Quadcopter autonomous navigation in a static or dynamic environment using a correct by construction controller

By

FRANCK DJEUMOU

Master's degree in Computer Science (**COMASIC**) final report.

March - September 2017

Under the supervision of :

DR. UFUK TOPCU
utopcu@utexas.edu

SYLVIE PUTOT
putot@lix.polytechnique.fr

ERIC GOUBAULT
goubault@lix.polytechnique.fr

Author's declaration

I declare that the work in this dissertation was carried out in accordance with the requirements of the University's Regulations and Code of Practice for Master's Degree Programs and that it has not been submitted for any other academic award. Except where indicated by specific reference in the text, the work is the candidate's own work. Work done in collaboration with DR. UFUK TOPCU, DR. ERIC GOUBAULT and DR. SYLVIE PUTOT. Any views expressed in the dissertation are those of the author and collaborators.

SIGNED: ....DMFB...................... DATE: ....09/02/2017.................

Acknowledgement

Foremost, I would like to express my sincere gratitude to my internship supervisor at the University of Texas at Austin, Dr. Ufuk Topcu, for the opportunity he gave to me, for his guideline, for making us focus on our work and only on our work and finally for his way of managing the team that leave time to evaluate much more possibilities and make me more independent and thus experiencing the research world.

I am also grateful to my Master's professors Dr. Eric Goubault and Dr. Sylvie Putot, for the huge help in finding the appropriate internship for my Master, their continuous implication in the project, the constant encouragements, and finally the valuable insight.

Last but not least, I would like to thanks my family , my friends for the insane and endless encouragement. Another special gratitude for the member of u/-t autonomous and radio navigation lab group who helped me for my internship in Austin.

Abstract

In many situations, the use of an robot such as an flying drone are used to help compliment humans in performing tasks such as package delivery, cleaning, agriculture, surveillance, search & rescue, construction and transportation. Often these robots require a human operator to control and oversee them. The use of an autonomous robot that can navigate in unknown environments autonomously removes the need for a human operator and allows for quicker task performing. The autonomous navigation of robots in uncontrolled environments is a challenge because it requires a set of subsystems to work together. It requires building a map of the environment, localizing the robot in that map, making a motion plan according to the map, executing that plan with a controller, and other tasks; all at the same time. In this paper, we assume that we already have a 3D occupancy grid of the environment we want to explore or perform some missions. This 3D occupancy grid can be obtained using these known methods: SLAM algorithm, computer vision, Probabilistic 3D Mapping (Octomap).

There is a lot of obstacle avoidance algorithm out there which build the map at the same time as exploring. The majority of them are based on iterative method so are not theoretically proven to be correct by construction. Moreover these algorithms do not handle well when there is moving objects/obstacles in the environment without huge assumptions on these moving objects. So the basic case where we already have built the map (static environment) and want to send multiple vehicles on autonomous mission can hardly be handle by these algorithms. A more complicated case is for example when multiple vehicles must be send for some autonomous mission in an environment where there exists uncontrolled moving vehicles (dynamic environment). In this paper we propose a solution for these type of problems (full or partial observability assumption) with a synthesized controller which can be proven by model checking to satisfy user's specifications.

Table of Contents

LIST OF FIGURES

Chapter 1

Introduction

1.1 Background

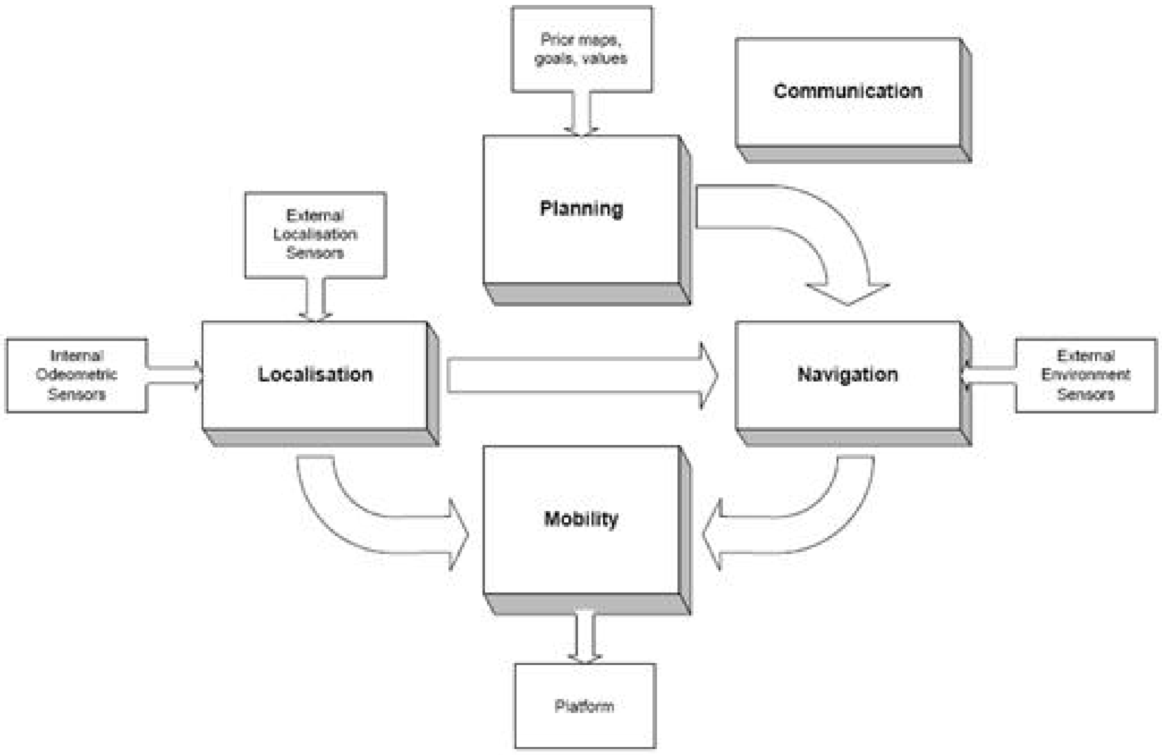
Generally speaking, why is it important to develop autonomous navigation robots ? The answer is obvious for most of the people. Basically, there are three kinds of robots: operated, autonomic and autonomous. Operated robots are those that require control by a human, e.g. teleoperated robots for surgery , army exploration or some Mars explorations rovers (not fully autonomous). Automatic robots do preprogrammed and repeated activities in controlled environments, e.g. follow trajectory or target for drone or robots in general. In contrast, autonomous robots must do tasks in uncontrolled environments and make their own decisions as a function of the given goal, e.g. driverless cars in cities or drone autonomous navigation. **The tendency is to give robots more autonomy, which means robots do tasks with as little human assistance as possible**.

Autonomous navigation is the capacity of a robot to navigate as human little assistance as possible. *Unstructured environment* is an environment that is NOT adapted to facilitate robot navigation, but is given as it is. Navigation in unstructured environments is a problem with many variants. Generally speaking, an autonomous navigation system may include the following steps:

- **Perception**: Interprets the numbers sent by sensors to recognize objects, places, and events that occurs in the environment or in the vehicle. In this way the vehicle can prevent damage, know where it is, or know how the environment is;

- **Map building**: Create a numerical model of the environment around the vehicle. The map allows the vehicle to make appropriate decisions and avoid damage;

- **Localization**: Estimate the vehicle's position relative to the map. This helps it plans and execute movements, and build a correct map of the environment.
- **Planning**: Decides the movements necessary to reach the goal without colliding.
- **Control**: Ensures the planned movements are executed, despite unexpected disturbances.
- **Obstacle avoidance**: Avoids crashing into moving objects, such as people, animals, doors OR others moving vehicles.


Prior maps, goals, values
Communication
Planning
External Localisation Sensors
Internal Odeometric Sensors
Localisation
Navigation
External Environment Sensors
Mobility
Platform
Figure 1.1: Some components of an autonomous navigation system

In this report, we focused on drone autonomous navigation. This has a lot of application in human activities. For example, package delivering, agriculture, surveillance, search and rescue, top secrets military operations. However a basic task that robots must do is to navigate in natural and human environments in order to achieve these applications. Obviously it is crucial for the drones to navigate in these unstructured environments. **Thus, the aim of this report is to propose a working solution (and its limitations) for drone autonomous navigation in a map containing static obstacles and uncontrollable moving vehicles using a correct by construction controller. The correct by construction property of the controller can easily be proved using model checking on the generated automaton**.

1.2 Hypothesis

In this report, the complete problem of autonomous navigation, as described in the previous section, will not be considered since it is a too large problem to be solved with only one study. Some steps will be supposed to be already done by some external agent to ease the problem:

- **Perception**: Our vehicles will only have default drone sensors for attitude and position control and cameras. it will have no other obstacle detection capabilities.
- **Map building**: This part is given by the user. The vehicle already have an occupancy grid[1] of the environment which basically is an evenly spaced field of binary random variables each representing the presence of an obstacle at that location in the environment. The occupancy grid can be created by hand by the user when testing on real drone, it can also be obtained in simulation or outside simulation using some vision based algorithms that will be present later in the report. **Here the occupancy grid only contains reference to static obstacles**.
- **Localization**: the position and orientation of all vehicles in the arena was given by a Motion Capture Systems such as **Vicon System [2]**. So the controlled drones have a global observation of everything happening in the arena. It is also important to precise that the solution propose here will also works with just a partial observation of the map which can be given by sensors embedded on the vehicle. This part will be explain at the end of this report.
- **Planning && Obstacle avoidance**: This part will be realized by a controller synthesized with the Temporal Logic Planning toolbox **TuLiP** with my implementation of minimum snap as trajectory generator.
- **Control**: We synthesized a controller that allows us to follow a trajectory with a perfect timing and a good static error for position , velocity and acceleration setpoints.

1.3 Structure of The Report

This report will be organized as following:

- **Chapter 2**. We introduce the strict minimum and standards tools required when working in a so-called autonomous team. Basically, we are going to give a basic understanding about the well-known robotics middle-ware *ROS*. Then we will present the simulation environments we chose for the different simulations we made and the drawbacks and advantages of each of them. Finally this chapter will be closed with a small description of *VICON System* and the arena we made our flight tests.

- **Chapter 3**. We present here the drone design and control. This chapter will be about the hardware and flight controller used for building the quad. We will also give more information about the quad attitude/position estimation based on the fusion of internal data from the main board and external data from *VICON System*. The quad attitude/position control and taking-off / landing strategies will be also presented in this chapter. We will then discuss about a huge limitation of the flight stack used which only allows us to have control of either position or velocity or acceleration but not the three at the same time. This limitation will be solve with the PVA Controller for assuring a correct state transition duration in our tulip model. Finally, We will present here some results obtained when combining a good attitude/position control with *FOVE Eye Tracking Virtual Realty Headset* : Basically a demonstration of a flying quad with only eye tracking and head movements.

- **Chapter 4**. We expose here our improved, fast and multi-threaded trajectory generator based on the original minimum snap article. And how it will be used for making possible autonomous mission.

- **Chapter 5**. We present a *PVA Controller* that takes as input Position, Velocity and Acceleration and give as output attitude value which will be feed to the controlled quad. This controller is a main feature in our problem.

- **Chapter 6**. Finally we will introduce the essential feature of this report which is the tool *TuLiP* used for synthesizing a discrete controller that guarantees to satisfy all the specifications given in *LTL* to the solver. We will give more details about the way it works and some applications in the autonomous navigation of drone. Finally we introduce its limitations and what improvement can be done to our solution.

1.4 Contribution

The main contributions of this work are the following:

- We propose an improvement to the original minimum snap trajectory generator which have no numerical instability , which very fast, which has a fast model for handling corridors constraints and inequality constraints.The implementation also includes a fast optimal time segment.
- We synthesized a *PVA to attitude* controller that enables the use of the trajectory generated by the minimum snap implementation.
- We wrote general specifications for autonomous navigation of N controlled quad in a static environment. Each of them can be assigned in real-time some goals to reach while not entering in collision with static obstacles and others controlled vehicles.
- We wrote general specifications for autonomous navigation of N controlled quad and M uncontrolled quad in a static environment. Each of them can be assigned in real-time some goals to reach while not entering in collision with static obstacles, controlled vehicles and uncontrolled vehicles.
- We proposed algorithms for going on a more continuous behavior when there are uncontrolled vehicle in the environment.

CHAPTER 2

ROS and Simulation Environments

When doing robotics based developments or experiments, *ROS* is one of the main middle-ware tool to use. *ROS* provides libraries and tools to help software developers create robot applications. *It provides hardware abstraction, device drivers, libraries, visualizers, message-passing, package management, and more*. *ROS is licensed under an open source, BSD license*.

We mainly used *ROS* for sharing tasks and exchanging messages between different hosts in the laboratory (distributed systems), for visualizing, monitoring and doing offboard control. After a short description of ROS, we will present the simulation environments we used to test our algorithms before launching them on a real quad. Finally we will give a short presentation of *VICON System* and our quadcopter testbed.

2.1 R.O.S

ROS[3] is a framework for building robotics software that emphasizes large-scale integrative robotics research. A system built using *ROS* consists of a number of processes called *nodes*, potentially on a number of different hosts, which communicate in a peer-to-peer fashion rather than forcing everything through a central server. Thus they need to be able to find each other at runtime: **This is done using a master which serves as a switchboard for all nodes**. The nodes we talked early can be written in any languages supported by *ROS*. So users can choose the programming language that fit them the most or that fit the most their application. ROS achieves this goal (*language neutral goal*) by specifying interaction at the messaging layer, but no deeper. The direct result of this is a language neutral message processing scheme where different languages can be mixed and matched as desired. The fundamental elements in *ROS-based* system are **nodes, messages, topics, services and actions**.

- **Messages** provides bidirectional(two-way) communication. Each message is sent over a topic. *ROS* utilizes a publish/subscribe paradigm to connect nodes through these messages. For example, IMU data from sensors can be obtained from an offboard computer by subscribing to the topic where the sensor is publishing the *IMU data*. Note that zero, one, or many nodes may publish on a topic, and zero, one, or many nodes may subscribe to a topic. Also it is important to precise that these nodes are communicating asynchronously.

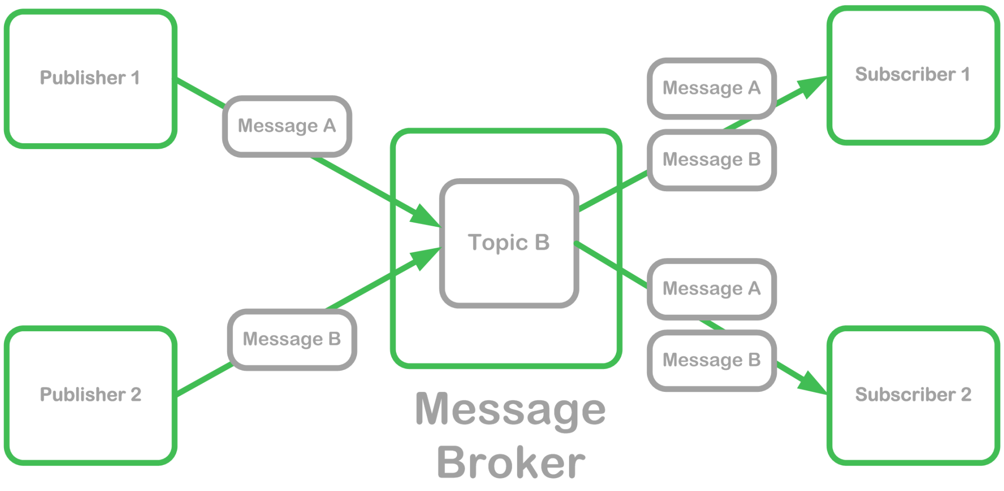


Figure 2.1: ROS publisher/subscriber illustration

- **Topics** are named buses over which nodes exchange messages. Topics have anonymous publish/subscribe semantics, which decouples the production of information from its consumption. They are intended for unidirectional and streaming communication.

- **Services** also provide bidirectional communication. A service can be thought of as a function that can be called on the same or different computers/processes. It can be called with specific messages as input parameters (referred to as the request) and will respond with the appropriate message as output value (referred to as the response). One or many services are implemented and executed within a single node, called *the server* of these services. Other services may be implemented in other server nodes. Service requests come from one or many calling nodes, referred to as clients of the service. Thus, only a single server node will respond to any number of client requests for a particular service (Contrast this with messages, which can be published from many nodes). Service calls can be used synchronously or asynchronously.

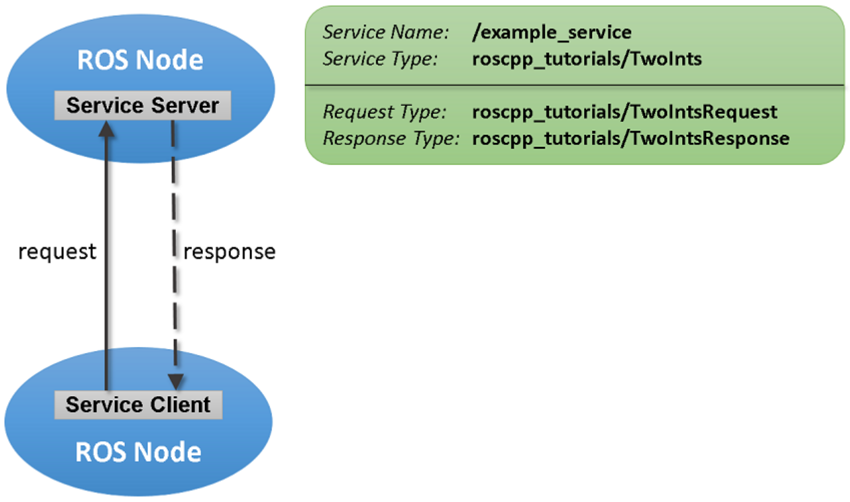

ROS Node
Service Server
request
response
Service Client
ROS Node
Service Name: /example_service
Service Type: roscpp_tutorials/TwoInts
Request Type: roscpp_tutorials/TwoIntsRequest
Response Type: roscpp_tutorials/TwoIntsResponse

Figure 2.2: ROS service/client illustration

- **Actions** provide a multi-directional (in this case, three-way) communication. An action is nearly identical to a service in three ways. First, actions typically take much longer to execute, whereas services are practically instantaneous. Second, actions using different but analogous names (an *action goal* is similar to a service request and an *action result* is similar to a service response. Third, because actions are executed for an extended period of time, they also provide *feedback*, information about the status of the action that can be sent at any time. Zero, one, or many feedback messages may be sent during the execution of an action.

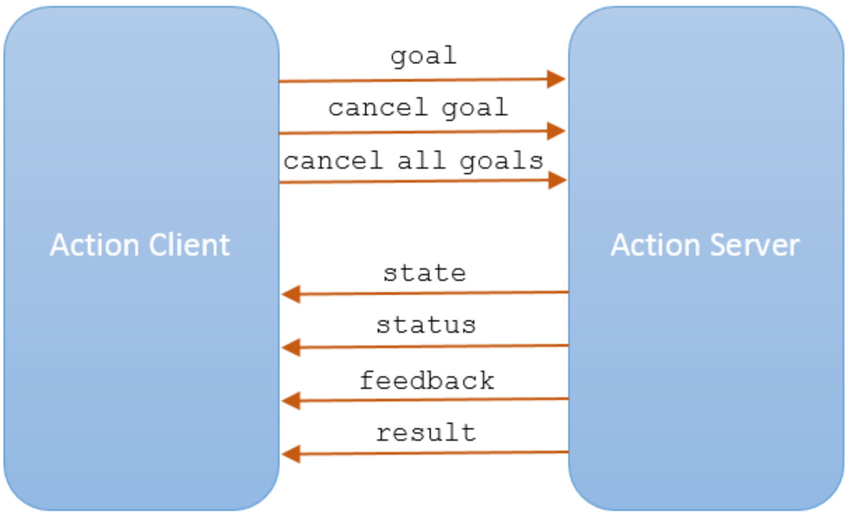


Figure 2.3: ROS actions illustration

More technical information about ROS, the installation procedure and tutorial about how to use it can all be found on their amazing wiki website [4].

2.2 Simulation Environments

If you are having hard time figuring out if you could be a drone pilot or if you have a code to test on a drone and you really don't want to crash your drone or worst, Then you must check out first a drone simulator. Generally speaking, these simulators involves a device that artificially re-creates drone flight and the environment in which it flies. It includes replicating the equations that govern how drones flight, how they react to applications of flight controls, the effect of others vehicles, and how the drone reacts to external factors such as air density, turbulence, wind shear, cloud, precipitation etc... These simulations are often operate in a virtual environment that is realistic and accurate, but without the risks and constraints of a real flight (crash doesn't broke the drone, unlimited battery etc..). We will only use in this report the following two modes of simulation:

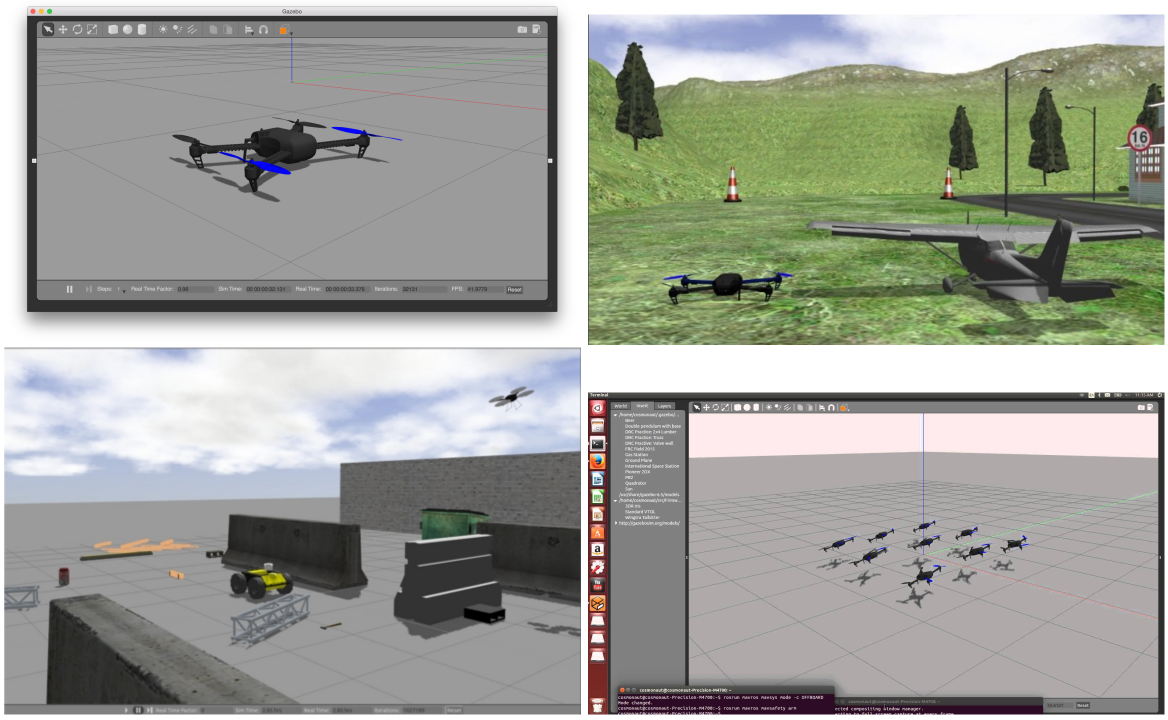
- **HITL** or **Hardware In The Loop**. This is a simulation mode where the autopilot is connected to the simulator and all flight code runs on the autopilot. This approach has the benefit of testing the actual flight code on the real processor.
- **SITL** or **Software In The Loop**. Contrary to the *HITL*, in this mode, you don't need any hardware for simulation. When running in *SITL*, the sensor data comes from a flight dynamic model in a flight simulation.

