AUTOMATIC CONTROL
LABORATORY (LA)
Station 11
CH-1015 LAUSANNE

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Spring Semester 2015-2016, **MT Master Project**

Model Based Control of Quadcopters

Author: Martí POMÉS ARNAU

Professor: Colin JONES

Assistants: Altug BITLISLIOGLU and Tomasz GORECKI

Jully 25, 2016

Acknowledgments

I would like to express my gratitude to Professor Colin Jones for giving me the opportunity to do my master thesis in his laboratory and allowing me to participate in the project. I will also like express my gratitude to my assistants Altug Bitlislioglu and Tomasz Gorecki for their support during the whole project both in technical and theoretical matters. No doubt their help regarding the improvement of my knowledge in automatic control has been key in the development of this project, as I hope it will be in my future.

Abstract

This project sets its objectives in the construction of a quadopter, in obtaining its mathematical model and the development of a linear MPC with the intention of a future implementation of this controller inside the onboard computer for a real time control of the position of the drone. First a general look is taken into the parts that form the drone and its software with practical observations directed to ease the introduction to future students that may continue with the project. Then the drone and its attitude controller are modeled and a linear MPC is implemented in simulation in order to test its feasibility for a future implementation. Both, setpoint tracking and trajectory tracking are tested. Possible improvements to the linear MPC, like the usage of sequential linearization, are presented in order to solve some of the constraints that a linear MPC may have when dealing with a non linear system.

Contents

Chapter 1: Introduction

The usage of drones in the industry, research and hobby has increased during the last years. In the world of academia (that includes this project) drones present a perfect test bench due its dynamics and many fields inside robotics and automatics use them in order to implement new control techniques, collision avoidance, machine learning algorithms and other.

The Automatic Control Laboratory (LA) at the EPFL has been working with them in a series of projects carried by students. Its main goal is the usage of Model Predictive Control (MPC) to control autonomous vehicles, not only drones (as it is the case of this project) but also motorcycles, cars or boats.

The current report corresponds to the last installment of this long term project. In precedent repots the CrazyFlie drone, a small and cheap quadcopter, was used. For this drone several features where developed in the laboratory like a PID for attitude control, an extended kalmann filter (EKF) and finally a non-linear MPC (NMPC) for position control. But the CrazyFlie had its limitations most importantly on the capacity of its onboard computer, which made impossible to run the non-linear MPC in real-time on board. The controller made its calculation on a land computer and transmitted the results to the CrazyFlie. This presented several problems like delays between the drone and the land computer plus it meant that the drone wasn't really autonomous. This is why for this report it was decided to start again with a new platform that could sustain onboard resolution of the optimal problem in real time.

The solution that was proposed was the usage on the computer Raspberry Pi (RPi) combined with the board Navio2, produced by the company Emlid, that creates an autopilot board (similar to more common boards like Pixhawk or APM) with all its features (IMU, pressure sensors, pwm input and output ports, etc.) but with the flexibility of a computer with Linux OS. Those two elements allows the usage of the, already developed, APM's autopilot software (referred in this project as ardupilot). This software includes some of the features developed for the old project such as an attitude controller and an EKF, allowing us to advance a few steps further instead of starting from scratch. That also presented a new challenge, that is, how to implement the position controller in coordination with the ardupilot software. Also, the new onboard computer (RPi+Navio2) required a new platform that could carry their weight. This new drone would be bigger and more powerful than the CrazyFlie but at the same time should remain in a reasonably small size so that it could easily fly, if

necessary, in confined spaces and its pieces are to be bought separately so that if anything fails it is easily replaceable and also would allow to repeat the structure of the drone so that we build more than one drone, since a long term objective of the project is to use several drones at the same time.

Although the RPi offers more computational power than the CrazyFlie computer, it is still limited and far from the one of a ground computer like the one who used to run the NMPC. That is the reason why this report makes a step backwards and tries to implement a linear MPC which solve a much easier optimal problem and could be more easily implemented and run inside the RPi.

The project has followed a similar course to the one that follows this report. The first months were dedicated to the building and testing the drone. After that some time was dedicated to the study of the autopilot software. Then the system was modeled, including the dynamics and the autopilot's controller, and the optimization problem was formulated and a linear MPC was obtained. Model and optimization problem were tested in simulation and with the results some improvements are proposed, and different configurations of the MPC are tried. Chapter 5 includes some of the experimental work that was done in parallel with the tests in simulation, mostly related with system identification.

It is important to remember while reading the part of the development of the controller that the objective is always to generate result that could be implemented in the RPi. More sophisticated methods of control (like the NMPC that was present in previous reports) have been left aside because they will be more difficult to run in the RPi.

Chapter 2: Hardware and Software of the drone

This chapter comments on the hardware configuration of the drone and the software already implemented in the on board computer. It shall be taken more of a practical guide and it is expressly addressed to a students or other interested people who may continue with the project. During my work with project I expended an important amount of time to the construction of the quadcoter and to the understanding of APM's code. Most of the people that work with quadcoters do it as hobby and most of the practical information needs to be taken from hobby forums, so the documentation is often outdate or somewhat unreliable. The majority of the people that do research with quadcopters normally don't present technical issues on their reports. Since this isn't a standalone report but the continuation of previous projects and the probable predecessor of many more it is interesting to give some information on technical matters that also reflect the time expended on such issues. The rectangles in color beige present some useful information regarding details of construction of this specific drone or some problems or doubts faced during the construction. This section also includes the explanation of the Motion Capture System and how it works. It is a tool that, although barely being used during this project, it will be useful for the future development of this project.

2.1. – The drone specifications

At the beginning of the project no clear specifications where given regarding the building of the drone, other than: it should be able to carry the onboard computer, it should have an average size and it should have a reasonable flying time. This leave a big margin to choose the components of the drone. The following table shows some of the characteristics of the actual drone. It can be used as a table of minimums, so that any changes made at least should be tried to maintain or improve some of these characteristics:

Computational Power	1.2GHz quad-core
Flight autonomy (*)	13-15min
Weight	550g
Thrust per Motor (**)	521g
Size of the frame	260mm (205X160)
Propeller's size	6X4.5 inches
Method communicating commands	PPM
Number of PWM outputs/minimum	14/4

(*) Result calculated for a normal usage of the drone
(**) Value given by the manufacturer for a 6X2 propeller

These are some basic values. They are explained in more detail in the following subsection.

2.2. - Hardware and basic electronics

When referring to the hardware it means all the physical objects that form the drone. The following is a list with the parts present in our drone:

- **Frame:** the skeleton of the quadcopter. All the other elements are attached to it. In past projects, the Crazyflie Nano Quadcopter 2.0 was used. With all the elements already incorporated into it, it was an easy to use platform, light (27 g), small (92 mm of diameter), somewhat resilient but without much processing power. In order to fit the new objectives, the most important of which is being capable to hold the Raspberry Pi, a new frame has been chosen. The LUMENIER QAV-R FPV is a 260mm diameter carbon fiber frame (the are other models with the same name but smaller sizes). Relatively light for its size (113g) it is an averaged sized frame (in terms of the sizes used in the hobby), big enough to hold the onboard computer on top and the battery inside its central structure, something essential in order to ensure that the weight of the drone is as centered as possible (the battery being the most massive component). The shape of the frame must be taken into account in order to properly model the quadcopter. Unlike the precedent frame and most of the X shaped frames, the frame is not symmetrical, i.e. the motors (at the end of the arms) are not forming 45° angles with the axes but they are closer to the y axe than to the x axe (the x axe being the angle that faces what could be called the "forward direction") forming a rectangle with a width of 205mm and a height of 160mm. This will cause the quadcopter to behave differently when trying to move in one direction or the other. It will be faster when moving in the y axe because it can roll easily (motors further from the y axe means that they can generate higher torque on the y axis). This must be added into the model and also in to the APM code that assumes that the X shaped frame is perfectly symmetrical.

  One practical observation to make regarding this frame is the lack of a proper landing gear other than 4 neoprene pads that come with the frame. They have the advantage of being extremely light (0.25g each), cheap and deformable, so they can soften the landing. On the other hand they are too short and that means that the propellers start and end very close to the ground (it augments the *ground effect* during the takeoff and landing) and if the landing is not straight the tips of the propeller may touch the ground before the pads and it will cause the drone to flip and crush into the ground and provably breaking the propellers.

It would be interesting to implement in the future a simple landing gear that would lift the base of the drone higher from the ground.

- **Motors and propellers:** The brushless motor DYS BE1806 2300KV is the one being used in combination with 6x45 propellers. The 2300KV represent the number of revolutions per minute that the motor will turn when 1V is supplied (without load, i.e. the propellers). That number by itself doesn't tell much but the manufacturer will normally provide a table with the lift that the motor with a certain propeller and a certain battery can provide. It is recommended by the hobby forums that every motor should be capable to lift half of the total mass of the drone (in the case of a quadcopter). These motors are more than capable of lifting twice this amount with the battery and the propellers that are being used. The size of the propeller is the one recommended by the size of the frame. 6x45 simply means 6 inches of diameter and 4.5 inches of pitch. In theory a smaller 5'' (referring to the diameter) propeller should be sufficient to lift the drone (using the manufacturers table) but real test during the project have shown that they don't perform as well as expected. A lower pitch could be used.

When buying new propellers is important to know that the motor doesn't make any difference between those that turn clockwise and counterclockwise: all the propellers go screwed in the same direction so if you buy a propeller that includes its own nut make sure that they aren't screw in different senses. Also, remember that you will need propellers that turn in both senses…

Item No.	VOLTAGE	NO LOAD		ON LOAD			EEP	LOAD TYPE
		CURRENT	SPEED	CURRENT	Pull	Power		Battery/prop
	V	A	rpm	A	g	W	%	
BE1806-13 (2300KV)	7.4	0.6	17060	6.5	390	48.1	8.1	LiPox2/6X2
				7.6	521	56.2	9.3	LiPox2/7X2.4
	11.1	0.6	25590	6.5	390	72.2	5.4	LiPox3/5X3
				7.6	521	84.4	6.2	LiPox3/6X2

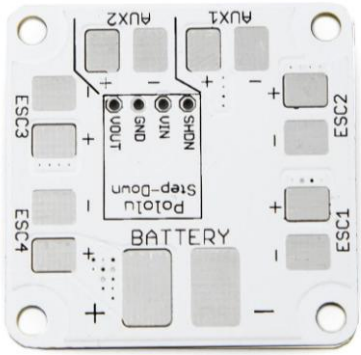
Figure 1: Table with the specifications of the motors provided by the manufacturer. Often it doesn't show results for the size of your propeller (as it is the case). If the value works for you with smaller propellers it will also works for yours.

- **Speed controllers (ESC):** The speed controller acts as the interface between the power circuit (connected to the battery) and the control circuit (signals coming from the Navio2 board). It transforms the PWM low voltage signal from the onboard computer into a signal powered by the battery that the brushless motor can understand and be powered by. The ESC must be select in concordance with the motor they are going to be powering and the battery. It is indispensable to secure that the maximal intensity demanded by the motor can be supported by the speed controller and it also can support the maximal voltage from the battery. For this project 4 16A opto ESC that support 2S to 4S batteries have been selected. The motor has a pick intensity of 8A and the battery used is a 3S. Opto means that it's only doing an optical isolation between the power circuit and the control circuit. There are other types of ESC that allow you to supply 5V through the PWM signal cables (connected to the board) that would allow to power the Raspberry Pi from one of the ESC by just connecting one of the PWM ports of the Navio2 with one of the ESC's. That is not the case so it requires another way to transform the 12V supplied by the battery into 5V that RPi can use.

The motors and the ESC chosen can be bought together. The ESC's and the motors are connected through three cables. Because two motors need to turn in one the direction and the other two on the opposite, you need to connect the to ESC differently. Just make sure that one of the motor cables is connected to the same cable of the ESC's and switch the order of connection of the other two cables. Don't worry to know if the motors are going to turn in one sense or the other, you can just rotate their position on the drone or configure the autopilot so that it understands where are position the motors sense-wise. The only mandatory constrain is that the motors that turn on the same sense need to be on the opposite side diagonally.

- **Power distribution board:** Dedicated to distribute the power supply from the battery to the ESC and other elements that need to be powered. It's just a tool to simplify the connection of the cables to the power source. It doesn't have any effect.

The frame comes with a power distribution board included that can be perfectly screwed to the base of the frame. Make sure to solder correctly the ESC's to the power board (red cable positive, black cable negative). Otherwise it will quickly burn the ESC's (literally) and make them unusable.



- **Battery:** Choosing the correct battery is an important part because its choice is constrained by other elements and its properties. This project uses a 3S 2200mAh battery. 3S means the number of cells of 3.7V that the battery contains, in this case 3, so the nominal voltage of the battery is 11.1V. When choosing a battery make sure that the elements can support this battery. The 2200mAh represents the capacity of the of the battery so if you know how much intensity is require by the elements that are being powered then you can calculate how much time will it take to empty the battery. In practical terms this value is related to the flying time (autonomy of the drone).

Since actual consumption is unknown because it depends how demanding we are with the motors (them being the most consuming elements) and the battery can't be fully emptied, there are several calculators on the Internet that would give you an approximation of the flying time. The approximate autonomy of the drone is about ***15min****, and although it is still limiting, it doubles the autonomy of the Crazyflie.*

More cells would mean more voltage and therefore the motors would be able to produce more thrust. More capacity would mean more flying time. But an increase on those two properties would also means an augment on the size and mass of the battery. The last term is the most constraining, being the battery the most massive piece of hardware on the drone. If the objective is to maximize the flying time, a bigger battery (with more capacity) may require bigger motors to be able to lift the drone, motors that would provably consume more and therefore reduce the flying time undermining the effect of the increase of the capacity. It's a matter of finding a compromise solution in order to maximize autonomy and minimize the mass.

It is import to be aware that the battery can't be completely discharged because then it will be impossible to be charged again. Plus, while the battery starts to get discharged, the voltage supplied starts decreasing and the performance of the motors will diminish. Never allow the charge to get under 10V. This is a safe value that will ensure a long life and a good performance of the battery. It can get under that value without major problems but it will diminish the life of the battery (number of cycles that it can be recharged) and if it gets too low the charger won't allow the battery to get charged. When charging the battery set the charging intensity to a value not higher than 2A because higher values will also diminish the life of the battery. The size is also important since this type of battery fits perfectly in the middle of the frame structure; a smaller battery would require some way to be fix in place and a bigger wouldn't fit.

- **Radio transmitter and receiver:** The final objective of the project is to have autonomous drones. Having a way to manually command the drone can be useful during the development phases and as a failsafe as well as being the easiest way to send simple messages to the drone like arm or disarm the motors or change the flying mode. During this project the TURNIGY 9X 2.4GHz transmitter and receiver have been used. One import characteristic of this transmitter is that can send the messages in a PPM signal that is the only type of signal that the Navio2 can read (through the PPM port). The problem is that the receiver can receive the signal but it can only transmit it (to the Navio2) in form of PWM, having eight outputs, one for every channel encoded inside the PPM signal. Therefore an encoder PWM to PPM is required in order to translate the eight PWM signals from the receiver back again into a single PPM signal. This solution requires adding a new element to the system. Normally the receiver would be powered through the Navio2 but in this case it is the encoder that is connected to the board so the receiver requires another source. The power will be supplied by a BEC (Battery Eliminator Circuit) that basically acts like a 5V volt regulator and it goes connected to the power distribution board with the ESC.

  The receiver needs to be bound to the transmitter (information on how can be found on the Internet) and the encoder require to be configured to PWM to PPM mode (there are other modes and how to do it can be found in the commentaries for this product in HobbyKing).

- **Onboard computer:** As it has been mentioned the Raspberry Pi 3 (RPi2 also works) is being used as the onboard computer in combination with the Navio2 board. The RPi has a 1.2GHz quad-core CPU and runs real-time linux and the ardupilot software. The Navio2 includes some sensors and ports necessaries to transform the RPi into an autopilot unit. Some of this elements that the Navio2 provides are:
  - Dual IMU (accelerometer plus gyroscope)
  - Pressure sensor (measures altitude with a 10cm resolution)
  - PPM input port (for the commands of the manual controller)
  - 14 PWM output ports (for the motors)

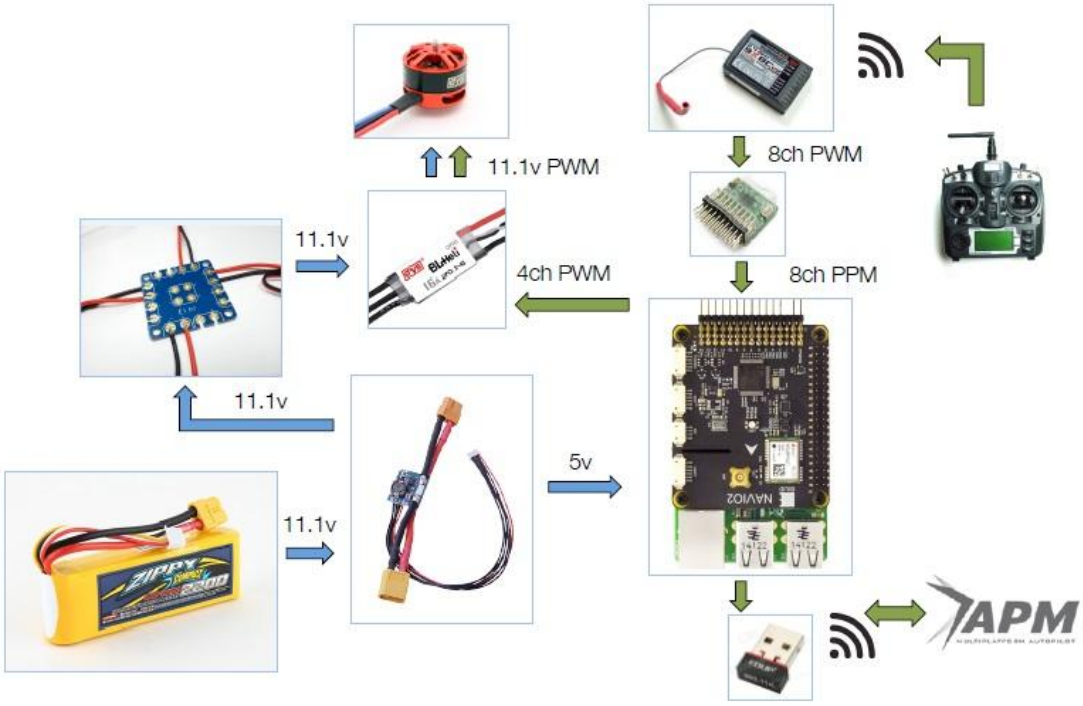
- ADC, I2C and UART ports (for extra sensors, you can use the UART to connect the Xbee radio receiver)

Details on how they operate are given in the following Software chapter. In terms of hardware: the board goes attached on the top of the frame's structure fitting inside the bottom half of a Raspberry Pi case that is screwed to the frame. This is an improvised solution and it doesn't ensure that the RPi is perfectly positioned on the middle of the frame.

