Summary

Design, development, and testing of an adaptive drone test rig

Drones rely heavily on PID-controllers for their control (position, velocity, attitude, attitude rate). The hyperparameters of the PID-controllers (the PID-gains) depend on the drone's morphology, its mass distribution as well as the ESCs and motors used. For the drone to perform satisfactorily the task at hand, its PID-gains must have been tuned. This hyperparameter tuning is commonly done by hand through educated guesses and on-the-fly visual inspection of the drone's step responses. This way of tuning the gains is highly dependent on the user's ability to visually assess the drone's responsiveness on-the-fly, requires to start from a drone able to fly decently enough and can endanger the user's (as well as the drone's) safety. For this reason, the goal of this project was to develop a tuning platform assisting the user during the tuning process. This platform takes the shape of a tuning software and a test rig, working hand-in-hand.

The tuning software assists the user during the tuning process of a PX4-based drone. It provides a command-line interface (CLI) for the user to easily modify the drone's PID-gains and send step commands to the drone. Additionally, it automatically handles the recording, plotting and analysis of the drone's step response. The analysis of the step response provides advises to the user (through the tuning software's user interface) as to how the current PID-gains need to be changed in order to obtain the desired step response. A framework was developed to allow for future works to implement their own methods of step response analysis. The Ziegler-Nichols method is implemented as a proof of concept, since it is one of the most commonly-used PID tuning heuristics. The majority of the tuning software's development was carried out using a Software-In-The-Loop (SITL) approach, making use of the Gazebo simulator. The software was then tested and further improved by using it with a real drone instead of a simulated one. In parallel to the tuning software's development, a test rig was developed. The drone is mounted on the rig during the tuning process, constraining it in position but not in orientation. This ensures the safety of both the user and the drone as well as allows for unstable PID-gains to be assessed.

Our tuning platform was tested in a Motion Capture Tracking Hall on a real quadcopter, yielding PID-gains allowing for stable flight. Additionally, we were able to corroborate our hypothesis that flight on the rig and in unconstrained flight presented an equivalent response. The automatic analysis of the step response yielded PID-gains allowing for stable flight while requiring few user inputs. Furthermore, the automatic plotting of the step response allowed for a more educated evaluation of the step response, and hence a more educated hand-tuning.

January 19, 2020

DESIGN, DEVELOPMENT, AND TESTING OF AN ADAPTIVE DRONE TEST RIG

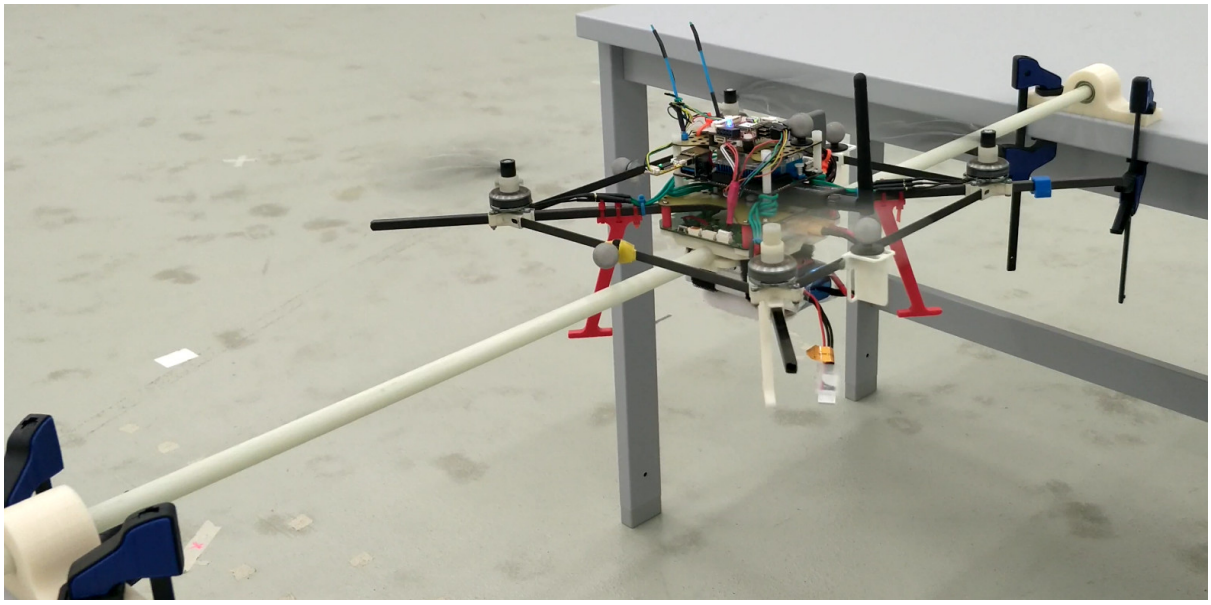
—

SEMESTER PROJECT (10 ECTS)

—

REPORT

January 19, 2020



Author:

Arthur Gassner

Supervisors:

Fabrizio Schiano

Przemyslaw Kornatowski

Anand Bhaskaran

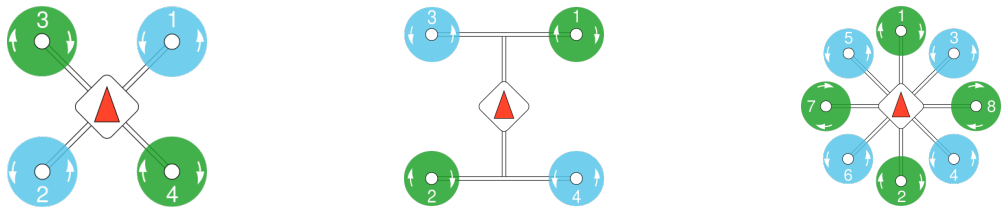
Professor:

Dario Floreano

Contents

1 INTRODUCTION

Unmanned aerial vehicles (UAVs) benefit from a wide range of possible applications, ranging from supply delivery [1] to agricultural monitoring [2], passing by surveillance [3] and safety inspection [4]. All these applications have in common that the drone's pose is required to be controlled well enough for the drone to reliably perform the task at hand. A common controller used for this purpose is a PID-controller, which has proved itself to work well in practice [5]. However, the use of a PID-controller requires the setting of hyperparameters (the PID-gains), which depend on the drone's ESCs, motors as well as the drone's inertia [6]. For most common air-frames, empirically found PID-gains are available (see Fig. 1). Those gains provide a good starting point from which the drone is able to take off and fly decently enough. However, further tuning of those gains allows the PID-controller to be more adapted to the drone's dynamics, and hence to improve the drone's responsiveness. Additionally, in cases where the drone's air-frame is too uncommon for empirical PID-gains to be available, the process of finding those gains needs to be started from scratch.



(a) X-shaped quadcopter **(b)** H-shaped quadcopter **(c)** Plus-shaped octocopter

Figure 1: Examples of multicopter air-frames for which PX4 – an open-source autopilot – provides empirical PID-gains

Currently, this parameter tuning is done by hand through educated guesses and on-the-fly visual inspection of the way the drone behaves when flying. This method lacks repeatability and is entirely dependent on the user's ability to visually assess the drone's responsiveness. Additionally, it is crucial that the drone is already able to take off and fly, as well as has enough space to do so (see Fig. 2). Another concern is the user's safety and drone's integrity. Indeed, this tuning process relies entirely on trying hyperparameters, which can make the drone become unstable, crash and hurt the user.

Figure 2: Evolution of an unconstrained quadcopter during the tuning of one of his PID-controllers

The goal of this project is to develop a platform helping the user throughout the tuning process of a multicopter. This platform takes the shape of a tuning software used in conjunction with a physical rig. The tuning software provides an interface for the user to interact with the drone and handles the analysis of the drone's behavior. The rig ensures that the drone is constrained in position without impairing its orientation, allowing for a safe assessment of the drone's responses.

2 STATE OF THE ART

To the best of our knowledge, no tuning software assisting the user in the tuning process is available. One work mentions the development of a GUI helping them assess the step responses [7], but the software was not made public.

Concerning the rig, the building of a test rig allowing for a safe and indoor recording of a drone's orientation has been already investigated in the literature. Oftentimes, the rig is only used within the context of a specific paper.

The main concern when designing a rig is to ensure the ability to reproduce in unconstrained flight the results obtained on the rig. In order to do that, the drone's dynamics on the rig must be as close as possible to the drone's dynamics when flying unconstrained. A mismatch between those two dynamics would result in a drone behaving differently depending on whether it is on the rig or flying unconstrained. Such a mismatch can come from two causes: added rotational inertia and offset between the center of rotation and the center of gravity.

- **Added rotational inertia**: When mounted on the rig, the drone is physically bounded to part of it. This bounded part adds weight to the drone and changes its rotational inertia, effectively changing the drone's dynamics.
- **Offset between the center of rotation and the center of gravity**: In unconstrained flight, the drone's center of rotation coincides with its center of gravity. When mounted on a rig, this is often not strictly the case and the drone's center of rotation is offset from its center of gravity, effectively changing the drone's dynamics.

An additional criteria for us is adaptability. The rig must be able to easily adapt to uncommon air-frames (either in shape or in size). The different types of rigs are summarized in Tab. 1 and further inspected below.

Table 1: Overview of the different types of rigs and their characteristics

Type of rig	DOF	Adding rotational inertia	Offset in the center of rotation	Adaptable
Gyroscopic	3	Yes, gimbals	Yes, but can be alleviated	No
Ball-joint & Universal-joint	3-4	Yes, but very little	Yes	Yes
Rod-based	1	Yes, but very little	Yes, but can be alleviated	Depends on the implementation

2.1 Gyroscopic solutions

A recent work shows the design of a 3DOF gyroscopic rig, allowing for full revolutions in roll, pitch and yaw [8]. It is however important to note that, due to the rig's design, not all 3DOF are controllable at all time. Indeed, the yaw is implement by mounting the drone directly on a bearing (see Fig. 3b). This allows the drone to yaw freely regardless of its roll/pitch angle. However, for the rig to physically allow the drone to roll/pitch, the drone's yaw angle must be as it is in Fig. 3a.

The vertical height of the drone can be adjusted when mounting it onto the rig, and thus the drone's center of gravity can coincide with the rig's axis of rotation. This eliminates the concern of an offset between the center of rotation and the center of gravity by allowing the user to manually tune this offset.

Additionally, the added rotational inertia in yaw and pitch (or roll, depending on the drone's orientation) is kept minimal. Indeed, the yaw rotation is enabled by mounting the drone on a bearing, while the pitch (or roll) is enabled by the rotation of a bar-like structure (see Fig. 3b) around its longitudinal axis, ensuring a low impact on the drone's rotational inertia. The added rotational inertia in roll (or pitch, depending on the drone's orientation) is however potentially greatly increased by the mass of the outer ring (which is made from low-density wood).

(a) 3DOF gyroscopic rig built out of low-density wood

(b) Exploded view of the 3DOF gyroscopic rig

Figure 3: Gyroscopic rig from [8]

A similar 3DOF gyroscopic rig has been commercialized[1] [9], allowing for 3 controllable DOF (roll, pitch and yaw) (see Fig. 4). Again, the rig allows for manual tuning of the drone's

[1] The rig in action can be seen here: `https://www.youtube.com/watch?v=wAxHFTzAP-U`

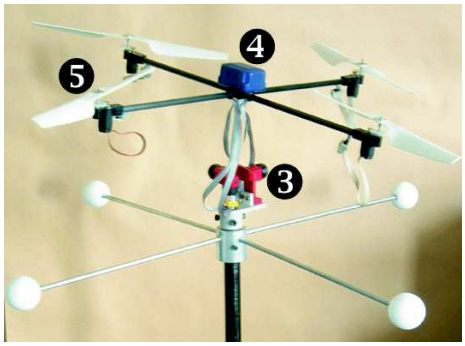
vertical height, eliminating the concern of an offset between center of gravity and center of rotation. The two outer rings and the metal stand however increases the rotational inertia in roll, pitch and yaw. This increase is mitigated by building the rings out of carbon fiber.

Figure 4: 3DOF gyroscopic rig from [9]

However, both gyroscopic test rigs suffer from adaptability issues when the UAV grows in size or drastically changes shape: as soon as the drone does not fit within the gimbals, an entirely new rig has to be built.

2.2 Universal-joint and ball-joint solutions

The use of a 3DOF universal-joint fixed under the drone's frame allows for a more compact design. Such designs have been used to assess stabilization strategies [10] and identify inertia moments [11] (see Fig. 5).

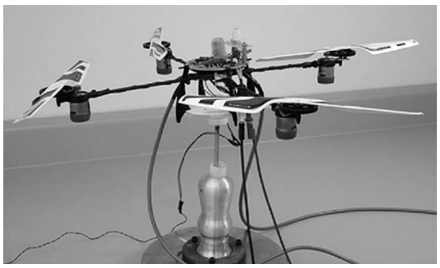


(a) 3DOF universal-joint (3) solution [10]

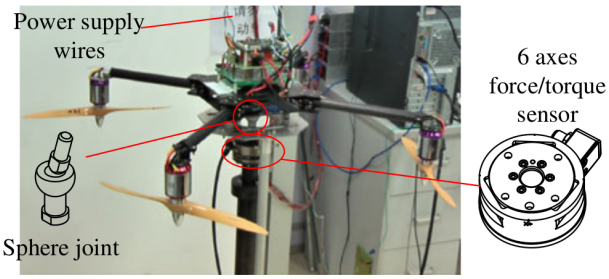
(b) 3DOF universal-joint solution from [11]

Figure 5: Two different rigs implementing the 3DOF universal-joint solution

Similar solutions using a ball-joint instead of a universal-joint have been developed [12, 13, 14, 15, 7] (see Fig. 6).

