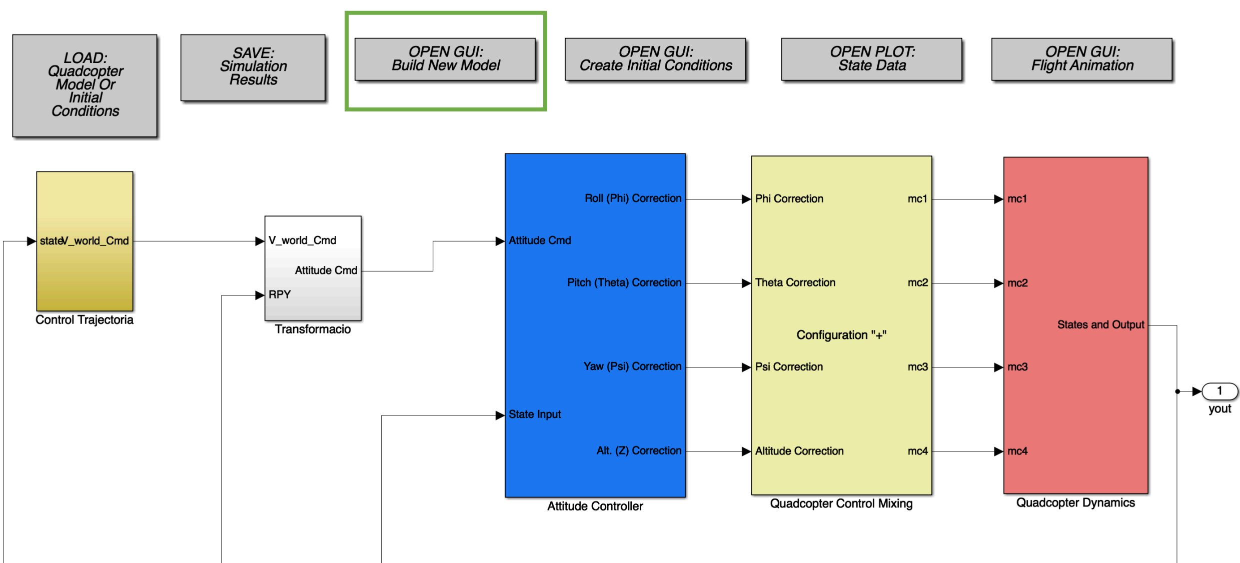


Figure 5-1 Button to open the GUI for building a new model

When clicking, the following GUI (Figure 5-2) opens where a whole new model of a quadcopter can be configured by entering all its mass and geometrical properties as well as data related to the motors.

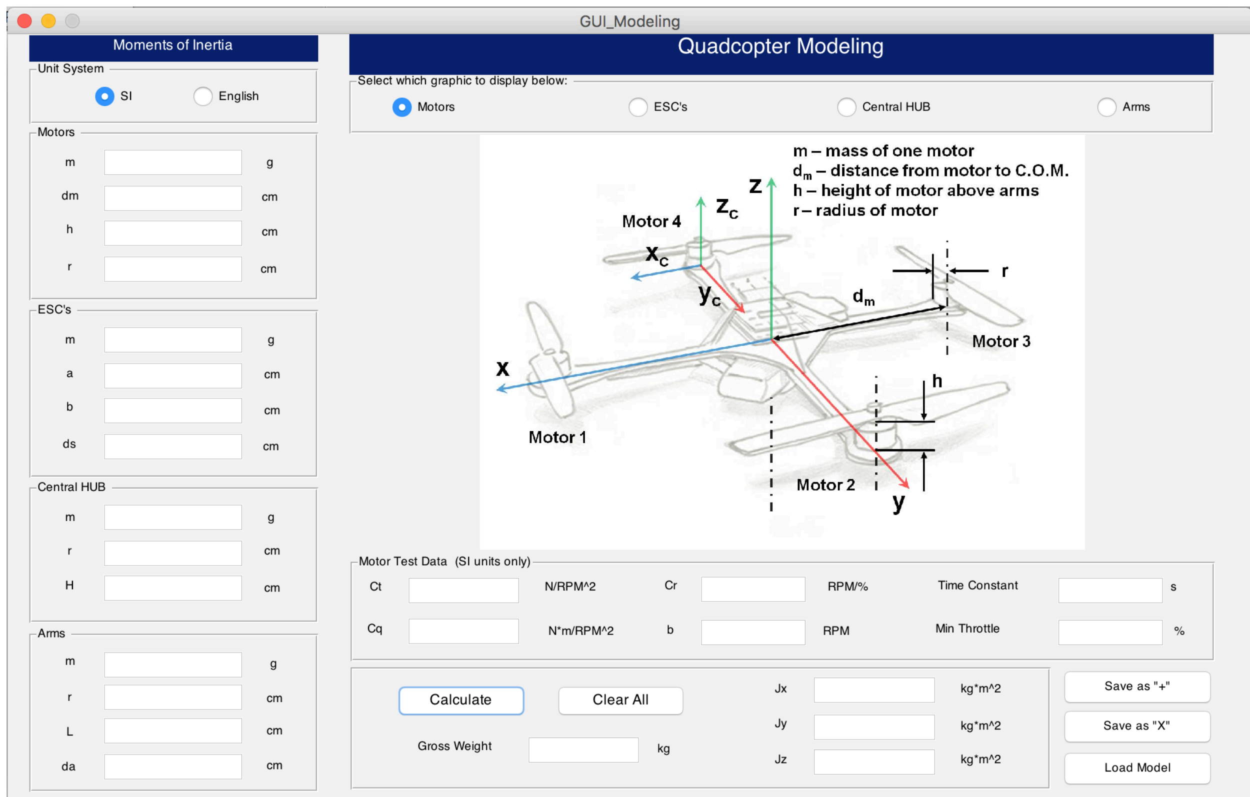


Figure 5-2 GUI to define a new quadcopter model in Quad-Sim

All of the data was obtained from the AscTec website and its documentation about the Hummingbird. [23] [26]

However, the gained data from the before-mentioned sources are not sufficient to generate a model with this tool as even more data are needed. For example, the used motors of the type X-BL-52S manufactured by the German company Hacker are especially designed for this UAV and therefore there is no datasheet publicly available.

Additionally, the modifications made to the vehicle by mounting odroid on top of it, removing the GPS module etc. have influence on the behavior and thus the real model of this project cannot be depicted by the data from Hummingbird in its default configuration anyhow as the modifications change, although maybe only minimally all the mass and geometrical properties.

For the above-mentioned reasons it was decided to use an existing model which was included in the used Quad-Sim package. The model *quadModel_+* is a model that is designed in the so-called "+" configuration as is the Hummingbird. Therefore their behavior will be similar.

The configurations of a UAV quadrotor can be in the so called “+“ or in the “x” configuration. The symbols describe the orientation of the coordinate system in the x-y plane of the body coordinate system with respect to the arms of the vehicle. For the “+”(“plus”) configuration, the axes are congruent with the arms. In contrast, in the “x” (“cross”) configuration the axes are 45° rotated to the arms being in the middle between two. Figure 5-3 below shows the difference between the two systems, where the green coordinate system is in the + configuration and the blue one in x configuration.

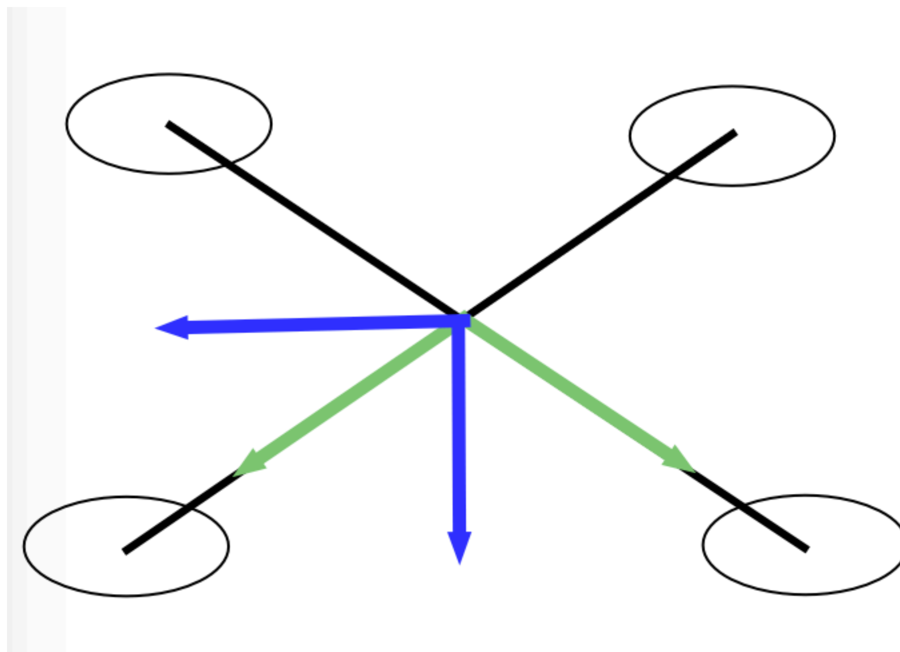


Figure 5-3 + and x configuration

Both of these configurations have their advantages and drawbacks which one has to take into account when designing a control for a quadcopter vehicle.
The cross configuration has more maneuverability as the turning around the x and y axes for generating roll and pitch, respectively are generated by operation of all 4 rotors and not only two in the case of plus configuration as the vehicle is in the latter case only turning around the arms and therefore the two other motors can maintain their speed if only one angle has to be changed. On the other hand, this is one of the great advantages of the plus configuration as it is easier to realize control algorithms because the acting of the motors is more intuitive without thinking about trigonometrical functions.
From a physical point of view, the distance from the rotation axis to the force which is in this case the motor is the whole arm length in plus configuration whereas it is the arm length multiplied by sin(45°) and therefore nearly 30% shorter for cross configuration. This leads to higher possible angular acceleration.

When using a camera mounted on the vehicle, the field of view of the camera is free when flying "forward" along the x- or y-axis. Of course the last point mentioned applies only to UAVs flown by persons because automatic control can realize the mixing for the motors easier than a person with a control channel in each direction.

After identifying the model, the parameters for the PID controllers had to be changed in order to get a behavior that fulfills the expectations. It has to be stated that the controllers used in this simulator are no typical PID controllers as can be seen through their implementation in Figure 5-4 as there were no pre-defined PID blocks used but rather self-defined blocks created.

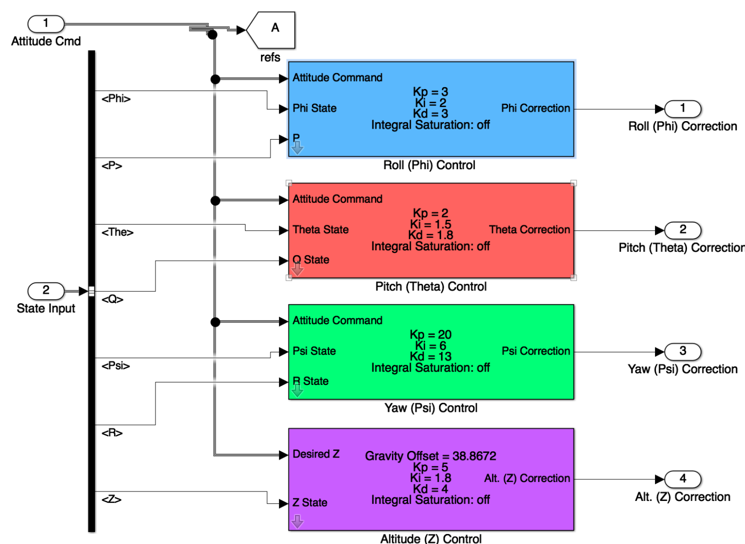


Figure 5-4 Self-created PID blocks

The figure above shows what the attitude control block contains.
Each of the three angle controllers use the following scheme of calculation as indicated in Figure 5-5 below.

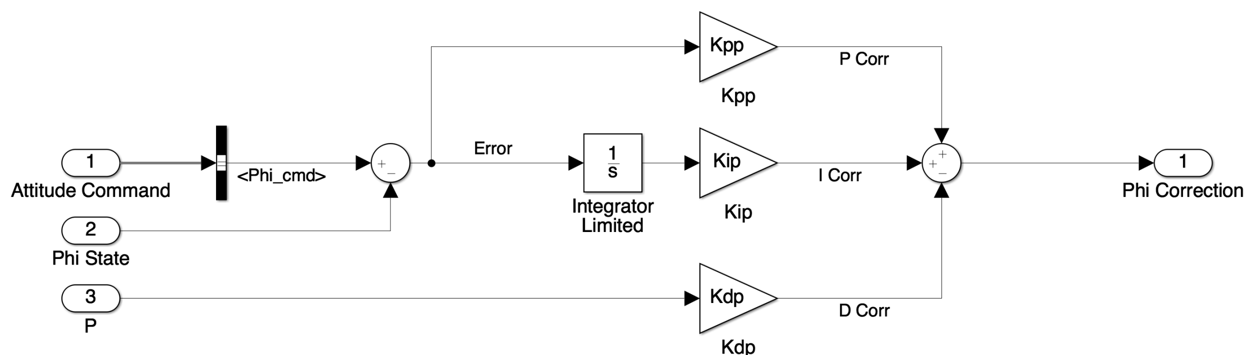


Figure 5-5 Structure of PID controllers

The inputs to this block are the desired angle, the actual angle as well as the actual angular velocity with the former mentioned obtained from the transformation block and both the latter mentioned received from the measurements of the vehicle.
The controller then calculates the error and forwards it to an integrator as well as a proportional part with their respective constants K_{pp} and K_{ip} as factors. As the D part of a PID controller needs the control variable to be derived, another way is chosen as the derivation is already measured as the angular velocity P. Therefore, it can be used directly with the corresponding constant K_{dp}. The parts are then added and forwarded to the output. The behavior of this controller could be described as PI + D controller.

All of the controllers for the three angles roll, pitch and yaw have the same structure. Nevertheless, the values for optimal behavior may be different from one controller to another.
In contrast, the altitude controller differs in its structure. On the one hand it uses a derivator for the state of the z variable. On the other hand there is some difference in the reaction of the system to controlling whether the vehicle has to climb or descend as in descent the gravity has to be considered so that the decline may not be too rapid to lead to an unstable flying condition. These two changes can be seen in Figure 5-6 .

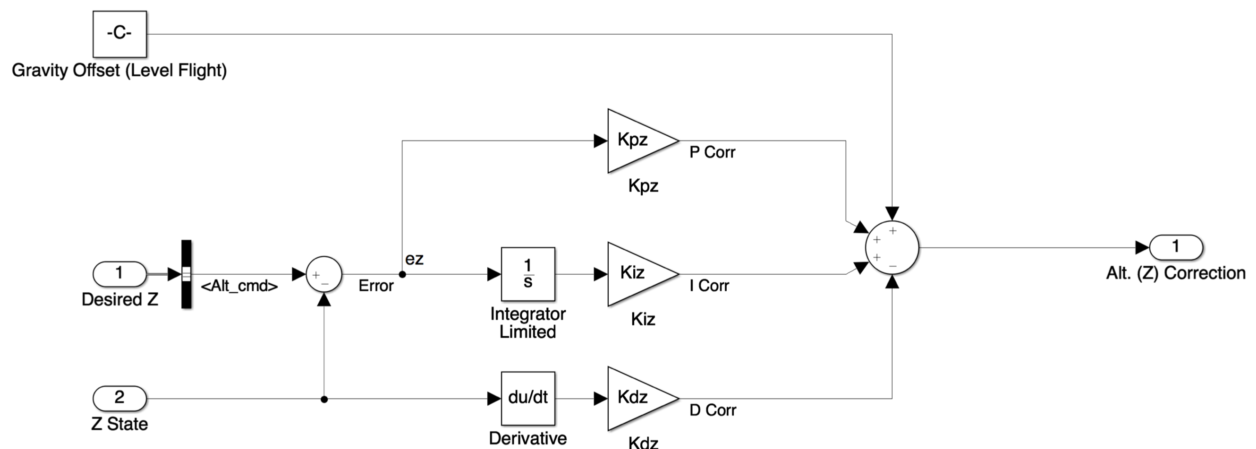


Figure 5-6 Altitude controller

The Quad-Sim simulator also uses some initial conditions to describe the state of the vehicle when the simulation started. Thus, all state variables of the UAV can be set to an initial value.