It might be recommendable to design a proper support. The half case allows access to the Navio2 board and also allows to connect the peripherals to the RPi. Take off the SD Card in order to insert or extract the RPi. The front of the drone is defined by the computer's orientation, being the front on the opposite side to the PWM ports. During the flight the board is powered by the battery through a power module manufactured by EMLID that is connected between the battery and the power distribution board with cables that supply 5V directly connected to the Navio2. During test on the ground it is important to power the RPi with its own power supply in order to save battery and it is also important to use only one power source (either the battery or RPi's power supply).

When trying to implement an MPC (or any other real-time application) the time it takes to run each new iteration (the sampling time) is essential. This time is related to the computational power of the computer. The computational power of the RPi is 12GHz quad-core. Although it may not give you the actual number of milliseconds it takes to solve the OCP, you can compare it with other computers. If the computer in which you are developing your code is a 3.3GHz 6-core (that is more than 3 times faster than the RPi) and it takes nearly 0.1s (10 Hz is the minimum rate to run our MPC) to solve the OCP, then you can be sure that the RPi won't be fast enough. Also take on account that the RPi would be running the MPC and the ardupilot at the same time so you need to make sure that it can run the optimization far below the sampling rate.

Here is a simple schematic to represent how the different electronics and signals are connected:



2.3. – Software

The program run by the RPi is called ardupilot and even though its code is commented and fully accessible to any developer it is badly documented specially if we consider that it contains more than 700000 lines of code, most of them written in C++. This sub-section doesn't pretend to be a full documentation of the code but a commentary that, I hope, will help those who follow this project to better understand and interact with the ardupilot code. Take in mind that the ardupilot code is a project in progress and constant change so some of the things mentioned in this section may have changed.

The whole program works around a scheduler that calls all the necessary functions periodically. We can find a more specific description of how it works in the documentation. The scheduler is found in the file ArduCopter.cpp. Here you can find all the functions that are going to be call periodically while the program runs, like the actualization of the pwm values for the motors or the generation of logs for the different sensors or parameters. A function can be called independently by the scheduler or can be added inside one of the loops that exists at several different frequencies. There are slow loops at 1Hz to fast loops at 100Hz. You can also change exiting functions from one loop to another but you need to take on account that then this loop may take more time to be completed (the maximum time that a function or loop can take to finish is defined in the scheduler in microseconds).

One interesting change to make is to change the functions that creates the log for the pwm values send to the motors and the pwm values received from the manual control from the 10 Hz log loop to 25 Hz log loop. This is a minor change but a valuable one if you are interested in using the log that the ardupilot generates. For example, in the case of system identification, 10Hz may not give enough points to properly identify the system. You can add them to a faster loop but that could cause timing problems if the logging takes to much time. If the scheduler is not capable to run at the proper frequency it will generate an error and the ardupilot program will stop.

Inside the folder ArduCopter you will find all the functions related to the class Copter defined at the header Copter.h. The functions are the most basic to run the ardupilot code. The most important to know are the ones that define the flying modes (called by *control_nameofthemode.cpp*), the parameters function, the attitude function (Attitude.cpp) and the radio function (radio.cpp). At least those are the ones that have been used or studied in order to model the behavior of the drone. To understand how the flying modes work is important since it could be that in order to implement the MPC you need to create a new mode.

How to create a new mode is explained in the documentation although it is not updated. The easiest way is to create a new mode following the example of another that already exists and that may have a similar behavior to the new mode that is being created.

All the modeling that have been done in the following chapters is based on the behavior of the stabilize mode. It is the easiest to understand. The parameters function contains all the parameters of the ardupilot, their definition and the range of values that they can take. The attitude and the radio functions are important because they show you some of the transformations that the values of the inputs of the system (normally this would be the manual command) follow before being send to the attitude controller. If you want to substitute the manual controller by the MPC you need to add this transformations so that the attitude controller understands the signal that you are sending.

The radio control send pwm values between 1100 and 1900. Then the ardupilot transforms them into a value between 0 and 1000 for the throttle and a value between 45 and -45 degrees for the angles.
Ultimately the attitude controller takes angle values between 45 and -45 degrees and a throttle command with values between 0 and 1.

Inside the libraries folder you will find all the classes related to sensors or specific functionalities (like the PID attitude controller). Normally the flying mode will call a function of the attitude controller (depending on the type of PID used). There are several functions that can be used depending on the configuration of your inner controller (whether we are controlling angles or angular rates; from the body frame or the world frame). The configuration depends on the flight mode. You can see what function from the attitude controller is being called inside the control_flight_mode.cpp file mentioned earlier. After applying all the calculations the attitude controller will send its results to the motor functions. These functions will transform the results of the PID into a pwm values than the motors can understand. The attitude controller and the motors functions run separetly. They are called periodically by the scheduler. The method of sending values between them (and other functions from other libraries) is by updating the values of global variables that they both can read and/or write. When you see a variable that starts with "_" (e.g. _motor_roll) it means that it is a global variable. This system could be compare to the publishing/subscribing topics that ROS uses. When you see a variable that starts with "_" and it is written in capital letters that means that it is a parameter (global or local). Sometimes tracking down where those parameters or global variables are declared or what is their default value can complicated. The declaration of the parameters normally includes a small explanation, its default value, and the range of values it can take.

This is a very basic explanation of how the system works. On the libraries you will find commentaries. The documentation tries to explain how it works but it is outdated (but you should take a look on it anyway).

One minor change that has to be made in the code is the one referring to the geometrical configuration of the drone. As it has been said the frame is not symmetrical but the ardupilot doesn't have an option to change it so you need to go to the AP_motors_Quad.cpp and change the angles of the frame configuration that you are using so it matches the reality (the default value is 45 degrees).

Inside the libraries there are other functions that the ardupilot uses but they are not indispensable in order to understand the basic functionalities of the ardupilot code.

2.4. – The Motion Capture System

The implementation of the MPC for the position control requires a measurement of the actual position of the drone and its speed (attitude and angular speed are given by the Extended Kalmann Filter embedded inside the ardupilot code).

When controlling the drone manually these measurements are done by the human eyes of the user. Obviously an autonomous drone requires a different method. The method chosen is the usage of a motion tracking room. The room consists of 16 high precision cameras connected to a central computer that runs the Motion Capture System (MCS) software. Several indicators, in the form of retroreflective balls, are put over the drone. The tracking system requires that, at any given time, four or more indicators are visible to the ensemble of the cameras. The cameras have been calibrated so that only retroreflective objects are detected.

Use more than 4 indicators as a safety measure and make sure that they are not position symmetrically so that the cameras don't have problems to identify the object. Make sure that sure that the indicator doesn't move from their position

Once the object is prepared with the indicators and is place inside the scope of the cameras you can visualize the indicators on the screen of the central computer. Then you will need to declare to the Motion Capture System that those indicators form a rigid solid, and you will need to define its center of inertia so that the program can give you the position and attitude of the rigid body. Then you can start the live recording and the MCS will keep track of the drone.

The data of the recording is store in a log file once the recording is finished. This file can be open with MATLAB and it is useful when doing offline calculations (e.g. during System Identification). If you need that data in real time (the case of the implantation of the MPC) the MCS allows you to stream that data through its Ethernet connection. Using a Python code developed by other students of the EPFL you can received that data on your own laptop and then send it to the drone. The method of transferring that data from your laptop to the drone is still a matter of discussion but during the last weeks we have been trying to implement a radio link using the Xbee radio transmitter; and although there is still work to be done in order to make it operative, early results seem promising. Adding the serial communication with Xbee to the Python code that reads the data streamed is fairly easy and only requires a few lines of code and the installation of a serial communication library. The complicated part is reading the data with the drone and communicating the information to the autopilot/MPC.

You can run a simple python code in the RPi to see that it is more than capable to read the data broadcast by the Xbee on the laptop. The Xbee on the RPi can be connect either through the UART port or one of the USB ports. The real issue is to do that in real time, while running the autopilot and the MCS and be able to transmit the data to both. Using a python function inside the RPi is easy but python tends to be slower than C++ (time is always one of the biggest constraints) and all the code in the ardupilot is written using this language

The data generated by the MCS presents a 8 column matrix. The columns represent time, the 3D position and the attitude in quaternion. The time is set to zero when a new session is started. The frequency of sampling is higher enough for our usage (it's set at 250Hz). The position is given from the origin of the room (set approximately in the middle of the room at ground level). The axes are not the same as the ones that are normally used (being y the vertical axe). The axes can be change before starting the recording but the recommendation is to leave them as they are and post-process the results. The Python code integrates the transformation to a more common frame (with the Z axe vertical). The transformation from quaternions to euler angles needs to be added to the python code. Remember to start the flight with the drone facing the X axe.

Initially the attitude used by the ardupilot and the MPC would come from the EKF inside the ardupilot but we could also use the values calculate by the Motion Capture System if they seem more reliable or accessible.

The ensemble of the tracking room system and the drone should look like the following scheme (once implemented the transmission of data to the ardupilot):

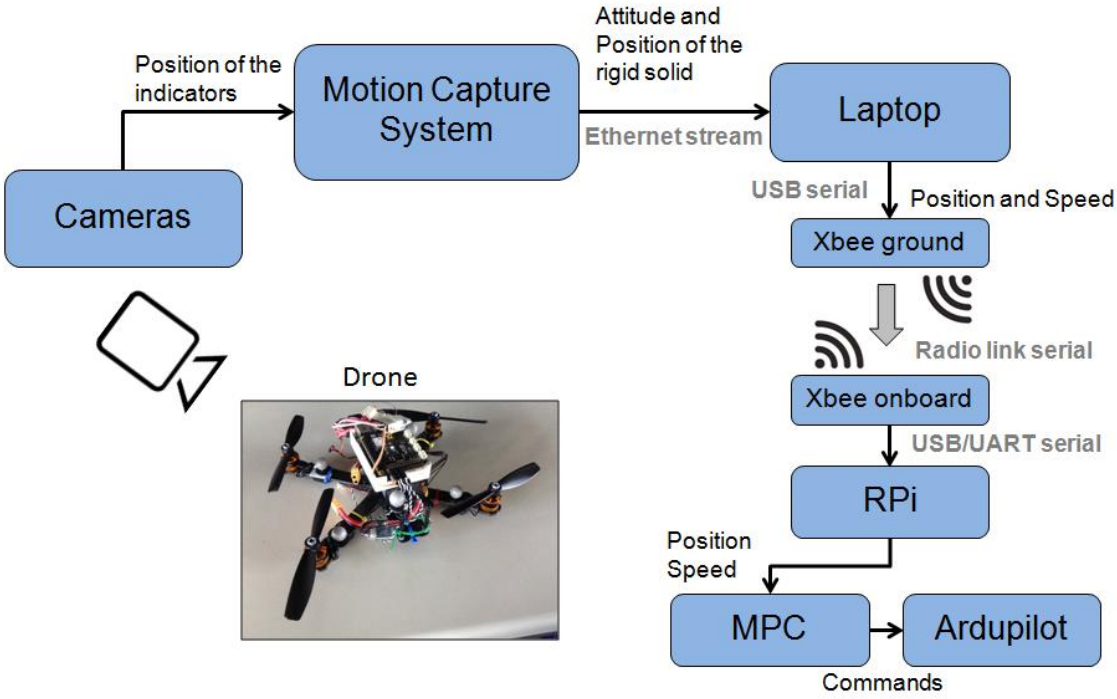


Figure 2: Scheme of all of the elements connected to the Motion Capture System. Includes the type of information transmitted and the type of communication

Chapter 3: Modeling of the system

This chapter follows the same path of precedent reports in terms of the modeling of the quadcopter, which at the same time follows the example of most of the literature, starting by defining the reference frames used in the modeling, followed by the representation of the attitude of the drone in those frames and the dynamics of the system. This first part of the chapter ends with the identification closed-loop model that includes the behavior of the PID-attitude control since this projects starts with the already programmed attitude controller of Ardupilot software and all the work of this project is made from outside the close-loop system.

3.1. - Reference frame

Two different references are used in this project. First, the fixed inertial frame called the World frame (W). The position of the origin of this frame depends on the method used to locate the drone. If the Motion Capture System is used then the origin will be found in the middle of the test room (depending on calibration). If GPS is used the origin will be in global origin of coordinates. A part from this fixed frame a rotating frame with origin on the center of mass of the quad will be used and called Body frame (B).

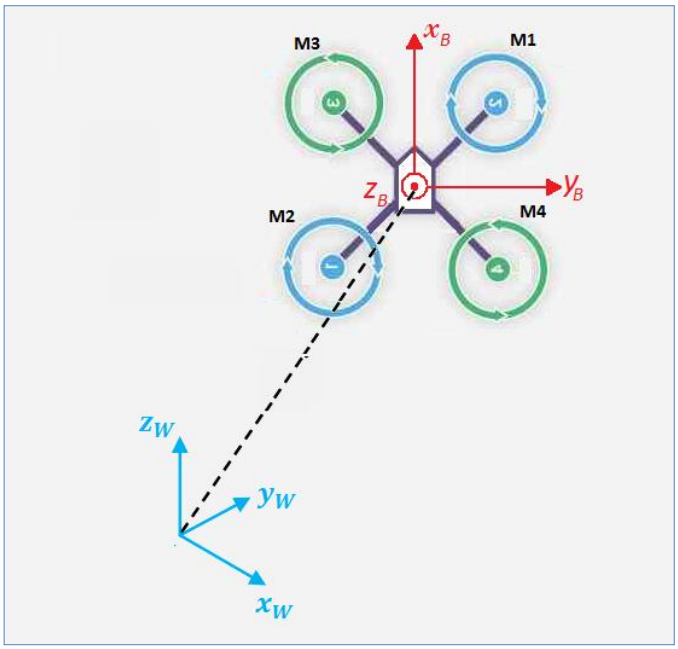


Figure 3: Graphical representation of the two reference frames used

The choice of the sense of the axes of the body frame and the numeration of the motor is made so that it coincides with the ones that the Ardupilot software uses. The two coordinate systems will be denoted as follows: [$\boldsymbol{x_W\, y_W\, z_W}$] for the world frame and [$\boldsymbol{x_B\, y_B\, z_B}$] for the body frame.

3.2. - Attitude representation

In order to orientate the body frame from the world frame the Euler angles representation will be used. Through a sequence of the following 3 right-hand rotations a vector can expressed in world frame can be expressed on the body frame:
- yaw angle (ψ) rotation around the $\boldsymbol{z_w}$ axis.
- pitch angle (θ) rotation around the $\boldsymbol{y_w}$ axis.
- roll angle (φ) rotation around the $\boldsymbol{x_w}$ axis.

For every rotation a rotation matrix can be defined:

$$\mathrm{R}_{\varphi} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos\varphi & \sin\varphi \\ 0 & -\sin\varphi & \cos\varphi \end{pmatrix}$$
$$\mathrm{R}_{\theta} = \begin{pmatrix} \cos\theta & 0 & -\sin\theta \\ 0 & 1 & 0 \\ \sin\theta & 0 & \cos\theta \end{pmatrix} \quad (3.1)$$
$$\mathrm{R}_{\psi} = \begin{pmatrix} \cos\psi & \sin\psi & 0 \\ -\sin\psi & \cos\psi & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Appling the sequence of rotations the rotation matrix from the world frame to the body frame can be obtained:

$$^{\mathrm{B}}\mathrm{R}_{\mathrm{W}} = \mathrm{R}_{\varphi} \cdot \mathrm{R}_{\theta} \cdot \mathrm{R}_{\psi} \quad (3.2)$$
$$^{\mathrm{B}}\mathrm{R}_{\mathrm{w}} = \begin{pmatrix} \mathrm{c}\psi\mathrm{c}\theta & \mathrm{s}\psi\mathrm{c}\theta & -\mathrm{s}\theta \\ \mathrm{c}\psi\mathrm{s}\theta\mathrm{s}\varphi - \mathrm{s}\psi\mathrm{c}\varphi & \mathrm{s}\psi\mathrm{s}\theta\mathrm{s}\varphi + \mathrm{c}\psi\mathrm{c}\varphi & \mathrm{c}\theta\mathrm{s}\varphi \\ \mathrm{c}\psi\mathrm{s}\theta\mathrm{c}\varphi + \mathrm{s}\psi\mathrm{s}\varphi & \mathrm{s}\psi\mathrm{s}\theta\mathrm{c}\varphi - \mathrm{c}\psi\mathrm{s}\varphi & \mathrm{c}\theta\mathrm{c}\varphi \end{pmatrix}$$

Where $\boldsymbol{c}$ and $\boldsymbol{s}$ are diminutive for cos and sin respectively.
The rotation matrix from the body frame to the world frame ($^{\mathrm{W}}\mathrm{R}_{\mathrm{B}}$) can be found by transposing the precedent matrix since is an orthonormal base.