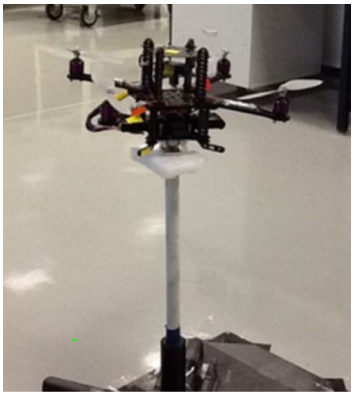


(a) 3DOF ball-joint solution from [12]



(b) 3DOF ball-joint solution from [13]

(c) 4DOF ball-joint solution from [14], allowing for roll, pitch and yaw rotations, as well as heave through linear bearings built-in the bar



(d) 3DOF ball-joint solution from [15]

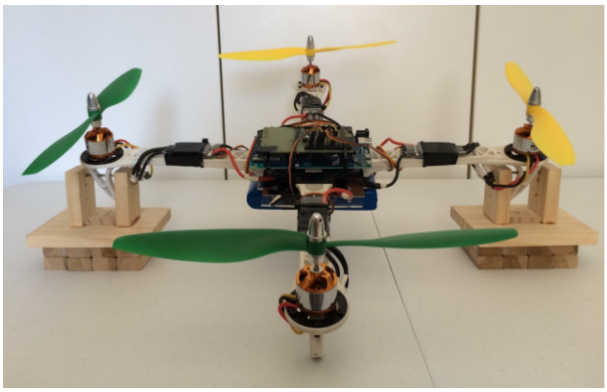
(e) 3DOF ball-joint solution from [7]

Figure 6: Several rigs implementing the ball-joint solution

Both the universal-joint and ball-joint solutions have the advantage of adding very little rotational inertia to the drone. Additionally, they benefit from great adaptability capabilities: any drone of any shape can be mounted on such a rig, provided that the rig can sustain the drone's mass. The main drawback comes from the introduction of an offset between the center of rotation and the center of gravity. Depending on the drone's structure, this offset could prove itself too big to be negligible. This is especially a concern given the battery placement of many drones (i.e. under the center of gravity), which would result in the ball-joint (or universal-joint) to have to be placed below it.

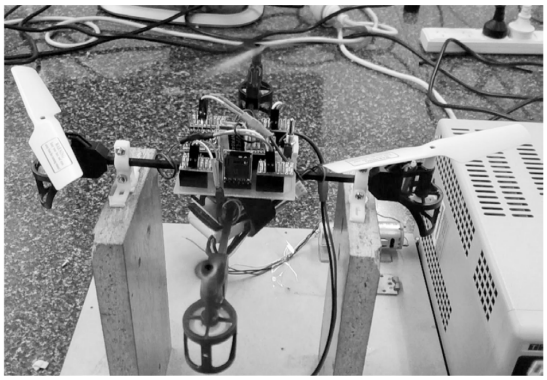
2.3 Rod-based solutions

The use of a 1DOF rod-based solution allows for attitude recording roll/pitch [11, 16, 17] (see Fig. 7).



(a) 1DOF rod-based solution from [11]

(b) 1DOF rod-based solution from [16]



(c) 1DOF rod-based solution from [17], using the drone's arms as the rod

Figure 7: Two rigs implementing the rod-based solution

While such a rig is able to accommodate both roll and pitch rotations (depending on how the drone is mounted, and hence not at the same time), drones mounted on it cannot perform rotations in yaw. However, the rig's impact on the drone's rotational inertia is minimal. Indeed, while mounted on the rig, the only rotations performed by the drone will be around the rod's longitudinal axis. The rotational inertia added by a rod around this axis is very small. Another aspect is the offset between the center of rotation and center of gravity, which depends on the rod placement. The rod can be made to go between the battery and the rest of the drone, nearing it from the drone's center of gravity. Another solution is to implement as solution similar to Fig. 4, where the drone's vertical height can be adjusted so that the center of rotation on the rig matches the drone's center of gravity.

Additionally, this solution can be somewhat adaptable, depending on how it is implemented. Indeed, a rod-based solution using a unique metal stand (e.g. in Fig. 7b) would need to change both the rod and the metal stand were the drone to become too large to fit on the rig. On the other hand, a rod-based solution using two separate platforms to support the rod (e.g. in Fig. 7c) would only need to spread apart the platforms and get a longer rod.

3 SIMULATION ENVIRONMENT

In robotics research, a simulator is often used to test ideas, reducing the cost and the time needed to experiment with new solutions. This allows for developing a solution that works in simulation before moving onto hardware. Such an approach is called Software-In-The-Loop (SITL), where the hardware is simulated through software. This is the approach taken in this project.

3.1 Gazebo simulator

Gazebo is an open-source simulator oriented towards robotics. One can load the model of a robot and render its interactions with the world. The dynamics of those interactions are computed through one of the physics engines supported by Gazebo (*ODE*, *Bullet*, *Simbody* and *DART*). The robot's sensors are simulated, as well as their noise. The whole simulation is rendered through *OGRE*, an open-source graphics rendering engine [18].

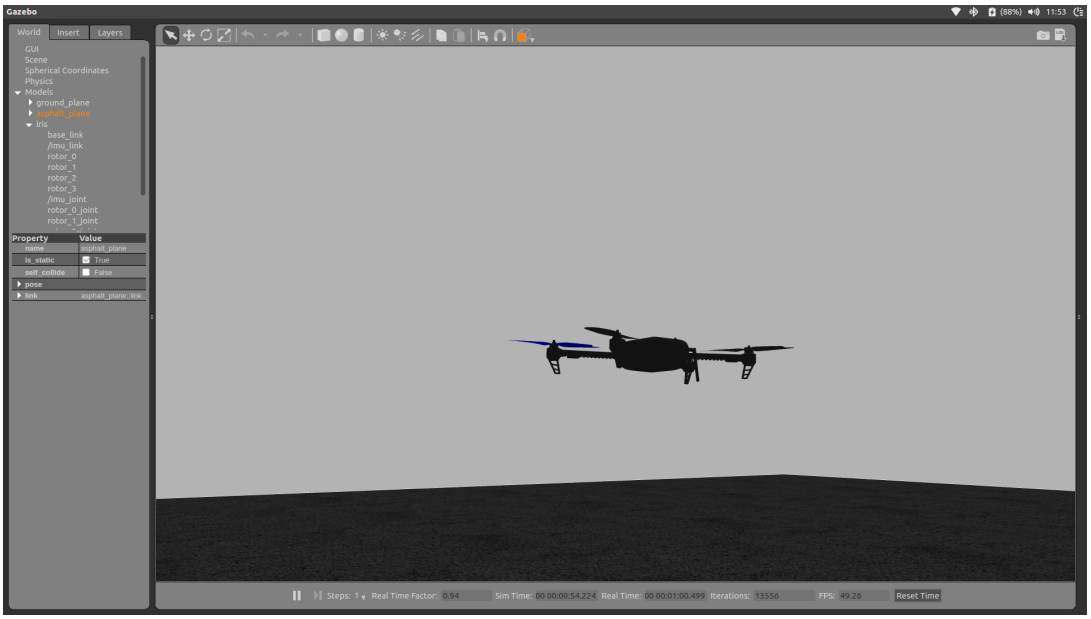


Figure 8: Example of a Gazebo simulation: a drone flying

3.2 Robotic Operating System (ROS)

ROS is an open-source operating system for robots [19]. It acts as a communication layer between different compute clusters. The reason for its birth can be seen in [20]: "*Writing software for robots is difficult, particularly as the scale and scope of robotics continues to grow. Different types of robots can have wildly varying hardware, making code reuse nontrivial. On top of this, the sheer size of the required code can be daunting, as it must contain a deep stack*

starting from driver-level software and continuing up through perception, abstract reasoning, and beyond. [...] *robotics software architectures must also support large-scale software integration efforts.*"

ROS handles those large-scale integration efforts by acting as a middle-ware and providing a common platform on which robotics researchers can develop software for their robots of interest. At its core, ROS is multi-lingual [20], allowing us to choose from a wide range of programming languages (including C++, Python and Lisp).

3.3 PX4

PX4 is an open-source autopilot flight stack. It allows for the control of many different vehicle frames/types, including air-crafts (multi-copter and fixed-wings), ground vehicles and underwater vehicles [21]. It can run on and interface with a wide choice of hardware.

PX4 is a core part of a broader ecosystem of drone-related software/hardware, amongst which the *Pixhawk* (a flight control unit (FCU)) and the *MAVSDK* library (allowing for the integration of hardware (e.g. companion computers, sensors) using the MAVLink protocol). The MAVLink protocol is a lightweight messaging protocol used for communicating with drones, as well as between on-board drone components.

Figure 9: The Flight Control Unit (FCU) called *Pixhawk 4*

Since this project relates to drones, the focus is given to the multicopter side of the PX4 software.

PX4 supports several flight modes[2], 3 of which are of interest for us:

- **STABILIZED**: The autopilot controls the attitude, meaning it regulates the roll and pitch angles to zero when the RC sticks are centered, consequently leveling-out the

[2]The list and description of all flight modes can be accessed here: https://docs.px4.io/v1.9.0/en/flight_modes/

attitude. However, in this mode the position of the vehicle is not controlled by the autopilot, hence the position can drift due to wind.

- **POSITION**: The autopilot controls the drone's attitude, meaning it regulates the roll and pitch angles to zero when the RC sticks are centered, consequently leveling-out the attitude. Additionally, in this mode the position of the vehicle is controlled by the autopilot, hence the position is held against wind.
- **OFFBOARD**: Vehicle obeys a position, velocity or attitude setpoint provided over MAVLink (often from a companion computer connected via serial cable or wifi).

3.4 MAVLink and MAVROS

As said above, PX4 supports the MAVLink communication protocol, allowing it to communicate with external hardware (e.g. companion computers, on-board sensors). MAVROS is a ROS package providing an API for communicating with autopilots supporting the MAVLink protocol. It acts as a bridge between ROS nodes and the PX4 autopilot.

3.5 Overview of the SITL

Now that the components of the simulation environment have been established, it is important to understand how they fit together.

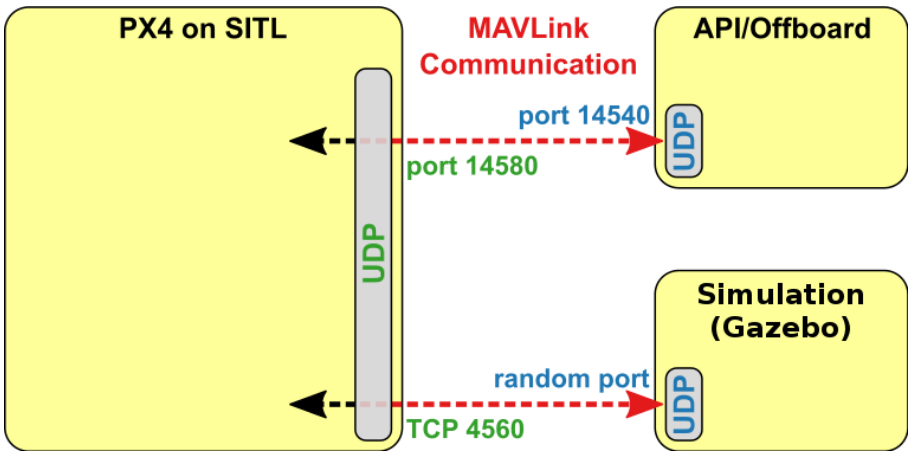


Figure 10: Overview of the SITL

As can be seen in Fig. 10, the ROS nodes running on the offboard computer communicate with the PX4. At the same time, the PX4 is communicating with the Gazebo instance running the simulation. Both communication channels are established through the through the MAVLink protocol. This approach allowed us to develop a software solution that worked on simulation before moving onto hardware.