For our experiments, we mainly used *SITL* with the virtual environments : **Gazebo** and **Unreal Engine**. But you should note that much more open source simulation environment can be found out there like *jMAVSim , DRL Drone Racing Simulator, Hotprops etc...*. The use of one instead of another drone simulator depends on what the user is looking for : Learn how to pilot, or want to implement some scenario etc... Thus what the users want to achieve condition the choice of the drone simulator and its capabilities.

2.2.1 Gazebo Simulation

Gazebo[5] is a 3D simulation environment for autonomous robot. It supports standalone (without ROS) or SITL + ROS. Gazebo offers the ability to accurately and efficiently simulate populations of robots in complex indoor and outdoor environments. In our project, this was the first virtual environment we use since it was the most known working with our flight stack. Basically, We will mainly communicate with gazebo over ROS: Our flight stack in SITL mode will communicate with gazebo using **MAVLink** protocol(which is another protocol usually used to communicate between **GCS** and vehicles). Our offboard controller will communicate with the flight stack on the drone over ROS. This will allow us to control the vehicles in the simulation environment.

More details about offboard controller and gazebo simulation will be found in the flight stack description in the next chapter.



```
cosmonaut@cosmonaut-Precision-M4700:~$ rosrun mavros mavsys mode -c OFFBOARD
Mode changed.
cosmonaut@cosmonaut-Precision-M4700:~$ rosrun mavros mavsafety arm
cosmonaut@cosmonaut-Precision-M4700:~$
```

```
ected compositing window manager.
```

Figure 2.4: Gazebo Environment

2.2.2 Unreal Engine Simulation

Gazebo is a really awesome robot simulation for almost everything but one of the main drawback for our project is the fact that it has limitations in term of photorealism and a legacy physics engine. We then look at a brand new project called **AirSim**[6] which is basically a simulator for drone based on *Unreal Engine*. AirSim plugin is still in active development on their github repository [7]. It is an open-source, cross platform and supports *HITL and SITL* with popular flight controllers such as PX4 for visually and realistic physic simulation. It is developped as an Unreal Engine plugin that can simply be dropped in to any Unreal environment you want (They are working on making it also available for *Unity*). Their goal was to develop AirSim as a platform for AI research to experiment with deep learning, computer vision and reinforcement learning algorithms for autonomous vehicles. For this purpose, AirSim also exposes APIs to retrieve data and control vehicles in a platform independent way.

More information can be obtained on their github website[7]. In this report, Autonomous navigation in static and dynamic environment will be simulate using principally *AirSim and Unreal Engine* due to their graphical environment and physic engine which is really close to the real world environment. The general procedure to simulate using Unreal Engine and AirSim is :

- Follow instruction on AirSim github for Unreal Engine installation and how to install AirSim plugin inside Unreal Engine.

- Choose a graphical environment for your simulation. This can be found on the market place of Unreal Engine and depending on you computer performance, you will be able to play around with fancy landscape.

- Launch the graphical environment and launch the AirSim mode depending on the flight controller and the simulation mode desired. Now you should refer to your flight controller instructions to see how to interact with the vehicle in the simulation environment.

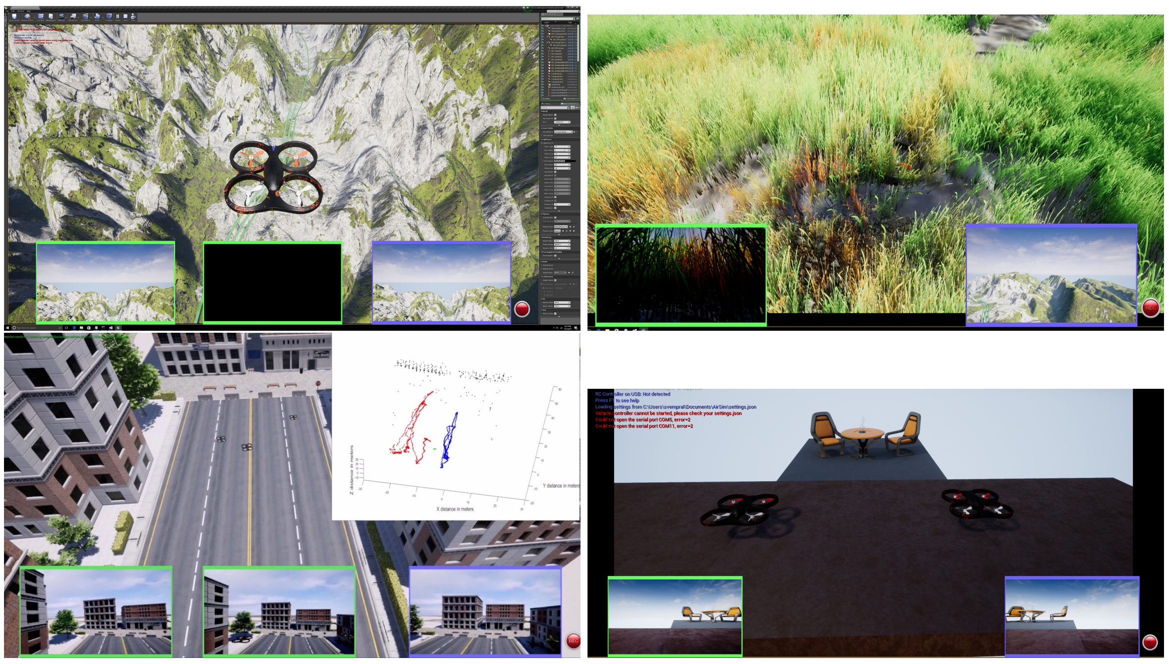


Figure 2.5: Airsim and Unreal Engine Simulation Environment

2.3 *VICON* System

***VICON*[2] is a real time , low latency, with a millimeter accuracy motion tracking system**. Perfectly optimized 3D accuracy is at the heart of robotics tracking systems, so we used VICON for position and orientation estimation for our indoor test with the drone. Basically, a set of optical systems are placed in an arena with the objective to cover a maximum of area. Then the data captured from image sensors are used to triangulate the 3D position and the orientation of an object in the arena. Note that the data acquisition is traditionally implemented using special markers attached to an object as shown in the figure below:

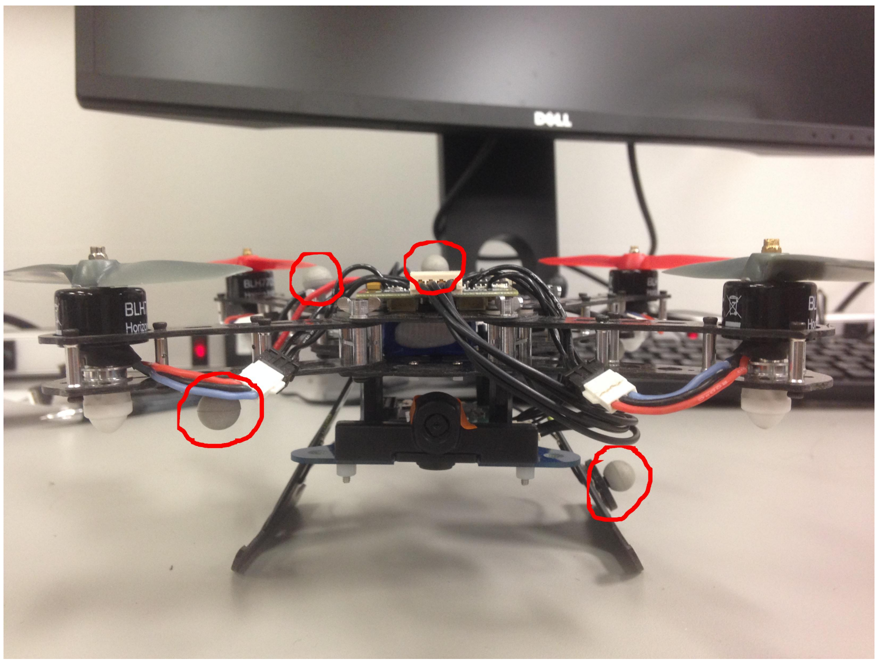


Figure 2.6: Vicon Marker illustration

It is also important to specify that retrieving position and orientation data at a certain rate can be done using **ROS** topic system connected to the Vicon 'server' machine. Finally Vicon gives users a practical 3D interface to visualize all tracked objects as in the figure below:

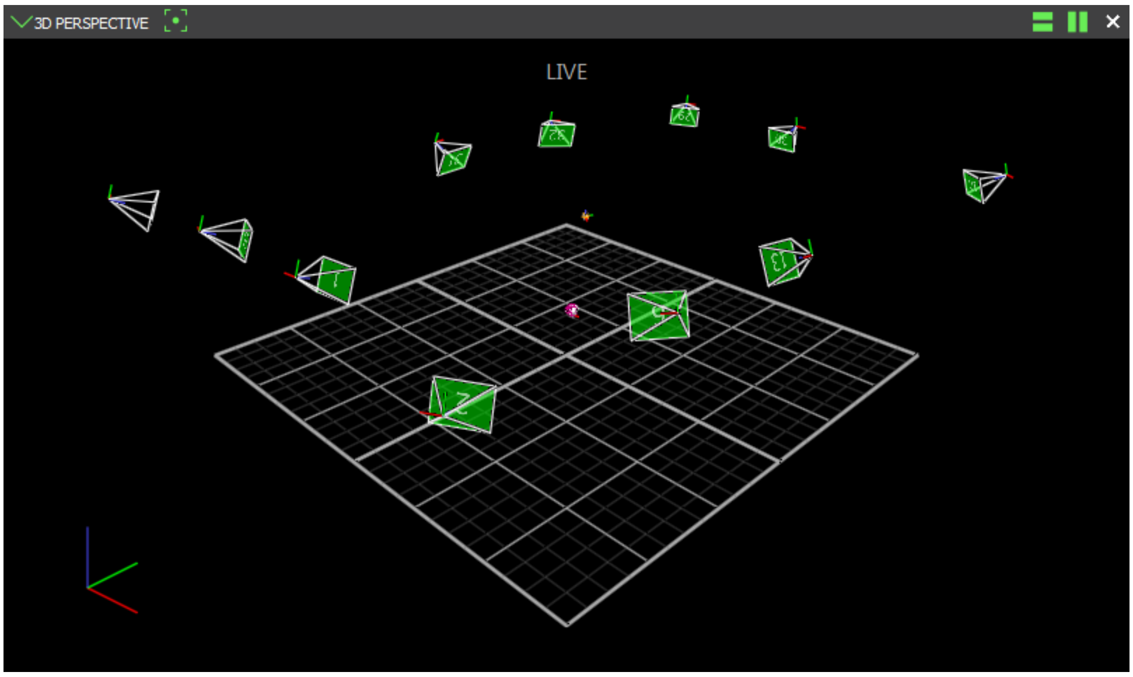


Figure 2.7: Vicon Tracker software

Chapter 3

Quadcopter Design and Control

The designing of an autonomous vehicle is a complicated and comprehensive task. Generally speaking, the following different steps have to be completed for getting the overall design of a quadcopter:

- *Filter design*: All of the sensors provide raw data, which contains unwanted noise. To reduce the noise levels, there are several different filters which can be applied to the sensor readings. The most known filters are **Low-pass Filter, High-pass Filter, Complementary Filter and Continuous-discrete EKF**.

- *Estimation of Roll and Pitch*: In order to control the quadcopter roll and pitch angles, the angles must be known. None of the sensors measure the angles directly, therefore an estimate has to be obtained from the *IMU*. There exists several estimation schemes for the roll and pitch angles. A well known estimator for this case is again the EKF.

- *Estimation of Yaw*: In order to control the quadcopter yaw angle directly, an estimate has to be obtained from the magnetometer and/or gyroscope sensor. These sensors readings are dependent on the roll and pitch angles, and relies on the estimates found for roll and pitch. For our built drone, We trust more measurement of Yaw coming from the Vicon system than magnetometer measurement.

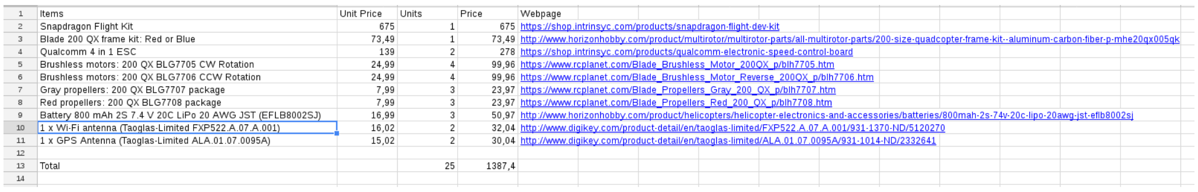
- *Estimation of position*: This is generally done by a **sensor fusion algorithm based on internal data coming from GPS, IMU, Ultrasonic sensor, barometer, optical flow sensors, or external data from vision/mocap system**. The flight stack we use mainly estimate position, velocity based on EKF.

- *Control of attitude*: If the estimates schemes of the attitude are accurate, the attitude can be controlled by varying the inputs of the motors. A difference in PWM input for the four motors is the source of this attitude control..
- *Control of Position*: If the estimates schemes of the position is accurate, the position can then be controlled by a *PID* controller that turns the position setpoint into attitude output.

In the following sections of this chapter, we will present the hardware and flight stack we chose for our quadcopter, then we will talk about our off-board controller and we will finish the chapter with showing how a good control can lead to a nice demo of a flying quad with only eye tracking and head orientation.

3.1 Quadcopter development and construction

In order to achieve our autonomous navigation, we obviously need to build the actual physical quadcopter. This can be achieve thanks to the following mechanical components which are the strict minimum required for building a flying quadcopter.



1	Items	Unit Price	Units	Price	Webpage
2	Snapdragon Flight Kit	675	1	675	https://shop.intrinsyc.com/products/snapdragon-flight-dev-kit
3	Blade 200 QX frame kit: Red or Blue	73,49	1	73,49	http://www.horizonhobby.com/product/multirotor/multirotor-parts/all-multirotor-parts/200-size-quadcopter-frame-kit--aluminum-carbon-fiber-p-mhe20qx005qk
4	Qualcomm 4 in 1 ESC	139	2	278	https://shop.intrinsyc.com/products/qualcomm-electronic-speed-control-board
5	Brushless motors: 200 QX BLG7705 CW Rotation	24,99	4	99,96	https://www.rcplanet.com/Blade_Brushless_Motor_200QX_p/blh7705.htm
6	Brushless motors: 200 QX BLG7706 CCW Rotation	24,99	4	99,96	https://www.rcplanet.com/Blade_Brushless_Motor_Reverse_200QX_p/blh7706.htm
7	Gray propellers: 200 QX BLG7707 package	7,99	3	23,97	https://www.rcplanet.com/Blade_Propellers_Gray_200_QX_p/blh7707.htm
8	Red propellers: 200 QX BLG7708 package	7,99	3	23,97	https://www.rcplanet.com/Blade_Propellers_Red_200_QX_p/blh7708.htm
9	Battery 800 mAh 2S 7.4 V 20C LiPo 20 AWG JST (EFLB8002SJ)	16,99	3	50,97	http://www.horizonhobby.com/product/helicopters/helicopter-electronics-and-accessories/batteries/800mah-2s-74v-20c-lipo-20awg-jst-eflb8002sj
10	1 x Wi-Fi antenna (Taoglas-Limited FXP522.A.07.A.001)	16,02	2	32,04	http://www.digikey.com/product-detail/en/taoglas-limited/FXP522.A.07.A.001/931-1370-ND/5120270
11	1 x GPS Antenna (Taoglas-Limited ALA.01.07.0095A)	15,02	2	30,04	http://www.digikey.com/product-detail/en/taoglas-limited/ALA.01.07.0095A/931-1014-ND/2332641
12					
13	Total		25	1387,4	
14					

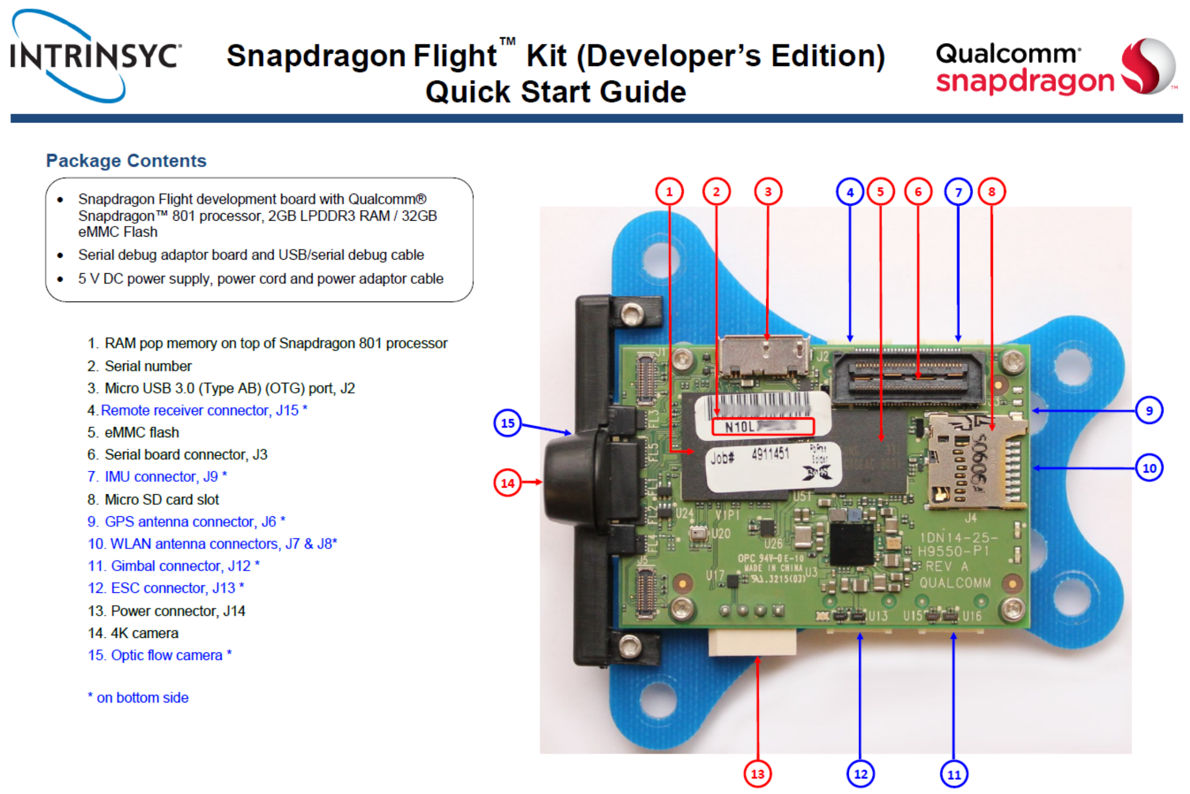
Figure 3.1: Quadcopter main components

3.1.1 Qualcomm Snapdragon Flight

For our project, we chose as flight controller board the *Snapdragon Flight* board built by ***Qualcomm Technologies, inc***[8]. *Why this board* ? Mainly because *PX4 was our target flight stack* and the fact that the Snapdragon board was a sort of **All in One board** had a really huge weight in our decision. *In fact Snapdragon Flight brings together photography, navigation and communication technologies in a compact and efficient package that fits on a single board. Obviously this helps reduce the size, weight and power consumption of drones that use board, which in turn supports longer flight times, safety, and easy to use form factors for consumer*.

In accordance to the Qualcomm website description, The Snapdragon Flight features advanced processing power, real-time flight control on the *Qualcomm Hexagon DSP*, built-in 2x2 Wi-Fi and Bluetooth connectivity, and *Qualcomm SirfStar V* global navigation satellite system (GNSS) optimized to support highly accurate location positioning. It's designed to provide the advanced features that drone consumers are looking for when making purchasing decisions, including:

- **4K Video** : 4K high resolution camera support, image enhancement and video processing capabilities with simultaneous 720p encoding for First Person View.
- **Advanced Communication and Navigation** : Dual-band 2x2 802.11n Wi-Fi, Bluetooth 4.0, and 5 Hz GNSS location capabilities with advanced real-time flight control on Hexagon DSP.
- **Robust Camera and Sensor Support** : Integrated 4K stereo VGA, optic flow cameras, inertial measurement unit (IMU), barometer sensor support and ports for additional sensors (camera options include a 4K forward-facing camera, stereo (depth-sensing) cameras, and a downward-facing "optic flow" camera).