In order to obtain the rotation matrix that relates the Euler rates vector $\dot{\Theta} = [\dot{\varphi}\ \dot{\theta}\ \dot{\psi}]^T$ to the body fixed angular rates ω_B=[p q r]^T a same sequence of rotations can be applied:

$$\omega_B = \begin{pmatrix} \dot{\varphi} \\ 0 \\ 0 \end{pmatrix} + \mathrm{R}_{\varphi} \left(\begin{pmatrix} 0 \\ \dot{\theta} \\ 0 \end{pmatrix} + \mathrm{R}_{\theta} \begin{pmatrix} 0 \\ 0 \\ \dot{\psi} \end{pmatrix} \right) \tag{3.3}$$

$$\omega_B = \begin{pmatrix} 1 & 0 & -sin\theta \\ 0 & cos\varphi & sin\varphi cos\theta \\ 0 & -sin\varphi & cos\varphi cos\theta \end{pmatrix} \begin{pmatrix} \dot{\varphi} \\ \dot{\theta} \\ \dot{\psi} \end{pmatrix} = T^{-1}\dot{\Theta}$$

If the matrix is inversed the inverse transformation is obtain (from body fixed angular rates to Euler rates):

$$\dot{\Theta} = T\omega_B = \begin{pmatrix} 1 & tan\theta sin\varphi & tan\theta cos\varphi \\ 0 & cos\varphi & -sin\varphi \\ 0 & \frac{sin\varphi}{cos\theta} & \frac{cos\varphi}{cos\theta} \end{pmatrix} \begin{pmatrix} p \\ q \\ r \end{pmatrix} \tag{3.4}$$

3.3. - Dynamics

The movement of the drone and its control is achieved by varying the angular speed of the four motors. The propellers attached to the motors will induce a force F_i and a moment M_i. Since the neither the motors nor the onboard computer have any way to measure the speed of the motors their speeds will be represented by the PWM value send to each motor and they will be considered proportional. The following figure represents the set of forces and moments that affect the drone:

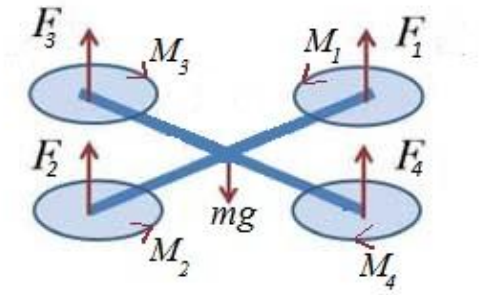


Figure 4: Scheme with the forces and moments that affect the drone

Again, the numeration of the motors and specially the sense of the moments have been chosen to coincide with the drone and ardupilot configuration. Following

the example of the precedent reports of the project, a quadratic relation can be established between the PWM and the forces and moments generated:

$$F_i = C_F PWM_i^2 \quad M_i = C_M PWM_i^2 \quad (3.4)$$

Where C_F is the lift coefficient and C_M is the drag coefficient, mostly depending on the motor properties and the propeller shape and size. Then the total values of the forces and moments applying to the quadcopter body can be calculated as follows:

$$\begin{pmatrix} u_1 \\ u_2 \\ u_3 \\ u_4 \end{pmatrix} = \begin{pmatrix} C_F & C_F & C_F & C_F \\ -0.8C_F L & 0.8C_F L & 0.8C_F L & -0.8C_F L \\ 0.6C_F L & -0.6C_F L & 0.6C_F L & -0.6C_F L \\ -C_M & -C_M & C_M & C_M \end{pmatrix} \begin{pmatrix} PWM_1^2 \\ PWM_2^2 \\ PWM_3^2 \\ PWM_4^2 \end{pmatrix} \quad (3.5)$$

Where u1 is the total thrust, u2 is the total roll moment, u3 is the total pitch moment and u4 is the total yaw moment. 0.8 and 0.6 represent the sine and the cosine of the angle between the arm and the $\boldsymbol{x_B}$ axe. Normally an X type drone like the one used would be symmetrical (in that case the 0.8 and the 0.6 values would be substituted by $\cos\frac{\pi}{4}$) but this is not the case of our drone and the different distances between the axes and the arms need to be taken on account. ***L*** represents the arm length.

Once defined the inputs of the system (forces and moments) the movement equations can be defined. But first the states of the model need to be identified. In this case, using the Euler-based representation of the orientation the following 12-dimenstional state vector is chosen:

$$X = [x \quad y \quad z \quad \dot{x} \quad \dot{y} \quad \dot{z} \quad \varphi \quad \theta \quad \psi \quad p \quad q \quad r]^T \quad (3.6)$$

- Position vector in the world frame $r=[x\ y\ z]^T$.
- Speed vector in the world frame $v=[\dot{x}\ \dot{y}\ \dot{z}]^T$.
- Euler attitude vector $\Theta = [\varphi\ \theta\ \psi]^T$.
- Angular velocity vector in the body frame $\omega_B=[p\ q\ r]^T$.

Now the dynamic model can be calculated using the rigid body mechanics defined by the Newton-Euler equations. The translational dynamics can be described in the following equation:

$$\dot{v} = g + \frac{u_1}{m} z_B$$
$$\begin{bmatrix} \ddot{x} \\ \ddot{y} \\ \ddot{z} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ -g \end{bmatrix} + {}^{W}R_B \begin{bmatrix} 0 \\ 0 \\ u_1/m \end{bmatrix} \quad (3.7)$$

Where ***m*** is the mass of the drone and ***g*** is the gravitational acceleration in the Earth surface.
The rotational dynamics can be described on the following equation:

$$\dot{\omega}_B = I^{-1}\left(\begin{pmatrix} u_2 \\ u_3 \\ u_4 \end{pmatrix} - \omega_B \times I\omega_B\right)$$
$$\begin{bmatrix} \dot{p} \\ \dot{q} \\ \dot{r} \end{bmatrix} = \begin{bmatrix} \frac{u_2 - qr(I_z - I_y)}{I_x} \\ \frac{u_3 - pr(I_x - I_z)}{I_y} \\ \frac{u_4 - pq(I_y - I_x)}{I_z} \end{bmatrix} \quad (3.8)$$

I represents the inertia matrix on the body frame. This matrix has been taken as diagonal. This is an approximation that has been taken before under the assumption of the symmetry of the drone used previously on this project (the Crazyflie). The actual drone is not perfectly symmetric but we will consider it to be an acceptable approximation, also for the sake of simplicity.

Taking on account 3.7 and 3.8 the open-loop model of the system will have the following form:

$$\dot{X} = f(X,u) = \begin{cases} \dot{x} = v \\ \dot{v} = g + \frac{1}{m} R_w^b(\varphi,\theta,\psi) u_1 \\ \dot{\theta} = T(\varphi,\theta,\psi)\omega_b \\ \dot{\omega}_b = I^{-1}\left(\begin{bmatrix} u_2 \\ u_3 \\ u_4 \end{bmatrix} - \omega_b \times I\omega_b\right) \end{cases} \quad (3.9)$$

3.4. – Incorporation of the inner controller to the model

The project uses the existing inner-controller of the Ardupilot code. This allows saving time that can be expended on the development of the MPC. The controller is the same as used by the stabilizing mode of Ardupilot, that is, a combination of

proportional and PID controllers for the angular position and the angular rates of the drone, respectively. The actual structure can be visualized in the following figure:

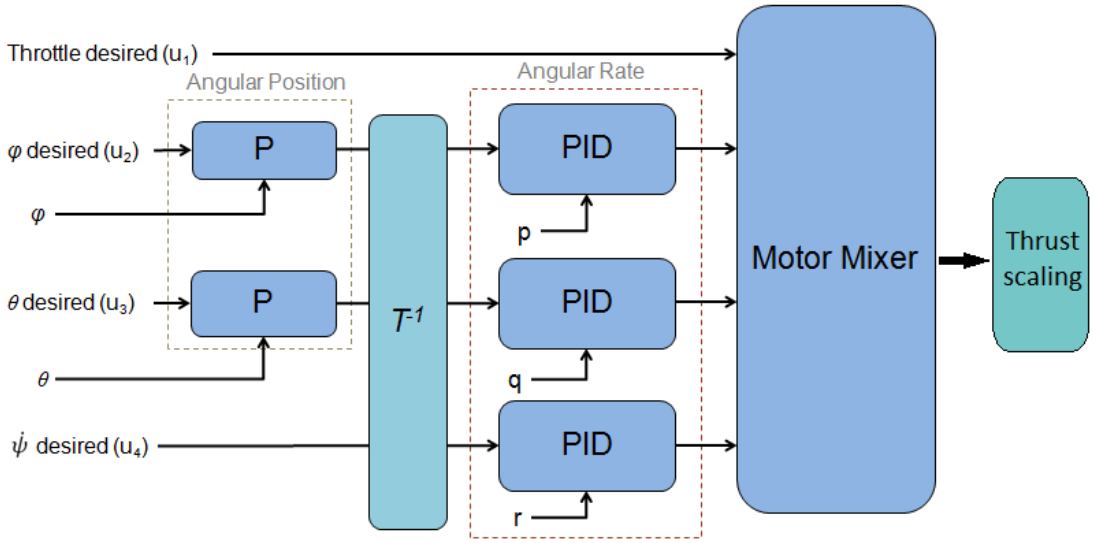


Figure 5: Representation of Ardupilot's attitude controller

Where $\boldsymbol{T^{-1}}$ is the rotation matrix that transform the euler angular rates to body angular rates. $\boldsymbol{u_i}$ represents the different inputs of the system as the Ardupilot code understand them. Normally those inputs would be provided by a manual command. In this project they'll be provided by the MPC, so that in terms of the inner controller there will be no difference between one form of command and the other. Ardupilot incorporates other types of attitude controller (normally used by other flying modes) e.g. the possibility of sending directly the desired rates for the roll and pitch as inputs. This project uses the one above and the general structure of the stabilizing mode because of their simplicity and similitude with the attitude controller used in precedent reports of this project and the literature.

The model also includes the motor mixer (using the same nomenclature as precedent reports). The motor mixer is the algorithm that transforms the result of the attitude controller into PWM signal for the motors and it does so by redistributing the signal into each motor (for example if rolling is required, the signal produce by the PID of the roll rate, shall be distribute so that the motors on the same side should receive the same PWM but this shall be different from the one received by the motors on the other side). The algorithm as it is implemented in Ardupilot is complex (difficult to transform into mathematical equations) because it tries to rebalance the signal and establish a priority between commands and incorporates several saturation functions to ensure that the result is inside the boundaries. The whole algorithm has been substituted by the following matrix, in an attempt to simplify the model:

$$M_{motor\ mix} = \begin{bmatrix} 1 & -0.8 & 0.6 & -1 \\ 1 & 0.8 & -0.6 & -1 \\ 1 & 0.8 & 0.6 & 1 \\ 1 & -0.8 & -0.6 & 1 \end{bmatrix}$$
$$PWM = M_{motor\ mix}\, u' \tag{3.10}$$

Where $\boldsymbol{u'}$ is the resulting signal of the PID's. It has been proved on simulation that the result of applying this matrix is identical to the application of the Ardupilot's algorithm in most case scenarios especially if the system is not being required to do extremely demanding maneuvers (e.g. fast takeoff sequence or pirouettes).

Finally the signal passes through a function called *Thrust scaling.* Its objective is to rectify the curve that relates the angular speed/PWM of the motor and the thrust it produces. How this function behaves can be set by changing some of the parameters of the Ardupilot (through the control ground station) and its full explanation can be found on the official web page for Ardupilot. Since in the theoretical model of the drone the relation between thrust and PWM is considered to be quadratic, the parameters can be chosen so that the *Thrust scaling* acts like a square root function therefore transforming the quadratic relation into a proportional one. The importance of this function has been proven through simulation specially because the usage of the linearize model for the MPC. Without the *Thrust scaling* big disparities appear between the predicted behavior by the linear MPC and the one that can be observed by the non-linear system.

3.5. - Modeling the Attitude controller

The proportional angle controller can be expressed in the following way:

$$rate_d = T^{-1}\left(KP \left[\begin{bmatrix} u_2 \\ u_3 \\ u_4 \end{bmatrix} - \begin{bmatrix} \varphi \\ \theta \\ 0 \end{bmatrix} \right] \right)$$
$$KP = \begin{bmatrix} KP_1 \\ KP_2 \\ 1 \end{bmatrix} \tag{3.11}$$

Where $\boldsymbol{rate_d}$ is the desired rate in the body frame. $\boldsymbol{u_4}$ is included in the formulation but is not affected by the proportional controller in order to simplify and reduce the number of equation of the model.

The modeling of the PID for the rate control is far more complex and it requires the introduction of 6 new states in order to be able to compute the integral and derivative parts of the controller. First of all a new set of 3 states (called $\boldsymbol{n_i}$, following the same nomenclature as precedent reports) are created that represent the integrated rate error:

$$\dot{n} = e$$
$$\begin{bmatrix} \dot{n}_1 \\ \dot{n}_2 \\ \dot{n}_3 \end{bmatrix} = rates_d - \begin{bmatrix} p \\ q \\ r \end{bmatrix} \quad (3.12)$$

The equation of the controller can be, then, expressed in terms of these states in a Laplace transformation of the PID:

$$u' = Ki\, n + Kp\, s\, n + Kd \frac{N}{1 + \frac{N}{s}}\, s\, n \quad (3.13)$$

Where ***Ki***, ***Kp*** and ***Kd*** represent the integral, proportional and derivative gains respectively; N is the gain of the filter for the derivative part. In order to implement this equation into the model it requires a reformulation and the introduction of 3 new states ($\boldsymbol{m_i}$):

$$A = Ki\, n + Kp\, s\, n \quad (3.14)$$
$$u' - A = Kd \frac{N}{1 + \frac{N}{s}}\, sn \quad (3.15)$$
$$(u' - A)\left(1 + \frac{N}{s}\right) = Kd\, N\, s\, n \quad (3.16)$$
$$u' = A + Kd\, N\, sn + N \frac{(u' - A)}{s} \quad (3.17)$$
$$s\, m = u' - A \quad (3.18)$$

Expressed out of the Laplace transformation:

$$\dot{m} = u' - A \quad (3.18)$$

$$u' = Ki\, n + Kp\, \dot{n} + Kd\, N\, \dot{n} + N\, m \quad (3.19)$$

Therefore the close-loop system can be represented mathematically as follows:

$$\dot{X}_a = f(X_a, u) = \begin{cases} \dot{X} = f(X, u_{int}) \\ u_{int} = M \begin{bmatrix} PWM_1 \\ PWM_2 \\ PWM_3 \\ PWM_4 \end{bmatrix} (*) \\ rate_d = T^{-1} \left(KP \left(\begin{bmatrix} u_2 \\ u_3 \\ u_4 \end{bmatrix} - \begin{bmatrix} \varphi \\ \theta \\ 0 \end{bmatrix} \right) \right) \\ \begin{bmatrix} \dot{n}_1 \\ \dot{n}_2 \\ \dot{n}_3 \end{bmatrix} = rates_d - \begin{bmatrix} p \\ q \\ r \end{bmatrix} \\ \begin{bmatrix} \dot{m}_1 \\ \dot{m}_2 \\ \dot{m}_3 \end{bmatrix} = \begin{bmatrix} u'_2 \\ u'_3 \\ u'_4 \end{bmatrix} - \begin{bmatrix} Kp_1\, \dot{n}_1 + Ki_1\, n_1 \\ Kp_2\, \dot{n}_2 + Ki_2\, n_2 \\ Kp_3\, \dot{n}_3 + Ki_3\, n_3 \end{bmatrix} \\ u' = \begin{bmatrix} u'_1 \\ u'_2 \\ u'_3 \\ u'_4 \end{bmatrix} = \begin{bmatrix} u_1 \\ Kp_1\, \dot{n}_1 + Ki_1\, n_1 + N_1 Kd_1\, \dot{n}_1 - N_1 m_1 \\ Kp_2\, \dot{n}_2 + Ki_2\, n_2 + N_2 Kd_2\, \dot{n}_2 - N_2 m_2 \\ Kp_3\, \dot{n}_3 + Ki_3\, n_3 + N_3 Kd_3\, \dot{n}_3 - N_3 m_3 \end{bmatrix} \\ PWM = M_{motor\ mix}\, u' \end{cases} \quad (3.20)$$

** The function Thrust Scaling has been applied so in reality what we would have is something like $u_{int} = M(\sqrt{PWM})^2$. The square root represents Thrust scaling applied to the output of the motor mixer.*

This system has a total of 18 states. When computing the model for the optimization problem a new state needs to be considered: gravity. Even though it is treated as constant, when calculating the jacobian to obtain a linear approximation of the non-linear system (as it'll be explained in the following section) it is interesting to have gravity as an state in order to end up the canonical affine form for the state space**:** $\dot{\boldsymbol{X}} = \boldsymbol{AX} + \boldsymbol{Bu}$ (otherwise $\dot{\boldsymbol{X}} = \boldsymbol{AX} + \boldsymbol{Bu} - \boldsymbol{g}$). This is not a must. It was done in order to easily introduce the linear system in Matlab and, also, in case that some OCP solver requires you to introduce the linear system in a canonical form. The introduction of gravity as state can also be uses to estimate its value. At the end the system has a total of 19 states with the following state vector:

$$X_a = [x \quad y \quad z \quad \dot{x} \quad \dot{y} \quad \dot{z} \quad \varphi \quad \theta \quad \psi \quad p \quad q \quad r \quad n_1 \quad n_2 \quad n_3 \quad m_1 \quad m_2 \quad m_3 \quad g]^T \quad (3.21)$$

Chapter 4: Formulation of the optimization problem

4.1. - Formulation of the optimization problem

For the control of the position of the drone an optimal controller is going to be used. More precisely a linear MPC is going to be implemented both in the simulation in Matlab and on the raspberry pi. First of all the optimization problem needs to be defined through a cost function to minimize. The cost function $\boldsymbol{J}$ can be written mathematically as follows:

$$\begin{aligned} \mathrm{J} &= J_e + J_u \\ J_e &= \int_0^t \vec{e}^T(\tau) \cdot Q \cdot \vec{e}(\tau)\, d\tau \\ \vec{e}(t) &= \overrightarrow{ref} - C\vec{x}(t) \\ J_u &= \int_0^t \vec{u}^T(\tau) \cdot R \cdot \vec{u}(\tau) d\tau \end{aligned} \tag{4.22}$$