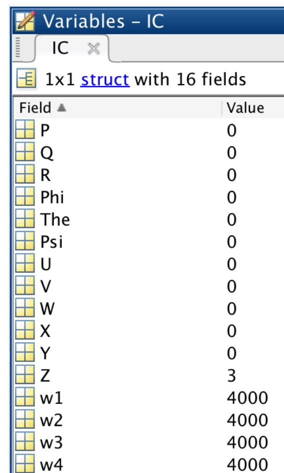


Figure 5-7 MATLAB struct with all the initial conditions

Figure 5-7 above shows the corresponding struct in MATLAB which contains all the initial conditions used for this simulation. All values are set to zero, except the z value to a height if 3 meters and the rotor angular velocity for every rotor (w1…w4) to 4000 rpm. This creates a hovering condition for the vehicle when starting the simulation.

Simulink offers different solvers for its simulation which can be configured in detail. Following solving method and affiliated properties are chosen and can be seen in Figure 5-8.

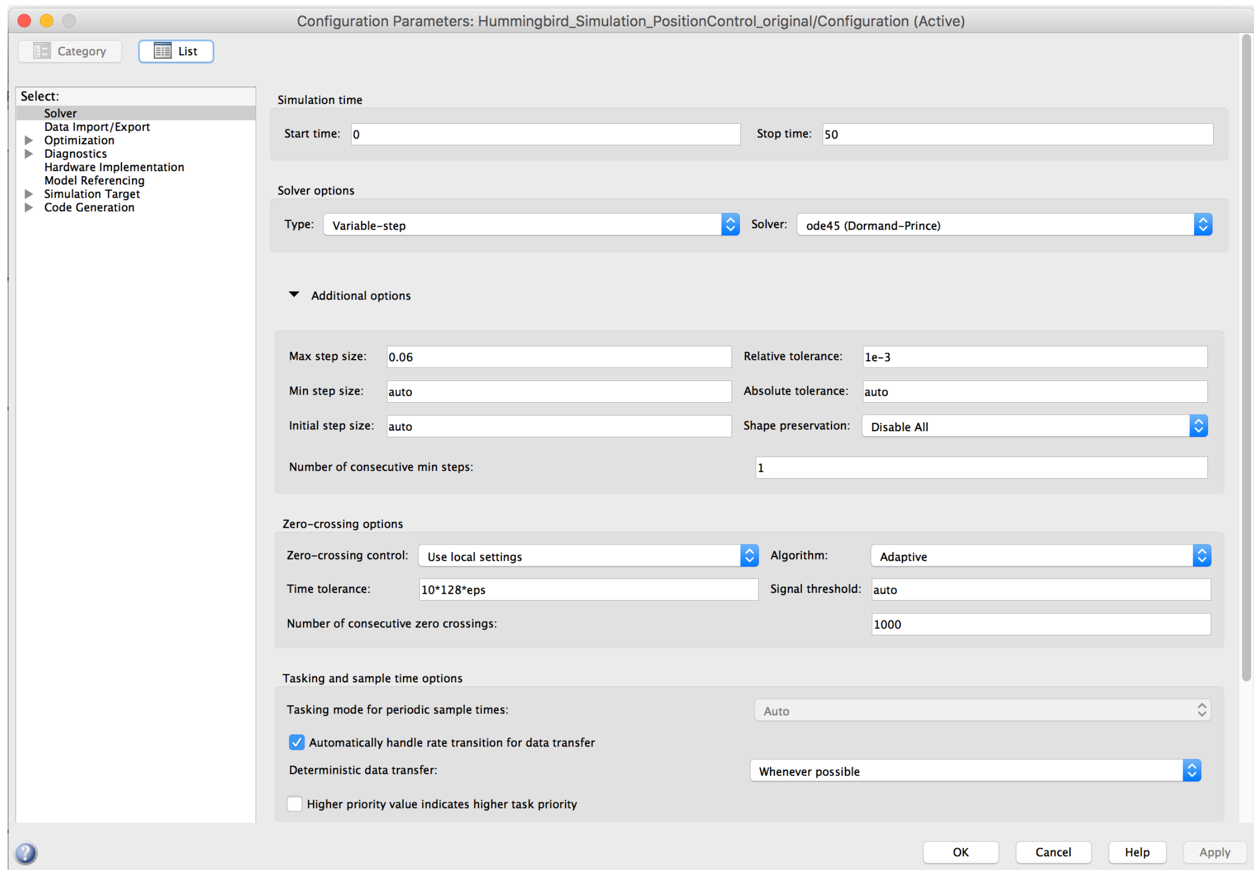


Figure 5-8 Solver properties for Simulink model

The simulation duration of 50 seconds was chosen in order to see if any instabilities can be found in the solution later on which may be not seen in the first 10 or 20 seconds although the vehicle has reached its set points before.

The set points are chosen in the position control block as can be seen in Figure 4-9 above. First only one of the position variables x, y and z is changed and the other two remain the same as the it is easier to see the effects of changing parameters. As there are already 3 parameters for each PID controller to adapt and the system is highly dependent on each of the other states, changes and their result will not be easy to understand.

First tests were made by keeping the z value constant at 3 meters, the y value at 0 and using a ramp with slope 1 as input to the x value.

The simulator offers the possibility to watch graphs of the three axes with their corresponding position, velocity as well as angle and angular velocity around the matching axis by clicking the button “OPEN PLOT: State Data” in the main panel which can be seen in Figure 4-8. Furthermore this button opens another figure displaying the 4 rotor speeds.

In addition to the graphical representation for the PID controller as can be seen in Figure 5-5, it can also be described by its equation. However, it has to be taken into account, that the controllers used in this project are, with the exception of the altitude controller, no pure PID controllers but rather PI+D controllers. This is due to the fact, that the derivative part of the controller is not computed but the derivative of the variable, which is already available as measurement, is used.
The equation of a PID controller would look as can be seen in equation 7.3 in [27]

$$u(t) = K_P \left[e(t) + \frac{1}{T_I} \int_0^t e(\tau) d\tau + T_D \frac{d}{dt} e(t) \right] \tag{5.1}$$

where
u(t)…manipulated variable
K_P…proportional amplification factor
T_I…reset time (German-speaking area: T_N)
T_D…derivative action time (T_V)

Because of the above-mentioned changes to a pure PID controller, this equation can be written as

$$u(t) = K_P \left[e(t) + \frac{1}{T_I} \int_0^t e(\tau) d\tau + T_D \dot{y}(t) \right] \tag{5.2}$$

with
$\dot{y}(t)$…derivative of the controlled variable

Because the blocks in the simulation use K-factors instead of time, the former equation (5.2) could also be written using the following relations

$$K_I = \frac{K_P}{T_I} \tag{5.3}$$

$$K_D = K_P T_D \tag{5.4}$$

as

$$u(t) = K_P e(t) + K_I \int_0^t e(\tau) d\tau + K_D \dot{y}(t) \tag{5.5}$$

with the Laplace transformation, the transfer function is written as

$$G_R(s) = K_P + \frac{K_I}{s} + K_D s \tag{5.6}$$

5.2. Empirical controller tuning

As no earlier experience was made with this model regarding the tuning of parameters, a first approach to get the parameters was to use a trial-and-error method based on knowledge about the general behavior of PID controllers and its different parts P, I and D. The values chosen for the first test were the ones used in Quad-Sim (Table 5.1) because of the lack of experience about which values to pick. The names are according to the representation of each factor using following scheme:

K_{xy}

with
x representing the part of the PID controller (p, i or d)
y representing the controlled variable (p=ϕ, t=θ, s=ψ and z)

K_{pp}	K_{ip}	K_{dp}	K_{pt}	K_{it}	K_{dt}	K_{ps}	K_{is}	K_{ds}	K_{pz}	K_{iz}	K_{dz}
2.8	3	1.1	1.3	1	1.1	20	6	13	5	1	2.8

Table 5.1 First set of PID factors

The following results shown in below are obtained from this first test.

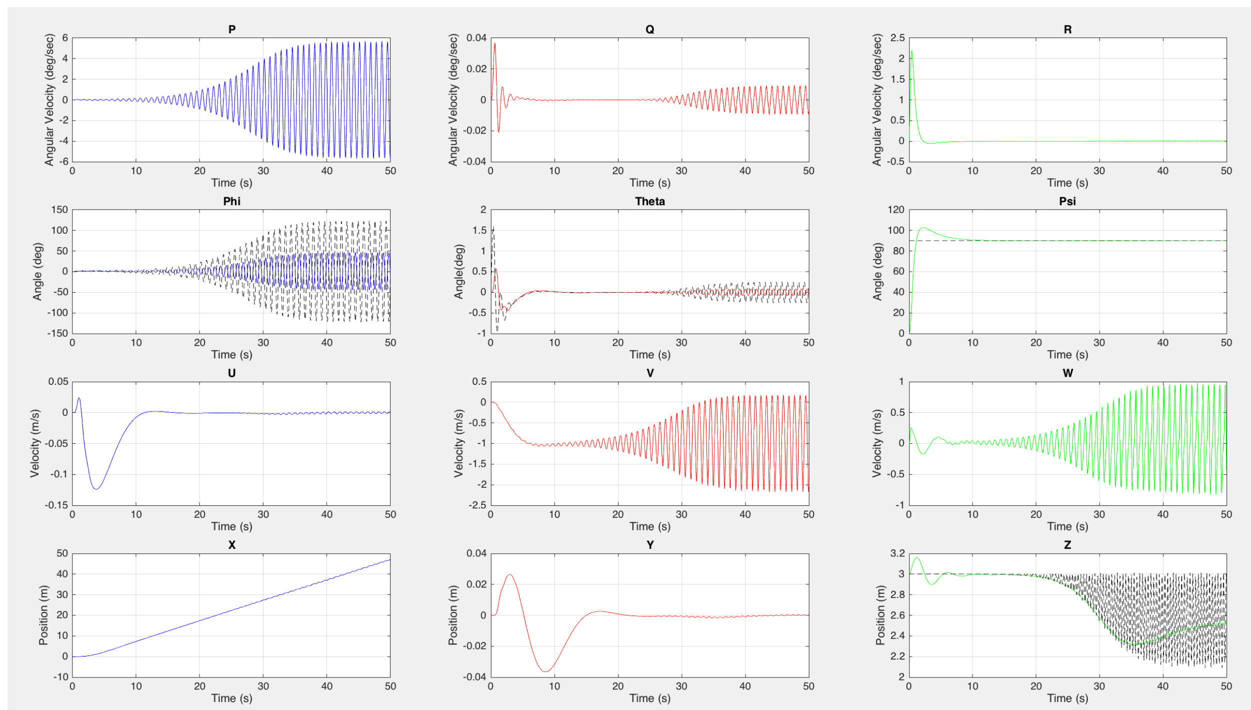


Figure 5-9 MATLAB plots of all state values with ramp slope 1 in x

The grey dashed line in every plot represents the value of the set point whereas the blue, red and green lines are displaying the actual values obtained from the model. Although the x plot follows the slope, it can be seen that the system becomes unstable and is oscillating with angles of ϕ up to 50° which would lead to a crash when applied to the real system.

Furthermore, the graphs of y and z show, that the change in x also affects the other two variables and they fluctuate around their initial values. The z value later on oscillates more and the y value is then quite stable.
By slightly increasing the D-part to $K_{dp} = 2$, the oscillations can be regulated to a level that is acceptable. Then the ramp was changed for a step signal from 0 to 1 acting with a delay of 10 seconds to test the behavior with this more challenging input to demonstrate the stability of controlling. The result can be seen in Figure 5-10 below.

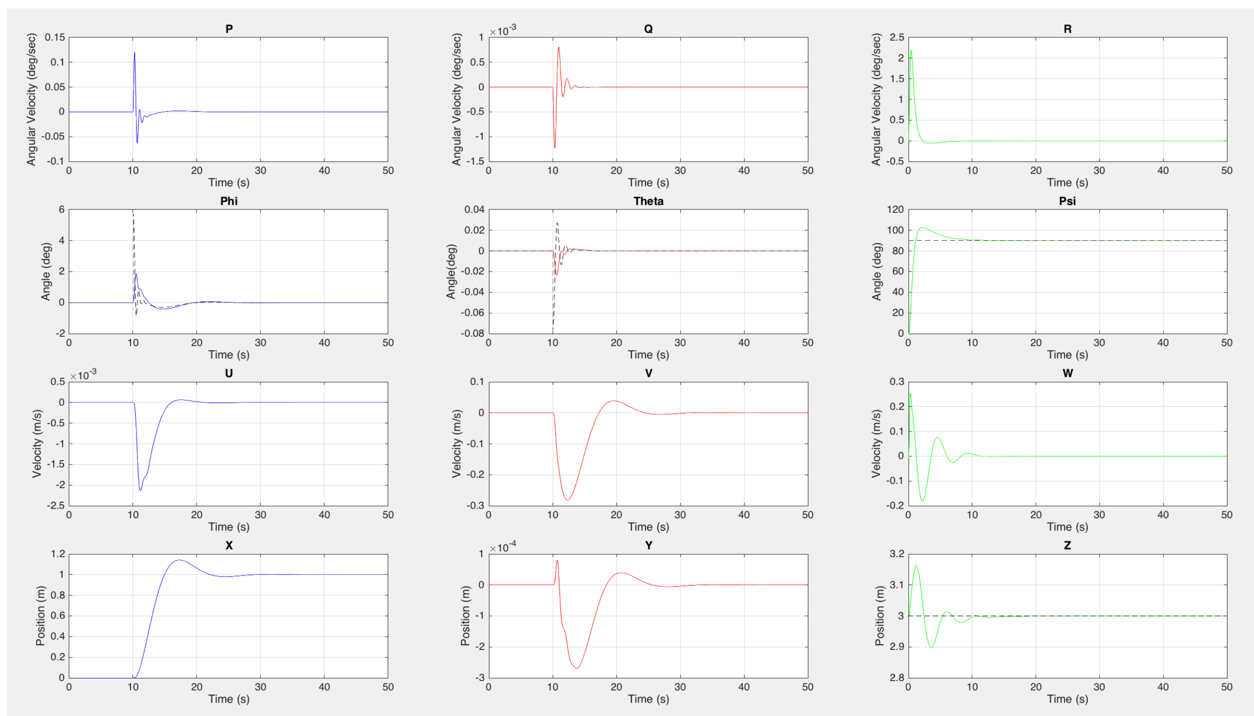


Figure 5-10 Output graphs with step input signal

The x graph shows that the signal follows the input pretty good and reaches the desired value after about 5 seconds but then tends to overshoot. Slight oscillations as for p are acceptable as the peak is at about 1/10 °/sec and therefore very small.