INTRINSYC

Snapdragon Flight™ Kit (Developer's Edition)
Quick Start Guide

Qualcomm snapdragon

Package Contents

- Snapdragon Flight development board with Qualcomm® Snapdragon™ 801 processor, 2GB LPDDR3 RAM / 32GB eMMC Flash
- Serial debug adaptor board and USB/serial debug cable
- 5 V DC power supply, power cord and power adaptor cable

1. RAM pop memory on top of Snapdragon 801 processor
2. Serial number
3. Micro USB 3.0 (Type AB) (OTG) port, J2
4. Remote receiver connector, J15 *
5. eMMC flash
6. Serial board connector, J3
7. IMU connector, J9 *
8. Micro SD card slot
9. GPS antenna connector, J6 *
10. WLAN antenna connectors, J7 & J8*
11. Gimbal connector, J12 *
12. ESC connector, J13 *
13. Power connector, J14
14. 4K camera
15. Optic flow camera *

* on bottom side

Figure 3.2: Qualcomm Snapdragon Flight board

Finally, The Snapdragon Flight supports Linaro Linux, so users can installed for example ROS , OpenCV, flight stack and more software on the board. Moreover, Snapdragon Flight is part of the Qualcomm Drone Development Platform that includes both hardware and advanced software modules for navigation, optical flow, gimbal control, and it utilizes Qualcomm Technologies strong computer vision expertise for features such as object avoidance, following an object or person, and more. More information about this board can be obtained on Snapdragon section of Qualcomm website[8].

3.1.2 Qualcomm E.S.C

An ESC is an electric circuit with the purpose to vary an electric motor speed. The ESC has to be fast and reliable. Usually ESC takes as input PWM command from the main board and use it to control the motors.

Qualcomm Technologies recommends their official ESC to use with the Snapdragon board. Their ESC is a *4 in 1 Electronic Speed Controller* which uses UART. This is not the case for ESC often used on the market. At the time we started to work with Snapdragon, no solution had been implemented for using off-the-shell ESC. So we had no choice to use the Qualcomm UART ESC. It is a fast and reliable ESC but the main drawback is that **the ESC is specced for maximum 7 amps of continuous current and 10 amps burst current per motor. In reality, this specifications limits the choice of the motors**.

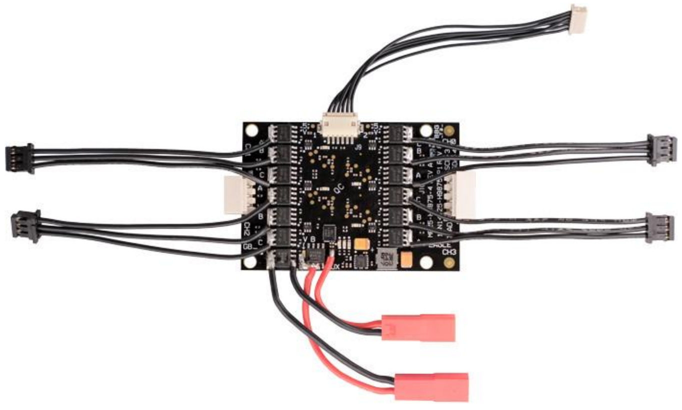


Figure 3.3: Qualcomm 4 in 1 ESC

Now it is possible to use off-the-shell ESC. You can either use a UART to PWM bridge or use the PWM support recently added and documented by Qualcomm. Obviously this allows you to have more option on the motors for your drone?. You can find more information about this functionality on PX4 developer website with their custom drone with Snapdragon board[9].

3.1.3 Motors, Frame, Propellers and Battery

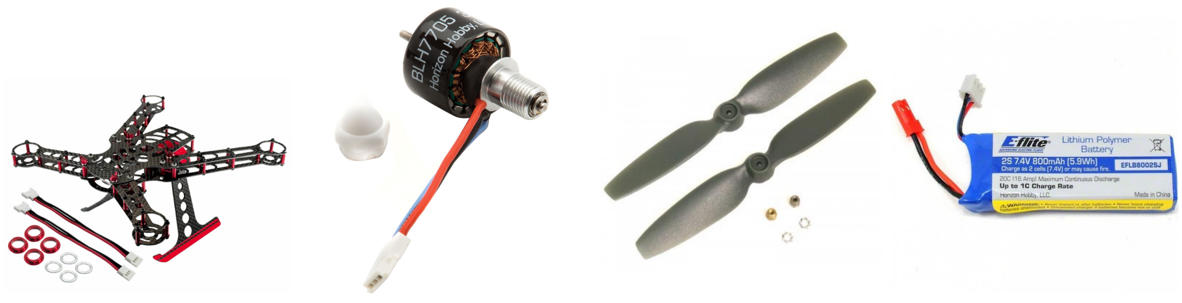


Figure 3.4: motors-frame-propellers-battery for the quadcopter

Qualcomm recommends the use of the well known *Blade 200 QX* as the frame to use with Snapdragon Flight. Basically, this frame is made by Aluminium and Carbon fiber. It weights only 53.8g. The recommended motors with this frame are the *BLH 7705/7706 series* which draw less current than what can be furnished by the Qualcomm ESC. Looking at the dimension of the frame and the choice of the motors, the choice for the propellers is limited and the *BLH 7707/7708 series* is the choice we made. Finally, as a battery we chose a *LiPo 2S, 800 mAh, 20C with a 20 AWG JST connector* (Also recommended for the 200 QX frame).

3.1.4 Extras components and Final results

With all the components mentioned previously, it is possible to realize the same experiments we made in our project. However, if you want to do some outdoor flight with position control, you obviously need a GPS. We propose here to add the *Taoglas-Limited ALA.01.07.0095A (GPS Antenna)* to the Snapdragon board. Moreover, Since all the operation and all the offboard control will be done over WIFI, For increasing the range of the wifi and reduce eventual problem link to WIFI lags, it is recommended to add two WIFI Antennas. We added the *Taoglas-Limited FXP522.A.07.A.001* to all our built Quadcopter. Finally, you may need to buy **screws and standoffs for fixing the Snapdragon board on the frame**.

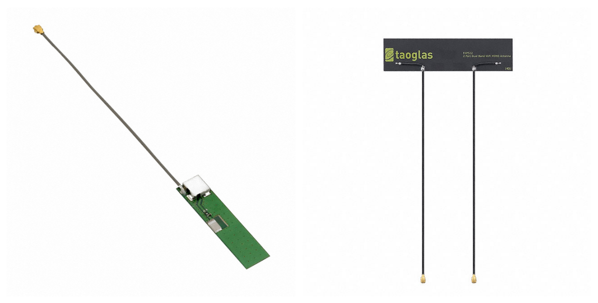


Figure 3.5: GPS (left) and WIFI (right) Antenna

Vibration damper should also be bought for security. After months, for some reason the position estimation was damage by huge variation of acceleration given by the accelerometer. We used vibration damper to correct the problem after we identified that it was a vibration problem. What happened was that, we didn't store well our Snapdragon board and the IMU get more sensible to vibration. So it is not a bad investment to have vibration damper store somewhere. In the next figure, You can see the quadcopters we built based on the components specified here. **The flight duration of this quadcopter is about 8 min without the use of the cameras and it drops down to 6min with the two cameras turned ON**.

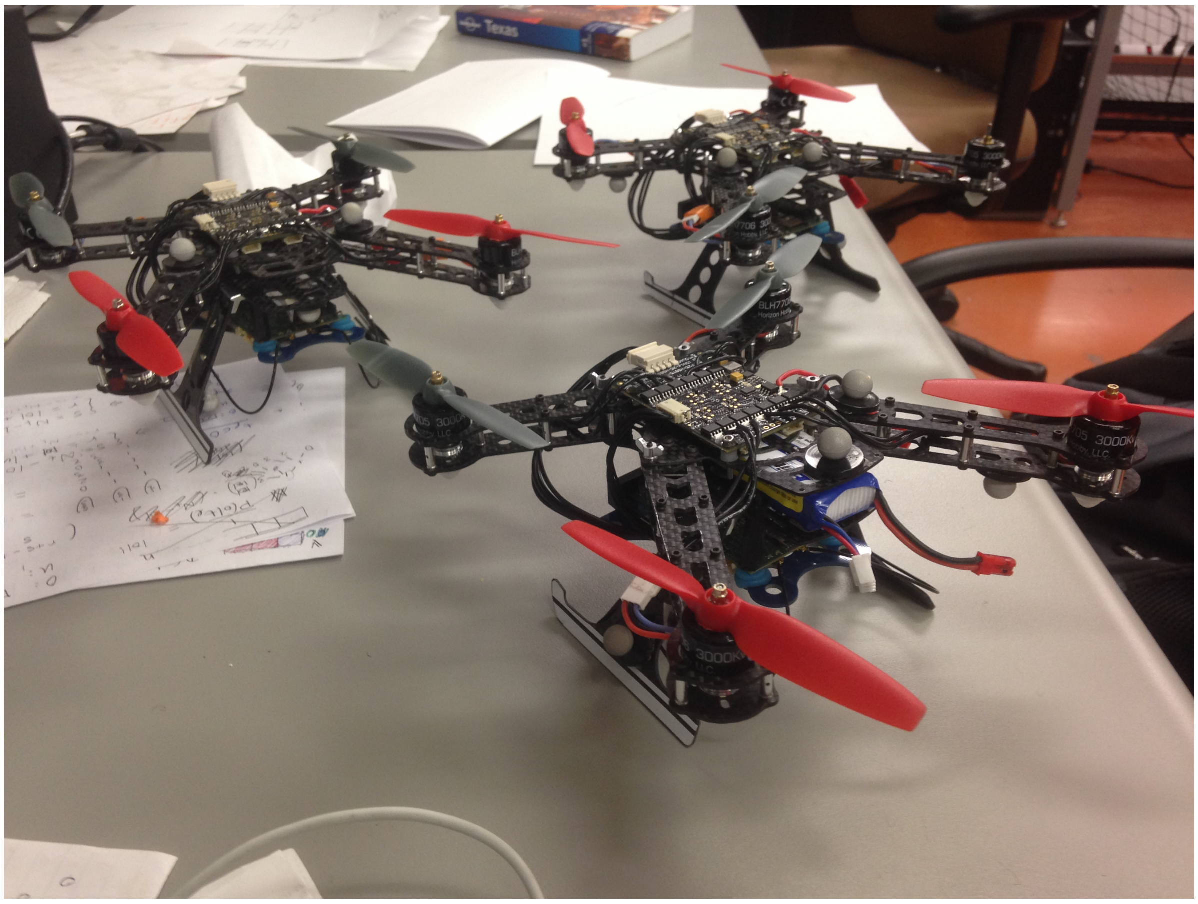


Figure 3.6: Final Quadcopters Version

3.2 PX4 Autopilot

PX4 autopilot platform is a low cost, modular, open hardware and software design targeting high-end research, hobby and industrial autopilot applications. It is an expandable, modular system comprising **PX4 Flight stack** (the autopilot software solution) and **PX4 Middleware** (it support any type of autonomous robot). Moreover PX4 introduces a sophisticated, modular software environment built on top of a POSIX-like realtime operating system (**NuttX Real-Time Operating System** or **QuRT/Hexagon** for the Snapdragon board). The modular architecture and operating system support greatly simplify the process of experimenting with specific components of the system, as well as reducing the barriers to entry for new developers. Adding support for new sensors, peripherals and expansion modules is straightforward due to standardized interface protocols between software components.Note that The **uORB**, used by PX4 for internal communication between components, is an asynchronous publish/subscribe messaging API used for inter-thread/inter-process communication. Onboard microSD storage permits high-rate logging and data storage for custom applications. **MAVLink protocol support** provides direct integration with existing ground control systems including **QGroundControl**. The Mavlink protocol support gives also the possibility to do *SITL*/*HITL* with an external simulation environment such as the one presented in the previous chapter. Here we just give a global overview of PX4. for more details, PX4 developer website[10] is their crazy documentation website where you can find almost everything if you want to use PX4 as your autopilot.

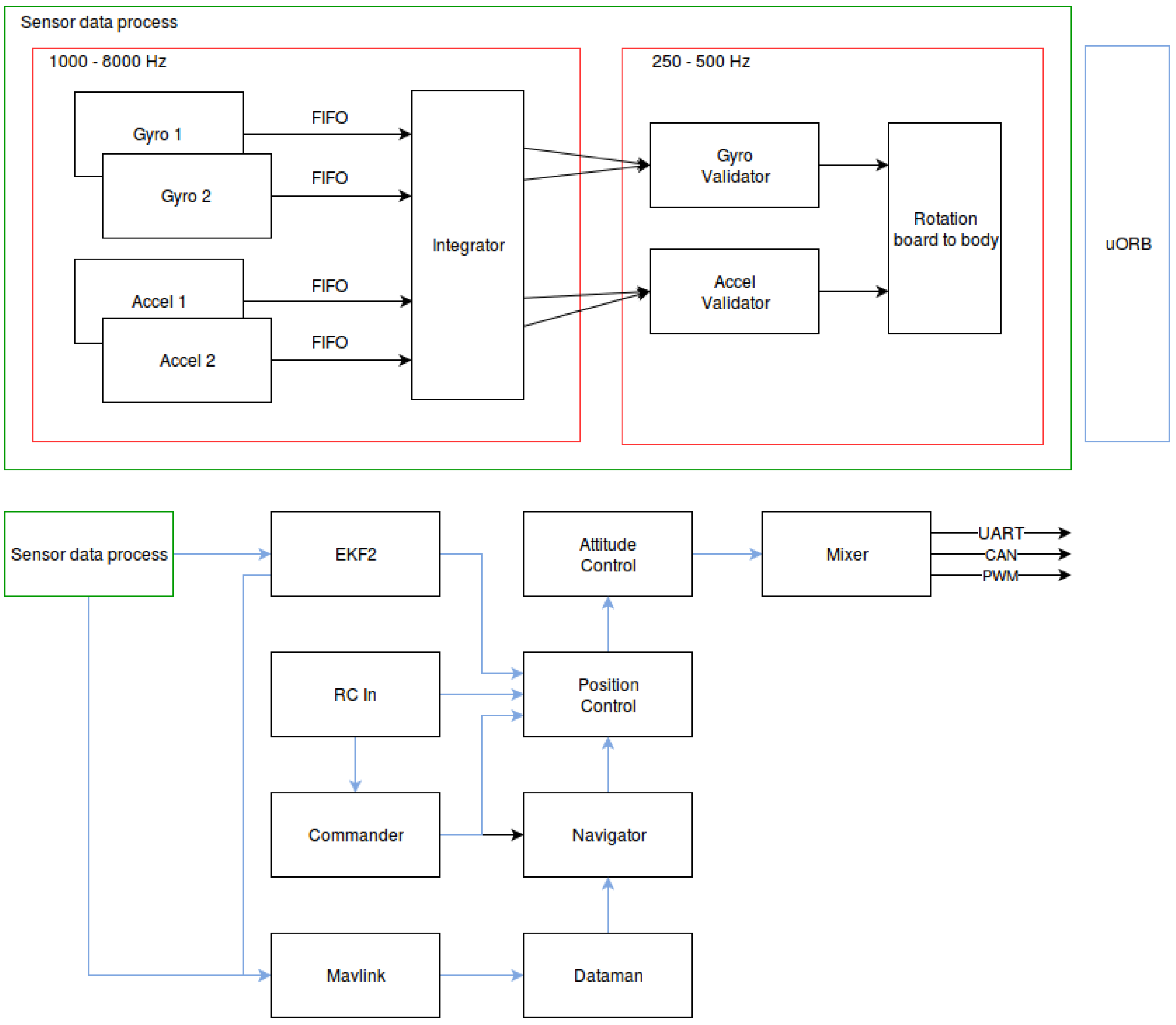


Figure 3.7: PX4 Autopilot architecture

3.2.1 PX4 Middleware

The PX4 Middleware consists of primarily of device drivers for embedded sensors and a publish-subscribe based middleware to connect these sensors to applications running the flight controls. The use of the publish-subscribe scheme means that:

- *The system is reactive*: It will update instantly when new data is available.
- *It is running fully parallelized*.
- *Data consumption from any component*: A system component can consume data from anywhere in a thread-safe fashion.

3.2.2 PX4 Flight Stack

The PX4 flight stack is a collection of guidance, navigation and control algorithms for autonomous drones. It includes controllers for fixed wing, multirotor and VTOL airframes as well as estimators for attitude and position. The estimation and control Architecture is shown in its simple form on the next figure. Note that for our project: *GPS was replaced with VICON system*.

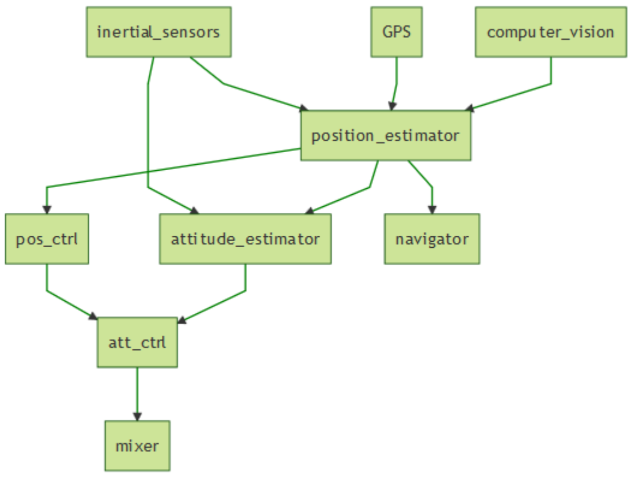


Figure 3.8: PX4 Flight stack

3.2.3 PX4 Simulator

Simulators allow PX4 flight code to control a computer modeled vehicle in a simulated world. **You can inyteract with this vehicle just as you might with a real vehicle** using QGroundControl, an offboard API, or a radio controller/gamepad. All simulators communicate with PX4 using the Simulator MAVLink API. This API defines a set of MAVLink messages that supply sensor data from the simulated world to PX4 and return motor and actuator values from the flight code that will be applied to the simulated vehicle. The messages description can be find on the PX4 developer website[10].

The diagram below shows a typical SITL simulation environment for any of the supported simulators. The different parts of the system connect via UDP, and can be run on either the same computer or another computer on the same network.

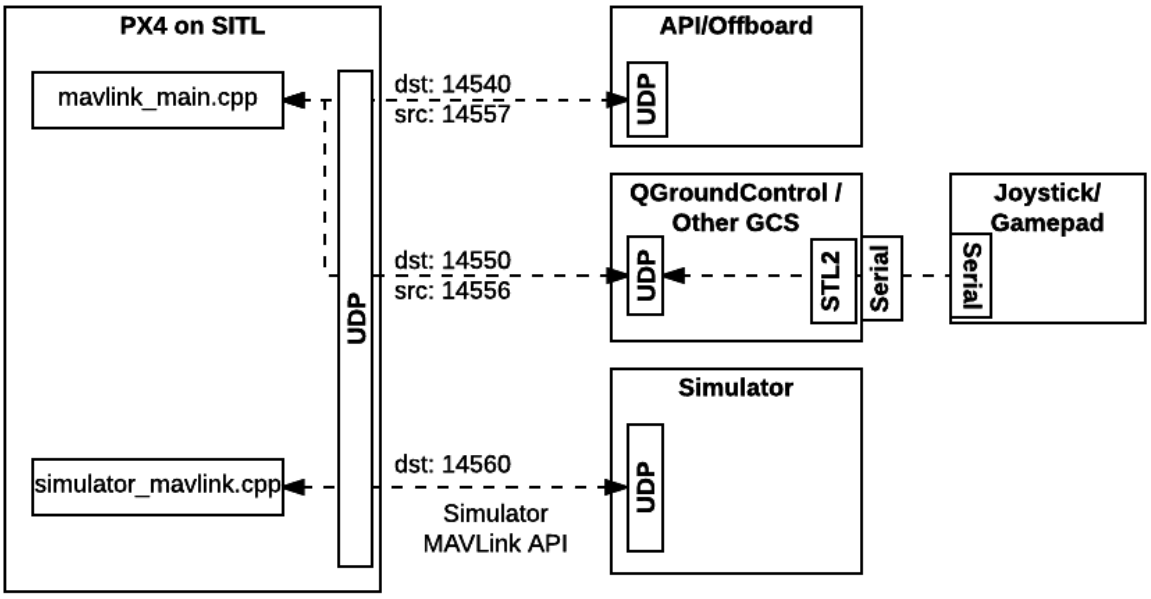


Figure 3.9: SITL Simulation environment with MAVLINK

3.3 Off-board Control

The idea behind off-board control is to be able to control the PX4 flight stack using software running outside the autopilot. This is mainly done through the Mavlink protocol. In our case, since Snapdragon Flight already includes WIFI, The communication with PX4 built on Snapdragon is done using **mavros**. Basically, what mavros does is to translate every mavlink message from PX4 or to PX4 in a ROS topic/service that can be used on an off-board computer connected with mavros to the Snapdragon Flight.

Our goal here is to make a high level ROS node that will hide to the user all lower interaction with PX4. Basically, the user will only communicate with that node to control the quadcopter and will get some information about the current state of the quadcopter from that node. Associate to that node is another node made for the visualization of the quadcopter using a ROS tool named RVIZ and finally a third node made for getting back control of the quad using a Joystick connected to the off-board computer via ROS. In the following figure, you can have a partial overview of our off-board system and an example with a joystick connected to it for controlling the quadcopter. The goal of the *C++ off-board node* is to reduce time spent by the user for his application purpose.