4 REAL-WORLD ENVIRONMENT

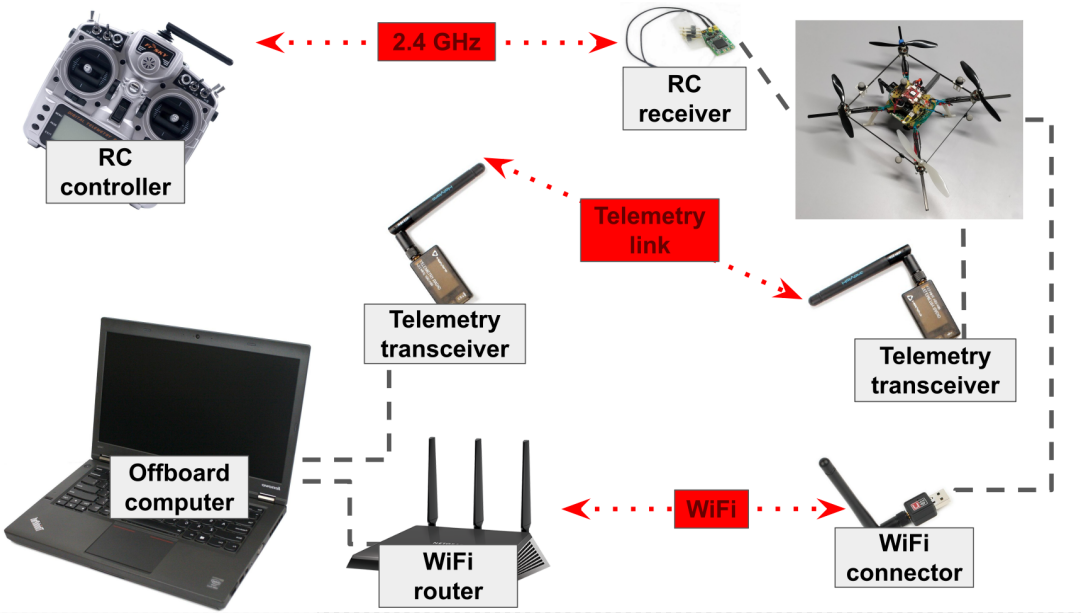


Figure 11: Overview of the drone's wireless connectivity

4.1 QGroundControl

QGroundControl is a ground control station software part of the same ecosystem as PX4. It is run on the offboard computer to monitor the drone's status, perform sensor calibrations and modify PX4 parameters on-the-fly.

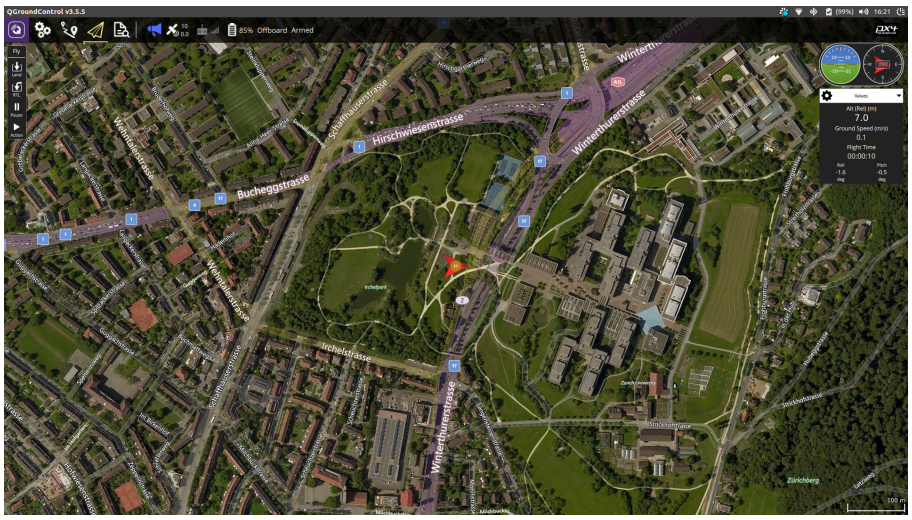


Figure 12: Overview of the QGroundControl interface

4.2 Drone overview

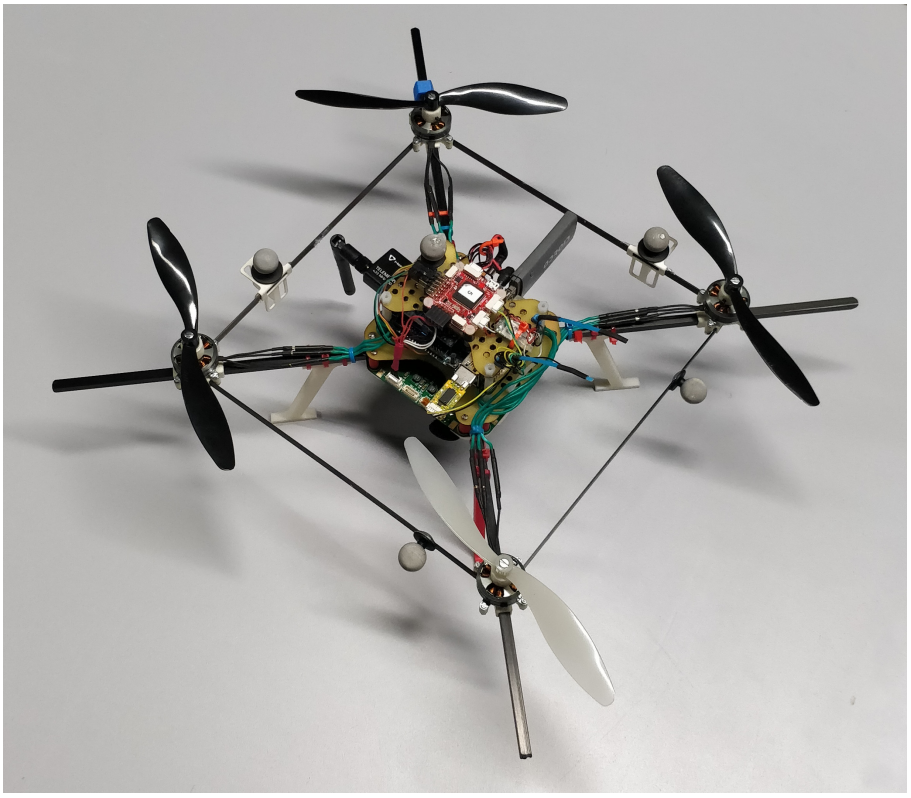


Figure 13: Overview of the drone used in this project

The drone used in this project is a X-shaped quadcopter (see Fig. 13), whose onboard components are summarized in Tab. 2. An overview of how those components are wired can be seen in Fig. 14.

Table 2: Hardware used in this project's drone

Flight Control Unit (FCU)	PixRacer R14
Companion computer	ODROID-XU4
Telemetry radio	Holybro Transceiver Telemetry Radio V3 433MHz
ESC	Turnigy MultiStar 4 in 1 20A Race Spec ESC 2 4S
Rotors	AS2204 KV1700

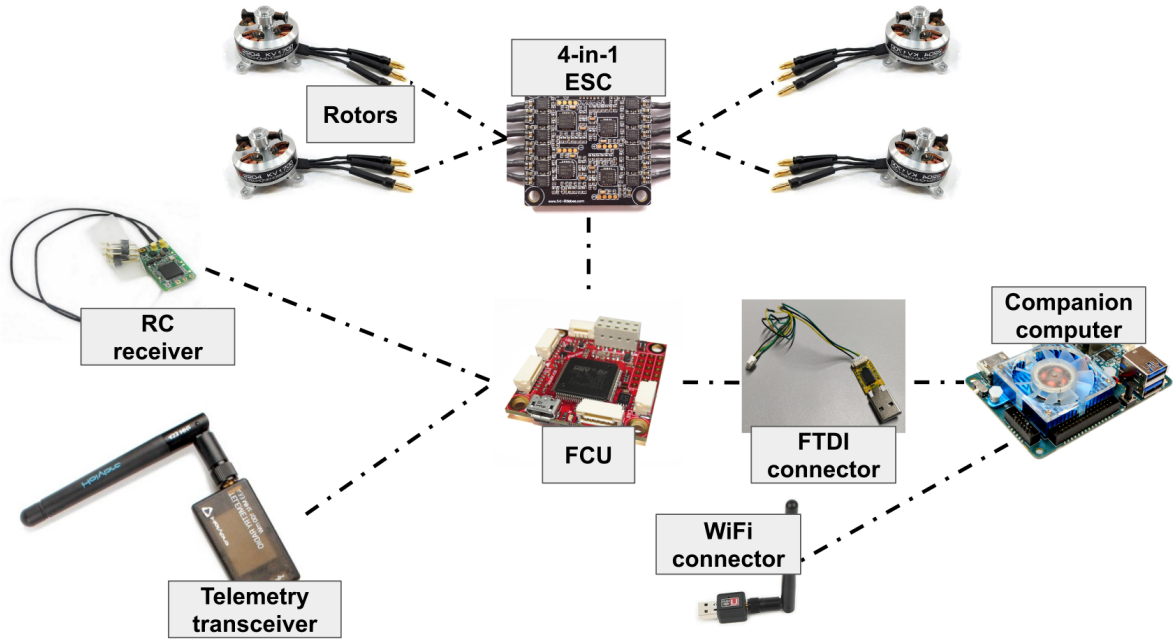


Figure 14: Overview of the wiring of the drone's onboard components

In order to maintain full control of the drone, a RC controller (FrSky TARANIS X9D) was used at all time (see Fig. 15).

Figure 15: The RC controller used in this project: FrSky TARANIS X9D

4.2.1 Companion computer

The FCU allows for the basic control of the drone's flight. For anything more computationally intense, a companion computer is required. A companion computer takes the form of a small computer mounted on the drone and connected to the FCU.

In our case, the companion computer is an ODROID-XU4 running Ubuntu 16.04. The connection to the FCU is established through a FTDI connection (see Fig. 16). The companion computer is used to run MAVROS and allows for the communication between the drone and the offboard computer.

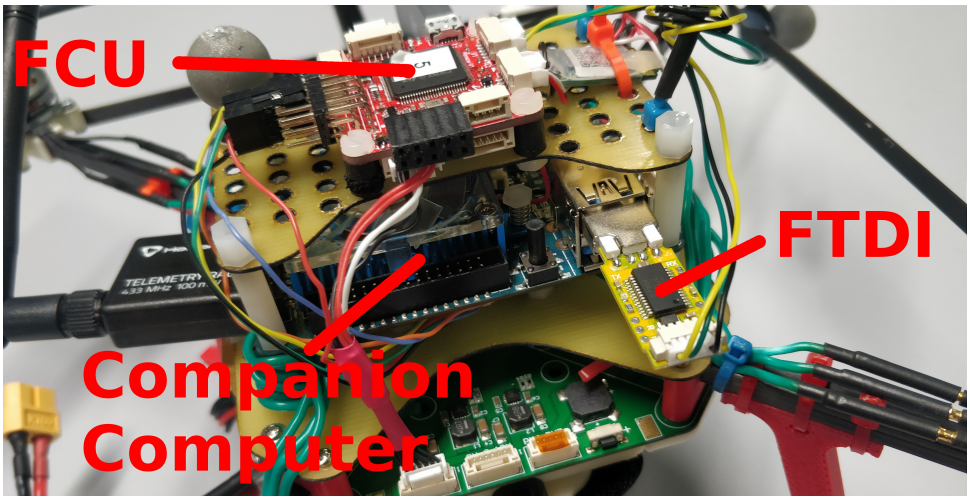


Figure 16: Companion computer mounted on the drone

4.3 DroneDome

The DroneDome is a 10x11x5m room located in the basement of the MED building[3]. It is equipped with a Motion Capture System (OptiTrack).

Figure 17: Overview of the DroneDome

The room consists of a metal frame mounted with IR cameras (see Fig. 17). The absolute pose of an object can then be tracked in real-time (see Fig. 19) by mounting IR markers on it (see Fig. 18). This tracking is done through the Motive software.

[3]EPFL, 1015 Lausanne, Switzerland

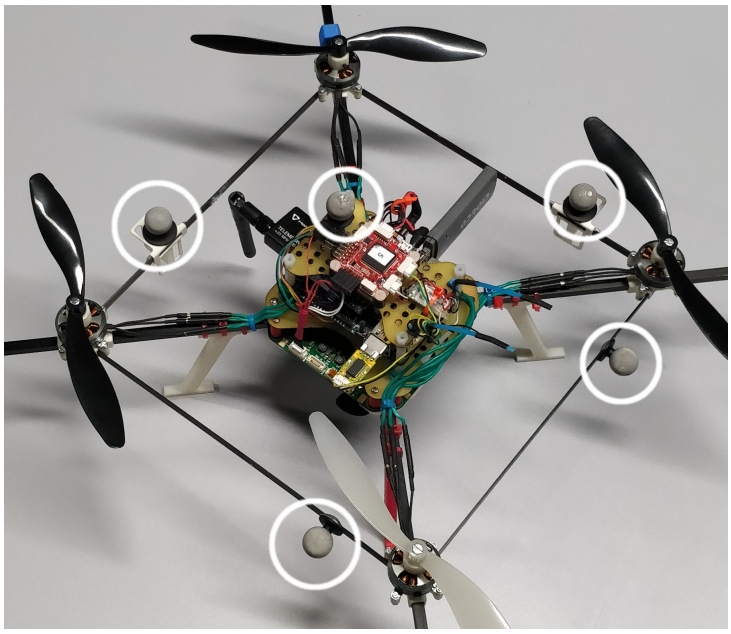


Figure 18: Drone equipped with IR markers

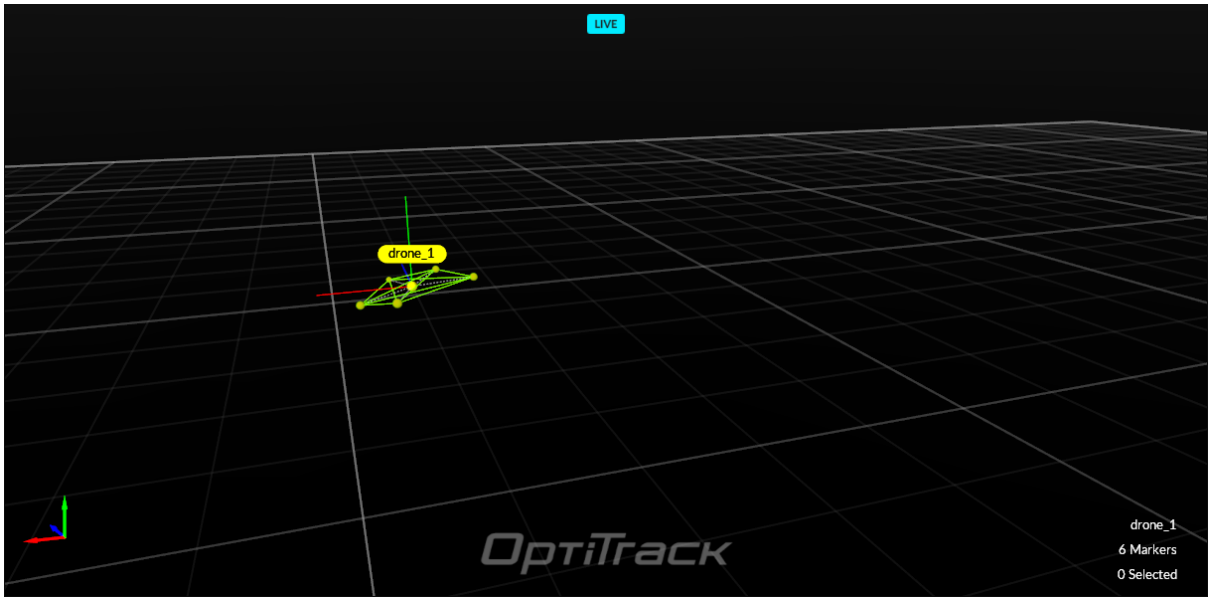


Figure 19: IR markers as seen by the Motive software

4.3.1 How to use the DroneDome to publish the drone's absolute position on a ROS topic?

The drone's absolute pose can be published on a ROS topic. The steps necessary to do so through the OptiTrack setup are listed below.

