Department of Information Engineering and Computer Science

BACHELOR'S DEGREE IN
SOFTWARE ENGINEERING

FINAL DISSERTATION

AUTONOMOUS QUADCOPTER FLIGHT

Supervisor
Prof./Dott. Luigi Palopoli

Student
Massimo Girardelli

Academic year 2019/2020

Abstract

This thesis is the design and development of software applications for controlling quadcopters and other flying drones.
Quadcopters have gained more attention in the past years due to their versatility and potential in various fields, including surveillance, aerial photography and small object transportation and delivery; an application to interface with the drone is a useful tool and a first step for completing these tasks.
Although ready to use solutions and similar open source projects exist, an easily configurable, with a graphical interface and indoor capabilities that responds to external inputs doesn't and this project aims to provide a working example of all these features that can be customized for different application fields.
The project was developed around the ROS2 middleware, PX4 flight control software [1] and the OptiTrack [2] mocap system; the drone can work in conjunction with other robots that support either of those platforms and in a modular fashion, where each component can be substituted with a similar one and expanded with more functionality.
The final result is a working flying autonomous drone that can be interfaced both via software directly or through a graphical interface to perform various basic tasks like taking off, landing in a specific location, orbiting around a point, following a path, or be controlled manually from the computer.

All the code developed for this thesis is available in the following GitHub repositories:

Main repository https://github.com/Massiccio1/px4-py

Docker image repository: https://github.com/Massiccio1/px4-22

Commander messages repository https://github.com/Massiccio1/px4_msgs

Optitrack client repository https://github.com/Massiccio1/optitrack_interface

Rviz visualizer (fork) https://github.com/Massiccio1/px4-offboard

Chapter index

Table of Figures

ACRONYMS, ABBREVIATIONS AND IMPORTANT TERMS

PX4	PX4 is the Professional Autopilot. Developed by world-class developers from industry and academia, and supported by an active world wide community, it powers all kinds of vehicles from racing and cargo drones through to ground vehicles and submersibles.
ROS2	robot Operating System version 2, middleware that enables fast and easy messages communications between different nodes on a network
NED	North-East-Down coordinate system where the X-axis is pointing towards the true North, the Y-axis is pointing East and the Z-axis is pointing Down, completing the right-hand rule.
FRD	Front-Right-Down coordinate system where the X-axis is pointing towards the Front of the vehicle, the Y-axis is pointing Right and the Z-axis is pointing Down, completing the right-hand rule.
ENU	xEast-yNorth-zUP coordinate system, the classic Cartesian coordinate system
FLU	xFront-yLeft-zUp coordinate system, coordinates from the drone point of view
SITL	Software in the Loop, a simulation tool that reproduces the flight controller behavior
LTS	“Long Term Support”, an LTS version of software receives supports for many years to come and thus is preferred for compatibility
GUI	Graphical User Interface
OptiTrack	Motion capture system, used to detect the drone position with a tracker
Commander	Main computer from where the drone is controlled
Companion	Single board computer mounted on the drone

1 Introduction

Drones and Unmanned Aerial Vehicles (UAVs) are used more and more in the field of robotics because of their ability to navigate more freely in the air and not be limited on the ground while at the same time keep small dimensions.
This thesis focuses on the creation if a working flying drone that can be used a starting point or template to create a more specialized version of a specific field as it offers the code infrastructure that can be expanded to interact with more sensors commands or events.
To be able to perform flight mission the drone must be able take off, follow a moving point in space, follow a path of points and land; with the combination of these functionalities the drone can perform almost any flight task.
With the python and ROS interface to allow to interact with other sensors and actuators the goal of the project is complete.
This was not an attempt to create the perfect algorithm for path planning and minimization of error/energy.

1.1 Thesis organization

The work of this thesis is organized in six chapters, here briefly summarized.

Chapter 1 – Introduction: an introduction to the thesis, the goals and the difference with drone systems that already exist.

Chapter 2 – Hardware: a description of the hardware used and the features they provide, the decisions made behind these choices and the problems related to the components.

Chapter 3 – Software: a description of software needed, from the operating system to the software modules and programs developed and how the interact and exchange information.

Chapter 4 – Simulation: a chapter dedicated to the simulation, simulator and the monitoring of the system during development.

Chapter 5 – Flight: summary of the steps to launch the drone, the behavior mid air, the features it provides and some flight data analysis.

Chapter 6 – Conclusions and Discussion: this final chapter presents the conclusions drawn from the project and includes personal reflections on the process and outcomes.

Words highlighted are related to the software code.

1.2 Background and motivation

This project was born from the need to a way to control a drone that could be capable to communicate and work together with a terrestrial land crawler in an agriculture field. Professor Palopoli offered this opportunity to create a general purpose system for the flying drone that could be used in the future. With the possibility to control a swarm of drones at the same same time.

I wanted to work on this project because I'm fascinated by drones and automation and the ability to control robots with software and without human interaction.

The drone, OptiTrack and single board computer were provided by the laboratory.

1.3 **Existing drones**

The concept of a flying drone is now a novelty in the world, and already offer functionalities like pathing, missions, return to home and *follow me* features, but these features are handled in their entirely inside the system and it's almost impossible to create a dynamic custom action plan based on sensor data. That's why this open source controller and system was created, every action and reaction can be changed in the software to fit every possible need.

2 **Hardware**

This is the list of components used and some reasons behind the choices that were made.

This hardware can be swapped with other alternatives, depending on the availability of the components or specific needs.

2.1 **Holybro X500 V2 Development Kit**

Main frame of the drone, it's a good all-in-one solution for a quadcopter, it mounts 4 920 Kv motors, perfect for this application.

The Kv rating of a motor describes the relationship between the voltage provided and rotation speed without load and is measured in $\frac{RPM}{V}$.

Lower Kv motors offer more torque and less fan speed, which is desired in this application give the weight of the whole system, a higher Kv motor would struggle more with lifting and excel more in maximum speed.

The Pixhawk 6C flight controller offers the right connectivity and easily provides the CPU and RAM that we need.

The frame is a lightweight carbon fiber and is strong enough to lift the drone fully loaded.

Figure 1: Holybro X500 V2 Development Kit [3]

2.2 **5000 mAh Li-Po batteries**

A massive 5000 mAh Li-Po battery powers the motors, the flight controller and the companion computer.

The 4S configuration offers 14.8V, when charged it can reach 16.8V.

To preserve the lifespan of the battery is not advisable to discharge the battery below 12,8V.

The flight time is around 15 minutes with moderate use and 10 minutes with heavy use of motors like in case of frequent height change.

The exact model used is: “Dinogy Ultra Graphene 4S 5000mAh 80C”.

2.3 **ROCK 4 C+ (Companion)**

A single board computer, similar to a Raspberry PI but with a 64-bit architecture. It's a hexacore ARM processor with 4GB of LPDDR4 RAM.

The 64-bit architecture proved to really useful for compatibility for docker and possible expansions later; 4GB of RAM are plenty for this use but the initial setup required a lot of compilation time; storage is provided with an external SD card.

The GPIO PINS are used for serial communication with the flight controller. Connection is done with WiFi and a SSH shell, power is delivered via USB with a splitter and converted from the battery.

The main issue with this board is the connectivity, like many other single board computers, a big part of the I/O is connected through a shared bus and this lowers the overall throughput of data; for this particular application the network connectivity was consistently low with random spikes in latency.

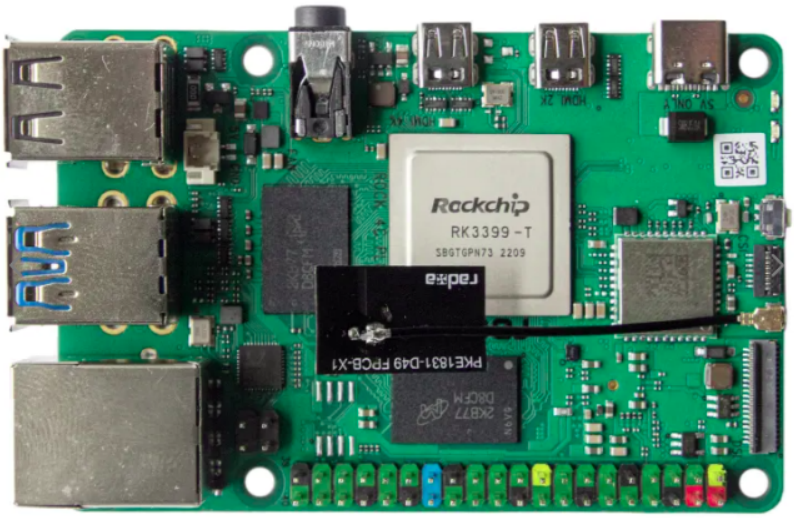

Figure 2: ROCK 4 Model C+ 4GB Single Board Computer [4]

The problem was not the WiFi module itself because alternatives adapters and Ethernet cables were used and the underlying problem was still present..