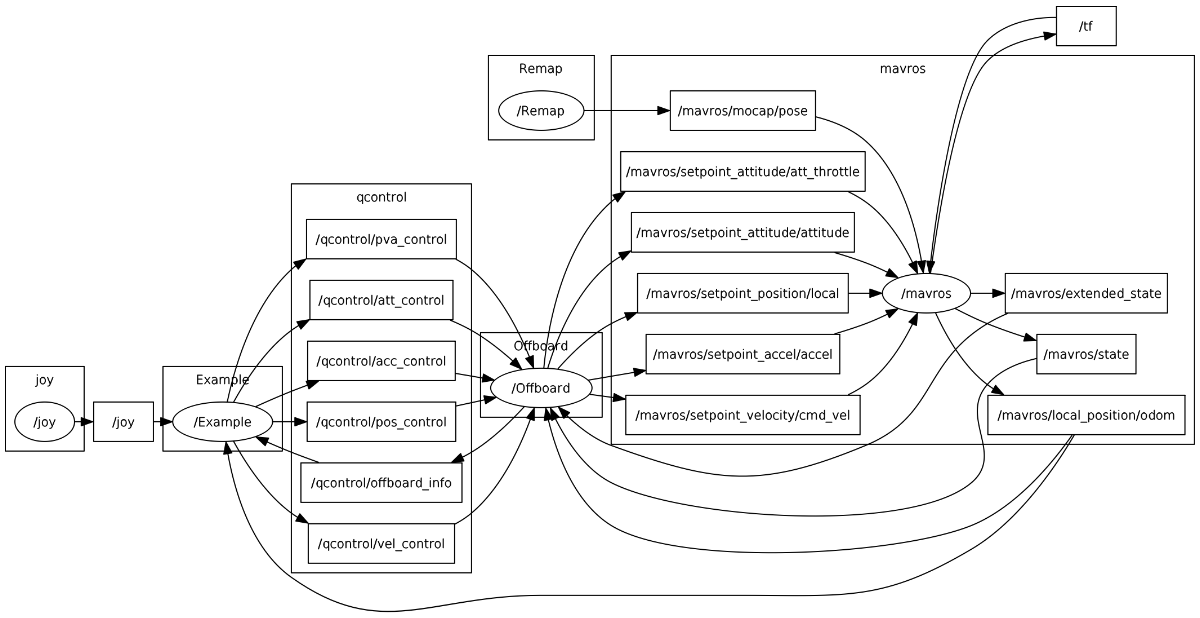


Figure 3.10: Off-board Node connections

First of all, in the previous graph, There is only nodes and topics (ROS rqt_graph package). The Off-board node contains actually two services:

- **CommandAction service**: This is the most important service to call for controlling the quadcopter. Basically, all requests for changing the mode in which the quadcopter is must be done using this service before actually publishing data to do some control. The different states/modes of the off-board node are :

  - *Attitude Control Mode*: In this state, the user must publish attitude target value on the *qcontrolatt_control* topic in order to control the attitude and thrust level of the quadcopter. Some customizations using a ros launch file can be done by the user in order to increase or decrease the rate for publishing data to the main board, the max roll, pitch, yaw,thrust values that will be send to PX4 on the snapdragon board.
  - *Position, Velocity, Acceleration Control Modes*: After requesting one of this 3 modes, the user must publish position or velocity or acceleration target value respectively on the *qcontrolpos_control, qcontrolvel_control, qcontrolacc_control*. They can also specify in the message the frame used by the data. Again, customizations about the rate for publishing back informations to PX4 can be specified in a launch file.
  - *PVA control mode*: This mode will be described in detail in the chapter 5.
  - *Take-off and Land request*: After requesting one of this 2 modes, the quadcopter will take-off or land depending on the request. In the meanwhile, the user can subscribe to a dedicated topic to check if the taking-off or landing procedure have been complete.

Basically, because of **Ground effect** which makes the quadcopter deviates too much on the X,Y axis from his initial position at taking off, we needed our own implementation of taking off based on a simple *PD Controller on the Velocity*. Thus we also decide to use this working controller in order to add a landing functionality. Parameters for these requests are the desired altitude to reach when taking-off or landing to be notify that the request was complete. An improvement to these functionality is to use the PVA Controller that we will present in the chaper 5 because we will have control at the same time over Position and Velocity.

- *Arm/Disarm motors request*: Basically, this is intend to shutdown or start the motors. Obviously, before doing all kind of control, the user is supposed to start the motors. For taking off, No need to do this since the node does it itself. For landing, No need to shutdown the motors because when the altitude threshold is detected, the node shutdown automatically the motors.

- **PVA tunning service**: This service is the one to called for tunning in real-time the PID parameters for the PVA controller. More details about this will be given in the chapter 5.

Basically, the *CommandAction service* message has a lot of fields with three states : *undefined, true and false*. Each of this field can be set in order to make a request. Note that on the node code side, a priority is given to each request in order to not have some erratic behavior. For example, if a user want to turn on the motors and go on position control using the same message, he doesn't have to worry since the node will start by turning on the motors and then open a thread to listen on the position data topic. All these priority cases can be handle by the off-board node. It is important to precise that the node will never listen to two control topics at the same time so it is important to associate a priority to the fields in the service message in order to decide when the user will make a mistake and send a wrong message to the service. Moreover, this off-board node is multi-threaded for obvious reasons. The main thread is by default at 100Hz (But can be changed by the user) and his goal is to enable off-board mode on the PX4 side and gathers information about the quad position, attitude and internal state. All control topic has his own thread that can be start or sleep on a request and **transition between each control is done in a continuous way using the data given by PX4 odometry**.

On the figure above, the node call *remap* feeds PX4 estimator with VICON position and orientation of the quadcopter. Estimation is done by PX4 navigator module and the output of that estimation can be obtained by subscribing to the odometry topic. The node *Example* is just an example of how to control the quadcopter using a Joystick. This can be used as a C++ example for making call to our off-board node. More examples and utilities functions for Python users can be found on the github of *u-t-autonomous group*[11]. The off-board node code[12] and its documentations are still on development on github.

3.4 Estimation and Control

In this section, we will present a procedure for a user to be sure that the estimation of Position , velocity or attitude by PX4 is correct before trying to do an off-board control. Indeed, bad estimation by the PX4 flightstack can happen for a lot of different reasons :

- **ENU and NED coordinate system**: Knowing that PX4 works on the NED coordinate system, some may think that the value to send over ROS must also be in NED. **That's not the case** . They should be in ENU coordinate system and mavros will automatically make the conversion for the user. that allows the user to think in a more natural coordinate system which is the NED coordinate system.
- **Differences between PX4 and VICON attitude estimation**: First it is important to check if the orientation from VICON is used for attitude estimation. Normally, increasing the weight of VICON for yaw estimation is the only thing to do knowing that the Snapdragon magnetometer is not as reliable as VICON and that this can cause some erratic behaviors during the flight. However for pitch and roll it is important to not give an important weight for mocap pitch and roll. As it will be shown in the next section, IMU integrated with Snapdragon are very reliable and there is a lot of factor that can affect VICON orientation (more than position) estimation. For example, bad calibration of VICON, or one of it cameras moved, or the markers are not well placed efficiently or have been moved from the position VICON think it is and more ...
- **Wrong PX4 position estimator tuning**: Since we are using a mocap system for estimating the position, we need to specify this to PX4. That can be easily done by changing some configuration parameters of PX4. First since VICON is reliable, we can set the estimator to only use vicon data as input for the position estimator. For that the parameter *LPE_FUSION* of the *local_position_estimator* must be changed according to this specification. Generally after this step, the odometry topics should show values that are similar to what VICON is sending. However you should notice that **speed estimation is sometime not good**. For fixing it, you should increase the parameter **LPE_PN_V** for forcing the estimator to trust more the data coming from VICON for speed prediction than the model. This should be sufficient for getting a good position and velocity estimation based on VICON.

We include in the github repository the working params file with the parameters we used for Snapdragon board : parameters for attitude and position estimation and PID coefficient for attitude, position and velocity control.

3.4.1 Attitude, Position and Velocity estimations

This is the first step before trying to fly the Quadcopter. Put the quadcopter in the arena surrounded by VICON, arm the motors in order to make PX4 record data, then move the quadcopter around with your hands (be sure to change position and orientation of the quadcopter for getting useful data). Finally make a comparison of the data recorded by the PX4 and the one sent by VICON.

Once this is done, you should get as shown in the next figures almost a perfect match from the position estimator if the tunning of the position estimator have been correctly done.

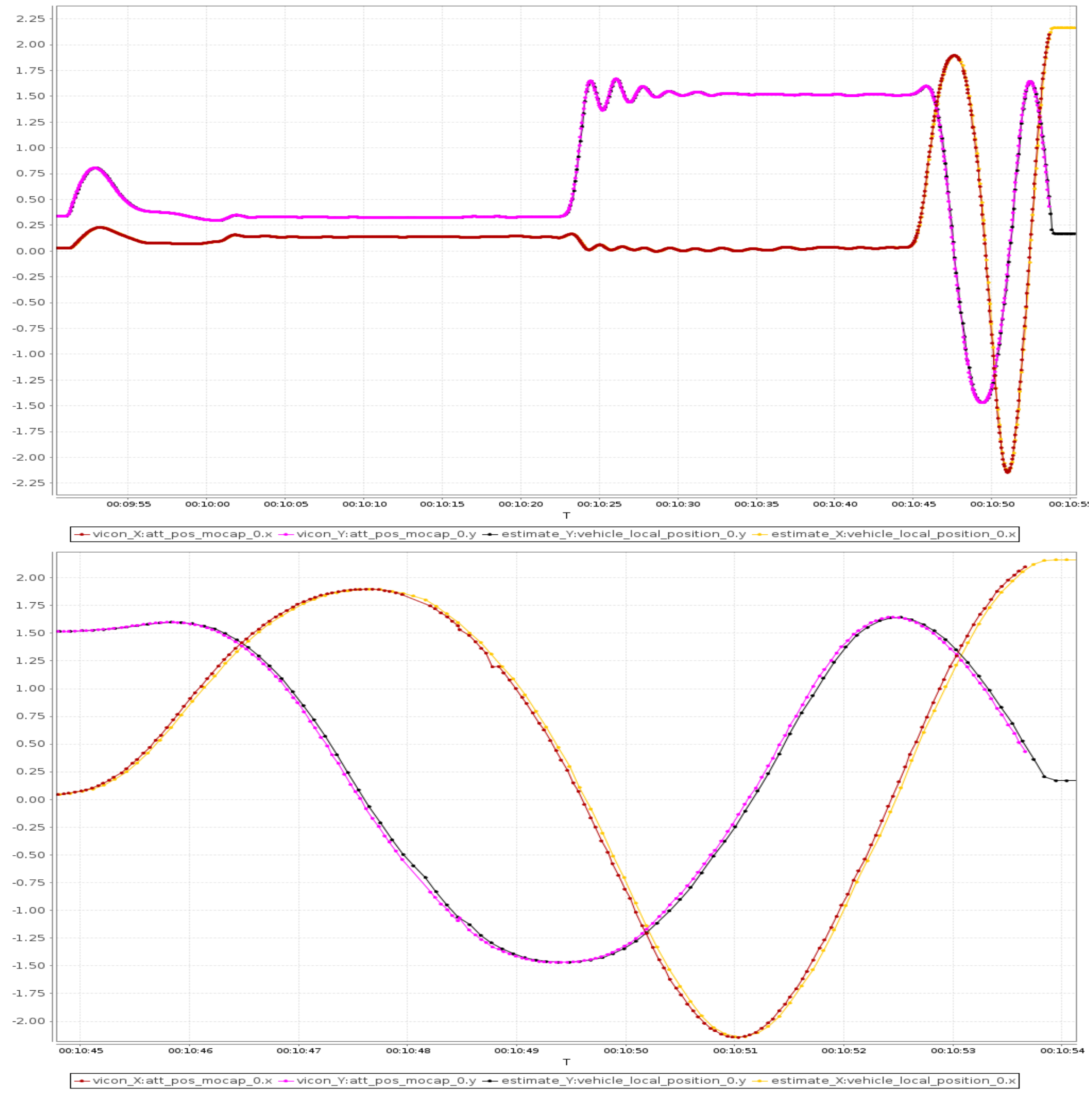


Figure 3.11: XY estimated by PX4 vs VICON estimation.

You can observe on the previous figure how conform is the estimation made by PX4 with the data given by VICON. You can also notice the lack of significant delay between what is given by VICON and the output of PX4 EKF estimator. the next figures show the same result for Z estimation and for the Velocity estimation. The next figure is comparing the velocity on Z axis

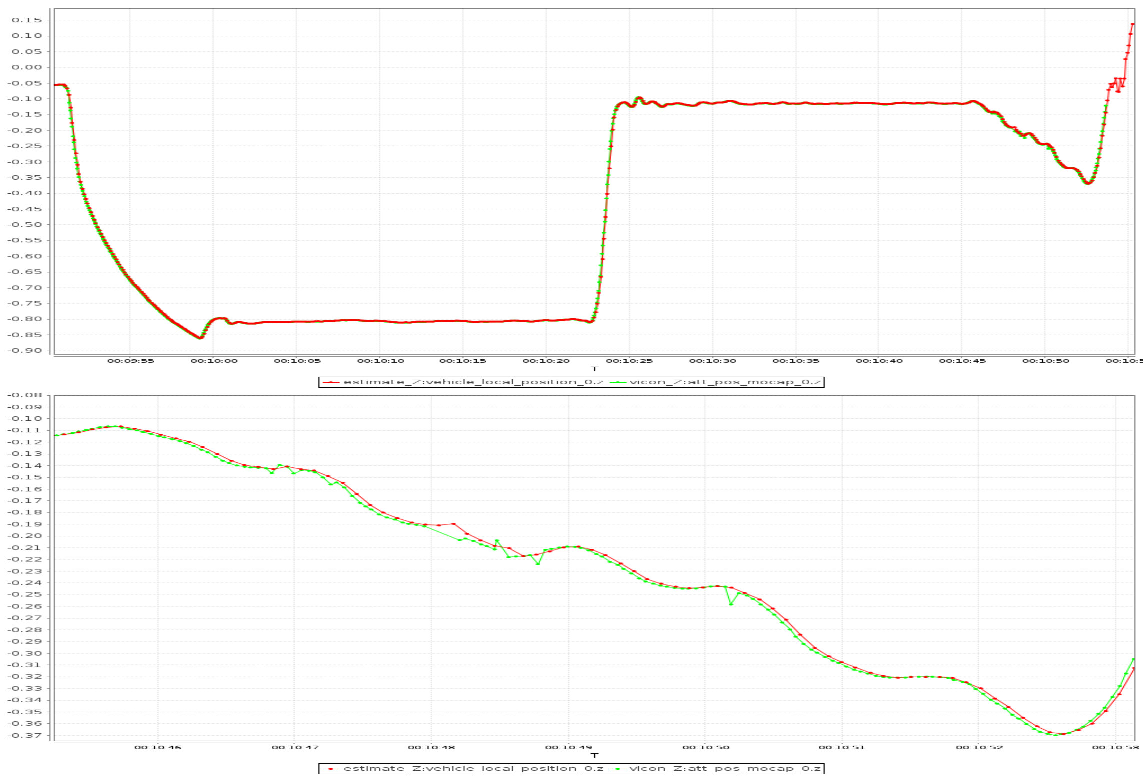


Figure 3.12: Z estimated by PX4 vs VICON estimation.

estimated by PX4 and a simple derivation of Z position sent by VICON. You can observe that the two curves behave in the same way. The only difference appears to be that offset between the two curves. This is something you need to fix before launching the quadcopter. Generally this is due to bad tunning of the *LPE_PN_V* parameters which should often be increased. **The direct consequence of this is obviously a bad control on the Z axis where an offset between the current Z position and the setpoint will be always observed**.

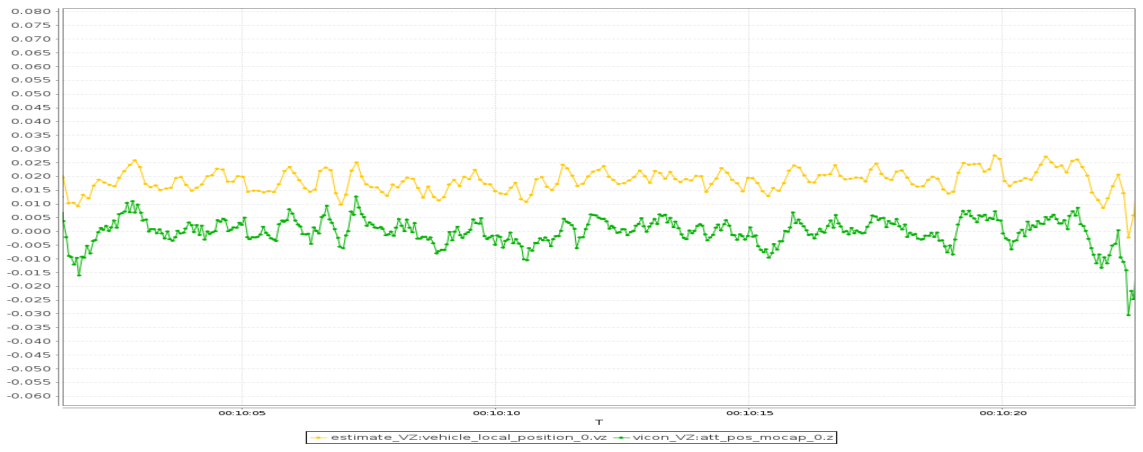


Figure 3.13: VZ estimated by PX4 vs Derivative z from VICON.

The estimation of attitude won't be as perfect as position because of the reasons we explained previously (VICON calibration, markers position slightly changed etc...). However we don't need it to match perfectly what VICON has since VICON can be a source of noise. In fact the IMU inside Snapdragon FLight is reliable at a point that we don't even need to include pitch and roll from VICON into the attitude estimation. The following figures show some differences between value from IMU and value from VICON. The previous figure contains case to avoid. The last plot

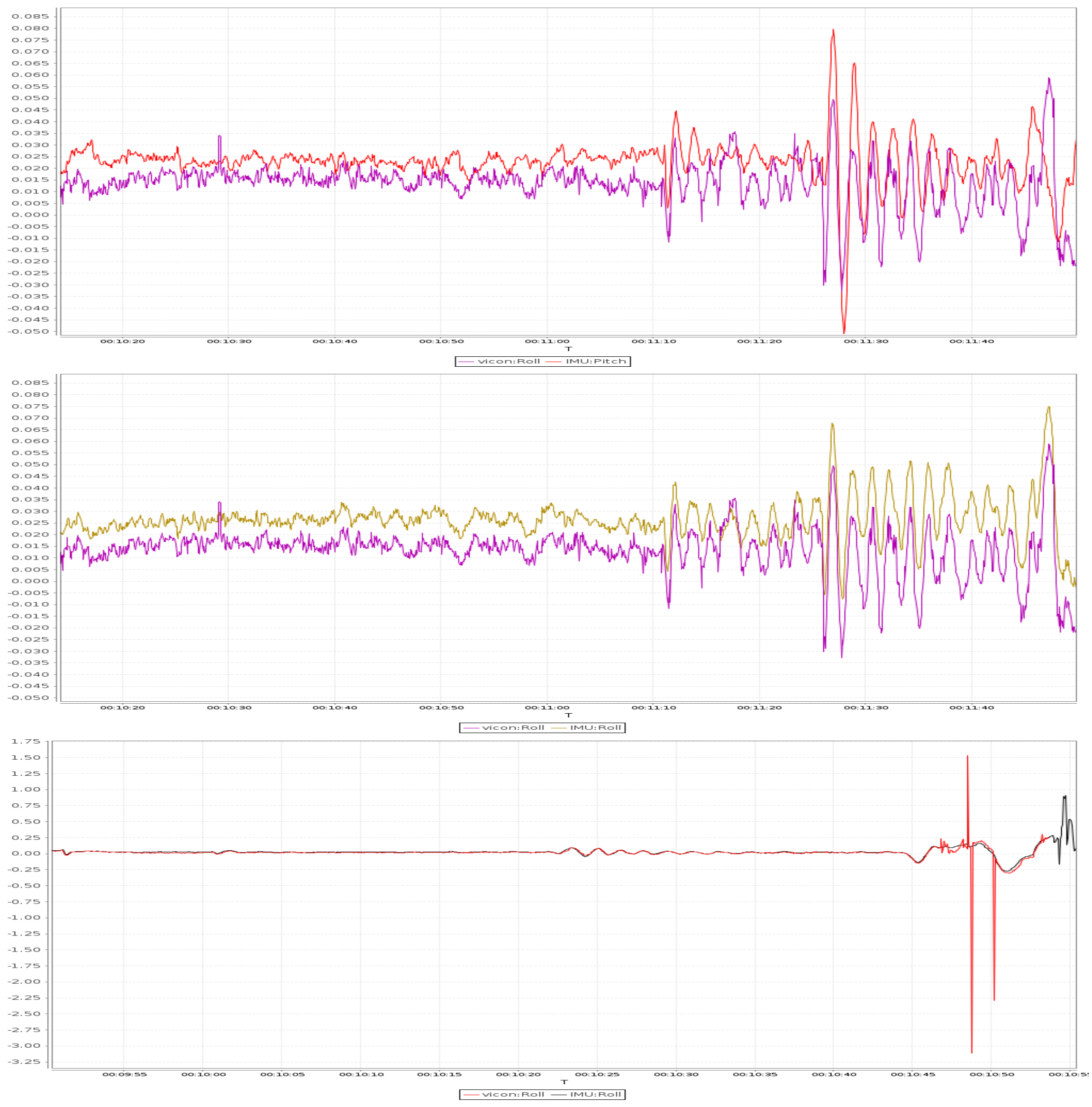


Figure 3.14: Why not using VICON for roll and pitch estimation

contains a case where the two values from VICON and from the IMU are almost identical but at some point VICON send an obvious wrong value (the two Dirac you can see on the plot). This error in particular was due to the fact that markers were placed in a symetric pattern. What happened is that at some point during the flight, VICON thought the quad was reversed. This can affect the yaw value (since we highly rely on VICON) and it estimation but not the roll and pitch estimations.

About the Yaw estimation, we will not use the magnetometer for yaw estimation. Instead, we will set a high weight for external yaw in the estimator. This way, we will have a fast convergence of the estimation by PX4 to the value sense by VICON. The following figures show how it pitch, roll and yaw should look like compared to vicon input.