- **Place IR markers** on the drone's frame (see Fig. 18).
- **Physically place the drone** in the center of the room. It is important to make sure that the drone's forward direction (indicated by a little arrow on the FCU) coincides with

the OptiTrack's X-axis (visible on the Motive software).

- On the Motive software, go on the Data Streaming Pane (`View > Data Streaming Pane`) and make sure that:
  - `Broadcast Frame Data`: **On**
  - `Rigid Bodies`: **On**
  - `Up-Axis`: **Z Up**
  - `Broadcast VRPN Data`: **On**
- On the Motive software, **create a rigid body**. Do so by selecting the drone's IR markers, right-clicking and selecting `Rigid Body > Create From Selected Markers` (see Fig. 20

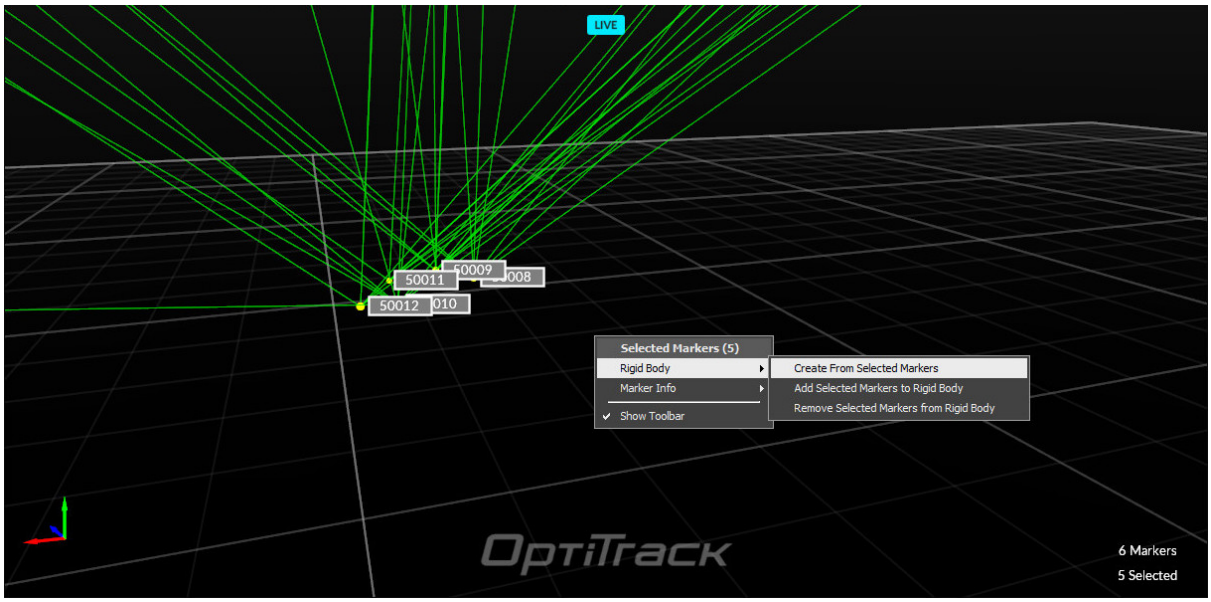


Figure 20: Selecting the drone's IR markers and creating a rigid body

- **Rename the newly created rigid body** into `drone_1`. Do so by going in the Assets Pane (`View > Assets Pane`).
- The drone's absolute pose is now published on the ROS topic called `/vrpn_client_node/drone_1/pose`.

5 THE TUNING SOFTWARE

PX4 uses cascaded controllers to control the drone's position [22], where the inner-most loop (the attitude rate controller) is a PID-controller and the above loop (the attitude controller) is a P-controller [6, 23] (see Fig. 21).

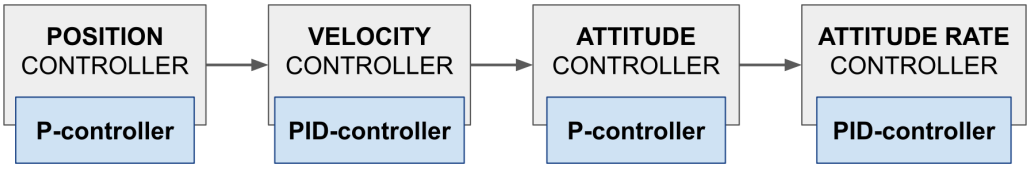


Figure 21: Cascaded controller used by PX4

Each Euler angle has its own cascaded controller. Hence, in order to be able to control the drone's position satisfactorily, each inner-most controller (i.e. drone's attitude rate controller) must be well tuned to begin with.

The PID gains of each of those 3 attitude rate controllers (roll-rate, pitch-rate and yaw-rate), as well as the P gain of each of those 3 attitude controllers correspond to a PX4 parameter (see Tab. 3 and Tab. 4).

Table 3: PX4 parameter corresponding to the PID gains of each attitude rate controller

	Roll-rate controller	**Pitch-rate controller**	**Yaw-rate controller**
P gain	MC_ROLLRATE_P	MC_PITCHRATE_P	MC_YAWRATE_P
I gain	MC_ROLLRATE_I	MC_PITCHRATE_I	MC_YAWRATE_I
D gain	MC_ROLLRATE_D	MC_PITCHRATE_D	MC_YAWRATE_D

Table 4: PX4 parameter corresponding to the P gain of each attitude controller

	Roll controller	**Pitch controller**	**Yaw controller**
P gain	MC_ROLL_P	MC_PITCH_P	MC_YAW_P

The purpose of the tuning software is to assist the user in the tuning process. This assistance takes the shape of:

- An **easy-to-handle interface**, allowing the user to **send attitude step commands** to the drone and to **change PID gains**
- **Automated plotting** of the drone's step response
- Computing advises from the step response, in order to **advise the user** on what to do next

5.1 Architecture

The tuning software takes the shape of ROS nodes, each written in Python. Each node runs in parallel and communicates with the other nodes through ROS services and ROS messages (see Fig. 22 and Fig. 23). Each node handles certain responsibilities. This modularity was used to allow for greater maintainability as well as easier debugging. The greater maintainability makes it easy to add, change or improve upon the software's functionalities.

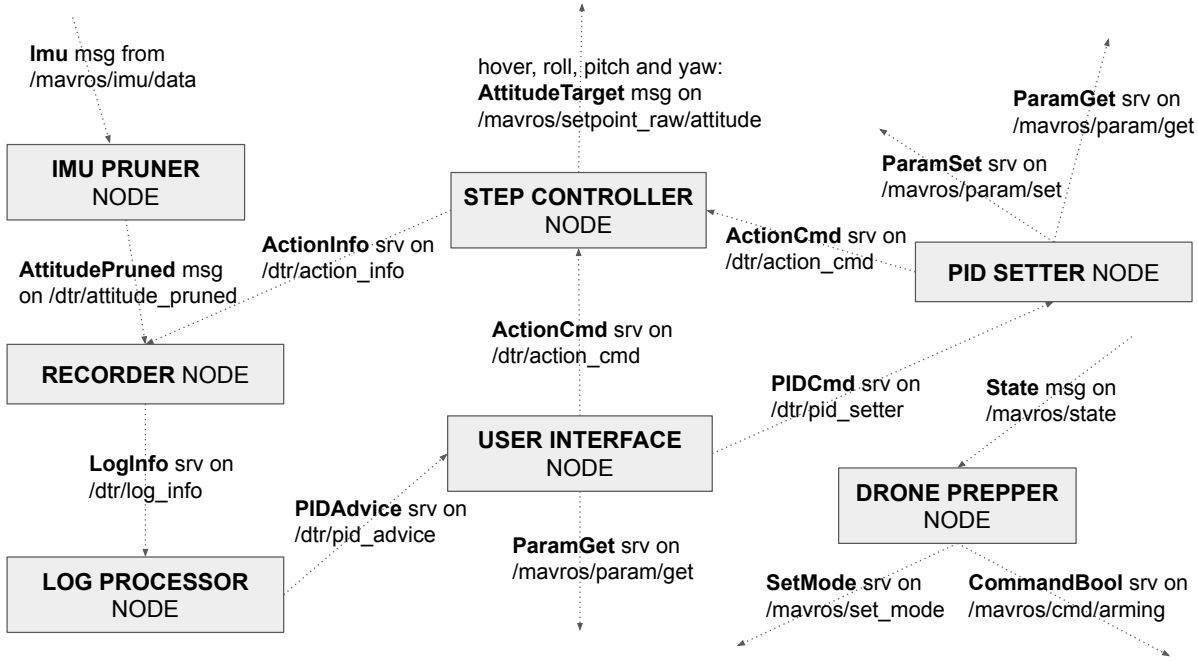


Figure 22: Overview of all the nodes and how they interact

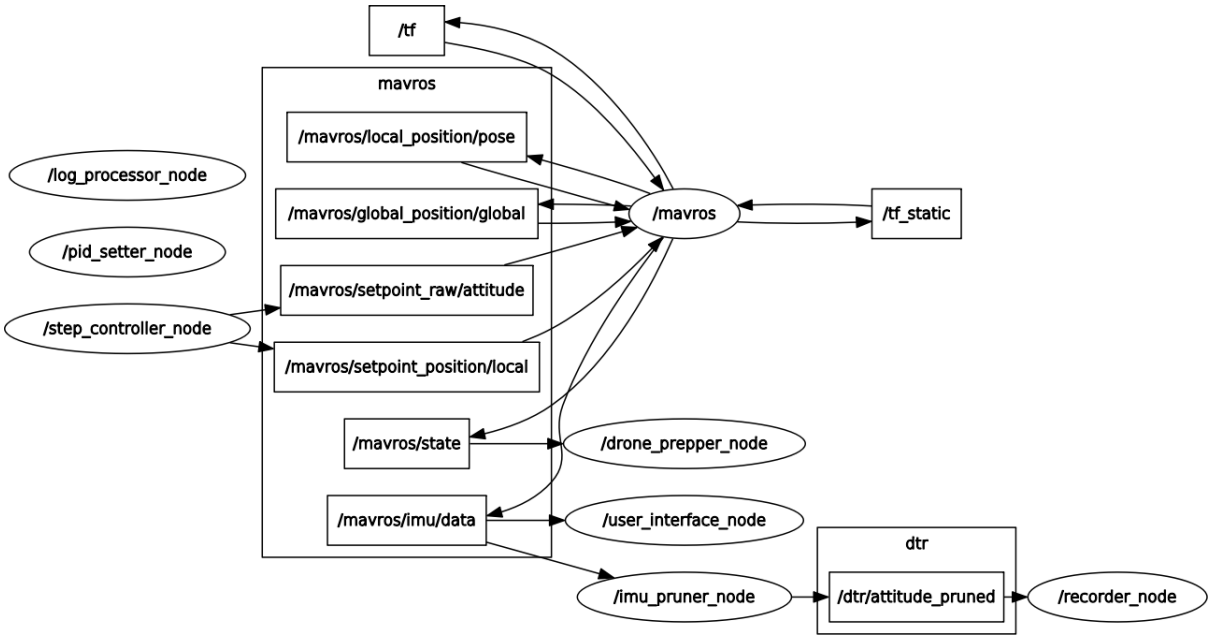


Figure 23: rqt graph of the tuning software

5.1.1 ROS nodes

- **User Interface node**: The User Interface node is the **only node the user interacts with**. It is a CLI and displays the current PID gains, information about the action being currently performed (e.g. "Rolling...") as well as advises as to what the next PID gains should be. Depending on the user's input, it can communicate with the PID Setter node or the Step Controller node.