In the end the limited networking was a compromise that caused troubles in development but the final result is almost unaffected since the system can operate at a slow speed and even with 100ms of delay in time the system was stable.

Another problem was the old operating system: the company behind this board only supported version of Ubuntu up to the version 20.04, to keep this project up to date a Docker container was used to run Ubuntu 22 and ROS2 Humble.

2.4 **Commander computer**

Any computer that can run the commander programs, preferably Linux with Ubuntu 22 (reasons at Chapter 3.1), alternatively a computer that can run Docker.

It’s possible to use the Companion itself also as a commander.

2.5 OptiTrack

Motion Capture Systems mounted in the laboratory, the precision is <1 cm, latency is around 30ms, publish rate is 360Hz by default but the drone remains stable up to 50 Hz of data received in case of network congestion.

Tracking is done by recognition of a special tracking marker placed on the drone, each model saved has it's own ID that will be transmitted with the data.

The 8 cameras placed around the flying area, provide an accurate position estimation. The position is sent through the network and must be decoded by a client.

This system is heavily dependent on a proper configuration of the cameras and a correct model of the tracking marker.

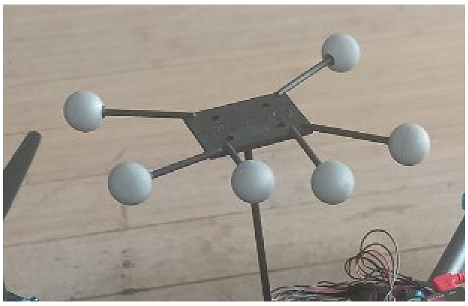

Figure 3: tracking marker on top of the drone

3 Software

In this software chapter I will explain the systems and tools used to bring this project to life and the inner workings of the components necessary to make the drone fly.

3.1 System

In this section I will focus on the operating system and system-wide tools, operating systems and software used, every other program runs on top of these:

3.1.1 Ubuntu 22

This was a very important decision since every distribution of Ubuntu has it's own ROS2 distribution and the latest LTS (for now) only supports Ubuntu 22.

It's possible to run ROS2 Humble on other distributions but it must be compiled from source likelihood of error or incompatibilities grows bigger, especially on embedded systems.

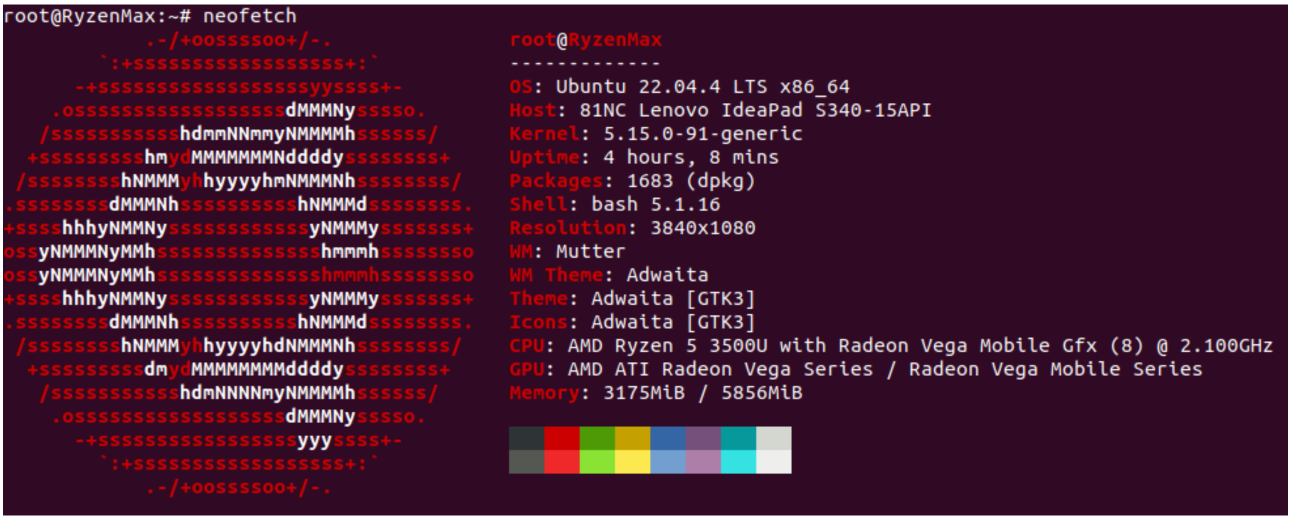

Figure 4: My Commander pc specifications

3.1.2 Docker

Docker is an amazing tool for virtualization and portability; with docker it's possible to create an image of a system and run a container almost like a lightweight virtual machine.

The container has all the libraries needed to run an entire system and is widely used for testing and compatibility since it's possible to run a completely different operating system inside a container.

In docker you can change and customize a base image by running shell commands declared in a **Dockerfile**, each line of command produces a **layer** that contains the changes tin the filesystem, then all the layers are stacked together to produce the new **image**.

If it's impossible to use or install Ubuntu 22 a good alternative is a docker container, like I did.

Two main docker images were built, one for the commander and the other one for the companion [5].

The base image is the official ros:humble, it supports both **amd64** and **arm64/v8**; a core image with PX4 dependencies and messages installed is built on top, this middle step was useful in development to speed up future builds.

```
RUN pip install setuptools==58.2.0

WORKDIR /root

FROM stage1 as px4

RUN git clone https://github.com/eProsima/Micro-XRCE-DDS-Agent.git
WORKDIR /root/Micro-XRCE-DDS-Agent
RUN mkdir build
WORKDIR /root/Micro-XRCE-DDS-Agent/build
RUN cmake ..
RUN make -j8
RUN make install
RUN ldconfig /usr/local/lib/
```

Figure 5: Snippet from the core Dockerfile

On top of the core, two separate images for companion and commander were built.

Docker uses the **docker build cache**, this smart caching system allows the build to reuse an already created layer without the need to run the command again; to optimize for this technology, we can apply some best practices like splitting commands into multiple smaller commands, put the less likely to change commands at the beginning and delaying as much as possible the COPY docker commands.

This way we can have better build times after the first one at the cost of higher disk usage but it's a worth trade while developing; for a release build we can use the experimental –squash tag to compress all the layers into one, saving disk space and improving performance.

For my system I used this 3 stage build:

- **core**: common image with all the PX4 libraries
- **companion** or **commander**: platform specific image
- **update**: always pulls the up to date code written by me

3.1.3 **ROS2 Humble**

Middleware used to share all messages between applications and to launch most of the applications, ROS resolves around Nodes, Topics and Messages:

- **Nodes:** a process interfacing with the ROS libraries, different nodes can run on different machines while being connected through networking and be visible to each other.

- **Topics:** a pipe-like virtual structure within ROS that can be used to share messages between different nodes, it can be interfaced as a publisher or as a subscriber, each topic routes one specific message type.
- **Messages:** data structures that travel through the nodes, it is not system-dependent and provides primitive types like booleans, integers, floats, arrays and nested messages types. Custom messages were built for this project.

The optimal QoS (Quality of service) options were used:

➢ **Best effort**, to reduce network congestion, most messages are ephemeral and gets overridden a few milliseconds later, therefore if some messages are dropped it doesn't affect the system

➢ **Transient local**: the publisher is responsible for delivering the last message tho subscribers who joined after it was published

➢ **Keep last**: only keeps the last message, previous values are invalid anyway

The LTS version "*Humble*" for long term support [6].