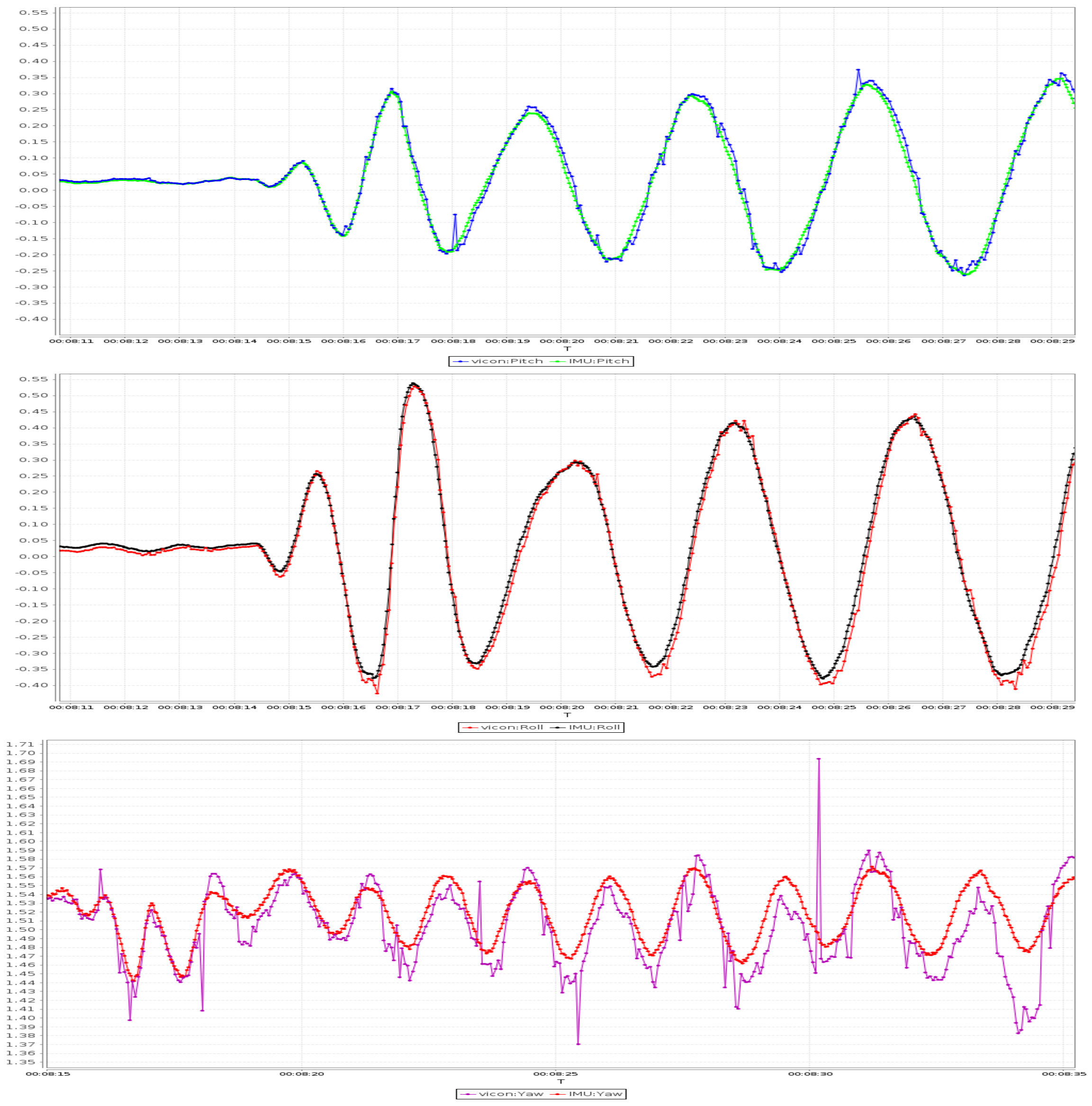


Figure 3.15: Good roll-pitch estimation and fast yaw convergence

3.4.2 Attitude, Position and Velocity Control

PX4 control algorithms are all PID based algorithms. So for having a good control, it is useful to well tune the PID coefficient. *The most known method for tuning PID is the heuristic method of Ziegler-Nichols*[13]. PX4 gives users three ways of tunning its controllers : the params file, QGroundControl application and ROS service call for PID tunning.

Basically, PX4 by default has a pretty much good attitude control. So there is no additional important work to do for tunning the attitude controller as you can see on the figure below:

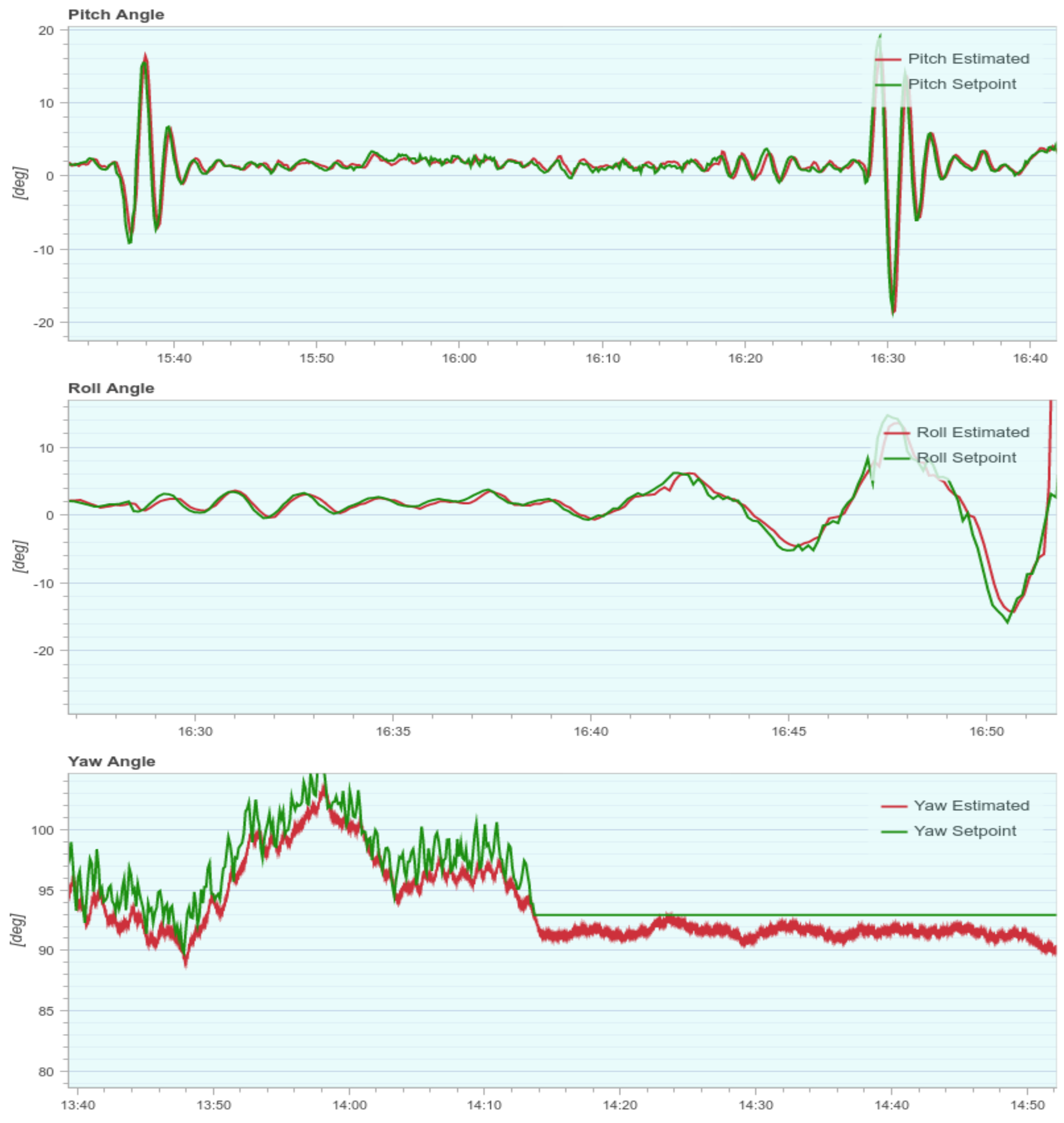


Figure 3.16: Good roll-pitch-yaw Control

However, in our case, since the yaw input is from VICON instead of the magnetometer, a simple tunning have been done here for good control of the yaw. As you can see in the previous figure, attitude control is really good and will constitute the base for our PVA controller in the chapter 5.

For position or velocity control, it is important to tune the controller since our input are from VICON instead of GPS (and other sensors). This step can require a lot of time depending if you manage to make the tunning during the flight (this will save you a lot of time). We didn't allocate that much time and resource in our position controller tuning.

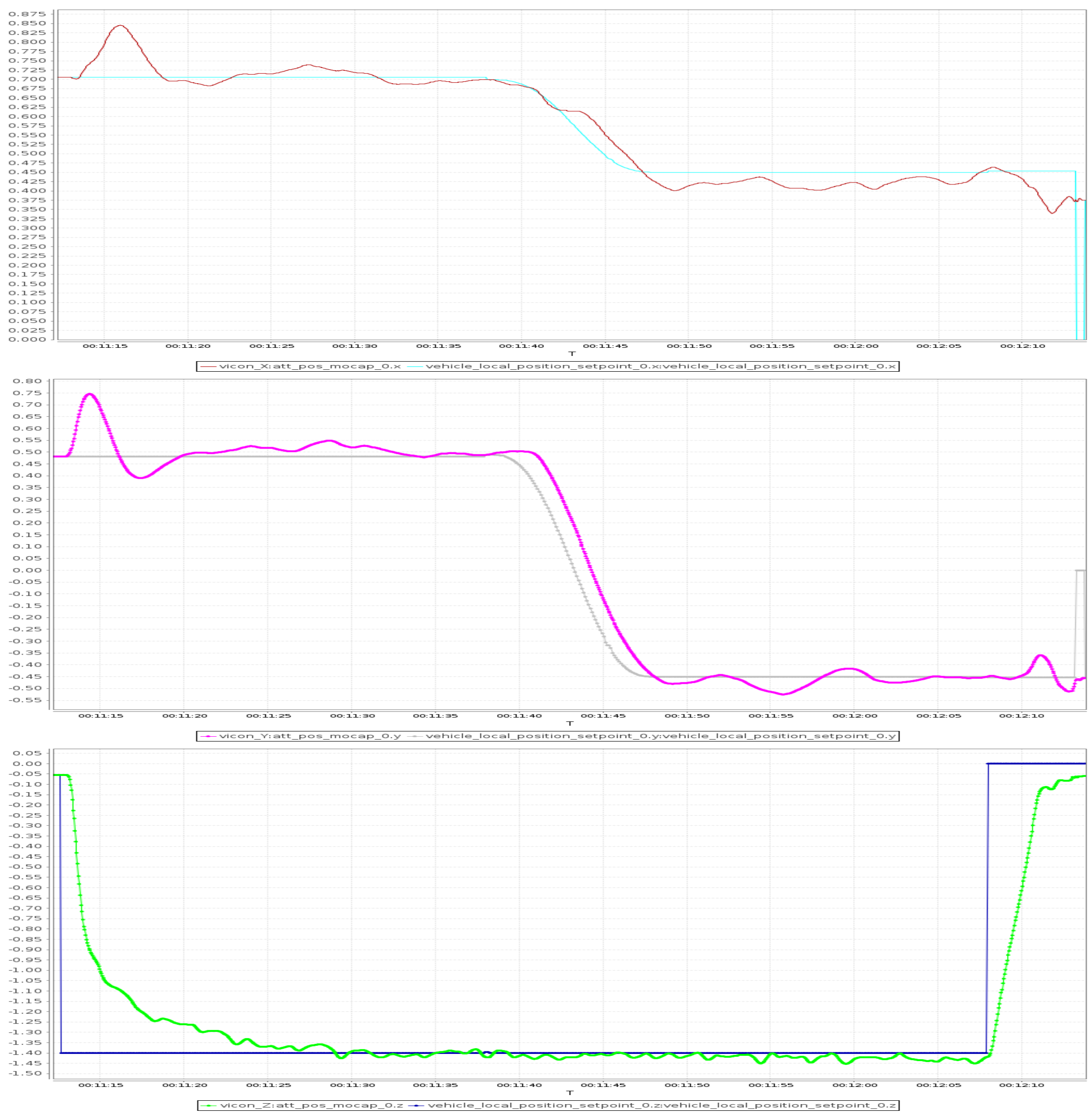


Figure 3.17: From top to bottom, X, Y and Z control

Note that here we have an accuracy of +/- 5cm on Z axis , +/- 10cm on XY axis. Good tunning will lead to better control on this board using VICON.

Finally, tunning velocity controller is obviously needed for off-board velocity control. In the next figure, you can see good results we obtained when tunning this controller:

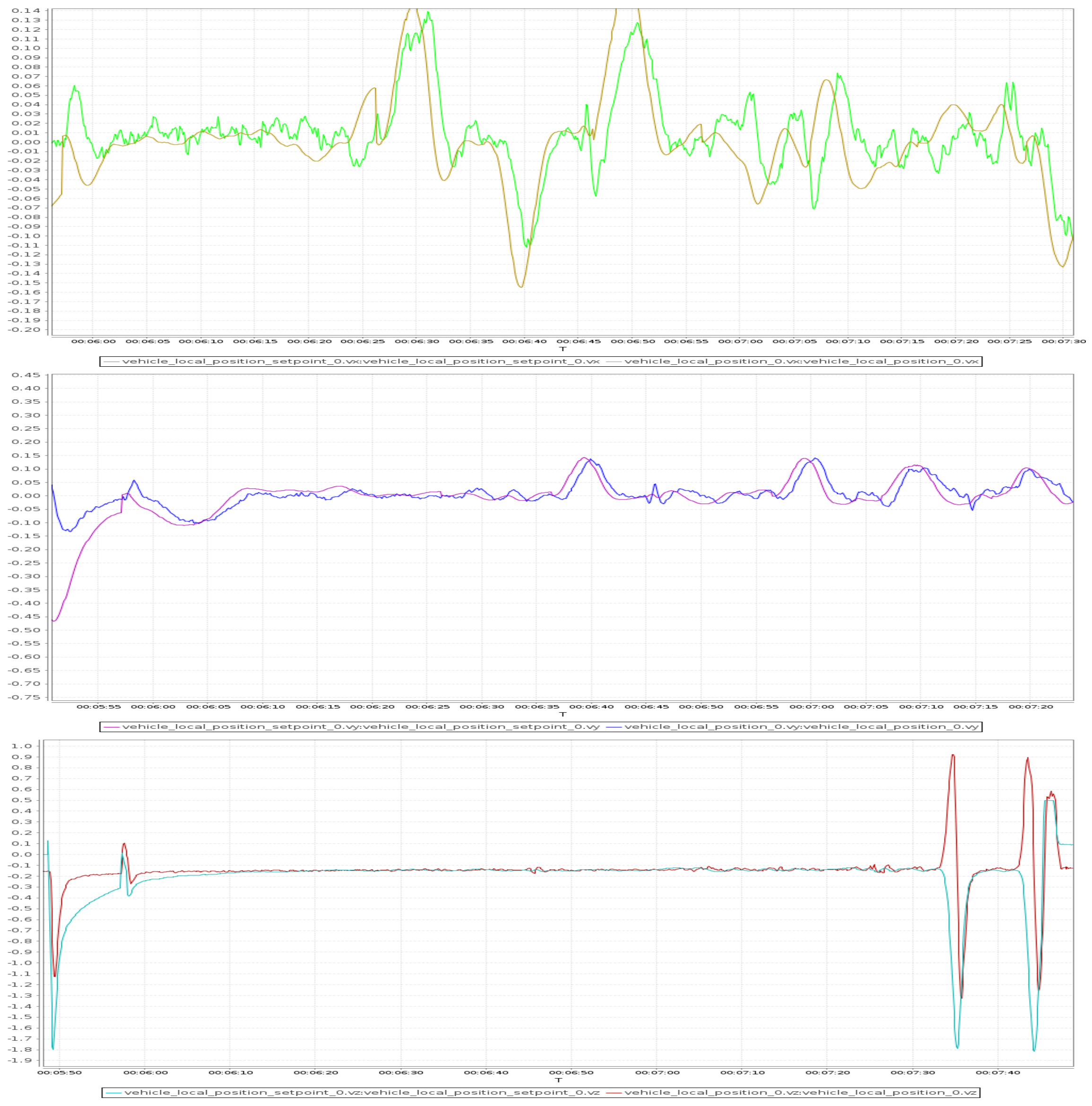


Figure 3.18: From top to bottom, Vx, Vy and Vz control

When the tunning step is completely done, you are free to start flying the quadcopter around using some example code that will be proposed on our github.

3.5 Simple cube trajectory

For testing the overall of our estimation and tuning, we made a simple python script that take-off the quadcopter, send it around the corner of a cube and then land the quadcopter. The result of flight is shown below:

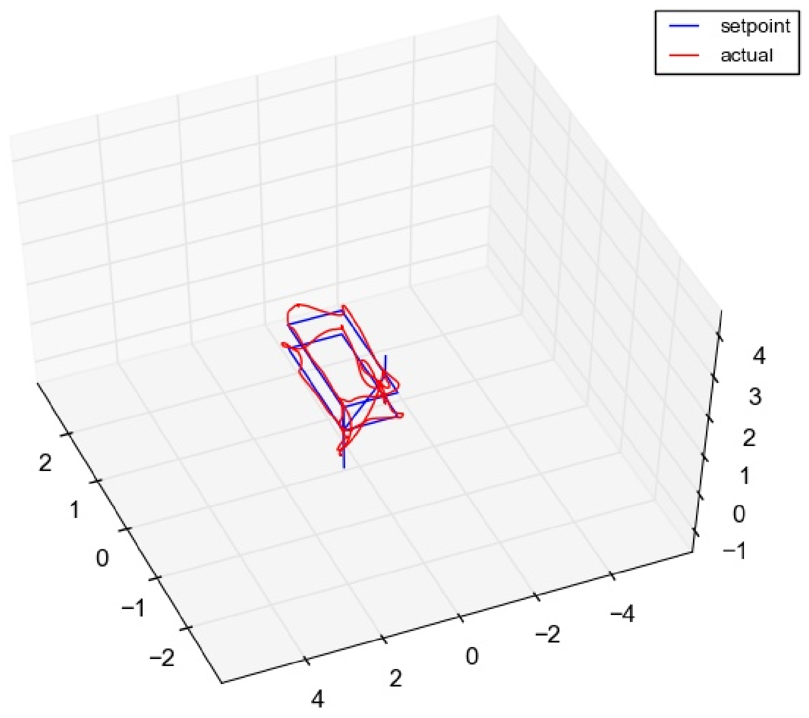


Figure 3.19: Cube trajectory by the quadcopter

As you can see on the figure above, the quadcopter was following well the "cube" edge. **Note that this output to the quad here are just position target**. The dirty path you can see on the plot is due to the take-off procedure. Indeed, because of the ground effect, taking off don't just send the quadcopter straight up : it also deviates a lot on the X and Y axis. In the chapter 5, a more reliable, efficient and precise controller for tracking trajectory will be presented.

3.6 Flying quadcopter with eye tracking

After tunning PX4 controller and making test with some simple trajectories, we decide to integrate the quad in another simulation that was about eye tracking in Virtual environment using a new release headset FOVE Eye Tracking VR. The base of the experiment was simple : **Fly the quadcopter with only eye tracking**. FOVE VR was interacting with Unity installed on a Window computer in the laboratory. The experiment was just to prove that this can actually be done. Thus we needed to complete the following steps:

- **Been able to publish image frames in a compress and efficient way to avoid lag**:

For this step, Qualcomm developer had release a ROS node that can be launched from inside Snapdragon and with the appropriate ros package, you can manage to publish compress image on a ROS topic. For this experiment we obviously decide to use the **4K Camera** of Snapdragon but we didn't use his full capability since sending 720p image over WIFI for real-time control is clearly not going to work due to huge lag and delays. So we decide to send VGA quality image with black and white color for having a fluid rendering on the Unity side for the VR.

- **Been able to send image frames from Snapdragon board to the Windows computer hosting Unity**: The main problem here is to communicate with the windows machine. To be more precise, it is to communicate using a publish/subscribe style with the C# script running into Unity and that communicate with the VR. After some research, we found the project named **ROS.NET** with the goal of implementing a 'roscore' for C# and Unity interface on windows. We manage after some time to install and use it with Unity. The Unity C# code was able to subscribe, publish and make a service call like all ROS node.
- **Been able to feed FOVE with image received via ROS - Detect eye movements and gaze direction - Find solutions for the depth problem (When looking somewhere it's hard to figure out how far to go)**: This is the main work of ***Cyril Mansour*** who greatly succeed in doing that with an almost non documented/existent FOVE SDK. More informations about this step can be found in his report. Basically for the depth problem, we went *on a simple approach where the quad should just go at a certain speed on the direction the user is looking*. The quadcopter was then able to be in two modes: *Control mode* where the quadcopter goes always in the direction where the user is looking at (using the current heading of the quad). Finally an *Observation mode* where the quadcopter stop moving and the user can observe the environment using head rotation(mainly a control of the Yaw angle).
- **Send request to the Snapdragon board**: This is done thanks to the simplify off-board node described earlier that handles low interaction with PX4 and gives the user high functionality to control the quadcopter.

A video of this achievement can be find on Cyril's channel on **youtube[14]** with a small description of the experiment.

Chapter 4

Minimum Snap Trajectory

In this chapter, we explore the challenge of generating trajectories for quadrotors. After some research about trajectory optimization and generation for drone, we found the **Minimum-snap article[15]** which has been proven to be very effective for generating high-quality and smooth trajectories for drone. Basically, in the article, they develop an algorithm that enables the generation of optimal trajectories through a series of keyframes or waypoints in the set of positions and orientations. The trajectories found are optimal in the sense that they minimize cost functionals that are derived from the square of the norm of the snap (the fourth derivative of the position).

Why did they minimized the snap and not some other quantity ? While reading the article, it appears that to design an efficient trajectory, it is important to care about minimizing control efforts (**force and torque**), and this is done indirectly by minimizing snap. Basically, without entering in details about the quadrotors dynamic:

- The quadrotor can create a net force along the direction of its propellers. This net force has to move the quad along the trajectory, thus the quad's orientation needs to be tangent to the curve. Furthermore, propellers create a net force, so calculating this acceleration allows us to set our cumulative motor speeds

- The quadrotor can create a torque around all 3 world axis. Thus the quad's torque needs to change the quad's orientation as it moves along the trajectory. Knowing the torque lets us calculate motor speeds as well (See the paper for details).