As it is shown the cost function is divided into two terms $\boldsymbol{J_e}$ and $\boldsymbol{J_u}$ the first representing the precision cost and the second representing the energetic cost. They can be expressed in as the weighted module of the error and the command respectively, where $\boldsymbol{Q}$ and $\boldsymbol{R}$ are the weight matrixes and $\boldsymbol{C}$ is the state matrix that relates the output of the system y with the states of the system. The objective of the optimization problem will be to find the sequence of inputs that minimize the previous cost function taking into account the constraints imposed by the system dynamics and physical limitations of the controller:

$$\begin{aligned} \min_{\mathrm{u(t)}} \ \mathrm{J} &= \min_{\mathrm{u(t)}} (J_e + J_u) \\ \min_{\mathrm{u(t)}} \ \mathrm{J} &= \min_{\mathrm{u(t)}} \frac{1}{2} \int_0^t [\vec{e}^T(\tau) \cdot Q \cdot \vec{e}(\tau) + \ \vec{u}^T(\tau) \cdot R \cdot \vec{u}(\tau)]\, d\tau \end{aligned} \tag{4.23}$$

$$Constraints: \begin{cases} \dot{x}(t) = f(x(t), u(t)) \\ x(0) = x_0 \\ U_l \leq u \ \leq \ U_u \end{cases} \tag{4.24}$$

Where $\boldsymbol{x_0}$ is the initial state, $\boldsymbol{U_l}$ is the lower boundary and $\boldsymbol{U_u}$ is the upper boundary for the control signal. For now no limitations on the states have been included inside the optimization problem in order to reduce the complexity of the problem (some constraints will be introduced in the next subsections). This optimization problem will need to be discretized in order to be computable by and its results useful for the RPi, plus the system needs to be linearized to obtain a linear MPC. The linearization as it has been mentioned briefly earlier in the

report will be done by the computation of the jacobian of the non-linear system (first order approximation):

$$\begin{aligned} \dot{x} = f(x,u) &= f(x^*,u^*) + \left.\frac{\partial f}{\partial x}\right|_{x^*,u^*}(x-x^*) + \left.\frac{\partial f}{\partial u}\right|_{x^*,u^*}(u-u^*) + O(2) \\ \dot{x} &\approx f(x^*,u^*) + \left.\frac{\partial f}{\partial x}\right|_{x^*,u^*}(x-x^*) + \left.\frac{\partial f}{\partial u}\right|_{x^*,u^*}(u-u^*) \end{aligned} \tag{4.25}$$

Where $\boldsymbol{x}^*$ and $\boldsymbol{u}^*$ are the values for the state and command for whom the system is linearize. Since it's common to use an equilibrium point as linearizing point (that will be initially our case) the first term of the equation is equal to 0. If the following change of variables is made the system can be then express in the canonical space state form:

$$\begin{aligned} \mathbf{x} &= (x - x^*) \\ \mathbf{u} &= (u - u^*) \\ \dot{x} &= A\mathbf{x} + \mathrm{B}\mathbf{u} \end{aligned} \tag{4.26}$$

Once the system is linearized it can be discretized given a sampling time. We use a Zero-order hold in Matlab to obtain the state matrices. The sampling time has been chosen to be 0.1 s (MPC running at 10Hz), being slow enough to give time to the RPi to run the optimization and fast enough to ensure stability.

Now the optimization problem can be redefined as follows:

$$\begin{aligned} J_e &= \sum_{k=1}^{N} \vec{e}^T(k) \cdot Q \cdot \vec{e}(k) \\ \vec{e}(k) &= \overrightarrow{ref} - C\vec{x}(k) \\ J_u &= \sum_{k=0}^{N-1} \vec{u}^T(k) \cdot R \cdot \vec{u}(k) \end{aligned} \tag{4.27}$$

$$\min_{\mathrm{u(k)}} \mathrm{J} = \min_{\mathrm{u(k)}} \frac{1}{2} \sum_{k=0}^{N-1} [\vec{e}^T(k) \cdot Q \cdot \vec{e}(k) + \vec{u}^T(k) \cdot R \cdot \vec{u}(k)]$$

$$Constraints: \begin{cases} \vec{x}(k+1) = A\vec{x}(k) + B\vec{u}(k) \\ \vec{x}(0) = \overrightarrow{x_0} \\ U_l \leq u \leq U_u \end{cases} \tag{4.28}$$

Where $\boldsymbol{N}$ is the prediction horizon. Choosing this value is important for the correct tuning of the MPC. For this configuration a prediction horizon of 20 has been chosen (i.e. 2 seconds). A bigger prediction horizon would ensure a better stability and the security that the optimizer can find optimal solution inside the constraints. At the same time it would increase the computational time. N=20 seems a good compromise solution by proving to be (though simulation) a big enough time horizon but at the same time it seems small enough to help the RPi

to run the optimization at the desired frequency since this is most certainly one of the challenging parts of the practical implementation.

4.1.1 Soft constraints

As can be seen in the previous definition of the optimization problem there aren't any constraints impose on the states (apart from the initial conditions). That is made in order to ensure that the MPC can always find a solution and doesn't generate an error that may cause the crush of the drone. At the same time it has been observed that as it gets far from the linearization state (hovering position) less reliable is the linear system and less reliable are the MPC and its predictions, so, it is of our interest to ensure that the drone doesn't get out of a "controlling" zone without imposing too restrictive constraints. We introduce soft constraints as an intermediate solution between having or not having restrictive constraints. In order to simplify as much as possible the optimization problem they are only going to be applied onto the roll and pitch states that seem the most relevant. For that a new set of variables ($\boldsymbol{s_i}$) are introduced they are used to evaluate how much the drone trespasses the limits we've imposed:

$$Angle_{max} + s_i \geq X_i \quad (4.29)$$

Then those variables are add to the cost function with a weight matrix $\boldsymbol{T}$ with values several orders higher than the ones of $\boldsymbol{Q}$ and $\boldsymbol{R}$, so that it really punishes any solution that gets out of boundaries but that this solution can always be found. Therefore the new formulation of the problem is the following:

$$\min_{u(k)} J = \min_{u(k)} \frac{1}{2} \sum_{k=0}^{N-1} [\vec{e}^T(k) \cdot Q \cdot \vec{e}(k) + \vec{u}^T(k) \cdot R \cdot \vec{u}(k) + \vec{s}^T(k) \cdot T \cdot \vec{s}(k)] \quad (4.30)$$

$$Constraints: \begin{cases} \vec{x}(k+1) = A\vec{x}(k) + B\vec{u}(k) \\ \vec{x}(0) = \overrightarrow{x_0} \\ Angle_{max} + s \geq |x| \\ U_l \leq u \leq U_u \end{cases} \quad (4.31)$$

This optimization problem is a Quadratic program with 22 variables (16 if the system is simplified without the integral states of the inner controller). It has N equality constraints (N being the predictive horizon) plus 7N inequality constraints (3 soft constraints and 4 constraints for the values of u for every step). The required minimum sampling rate is of 5Hz.

With the addition of all the elements defined until now, including the MPC and the Motion Capture System, we can represent the whole close-loop system in the following diagram:

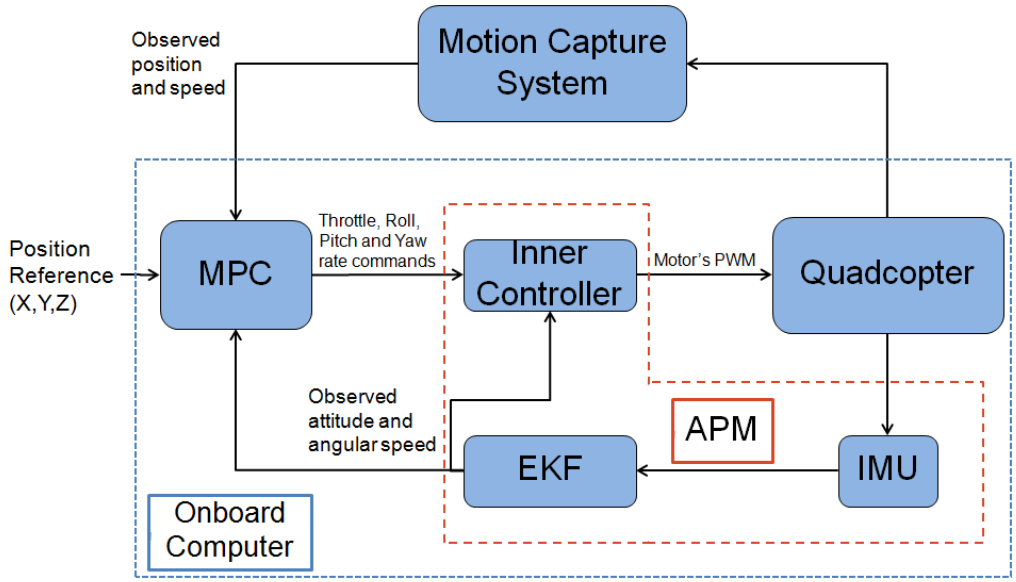


Figure 6: Representation of the close loop system

4.2. - Simulation Results

The simulation has been done through Simulink and Matlab. The Yalmip libraries have been used in order to solve the optimization problem. In a single Simulink model has been represented the different parts that intervene in the system: the MPC controller, the inner (attitude) controller and drone dynamics:

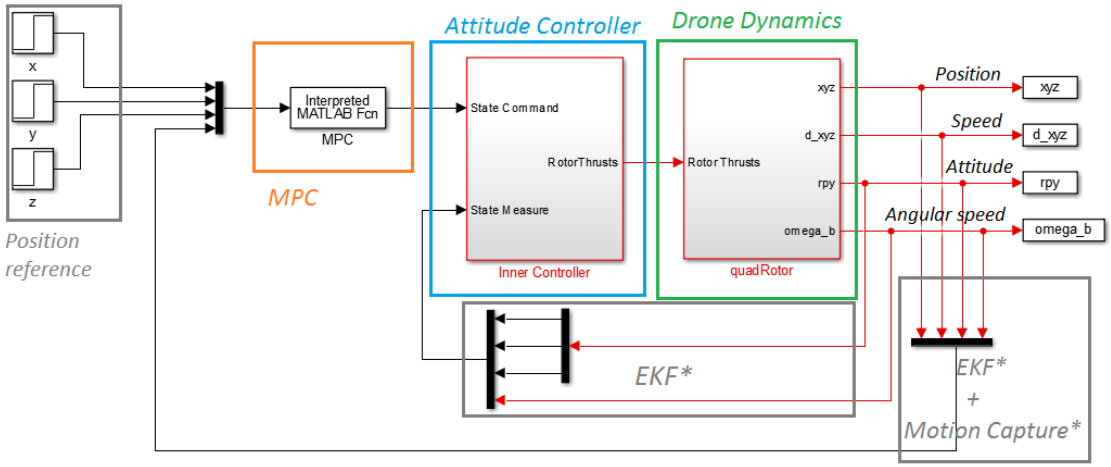


Figure 7: MATLAB Scheme of the drone with the MPC controller. (*) Both the EKF and the motion capture system are not represented on simulation. In the simulation the attitude controller and the MPC can read directly the states of the drone.

In these simple tests the drone is given a certain ending position (disregarding trajectories) and its performance is observe and evaluated in terms of time needed to reach the objective (with a 5% margin), potential overshooting, behavior of the controller and comparison between an angle command (when given) and the angle response of the drone.

For the first test the system is asked to take off (reach altitude of 1 m). This is the simplest operation the drone can be ask for and the where the linearized system is the most reliable since the drone doesn't change of attitude.

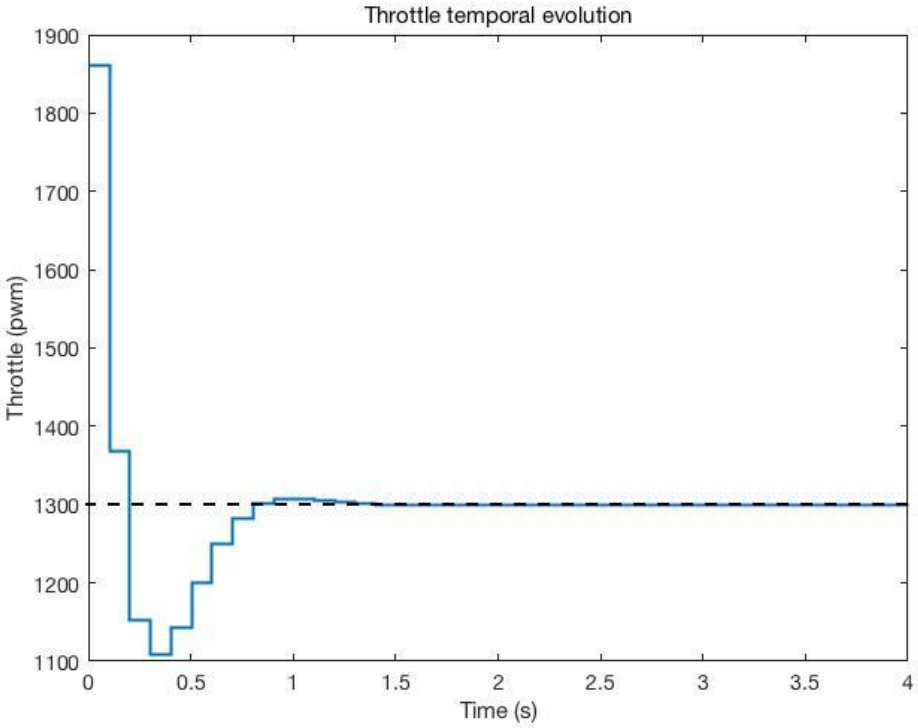


Figure 8

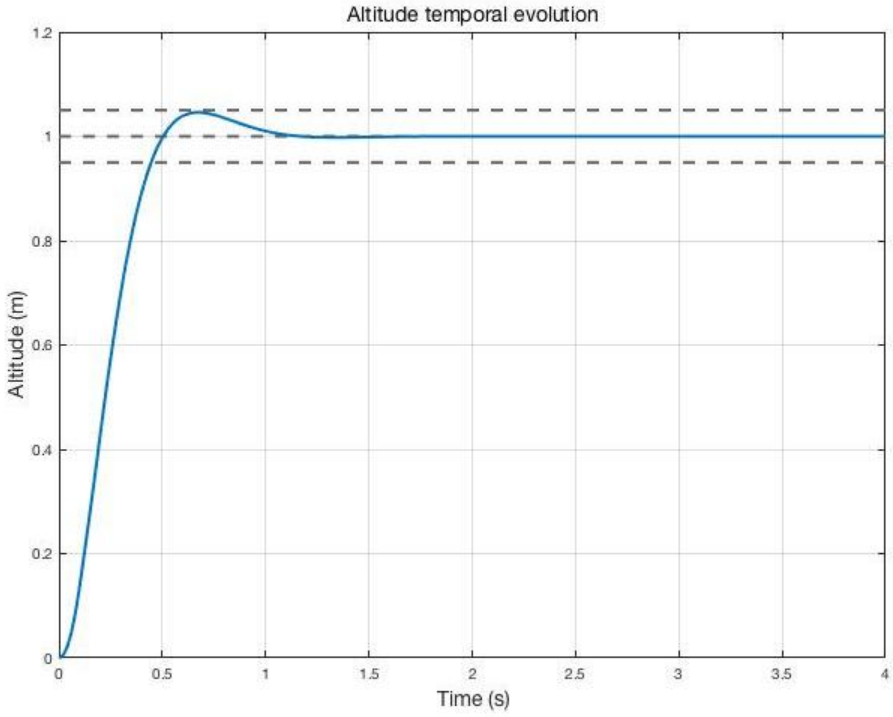


Figure 9

Observations: As it can be seen the system is extremely fast reaching the reference position in less than 0.5 seconds. System presents a slight

overshooting that doesn't exceed the 5% of the reference. Although it is not shown in the figures the simulation gives a peak velocity of 3m/s with peak acceleration during the first 0.1 seconds of 27.5m/s^2. These values are extreme and they correspond to the demands of the MPC which, as it can be seen, starts a nearly full throttle (max throttle being at 1900) and quickly passes to close to no throttle at 0.3 seconds from the start (min throttle being at 1100). This is undesirable, because it's not only demanding an aggressive behavior to the drone but at the same time it's demanding it close to ground. The *Ground effects* i.e. aerodynamical effects due to the proximity of propellers to the ground that prevent the proper flux of air through the propellers have not been modeled so the simulation doesn't take them on account and it's difficult to predicted the behavior of the quadcopter close to the ground especially when demanding full throttle. This can be solved (although the ground effects will always appear) by adding some constraints, which will be shown in the next section, the will force the MPC to take a more conservative approach.