3.1.4 **Python 3.10.12**

Python was used for the ease of development and speed of deployment as it doesn't need compilation; python with PX4 has also much more documentation and examples than any other language.
Other programming languages that have better performance like C++ were considered but discarded for the following reasons:

- ROS2 compilations have massive overheads on C++, even for well written cmake and headers files while python has none.
- Very hard to debug a ROS application that interface with a real machines.
- a part of the code runs on the companion computer that, while powerful, is not really suited for continuous compilations (installing all the PX4 dependencies and building the OptiTrack package took more that 1 hour and a half).

The runtime overhead of python gets completely eclipsed by the network delay and the program itself is not heavy, most of the complexity is in the logic of the operations and not in the execution itself.

The version 3.10.12 is the officially supported by ROS2 Humble.

3.1.5 **PX4**

Open source flight control solution for drones, it's used in a wide range of use-cases. It provides a flexible set of tools for drones and abstraction from the hardware. It's funded by the Linux Foundation.

PX4 takes care of all the algorithms, data analysis and merging of data into readable messages that cam be interfaced with ROS.

For this project the **position** control mode was used to be able to sent 3D points as a **setpoint** destination and the **offboard mode** that allows controlling of the drone with software and without a radio.

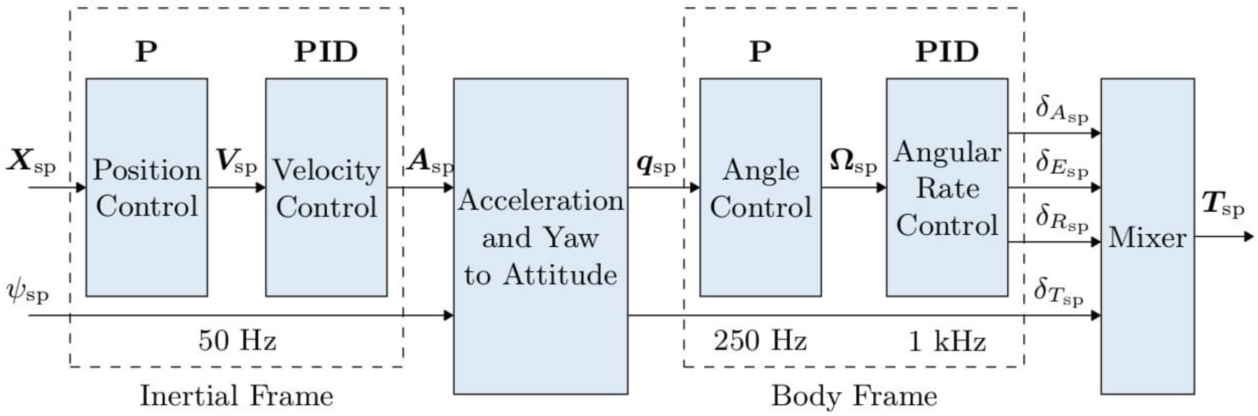

Figure 6: PX4 Multicopter Control Architecture

the position X and angle ψ are processed into acceleration A and then into thrust force T for the motors

To translate from a high level position to low level controls, PX4 use a combination of P and PID controllers:

The P controller (Proportional Controller) is a simple feedback controller that adjusts the output based on the current error between he desired setpoint and the current position, it provides a fast start while slowing down when close to the target. It only has one parameter P that regulates the strength.

The PID controller (Proportional-Integral-Derivative Controller) on the other hand is the most common controller for drones and it's widely used in the UAV field; it is based on tree parameters:

- **P**: proportional term, similar to the P controller, it provides an output proportional to the position error.
- **I**: integral term, it takes in consideration the past errors and reduces steady-state errors.
- **D**: derivative term, predicts the future error based on the rate of change, providing a damping effect before reaching the destination, reducing over shooting.

PID setting were slightly changed in favor of a more steady system sacrificing responsiveness to prevent problems in case of delays in the tracking signals.

Sensor data is processed by **EKF2** (Extended Kalman Filter) that combines all the data from sensors to produce the state of the vehicle, including and not limited to: position, rotation quaternions, velocity acceleration, magnetic field and wind velocity.

PX4 supports both versions of the robotic operating system, but while the communication between ROS2 and the autopilot is done through a ROS2-PX4 bridge (MicroXRCEAgent).

There are many ways to interface with the PX4 messages but this is the preferred way; the agent will run on the companion.

3.2 **Companion**

The companion is a single board computer mounted on the drone, it's role is to run the **MicroXRCEAgent** service, the command to run is:

sudo MicroXRCEAgent serial --dev /dev/ttyS2 -b 921600

The MicroXRCEAgent starts a ROS2 node that published topics with information like current position, velocity, status, and subscribes to other topics to receive commands, tracked position and keep-alive signals. It's a bridge from the flight controller running PX4 software and the other components [7].

Communication with the flight controller is done on the serial port ttyS2 with a baud rate of 921600 to transmit all the data.

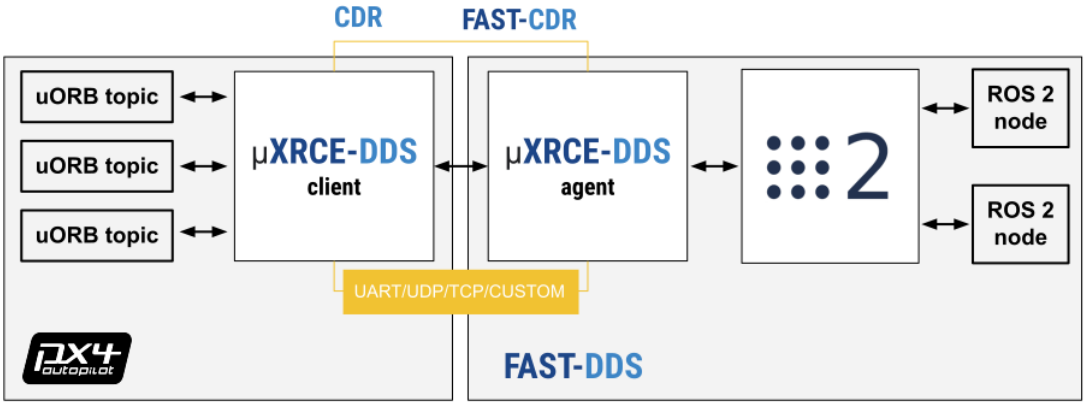

Figure 7: MicroXRCEAgent scheme

flight controller on the left and companion *board* on the right

For this particular use case a custom firmware was build to access particular information like land detection, battery status [8].

3.3 **Commander**

The commander computer is the main computer from where the drone is controlled, it's role is to run the following software:

3.3.1 **px4-py.py**

(also known as **controller** from her on) The main process of the system [8], keeps the connection with the companion, receives updates on position, velocity, status etc. from the companion, receives data from the GUI, and most importantly elaborates all the data and messages to produce an action to send to the drone.

To control the companion, a 2 Hz message must be kept alive, this signal tells the companion that the connection is healthy and sets the PX4 mode to **Offboard**, this mode allows the drone to be controlled remotely without a radio.

This process is an infinite loop with callbacks when a new message from the drone is detected, the main class OffboardControl(Node) keeps a record of the latest received in global variables so any part of the system has access to them.

Tasks and actions are processed in separate threads and only one action can be executed at a time, if a new actions comes, the previous one is canceled.

Some actions are simply redirected to the companion (like arm, disarm, shutdown), other action that continue over time start and are handled by a thread.

If there is no GUI (set in config.py), it proceeds with the selected mode in the configuration file, otherwise it listens for action messages.

The controller is the only one who sends commands to the companion, this way the GUI becomes an extension of the program and can be closed/relaunched at will without causing problems to the main process.

To add more sensors of actuators, it's just necessary to add the subscriber and a callback for the Ros message.

3.3.2 **gui.py**

This graphical interface is a quality of life tool that considerably speeds up testing and validation since it makes possible to interface (in a human mode) with the controller, enabling runtime changes to the operating mode without editing and restarting the whole system.

It was built from scratch using the python library PySimpleGUI version 4.60.5, which is free to use and the window can be passed through docker with the X11 compositor.