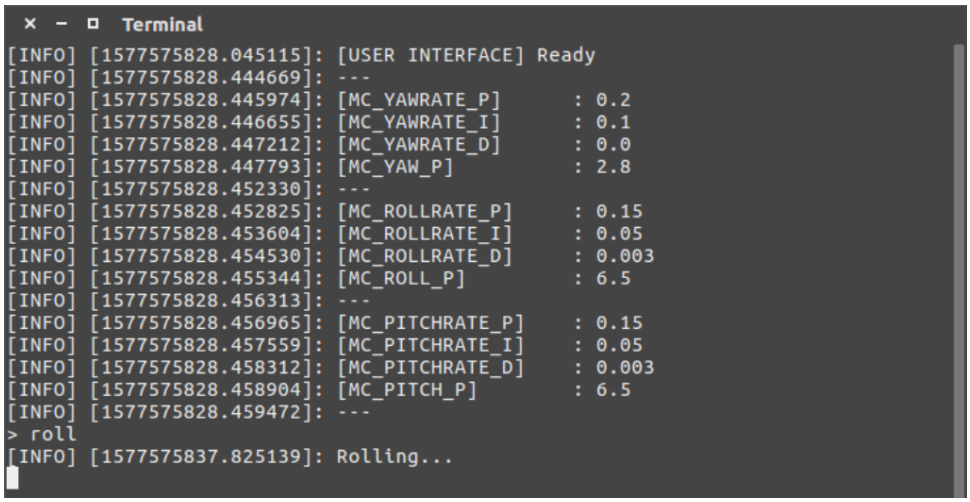


Figure 24: User Interface being used to send a step command in roll

- **PID Setter node**: The PID Setter node is **in charge of modifying the drone's PID gains**. It does so when requested by the User Interface node through a `PIDCmd` service. Once the PID Setter has changed a PID gain on the drone, an `ActionCmd` is sent to the Step Controller node, instructing it to perform the action relating to the previously changed PID gain (e.g. if the P gain of the roll rate has just been changed, the PID Setter node instructs the Step Controller to send a roll command).

- **Step Controller node**: The Step Controller node is **responsible for sending attitude commands** to the drone, be it hover, pitch, roll or yaw. It sends a hover command by default, until told otherwise through an `ActionCmd` service (sent by either the PID Setter node or the User Interface node). Once the Step Controller node instructs the drone to perform an action other than "hovering", it notifies the Recorder node by sending an `ActionInfo` service. Once the Step Controller node stops performing an action other than "hovering" (e.g. "rolling"), it notifies the Recorder node by sending an `ActionInfo` service.

- **Recorder node**: The Recorder node is **in charge of recording the drone's attitude**, expressed as a quaternion, into a log file. It starts (and stops) recording as soon as notified by the Step Controller through an `ActionInfo` service. It fetches the drone's

attitude by listening to the `ActionInfo` messages published by the IMU Pruner node. The frequency at which this attitude is logged can be specified by the user.

- **IMU Pruner node**: The IMU Pruner node is **in charge of fetching the IMU data** of the drone, **pruning it** (to only keep the attitude data) and **publishing it** as `AttitudePruned` messages onto a topic. This is done continuously, independently of what other nodes are doing. It is important to note that this IMU data could be the result of sensor fusion, with e.g. a motion capture system.

- **Log Processor node**: The Log Processor node is **in charge of processing the log file** produced by the Recorder node. Once the Recorder node is done recording the drone's step response, it notifies the Log Processor node through a `LogInfo` service that a log file is ready to be processed. The Log Processor node then reads the log file, plots it (saving the plot into a user-specified folder) and calls a `PIDAdvisor` to analyse it (see Section 5.2 for further explanation of the `PIDAdvisor`). The `PIDAdvisor` then produces an advice as to what to do next (e.g. set the P gain to 1.3). The Log Processor node sends said advice to the User Interface through a `PIDAdvice` service.

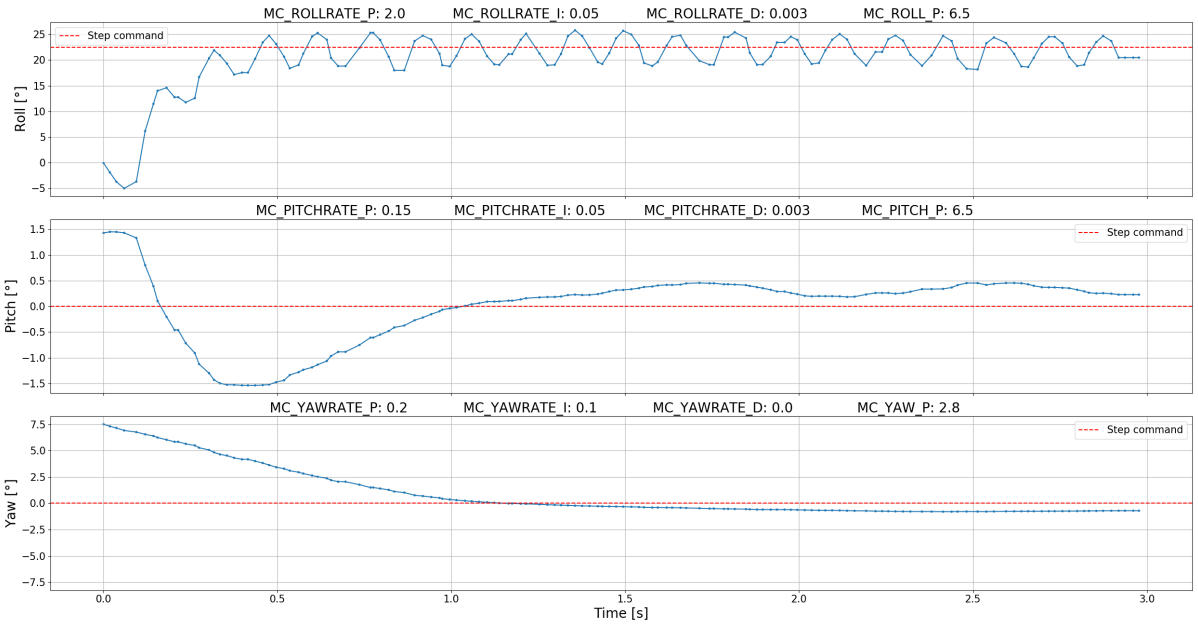


Figure 25: Roll step response as automatically plotted by the Log Processor node. The pitch and yaw are plotted for completeness.

- **Drone Prepper node**: The Drone Prepper node is **in charge of prepping the drone for the tuning software**. In practice, this means arming it and switching its flight mode to OFFBOARD mode. This node does not communicate with any of the other nodes and is only useful in simulation. Indeed, when using a real multi-copter, the human user is in charge of both arming the multi-copter and switching it to OFFBOARD mode.

5.1.2 ROS services and ROS messages

Since the software has been built by taking a modular approach, it is important to have a reliable way for nodes to communicate. This communication is established through custom ROS services and messages (see Fig. 22). The content and purpose of each custom ROS service/message is outlined below.

- **ActionCmd service**

Listing 1: `ActionCmd` service

```
1 string action_id
2 ---
3 bool success
```

The `ActionCmd` service handles the sending of action commands to the Step Controller node (see Fig. 22). The action to be performed is specified as a string in the `action_id` field. The possible actions are: `hover`, `roll`, `pitch` and `yaw`.

- **PIDCmd service**

Listing 2: `PIDCmd` service

```
1  float64 MC_ROLLRATE_P
2  float64 MC_ROLLRATE_I
3  float64 MC_ROLLRATE_D
4  float64 MC_ROLL_P
5  float64 MC_PITCHRATE_P
6  float64 MC_PITCHRATE_I
7  float64 MC_PITCHRATE_D
8  float64 MC_PITCH_P
9  float64 MC_YAWRATE_P
10 float64 MC_YAWRATE_I
11 float64 MC_YAWRATE_D
12 float64 MC_YAW_P
13 ---
14 bool success
```

The `PIDCmd` service is in charge of notifying the PID Setter node that one (or more) PID gain needs to be changed. The PID gains (e.g. `MC_ROLLRATE_P`) that need to be changed can be specified as float. Given the ROS service's format, a PID gain that does *not* need to be changed must be set to `-1`.

- **ActionInfo service**

Listing 3: `ActionInfo` service

```
1 string action_id
2 ---
3 bool success
```

The `ActionInfo` service is in charge of notifying the Recorder node that the drone has just started performing a certain action. The action that just started being performed is specified as a string in the `action_id` field. The possible actions are: `hover`, `roll`, `pitch` and `yaw`.

- **LogInfo service**

Listing 4: `LogInfo` service

```
1 string log_filename
2 string action_id
3 ---
4 bool success
```

The `LogInfo` service is in charge of notifying the LogProcessor node that a log file is ready to be processed. The name of the log file is specified by the `log_filename` field, and the action of interest (e.g. rolling) is specified by the `action_id` field. This allows the Log Processor node to know which Euler angle it should analyse (roll, pitch or yaw).

- **LogInfo service**

Listing 5: `LogInfo` service

```
1 string log_filename
2 string action_id
3 ---
4 bool success
```

The `LogInfo` service is in charge of notifying the LogProcessor node that a log file is ready to be processed. The name of the log file is specified by the `log_filename` field, and the action of interest (e.g. rolling) is specified by the `action_id` field. This allows the Log Processor node to know which Euler angle it should analyse (roll, pitch or yaw).

- **PIDAdvice service**

Listing 6: `PIDAdvice` service

```
1 string MC_ROLLRATE_P
2 string MC_ROLLRATE_I
3 string MC_ROLLRATE_D
4 string MC_ROLL_P
```

```
5  string MC_PITCHRATE_P
6  string MC_PITCHRATE_I
7  string MC_PITCHRATE_D
8  string MC_PITCH_P
9  string MC_YAWRATE_P
10 string MC_YAWRATE_I
11 string MC_YAWRATE_D
12 string MC_YAW_P
13 ---
14 bool success
```

The `PIDAdvice` service is in charge of notifying the User Interface node of the advises computed by the Log Processor node. The advice concerning each PID gain (e.g. `MC_ROLLRATE_P`) can be specified as string (or left blank if no advice needs to be sent for this PID gain.

- **AttitudePruned message**

Listing 7: `AttitudePruned` message

```
1 Header header
2
3 geometry_msgs/Quaternion orientation
```

The `AttitudePruned` message stores the drone's current attitude as a quaternion. It is continuously published by the IMU Pruner node.

5.2 PIDAdvisor

5.2.1 What is a PIDAdvisor?

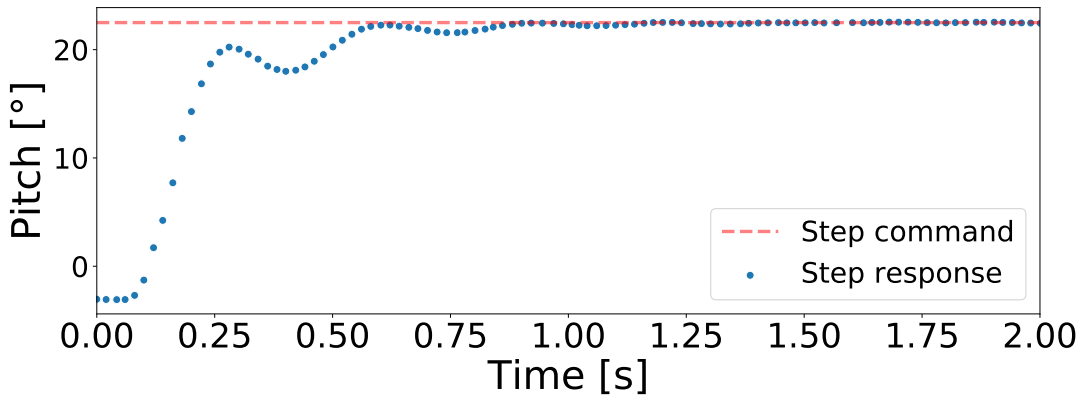


Figure 26: Example of a step response for a 22.5° step command (2 seconds duration) in pitch angle

The Log Processor node is the node in charge of reading, plotting and analysing the log containing the step response. This analysis outputs a `PIDAdvice` (see Listing 6) to help the user choose which PID gains should be tuned next, as well as how they should be tuned. The Log Processor node delegates this logic to a Python object called a `PIDAdvisor`, whose sole purpose is to analyse the log file. The user can use an already implemented `PIDAdvisor`, or freely implement its own `PIDAdvisor` if they wish to. A guide on how to do so is provided below.

5.2.2 How to implement your own `PIDAdvisor`

A `PIDAdvisor` is an object handling the "log file → `PIDAdvice`" logic.