Thereafter, the step signal was set as input for the y set point and the factors for the θ controller were set to the exact same values as the factors for the ϕ controller as the behavior should be the same which was confirmed by the same experiment as above obtaining the same results but this time with the graphs of the first and second column exchanged.
For the z component, the behavior is different because of the gravitational forces as explained above. When applying a step signal to the z set point, the output showed oscillation that could be reduced by increasing as above the D-factor to $K_{dz} = 4$.

In summary, the following values seen in Table 5.2 were obtained by optimizing for a step signal at only one of the inputs not taking into account possible interactions.

K_{pp}	K_{ip}	K_{dp}	K_{pt}	K_{it}	K_{dt}	K_{ps}	K_{is}	K_{ds}	K_{pz}	K_{iz}	K_{dz}
2	1.5	2	2	1.5	2	20	6	13	5	1	4

Table 5.2 First optimization results

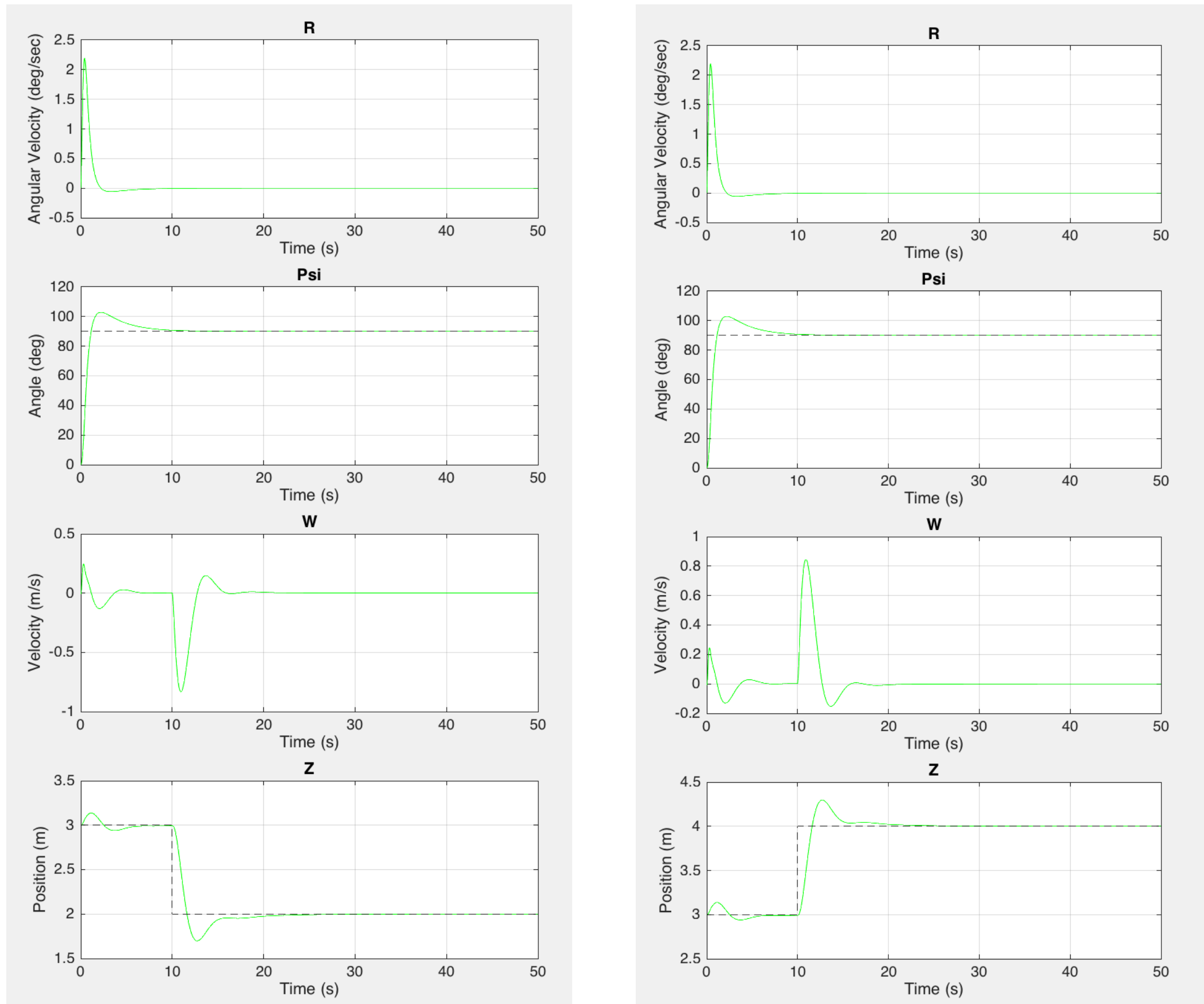


Figure 5-11 Results optimization of z axis PID controller

Figure 5-11 above shows the graphs for the z-axis only where the input signal was a step signal from 3m to 2m starting at 10 seconds with the results in the left column whereas the plots in the right column show the effect of a step signal from 3m to 4m also starting at 10 seconds. Because of the different controller and the compensation of gravity it was necessary to find out how the vehicle reacts to such an input in both ascending and descending direction. It can be stated that the behavior is very similar and also satisfactory with only slight overshooting in both directions. Figure 5-11 Results optimization of z axis PID controller

After optimizing every single controller for step signals just on one channel, the task was to test the performance of the vehicle when input signals requiring changes on every of the axes enter the system. Due to the complexity of the system UAV, the interaction between the different controllers had to be found. This was easier after optimizing every single controller as dominating many variables and seeing the effects on such a system is very difficult. Furthermore, finding satisfying properties is always a procedure of trial-and-error although there are helpful thumb-rules for changing the factors according to a certain behavior: Nevertheless, every system is special in its reactions to particular changes.

The first test was made by altering the position in every three axes with respect to the initial conditions of [0, 0, 3]. A step signal required the vehicle to move 1m in the x direction where the input acts with a delay of 10 seconds. The y set point was directly set to 1m with a constant as well as the z set point which commanded the quadcopter to descend to 2.9m. With the above obtained values for the factors of the PID controller remaining the same, the following result was found (Figure 5-12).

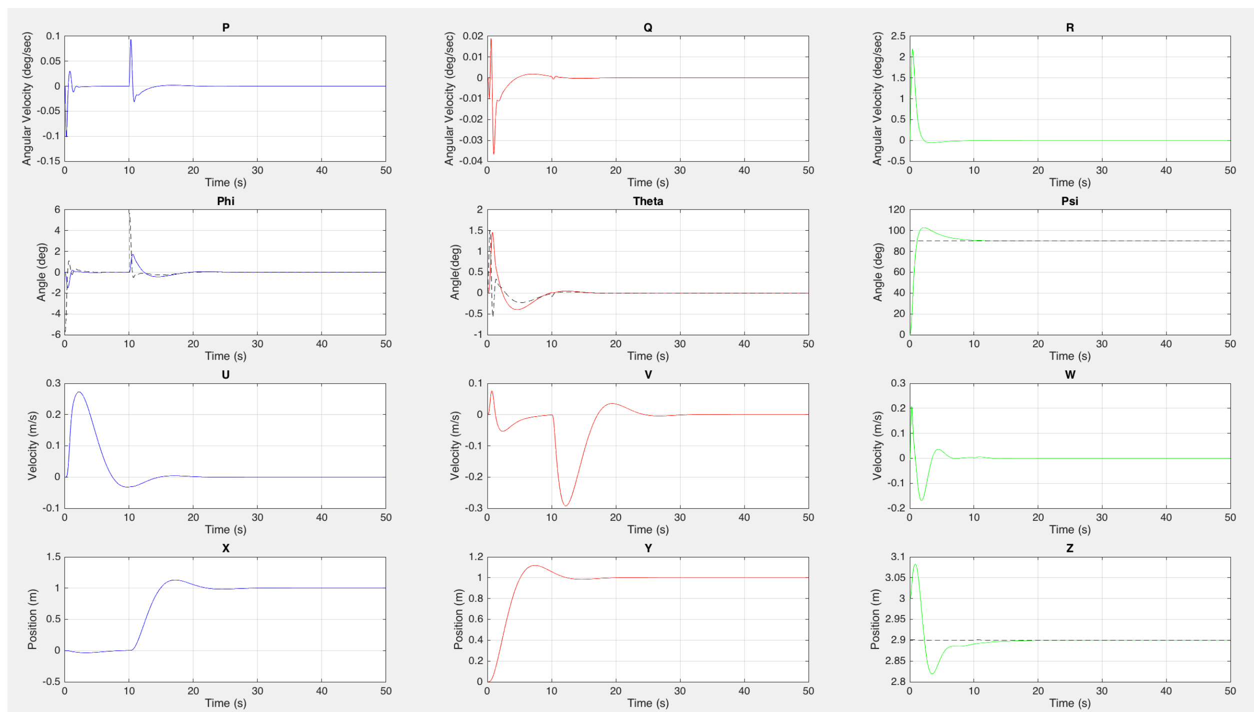


Figure 5-12 Simulink result for inputs to every axis

The plots show that the set points in every direction are reached without oscillating and without remaining error. Furthermore, the desired values are reached in nearly 5 seconds after the input in the plane axes x and y and in about 2 seconds in the height axis z. But it can also be stated that there is overshoot in every axis with peak values which deviate up to 10% from the desired distance to be traveled. The required velocities and angular velocities are in an acceptable range for such deviations as they barely reach 0.3 m/s or 0.04 °/s, respectively.

5.3. Numerical optimization for tuning

The above determined values can be taken used as good starting values for first testing operations on the vehicle, as it would behave in a predictable way and no major dysfunction because of wrong parameters should happen. However, due to the complex system and its interactions one can never find the optimal parameters of the controllers in an empirical way as there impacts on the behavior because of the coupled system where they might not be expected or the changes made to the values may be too little or to big as to find the optimal position. Furthermore, experimenting would in any case take a lot of time and an optimal result is not guaranteed by no means.

For this reasons, it was decided to improve the parameters obtained using numerical methods. As the model was generated in Simulink, a tool from the calculating software Matlab, the easiest way to get optimization was to use an already implemented algorithm of Matlab because it can be connected with the simulator and optimize the values right in the model. As a result, the effects can be seen immediately on the graphs of the simulator as seen above.

Before starting the optimizing procedure it was necessary to define a criteria for which an extremum can be found in order to get optimal values for the controller parameters.

As it is a control loop, the control error was the value of choice for this problem, albeit squared and summed up over the whole time period of the simulation as a quality function for the difference between command variable and controlled variable.

To find the optimal value of this quality function, an optimizer already implemented in Matlab was chosen. The solver *fmincon* is designed to find a minimum in an iterative process and gives the possibility to introduce constraints to the parameters to be optimized. Furthermore, there are a lot of properties to define as well as options to choose when using this optimizer which can all be found in the Matlab documentation.

The implementation of the optimization in Matlab can be seen in Figure 5-13 below.

```
function Optimizer
tic;
clear Param_Value J;
startSimulator;
Param_Start=[2;1.5;2;2;1.5;2;5;1;2.8];
lb=[0;0;0;0;0;0;0;0;0];%lower boundary
ub=[5;5;5;5;5;5;5;5;5];%upper boundary

options = optimoptions('fmincon','MaxIter',5,'Display','iter-detailed',...
    'DiffMinChange',0.01,'MaxFunEval',100);
func=@(Param_Value)fun(Param_Value);
[Param_Optim,fval]= fmincon(func,Param_Start,[],[],[],[],lb,ub,[],options);
Param_Optim
fval
assignin('base','Param_Optim',Param_Optim);
save Param_Optim;
plotting(Param_Optim);
toc;
end
function J = fun(Param_Value)
sim_obj=sdo.SimulationTest('Hummingbird_Simulation_PositionControl_original');
assignin('base','Param_Value',Param_Value);
sim_obj.LoggingInfo.LoggingMode='LogAllAsSpecifiedInModel';
sim_out=sim(sim_obj);
Y_x=sim_out.LoggedData.get('logsout').getElement('X').Values;
Y_y=sim_out.LoggedData.get('logsout').getElement('Y').Values;
Y_z=sim_out.LoggedData.get('logsout').getElement('Z').Values;
W_x=sim_out.LoggedData.get('logsout').getElement('X_cmd').Values;
W_y=sim_out.LoggedData.get('logsout').getElement('Y_cmd').Values;
W_z=sim_out.LoggedData.get('logsout').getElement('Z_cmd').Values;
W=[W_x.get('Data') W_y.get('Data') W_z.get('Data')];
Y=[Y_x.get('Data') Y_y.get('Data') Y_z.get('Data')];
length_W=length(W);
E=[(W(1:length_W,1)-Y(1:length_W,1)).^2 (W(1:length_W,2)-Y(1:length_W,2)).^2 (W(1:length_W,3)↙
-Y(1:length_W,3)).^2 ];
J=sum(sum(E));
end
function plotting(Param_Optim)
Param_Value=Param_Optim;
sim_obj=sdo.SimulationTest('Hummingbird_Simulation_PositionControl_original');
sim_obj.LoggingInfo.LoggingMode='LogAllAsSpecifiedInModel';
sim_out=sim(sim_obj);
Y_x=sim_out.LoggedData.get('logsout').getElement('X').Values;
Y_y=sim_out.LoggedData.get('logsout').getElement('Y').Values;
Y_z=sim_out.LoggedData.get('logsout').getElement('Z').Values;
W_x=sim_out.LoggedData.get('logsout').getElement('X_cmd').Values;
W_y=sim_out.LoggedData.get('logsout').getElement('Y_cmd').Values;
W_z=sim_out.LoggedData.get('logsout').getElement('Z_cmd').Values;
W=[W_x.get('Data') W_y.get('Data') W_z.get('Data')];
Y=[Y_x.get('Data') Y_y.get('Data') Y_z.get('Data')];
length_W=length(W);
time=W_x.time;
hold on
plot(time,W(1:length_W,1),time,Y(1:length_W,1));
figure
plot(time,W(1:length_W,2),time,Y(1:length_W,2));
figure
plot(time,W(1:length_W,3),time,Y(1:length_W,3));
end
```

Figure 5-13 Optimize function

The function *Optimizer* is the main function where the simulator is started with the command *startSimulator* which executes another program starting Simulink and loading the model and initial conditions. Thereafter the values for executing *fmincon* are implemented, namely the start value and upper and lower boundaries as maximum values the modulated parameters can take on. Starting values for the first run of the program were the controller parameters obtained in the empirical process. The lower boundary is set to 0 for every value because PID controller gains cannot take on negative values. The upper boundaries were set to 5 for every value as the highest value was 5 before. In case that one value would be optimized near 5, these upper boundaries should be raised. The options for the optimizer may be chosen to specify certain properties for *fmincon*. In this case it was necessary to set the value *DiffMinChange* to 0.01 because before this, the optimizer kept the original parameters and did not work. The *MaxIter* option limits the maximum number of iterations done by the optimizer and *MaxFunEval* does the same for the number of function evaluations during each iteration. *Display* shows the result after each iteration in the command window. After that, the function to be optimized has to be defined. It is the anonymous function *func* which is dependent on the function *fun*.
The latter mentioned function is evaluated during each iteration. It creates a *SimulationTest* object which is run by the command *sim*. All outputs of this simulation is logged in the struct *sim_out*. As the simulation is logging in timeseries data format, the data has to be converted into matrices of type double. Thereafter, the squared error is calculated and summed up in the quality function J which is the output of the function *fun*. The output of this function is the value that *fmincon* tries to minimize and the input of the function will be varied (*Param_Value* in this case).
The third function is just called for plotting the controlled variable and the command variable for the optimized parameters.