It has many functionalities and offers all the relevant information on screen:

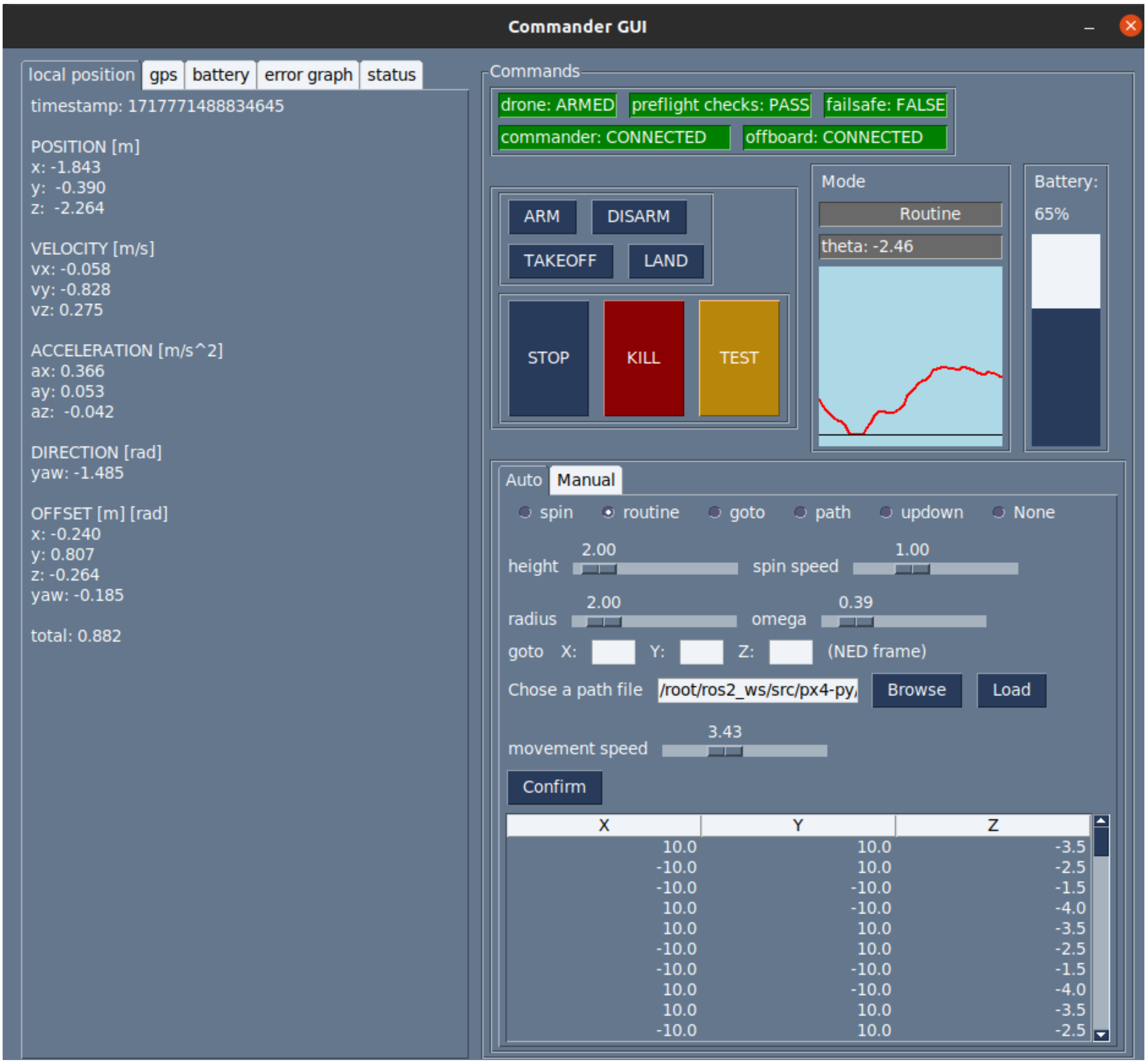

Figure 8: GUI main screen

- ➢ **ARM** arms the drone if the above conditions are met
- ➢ **DISARM** disarms the drone before it takes off or when it landed, does not work mid flight
- ➢ **TAKEOFF** takes of to the height declared in src/config.py by the variable takeoff_height
- ➢ **LAND** lands at (0,0) local frame, position cannot be overridden, this is a limitation of the off-board mode because it would require a different mode to land normally on location, discon-necting from the commander
- ➢ **STOP** stops any activity and hold the position

- **TEST** button to test features, currently lands on a random position
- **KILL** emergency stop, terminates the flight immediately and stops the propellers, even mid air (USE WITH CAUTION)
- **SPIN** spin on self with omega = spin speed slider
- **ROUTINE** starts hovering in circle around (0,0), at height = height , omega = omega slider,radius = radius
- **GOTO** goes to position x,y,z in NED frame (CAUTION WITH Z COORDINATES)
- **PATH** load a path file like src/medium.json and the drone fill follow all points with speed = movement speed slider
- **UPDOWN** goes up and lands, mainly for testing
- **NONE** default, no activity, the drone will hover
- **GRAPH** shows a simple graph of position offset between the set position and the current position, gives a rough estimate of the offset, X axis is time while the Y axis is the euclidean distance between the set point and the current position. I doesn't offer a 3D representation but a quick visual information which can be useful for the user
- **MANUAL** is a tab from which a very simple interface to manually control the drone and help make small movements or adjustments, there are sliders to modulate transmission rate and speed of movement

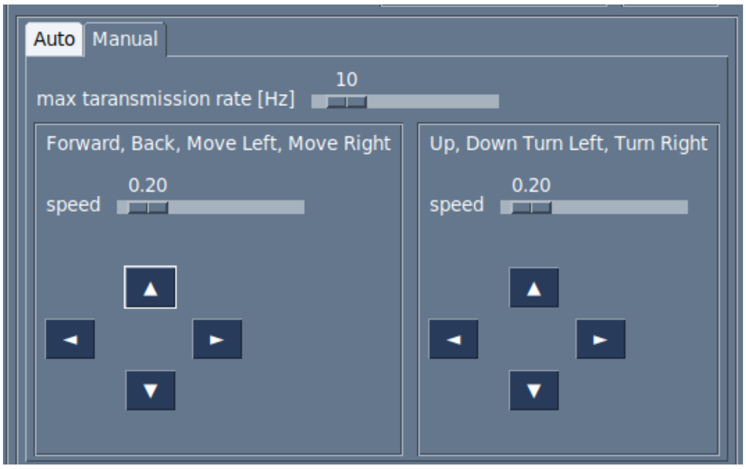

Figure 9: GUI manual controlls

There is also a simple console to control the drone manually with simple 3D positioning and 2D rotation, with speed sliders and transmission rate, this is very useful while testing, allowing the user to reposition the drone and validate the tracking.

```
uint64 timestamp

#bool ready

#check config.py for modes
uint32 mode

bool ready

##description of parameters for each mode
# routine f1=omega, f2=radius, f3 = height
# path points = array of points
# spin f1 = omega, f2 = height
# updown no params
# goto fa1 xyz, f1 yaw, f2 speed
# landing fa1 xyz

float32 f1
float32 f2
float32 f3
float32 f4

float32[] fa1
float32[] fa2
float32[] fa3

commander_msg/CommanderPathPoint[] points
#int8 path_index

bool b1
bool b2
bool b3

int32 i1
int32 i2
int32 i3

```

Figure 10: Comamnder message structure for mode selection

General purpose variables are used, it’s up to the program to correctly decode the information

The GUI and the controller communicate with custom Commander messages made by me [9], the style is similar to the original PX4 messages with an integer defining the mode and a series of arrays where to pass thee data, it’s up to the receiver to correctly parsing the meaning.

This program is independent and the whole system can work without this piece of software since it doesn’t communicate directly with the companion but only to the commander, this is a design choice to improve compatibility and automation.

Every information is accessible with the command line or a program with a ROS2 interface and every action is applicable with a script.

3.4 **Tracking**

An external tracking system is required for the drone to know it’s correct position in space. The internal gyroscope and acceleration sensor provide difference in position or deviation from the original start point.