- To calculate a net force (F=ma), we can write acceleration as the second derivative of position. Notice we take the second derivative of position to get the quad's net acceleration.

- To calculate torque needed to change the orientation of the quad along the curve, we take the second derivative of the quad's orientation. This is defined by the acceleration direction, so we take the second derivative of acceleration, which is the fourth derivative of position. This is the snap.

Finally, the torque is directly related to the fourth derivative of position - the snap. Any discontinuities in snap would require infinite torque, so we want it to be smooth. Minimizing snap will give us the curve that requires the least reorientation of our quad along it, and will guarantee smoothness so we can calculate motor speeds. The only justification given in the article is : *"...human reaching trajectories appear to minimize the integral of the square of the norm of the jerk... In our system, since the inputs...appear as functions of the fourth derivatives of the positions, we generate trajectories that minimize the integral of the square of the norm of the snap... "*. This justification is weak and hadn't been proven by the authors but it is in accordance with the description we made earlier.

In the following sections, we are going to discuss the problem outline and the solution proposed by the article, then we will talk about the numerical instability of the solution proposed by the article and how to reformulate the problem to solve the numerical instability. We will then present the optimal time segment solution, our own model for handling corridors constraints and a comparison to the model in the paper. While describing solutions used, we will introduce elements which allowed us to get a fast minimum snap implementation in addition to the multi-threading improvement.

4.1 Problem Statement

Here we want to find an optimal trajectory for the flat output x, y, z and yaw that smoothly transitions through the keyframes at the given times while staying in safe corridors and respect some constraints on the different derivatives of the flat outputs. The choice of polynomial trajectories is natural for highly dynamic vehicles and robots since these trajectories can be obtained efficiently as the solution to a QP that minimizes a cost function of the path derivative. Thus it is convenient to write these trajectories as piecewise polynomial functions of order n over m time intervals (basically we have **m + 1** keyframes).

$$\sigma_T(t) = \begin{cases} \sum_0^n \sigma_{T1} t^i & t_0 \le t < t_1 \\ \sum_0^n \sigma_{T2} t^i & t_1 \le t < t_2 \\ \vdots \\ \sum_0^n \sigma_{Tm} t^i & t_{m-1} \le t < t_m \end{cases}$$

We are interested in finding trajectories that minimize the integral of the k_r**th** derivative (here the snap: $k_r = 4$) and the k_ϕ**th** derivative($k_\phi = 2$) of yaw angle squared:

$$\min \int_{t_0}^{t_m} \mu_r \| \frac{d_r^k r_T}{dt^{k_r}} \|^2 \quad + \quad \mu_\phi \frac{d^{k_\phi}\phi_T}{dt^{k_\phi}}^2 \quad dt$$

$$\text{s.t} \quad \begin{cases} r_T = [x_T, y_T, z_T]^T = [\sigma_{Tx}, \sigma_{Ty}, \sigma_{Tz}]^T \quad , \quad \phi_T = \sigma_{T_\phi} \\ r_T(t_i) = r_i = [x_i, y_i, z_i] \quad , \quad \phi_T(t_i) = \phi_i & i = 0, \ldots, m \\ \frac{d^p r_T}{dt^p}|_{t=t_i} <>= \text{vector or free}, & i = 0, m; \quad p = 1, \ldots, k_r \\ \frac{d^p \phi_T}{dt^p}|_{t=t_i} <>= \text{scalar or free}, & i = 0, m; \quad p = 1, \ldots, k_\phi \end{cases}$$

Note that the two last conditions is to enforce continuity of the first k_r derivatives of r_T and first k_ϕ derivatives of ϕ_T at $t_1, \ldots, t_{m-1}$. In the expression of the function to minimize (without thinking about constraints), we can observe that the variables $X_T, Y_T, Z_T and \phi_T$ are decoupled so we can split the problem into four QP problem. Actually, because of corridors constraints, there will be dependencies between X_T, Y_T, Z_T components so let say for now that the problem will be split in two QP problems. To generalize, we will try to solve the following problem:

$$\min \int_{t_0}^{t_m} (\frac{d^k r}{dt^k})^t \cdot (\frac{d^k r}{dt^k}) \quad dt$$

$$\text{s.t} \quad \begin{cases} r \in R_N[X]^p, \qquad p \in N \\ r(t_i) = r_i = \sigma_i \in R^p, & i = 0, \ldots, m \\ \frac{d^j r}{dt^j} =<> constrained|free, & j = 1, \ldots, k \end{cases}$$

where **p** is the number of variables (number of output for the minimization problem), **N** is the polynomial order for the trajectories and **k** is the derivate order. **It's important** Basically, solving the problem including just the yaw angle will have the following parameters $p = 1, \quad k = 2, \quad N = 7$ and solving the problem with just X and Y as keyframes will have the following parameters $p = 2$, $k = 4, N = 7$.

Let's consider the previous formulation of the problem. Without loss of generality, let's suppose we have only one segment (the $i \in [1, m]$ **th** segment from the input list of positions) and the parameter $p = 1$ (one Variable) the function to minimize will be:

$$F_i(r_{si}) = \int_{t_{i-1}}^{t_i} (\frac{d^k r_{si}}{dt^k})^t \cdot (\frac{d^k r_{si}}{dt^k}) \quad dt$$

Let:

$$T = \begin{bmatrix} t^0 & t^1 & t^2 & \ldots & t^{N-1} \end{bmatrix}, \quad c = \begin{bmatrix} c_0 \\ c_1 \\ c_2 \\ \vdots \\ c_{N-1} \end{bmatrix}$$

Considering:

$$\begin{aligned} r_{si} &= c_0 + c_1 t + c_2 t^2 + \cdots + c_{N-1} t^{N-1} \\ &= T \quad * \quad c \end{aligned}$$

We can obtain:

$$\begin{aligned} F_i(r_{si}) &= \int_{t_{i-1}}^{t_i} (\frac{d^k(T*c)}{dt^k})^t * (\frac{d^k(T*c)}{dt^k}) \quad dt \\ &= \int_{t_{i-1}}^{t_i} c^t * (\frac{d^k T}{dt^k})^t * (\frac{d^k T}{dt^k}) * c \quad dt \\ &= c^t * \quad (\int_{t_{i-1}}^{t_i} \frac{d^k T^t}{dt^k} * \frac{d^k T}{dt^k}) \quad * \quad c \\ &= c^t * H_i * c \end{aligned}$$

We can then identify the cost matrix for that case by simple calculation of the following expression:

$$H_i = (\int_{t_{i-1}}^{t_i} \frac{d^k T^t}{dt^k} * \frac{d^k T}{dt^k}) = \left(\begin{cases} 0 \quad if \quad l < k \quad or \quad j < k \\ \frac{l*(l-1)*\cdots*(l-k+1)*j*(j-1)*\cdots*(j-k+1)}{i+j-2k} (t_i^{l+j-2k+1} - t_{i-1}^{l+j-2k+1}) \quad else \end{cases} \right)_{1 \le l \le N, 1 \le j \le N}$$

Thus, solving the entire problem for p variable(s) is equivalent to solve the following QP problem:

$$\begin{aligned} min \quad c^t * H * c \\ s.t \quad A * c \quad <=> \quad b \end{aligned}$$

Where **c** is the column vector we want to obtain. It number of column is $p*m*N$ and an efficient storage for trajectory for each variable should be to first put the coefficients for all segment polynomial of the first variable. Then do that for the $p-1$ remaining variables. The *cost matrix* here is **obviously the same for every variables** so we just have to obtain one of them called H_val and the global H matrix should be a *Diagonal matrix of size p*m*N* where the p block of N*m * N*m matrix on the diagonal are H_val. Having obtained an expression for the cost matrix of one segment, it is obvious that H_val is also a diagonal matrix where the m matrix in the diagonal are H_i found earlier. Finally the constraint matrix will directly be obtained from constraints on the derivatives of the p variables. We didn't include their formula here since the process to get the matrix A is the same. And **b** is just the vector containing the different values for the constrained derivatives.

4.2 Solve the QP problem

Let's first **suppose that there is no inequality constraints in the problem** (no corridors constraints, no interval for derivatives values). The problem then become a simple QP problem that can be solve with the *lagranger multiplier*. The equivalent problem to solve using the lagrange multiplier is just the following system of linear equations:

$$\begin{bmatrix} 2*H & A^t \\ A & 0 \end{bmatrix} \begin{bmatrix} c \\ \lambda \end{bmatrix} = \begin{bmatrix} 0 \\ b \end{bmatrix}$$

The following properties are the essential properties to take in account when building the model:

- **The cost matrix and the constraint matrix are sparse :** This result is obvious looking at the formula for H matrix. We can even say that for the cost matrix ($p*m*N*p*m*N$ elements) we will have a maximum of $p*m*(N-k)^2$ non-zeros coefficients. The constraint matrix sparsity is less trivial but can be sense since constraints are applied segment by segment. This means that each row of the A matrix ($p*N*m$ elements) will contain at maximum either N non-zeros elements (non free constraints) nor $2*N$ non-zeros elements(free constraints and continuity). **Thus we mainly based on Sparse Matrix/vector structure to solve the QP problem**.

- **The cost matrix is Symmetric and Semi-definite Positive :** The fact that the cost matrix is symmetric is quite obvious when looking again at the expression for $H_{i,j}$ coefficients. Thus we don't need to store all the sparse matrix but only the lower/greater triangular view. It is easy to prove that the matrix is not definite positive but only semi-definite positive: Let's just go down to a sub-matrix cost corresponding to the small problem on a segment. We know by the formula given for this matrix that the first $p-1$ line and rows will be zero. It is then easy to find a combination of non-zeros coefficients for the polynomial where the evaluation of the function to minimize will be zeros. **The fact that H is only semi-definite positive restrains our choice of the solver to use for the linear system**. Basically here **only a QR decomposition is the fastest and reliable way to solve the system**. Thus we choose to use existent **Multifrontal multithreaded rank-revealing sparse QR factorization**[16][17] to solve our problem. Why not using multi-threading when we can ? So we did it. Finally, a small trick is to notice that the matrix to apply the QR decomposition is also sparse and symmetric so by just considering the triangular lower view, we are able to accelerate calculation that will be made by the SPQR solver.

When **there is inequality constraints**, instead of using an external library that will solve the entire problem (which can take a lot of time for open source library), we decide to **use as a base our fast solver for only equality constraints and transform the inequality constrained problem to an equality problem by adding a barrier function**. Since the inequality constraints are intervals, it is easy to take an initial feasible set to start the problem. Thus we decide to choose the logarithm barrier function for our problem and using a gradient descent algorithm (maybe not the optimal scheme in this), we were able to solve our problem efficiently.

4.3 Solving Numerical instability of Mellinger formulation

When implementing the previous solution and testing it, you may quickly notice that this formulation of the problem becomes ill-conditioned for more than several segments, polynomial of high orders, and when widely varying segment times are involved. The fact that it becomes ill-conditioned just break down the solution for the lagrange linear system. Hence this formulation is only useful for short trajectories and must be improved to be practical for optimizing long range paths requiring many waypoints, segments and varying segment times.

The problem is the way we construct the cost matrix in the initial formulation. The idea here (article from 2016) [18] is that instead of solving the problem for polynomial coefficient like it is done in the original formulation, **we should solve directly for endpoint derivatives as decision variables**. So far, we didn't encounter numerical issues while using this new formulation (and we went really far on the number of polynomial segments in single matrix). *Actually we encounter a problem that causes numerical issues and that is handle in a non-efficient way in the article. We will present the problem and the solution later.*

In practice, we will have the new following QP problem to solve:

$$\begin{aligned} c_{new} &= M * c \\ min \quad c_{new}^t * M^{-1} &* H * M * c_{new} \\ A * M^{-1} * c_{new} &<=> b \end{aligned}$$

With M defined as following:

$$M = \begin{bmatrix} M_{sub} & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & M_{sub} \end{bmatrix}_{p*m*N} \quad and \quad M_{sub} = \begin{bmatrix} M_1 & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & M_m \end{bmatrix}_{m*N}$$

$$M_i = \left(\begin{cases} \dfrac{d^{j-1}\begin{bmatrix} 1 & t_i & t_i^2 & \ldots & t_i^{N-1} \end{bmatrix}}{dt^j} & if \quad 1 \leq j \leq (k = \frac{N}{2} - 1) \\ \dfrac{d^{(j-1)/2}\begin{bmatrix} 1 & t_{i+1} & t_{i+1}^2 & \ldots & t_{i+1}^{N-1} \end{bmatrix}}{dt^j} & if \quad j \leq (k = \frac{N}{2}) \quad if \quad \frac{N}{2} \leq j \leq 2(k+1) = N \end{cases} \right)_{1 \leq j \leq N}$$

One can immediately notice that this formulation need to consider $N = 2(k+1)$ in order to be able to inverse the all the matrix M_i. As we said, c_{new} is just a vector containing the derivatives of the polynomial trajectories on each segment for the initial point and end point so basically for one axis it will be a vector of pos_{init}, vel_{init}, acc_{init}, $jerk_{init}$, $snap_{init}$, pos_{end}, vel_{end}, acc_{end}, $jerk_{end}$, $snap_{end}$. So the expression of M_i matrix is as simple as proposed previously.

Finally, a last improvement for numerical stability is to consider **the duration of a segment instead of considering the initial time and end time of the segment**. As you may have notified, The first expression obtained for the sub-matrix H_i was depending of the calculation of

$t_i^{l+j-2k+1} - t_{i-1}^{l+j-2k+1}$ which coded this way can bring a lot of numerical errors when solving the problem. The solution is that by considering the duration instead, the integration will always be between 0 and T_i. With this condition, the continuity for all derivatives must be modified. With this formulation (considering that every segment start time is zero and end time is the duration of the segment), we manage to *increase the sparsity of the A matrix, reduce the numerical errors, speed up the inversion, creation of the C matrix in addition to his operation (multiplication) with others matrix*. Basically the matrix on each segment will be similar to this matrix:

$$M_i = \begin{bmatrix} A & 0 \\ C & D \end{bmatrix} \quad => \quad M_i^{-1} = \begin{bmatrix} A^{-1} & 0 \\ -D^{-1}CA^{-1} & D^{-1} \end{bmatrix}$$

Since A is a diagonal matrix, to inverse M_i it is just necessary to inverse the reduced matrix D. Performance results will be shown at the end of the chapter.

4.4 Corridors constraints integration

Corridors constraints are simply for each segment the maximum of deviation the generated trajectory should have from the straight line joining the initial point and end point of the segment. In some cases, the trajectory generated by minimum snap can deviate too much from the straight line and you may not want that margin because for example the quadrotors will possibly enter in collision with an obstacle or in our initial problem you may want to ensure that moving from one state to another one, the generated trajectory do not intersect the state bounds. In Mellinger paper, this problem is solve by adding inequality constraints in the QP problem (details can be found in the article). Well, with this solution, you are ensure to generate the optimal path with these added specifications but the time to solve the problem explode. Moreover, in the article, they set the number of intermediate point for the constraint to a fixed value. **In a practical point of view, This can not ensure that all the corridors constraint will be placed**. One solution is to start with a number of intermediate fixed points and iterate, check if "bounds" have been respected and decide to stop or try with a higher value of intermediate point if not. This algorithm will converge but will take too much time if many points must be set to respect constraints.

We formulate this problem with a more geometry approach. the solution given here is not the optimal in term of the cost of the final solution found but it is fast to implement, fast at execution time and is close enough from the ideal solution. The global idea is to not include inequality constraints in the QP problem but instead to quickly detect on each segment if the generated trajectory has points out of the corridors. The figure following will be used to explain the procedure:

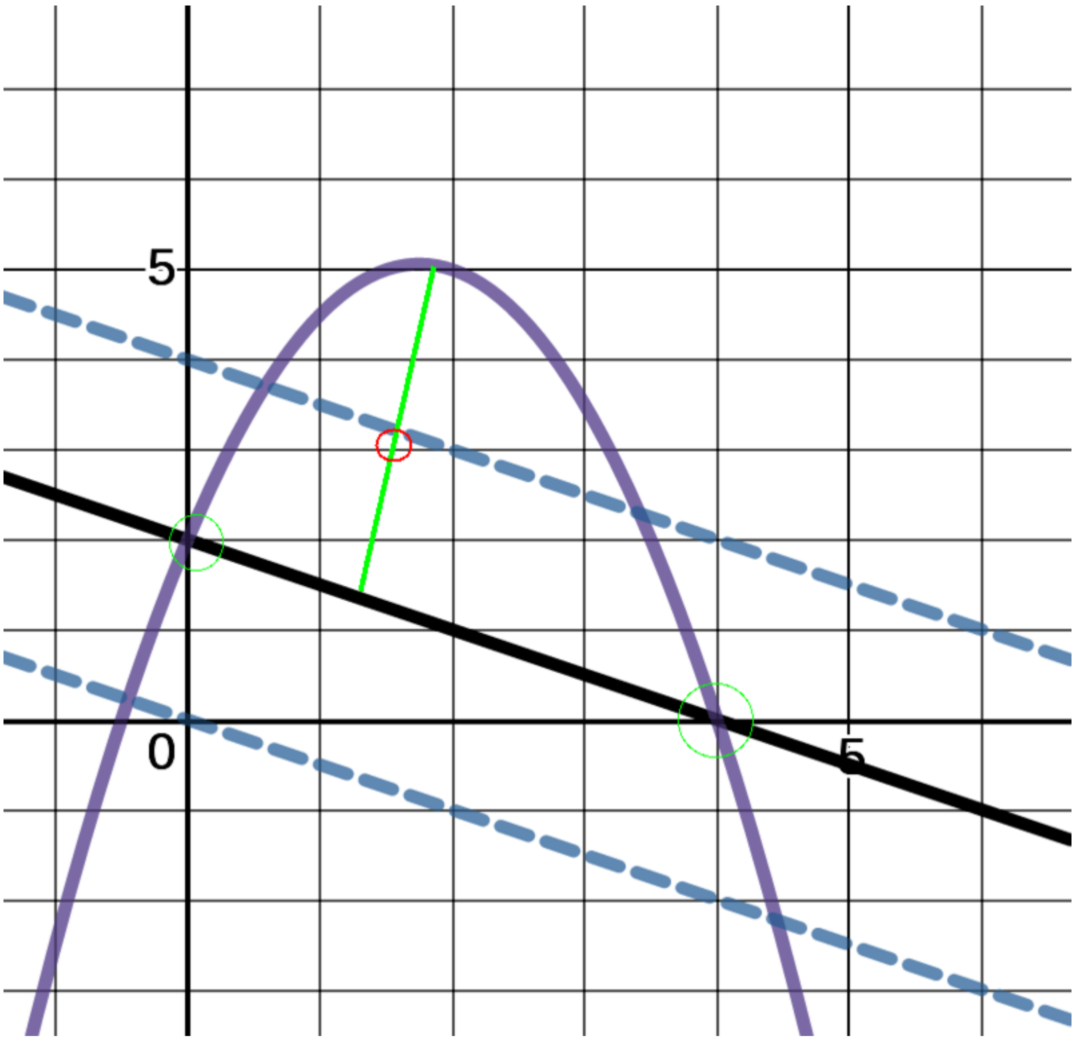


Figure 4.1: Simple illustration of corridors constraints formulation

Basically on the figure, you can observe one black line which is a straight line between the init point and the end point (green circles) of the segment. The dashed line represents the corridors constraints applied to the current segment. Thus what must be done is to determine the point of the curve that maximizes the distance to the segment between the initial(A) and end point(B). When we have this point, and the time parameter corresponding to that point, the simplest solution is to project that point around the bounds constraints, add the point in the keyframes and start again the solution of the QP.

Now considering the following function to return the distance between the point at the instant t of the solution and the straight line between A and B:

$$d(t)^2 = \|p(t) - A - \frac{(p(t)-A)\cdot\overrightarrow{AB}}{\|\overrightarrow{AB}\|}\overrightarrow{AB}\|^2$$

By writing the norm as a product scalar and then derivate the formula above, one can easily obtained the expression of the derivative of **d** and will find out that **finding the zeros of the**

derivative is finding the zeros of a polynomial function. this is important since if we had to use some optimization algorithm to find Thus it will have almost no cost in term of computational time to determine the zeros of the polynomial function and then have all the extrema of the distance function. Having all the extrema, we can have the points that are out of the bounds and then project them back around the bounds, while adding them to the keyframes of the new QR problem to solve. The algorithm for this operation is the following:

Algorithm 1 Compute minimum snap with corridors constraints

```
while True do
    $sol \leftarrow$ compute_min_snap(inputs)
    for $i = 0; i \leq segmentLength; i++$ do
        extrema_times $\leftarrow$ polynomial_root($d'(t)$)
        for time in extrema_times do
            if $d(t) \geq currentBound$ then
                current_point $\leftarrow$ point_from_time(time)
                add_point_and_time_to(inputs,current_point,time)
            else
                pass
            end if
        end for
    end for
    if $not\ \ inputs\ \ list\ \ changed$ then
        break
    end if
end while
```

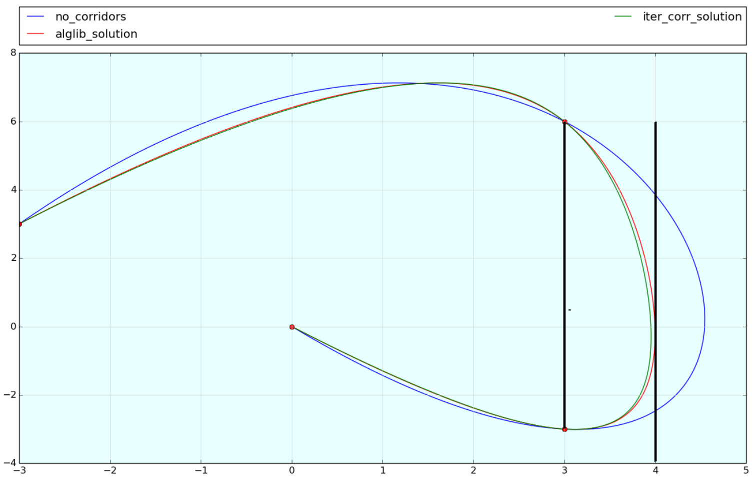


Figure 4.2: Iterative Corridors iteration and comparison with result when solving the problem with inequality

On the figure above, the blue trajectory represents the generated trajectory when there is no constraints, the red trajectory is obtained by injecting the QR problem with inequality in the alglib library. And the green trajectory is obtained using the algorithm presented above. We can observe that the iterative and alglib solution respected the corridors constraints and the trajectory returned are pretty much similar. However on the computational time, there is no comparison to do since our base implementation for solving the problem with only equality is really fast.