For a second test, the drone is ask to reach the position (1, 1, 1) m, in order to observe how well it performs with a more complex command that involves a change in the attitude of the drone and therefore a distance from the linearization point:

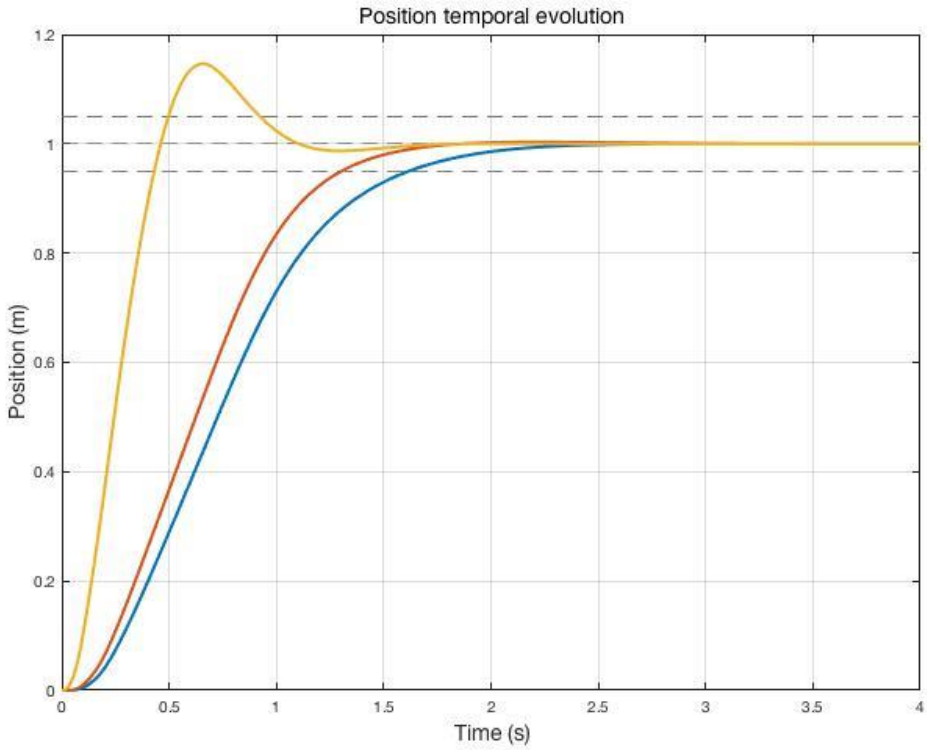


Figure 10: Orange, red and blue lines represent movement in the Z-axis, Y-axis and X-axis respectively

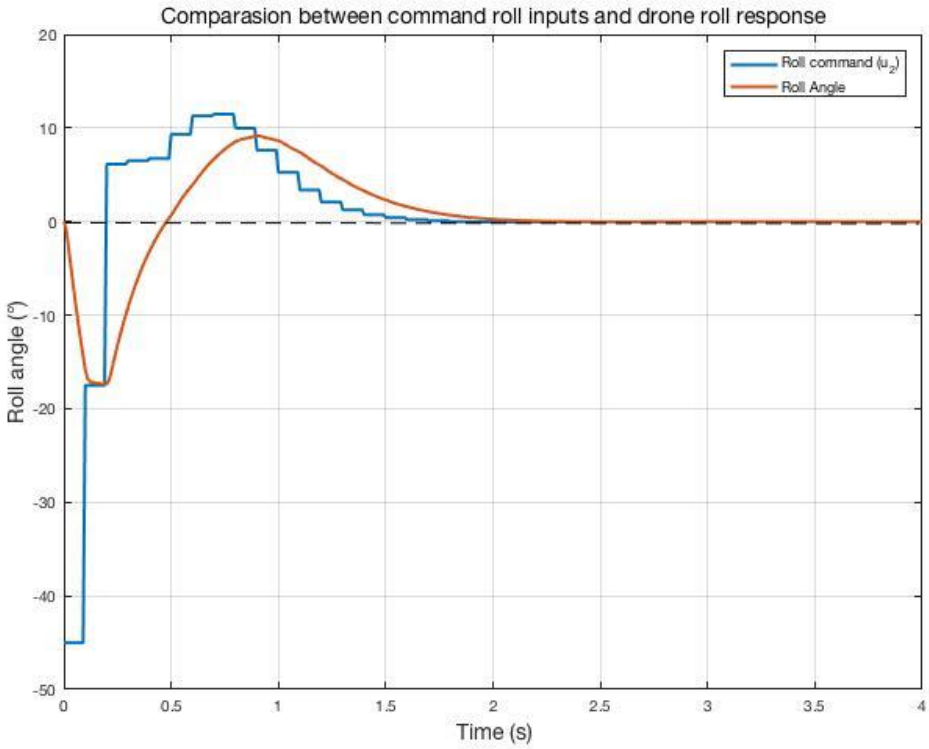


Figure 11: Result of the test

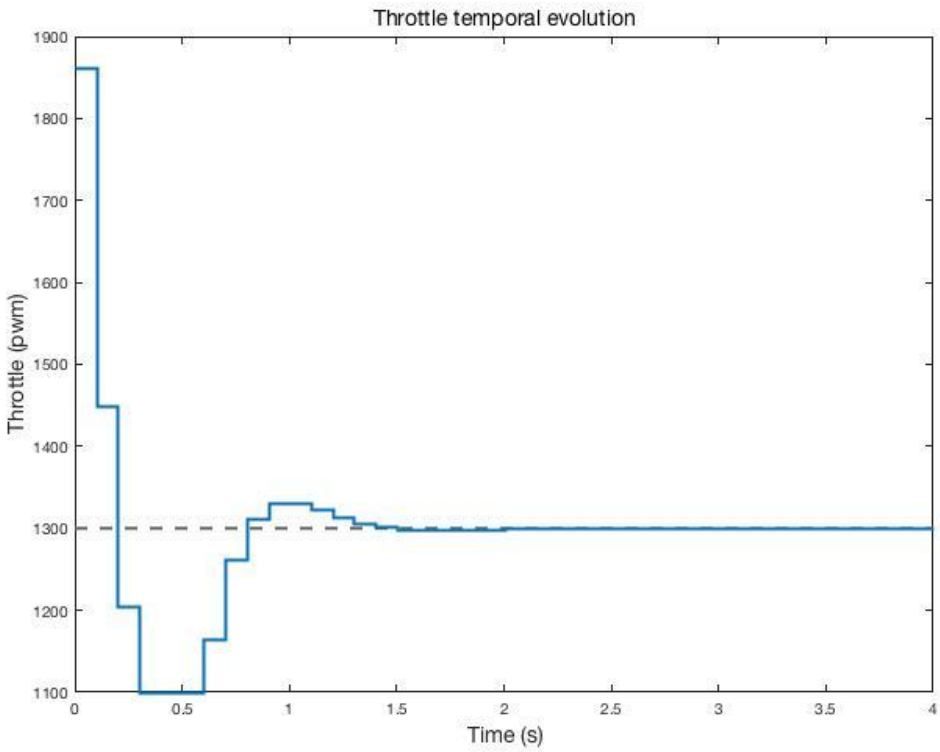


Figure 12: Result of the test

Observations: As can be observed the movement of the system in the Z-axis is much faster than the movement of the system in the horizontal plane, as it is to be expected since horizontal movement requires the turning of the drone. While this time the drone takes a little less than 1 second to stabilize in the Z-axis, it takes around 1.5 seconds to stabilize to its horizontal position. The movement on the Y-axis is slightly faster than the one in the X-axis and that is due to the asymmetry of the drone's configuration (not being a perfect X, with the motors closer to the Y-axis). In the comparison between roll command and roll observed, it is shown how the angle manages to follow the command with a certain (acceptable) delay due to drone's dynamics. The behavior of the throttle command is similar to the one shown for the precedent test. It has still an extremely aggressive stance and in this case produces an overshoot superior to the 10%. At the same time this aggressiveness allows for a faster movement in the horizontal axes even if the controller (due to the linear model) doesn't understand that.

In the last simulation the drone will be requested to make a big movement in Y-axis (3 m) while keeping its altitude after a theoretical takeoff. This test is made in order to show one of the disadvantages of the linearized system:

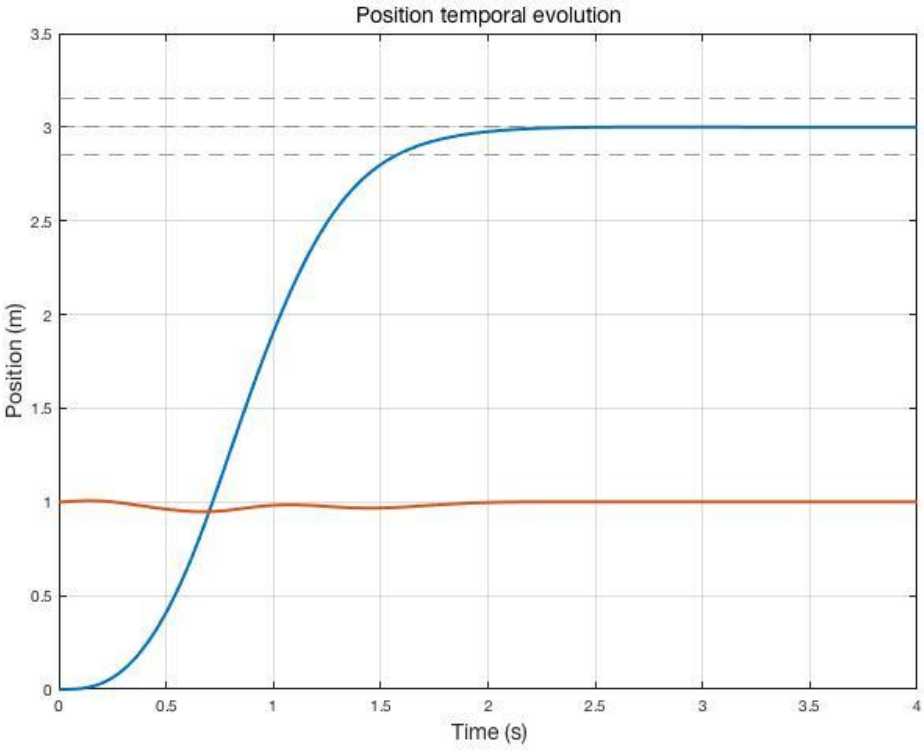


Figure 13: Blue and red lines represent the movement in the Y-axis and the Z-axis respectively

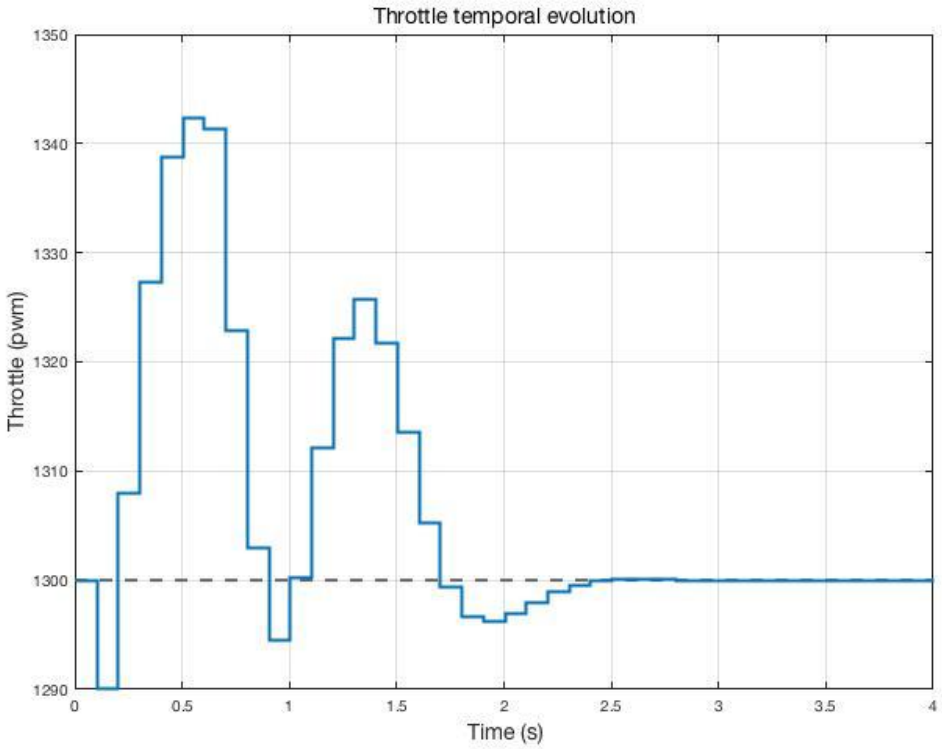


Figure 14: Result of the test

Observation: The problem using a linearized system is that the approximation is technically only valid for points close to the linearization point, in this case a hovering stance. For only vertical movements or small horizontal movements (that is movements that require small roll and pitch angles) the linearize system is a good solution. Although simulation shows that the quadcopter is capable of achieving positions out of the equilibrium with angles that surpass what would be consider "small angles", its behavior seems unrealistic and it poses the question if the results on simulation are a good representation of the reality. Plus the strategy used by the MPC controller in order to achieve distant positions presents significant differences from what it would be expected from a human user. A clear example would be when, after taking off, a large movement in the Y axe (for example 3 meters) is requested. A human user would turn a certain roll angle; apply a higher throttle to accelerate in that direction and then turn the roll angle in the other sense and decelerate to until reaching the position. The solution found by the optimizer is similar except for the second step. The drone doesn't know that if you are inclined a certain roll angle and you augment the throttle; this would produce acceleration in the Y axe. That is due to the linearized model. For the MPC an increase in throttle only produces acceleration in Z axe so it tries to reach the objective by just sending rolling commands.

4.3. – About the choosing of weight matrices

On the equation (4.30) 3 weight matrices are defined in order to control the behavior of the MPC controller: Q, R and T. As explained in the chapter 4.1.1 (soft constraints) matrix T just needs to be several orders higher than Q and R so that drone never trespasses the boundaries but if it needs to then allow the controller to find a solution. So the actual value is not significant. For the simulation T is a diagonal matrix with all its values equal to 10^6. The choosing of the values of Q and R has a more immediate effect over the behavior of the drone. There are several criteria for picking the values. First of all they are both diagonal. The values of Q will be kept constant at 1 or 0, depending if we want that state to affect into the result of the optimal problem. The only states that will be taken into account are the 3 position states and the yaw angle. This simplifies the resolution of the optimal problem and since the speed of the resolution is a critical constraint of the project is to our interest to simplify the problem as much as possible. The choice of those 4 states is simple. They are the only states that having a difference with the reference (a stable position) won't destabilize the system. For example, having a pitch value different that 0, with this Q matrix, won't cause the controller to react but it most certainly will destabilize the system causing it to change its position and then the MPC will act. The same will happen with the speed and the angular speed. Plus it helps the system to prioritize into achieving its objective position. The choice of the value 1 for those states that are being taken in account it's just a compromise solution that makes the system fast but not too aggressive. Plus the value 1 is of a similar order to the inverse of the maximum square error of the states (this is a common and simple criteria of choosing the values of Q). The values of Q will remain constant while the values of R will be used to tune the MPC. The final values chosen, with whom all the tests have been made, are: R=diag([0.8 0.5 0.5 1]), the first value affecting the throttle command, the second the roll, the third the pitch and the fourth the yaw. Those values are not definitive and most certainly will require tuning when they are tried with a real world. Again, they are a compromise that ensures a fast but not excessively aggressive behavior. Lower values would mean a faster system because the controller will use higher commands, but it will result in a bigger overshot or even an unstable close loop system. Higher values would mean a slower response. The values for the throttle and yaw can be set higher but no more than one order of magnitude with respect to the values of the roll and yaw command (observation made during simulation) since both vertical moment and yawing are already fast and very reactive. The weights for the roll and pitch command need to be kept low because most of the time when a certain pitch or roll is demanded is due to our will to move on the X or Y axe, respectably, and those movements are relatively slow (specially compared with the vertical movement) so we are interested in a more aggressive response from the controller.

4.4. - Improvements

This subsection treats with the improvement that have been tried in order to obtain a better performance and, what might be more important, a more realistic behavior of the controller. Due to the fact that this report only deals with

simulations it would be easy to take this results as a certainty but there a couple of factors that could compromise the performance of the MPC controller implemented on a real drone. The first factor is how aggressive is the controller on the first instants when a new reference is given. The second factor is how well the linearized system used by the controller adapts itself to reality specially when we are trying to move sideway.

4.4.1. - Constrains on the commands

Regarding the problem observed with the aggressiveness of the MPC especially for the case the throttle command. As it was observed after the results of the first test, the controller starts the taking off on full throttle command and this could have dangerous consequences for drone (already explained in the observations). One possible solution would be to increase the restrains on the command (they are already constrained by the maximum value that the Ardupilot software accepts). This might not be the best solution since there might well be some occasions where it is legitimate to demand maximum throttle or any angle command. An alternative solution is to add constrains in the maximal variance between the command at one given point and the command at the precedent step. Mathematically speaking:

$$\|u(k) - u(k-1)\| \leq Limit \quad (4.31)$$

This shouldn't stop the MPC from using extreme values for the command but it will force it to be more conservative. Results:

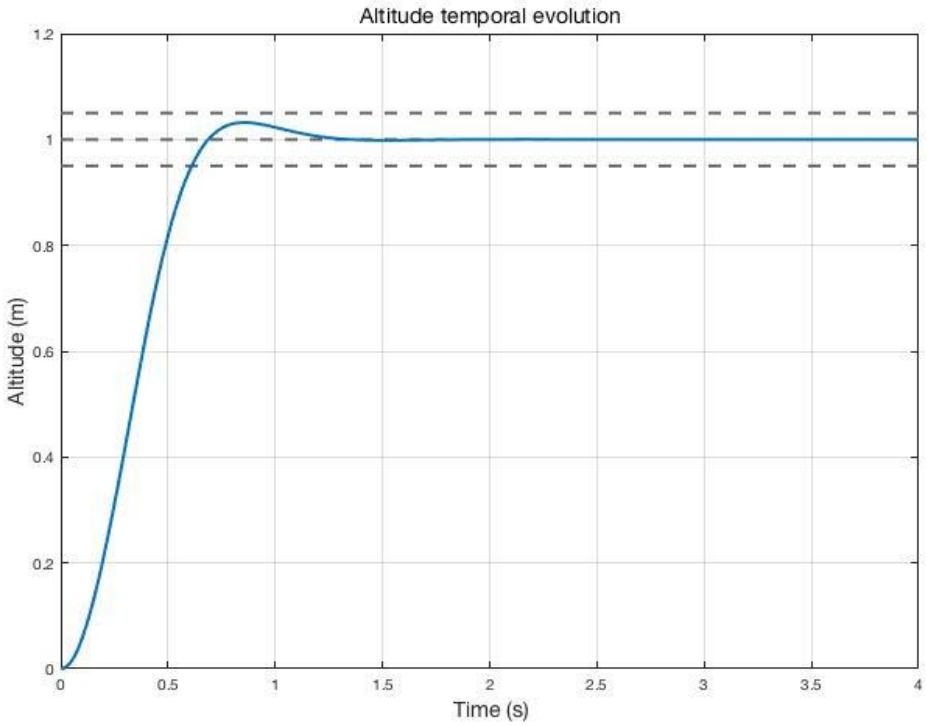


Figure 15: Result of the takeoff test with constraints on the command variation

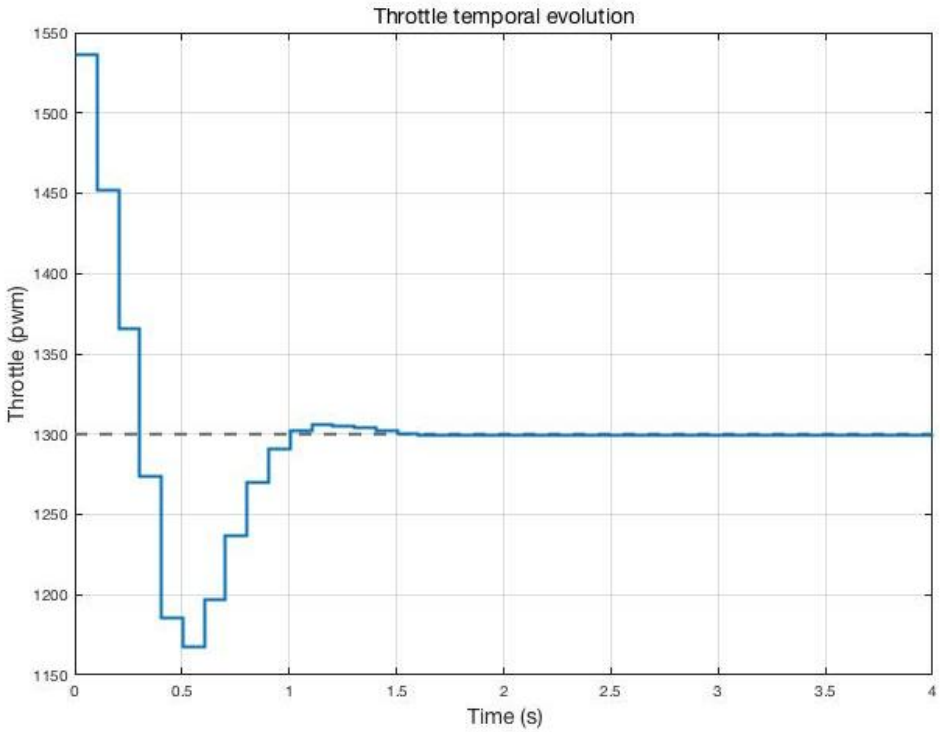


Figure 16: Result of the takeoff test with constraints on the command variation

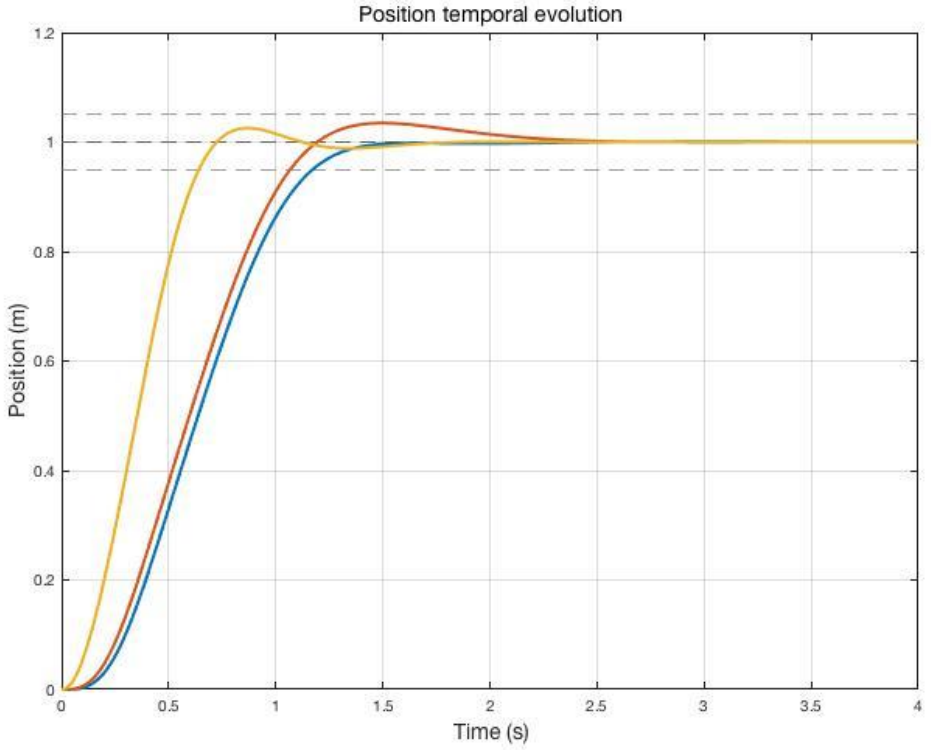


Figure 17: Result of the test for a objective position of (1,1,1)m with constraints on the command variation. In orange is the altitude (m) in red is the position in Y (m) and in blue is the position in X (m),

Observations: The first two figures shown above are the results for the simple takeoff test and the third is the result when the position of reference is (1,1,1)m. The drone is slightly slower (it takes more than 0.5 seconds to reach the position while before it was under the 0.5 seconds time mark) but it keeps being relatively fast. Where the big change can be appreciated is on the behavior of the command. While the curve seems similar to the one shown before its values has been reduced, not getting even close to maximum throttle. Also the peak speed and acceleration has been reduced (from 3 m/s to 2.2 m/s and from 27.5 m/s^2 to 11.5 m/s^2).

4.4.2. - Sequential linearization

Another inconvenient found to the linear MPC is that the model given represents a drone in a hovering position. This means that the MPC doesn't know how to move horizontally other than oscillating the pitch or the roll angles. A possible solution that allows keeping the formulation of the optimization problem and at the same time adapting the problem to a more realistic approximation of the system is the usage of sequential linearization. This technique consists of the following steps:

- Step 0: run the optimizer and obtain the predicted commands for the predicted horizon.
- Step 1: Use the predicted command to integrate the non-linear system and obtain a prediction of the states (inside the prediction horizon).
- Step 2: Re-calculate the linear system for every step in the prediction horizon using the predicted states of the Step 1.
- Step 3: run the optimizer. But this time the matrices A and B of the model depend on the step. Obtain the new set of predicted commands.
- Step 4: Repeat the last three steps until the predicted states (Step 2) converge.

For practical matters the number of iterations is reduced to a fixed number. The sequential linearization can be put mathematically as follows:

Given $\vec{u}_0$ *as a result of the optimization problem formulated at (4.30) and (4.31)*
While $\left|\vec{x}_j - \vec{x}_{j-1}\right| < \varepsilon$ *:*

$\vec{x}_j$ *= odesolver(* f *,* $\vec{u}_{j-1}$ *)*

For k=0, 1, …, N-1:

$$A_j(k) = \frac{\partial f}{\partial x}(x_j(k), u_{j-1}(k))$$
$$B_j(k) = \frac{\partial f}{\partial u}(x_j(k), u_{j-1}(k))$$

End_for

$$J = \frac{1}{2}\sum_{k=0}^{N-1}[\vec{e}^T(k)\cdot Q\cdot\vec{e}(k) + \vec{u}^T(k)\cdot R\cdot\vec{u}(k) + \vec{s}^T(k)\cdot T\cdot\vec{s}(k)\,]$$

$$Constraints: \begin{cases} \vec{x}_j(k+1) = A_j(k)\vec{x}_j(k) + B_j(k)\vec{u}_j(k) \\ \vec{x}(0) = \overrightarrow{X_0} \\ Angle_{max} + s \geq |x_j| \\ U_l \leq u_j \leq U_u \end{cases}$$

$$\vec{u}_j(k) = argmin(J, Constraints\,)$$

$$j = j + 1$$

End_while

The overall formulation of the problem doesn't change much. The major modification is in the constraints where now A and B depend on the step. The following are the results of the test when the drone tries to move from the position (0 0 1) to the position (0 3 1) using sequential linearization:

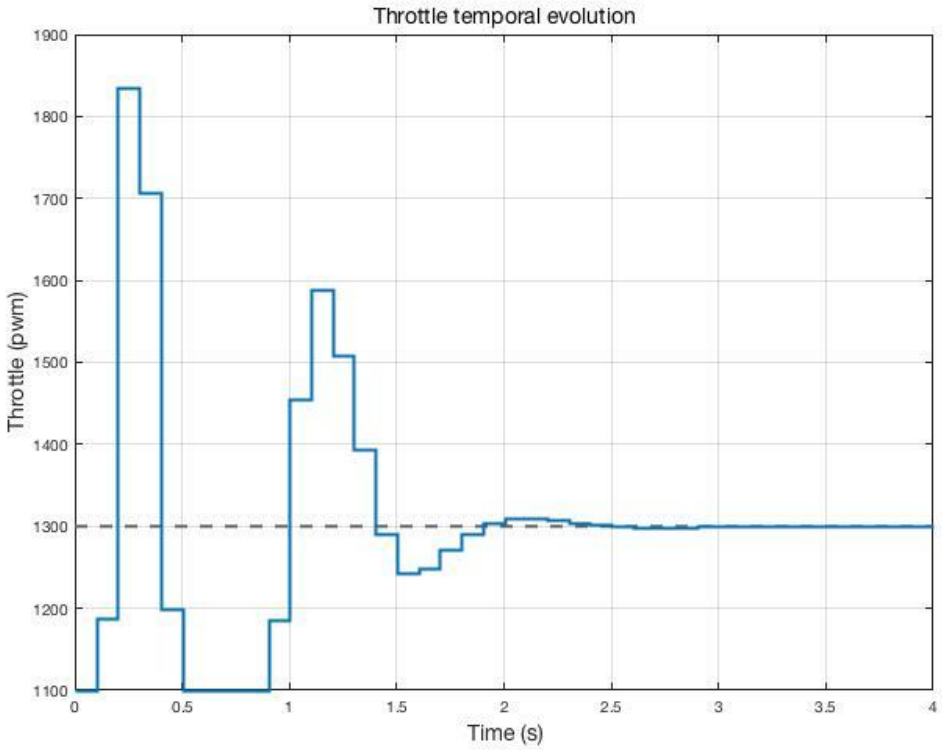


Figure 18: Result of test for the MPC with sequential linearization

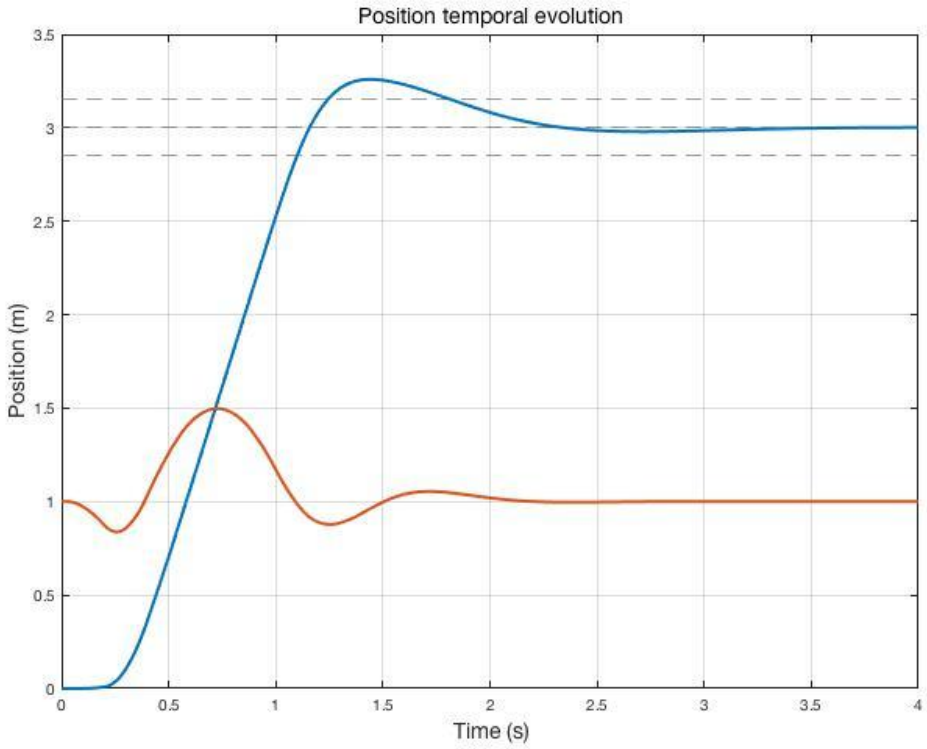


Figure 19: Result of test for the MPC with sequential linearization

Observations: The results obtained are for the most part underwhelming. Only one test has been done although it's a significant one because it is the typical situation when the sequential linearization should be most useful: a big horizontal movement. Although the system seems to react more aggressively, counting the overshoot the system reaches at the same time as the system without sequential linearization. Plus it affects the altitude. This is to be expected due to the fact that controller is now using the throttle command to move faster horizontally but its effects are disproportioned, displacing the drone a 50% from the altitude that it is supposed to hold. Several tests have been tried varying the number of linearization but the results don't change substantially from the ones shown in the figure.
So from the results and the fact that it would take more time for the onboard computer to run the optimizer and the sequence of the linearization it arrives to the conclusion that it is not worth to implement the sequential linearization.

4.5. – Alternative configurations

So far, the results presented have been for a determined configuration of the controller system. This configuration is the one shown at the figure 5. The command send by the MPC to the inner controller represents the thrust requested and the reference for the roll, pitch and yaw rate. This configuration is intuitive because it is basically substituting the manual command with the MPC leaving the rest intact. But this configuration is not unique. The ardupilot code allows us to change the type of PID/attitude controller but also other

modifications can be added. In this sub-section are presented two configurations for the ensemble of MPC-inner controller-Drone:

- Configuration 1: The inner controller is change so that instead of supplying the reference values of the pitch and roll, we supply the reference for the pitch rate and roll rate (take the figure 4 and erase the proportional controllers).
-
- Configuration 2: The inner controller is eliminated and the MPC provides the PWM values directly to the motors.
-

Those new configurations should provide a more aggressive style of control. The next table provides some of the results obtained when they are tested with the same examples shown above:

	Conf 0 (**)	Conf 1	Conf 2
Minimum sampling rate	5Hz	10Hz	10Hz
Sampling rate used (*)	10Hz	20Hz	50Hz
Minimum Prediction horizon	5	20	20
Prediction horizon used (*)	20	40	80
Time taking off	0.5s	0.5s	0.4s
Overshot	<5%	<5%	<5%
Time from pos (0 0 0)->(1 1 1)	1.5s	1.15s	1s
Overshot	<5%	<2%	<2%
Time from pos (0 0 1)->(0 3 1)	1.6s	1.3s	1.55s
Overshot	Non	<5%	<5%
Proportional time cost of the computation (in Simulation)	1	2.5 - 3	8 - 10

(*) Those are considered to be optimal (between performance and time consuming)
(**) Configuration 0 is the one that has been used for the other sections of the report.

Observations: The two new configurations are slightly more aggressive but they don't improve much the performance. What does increase is the time of simulation that might give us an idea of the computational cost to run those programs in the RPi. Note that in terms of vertical acceleration they are all equally fast since the inner controller doesn't affect the throttle command. In terms of horizontal movement they all suffer from the same problem already mentioned in the sections above. They are only been tested with setpoint tracking so it might be that for a more aggressive command (something more acrobatic) the new configurations respond better. That being said, I wouldn't recommend trying this linear MPC with angles far from the hovering position.

4.6. - Trajectory tracking

All the simulations that had been shown so far have been tested by setpoint tracking, i.e. giving one specific position in the space as constant reference for the MPC controller. The references for the angles or the speeds are kept to 0. This allows us to get a general idea of the behavior of the controller (and also the modeled system) and easily compare the performance when changes are introduced or the parameters of the controller are tuned. But this kind of tracking may not reflect the final objective of the project. Trajectory tracking was one feature incorporated on precedent reports and it should be a final objective for future reports, so it has been decided to test the linear MPC controller implemented in simulation with a couple of reference trajectories. In the case of the trajectory tracking a series of values for the position and speeds are given at a prescribed time while the references for the angles and the angular speeds are kept to zero. This would require a reference generator. This generator hasn't been implemented for this report so simple trajectories already generated by older reports are going to be used adapted to the characteristics of this new drone. Also, since a very simple weight matrix Q is being used where the diagonal values for the speed are set to zero, the reference trajectories for the speed are set to zero, only imposing the position.