The estimation is crucial for the correct functioning of the drone but is can not be used alone, it must be paired with another tracking system. The merging of the two measurements produces the final local position of the drone; normally this system is a GPS mounted on the drone but it doesn’t always provide the precision that we need for a precise flight. Drifts of **up to 5 meters** were recorded in the position with the drone on the ground. Also GPS doesn't work indoor and the area around the university has strong magnetic fields that interfere with other sensors like the magnetometer.

These problems led to the need of a different method to provide the position to the drone, the **OptiTrack** motion capture was already present in the laboratory and provides sub-centimeter precision when correctly calibrated, but any other tracking device can be used as long as it can publish ROS pose messages directly or indirectly works.

The software needed to run the tracking system are:

3.4.1 **Motive**

The proprietary software that is physically attached to the cameras, it published multicast messages int he network, must be configured with the shape of the tracker used.

Must be interfaced with a client to access the date published.

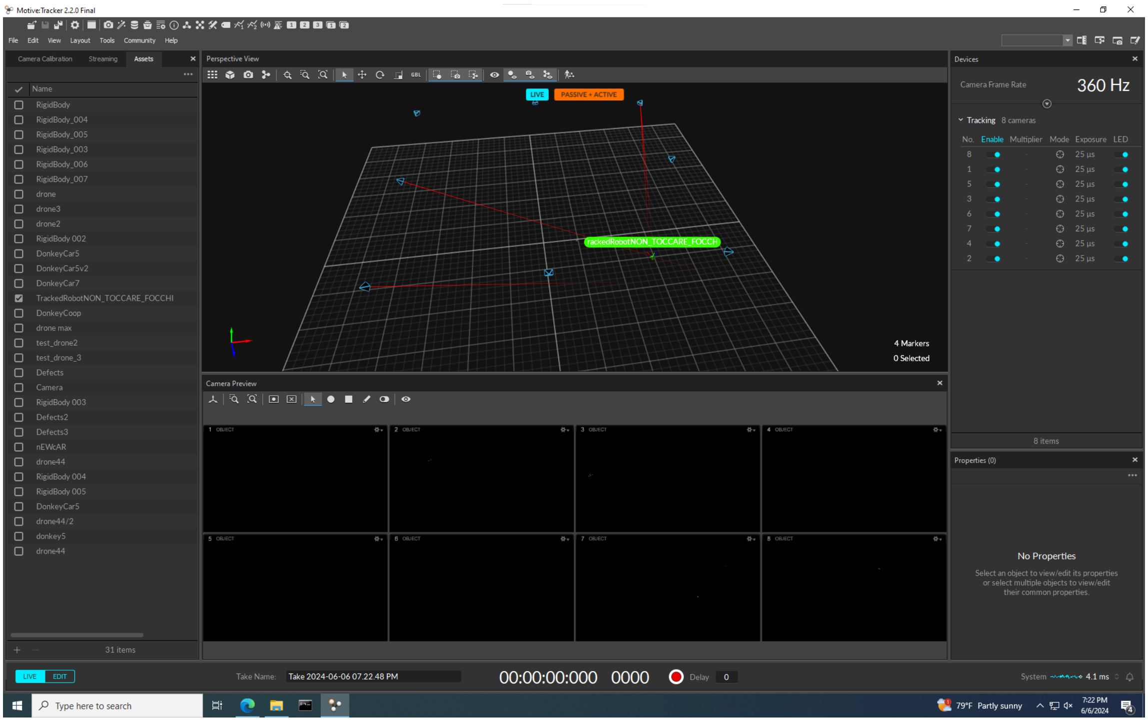

Figure 11: Motive software

3.4.2 **optitrack_interface**

python nat net client, it decodes the OptiTrack messages and publishes geometry_msgs.PoseStamped, a standard ROS message that can be accessed from the other programs.

It's configured with the ID of the tracking device registered in Motive to avoid publishing messages from a wrongly detected body.

3.4.3 **opti-to-px4.py**

Python program listens to the OptiTrack messages and converts geometry_msgs.PoseStamped to VehicleOdometry messages, a PX4 message type and throttles the output to a stable 100Hz

WHY?

Why doing all these conversion of messages?

Let's proceed with steps: the first conversion from Motive to OptiTrack_interface is need because Motive is a general purpose tracking system and publishes it's own format and not designed to work out of the with ROS2 so this step is unavoidable.

But in the second part from OptiTrack_interface to PX4 we convert ROS2 message to ROS2 message, can't we just skip the middle part and do Motive-PX4?

The answer is yes, but not always.

Let me explain, this setup was done in an existing laboratory and a standard PoseStamped is recognized by any ROS2 program and is the most suitable for the data transferred, so an existing setup might alerady have this system setup.

So a PoseStamped message is more likely to be supported by existing setups and it's more easily replicable and faked for testing purposes.

In any case I developed an all in one solution for a direct Motive-PX4 and it's called all_in_one.py

This is all the software that is used to make the drone fly which is either proprietary software, forked and modified code or developed from scratch by me, all details are in the main repository.

More programs are used in the simulation section but it's not directly needed for the drone to fly.

3.5 **Extra**

A list of other programs necessary for the setup, development and debug of this project

3.5.1 **QGroundControl**

QGroundControl or QGC, is a software for UAVs, it's an interface tot he flight controller and it was used to set parameters and retrieve flight logs, install firmware and PID tuning.

It can communicate with flight controllers using the MavLink protocol.

3.5.2 **PlotJuggler**

A graphical interface to plot and filter flight logs and ROS messages, analyzing the logs is an essential steps for debugging and prevent crashes.