The first run of the program led to the parameters which can be seen in Table 5.3. The parameters of the controller for the yaw angle are not considered in this optimization as the controllers for pitch, roll and z are much more important for the stabilization of the vehicle.

K_{pp}	K_{ip}	K_{dp}	K_{pt}	K_{it}	K_{dt}	K_{pz}	K_{iz}	K_{dz}
2.3015	2.1527	0.9715	1.6895	1.1582	2.5027	4.1886	0.8031	3.1032

Table 5.3 First numerical optimization result

The corresponding graphs are shown in Figure 5-14, Figure 5-15 and Figure 5-16 below.

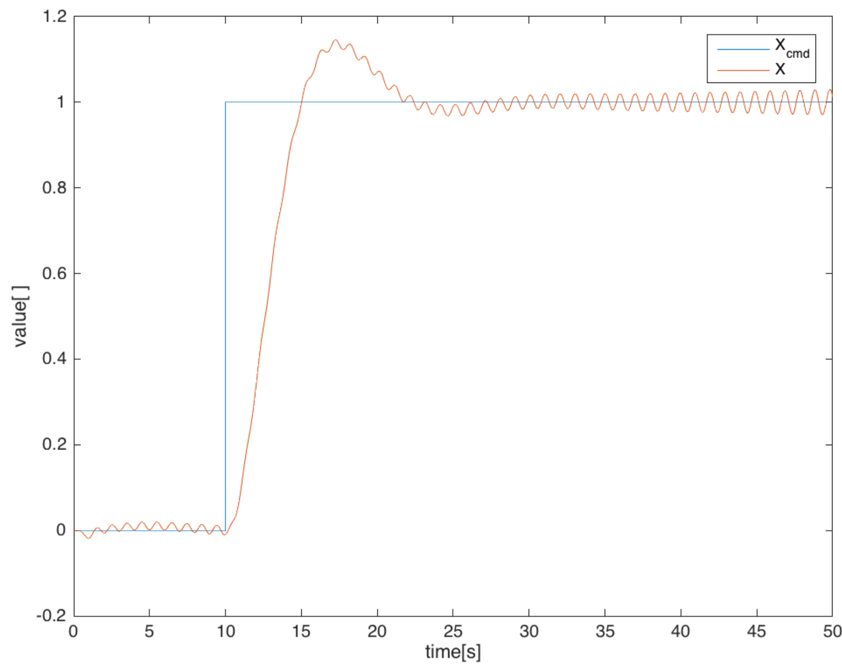


Figure 5-14 x-axis first optimization

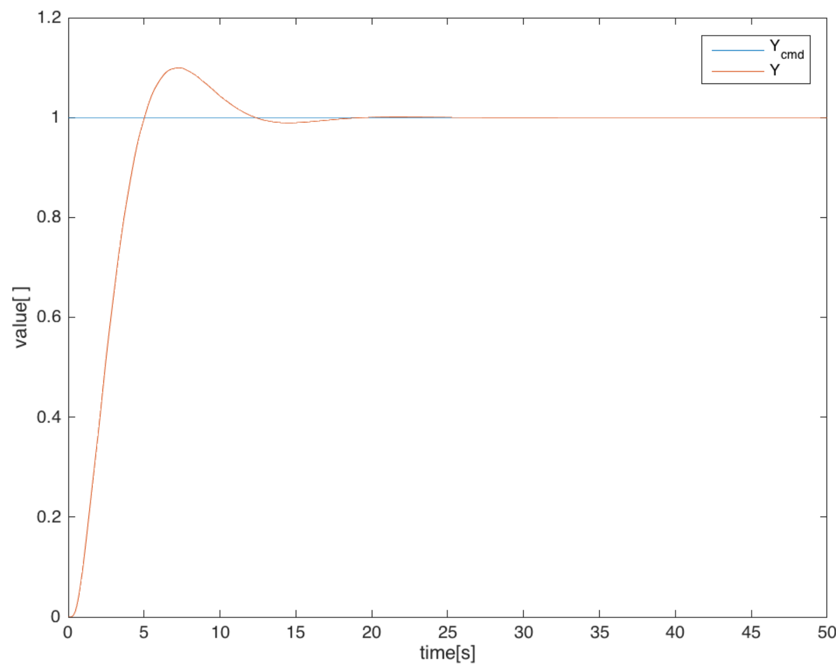


Figure 5-15 y-axis first optimization

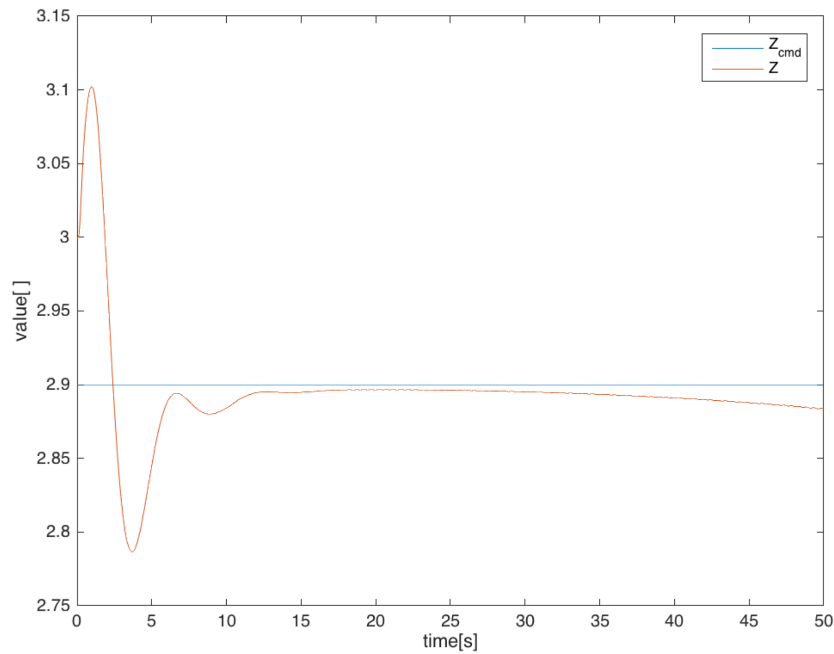


Figure 5-16 z-axis first optimization

It can be seen, that the controller for the y-axis is already stable. But the two others show unwanted behavior. The x-axis controller oscillates with an increasing amplitude, whereas the z-axis controller deviates from the command variable over time.

A first approach to tackle this problems was to simulate for a longer period of time, as the deviations would then be even higher causing the optimizer to react differently because the quality function J would also increase in its value. The simulation time was raised to 200 seconds in a second attempt. Furthermore, the result of the first optimization was chosen as start parameter for the second optimization.

The resulting parameters for the second optimization are shown in Table 5.4.

K_{pp}	K_{ip}	K_{dp}	K_{pt}	K_{it}	K_{dt}	K_{pz}	K_{iz}	K_{dz}
2.1866	3.6072	1.1188	1.6739	1.1403	2.5284	4.1918	0.7995	3.1107

Table 5.4 Second numerical optimization result

As can be seen easily, the values for the parameters of controller 2 and 3 (pitch and z) did not change much compared with the first result. But there are noticeable changes in K_{ip} and K_{dp}. This is can also be seen in the graphs in Figure 5-18, Figure 5-17 and Figure 5-19, where the controller 3 is already stable and does not deviate any more, whereas the controller 1 is still oscillating, albeit decreasing over time.

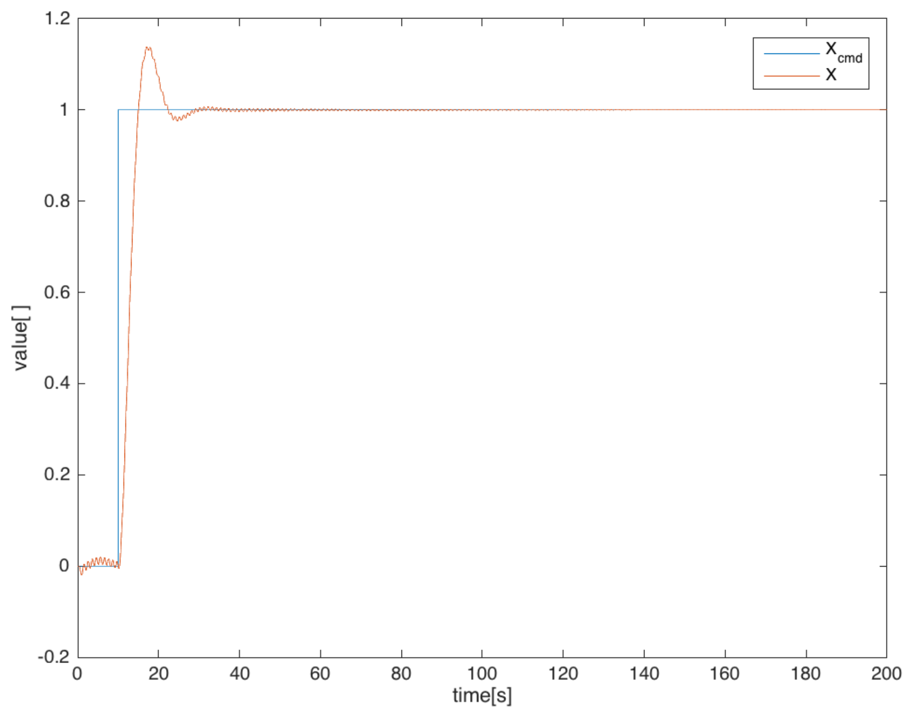


Figure 5-18 x-axis second optimization

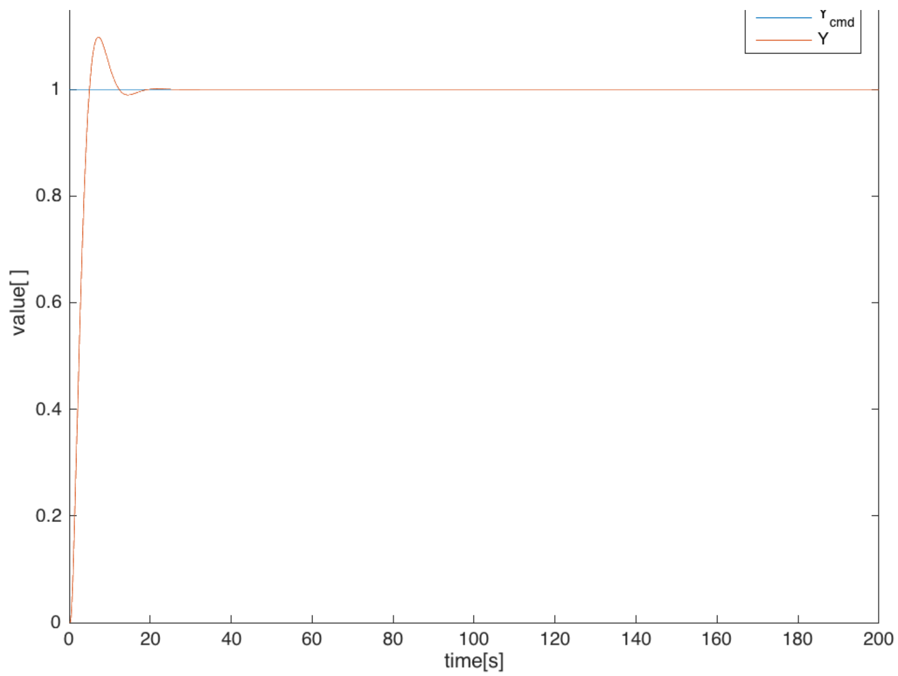


Figure 5-17 y-axis second optimization

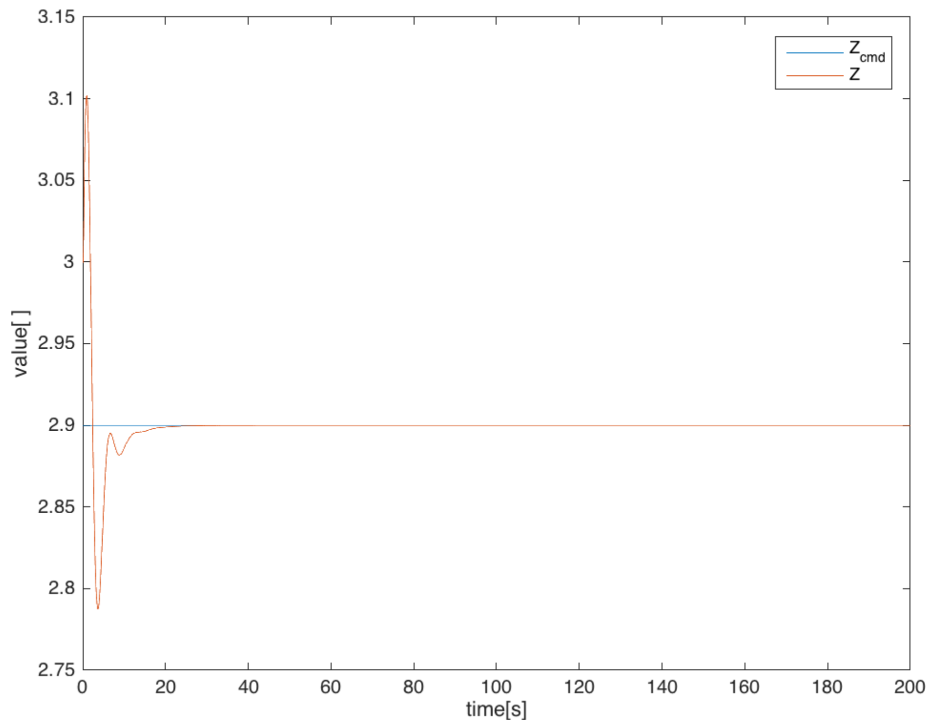


Figure 5-19 z-axis second optimization

As can be seen in the plots above, all of the controllers are stable with controller 1 controlling the low-frequent oscillation well with only slight overshoot but struggling to damp a system part oscillating at a high frequency. However, stability is given, as the error tends to 0 over time.

To tackle this problem, another optimization round was run, although for only 50 seconds as in the first round because there are no significant changes between 50 and 200 seconds and computation is significantly more expensive. As the parameters of controller 2 and 3 are already optimized, they were not changed any more in the last experiment which also saves computation time. The optimization process can also be improved by adding a term to the quality function. The first derivative of the error is calculated, squared and summed up as can be seen in the additional code lines in Figure 5-20.

```
E_dot=[(W_dot(1:length_W-1,1)-Y_dot(1:length_W-1,1)).^2,
    (W_dot(1:length_W-1,2)-Y_dot(1:length_W-1,2)).^2,
    (W_dot(1:length_W-1,3)-Y_dot(1:length_W-1,3)).^2 ];
E=[(W(1:length_W,1)-Y(1:length_W,1)).^2,
    (W(1:length_W,2)-Y(1:length_W,2)).^2,
    (W(1:length_W,3)-Y(1:length_W,3)).^2 ];
J=sum(sum(E))+sum(sum(E_dot));
```

Figure 5-20 Additional code lines to calculate first derivative of error

Nevertheless, this brings only minor improvements in the result concerning the first controller. In another attempt for enhancing the performance of the roll controller, the boundaries were set near the value which was the result of the empirical trial-and-error method since there hasn't been any such oscillation. The result can be seen in Figure 5-21 below. The plot called X_1 represents the solution found with boundaries set to:

$$lb_1 = [0; 0; 0; 1.6739; 1.1403; 2.5284; 4.1918; 0.7995; 3.1107]$$

$$ub_1 = [5; 5; 5; 1.6739; 1.1403; 2.5284; 4.1918; 0.7995; 3.1107]$$

Whereas the boundaries for the result X_2 were as follows:

$$lb_2 = [0; 1.2; 1.3; 1.6739; 1.1403; 2.5284; 4.1918; 0.7995; 3.1107]$$

$$ub_2 = [5; 1.8; 2.2; 1.6739; 1.1403; 2.5284; 4.1918; 0.7995; 3.1107]$$

The values for the boundaries were set from the empirical obtained parameters restraining the range of possible outcome around their values.