4.5 Optimal time segment and Implementation performance

Sometimes, instead of sending duration of each segment with the list of waypoints, one may want to just send the total duration of the flight and let's minimum snap decide what is the optimal trajectory using that total duration. In the article, they describe well how to find the optimal relative segment times for a given set of keyframes (see the article[15] for more details). Here we are going to discuss about a problem we didn't mention earlier which the *singularity of the matrix M*. In fact, theorically speaking, the matrix M is always non singular. However for segment duration which are lower than let say 1, The sub-matrix M become singular because of computer precision and user can get numerical errors. The fix we made for this problem was (for each trajectory request) to scale time depending of the entry target duration for each segment. So if the duration of one of the segment is under a certain time threshold, we scale all the duration inputs in order to be able to inverse M matrix. Scaling the time doesn't change the problem or the solution (because you can just scale it back) but it actually change the returned cost when minimizing the problem. For generated trajectory where the user gives duration for every segments, it is not important to return the cost found when solving the original QP problem. However since finding optimal time segment is basically a gradient descent algorithm, so at some point, some comparison have to be made between found cost values. The problem here is that the values to be compared were obtained from different scaling factors - because of distinct set of duration when doing steepest descent - Thus we need to find a way to compare these values knowing the scaling factor. To solve that problem, we demonstrate that **If k is the derivative order, and α the scaling factor such that $T_{old} = \alpha T_{new}$ then:**

$$costFunction(T_{old}) = \alpha^{2*k-1} * costFunction(T_{new})$$

Using this trick, we managed to make optimal time segment without numerical errors. The following figures shows some advantages of computing optimal time segment if you have a total duration instead of setting duration for segments manually: The trajectory looks more natural and the the acceleration/velocity are well chosen by the algorithm.

In the figure 4.3, we can actually see that better cost function return trajectory that seems natural to human point of view. I believe it's preferable to use optimal time segment as usual as possible because behind it formulation the solution return guarantee you that the quadrotors won't make brutal acceleration and then slow down etc...

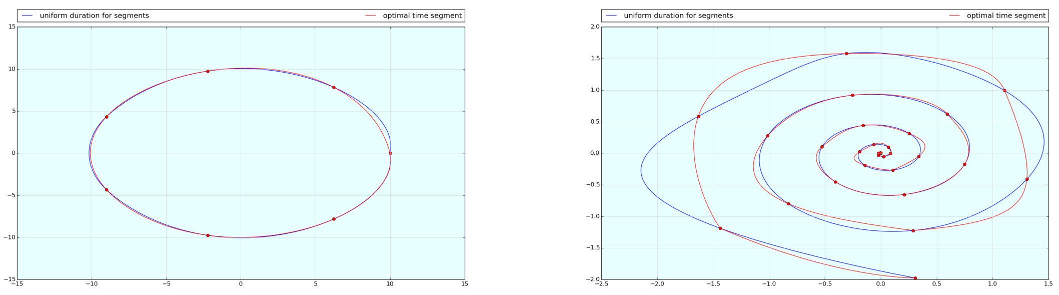


Figure 4.3: Optimal time segment vs uniform duration repartition

Finally, the next figures are about the performance of our implementation on a simple case of a conical helix trajectory with a *i7, 8 cores, 2.6Ghz* computer. Basically, We are trying to generate the conical helix trajectory with a set of keyframes where the size vary from 100 to 20000. So on the figure below, the X axis represent the number of keyframes as input and the Y axis is the time in millisecond before getting the trajectory from our implementation.

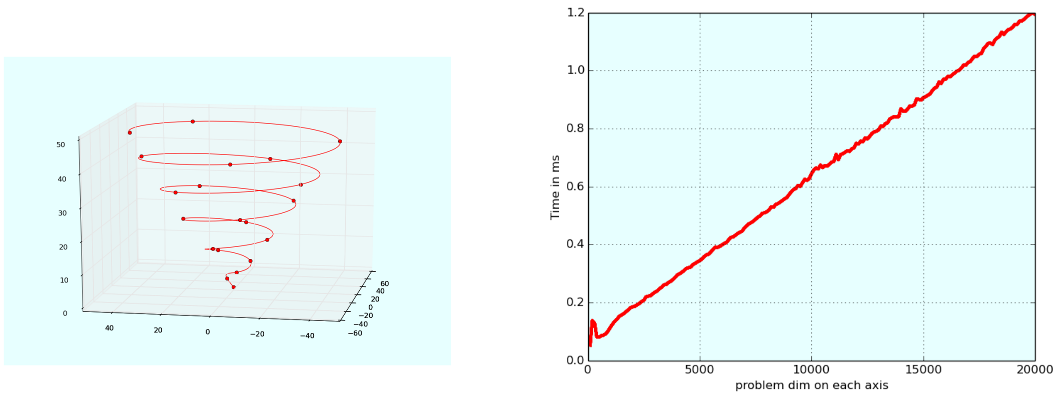


Figure 4.4: Optimal time segment vs uniform duration repartition

To summarize the previous performance figure, for a list of 20 000 keyframes, we are able to build the entire QP problem, then solve the equivalent linear system where the matrix to factorize is a $200000 * 200000$ **and return the solution in 1.2ms .** The current version of the implementation is on the u-t-autonomous group github[11].

Chapter 5

PVA Controller

This chapter is a brief chapter about the implementation of a PVA controller embedded in our off-board node. This implementation is based on this article[19] but uses a simplest formulation than the one proposed by the article. Basically, what we are implementing here is just a 3DoF (degree of freedom) PID controller that takes in input Position and Velocity and Acceleration and output the attitude value that will be controlled by PX4. This is the ideal choice since as we saw here 3 the attitude control done by PX4 was very good. What our off-board node is doing is very basic:

- When gathering PVA data from another node (Typically a node that called minimum snap implementation), the main thread should start one thread for Attitude control and another one for executing the PID loop (this is the PVA thread). The PVA thread is a loop at 30Hz realizing the PID algorithm and exchanging the output with the Attitude control thread.
- It ensures that the odometry PX4 publish at a high rate to be able to do a proper control. Solving this problem was done by changing the default rate at which PX4 publish odometry data and we choose to publish odometry at 100Hz.This will be time consuming and processor resource consuming but it is mandatory for achieving what we achive here since it is the only one that can give me also estimation of the speed.
- It offers the possibility to the user to also tune the pid parameter thanks to a call of a service inside the off-board node. This call to a service can be done on a matlab, python, C++ , all language that supports ROS.
- We have predefined tunning for gazebo simulation and AirSim simulation that are already used.

The next tests I am going to present have been done using an *uniform helix trajectory* as trajectory that must follow the quadrotor. The time for the complete trajectory was fixed to 30s and the quadcopter terminates the trajectory in approximatively 31s which is really close from the value initially set. Let's compare now the performance of the control PVA controller.

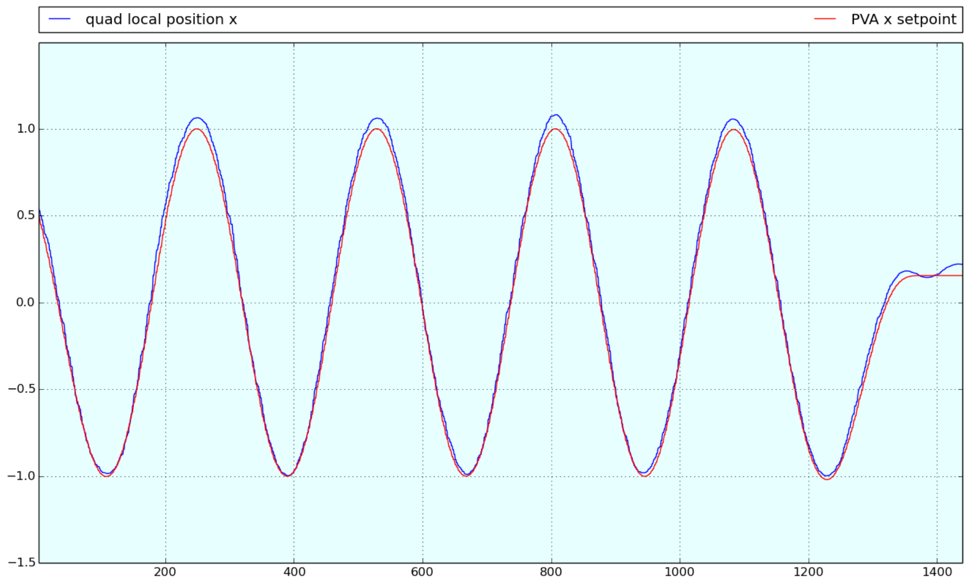


Figure 5.1: PVA Control on the X axis

The figure 5.1 is showing some pretty good trajectory tracking skill by the quadcopter. We can see that when our Controller get some delay from the setpoint, it shortcut his trajectory to respect the time specification. It is obvious that this controller , at least on the X axis has better performance compare to the position controller integrated in PX4. Note that we also didn't spend time on tunning this controller with the appropriate tool so greater things can still be done with the quadcopter.

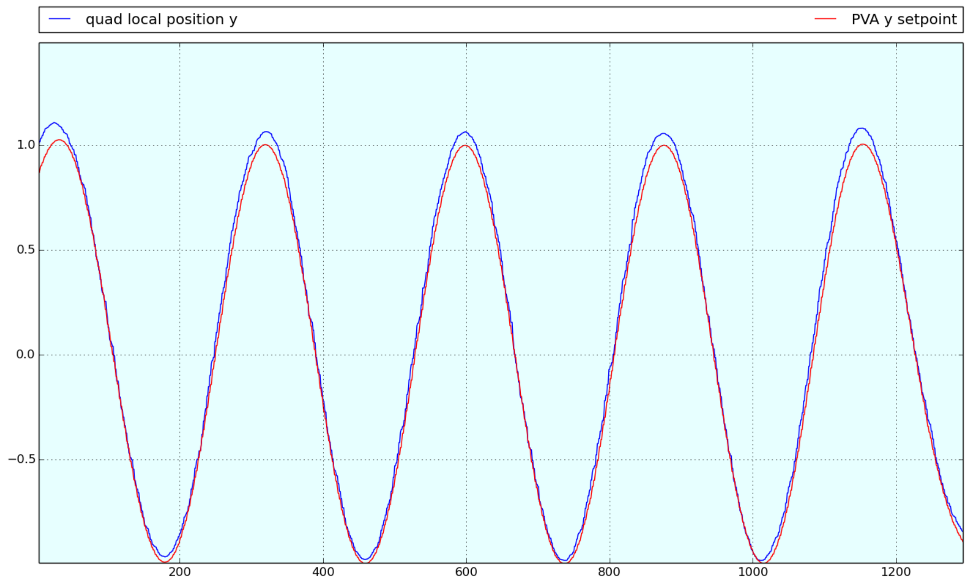


Figure 5.2: PVA Control on the Y axis

The figure 5.2 is showing the same performance as X for the control of Y. We are expecting almost the same result since the PID coefficient for the two variable are identical. Note that our actual tunning doesn't include the integrator term. The figure 5.3 also show us that the controller

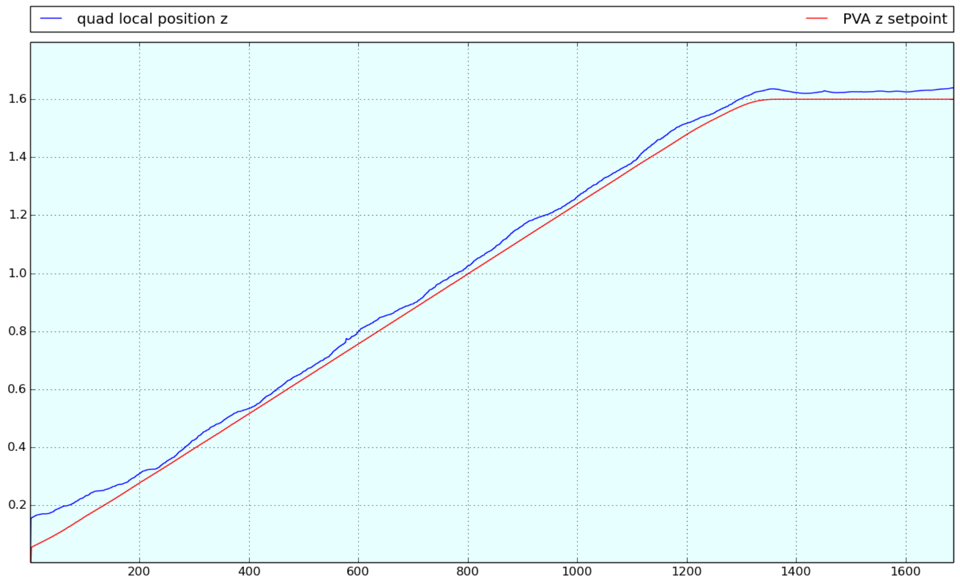


Figure 5.3: PVA Control on the Z axis

have almost a good control over z variable. the problem here is that offset between the setpoint and the actual value. I believe adding an integrator term should fix that static error.

Finally the plot in 3D 5.4 is good except from the impression of huge deviation from the setpoint trajectory due to the static error on the Z axis control. This controller will be the one used when doing a transition from one state to another state in our tulip model because we want to ensure some duration of the transition and also some good tracking of the trajectory to avoid collision.

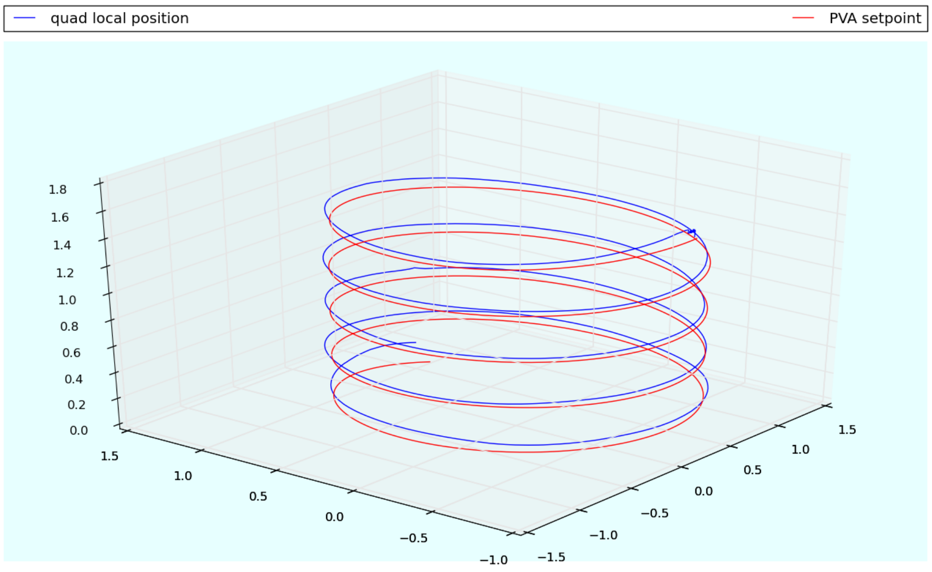


Figure 5.4: PVA Control on the Z axis

CHAPTER 6

PLANIFICATION OF QUADROTORS AUTONOMOUS MISSION USING TEMPORAL LOGIC PLANNING

In this final chapter, we are going to talk about planning quadrotors mission in a non completely unknown environment using correct by construction controller. By non completely unknown environment, we assume that we already know the position of static objects in the map and we also assume that we have a global observation of the world. In our case, this means that we will have at every time our location and the location of others vehicles in the environment. The global observation hypothesis is not mandatory as we will explain in the conclusion. The goal here will then be to synthesize controllers for the quadrotors which are provably correct in respect of some specifications on the quadrotors mission. Typically these specifications will be written in *LTL* then feed to a solver (*gr1c inside TuLiP*) and translate the output in a way the quadrotors could understand.

To summarize, we are interested in the following problem: we are given in the arena a system of severals quadrotors. We have full control on some of the given quadrotors and the uncontrolled quadrotors will be what we will call *environments*. Given the model of the quadrotors and the task as specification expressed in Linear Temporal Logic (LTL)[20], we synthesize the reactive control protocol such that the controlled quadrotors behavior satisfies the specification for all admissible environment behaviors. Since the environments are uncontrollable we can just make assumptions on their behaviors. This will also be done using LTL and integrate into the model. To solve the problem written using LTL we will use the toolbox TuLiP[21] to synthesize the discrete reactive controller with correctness guarantee of satisfying the specifications. If the problem is feasible then we ca get a reactive controller as a Finite Automaton where the transitions represent on winning strategy for the quadrotors we are able to control. After getting the finite automaton, there is a need to translate it in order to do the path planning on the quadrotors side. The path planning step will then require our PVA controller an dthe minimum snap trajectory. More information about LTL can be found on wikipedia[20]

6.1 Fast overview of TuLiP toolbox

tuLiP[22] is a collection of python based code for automatic synthesis of correct-by-construction embedded control software. TuLiP is designed to synthesize discrete-state controllers for hybrid systems operating in a (potentially dynamic and unknown) environment. The system specification is given in terms of a temporal logic formula. Before starting using TuLiP, you may need to define the reachibility game you want to solve. Considering the application on the quadrotors, the games we are solving are all multi-robot path planning problems. The definition of the game contains two parts: the model of the whole system, which can be described as an automaton, and the task we want to implement, which is expressed in the language of LTL formulae. The model is divided into two parts: in the first part, you will have the 'environment' which are supposed to be uncontrollable; the second part, the 'system' will have fully controllable robots. We need to define the environment variables and system variables, their ranges of value, and the allowable transitions restricted by the model.

The next step is to write the specification i the language of LTL formula. According to the structure of GR[1] specifications, we divide all assumptions on the environment part and all guarantees on the system part into 3 different kinds: initial conditions, safety conditions and liveness conditions. Setting initial con- ditions will limit the choice of the initial states, but have no effect on the feasibility of the problem. Safety conditions can be used to avoid undesirable 'dangerous' states, typically collisions. The transitions allowed by the model should also be written as safety conditions. Liveness conditions can be used to deal with fairness requirements, usually used to prevent a robot from staying still for a long time. So this last one helps to create movement. It is important to not that for GR[1] formulae, the usage of temporal operators is largely restricted: only a single 'next' operator can be used in the formulae ϕ_{safe} , $\phi_{liveness}$, ψ_{safe} and $\psi_{liveness}$; logics containing eventually ($\Diamond$) and until (U) will have to be transformed to other forms. To do this, sometimes we need to introduce some auxiliary variables.

After setting the state variables and the specifications, TuLiP can help us synthesize a controller. If such controller exists, TuLiP will return us with a Finite Automaton, which represents one winning strategy for the system; otherwise TuLiP will return a 'NoneType' variable. Then we can save the automaton transitions in a text file or with a Python pickle module for later use. When doing simulations, a 'manager' should read the transitions, specify an initial state allowed by the initial conditions, pick a set of valid assignment for all the environment variables, get the next state from the transitions and the current state, and then since everything is discrete, move the environments and the systems. Doing this is equivalent of having the same clock for the system and the environment. In the following section we will use this procedure to implement some autonomous mission and we will try to propose algorithms that could allow us to avoid having to launch the system and the environment at the same time when a decision is made. We want the environments to move independently and make the system react to them in a more continuous way.

6.2 Experiments with environments of lower frequency change of state

In this section, we will present a class of problem which are quadrotors mission in an environment where it has positions of all obstacle, full observation of the environments and moreover the systems knows that a change of state of the environment may appear at low rate. You can imagine a lot of domain where the robot has his own task to achieve but sometimes need to stop his task, execute a new human task and start his task where he stops it. Here we will discuss about the simple case where we have only one quadrotor we fully control (the system) and one environment variable. The quadrotors has pre defined waypoints he must go infinitely often and if a user want the quadrotor to go at a certain place in the map he must stop his 'surveillance' and go for example to the position specified by the user while avoiding fixed obstacles in the map and eventually but controlled other quadrotors.

The difficulty here is to write the specifications in LTL. the quadrotors must go infinitely to pre defined waypoints is a basic specification that can be written easily depending on if an order of reaching the waypoints is defined or not. In case no, $[]\Diamond(loc = pos_i)$ for all associate state of position in the set of waypoints, where loc is the variable representing the state of the quadrotor. In case yes, the trick is to introduce in the system variable a second variable named stage that will be incremented only if the quadrotor arrives at the waypoint i while been on the stage i. Thus adding the last stage and the initial stage in the sys_prog will create a controller which will loop over the list of waypoints.

Considering now the fact that the user can change the path of the system, the problem is the fact that the system may not be able to achieve his mission if the user constantly ask the controller to go somewhere. In a way,in LTL we need to ensure the controller that he can still reach his sys_safe specifications. This is done by manually increase the stage variable on a user request in order to make the system understand that he will always eventually go on the last stage and init stage. Moreover, we need to add in the system variable a boolean for checking if the request has been complete or not in order to get back to his work. This should be sufficient to implement the above specifications. On the github repository, we have a lot of example of scenario in TuLiP. For this problem we synthesized a general controller, basically a python class where the user give to the constructor dimension of the grid, number of controlled quad,a list of state where eventually the user will ask each quad to go, specification on the geometry of the quadrotor (useful when the cell size is smaller that the quad size). And then call the createController method of the class to get an instance of the TuLiP controller that you can use with your path planning.