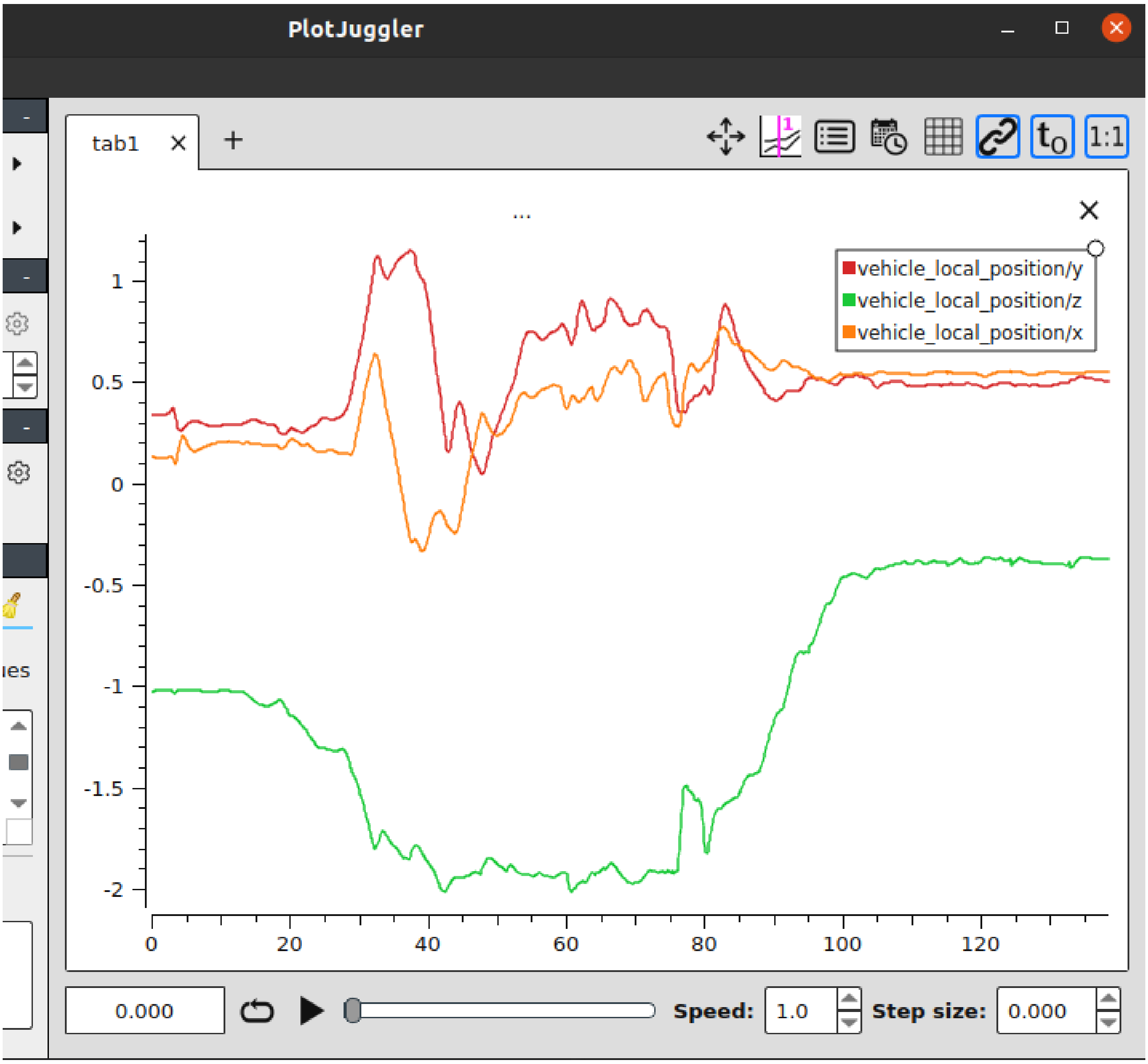

Figure 12: PlotJuggler interface

Small flight sample

4 **Simulation**

Flying simulations offer numerous advantages, the most important is being able to test new code in a safe virtual environment, there are zero consequences for a crash; meanwhile flying a real drone in a close environment while testing functionalities is definitely a recipe for disaster.

The simulator used is **Gazebo.**

4.1 **Gazebo**

Gazebo is a collection of open source software libraries designed to simplify development of high-performance applications.

It’s widely used in robotics because it supports many plugins that can be developed for each system, it’s very modular and offers a fine customization.

Gazebo supports the SITL drone simulation that was needed directly from the PX4 package.

This simulator was used to verify the response of the drone to the different commands and possible situations that might occur.

Gazebo could not be used alongside with the physical tracking system due to the discrepancy between the simulated position and the real position while, so a loopback of the current simulated position was used to simulate an external tracking.

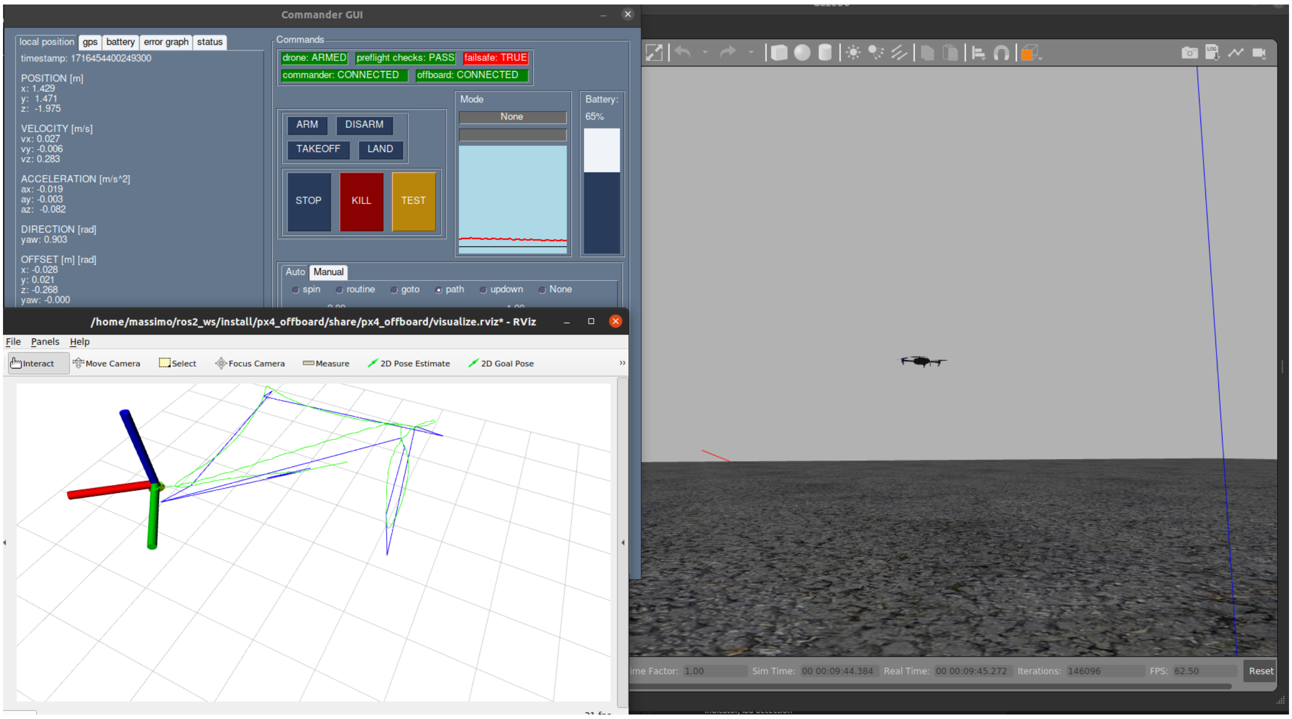

Figure 13: Fully functional drone simulation.

GUI on top left, RVIZ in bottom left and Gazebo on the right

4.2 **RVIZ**

RVIZ is another program usually paired with Gazebo, it doesn’t directly simulate but allows the user to visualize ROS messages in a very convenient way.

This tool was really useful when debugging the tracking system because I could overlay the tracking position what the current drone position and visually validate the if the tracking was correct.

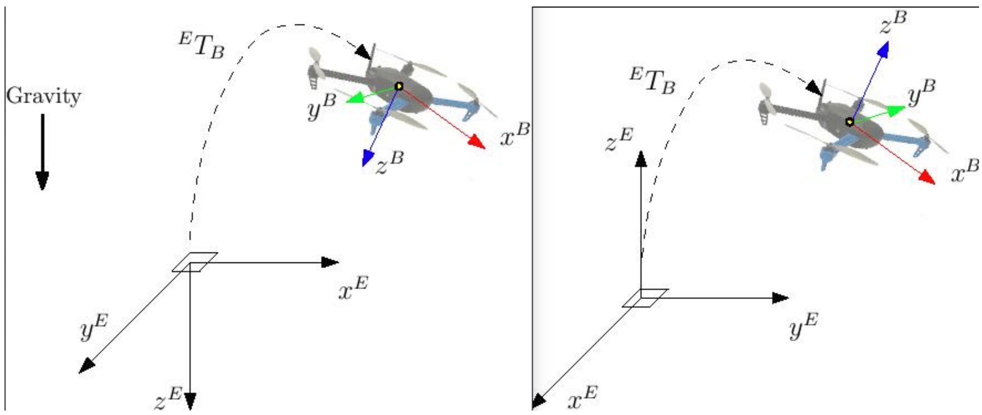

Figure 14: NED to ENU coordinate conversion

While the front end displays the evolution of the system, the backend is a fork of an existing project [9], properly changed to receive PX4 messages, organizes a history of messages and set, converts the coordinates from NED and FRD frames to ENU and FLU, the former are commonly used with flying vehicles where the Z axis represents the negative distance from the ground while the latter are the common coordinates; this step is done for visualization purposes, every aspect of the flying drone is done in NED and FRD frames.

```
---
timestamp: 1717771298323152
timestamp_sample: 1717771298323152
xy_valid: true
z_valid: true
v_xy_valid: true
v_z_valid: true
x: 0.03394895792007446
y: 0.004721784498542547
z: 1.4242117404937744
delta_xy:
- -0.0009327434818260372
- 0.007269835565239191
xy_reset_counter: 2
delta_z: 1.131436147261411e-05
z_reset_counter: 2
vx: 0.0380549281835556
vy: 0.0012560335453599691
vz: 0.37913405895233154
z_deriv: 0.037182532250881195
delta_vxy:
- 0.018127841874957085
- 0.010561088100075722
vxy_reset_counter: 2
delta_vz: -0.030338719487190247
vz_reset_counter: 2
ax: -0.0017338460311293602
ay: -0.009593269787728786
az: 0.10591185837984085
heading: 1.593224287033081
unaided_heading: 0.03209805116057396
delta_heading: -0.00022814940894022584
heading_reset_counter: 89
heading_good_for_control: true
xy_global: true
z_global: true
ref_timestamp: 1104758609980
```