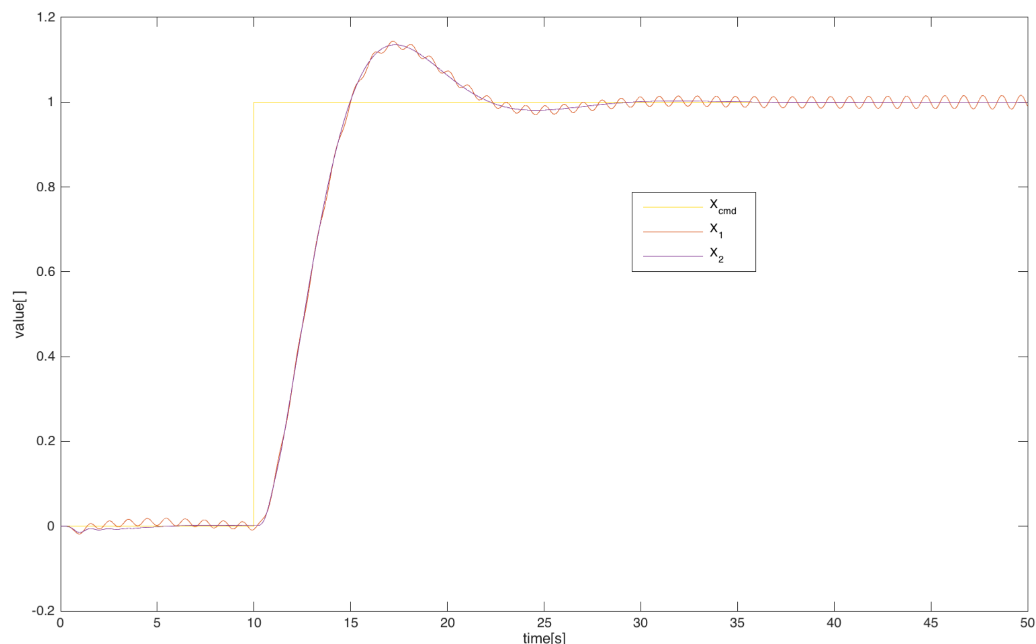


Figure 5-21 Comparison of optimization results

This setting of the boundaries may lead the optimizer to other values as they might be closer to a local minimum or eventually the global minimum. In the other case, when the boundaries are set in a wide range, the optimizer might find another local minimum and this minimum results in the oscillations as seen above.

Although the oscillations do not have high amplitudes and disappear over time (Figure 5-18) , they could be problematic when applying the parameters to the real systems as the model is not totally identical and small oscillations in the model could lead to an unstable behavior of the vehicle. Therefore, it is desirable that there are no oscillations at all in the model.

5.4. Programming

The system of control generated for the simulator had then to be applied on the real system to establish the desired control of the UAV. Of course, the representation in Simulink is somewhat abstract with its block diagram. For example, there are different blocks for the stages of control like position and attitude control block are separated. However, in the real system there are only two systems of a control: the controller itself and the controlled system, which is in this case the Hummingbird quadrotor. Therefore, the whole controlling part in the simulator had to be converted into the control program and therefore written in executable code, whereas the system with its properties weren't touched. ROS nodes allow executables written in two different programming languages to be run: C++ and Python. In this work it was decided to write the code in C++.

Like in the simulator, the following tasks had to be realized in the program:

- Obtaining current orientation and position from the quadcopter LLP
- Implementing PID controllers for each axis
- Sending commands to control the quadcopter

Sending and receiving of data with ROS works with so-called publishers and subscribers. As mentioned above, the former publishes data on a certain topic without knowing whether or not other node(s) is(are) listening. This behavior is therefore distinguishable from many other communication in computers as typically sender are receiver are known and directly coupled. But with ROS any part in the system could be a subscriber and thus receive the sent information. Therefore, the desired program needs a subscriber to the topic on which the Quadcopter node publishes the current position and orientation data as well as a publisher that sends the commands to the vehicle at the end of the program to move or turn the quadcopter to the set point.

The software package delivers a program to test if the control of the Hummingbird via ROS works. The program *ctrl_test* is designed to call the service to turn the motors on and off as well as to send commands to move the quadcopter to a desired point. One has to take into account that by default the commands are only sent for 1 second and thereafter the vehicle will no longer receive commands so it returns to its initial state determined by the positions of the sticks of the radio control before the execution of the program. Therefore it is recommended to do these tests with the vehicle in a fixed position, e.g. on a workbench.

If the request to turn the motors on is sent, they run in idle mode unless the left stick on the radio control is not in the lowest position.

Thereafter, it is possible to send commands to navigate the Hummingbird by entering the corresponding coordinates in the command line in the terminal of Linux.
It was decided to take the program *ctrl_test* (Annex II)as template as it already consists of two of the main parts, namely the subscription and publishing of data.

5.4.1. Data acquisition

The Hummingbird publishes and subscribes to different topics and also offers the above-mentioned service to start the motors. Figure 5-22 shows all the topics with a short description. All these are included in the *asctec_hl_interface* package that communicates with the HLP of the Asctec Autopilot mounted on the Hummingbird quadrotor. [28]

5.1.1 Subscribed Topics

`fcu/control` (asctec_hl_comm/mav_ctrl)
listens to control commands on this topic

`fcu/pose` (geometry_msgs/PoseWithCovarianceStamped)
listens on this topic for pose updates

`fcu/state` (asctec_hl_comm/mav_state)
listens on this topic for state updates. This bypasses the state estimation on the HLP

`fcu/ekf_state_in` (sensor_fusion_comm/ExtEkf)
Listens for updates from ethzasl_sensor_fusion

5.1.2 Published Topics

`fcu/imu_custom` (asctec_hl_comm/mav_imu)
custom imu package without covariance, with height and differential height

`fcu/imu` (sensor_msgs/Imu)
Imu sensor data

`fcu/gps` (sensor_msgs/NavSatFix)

`fcu/rcdata` (asctec_hl_comm/mav_rcdata)
values of the RC channels receiver by the AutoPilot

`fcu/status` (asctec_hl_comm/mav_status)
status of the helicopter

`fcu/debug` (asctec_hl_comm/DoubleArrayStamped)
debug values from the SSDK

`fcu/current_pose` (geometry_msgs/PoseStamped)
publishes the pose that is currently used for position control on the HLP

`fcu/ekf_state_out` (sensor_fusion_comm/ExtEkf)
publishes the current prediction for ethzasl_sensor_fusion

5.1.3 Services

`fcu/motor_control` (asctec_hl_comm/mav_ctrl_motors)
service to switch the motors on/off. **Use this carefully, it also works during flight!**.

Of the above possible topics that the Hummingbird publishes, the topic *fcu/imu* was chosen. This topic delivers measurements from the Inertial Measurement Unit (IMU) where the type of the message is *sensor_msgs*, a self-defined data format. The definition can be seen in Figure 5-23.

File: **sensor_msgs/Imu.msg**

Raw Message Definition

```
# This is a message to hold data from an IMU (Inertial Measurement Unit)
#
# Accelerations should be in m/s^2 (not in g's), and rotational velocity should be in rad/sec
#
# If the covariance of the measurement is known, it should be filled in (if all you know is the
# variance of each measurement, e.g. from the datasheet, just put those along the diagonal)
# A covariance matrix of all zeros will be interpreted as "covariance unknown", and to use the
# data a covariance will have to be assumed or gotten from some other source
#
# If you have no estimate for one of the data elements (e.g. your IMU doesn't produce an orientation
# estimate), please set element 0 of the associated covariance matrix to -1
# If you are interpreting this message, please check for a value of -1 in the first element of each
# covariance matrix, and disregard the associated estimate.

Header header

geometry_msgs/Quaternion orientation
float64[9] orientation_covariance # Row major about x, y, z axes

geometry_msgs/Vector3 angular_velocity
float64[9] angular_velocity_covariance # Row major about x, y, z axes

geometry_msgs/Vector3 linear_acceleration
float64[9] linear_acceleration_covariance # Row major x, y z
```

Figure 5-23 Message definition of the topic fcu/imu [28]

This file defines the messages that are sent on the topic. The message *Imu.msg* consists of 6 different measurements, namely the orientation, the angular velocity and the linear acceleration with their corresponding covariance. The data types can be seen before the variable names. For example, the second line after the header shows the definition of *orientation_covariance* which is the variable name with the data type *float64[9]*. The other variables are defined with data types typically used in ROS. The package *geometry_msgs* provides different message types to describe common geometric primitives as points, vectors or poses. This package also includes the message types Vector3 and Quaternion

Figure 5-22 Published topics, subscribed topics and services of asctec_hl_interface [27]

which are used in *Imu.msg*. The Vector3 format is a simple column vector with the three values of x, y and z. Whereas the Quaternion message offers the four values x, y, z and w which are described as follows. [29] The definition and explanation of quaternions can be found in Annex I.

With the formula for quaternions in mind, the four values a, b, c and d can be obtained and these are transmitted as a message via the topic *fcu/imu.*

However, for further computations and the PID controllers, it is easier to use the values of the angles of attitude, namely roll, pitch and yaw in the program. Therefore, a function was created to convert the values x, y, z and w to the angles ϕ, θ and ψ. The formulas that can be seen in Figure 5-24 were developed using the template given in [30].

```
// roll (x-axis rotation)
    double sinr = +2.0 * (q_w * q_x + q_y * q_z);
    double cosr = +1.0 - 2.0 * (q_x * q_x + q_y * q_y);
    roll = atan2(sinr, cosr);

    // pitch (y-axis rotation)
    double sinp = +2.0 * (q_w * q_y - q_z * q_x);
    if (fabs(sinp) >= 1)
        pitch = copysign(M_PI / 2, sinp); // use 90 degrees if out of range
    else
        pitch = asin(sinp);

    // yaw (z-axis rotation)
    double siny = +2.0 * (q_w * q_z + q_x * q_y);
    double cosy = +1.0 - 2.0 * (q_y * q_y + q_z * q_z);
    yaw = atan2(siny, cosy);
```

Figure 5-24 Conversion from quaternions to Euler angles in C++

The values q_w, q_x , q_y and q_z represent the quaternions' four values and are obtained using a ROS subscriber with the corresponding callback function where the conversion is also integrated. The necessary code lines used to implement a subscriber are shown in Figure 5-25.

```
ros::init(argc,argv,"attitude_control");

ros::NodeHandle n;

ros::Subscriber sub = n.subscribe("fcu/imu", 1, ImuCallback);

ros::spinOnce();
```

The first line initializes ROS and allows it also to do name remapping through the command line. In this line, the name of the node is also specified (*attitude_control* in this case) and must be unique in the system. The *NodeHandle* creates a handle to the node of this process and actually initializes it. The third line is the initialization of the subscriber and it receives messages on the topic *fcu/imu* as indicated in the first argument in the brackets. The second argument defines the queue size and therefore the maximum size of messages stored before the last one will be deleted. The third argument is the callback function with the name *ImuCallback* which will be called by ROS every time a new message arrives. This line returns a so-called subscriber object and must be hold on until one wants to unsubscribe from a topic. This can be done by a destructor. The last line forces the program call message callbacks as fast as possible. The same procedure is also executed for the topic *fcu/current_pose,* which delivers the current position of the vehicle [31]

The *ImuCallback* function is used to obtain the needed values from the messages published on the *fcu/Imu* topic. The above explained conversion to Euler angles is also integrated in this function which is shown in Figure 5-26.

```
void ImuCallback(const sensor_msgs::Imu::ConstPtr& msg)
{

    q_x = (msg->orientation).x;
    q_y = (msg->orientation).y;
    q_z = (msg->orientation).z;
    q_w = (msg->orientation).w;

    p = (msg->angular_velocity).x;
    q = (msg->angular_velocity).y;
    r = (msg->angular_velocity).z;

// roll (x-axis rotation)
    double sinr = +2.0 * (q_w * q_x + q_y * q_z);
    double cosr = +1.0 - 2.0 * (q_x * q_x + q_y * q_y);
    roll = atan2(sinr, cosr);

    // pitch (y-axis rotation)
    double sinp = +2.0 * (q_w * q_y - q_z * q_x);
    if (fabs(sinp) >= 1)
        pitch = copysign(M_PI / 2, sinp); // use 90 degrees if out of range
    else
        pitch = asin(sinp);

    // yaw (z-axis rotation)
    double siny = +2.0 * (q_w * q_z + q_x * q_y);
    double cosy = +1.0 - 2.0 * (q_y * q_y + q_z * q_z);
    yaw = atan2(siny, cosy);
}
```

Figure 5-26 ImuCallback function

The message that is published enters the function and contains the values as specified in Figure 5-23 . The values of orientation and angular velocity are obtained and assigned to the corresponding variables.

5.4.2. Control computation

After the data is obtained from the vehicle, the inputs of the control system are known. Thus, the outputs can be computed according to the implemented control scheme shown and described in the simulation part above: Therefore, the task in this section is to write all the functions executed by the blocks above in programming language C++.

As can be seen above, some transformations between coordinate systems have to be made in order that the physical equations are applicable when using the same coordinate system or when an inertial system is needed so that the equation is valid.

For this reason, a transformation function was created that uses a transformation matrix and elements of a vector as input and delivers a transformed vector as output. Figure 5-27 shows the implemented function for a transformation from system 1 to 2 where the matrix is defined as a two-dimensional double array.

```
void transformation_1_2(double A_1_2[3][3],double U_1, double V_1, double W_1,double &U_2, double &V_2, double &W_2)
{
    U_2 = A_1_2[0][0]*U_1+A_1_2[0][1]*V_1+A_1_2[0][2]*W_1;
    V_2 = A_1_2[1][0]*U_1+A_1_2[1][1]*V_1+A_1_2[1][2]*W_1;
    V_2 = A_1_2[2][0]*U_1+A_1_2[2][1]*V_1+A_1_2[2][2]*W_1;
}
```

Figure 5-27 Transformation function

In addition, the function used in the position control block described in chapter 4.3 had to be implemented.

```
void control_posicion(double eN, double eE, double eD, double &VN_sp, double &VE_sp)
{
    double theta = atan2(eD,sqrt((eN*eN)+(eE*eE)));
    double psi = atan2(eN,eE);

    double R = sqrt((eN*eN)+(eE*eE)+(eD*eD));

    double vel = min((R*0.1),0.5);

    VN_sp = vel*cos(theta)*sin(psi);

    VE_sp = vel*cos(theta)*cos(psi);

}
```

Figure 5-28 Function from the position control block

Figure 5-28 above shows the errors denoted in system NED as inputs and the calculated velocities in North and East as output of this function.