For this test the following trajectory will feed to the MPC as a reference:

$$\begin{cases} x(t) = a \cdot t \cdot \sin(b \cdot t) \\ y(t) = a \cdot t \cdot \cos(b \cdot t) \\ z(t) = c \cdot t \end{cases} \quad (4.32)$$

In this particular case $\boldsymbol{a}$, $\boldsymbol{b}$ and $\boldsymbol{c}$ are equal to 0.1, 1 and 0.1, respectably. The following figures show the results of the simulation:

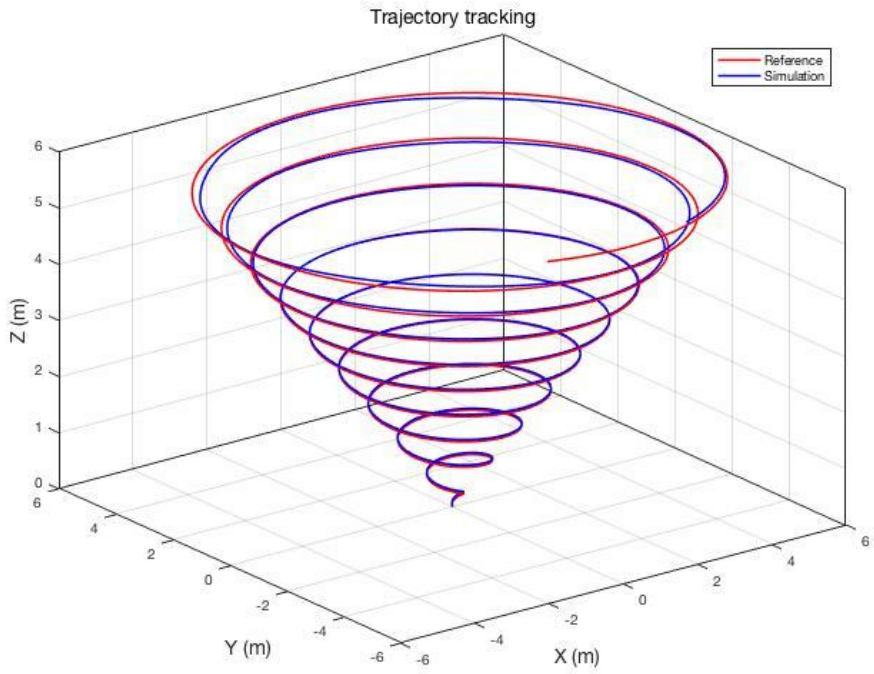


Figure 20: Results for the test of trajectory tracking

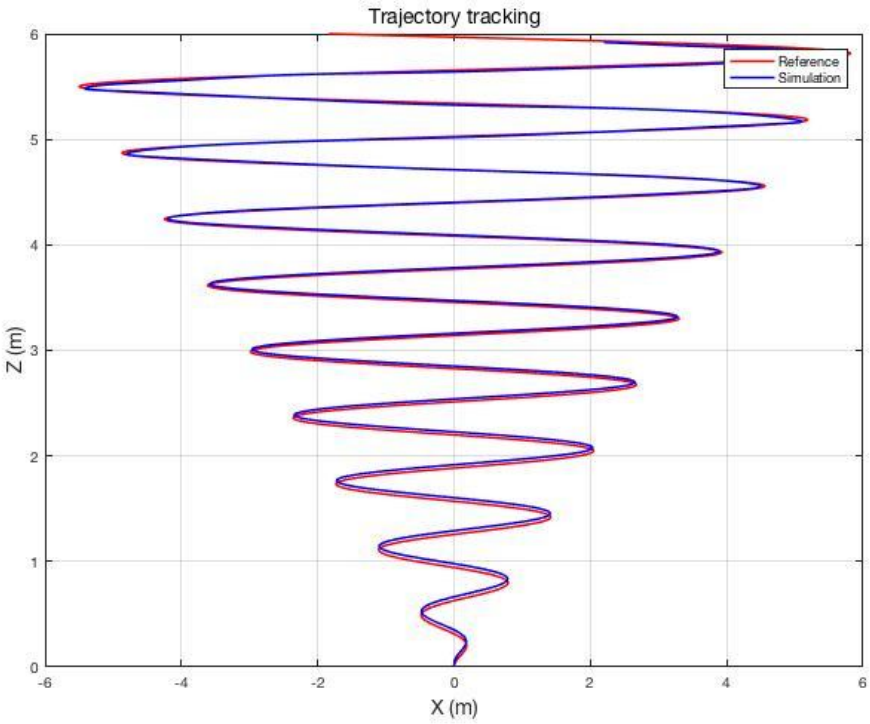


Figure 21: Results for the test of trajectory tracking

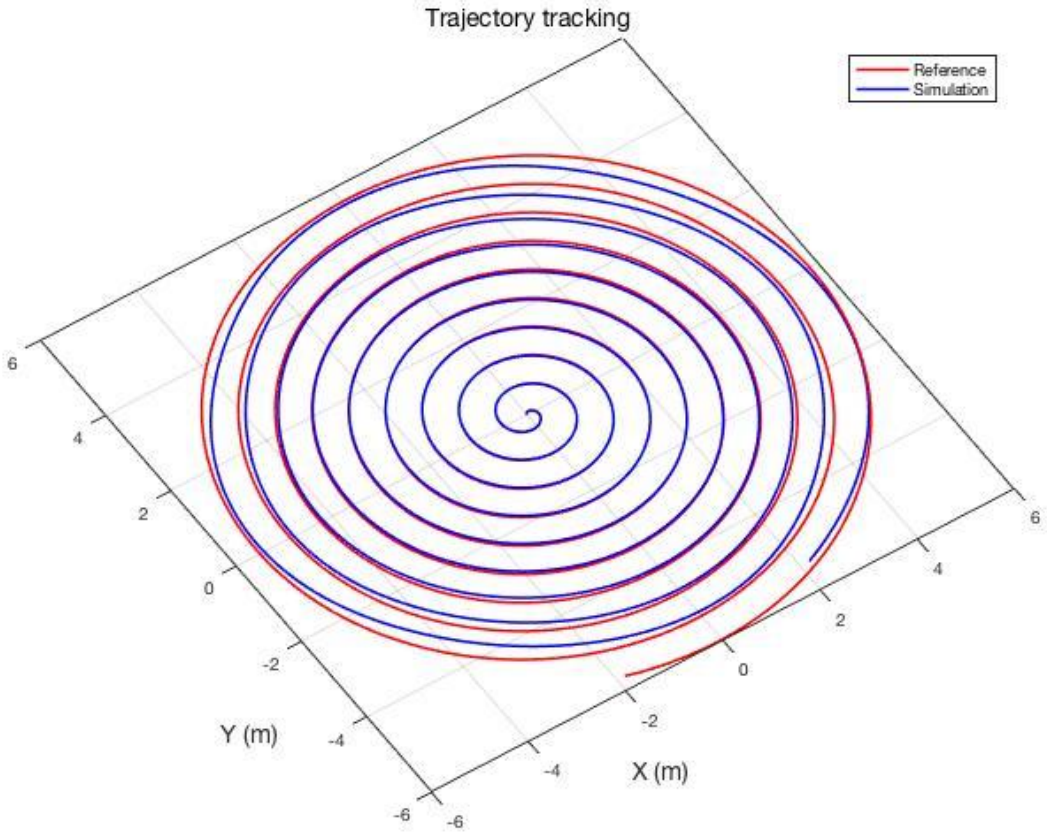


Figure 22: Results for the test of trajectory tracking

Observations:
The MPC controller is quite capable of following the trajectory, and it only gets slightly out of the trajectory on the last turns, being the maximal distance between reference and trahectory no more than 10 cm in a circumference of more than 10 m of diameter. In simulation the drone is capable of move through the whole trajectory in 60 seconds reaching a top absolute speed of 6 m/s approximately. It is also important to mention that on the last turns the drone reaches the limit of 30 degrees for the pitch and roll angle and that is another reason why it seems to not be able to reach certain points in the trajectory. Remember that this test is using the MPC without sequential linearization and therefore in order to move sideways it only uses pitch and roll commands. This would have a good side: since throttle command is dedicated to control the altitude the drone follows the trajectory on the Z axe perfectly as it can be seen on the second figure shown for this test.

This test has also been tried with different values for $\boldsymbol{a}$, $\boldsymbol{b}$ and $\boldsymbol{c}$ for the equation 4. 32 and lower values for $\boldsymbol{N}$, the prediction horizon of the MPC. Although the results are not shown (they can be reproduced with the MATLAB simulation) here are some observations: Increasing $\boldsymbol{a}$ means that a bigger horizontal movement is required and the drone reaches faster its limits for pitch and roll. Therefore it have some problems to perfectly follow the trajectory (same effect than can be observed on the last turns of the drone for the precedent test). This has more to do with the maximal horizontal speed and acceleration that the drone can provide, therefore for any given trajectory either make sure that the drone is not requested to reach those accelerations or increase the limit of the pitch and roll in the soft constraints of the optimal problem. Increasing $\boldsymbol{b}$ means demanding a higher frequency for the periodical horizontal movement. If we consider that the whole system behaves like a second-order transfer function, a classical frequency-domain analysis of a second order system tells us that the amplitude of the response should decrease and a phase difference of 180° should appear between input and response. During the test a similar behavior can be observe while increasing $\boldsymbol{b}$: decrease in amplitude and a bigger phase shift. Not all the values of $\boldsymbol{b}$ can be tested because the MPC is sampling at 10Hz (0.1s) and, following the Nyquist-Shannon sampling theorem, only signals with a lower than 5Hz would be acceptable. But there is no need to reach those frequencies (really high if we think on a moving object) because, by that point, the amplitude of the response would be less than a 10% of the input due to the second-degree dynamics. In fact for a movement of a frequency higher than 0.25Hz the amplitude of the response is lower than a 90% percent of that of the input. Increasing $\boldsymbol{c}$ doesn't have a great effect since, at it has been shown, it's more reactive to altitude changes. In fact a (reasonably) bigger value would generate a better performance. This is due to the fact the over all thrust of the motors would be higher and although the MPC doesn't understand that a higher thrust plus an inclination means an horizontal acceleration this doesn't stops the real drone from using the extra throttle to move faster without the need of extremely pitching or rolling to achieve that horizontal acceleration/speed. In terms of varying $\boldsymbol{N}$, the tests show that it can be decrease to 10 (that is a horizon of 1s, half of what we've been using) without any major decreases in performance:

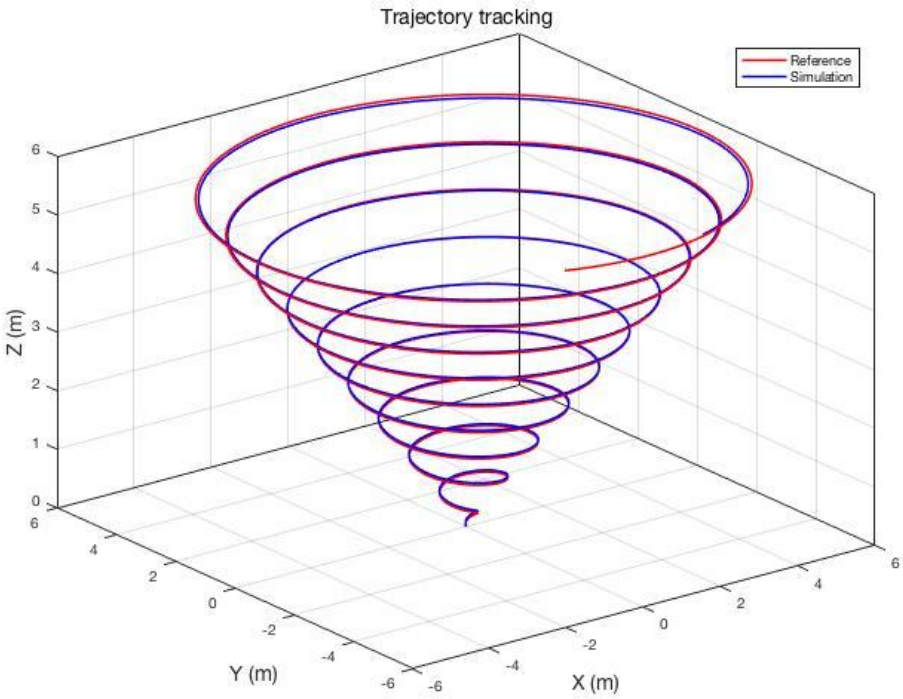


Figure 23: Results for the test of trajectory tracking given a prediction horizon of N=10 (1s)

This is an important result since a lower ***N*** would mean less computational time. The only problem is that, if we compare again the system to a second order function, its cutoff frequency is smaller so the performance for high frequency movements decreases.

Tests have also been made with the sequential linearization MPC in order to see if, in terms of following a trajectory, it improves the results. Those are the results of the test:

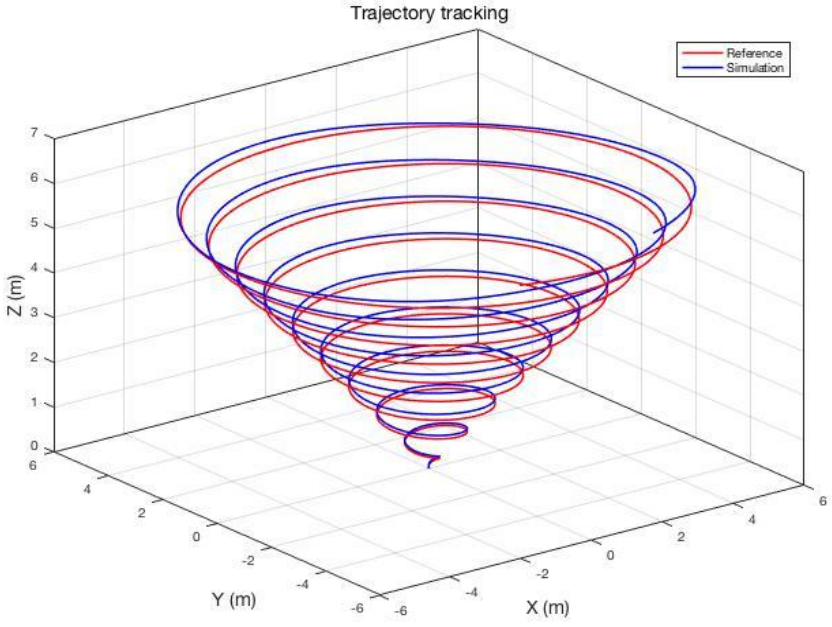


Figure 24: Results for the test of trajectory tracking with sequential linearization

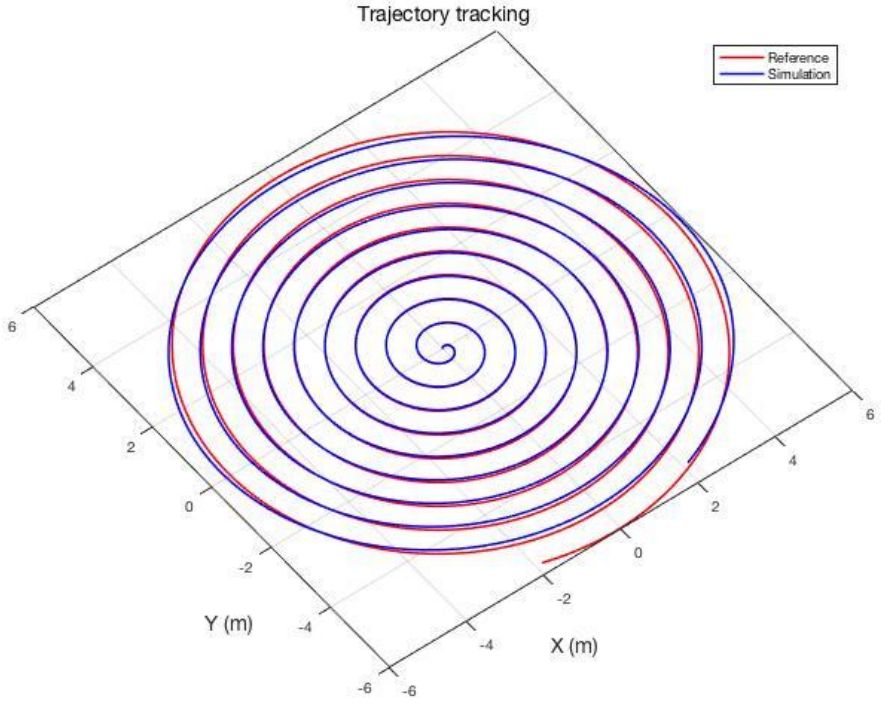


Figure 25: Results for the test of trajectory tracking with sequential linearization

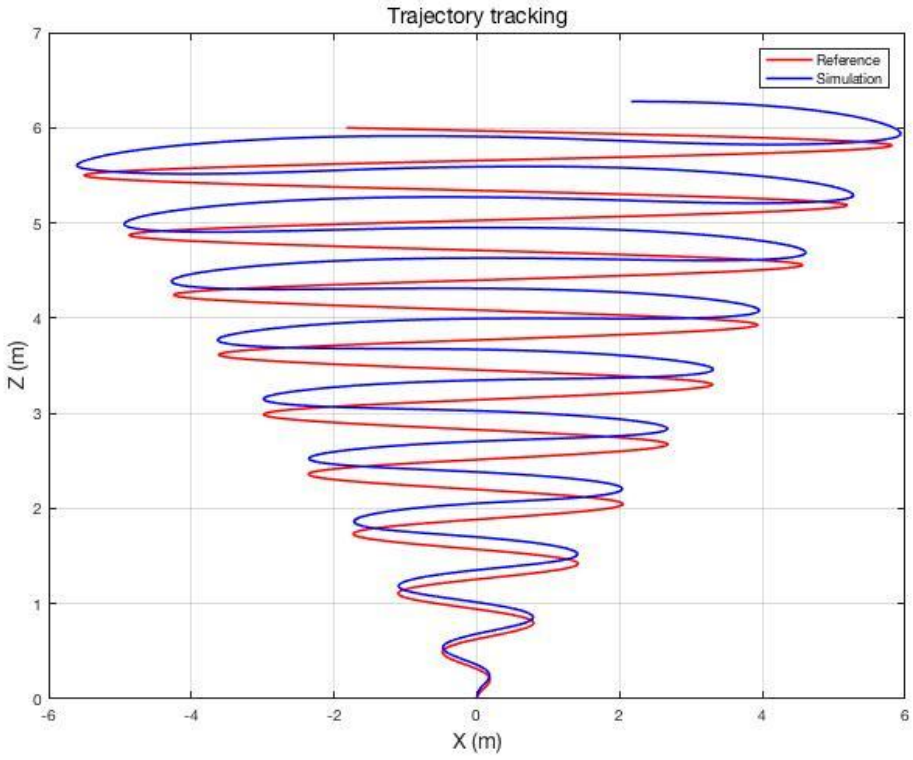


Figure 26: Results for the test of trajectory tracking with sequential linearization

Observation: As it can be seen the results are not bad but they are far from the ones obtained without the sequential linearization. Again, the system

compromises its precision in terms of altitude positioning in order to use the throttle command to move faster horizontally. Off course the movement of the drone is more realistic and, as it has been said, it is probable that a real drone would perform better with the sequential linearization, but, as it stands right now and using only the results of the simulations the sequential linearization shows a worst performance with a higher computational cost.

Chapter 5: Experimental work

Apart from simulation, some experimental work has been made to test the drone and the tracking room. All those tests were made with the manual controller. The first test with the drone served to assure that the drone could fly, that it was well calibrated and that the PID's of the inner controller worked well with the default values. All of those tests were satisfactory and the drone could easily be flight manually. The next step was to try system identification, essential if we wanted to implement the MPC.