Figure 15: Raw local position message from PX4

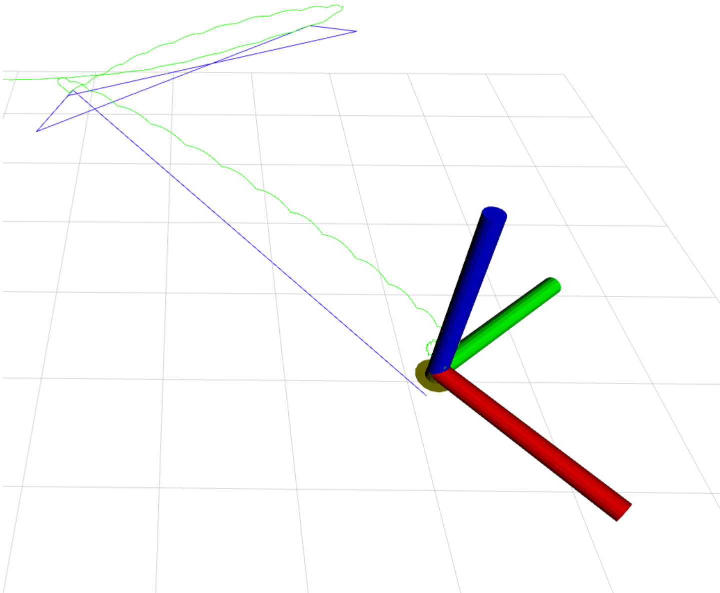

Figure 16: RVIZ screen

the blue line is the theoretical path, the green line is the real path taken

5 Flying

Now that we know what each component does I will explain the whole procedure to launch the drone and the connections between all the software:

- Start Motive and the OptiTrack system.
- From the companion MicroXRCEAgent to interface the flight controller with ROS.
- From the companion launch OptiTrack_interface and px4-py.py to connect the drone with the OptiTrack.
- From the companion launch the controller px4-py.py
- (optionally) launch the gui.py
- set the action through GUI or ROS2 message

For ease of use I created the 2 aliases *dockerdrone* and *dockercommander* to launch everything in one go.

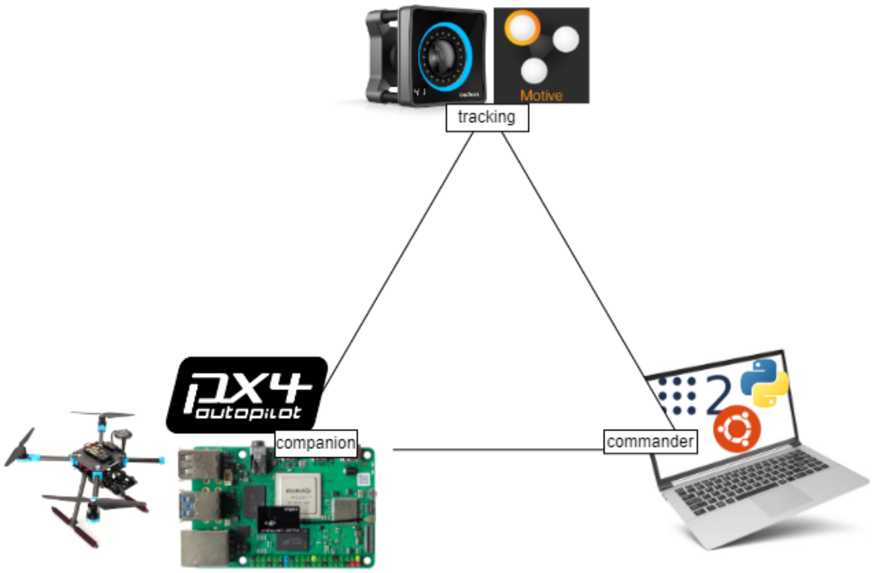

Figure 17: System overview

5.1 Arming

The arming is the process of activating the propellers so the drone can take off, the arming could also be denied by the system if the pre-flight checklist didn't complete or the system has some error; for example a misalignment of the tracking position, missing heartbeat signal from the commander, hardware problems.

5.2 Takeoff and routine

Once the drone is armed a simple setpoint action will trigger the propeller to speed up and lift the drone. The motors on a hardware level are controlled by the ESC (Electronic speed controller) directly wired to the Pix-Hawk flight controller that handles the logic and flight control, stabilization and planning.

All the commands that we send are high level interfaces that are processed by the flight controller; for example if a signal to go forward 50cm comes, an ideal path is produced and the motors are tuned to follow this path, adjusting in real time based on the PID parameters.

Two types of movements are implemented: one for direct position control if the goal is to get to the destination and noting else, and a planned movement; the latter is based on the former but introduces steps in the path, planned delays and a final position adjustment.

In the circling routine, the drone constantly follows a virtual point orbiting around the center.

The *goto* and path routine follow a planned path and the speed can be regulated.

5.3 Landing

The landing uses the vehicle_land_detected message of the drone messages; for the vehicle to be marked as *landed,* a series of states in series:

- Ground contact: no vertical and horizontal movement, lower trust than the idle trust for 0.35 seconds.
- Maybe landed: even lower trust and no rotation for 0.25 seconds
- Landed: vehicle stays in *maybe landed* mode for more than 0,3 seconds.

When landed the drone shuts down and the flight is terminated.

5.5 **Flight data**

The PixHawk flight controller collects data during the flight that can be accessed with the **QGroundControl** application and plotted with **PlotJuggler.**

In this section we will take a look at some visual data from the drone actual position overlapped with the ideal position and some consideration.

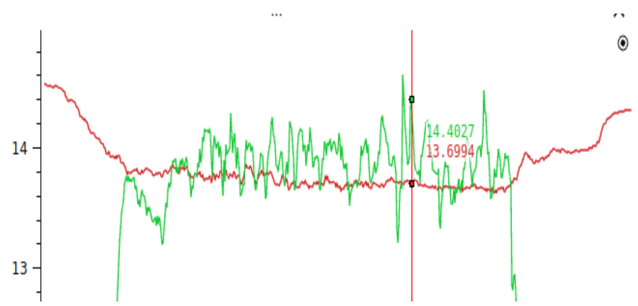

Figure 18: Flight controller power usage

Red: battery voltage [V]

Green: power draw [A]

The power drawn from the system is in the range 190-200 W for the drone and 15 W for the commander.

Considering a safe voltage value to stop at 13V at rest, the value is in line with the 15 minutes flight time assuming a usable range of 80% of the battery.

$$Usable\ power = nominal\ voltage * capacity * usable\ range$$

$$flight\ time = \frac{usable\ power}{power\ drawn} * \frac{3600}{60}$$

The total is around **16,5 minutes.**

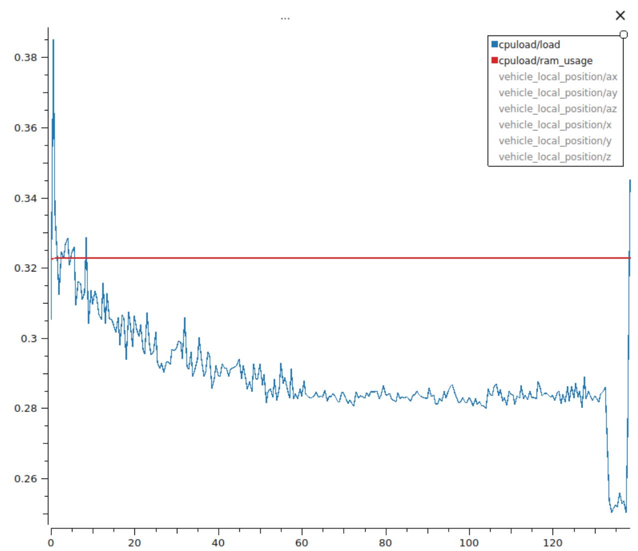

Figure 19: Flight controller CPU and RAM usage

Blue: Ram usage

Red: CPU load

The PixHawk is capable of handling the load with no problem. Idling at 32% CPU utilization and 28% RAM usage.