The most difficult part in creating an executable code from the existing control scheme was the PID controller itself. However, Åström and Hägglund proposed a computer code to tackle this problem on pages 107 to 108 in [32]. Nevertheless, the code is written in the programming language Pascal and had to be adapted to the syntax of C++. Furthermore, bumpless parameter changes as feature included in the offered code was omitted. The resulting code can be seen in Figure 5-29.

```
double PID(float K_p, float K_i, float K_d, double I_ac,double h,float T_t,double y_dot,double e)
{

    double b_i = K_i*h; //integral gain
    double a_d = K_d; //derivative gain
    double a_0 = h/T_t;

    double I=I_ac;
    double P = K_p*e; //proportional part
    double D=a_d*y_dot;//derivative part
    double v = P+I+D; //temporary output
    double u = min(max(v,-1.5),1.5); //saturation
    I += b_i*e+a_0*(u-v);
    double y = P+I+D;
    //I_ac = e;
    return y;
}
```

Figure 5-29 PID controller as function in C++

The inputs to the function are the factors K_P, K_I and K_D as well as the accumulated integral error I_{ac}, the sampling rate h, the tracking time constant T_t, the derivative of the input variable with respect to the time $\dot{y}$ and the error e.

The factors of the PID controllers for each controllers are as determined above. The accumulated error is added up over time. The sample rate is chosen at 60 Hz to be fast enough to control the system with highly dynamic behavior. [32] proposes a thumb rule to choose the tracking time as

$$T_t = \sqrt{T_i T_d} \tag{5.7}$$

with

reset time $T_i = \frac{K_P}{K_I}$

derivative action time $T_d = \frac{K_D}{K_P}$

Taking the values for the parameters of the PID controller into account, this would lead to a factor T_t of about 1,15 for pitch and roll controller.

The output of the controller function is the manipulated variable.

All of the above-mentioned function definitions are arranged above the main function block of the program.

In the *main* block, first the initialization of ROS is executed and the *NodeHandle* function is called. Thereafter, the service for starting and stopping the motors, which is already implemented in the topic *fcu/motor_control* is used to start the motors. First, variables of the type *Request* and *Response* , both defined by the *asctec_hl_comm* communication package that comes with the *asctec_mav_framework* package, are created.

The function to start the motors is called by set *req.startMotors* to 1. If one sets this value to 0, the motors will be turned off as is the case at the end of the program.
Then, the call of a ROS service works very similar to a subscriber with the arguments of the topic on which the message has to be sent as well as the message itself. In this case the message is to send the value 1 to start the motors.

```
int main(int argc, char **argv)
{
    ros::init(argc,argv,"attitude_control");

    ros::NodeHandle n;
    //block for starting motors
    asctec_hl_comm::mav_ctrl_motors::Request req;
    asctec_hl_comm::mav_ctrl_motors::Response res;
    req.startMotors = 1;
    ros::service::call("fcu/motor_control", req, res);
    std::cout << "motors running: " << (int)res.motorsRunning << std::endl;

    //subscriber to get quaternion and angular velocities
    ros::Subscriber sub = n.subscribe("fcu/imu", 1, ImuCallback);
    //subscriber to get current pose
    ros::Subscriber subPose = n.subscribe("fcu/current_pose",1,PoseCallback);
```

Figure 5-30 First part of main *program block*

As can be seen in Figure 5-30 above, after the before-mentioned steps, the subscribers for the IMU data and position data are called.

In the next block, the two transformation matrices to convert from the body to the world frame and vice versa are generated. The calculation uses the angles calculated from the received data of the IMU.

Thereafter follows the position control block, where all the calculations made in the position control block of the Simulink model are made. Set points for the position are set to [0,0,0] as the data sent from the current pose signal also sends [0,0,0] due to the missing signal of the GPS and no other input signal about the current position. Therefore, the set points urge the vehicle to stay in its actual position.
In the attitude control part, the parameters of the PID controllers are defined.

5.4.3. Command publication

Publication of messages works similar to the subscription to a topic. First, a publisher object has to be created and thereafter the message type is defined. In this case, the message type *directMotorControl* is chosen as this activates a control mode where the default attitude control implemented on the LLP is ignored and the self-designed program is used instead. With the same *Nodehandle* as above, a publisher sends messages on the topic *fcu/control*. These commands can be seen in Figure 5-31.

```
ros::Publisher pub;


asctec_hl_comm::mav_ctrl msg;
msg.type = asctec_hl_comm::mav_ctrl::directMotorControl;

pub = n.advertise<asctec_hl_comm::mav_ctrl> ("fcu/control",1);

ros::Rate f(h);
 for (int i = 0; i < 1200; i++)
{
ros::spinOnce();


e_N = x_sp-x;
e_E = y_sp-y;
e_D = z_sp-z;


control_posicion(e_N,e_E,e_D,VN_sp,VE_sp);

transformation_1_2(A_w_b,U_b,V_b,W_b,U_w,V_w,W_w);

float gain_V = 0.2;

VN_ref = VN_sp - gain_V*U_w;
VE_ref = VE_sp - gain_V*V_w;
VD_ref = z_sp;

/*Tranformation w_b */

transformation_1_2(A_b_w,VN_ref,VE_ref,VD_ref,U_w,V_w,W_w);

roll_sp = -V_w;
pitch_sp = U_w;
yaw_sp = -M_PI/2;

//error calculation
double e_r = roll_sp-roll;
double e_p = pitch_sp-pitch;
double e_y = yaw_sp-yaw;

/* PID Controller */
double roll_cmd = PID(Kpr,Kir,Kdr,I_acr,h,T_tr,p,e_r);
double pitch_cmd = PID(Kpp,Kip,Kdp,I_acp,h,T_tp,q,e_p);
double yaw_cmd = PID(Kpy,Kiy,Kdy,I_acy,h,T_ty,p,e_y);
```

Figure 5-31 Control loop in the program

The following command line is to define the execution rate of the program and it is chosen the same rate as used in the PID controller function.
Afterwards, the control loop is implemented with a duration of 1200 spins.

The errors in the NED system are calculated from the set points of x, y and z and their actual values.
Thereafter the program works in its steps as described above for the simulator.
Omitting the velocity control part, the set points for the angles are derived from the transformed velocities calculated in the position control function *control_posicion*.
The errors for the angles of attitude are then used as inputs when the PID controller functions for each angle roll, pitch and yaw is called.

Because of the needed input signal for the roll and pitch command, the output signals have to be scaled. The input range between -100% and +100% for these angles is scaled from 0 to 200. This means, if the maximum inclination in the negative direction of the axis shoud be reached, the signal must be 0 and in the positive direction of the axis, it must be 200. Also, the maximum angles are limited to +/-52°. This leads to the following two equations derived from the equation for a straight line.

$$y = k * x + d \tag{5.8}$$

$$0 = k * \frac{-52 * \pi}{180} + d \tag{5.9}$$

$$200 = k * \frac{52 * \pi}{180} + d \tag{5.10}$$

When solving this system of equations 5.9 and 5.10, the following constants arise:

$$k = 110{,}184191$$
$$d = 100$$

```
double roll_cmd_corr=110.184191*roll_cmd+100;
double pitch_cmd_corr=110.184191*pitch_cmd+100;

std::cout <<"PHI:" << roll << ", SP:"<< roll_sp<<std::endl;
std::cout <<"THETA:" << pitch  << ", SP:"<< pitch_sp<<std::endl;
std::cout <<"PSI:" << yaw <<std::endl;
std::cout << std::endl;
std::cout <<"PHI_CMD:" << roll_cmd_corr <<" original:"<<roll_cmd<<std::endl;
std::cout <<"THETA_CMD:" << pitch_cmd_corr <<" original:"<<pitch_cmd<<std::endl;
std::cout <<"PSI_CMD:" << yaw_cmd <<std::endl;
std::cout << std::endl;
std::cout <<"p:" << p <<std::endl;
std::cout <<"q:" << q <<std::endl;
std::cout <<"r:" << r <<std::endl;
std::cout << std::endl;
std::cout <<"x:" << x <<std::endl;
std::cout <<"y:" << y <<std::endl;
std::cout <<"z:" << z <<std::endl;
std::cout << std::endl;
std::cout <<"I_acp:" << I_acp << std::endl;
std::cout << std::endl;
```

Figure 5-32 Correction of command values and printing of outputs

Figure 5-32 above shows, that the calculated values are used to correct the command output of each angle.

For testing reasons, all of the output signals are then displayed to the terminal window in order to check, if they are in a valid range and if the control works properly as can be seen in . In addition, the correction for the scale can also be checked by printing the signal without the above-mentioned computations. Furthermore the angular velocities as well as the current position and are also displayed as well as the accumulated error in order to check its functionality.

```
    msg.x = pitch_cmd_corr;
    msg.y = roll_cmd_corr;
    msg.z = 0;
    msg.yaw = 98;
    msg.v_max_xy = -1;
    msg.v_max_z = -1;
    pub.publish(msg);

        if (!ros::ok())
         return 0;
         f.sleep();

    }
    //turn off motors
    req.startMotors = 0;
    ros::service::call("fcu/motor_control", req, res);
    std::cout << "motors running: " << (int)res.motorsRunning << std::endl;


    return 0;

}
```

Figure 5-33 Creation of messages an d publication

In Figure 5-33 can be seen, that the messages values are assigned according to the type of the message of *fcu/control* which sends the 6 values *x, y, z, yaw, v_max_xy* and *v_max_z* with the latter two being the maximum velocities in the direction of the x-axis and y-axis as well as in the direction of the z-axis, respectively. The *x* value is the roll angle, the *y* value for the pitch angle and the *z* value for the thrust.

The commands are then published by simply using the function *publish* with the message *msg* as argument.

The method *ros::ok* is used for shutting down the node and therefore destructing all subscribers and publishers.

Thereafter, the motors are shut down similar to the process of starting them as described above.

5.5. Experiments on the drone

After test runs on the simulation were completed successfully and the program representing the control process was written, first tests on the drone could be made. The control was still in a testing phase and the results from the simulation should always be considered with caution because of all the effects that exist on the real system and could not be depicted in the simulator to keep the model relatively simple and computation fast. For this reason, it was considered too dangerous to implement the algorithm directly on the drone and let it fly autonomously but rather put it in a test rig and run some tests first. On one hand, the task was to fix the drone so that it could not crash down on the ground or ascend to the ceiling. On the other hand, the vehicle must be able to move in order to check the functionality of the newly designed attitude controller. As the system already has a high degree of complexity and the model shows that all the angles have mathematical connections to each other, it is not as trivial to see the reaction of the quadcopter to the input made to it. Therefore, giving the vehicle only 1 degree of freedom, namely the rotation around one axis, makes it easier for humans to draw at least qualitative conclusions about the correctness of the behavior and if it was expected or not.

As the Hummingbird for this project has +-configuration as mentioned and explained above, the easiest way is to fix the two arms of one axis, letting it rotate around this axis and therefore judge about the roll or pitch control.

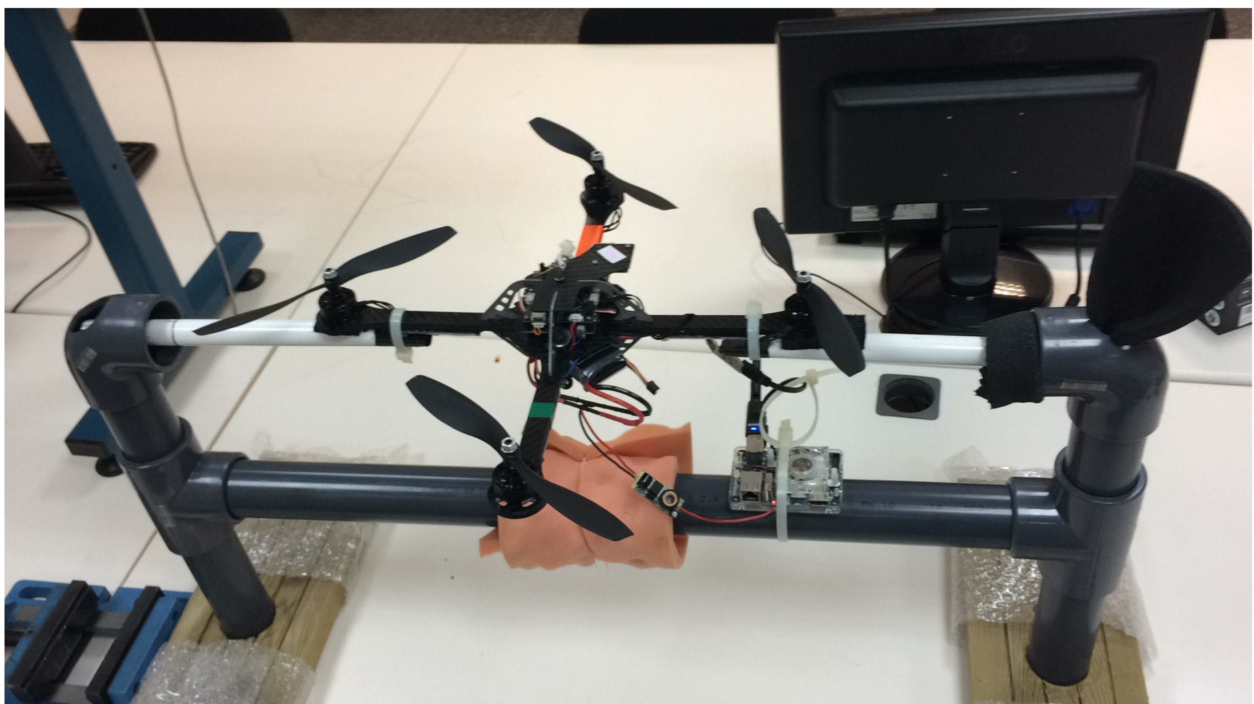


Figure 5-34 Hummingbird test rig

Figure 5-34 above shows the test rig used for conducting first experiments testing the attitude control algorithm. As can be seen, odroid was fixed on the test rig itself and not on Hummingbird. Furthermore, a bearing on each side in the tubes connecting the white rods and the tubes was integrated to minimize friction when the vehicle rotates.

To create an executable file in ROS, the package *asctec_hl_comm* had to be built new. First, the new C++ file had to be added in the sources folder *src* in the corresponding package. Thereafter, a new build process had to be started by using the command *catkin_make*. This command will build any packages located in the source folder of the current workspace. After the package is successfully built, it will be possible to execute the program via the terminal using the command *rosrun asctec_hl_comm ctrl_test_Florian.* The command *rosrun* is used for running programs via ROS using the terminal. The second term shows the actual package where the executable file is located and the last term is the name of the file with no ending necessary. Also, the file location does not have to be written like a usual path due to the ROS filesystem level mentioned in chapter 2.4.2. In contrast to the original *ctrl_test* program used as template for the newly created file, no more arguments are needed to start the process.
As already mentioned, before starting the experiment, one person has to be in charge of the radio control during the whole test in case anything unexpected happens. The switch for manual mode has to be put in the correct position, otherwise the default control algorithms from the LLP will take control of the vehicle and the new algorithm could not be tested.