5.1. – First attempt on System Identification

During the first stages of the project we made a first attempt on system identification. Because the tracking room was not available at that time we have no way to measure position of the drone so we tried the Sys ID only using data from the IMU of the Navio2 board (mostly accelerometer and gyroscope) and comparing the experimental values with the values obtain by the model and PWM values send to the motors. The parameters to indentify were: the lift and drag coefficient of the motors and diagonal values of the intertie matrix (see chapter 3). The idea was to use simple commands (like hovering or yawing) in order to identify the parameters separately. The data of the IMU and the PWM send to the motors could be extracted after the flight connecting the drone to the ground station and downloading the log files (they can be converted to MATLAB files). These are some of the results of the system identification:

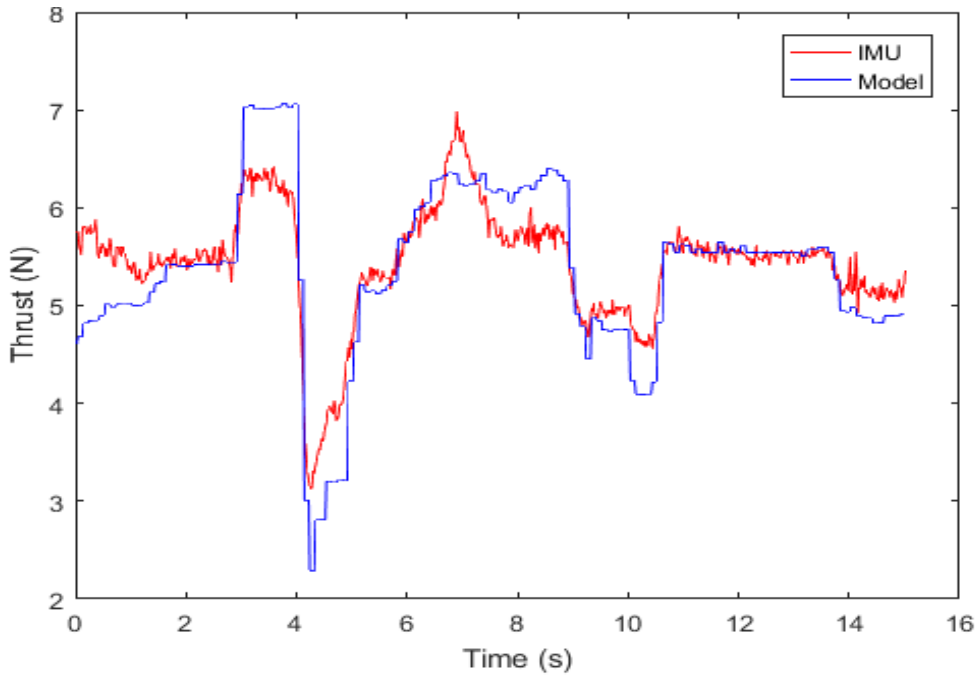


Figure 27: Comparison between Thrust calculated from the measurement of the IMU and the values expected from the model

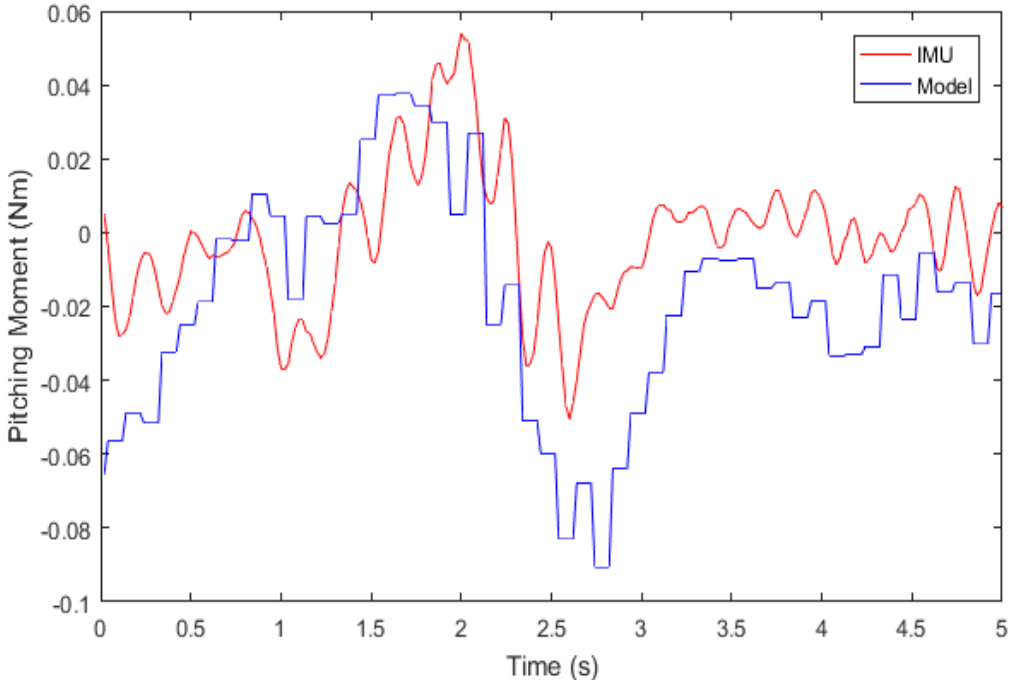


Figure 28: Comparison between Pitching Moment calculated from the measurement of the IMU and the values expected from the model

Although it can be appreciated some similitude between experimental data and the model, mathematically only for the identification of the lift coefficient (related to the results shown in the first of the figures shown above) we obtain correlation values higher than 90%. For the other coefficient the test obtained results with correlations under 60%. These derive mostly from two factors:

- Data from the IMU log is, sometimes, too noisy and somewhat unreliable.

- The simple movements we tried in order to identify the parameters separately are too short in time (for example pitching a certain angle without crashing the drone) and not enough data is obtained.

In order to solve these two problems it was proposed to use the tracking room (by the end of the project was already available), run long flying test with random movements and use all the position and attitude data to make an overall identification of the system.

5.1. – First Tests in the tracking room

How the tracking room works is defined in the chapter 2.3. The first attempts we used to test the streaming of the data from the Motion Capture System and how

well it is capable to follow the trajectory of the drone. The next figure shows some of the results of tracking obtained during a random flight:

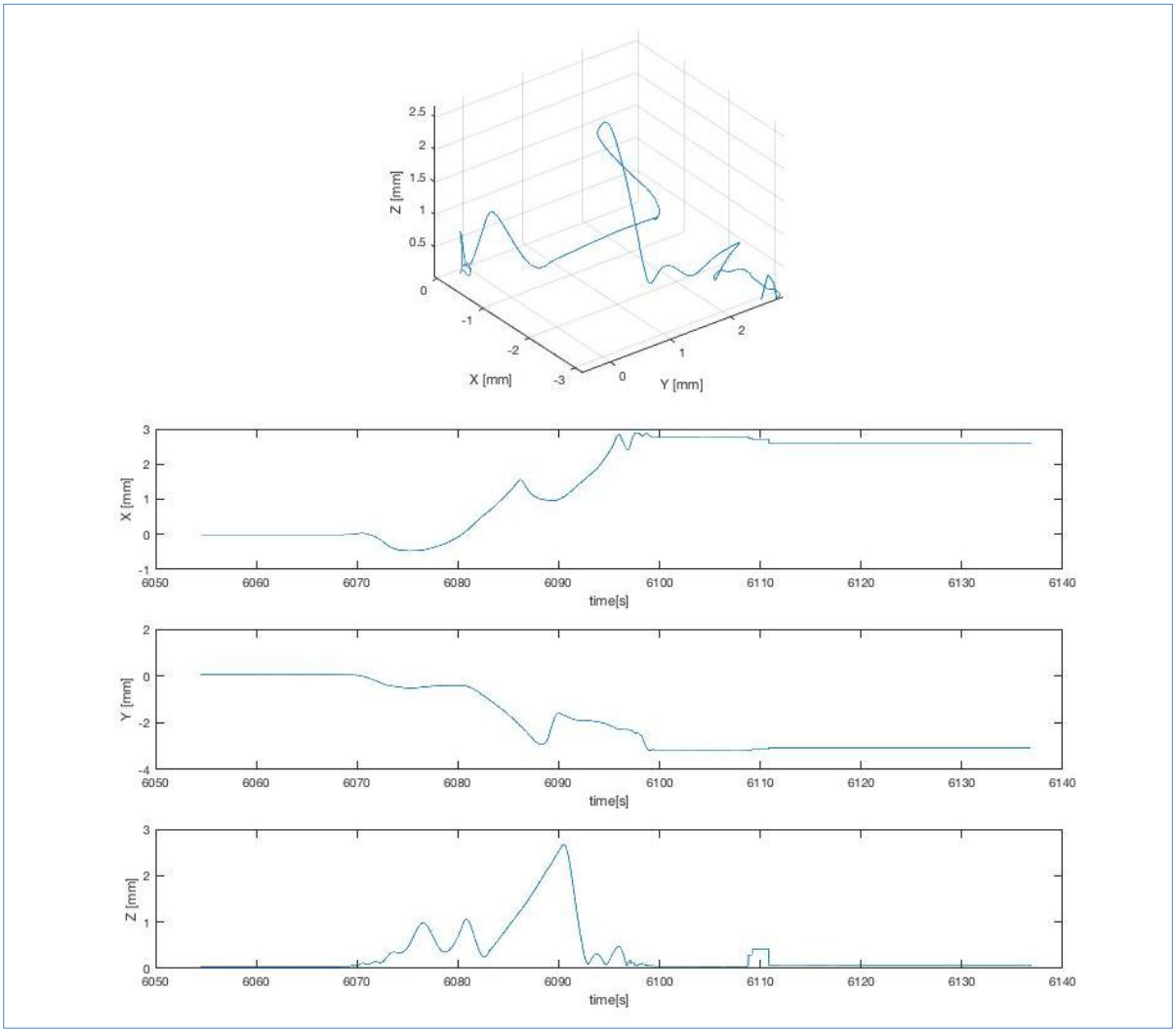


Figure 29: Results of MCS during a random trajectory

The results shown in the figure have been post-processed with MATLAB. As it can be seen the results are not as noisy as the ones obtained in the IMU. These results can be derived to obtain the speed of the drone. As it has been said the information of the attitude is presented in quaternions but can easily be translated to Euler angles during the post-process.

The next step would be to use that data in combination with the logs of the ardupilot (we need them to know the commands send to the motors) to do the System Identification. There is only one problem that stopped us from pursuing in that direction: The synchronization between the two sources of data. The ardupilot code and the Motion Capture System have different methods to time stamp the data recorded. The ardupilot starts counting the time when the drone is armed and creates a log file every time it stops. The Motion Capture System starts counting when a new session of the program is initialized and creates a file every time it starts recording. As it is, there is no way to synchronize automatically the two sets of data using the time stamps. One solution we proposed was to use the other data from IMU and try to correlate with the

attitude data of the Motion Capture System (by locating characteristic movements). This proved to be extremely laborious and inaccurate. To exemplify this, the next two figures show data from the two sources:

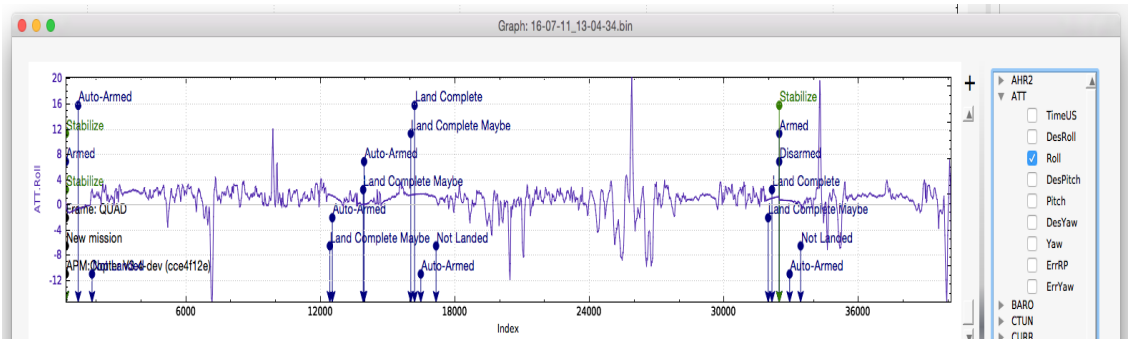


Figure 30: Roll recorded by the Ardupilot

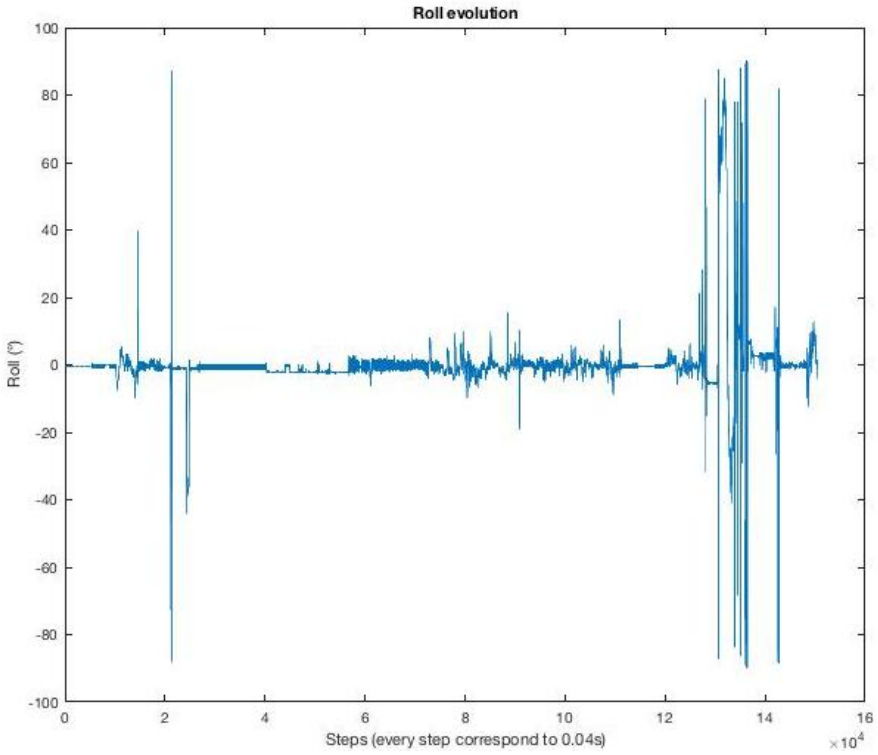


Figure 31: Roll recorded by the Motion Capture System

Technically the data from the log of the drone should be found inside the data captured by the MCS but to find a correlation is not an easy task.

One of the first questions to be addressed in future installments of this project should be to find a way to synchronize the two data. It could also be interesting to find a way to read the values send in real time to the drone by the manual controller without the need of reading the logs of the drone.

Chapter 6: Conclusion

This report presents the result of the implementation on simulation of a linear MPC with a non linear system. The report could be divided into two big parts. During the first part it is presented the building of the drone and a briefing about the software that comes incorporated. It also represents the first months of the project and the hours dedicated to the understanding of the drone and its setup. This part is directed to futures students that will continue this project and it should help them to understand the functioning of the drone so that they can quickly start addressing some points left aside during the project like the System Identification and the implementation on the real drone. It should also help them in the reproduction of the drone when project gets to the point of trying to fly several drones at the same time. The second part of the report enters more in the domain of the automatic and optimal control with the modeling of the close loop system and the formulation of optimal problem. Finally the MPC is tested on simulation.

The results obtained, though limited, are, for the most part, satisfactory with a good behavior of the drone, at least in simulation. The results are good enough to motivate the implementation of the linear MPC on the real drone. Sequential linearization, although in theory may seem like an improvement over the single linearization system, presents underwhelming results and an increase in computational power required in comparison with the standard linear MPC. Still, and as it has been observed in the precedent chapters, the system with sequential linearization presents a more realistic motion of the drone so it could be interesting to test its behavior in the future with the real drone (if it can be implemented at real time).

Three main issues have been left aside in this report that should be addressed and focused by the incoming student that takes on this project: System Identification, the finding of a reliable way to communicate and interact with the drone and its software other then the manual controller, and the implementation of an optimizer that could run the MPC at least at 10Hz inside the RPi. These three points are interconnected and they are all necessaries to achieve the final long term objective of the project. They pose more of a logistical problem and they should be addressed from the beginning of the project. In the final stages of this project some of the issues have been confronted but without too much success due, for the most part, to time constraints and that is the reason why they are not formally included in the report.

Lausanne, July 27, 2016

Martí POMÉS ARNAU

References

[1] R. Mahony, V. Kumar and P. Corke, “Multirotor Aerial Vehicles”, published in the IEEE Robotics & Automation magazine, September 2012.

[2] V. Streit, “Control of Quadcopter – Identification”, semester project done at the Swiss Federal Institute of Technology in Lausanne (EPFL), January 2015, Lausanne, Swiss.

[3] W. Amanhoud, “Model based control of quadcopters”, Master project done at the Swiss Federal Institute of Technology in Lausanne (EPFL), Jully 2015, Lausanne, Swiss.

[4] L. Dubois, “Control of Crazyflies”, semester project done at the Swiss Federal Institute of Technology in Lausanne (EPFL), January 2015, Lausanne, Swiss.

[5] Altug Bitlislioglu, “Time Optimal Cornering”, work done at the Swiss Federal Institute of Technology in Lausanne (EPFL), June 2014, Lausanne, Swiss.

[6] “CVXPI”, http://www.cvxpy.org//, Online, accessed June 2016.

[7] “Ardupilot developer guide”, http://ardupilot.org/dev/inde3.html, Online, accessed June 2016.

[8] “Navio2 documentation”, http://emlid.com, Online, accessed June 2016.

[9] “Ardupilot developer guide”, http://ardupilot.org/dev/index.html, Online, accessed June 2016.

[10] “Lipo battery calculator”, http://multicopter.forestblue.nl/lipo_need_calculator.html, Online, accessed June 2016.

[11] M. Zaare Mehrjerdi, B. Moshiri, G. Amoabediny, A. Ashktorab, B. Nadjar Araabi, “Model Predictive Control with Sequential Linearization Approach for a Nonlinear Time Varying Biological System”, published in 25th Chinese Control and Decision Conference (CCDC), 2013.

[12] R. Cagienard, P. Grieder, E.C. Kerrigany and M. Morari, “Move Blocking Strategies in Receding Horizon Control”, Published in Conference on Decision and Control, 2004. CDC. 43rd IEEE

[13] S. Yua, C. Böhm, H. Chen and F. Allgöwer, “Stabilizing Model Predictive Control for LPV Systems Subject to Constraints with Parameter-Dependent Control Law”, Published by Stuttgart Research Centre for Simulation Technology (SRC SimTech), June 2009.