To implement your own `PIDAdvisor`, you need to:

- Create a new module (e.g. `my_pid_advisor.py`) in `src/log_processor/src/pid_advisors/`
- In it, implement a class (e.g. `MyPIDAdvisor`) following the template below:

Listing 8: PIDAdvisor template

```
1  from dtr_msgs.srv import PIDAdvice
2
3  class MyPIDAdvisor:
4   def __init__(self):
5     '''Your code here'''
6
7   def computePIDAdvice(self, preprocessed_log, action_id, step_angles):
8     pid_advice = PIDAdvice()
9     '''Your algorithm here'''
10
11    return pid_advice
```

- Modify `pid_advisor_interface.py`:
  - Import your newly created class `MyPIDAdvisor` from the newly created module `my_pid_advisor`:

    Listing 9: Import statement to add in the `pid_advisor_interface.py` file

    ```
    1 from pid_advisors.my_pid_advisor import MyPIDAdvisor
    ```

  - Decide on an `advisor_id` (uniquely identifying your PIDAdvisor, e.g. `MY_ADVISOR`) and add your newly created PIDAdvisor to the available advisors by editing the `__init__(self, advisor_id)`:

Listing 10: elif statement to add in the `pid_advisor_interface.py` file

```
def __init__(self, advisor_id):
  if advisor_id == "DUMB":
    self.pid_advisor = DumbPIDAdvisor()
  elif advisor_id == "MY_ADVISOR":
    self.pid_advisor = MyPIDAdvisor()
  else:
    self.pid_advisor = None
    rospy.logerr("Unknown advisor_id: {}".format(advisor_id))
```

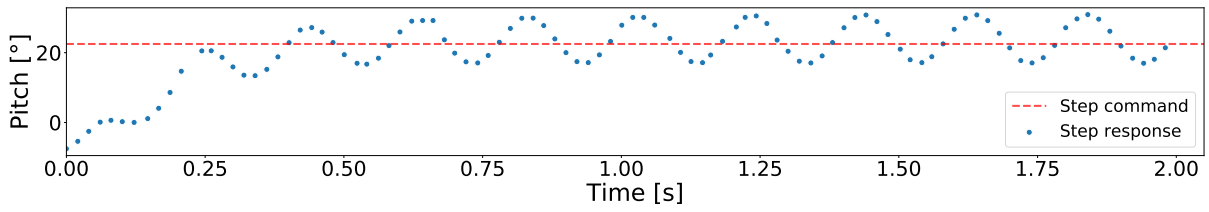
- Modify your configuration file (located in `src/user_interface/config/`) by changing the parameter `log_processor/pid_advisor_id` to your PIDAdvisor's `advisor_id`.

5.2.3 `PIDAdvisor` example: Ziegler-Nichols

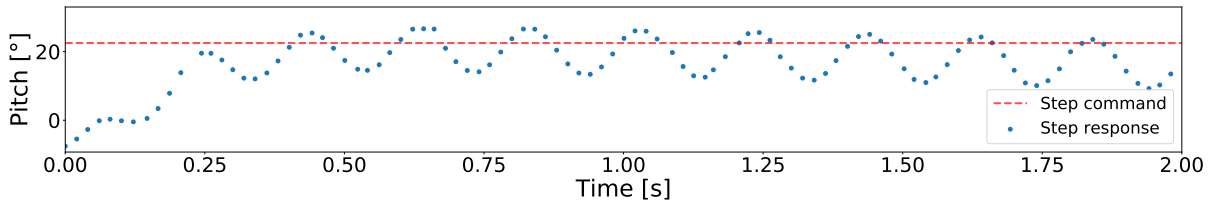
For the sake of example, a `PIDAdvisor` using the Ziegler-Nichols method has been implemented.

Ziegler-Nichols method

The Ziegler-Nichols method is a heuristic tuning method for PID controllers [24]. The method consists in setting all three PID gains to 0, and then slowly increasing the P gain until the step response has stable oscillations (see Fig. 27). This P gain is called the *ultimate gain* (K_u), and the corresponding oscillation period is labeled as T_u. The PID gains can then be computed from those two values. An outline of the algorithm's pseudo-code is given in Algorithm 1.



(a) Step response with stable oscillations around the step command



(b) Step response with stable oscillations, drifting away from the step command

Figure 27: Two different step responses with stable oscillations

Implementation of the Ziegler-Nichols method in a PIDAdvisor

The PIDAdvisor `ZieglerNicholsPIDAdvisor` (implemented in `/log_processor/src/pid_advisors/ziegler_nichols_pid_advisor.py`) uses the Ziegler-Nichols method to compute its `PIDAdvice`.

The implementation is as outlined in Algorithm 1.

Algorithm 1: Ziegler-Nichols method

$K_P \leftarrow 0$;
$K_I \leftarrow 0$;
$K_D \leftarrow 0$;
while *step response does* ***not*** *have stable oscillations* **do**
  increase K_P;
end
$K_u \leftarrow K_P$;
$T_u \leftarrow$ oscillation period;
$K_P \leftarrow 0.6K_u$;
$K_I \leftarrow \frac{1.2K_u}{T_u}$;
$K_D \leftarrow \frac{3K_uT_u}{40}$;

where the PID gains are K_P, K_I and K_D, respectively.

Out of all the steps listed in Algorithm 1, the least straight-forward one is the detection of stable oscillations. Indeed, as can been seen in Fig. 27, the stable oscillations will not necessarily occur around the step command. Hence, in order to determine whether the oscillations are stable, a tilted sinusoid (see Fig. 28) is fitted to the step response. The titled sinusoid's function $f(x)$ is defined as:

$$f(x) = \alpha x + A\sin(\frac{2\pi x}{T} + \phi)$$

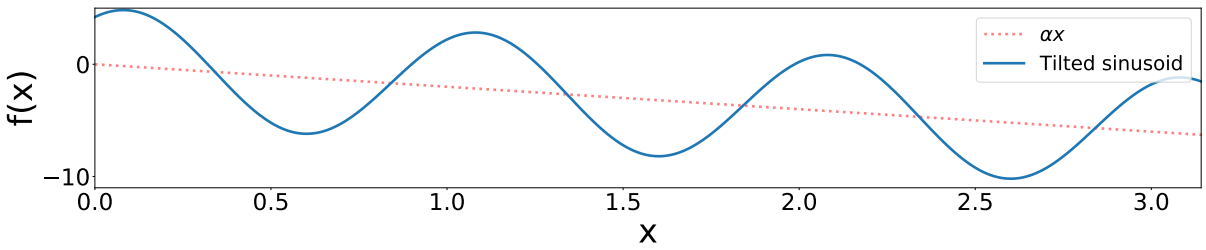
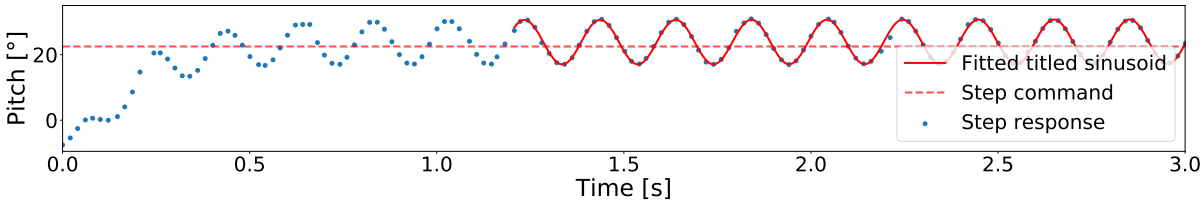


Figure 28: Tilted sinusoid

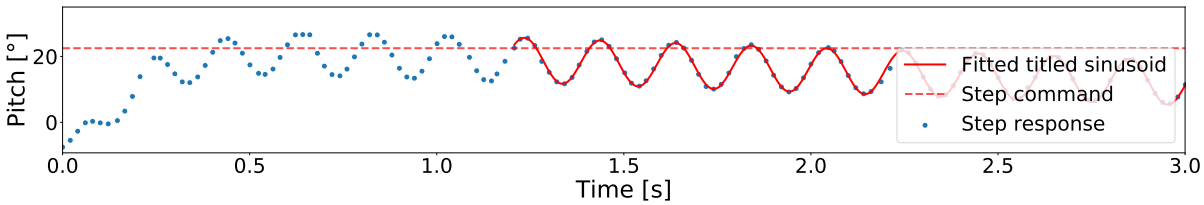
Such a tilted sinusoid is fitted to the part of the step response corresponding to time >

$\beta *$ risetime, where β is a hyper-parameter[4] of this PIDAdvisor (by default, $\beta = 3$). This is done to ensure that the step response will have settled into its stable oscillations phase.

Figure 29: A tilted sinusoid fitted to a step response with stable oscillations ($\beta = 3$)



(a) A tilted sinusoid fitted to a step response with stable oscillations (without drift) ($\beta = 3$)



(b) A tilted sinusoid fitted to a step response with stable oscillations (with drift) ($\beta = 3$)

Figure 30: Two titled sinusoid fitted to different step responses with stable oscillations

The PIDAdvisor then determines whether the oscillations are stable by computing the MSE[5] between the fitted titled sinusoid and the step response (e.g. the fitted titled sinusoid in Fig. 30a has a MSE of 0.6). If and only if this MSE is below a certain threshold[6], the step response is considered as having stable oscillations. The Ziegler PID gains can then be computed as per the Ziegler-Nichols method (see Alg. 1) and forwarded to the Log Processor node as a `PIDAdvice`. If the step response is *not* considered as having stable oscillations, then the `PIDAdvice` sent to the Log Processor node only consists in an increase of the P-gain.

[4] specified in the configuration file as `advisor/ziegler-nichols/risetime_factor`

[5] Mean Squared Error, defined as $\frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$), where y_i is the true signal and $\hat{y}_i$ the fitted sinusoid's signal

[6] specified in the configuration file as `advisor/ziegler-nichols/MSE_threshold`

6 THE RIG

The tuning software works regardless of whether the drone is mounted on a rig or not. However, flying the drone unconstrained requires the user to have enough space for them to observe the drone responding to the step command. If the controller being tuned was the position controller, the step command would consist of a position command. In this case, the space required corresponds to the distance between the drone's current position (prior to the position step command) and the position step command. However, this is not true when the controller being tuned is the attitude (or attitude-rate) controller, which are the controllers tuned by the tuning software.

Indeed, by sending an attitude step command (e.g. 20° in roll angle) for a certain duration, the drone is forced to roll. An unconstrained drone will hence simply fly in the direction it rolled towards (see Fig. 31). Additionally, tuning a drone in unconstrained flight assumes that the drone's controllers are already tuned well enough for the drone to be able fly, which might not be the case.

Figure 31: Evolution of an unconstrained drone during the tuning of its roll-rate controller

By using a rig to constrain the drone in position but not in orientation, the need for a wide space where the drone could fly is thrown away. Another aspect is the user's safety and the drone's integrity: As long as it stays constrained by the rig, the drone is ensured not to hurt the user, and not to damage itself (e.g. by hitting a wall) during the tuning process.

For those reasons, a rig was developed alongside the tuning software. It is however important to note that the software and the rig are entirely separate, and that the tuning software is in no way aware that the drone is constrained by a rig. This allows future works to be done on the rig without having to modify the tuning software.

6.1 Design specifications

The primary purpose of the rig is to constrain the position of the drone without constraining its orientation. A gimbal-based solution (see Fig. 32) was attractive at first due to its 3 DOF (roll, pitch and yaw). Unfortunately, this solution suffers from adaptability issues: As soon as the drone gets larger due to an impractical shape, the *entire* rig needs to be rebuilt to fit the

new drone inside of it. For this reason, the gimbal-based solution was not pursued.

Figure 32: Example of a gimbal-based solution [9]

A ball-joint solution (see Fig. 33) was initially attractive for its simplicity and 3 DOF (roll, pitch and yaw). However, given that in PX4, each Euler angle's controller is decoupled from the two other Euler angles, the ability to tune all three at the same time was not needed.

Additionally, the structure of the ball-joint solution requires the ball-joint to be located below the center of gravity of the drone. Unfortunately, most drones' batteries are located right below the drone's center of rotation. This means that the ball-joint would be located several centimeters away from the drone's center of gravity. This is an issue since, in unconstrained flight, the center of rotation of the drone coincides with its center of gravity. However, while on the rig, the center of rotation of the drone would be forced to coincide with the ball-joint. Tuning a drone whose center of rotation has been offset from its center of gravity means tuning a drone that responds differently to step commands than when it is in unconstrained flight. For those reasons, the ball-joint idea was not pursued.

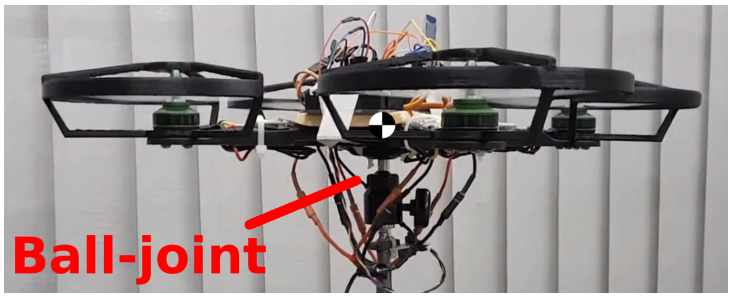


Figure 33: Example highlighting the ball-joint solution's issue (Figure adapted from [14])

As mentioned above, the tuning process of an unconstrained drone in roll and pitch requires to have enough space where the drone could evolve in (e.g. a hallway). This is not true for the tuning process in yaw. Indeed, provided that all the controllers (apart from the yaw and yaw-rate controllers) are well tuned, the drone should be able to maintain its position during the yaw tuning process, with only the yaw angle changing. For this reason, it was decided that the rig did not need to support tuning in the yaw angle, and hence could be limited to 2 DOF (roll and pitch).

Given that the roll and pitch controllers are decoupled in PX4, it was decided that having the user physically interact with the rig to switch between roll and pitch tuning was reasonable, since it would occur at most once per tuning session. Additionally, most common drones are symmetrical and therefore the roll and pitch controllers are treated as the same (and hence would have the same controller gains). However, unconventional drone morphologies are not always symmetric. This switch should therefore be made as practical as possible, for a smoother user experience.