6. Conclusion

6.1. Findings

The objective of the present master thesis was to create an attitude control algorithm for a quadcopter using the middleware ROS. The drone used in this project is a vehicle called Hummingbird and already existed in the laboratories of the UPC for experiments on control of an under actuated and highly agile system. The drone is controlled only by the angular velocities of its four rotors thus having four controllable variables in a three-dimensional system with 6 degrees of freedom. As the angular velocities of each rotor are hard to judge qualitatively about the behavior and the control of the vehicle it is easier to work with the three angles of attitude, namely roll pitch and yaw and with the altitude z. The creation of a model of the system shows the relations between the angles and the forces and moments the rotors generate which can be linked with the rotational speed using empirical determined factors.

To minimize the risk of failure and therefore demolition of the vehicle, the development and improvement of the algorithm was made using the simulation software Simulink. An existing model named Quad-Sim was used and modified to match the specifications of this thesis. The system consisted of three steps of control, namely position, velocity and attitude control. The velocity control block was eliminated but the position control block was retained and modified. Attitude control does not work without at least rudimentary position control as little deviations or permanent errors controlling the angles roll or pitch or the altitude z will lead to a drift of the vehicle over time. Position control is only possible if the current position is known. Many quadcopter research projects rely on position data delivered by a sensor connecting to a GPS/GLONASS system. As this project had to be developed indoors where the signal is either very weak or not available, other options of obtaining the current position had to be considered. A system implemented for a former thesis developed in the same laboratory using image recognition of four markers detected by four stationary cameras was taken into account. However, it turned out to be very difficult to mount the markers which were designed to be mounted on a model helicopter skid landing gear onto the quadcopter.

Therefore the project was focused on the development of the control algorithm assuming getting position data without specifying how to tackle this problem.

The simulation helped to obtain the parameters of the PID controllers by optimizing to a fast reaction of the agile system allowing 10% of overshoot in the controlled variables.

Due to their importance for attitude of the flying object, the work was concentrated on the angles roll and pitch because deviations in these angles would cause the quadcopter to drift.
The resulting control scheme was then written in the programming language C++ to create an executable file for ROS. This middleware which runs on Linux is used to communicate between parts of an automated system like the quadcopter and a ground computer can do. The code was implemented on the processor to run autonomously and can be started from the before-mentioned ground station via a command entered in a terminal window of Linux. Through a Wi-Fi connection, changes are possible whenever needed, also during flight.
Experiments on a test rig showed that the implemented algorithm works and that the parameters of the simulation were obtained in the correct range.

6.2. Outlook

The further optimization of the PID controller parameters is a duty on which further researchers could concentrate. The parameters obtained by the simulation are leading to a working system, nevertheless some influences on the system could not be depicted on the model because they are either not representable or would increase the complexity of the model by an unreasonable amount.
Possible successors may focus their work on implementing the mentioned image recognition system by designing markers adequate for the existing system and using the stationary cameras to fly in the defined space. Furthermore, one can think of mounting cameras onto the vehicle and recognizing the environment creating a map where it flies to obtain position data. This would make the quadcopter more flexible and it could not only be used in the limited space covered by the cameras.

7. Table of Figures

Figure 1-1 Markers used for image recognition....13
Figure 2-1 Intel Falcon 8+ drone [14]....16
Figure 2-2 Hummingbird quadrotor....17
Figure 2-3 Aluminum structure of the main body....18
Figure 2-4 Futaba F77 radio control....19
Figure 2-5 GPS Module on Hummingbird quadcopter [27]....20
Figure 2-6 Odroid mounted on Hummingbird....21
Figure 2-7 Main board ODROID [28]....22
Figure 2-8 Terminal window....26
Figure 3-1 Unity feedback system [19]....30
Figure 3-2 Block diagram of the control loop....34
Figure 4-1 Simulink system consisting of different connected blocks....37
Figure 4-2 Calculation part of a block....37
Figure 4-3 Quad-Sim simulator opened in Simulink....38
Figure 4-4 Position control block....39
Figure 4-5 Velocity control block....39
Figure 4-6 Attitude command block....40
Figure 4-7 Structure of PID block in attitude control block....41
Figure 4-8 Changed Simulink structure....41
Figure 4-9 Position control block....42
Figure 4-10 Matlab function for velocity calculation....43
Figure 4-11 Calculation of velocity based on errors....43
Figure 4-12 Transformation block....45
Figure 5-1 Button to open the GUI for building a new model....46
Figure 5-2 GUI to define a new quadcopter model in Quad-Sim....47
Figure 5-3 + and x configuration....48
Figure 5-4 Self-created PID blocks....49
Figure 5-5 Structure of PID controllers....49
Figure 5-6 Altitude controller....50
Figure 5-7 MATLAB struct with all the initial conditions....51
Figure 5-8 Solver properties for Simulink model....52
Figure 5-9 MATLAB plots of all state values with ramp slope 1 in x....54
Figure 5-10 Output graphs with step input signal....55
Figure 5-11 Results optimization of z axis PID controller....56
Figure 5-12 Simulink result for inputs to every axis....57
Figure 5-13 Optimize function....59
Figure 5-14 x-axis first optimization....61
Figure 5-15 y-axis first optimization....61
Figure 5-16 z-axis first optimization....62
Figure 5-17 y-axis second optimization....63
Figure 5-18 x-axis second optimization....63
Figure 5-19 z-axis second optimization....64
Figure 5-20 Additional code lines to calculate first derivative of error....64

Figure 5-21 Comparison of optimization results 65
Figure 5-22 Published topics, subscribed topics and services of asctec_hl_interface [27] ...68
Figure 5-23 Message definition of the topic fcu/imu [28] 68
Figure 5-24 Conversion from quaternions to Euler angles in C++ 69
Figure 5-25 ROS subscriber written in C++..69
Figure 5-26 ImuCallback function..70
Figure 5-27 Transformation function 71
Figure 5-28 Function from the position control block 71
Figure 5-29 PID controller as function in C++...72
Figure 5-30 First part of main program block 73
Figure 5-31 Control loop in the program...74
Figure 5-32 Correction of command values and printing of outputs..76
Figure 5-33 Creation of messages an d publication...77
Figure 5-34 Hummingbird test rig...78

8. Bibliography

[1] E. Altuğ, J. P. Ostrowski and R. Mahony, "Control of a Quadrotor Helicopter Using Visual Feedback," in *IEEE International Conference on Robotics &Automation*, Washington, DC, 2002.

[2] N. Guenard, T. Hamel and R. Mahony, "A Practical Visual Servo Control for an Unmanned Aerial Vehicle," *IEEE Transactions on Robotics,* vol. 24, no. 2, pp. 331-340, April 2008.

[3] S. Zingg, D. Scaramuzza, S. Weiss and R. Siegwart, "MAV Navigation through Indoor Corridors Using Optical Flow," in *2010 IEEE International Conference on Robotics and Automation*, Anchorage, Alaska, 2010.

[4] B. A. Kumar and D. Ghose, "Radar-Assisted Collision Avoidance/Guidance Strategy for Planar Flight," *Transactions on Aerospace and Electronic Systems,* vol. 37, no. 1, pp. 77-90, January 2001.

[5] Y. K. Kwag and J. W. Kang, "Obstacle awareness and collision avoidance radar sensor system for low-altitude flying smart uav," in *Digital Avionics Systems Conference*, 2004.

[6] M. Bloesch, S. Weiss, D. Scaramuzza and R. Siegwart, "Vision Based MAV Navigation in Unknown and Unstructured Environments," in *2010 IEEE International Conference on Robotics and Automation*, Anchorage, Alaska, 2010.

[7] M. Achtelik, M. Achtelik, S. Weiss and R. Siegwart, "Onboard IMU and Monocular Vision Based Control for MAVs in Unknown In- and Outdoor Environments," in *2011 IEEE International Conference on Robotics and Automation*, Shanghai, 2011.

[8] S. Park, D. Won and M. Kang, "RIC(Robust Internal-loop Compensator) Based Flight Control of a Quad-Rotor Type UAV," in *2005 IEEE/RSJ International Conference on Intelligent Robots and Systems*, Edmonton, Alberta, 2005.

[9] S. Klose, J. Wang, M. Achtelik , G. Panin, F. Holzapfel and A. Knoll, "Markerless, Vision-Assisted Flight Control of a Quadrocopter," in *The 2010 IEEE/RSJ International Conference on Intelligent Robots and Systems*, Taiwan, 2010.

[10] D. Eberli, D. Scaramuzza, S. Weiss and R. Siegwart, "Vision Based Position Control for MAVs Using One Single Circular Landmark," *Journal of Intelligent & Robotic Systems,* vol. 61, no. 1, pp. 495-512, 2011.

[11] C. Martínez Céspedes, "Disseny d'un sistema de detecció de marcadors amb ROS i visió per computador," Universitat Politècnica de Catalunya, Terrassa, 2016.

[12] C. E. Barrionuevo Sánchez, "Estudio de algoritmos de control de trayectoria con ROS para un helicóptero coaxial," Universitat Politècnica de Catalunya, Terrassa, 2017.

[13] S. Bouabdallah, "Design and control of quadrotors with application to autonomous flying," École Polytechnique Fédérale de Lausanne, Lausanne, 2007.

[14] Intel corporation, "Intel Falcon 8+ System," [Online]. Available: https://www.intel.com/content/www/us/en/products/drones/falcon-8.html. [Accessed 05 December 2017].

[15] Open Source Robotics Foundation, "ROS.org | About ROS," [Online]. Available: http://www.ros.org/about-ros/. [Accessed 10 November 2017].

[16] J. O'Kane, A Gentle Introduction to ROS, Columbia, South Carolina: Jason Matthew O'Kane, 2014.

[17] J. S. Lum, "Utilizing Robot Operating System (ROS) in robot vision and control," Naval Postgraduate School, Monterey,CA, 2015.

[18] Open Source Robotics Foundation, "Documentation - ROS Wiki," 31 March 2017. [Online]. Available: http://wiki.ros.org. [Accessed 10 October 2017].

[19] Conference Catalysts, LLC, "Control Systems are Ubiquitous | IEEE Control Systems Society," [Online]. Available: http://www.ieeecss.org/control-systems-are-ubiquitous-2016. [Accessed 04 January 2018].

[20] R. N. Clark, Control System Dynamics, New York: Cambridge University Press, 1996.

[21] A. Azzam and X. Wang, "Quad Rotor Arial Robot Dynamic Modeling and Configuration Stabilization," in *2nd International Asia Conference on Informatics in Control, Automation and Robotics*, Wuhan, 2010.

[22] P. Wang, Z. Man, Z. Cao, J. Zheng and Y. Zhao, "Dynamics Modelling and Linear Control of Quadcopter," in *International Conference on Advanced Mechatronic Systems*, Melbourne, 2016.

[23] Ascending Technologies GmbH, "CAD Models - AscTec Research - Ascending Technologies Costumer Wiki," 2016. [Online]. Available: http://wiki.asctec.de/display/AR/CAD+Models. [Accessed 11 January 2018].

[24] N. Michael, D. Mellinger, Q. Lindsey and V. Kumar, "The GRASP Multiple Micro-UAV Testbed," *IEEE Robotics & Automation Magazine,* vol. 17, no. 3, pp. 56-65, September 2010.

[25] GitHub, "GitHub-dch33/QuadSIm:A package of documentation and software supporting MATLAB/Simulink based dynamic modeling and simulation of quadcopter vehicles for control system design," 2017. [Online]. Available: https://github.com/dch33/Quad-Sim. [Accessed 20 December 2017].

[26] Ascending Technologies GmbH, "AscTec Hummingbird /// R&D UAS for swarming & control theory .: Ascending Technologies," 2018. [Online]. Available: http://www.asctec.de/en/uav-uas-drones-rpas-roav/asctec-hummingbird/. [Accessed 11 January 2018].

[27] S. Jakubek, Grundlagen der Regelungstechnik, Vienna: TU Wien, 2016.

[28] Open Source Robotics Foundation, "asctec_hl_interface - ROS Wiki," [Online]. Available: http://wiki.ros.org/asctec_hl_interface. [Accessed 13 February 2018].

[29] Open Source Robotics Foundation, "geometry_msgs - ROS Wiki," [Online]. Available: http://wiki.ros.org/geometry_msgs. [Accessed 13 February 2018].

[30] Wikipedia, "Conversion between quaternions and Euler angles - Wikipedia," 17 November 2017. [Online]. Available: https://en.wikipedia.org/wiki/Conversion_between_quaternions_and_Euler_angles. [Accessed 15 February 2018].

[31] Open Source Robotics Foundation, "ROS/Tutorials/WritingPublisherSubscriber(c++) - ROS Wiki," 28 September 2016. [Online]. Available: http://wiki.ros.org/ROS/Tutorials/WritingPublisherSubscriber%28c%2B%2B%29. [Accessed 15 February 2018].

[32] K. J. Åström and T. Hägglund, PID Controllers: Theory, Design and Tuning, Research Triangle Park, North Carolina: Instrument Society of America, 1995.

[33] Wikipedia, "Quaternion - Wikipedia," 8 February 2018. [Online]. Available: https://en.wikipedia.org/wiki/Quaternion. [Accessed 15 February 2018].

[34] IDM Research Group, "Identification and Decision Making Research Group," 2015. [Online]. Available: http://idm.kky.zcu.cz/files/robots/quadcopter09.JPG. [Accessed 19 December 2017].

[35] Hardkernel co., Ltd., "ODROID | Hardkernel," 2013. [Online]. Available: http://www.hardkernel.com/main/products/prdt_info.php?g_code=G143452239825&tab_idx=2. [Accessed 12 November 2017].

[36] Open Source Robotics Foundation, "sensor_msgs/Imu Documentation," [Online]. Available: http://docs.ros.org/api/sensor_msgs/html/msg/Imu.html. [Accessed 13 February 2018].

Annex I

Quaternions

Quaternions are an extension of the complex numbers often used in mechanics and in three-dimensional space. They were first defined by Irish mathematician William Rowan Hamilton as the quotient of two directed lines in a three-dimensional space or as the quotient of two vectors. Quaternions can be represented as follows:

$$a + b\boldsymbol{i} + c\boldsymbol{j} + d\boldsymbol{k}$$

with $a, b, c, d \in \mathbb{R}$
i,**j** and **k** are the so-called fundamental quaternion units, where:

$$\boldsymbol{i}^2 = \boldsymbol{j}^2 = \boldsymbol{k}^2 = \boldsymbol{ijk} = -1$$

One has to be careful with the application of mathematical laws on quaternions as the commutative law is not valid, for example.

In analogy to complex numbers, given a quaternion

$$q = a + bi + cj + dk$$

a is called the real part, whereas $bi + cj + dk$ is the imaginary part.
Another definition calls a the scalar and $bi + cj + dk$ the vector part, but this vector must not be confused with a polar vector in three-dimensional space and is therefore often called an "axial" vector or a "pseudo vector".
In many applications such as three-dimensional rotations, quaternions are used either alongside Euler angles or as an alternative to them.
The definition of quaternions can be used in the definition of rotations of rigid bodies according to Euler's rotation theorem. The theorem states that every rotation or sequence of rotations about a fixed point is equivalent to a single rotation by a given angle θ about a fixed Euler axis that runs through the fixed point. This axis is typically represented by a unit vector $\vec{u}$. So it is possible to denote any rotation as a combination of a vector $\vec{u}$ and a scalar θ. This representation can easily be realized by quaternions as they consist of a scalar and a vector part.