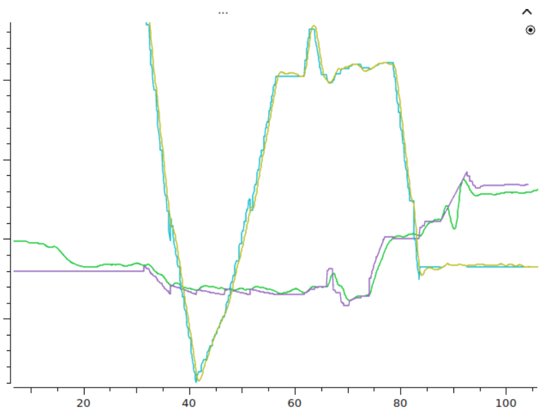

Figure 20: Flight 1, Z position and yaw

Purple: ideal Z position [NED frame]

Yellow: yaw

The drone handles quite well the Z position, the yaw is almost on point.

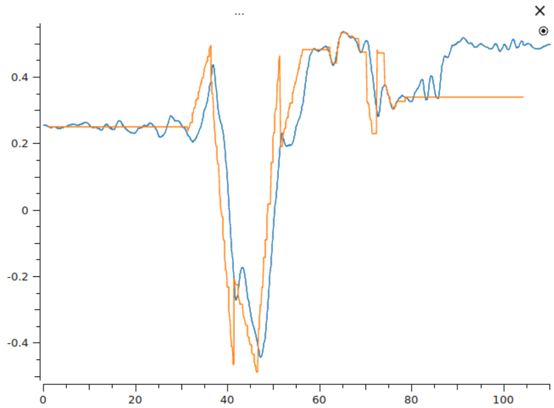

Figure 21: Flight 1, X position

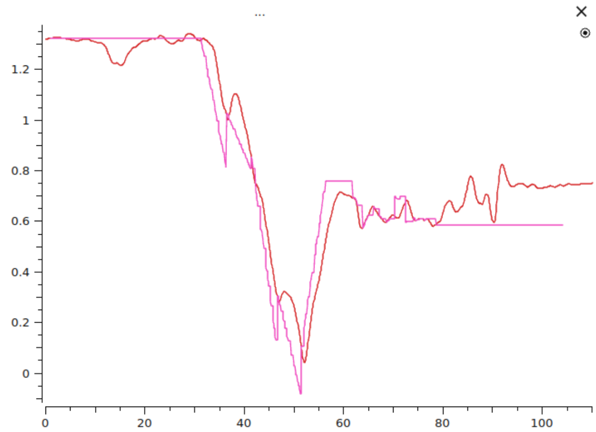

Figure 22: Flight 1, Y position

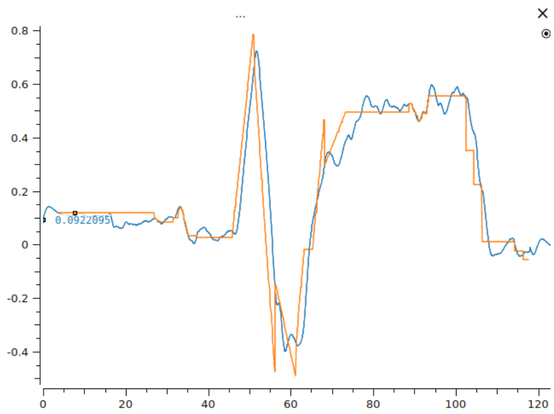

Figure 23: Flight 2, X position

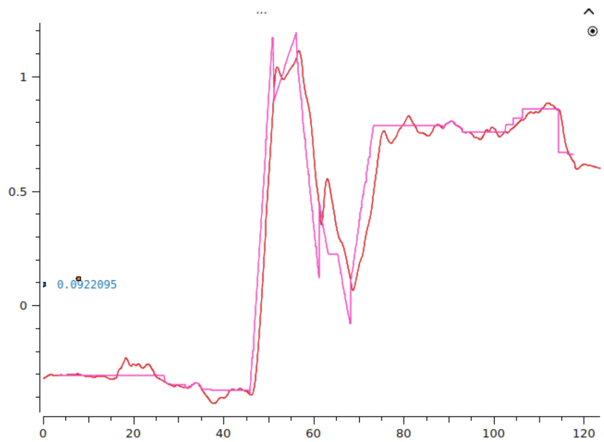

Figure 24: Flight 2, Y position

In the X and Y position there is an obvious delay from when the drone receives the command and from when it starts moving, there is also a small movement when the command changes because the drone has to reposition.

The error in the position of the drone is in the range of **10 centimeters.**

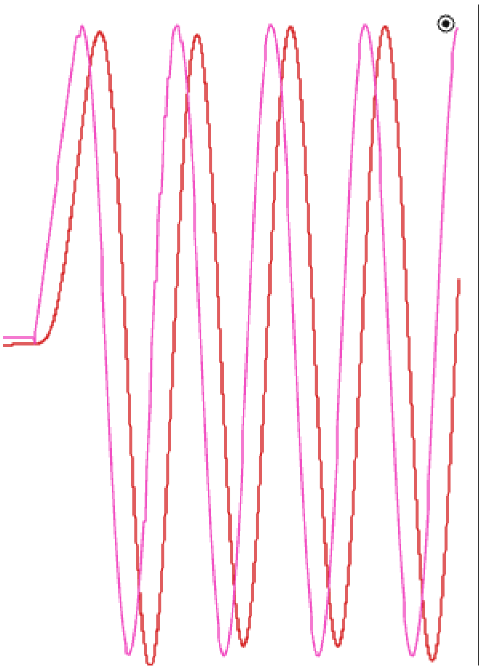

Figure 25: Flight 3, X position

Purple: ideal positioning

Orange: actual position

This flight was a routine following a point circling around in space.

When using harmonious movements the drone performed better

6 **Conclusions and discussion**

The goal of creating a working system to control a flying quadcopter was achieved, and this work can be used as a template for the future and modified at will.

In this section I will go over some of the design choices made and conclusions drawn from this project.

6.1 **Performance**

The main bottleneck of the system was the network connectivity as the laboratory and the university itself produces a lot of WiFi pollution that degrades performance, the network necessary throughput was measured ad around 4Mbps of data stream and the tolerated delay was round 100ms.

Position error was around 10 cm and the drone didn't appear to be unstable; the flight time of 15 minutes is reasonable.

This was never met to be a racing drone and the speed of movement was kept low on purpose, this also helped the stability of the system absorbing delays and avoid disaster.

6.3 **Indoor flight**

This was a tough topic to solve and a lot of the time and testing involved was around the OptiTrack system integration. Many attempt were done using the GPS but the drift was too high and unpredictable, unusable in a closed environment and not stable enough outside to be trusted.

The same technology can be used outside with tracking beacons and other system of positioning that cover a wide area.

6.4 **Challenges**

Most of the challenges came fro the lack of existing similar projects, the existing one provided a similar choices for the controller but the overall system had quite a bit of differences. A big problem was agreeing on a communication type between all programs and ROS was a great help since I just needed to create a converter of messages from almost any system to ROS. In the end I managed to unify many different platforms signals and protocols onto a working system.

I built many custom tools for my specific need, some from scratch, some as a fork of an existing software which helped me to develop a better understanding of software engineering; also a lot of tools that were developed were discarded or replaced with others.

6.5 **Future work and possible implementations**

The drone can be equipped with other instruments and tools like a depth camera to create a 3D map, with a tool like ORB SLAM paired with the inertial measurements of the local position and orientation; or a net-like containers that can be released with a software trigger.

Bibliography

[1] PX4 website - https://px4.io/
[2] OptiTrack - https://optitrack.com/
[3] Holybro X500 v2 Developement Kit - https://holybro.com/products/px4-development-kit-x500-v2
[4] ROCK 4C Plus - https://wiki.radxa.com/Rock4/4cplus
[5] Docker images repository - https://github.com/Massiccio1/px4-22
[6] ROS LTS - https://docs.ros.org/en/rolling/Releases.html
[7] PX4 messages repository - https://github.com/PX4/px4_msgs
[8] Main repository - https://github.com/Massiccio1/px4-py
[9] PX4 Visualizer - https://github.com/Massiccio1/px4-offboard