6.2 The test rig

It was decided to opt for a rod-based rig (see Fig. 34). This solution allows for 1 DOF (roll or pitch, depending on how the drone is mounted on the test rig). By having the rod go between the battery and the rest of the drone, the distance between the center of rotation when tuning and the drone's center of gravity is minimized.

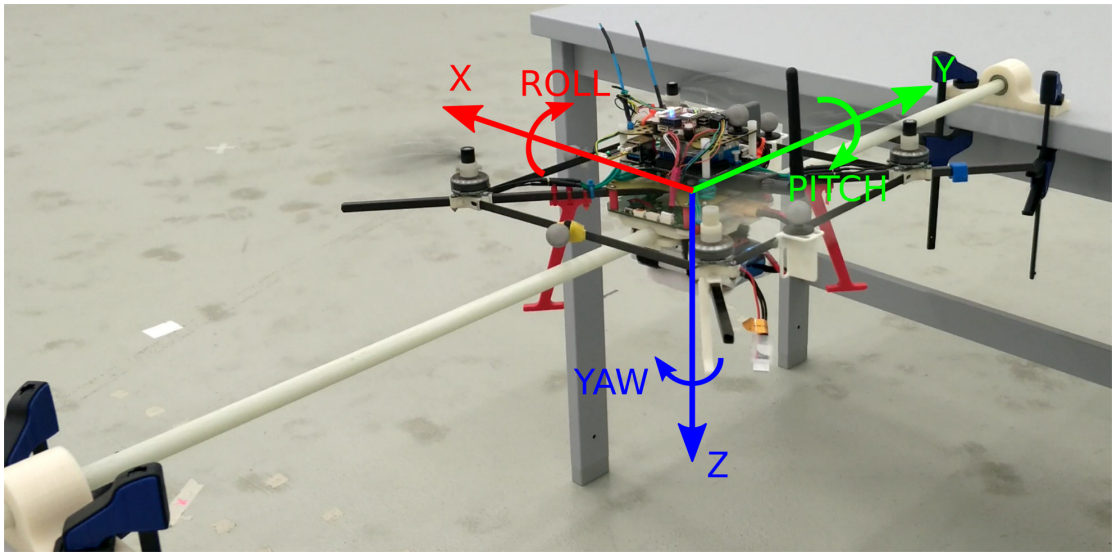


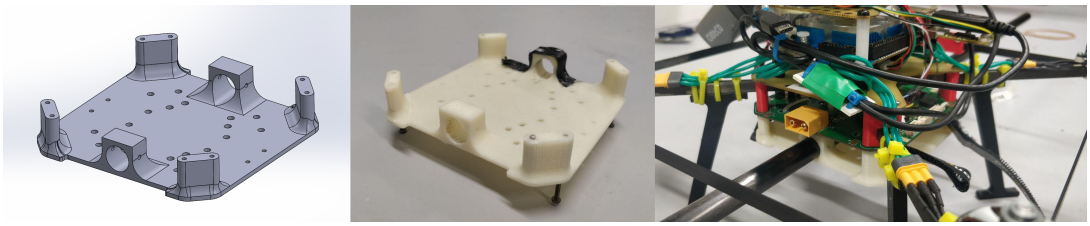
Figure 34: Final iteration of the test rig being used to tune the pitch-rate controller

The test rig remains somewhat adaptable by only having to get a longer rod if the drone gets larger. It is important to note that because of this, the rig is not fully adaptable. However,

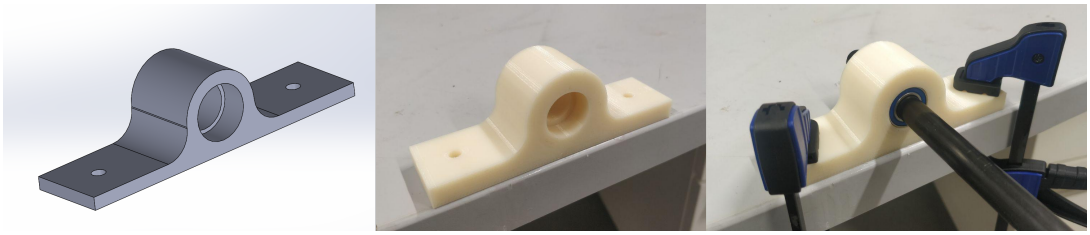
adapting it to a new drone's morphology is made straightforward and easy, by only having to change one readily-available element (i.e. the rod).

6.2.1 First iteration of the test rig

The test rig consists of 3 different parts : the ***drone part***, the ***table part*** and a **rod** (see Fig. 35).



(a) The ***drone part***, from CAD model to mounted



(b) The ***table part***, from CAD model to mounted

Figure 35: The ***drone part*** and the ***table part***, from CAD model to mounted

The *drone part* mounts directly between the drone and the battery. The rod is passed through the *drone part* (through two housing-holes) and is blocked by the tight-fitting of the housing-holes. By having the rod pass between the battery and the drone, the distance between the center of rotation while tuning and the center of rotation while flying is minimized, allowing us to tune a structure behaving closely to the way the (unconstrained) structure behaves when flying. Each end of the rod is then passed through bearings mounted on the *table parts*. Each *table part* is then clamped to an even surface (e.g. a table) (see Fig. 36).

Figure 36: Closeup of the *table part* with a bearing mounted on it and a rod going through it

In order to further reduce the impact of the test rig on the tuning process, the weight of the rig was kept to the bare minimum (see Tab. 5). For the *drone part* to remain sturdy enough, its minimum allowed thickness was set to 2mm.

Table 5: Mass of each individual part of the first iteration of the test rig

	Mass [g]	**Mass [% of drone's mass]**
drone part	26	4
table part	49	8
rod	70	11

It is important to note that what matters is not the total mass added to the drone, but rather the drone's change in rotational inertia. When pitching/rolling on the rig, the drone pitches its own mass, as well as the *drone part* and the rod attached to it. Their influence on the drone's rotational inertia is assumed to be negligible given their position with respect to the drone's axis of rotation, as well as how lightweight they are compared to the drone's mass. Indeed, the *drone part* is a rather small part fixed close to the drone's center of rotation. Additionally, the longitudinal axis of the rod coincides with the drone's rotational axis, massively reducing the rod's influence on the drone's rotational inertia (around this rotational axis).

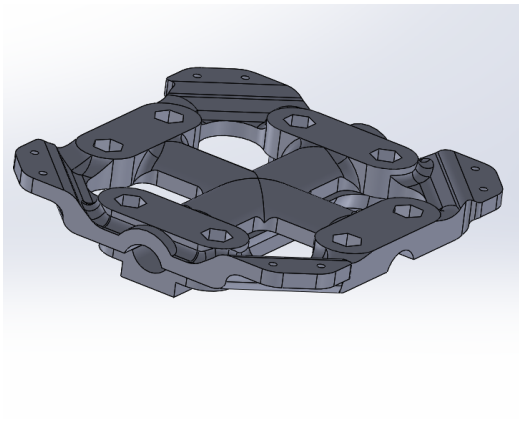
Issues:

Several issues were identified in the first iteration of the rig:

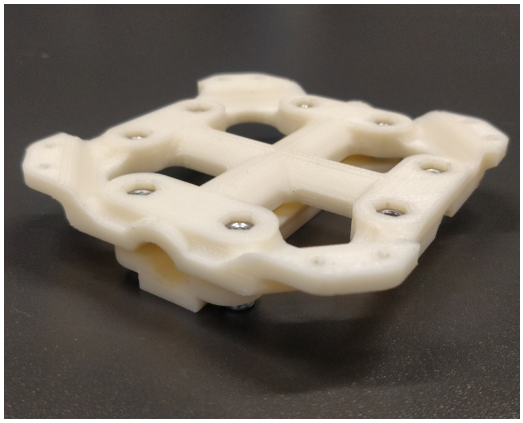
- **Structural integrity**: The minimal thickness of 2mm imposed on the *drone part* allowed for a lower mass at the cost of structural integrity, especially at the housing-holes where the constraints are high due to the tight-fitting of the rod. Additionally, the small thickness meant that the part could easily flex, which risked influencing the step response during the tuning process.
- **Housing-holes**: Having to insert the rod in the *drone part*'s housing-holes proved itself impractical and unsustainable (any extraction would loosen the fit of the housing-holes). Additionally, the housing-holes require a rod with exactly the right tolerance, lest it does not grip the rod firmly enough.
- **Switch between roll and pitch tuning**: Having to unscrew (8 screws), rotate and screw back the whole *drone part* to the drone in order to switch from roll to pitch tuning is impractical and time-consuming.
- **Battery positioning**: Drone batteries are commonly strapped to the drone by using Velcro straps. When mounted on the drone, the first iteration of the *drone part* blocks the access to those same Velcro straps, requiring the user to find another way of strapping the battery to the drone while tuning.

6.2.2 Second iteration of the test rig

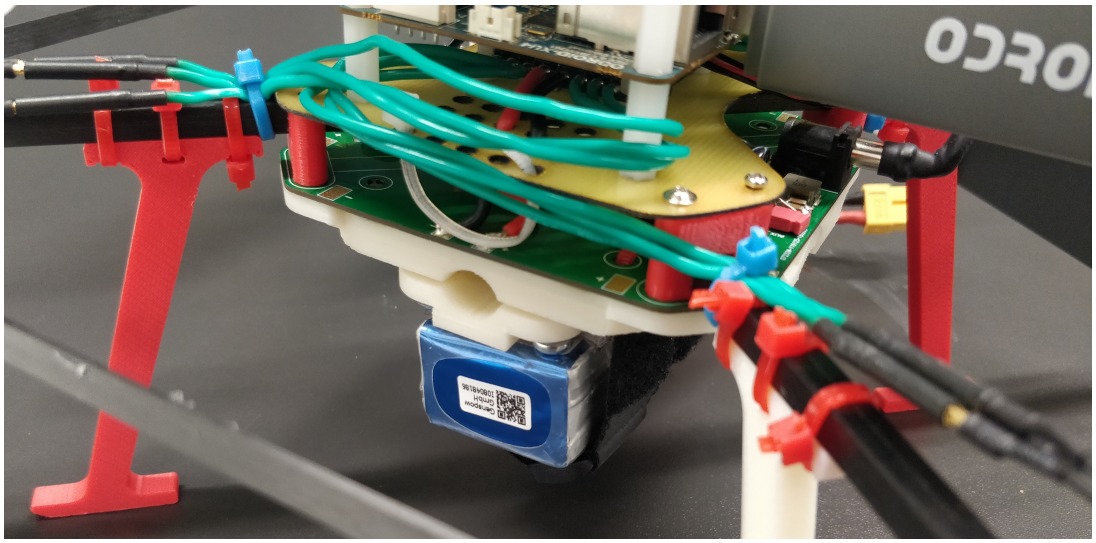
In order to alleviate the issues that arose from the rig's first iteration, a second iteration of the *drone part* was developed (see Fig. 37). The *table part* was kept unchanged. The changes are listed below.



(a) CAD model



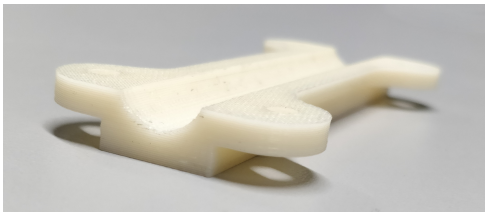
(b) Printed part



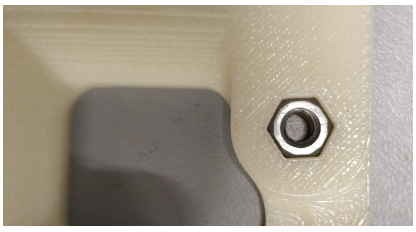
(c) Printed part mounted on the drone

Figure 37: Second iteration of the *drone part*, from CAD model to mounted on the drone

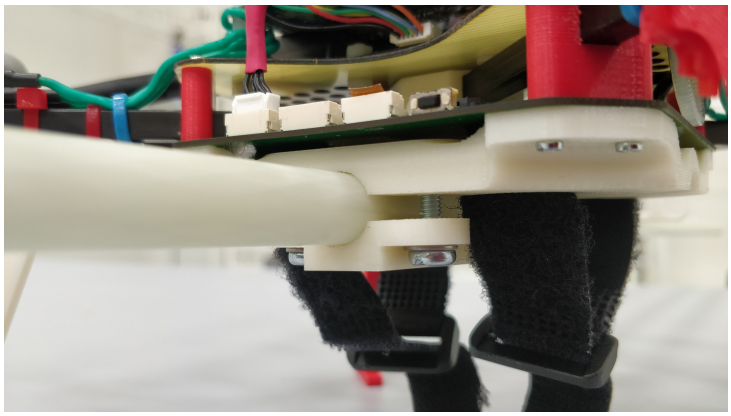
- **Housing-holes**: Fastening the rod to the *drone part* through only the tight-fitting of the housing-holes proved itself too unpractical to be a viable solution. Additionally, it required a rod with exactly the right tolerances. The *drone part* was modified to incorporate a *clasping part* (see Fig. 38). The rod is now pressed between the *drone part* and its *clasping part*. Hexagonal holes were added to the *drone part*, allowing for nuts to be lodged in them, prolonging the *drone part*'s life. Additionally, the tolerance needed on the rod is now loosened, given that tight-fitting is not needed anymore.