We simulate this experiment using Unreal Engine with Airsim since we wanted a real environment for all our multi-vehicles simulation. After integrating Airsim plugin with UnrealEngine, you can load a map. We tried to find a small map since it can take time to generate the controller for big size map. Having the map, the next step is normally to launch PX4, and start and offboard communication with the quadrotors. The problem after this step is that you need a 'map' of the envionment, a sort of 3D occupancy grid to be able to navigate here and have the position of all the obstacles. We then used the well known probability 3D mapping[23]. To be able to use it, one must feed the images received by the camera in the virtual environment into the octomap server node. This allows the reconstruction of the arena as shown below :

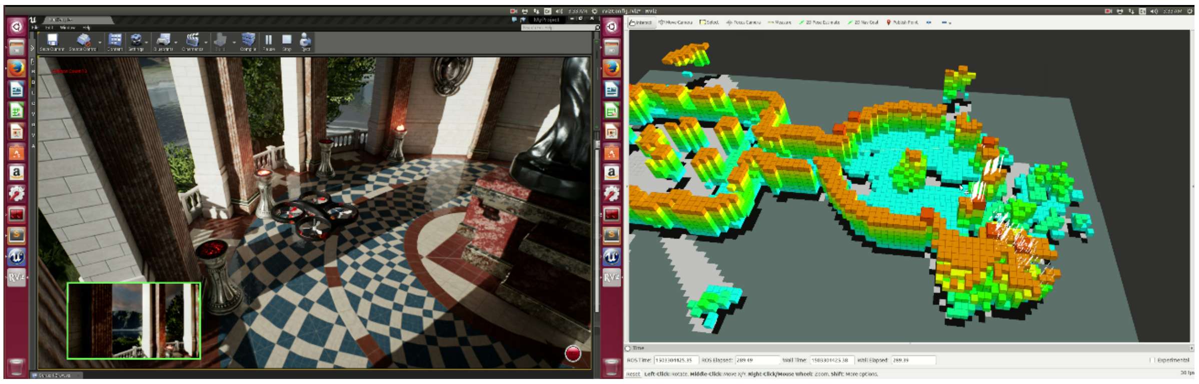


Figure 6.1: Reconstruction step

We made a node for easily sending images from Unreal engine to octomap and code for receiving the corresponding 3D occupancy grid. Then the occupancy grid is used to create the TuLiP model knowing the state of the position. We can then obtained a controller with the class above. Now since we made the assumption that the environment change of state will be at a low frequency, the strategy here towards a continuous behavior, is to precompute the trajectory using optimal time segment of minimum snap over a certain amount of time. In the meantime listen to state change event from the user. While there is no state change, start the previous precomputed path and launch in parallel a new path calculation. if there is a state change, add the new trajectory that take into consideration the new target and add it as the next trajectory to follow. We obtained pretty much good results in a realistic environment of simulation with this experiment. A video was recorded and a link will be link inside this report. The figure below shows the result obtained:

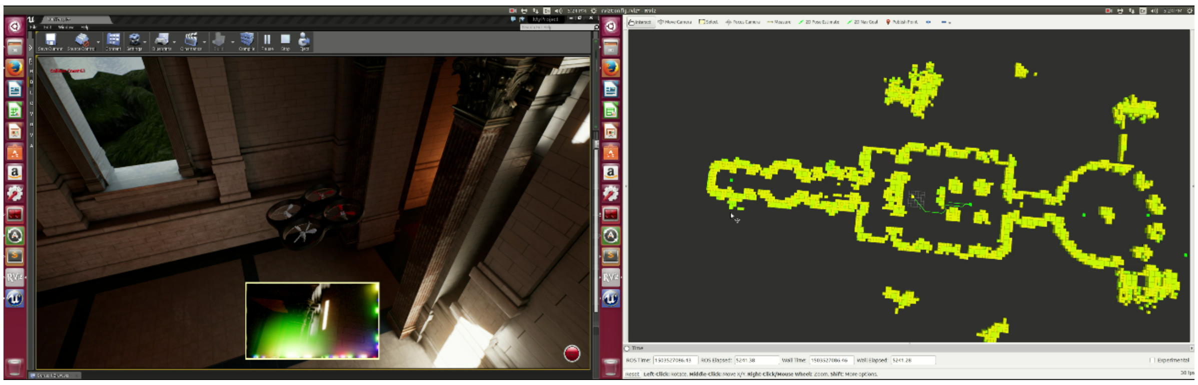


Figure 6.2: Quad path beginning experiment

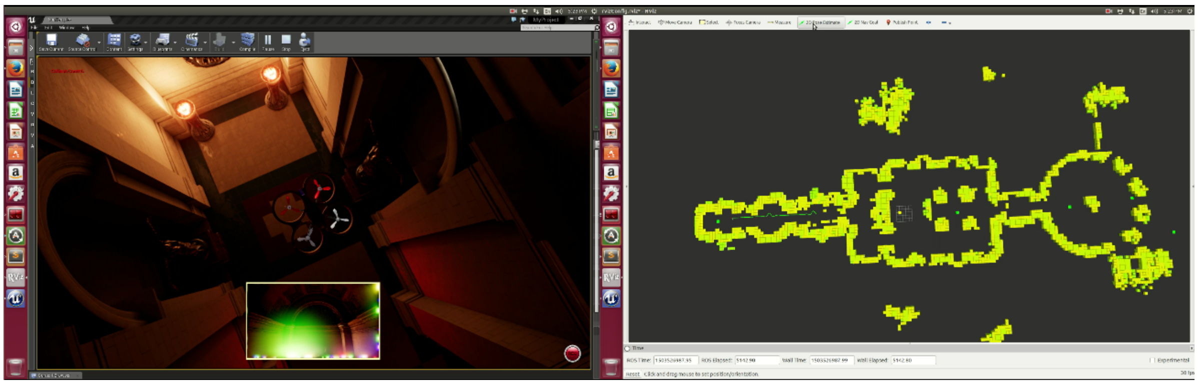


Figure 6.3: Quad path beginning experiment

The previous figures are just proof of the success of the experiment. The states returned from the synthesized controller are transformed back to positions and feed to minimum snap which return an optimal trajectory that respect the constraints of not getting out of a cell area. The PVA obtained from minimum snap are then feed to a PVA control who will ensure the control of the quad and principally ensure that the transition from one state to another state is execute in a given time without a huge offset that we can have with controlling only Position or velocity. What need to be done now is to flight it on a real quadcopter with all it complexity. Note that the grid size here is $192 * 160$ and we obtained the controller over 6-7 min. So this type of problem is something easily solvable by TuLiP controller.

6.3 Experiments with uncontrollable environments

At the beginning, we started our experiments with a basic example : follow me. In this example we have one environment and one system. Our assumptions on the environment was that it can only move on the 8 adjacent cells. We used the same assumption for the system except that we added another assumption about how fast the system can move on the border. This is because generally, the presence of the borders is a source of failure to synthesized a controller in TuLiP with an environment who is able to go everywhere. The reason is simple, if the goal of the environment is to collide the system he just have to try to restrain the movements of the system on a small corner and that's done. For avoiding this in order to have a controller, I added the possibility to the system to move from 2 cells on the border and only when he is on the border.

The follow me experiment as the name clearly explain is about a system which goal is to **always** follow an environment at a certain distance, here they must be always one free cell between the system and the environment. Since we are in a discrete state space, the distance will be count in number of neighbor cells. This specification is easy to implement in LTL. We test it both in simulation in gazebo and with the real drone. The first experiment implies a 10*10 grid where the quad was smaller than 1 cell size. We added in the grid some fixed obstacles and some target points where the control quadrotor must reach if possible while trying to follow the environment quadrotor. Since it exists solutions, we are guarantee that our specifications are respected. The experiment was recorded an the following figure shows the result on a 10 * 10 grid:

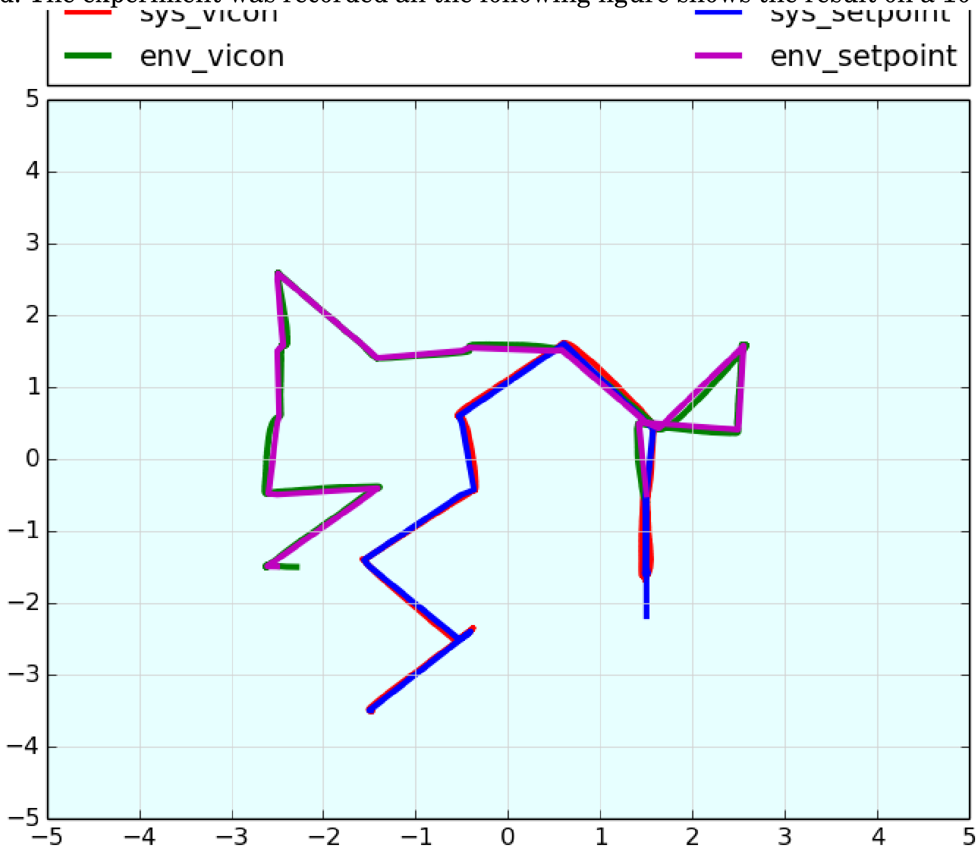


Figure 6.4: Follow environment in a 10 * 10 grid size in Gazebo

The blue trajectory is the system trajectory and the green trajectory is the environment trajectory. we can observe that the specifications here are at least respected at the beginning.

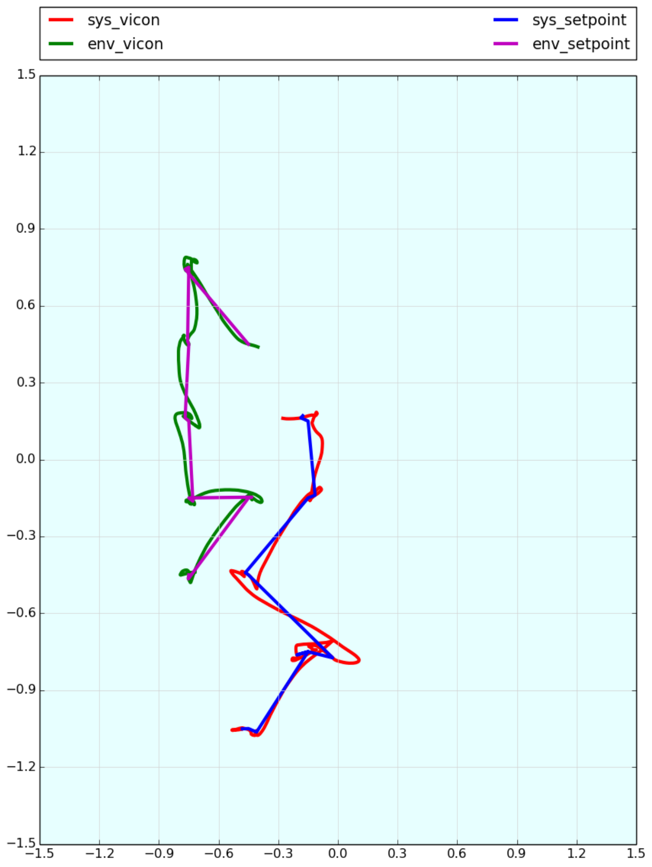
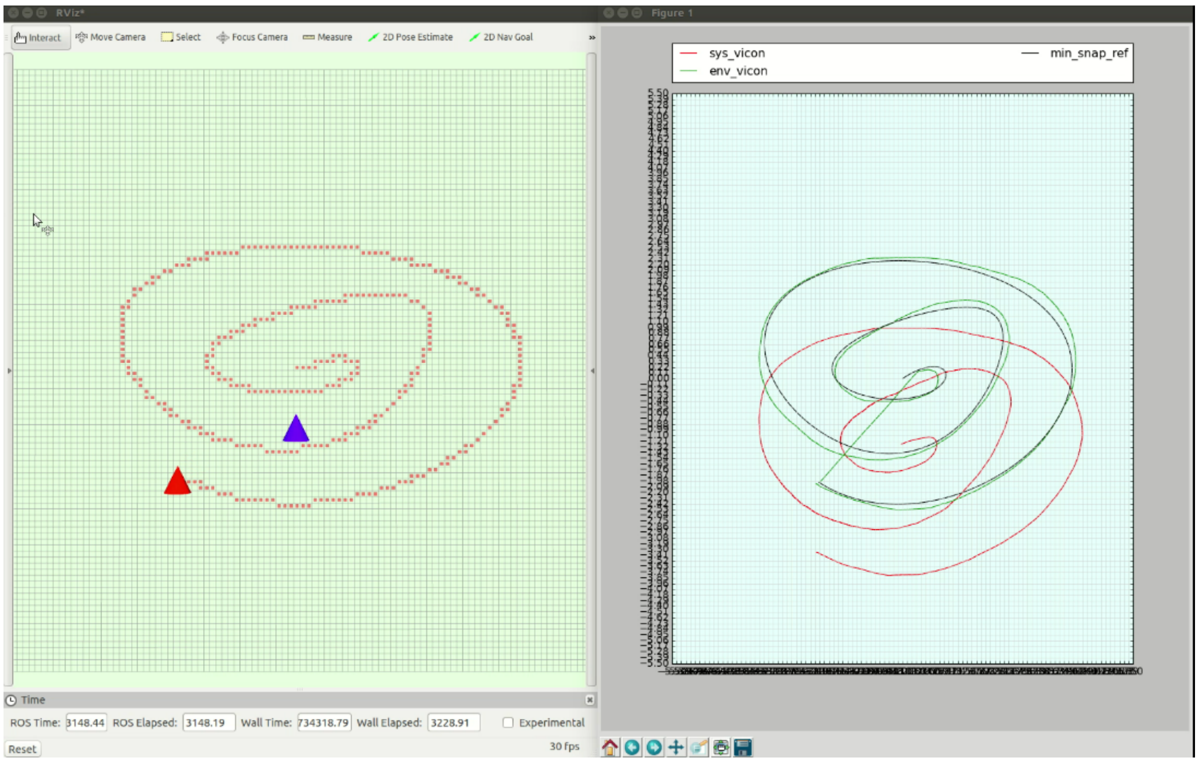


Figure 6.5: Follow environment in a 10 * 10 grid size with real Quad

We can observe that we the real quad the trajectory are not as good as in the simulation. That's the reason why we first worked on the PVA controller to improve the control. Note that this simulation was made using only position control.

After playing around with this experiment, I tried to make it more continuous. The first idea is just to reduce cell size so to increase the number of cell and state.Taking in consideration the fact that the geometry of the quadrotor is a square in a grid, We applied this specification on a 100 * 100 grid size where the cells were twice smaller than the quadcopter size. In the next figure, we will observe a more continuous behaviour.The trajectory describes by the system and the environment looks exactly the same except frome the offset which is due to our specification. On the implementation part, instead of calling the synthesized controller, then moving at the same time the environment and the system we do not use this anymore in this experiment. Here the environment moves independently on his own thread and the system has to check for every state change of the envrionment and react as fast as the state change appears. This is the way we manage to have this continuous behavior.



ROS Time: 3148.44 ROS Elapsed: 3148.19 Wall Time: 734318.79 Wall Elapsed: 3228.91

Figure 6.6: Follow environment in a 10 * 10 grid size in Gazebo 100*100

Finally, the last experience, we are going to talk about is an extension of the experience that imply environment with low rate change of state with uncontrollable environments which sometimes play again us. Basically,in this experience we will have 3 quadrotors. One is fully controlled and can get target point by an external user when he cliked on a cell in the grid. This quad must also avoid the two others quadrotor. The second quad is not controlled and use the controller synthesize here6.2 to move in the environment and receive input from an user but he can also avoid avoiding the third quadcopter but have no knowledge about the first quadrotor. The third quadcopter is just trolling around with no knowledge about the others.

The goal of this experiment is to show that we can make complex experiments with TuLiP but we will quickly get limited by the grid size we can have. We only made this experiment in code simulation and it was working. Someone need to execute that code with the real quadcopter to finish the works. Probably the next intern in u-t-autonomous.

Chapter 7

Conclusion

In conclusion, Our minimum snap implementation coupled with the PVA controller applied to a quadcopter can lead to a good trajectory tracking performance. This can thus be used for aggressive maneuverer in an environment where some aggressive maneuverer may be done. But for this to be possible, someone may waste time tunning the PID controller. About the minimum snap implementation, we made a C++ library that can be compile on every machine which contained some basic library that will be include in the documentation of the code. We also made a ROS node which is based on the C++ implementation and that can be call by other other in a service/client manner. The next step is to try to include this implementation directly inside a flight controller. The same thing should be done for PVA controller. We believe that making the PVA main loop directly inside the flight stack will increase highly the performance of the controller. But it is difficult for the moment to do so because, PX4 is in perpetual development and someone need to maintain a modified PX4 to be able to still get new PX4 release.

We learn a lot about Formal verification problem and LTL based synthesized controller that will facilitate our future classes about this subject. However, I wished I had more time to test all the experiments algorithms, controller and code I have created in Unreal Engine for the real quad. In particular the last experiment including the 3 quads. All the code written during the internship will be pushed soon on the github repository of the u-t-autonomous group[11]. One main drawback of TuLiP toolbox is the fact that the complexity is exponential with the number of state. So major experiments can be done with that but this is just a scalability problem. In a near future, computer will be power enough to let us enjoy more about this kind of tool.

Bibliography

[1] Wikipedia.
Occupancy grid mapping, .
URL `https://en.wikipedia.org/wiki/Occupancy_grid_mapping`.

[2] Vicon Company.
Vicon.
URL `https://www.vicon.com/what-is-motion-capture`.

[3] Morgan Quigley, Brian Gerkey, Ken Conley, Josh Faust, Tully Foote, Jeremy Leibs, Eric Berger, Rob Wheeter, and Andrew Ng.
Ros: an open-source robot operating system.
In *Proc. of the IEE Intl. Conf. on Robotics and Automation (ICRA) Workshop on Open Source Robotics*, May 2009.

[4] Open Source Robotics Foundation.
What is ros ?
URL `https://wiki.ros.org/`.

[5] Gazebo.
Gazebo for robot simulation.
URL `http://gazebosim.org/`.

[6] Shital Shah, Debadeepta Dey, Chris Lovett, and Ashish Kapoor.
Aerial Informatics and Robotics platform.
Technical Report MSR-TR-2017-9, Microsoft Research, 2017.

[7] AirSim.
Airsim microsoft.
URL `https://github.com/Microsoft/AirSim`.

[8] Inc Qualcomm Technologies.
Qualcomm snapdragon flight.
URL `https://developer.qualcomm.com/hardware/snapdragon-flight`.

[9] PX4.
Off-the-shell esc with snapdragon flight, .
URL https://dev.px4.io/en/flight_controller/snapdragon_flight.html.

[10] PX4.
Px4 developer website, .
URL https://dev.px4.io/.

[11] u-t autonomous.
Ut autonomous group github, .
URL https://github.com/u-t-autonomous.

[12] u-t autonomous.
Px4 based centralized off-board node for quadcopter autonomous control, .

[13] Wikipedia.
Method of ziegler-nichols, .
URL https://fr.wikipedia.org/wiki/M%C3%A9thode_de_Ziegler-Nichols.

[14] Cyril Mansour Franck Djeumou.
Flying quadcopter with eye tracking.
URL https://www.youtube.com/watch?v=AfosHcUJR9M.

[15] Daniel Mellinger and Vijay Kumar.
Minimum snap trajectory generation and control for quadrotors.
pages 2520–2525. IEEE, 2011.

[16] Timothy A. Davis.
Algorithm 915, suitesparseqr: Multifrontal multithreaded rank-revealing sparse qr factorization.
2011.

[17] Timothy A. Davis.
Methods for sparse linear systems.
URL http://faculty.cse.tamu.edu/davis/publications.html.

[18] Charles Richter, Adam Bry, and Nicholas Roy.
Polynomial trajectory planning for aggressive quadrotor flight in dense indoor environments.
In *Robotics Research*, pages 649–666. Springer, 2016.

[19] Taeyoung Lee, Melvin Leok, and N.H. Mcclamroch.
Geometric tracking control of a quadrotor uav on se(3).
pages 5420 – 5425, 01 2011.

[20] Wikipedia.
Linear temporal logic (ltl), .
URL `https://en.wikipedia.org/wiki/Linear_temporal_logic`.

[21] N. Ozay H. Xu T. Wongpiromsarn, U. Topcu and R. M. Murray.
Tulip: A software toolbox for receding horizon temporal logic planning.
In *in Proceedings of the 14th International Conference on Hybrid Systems: Computation and Control, ser. HSCC*, 2011.

[22] TuLiP Website.
Tulip.
URL `https://tulip-control.sourceforge.io/doc/intro.html`.

[23] Armin Hornung, Kai M. Wurm, Maren Bennewitz, Cyrill Stachniss, and Wolfram Burgard.
OctoMap: An efficient probabilistic 3D mapping framework based on octrees.
Autonomous Robots, 2013.
doi: 10.1007/s10514-012-9321-0.
URL `http://octomap.github.com`.
Software available at `http://octomap.github.com`.