With a vector

$$\vec{u} = \{u_x, u_y, u_z\}$$

in mind, an extension of Euler's formula can be written as

$$\boldsymbol{q} = e^{\frac{\theta}{2}(u_x\boldsymbol{i}+u_y\boldsymbol{j}+u_z\boldsymbol{k})} = cos\frac{\theta}{2} + (u_x\boldsymbol{i} + u_y\boldsymbol{j} + u_z\boldsymbol{k})\, sin\frac{\theta}{2}$$

[33]

Annex II

```
/*

Copyright (c) 2011, Markus Achtelik, ASL, ETH Zurich, Switzerland
You can contact the author at <markus dot achtelik at mavt dot ethz
dot ch>

All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions are
met:
* Redistributions of source code must retain the above copyright
notice, this list of conditions and the following disclaimer.
* Redistributions in binary form must reproduce the above copyright
notice, this list of conditions and the following disclaimer in the
documentation and/or other materials provided with the distribution.
* Neither the name of ETHZ-ASL nor the
names of its contributors may be used to endorse or promote products
derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
"AS IS" AND
ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
IMPLIED
WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE
DISCLAIMED. IN NO EVENT SHALL ETHZ-ASL BE LIABLE FOR ANY
DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
DAMAGES
(INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR
SERVICES;
LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER
CAUSED AND
ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR
TORT
(INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
OF THIS
SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

*/

#include <string.h>
#include <stdio.h>
#include <math.h>

#include <ros/ros.h>
#include <asctec_hl_comm/mav_ctrl.h>
#include <asctec_hl_comm/mav_ctrl_motors.h>
```

```
void usage()
{
  std::string text("usage: \n\n");
  text = "ctrl_test motors [0 | 1]\n";
  text += "ctrl_test ctrl [acc | vel | pos | vel_b | pos_b] x y z
yaw\n";
  text += "position / velocity in [m] / [m/s] and yaw in [deg] /
[deg/s] (-180 ... 180)\n";
  std::cout << text << std::endl;
}

int main(int argc, char ** argv)
{

  ros::init(argc, argv, "interfacetest");
  ros::NodeHandle nh;

  ros::Publisher pub;

  if (argc == 1)
  {
    ROS_ERROR("Wrong number of arguments!!!");
    usage();
    return -1;
  }

  std::string command = std::string(argv[1]);

  if (command == "motors")
  {
    if (argc != 3)
    {
      ROS_ERROR("Wrong number of arguments!!!");
      usage();
      return -1;
    }

    asctec_hl_comm::mav_ctrl_motors::Request req;
    asctec_hl_comm::mav_ctrl_motors::Response res;
    req.startMotors = atoi(argv[2]);
    ros::service::call("fcu/motor_control", req, res);
    std::cout << "motors running: " << (int)res.motorsRunning <<
std::endl;
  }
  else if (command == "ctrl")
  {
    if (argc != 7)
    {
      ROS_ERROR("Wrong number of arguments!");
      usage();
      return -1;
    }
```

```
    asctec_hl_comm::mav_ctrl msg;
    msg.x = atof(argv[3]);
    msg.y = atof(argv[4]);
    msg.z = atof(argv[5]);
    msg.yaw = atof(argv[6]) * M_PI / 180.0;
    msg.v_max_xy = -1; // use max velocity from config
    msg.v_max_z = -1;

    std::string type(argv[2]);
    if (type == "acc")
      msg.type = asctec_hl_comm::mav_ctrl::acceleration;
    else if (type == "vel")
      msg.type = asctec_hl_comm::mav_ctrl::velocity;
    else if (type == "pos")
      msg.type = asctec_hl_comm::mav_ctrl::position;
    else if (type == "vel_b")
      msg.type = asctec_hl_comm::mav_ctrl::velocity_body;
    else if (type == "pos_b")
      msg.type = asctec_hl_comm::mav_ctrl::position_body;
    else
    {
      ROS_ERROR("Command type not recognized");
      usage();
      return -1;
    }

    pub = nh.advertise<asctec_hl_comm::mav_ctrl> ("fcu/control", 1);
    ros::Rate r(15); // ~15 Hz

    for (int i = 0; i < 15; i++)
    {
      pub.publish(msg);
      if (!ros::ok())
        return 0;
      r.sleep();
    }

    // reset
    if (type != "pos" && type != "pos_b")
    {
      msg.x = 0;
      msg.y = 0;
      msg.z = 0;
      msg.yaw = 0;
    }

    for(int i=0; i<5; i++){
      pub.publish(msg);
      r.sleep();
    }
    ros::spinOnce();
  }
  return 0;
}
```

Annex III

```
/*

 Developed by Florian Schebesta
 Master thesis "Attitude control algorithm for Quadcopter"
 derived at Universitat Politècnica de Catalunya
 submitted at Vienna Technical University

 Latest Version 26-01-2018

 */

#include <iostream>
using namespace std;
#include <string.h>
#include <stdio.h>
#include <math.h>

#include <ros/ros.h>
#include <asctec_hl_comm/mav_ctrl.h>
#include <asctec_hl_comm/mav_ctrl_motors.h>
#include "geometry_msgs/Quaternion.h"
#include "geometry_msgs/PoseStamped.h"
#include "sensor_msgs/Imu.h"

double q_x;
double q_y;
double q_z;
double q_w;

double p;
double q;
double r;

double roll;
double pitch;
double yaw;

double x=0;
double y=0;
double z=0;

double U_b;
double V_b;
double W_b;

double U_w;
double V_w;
double W_w;

double VN_sp;
double VE_sp;
```

```
double A_w_b[3][3];

double A_b_w[3][3];

double A_1_2[3][3];


//accumulated integral parts
    double I_acr;
    double I_acp;
    double I_acy;


void PoseCallback(const geometry_msgs::PoseStamped::ConstPtr& msg)
{ x = (msg->pose).position.x;
  y = (msg->pose).position.y;
  z = (msg->pose).position.z;
}

void ImuCallback(const sensor_msgs::Imu::ConstPtr& msg)
{

    q_x = (msg->orientation).x;
    q_y = (msg->orientation).y;
    q_z = (msg->orientation).z;
    q_w = (msg->orientation).w;

    p = (msg->angular_velocity).x;
    q = (msg->angular_velocity).y;
    r = (msg->angular_velocity).z;

// roll (x-axis rotation)
    double sinr = +2.0 * (q_w * q_x + q_y * q_z);
    double cosr = +1.0 - 2.0 * (q_x * q_x + q_y * q_y);
    roll = atan2(sinr, cosr);

    // pitch (y-axis rotation)
    double sinp = +2.0 * (q_w * q_y - q_z * q_x);
    if (fabs(sinp) >= 1)
        pitch = copysign(M_PI / 2, sinp); // use 90 degrees if out of
range
    else
        pitch = asin(sinp);

    // yaw (z-axis rotation)
    double siny = +2.0 * (q_w * q_z + q_x * q_y);
    double cosy = +1.0 - 2.0 * (q_y * q_y + q_z * q_z);
    yaw = atan2(siny, cosy);
}
//quaternion to euler angles conversion (from
https://en.wikipedia.org/wiki/Conversion_between_quaternions_and_Euler
_angles )
```

```
void transformation_1_2(double A_1_2[3][3],double U_1, double V_1,
double W_1,double &U_2, double &V_2, double &W_2)
{
    U_2 = A_1_2[0][0]*U_1+A_1_2[0][1]*V_1+A_1_2[0][2]*W_1;
    V_2 = A_1_2[1][0]*U_1+A_1_2[1][1]*V_1+A_1_2[1][2]*W_1;
    V_2 = A_1_2[2][0]*U_1+A_1_2[2][1]*V_1+A_1_2[2][2]*W_1;
}

void control_posicion(double eN, double eE, double eD, double &VN_sp,
double &VE_sp)
{
    double theta = atan2(eD,sqrt((eN*eN)+(eE*eE)));
    double psi = atan2(eN,eE);

    double R = sqrt((eN*eN)+(eE*eE)+(eD*eD));

    double vel = min((R*0.1),0.5);

    VN_sp = vel*cos(theta)*sin(psi);

    VE_sp = vel*cos(theta)*cos(psi);

}

//controller implementation according to astroem p107-108
double PID(float K_p, float K_i, float K_d, double I_ac,double h,float
T_t,double y_dot,double e)
{

    double b_i = K_i*h; //integral gain
    double a_d = K_d; //derivative gain
    double a_0 = h/T_t;

    double I=I_ac;
    double P = K_p*e; //proportional part
    double D=a_d*y_dot;//derivative part
    double v = P+I+D; //temporary output
    double u = min(max(v,-1.5),1.5); //saturation
    I += b_i*e+a_0*(u-v);
    double y = P+I+D;
    //I_ac = e;
    return y;
}




int main(int argc, char **argv)
{
    ros::init(argc,argv,"attitude_control");

    ros::NodeHandle n;
    //block for starting motors
```

```
    asctec_hl_comm::mav_ctrl_motors::Request req;
    asctec_hl_comm::mav_ctrl_motors::Response res;
    req.startMotors = 1;
    ros::service::call("fcu/motor_control", req, res);
    std::cout << "motors running: " << (int)res.motorsRunning <<
std::endl;

    //subscriber to get quaternion and angular velocities
    ros::Subscriber sub = n.subscribe("fcu/imu", 1, ImuCallback);
    //subscriber to get current pose
    ros::Subscriber subPose =
n.subscribe("fcu/current_pose",1,PoseCallback);

    A_w_b[0][0] = cos(yaw)*cos(pitch);
    A_w_b[0][1] = cos(yaw)*sin(pitch)*sin(roll)-sin(yaw)*cos(roll);
    A_w_b[0][2] = cos(yaw)*sin(pitch)*cos(roll)+sin(yaw)*sin(roll);
    A_w_b[1][0] = sin(yaw)*cos(pitch);
    A_w_b[1][1] = sin(yaw)*sin(pitch)*sin(roll)+cos(yaw)*cos(roll);
    A_w_b[1][2] = sin(yaw)*sin(pitch)*cos(roll)-cos(yaw)*sin(roll);
    A_w_b[2][0] = -sin(pitch);
    A_w_b[2][1] = cos(pitch)*sin(roll);
    A_w_b[2][2] = cos(pitch)*cos(roll);

    A_b_w[0][0] = cos(yaw)*cos(pitch);
    A_b_w[1][0] = cos(yaw)*sin(pitch)*sin(roll)-sin(yaw)*cos(roll);
    A_b_w[2][0] = cos(yaw)*sin(pitch)*cos(roll)+sin(yaw)*sin(roll);
    A_b_w[0][1] = sin(yaw)*cos(pitch);
    A_b_w[1][1] = sin(yaw)*sin(pitch)*sin(roll)+cos(yaw)*cos(roll);
    A_b_w[2][1] = sin(yaw)*sin(pitch)*cos(roll)-cos(yaw)*sin(roll);
    A_b_w[0][2] = -sin(pitch);
    A_b_w[1][2] = cos(pitch)*sin(roll);
    A_b_w[2][2] = cos(pitch)*cos(roll);




    /* Position control */

    //setpoint values position
    double x_sp=0;
    double y_sp=0;
    double z_sp=0;

    double e_N;
    double e_E;
    double e_D;


    control_posicion(e_N,e_E,e_D,VN_sp,VE_sp);

    transformation_1_2(A_w_b,U_b,V_b,W_b,U_w,V_w,W_w);

    float gain_V;
```

```
double VN_ref;
double VE_ref;
double VD_ref;

/*Tranformation w_b */

transformation_1_2(A_b_w,VN_ref,VE_ref,VD_ref,U_w,V_w,W_w);

double roll_sp;
double pitch_sp;
double yaw_sp;


/* Attitude control */



//gain values pr...p-value for roll (p...pitch, y...yaw)

float Kpr = 3;
float Kir = 2;
float Kdr = 3;

float Kpp = 5;//2
float Kip = 3;//1.5
float Kdp = 2.8;//1.8

float Kpy = 20;
float Kiy = 6;
float Kdy = 13;


//according to astroem between 8 and 20
double T_tr = 1.15;
double T_tp = 1.15;
double T_ty = 1.15;

int h=60;

ros::Publisher pub;



asctec_hl_comm::mav_ctrl msg;
msg.type = asctec_hl_comm::mav_ctrl::directMotorControl;

pub = n.advertise<asctec_hl_comm::mav_ctrl> ("fcu/control",1);

ros::Rate f(h);
 for (int i = 0; i < 1200; i++)
{
ros::spinOnce();
```

```
e_N = x_sp-x;
e_E = y_sp-y;
e_D = z_sp-z;


control_posicion(e_N,e_E,e_D,VN_sp,VE_sp);

transformation_1_2(A_w_b,U_b,V_b,W_b,U_w,V_w,W_w);

float gain_V = 0.2;

VN_ref = VN_sp - gain_V*U_w;
VE_ref = VE_sp - gain_V*V_w;
VD_ref = z_sp;

/*Tranformation w_b */

transformation_1_2(A_b_w,VN_ref,VE_ref,VD_ref,U_w,V_w,W_w);

roll_sp = -V_w;
pitch_sp = U_w;
yaw_sp = -M_PI/2;

//error calculation
double e_r = roll_sp-roll;
double e_p = pitch_sp-pitch;
double e_y = yaw_sp-yaw;

/* PID Controller */
double roll_cmd = PID(Kpr,Kir,Kdr,I_acr,h,T_tr,p,e_r);
double pitch_cmd = PID(Kpp,Kip,Kdp,I_acp,h,T_tp,q,e_p);
double yaw_cmd = PID(Kpy,Kiy,Kdy,I_acy,h,T_ty,p,e_y);



//saturation
//according to maximum angles AscTec homepage
//roll_cmd = min(max((-52*M_PI/180),roll_cmd),(52*M_PI*180));
//pitch_cmd = min(max((-52*M_PI/180),pitch_cmd),(52*M_PI*180));

//scaling for correct output : -100% to 100% ==> 0...200
double roll_cmd_corr=0.907571211*roll_cmd+100;
double pitch_cmd_corr=0.907571211*pitch_cmd+100;
```

```
    std::cout <<"PHI:" << roll << ", SP:"<< roll_sp<<std::endl;
    std::cout <<"THETA:" << pitch  << ", SP:"<< pitch_sp<<std::endl;
    std::cout <<"PSI:" << yaw <<std::endl;
    std::cout << std::endl;
    std::cout <<"PHI_CMD:" << roll_cmd_corr <<"
original:"<<roll_cmd<<std::endl;
    std::cout <<"THETA_CMD:" << pitch_cmd_corr <<"
original:"<<pitch_cmd<<std::endl;
    std::cout <<"PSI_CMD:" << yaw_cmd <<std::endl;
    std::cout << std::endl;
    std::cout <<"p:" << p <<std::endl;
    std::cout <<"q:" << q <<std::endl;
    std::cout <<"r:" << r <<std::endl;
    std::cout << std::endl;
    std::cout <<"x:" << x <<std::endl;
    std::cout <<"y:" << y <<std::endl;
    std::cout <<"z:" << z <<std::endl;
    std::cout << std::endl;
    std::cout <<"I_acp:" << I_acp << std::endl;
    std::cout << std::endl;

    msg.x = pitch_cmd_corr;
    msg.y = roll_cmd_corr;
    msg.z = 0;
    msg.yaw = 98;
    msg.v_max_xy = -1;
    msg.v_max_z = -1;
    pub.publish(msg);

           if (!ros::ok())
            return 0;
            f.sleep();

    }
    //turn off motors
    req.startMotors = 0;
    ros::service::call("fcu/motor_control", req, res);
    std::cout << "motors running: " << (int)res.motorsRunning <<
std::endl;


    return 0;

}
```