(a) *Clasping part* alone



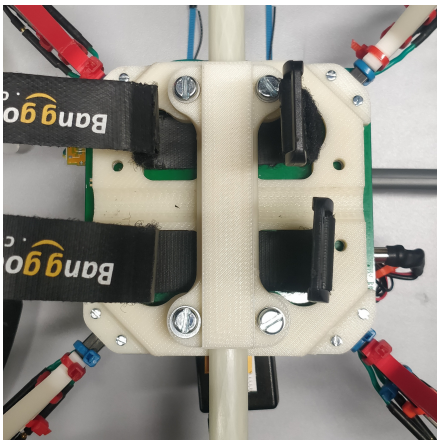
(b) One of the 8 hexagonal holes in the *drone part* with a nut lodged in it



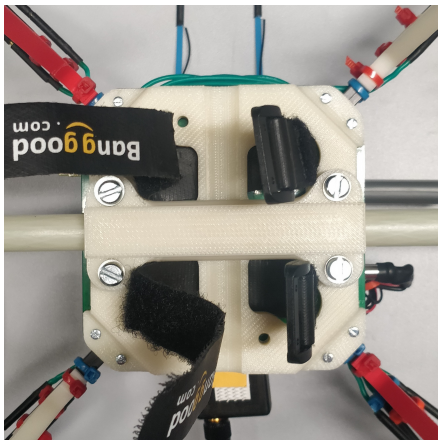
(c) Clasping system fastening the rod

Figure 38: Clasping system of the *drone part*'s second iteration

- **Switch between roll and pitch tuning**: Having to unscrew (8 screws), rotate and screw back the whole *drone part* to perform the switch between roll and pitch tuning proved itself too time-consuming and impractical. To simplify the process, the clasping system was used instead (see Fig. 38). Once screwed to the drone, the *drone part* should not be unscrewed throughout the whole tuning process. Instead, the *drone part* was made symmetrical in rotation so that switching between roll and pitch tuning only requires the user to unscrew the *clasping part*, rotate it and screw it back (see Fig. 39).



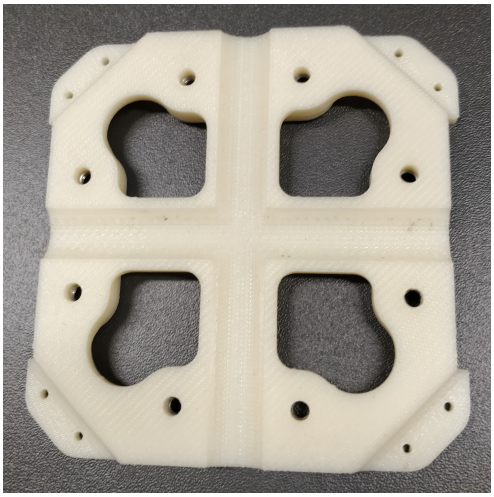
(a) Mounted for roll tuning



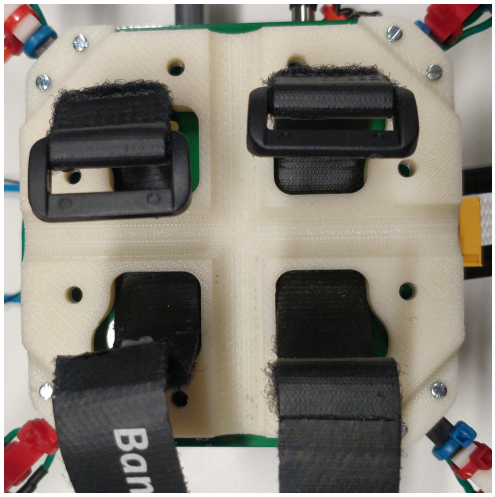
(b) Mounted for pitch tuning

Figure 39: The two ways of mounting the *clasping part*

- **Battery positioning**: Four holes were added to the *drone part* to allow for the Velcro straps to pass through it (see Fig. 40). The user can now use those same straps to strap the battery to the drone while tuning.



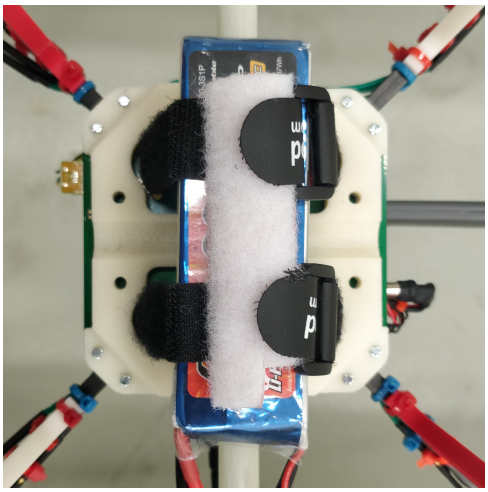
(a) The *drone part*'s second iteration, unmounted



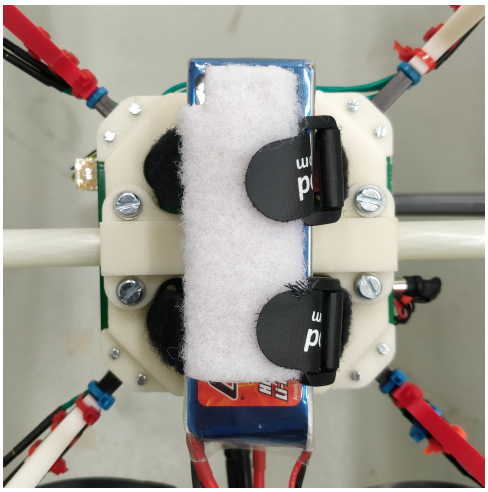
(b) The *drone part*'s second iteration, with Velcro straps going through the holes

Figure 40: The use of the holes in the *drone part*'s second iteration

Additionally, in order to maintain the battery against the drone regardless of whether roll or pitch is being tuned, the *clasping part* was designed with a flat surface, allowing the user to strap the battery to the drone in a stable manner, regardless of the controller being tuned (roll or pitch) (see Fig. 41).



(a) Battery placing when tuning the roll



(b) Battery placing when tuning the pitch

Figure 41: Battery placings on the *drone part*'s second iteration

- **Thickness**: In order to strengthen the second iteration of the *drone part*, the minimal thickness was raised from 2mm to 4mm. The new mass of the *drone part* can bee seen in Tab. 6.

Table 6: Mass of each iteration of the *drone part*

	Mass [g]	**Mass [% of drone's mass]**
first iteration	26	4
second iteration (including the *clasping part*)	45	7

7 MOVING FROM SIMULATION TO REALITY

The transition from simulation to reality was facilitated by the SITL approach, allowing us to have a tuning software functioning well in simulation before moving onto hardware. However, moving to hardware was not without its unforeseen issues, which are listed below:

- **PX4-related discrepancies**:
  - **Publishing the drone's arming status and flight mode**: The simulated PX4 is constantly publishing its state information (including arming status and flight mode) on the `/mavros/state` topic. Unfortunately, the PX4 running on hardware does not publish anything on this topic at runtime. Additionally, simulated PX4 accepts to be switched to OFFBOARD mode through a service call while PX4 running on hardware does not. This led to the development of a ROS node exclusive to simulation: the **Prepper node**. In simulation, it bears the responsibility of arming and switching the drone to OFFBOARD. This responsibility is assigned to the user (which can do it through the RC controller) when using the tuning software on real hardware.
  - **Getting and setting PX4 parameters**: PX4 parameters can be fetched and set through a service call (`/mavros/param/get` and `/mavros/param/set`, respectively). Such service calls are made by the tuning software to fetch and set PID gains. This works out of the box on the simulated PX4. Unfortunately, it does not work out of the box with the real PX4. For PX4 parameters to be fetched/set, one needs to first change them once through telemetry (by using *QGroundControl*). It is important to note that this issue could not be reproduced when using another FCU (the PixHawk 4), which could indicate that this issue is specific to the FCU used in this project.
- **Mismatch between the true (constrained) attitude and the attitude command**: Unfortunately, the PX4 OFFBOARD mode does not allow for sending individual attitude commands directly. Instead, one needs to send all three attitude commands. This requires the software to know the drone's current orientation. This can lead to issues if the software is wrong. For example, in the case of a mismatch between the current true yaw angle and the "believed" (by the software) yaw angle, the tuning software will try to steer the drone towards the "believed" yaw angle. Given that the drone is constrained in yaw by the rig, it is unable to yaw, which leads to rotors overheating, and eventually burning. This led to the implementation of the `calibrate hover` command (see Section 8.3.3), which fetches the drone's current pose (except roll and pitch) and sets it as the "hover pose", i.e. the pose that the drone needs to achieve when hovering. When mounted on the rig, the yaw angle of this hover pose is sent alongside any attitude command sent to the drone.

8 USER GUIDE

8.1 How to install the tuning software?

The tuning software is available on the Github of the LIS at:
`https://github.com/lis-epfl/drone-test-rig`

8.1.1 Requirements

The tuning software has been tested with:

- Ubuntu 16.04
- Python 2.7.16
- ROS Kinetic
- MAVROS for ROS Kinetic

8.1.2 Tuning software installation

- **Clone the repository** by running:

```
1    git clone --recursive https://github.com/lis-epfl/drone-test-rig
```

  The `-recursive` flag is needed to install the `px4-firmware` submodule.

- In the `px4-firmware` folder, **run the command below**. Keep installing missing Python packages until `make` hits 100%.
  (e.g. you might be missing `libprotobuf-dev` or `protobuf-compiler`).

```
1    make px4_sitl_default
```

- In the `px4-firmware` folder, **run the command below**. Keep installing missing python packages until `make` hits 100%. At which point, a Gazebo instance with a drone on the ground (unarmed) should be launched automatically.

```
1    make px4_sitl_default gazebo
```

- In the root folder, **build the tuning software** by running:

```
1    catkin_make
```

8.1.3 Sanity check

To make sure that everything has been installed correctly, you can **run the offboard example** as a sanity check:

- **Open three terminals** in the root folder
- On the *first* terminal, **launch a Gazebo instance** by going in the `px4-firmware` folder and running:

```
make px4_sitl_default gazebo
```

- **Wait** for the gazebo instance to be fully loaded (this might take a minute).
- On the *second* terminal, **launch MAVROS** by running:

```
roslaunch mavros px4.launch fcu_url:=udp://:14540@localhost:14557
```

- On the *third* terminal, **launch the offboard example node** by running:

```
source devel/setup.bash
rosrun offb offb_node
```

  The line `source devel/setup.bash` is important because it lets the terminal know about the packages you've just built, e.g. the `offb` package.

If everything works as intended, a Gazebo instance with a drone should appear. The drone should then arm, start hovering, and alternate between pitching, rolling and yawing (see Fig. 42).

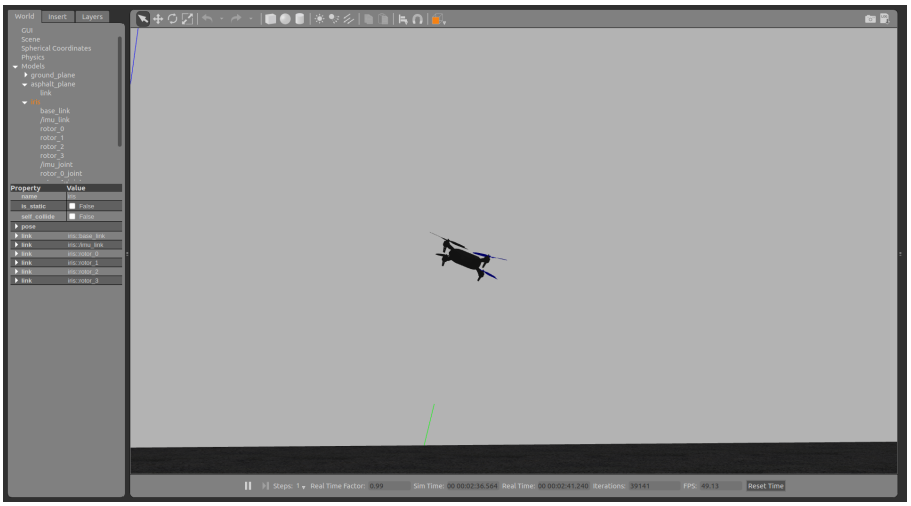


Figure 42: Gazebo simulation of the drone pitching during the offboard example

8.2 How to set up a drone to use it with the tuning software?

8.2.1 How to set up the drone to be able to switch to OFFBOARD mode

In order to be used with the tuning software, a drone needs to be in OFFBOARD mode, else it will not accept the pose commands sent by the offboard computer. This section serves as a guide on how to set up the drone for it to switch to OFFBOARD mode.

- **Prepare the drone**: Make sure that the drone is powered on and that its companion computer is connected to the same LAN network as your offboard computer.
- **SSH into the companion computer**: To SSH into the companion computer from your offboard computer, run the command below:

```
1    ssh odroid@COMPANION_COMPUTER_IP
```

  replacing `COMPANION_COMPUTER_IP` by the companion computer's private IP address. If you are using an ODROID, the password should be `odroid`.

- **Change the companion computer's** `.bashrc`: Change (e.g. with `nano`) the end of the `.bashrc` file so that the `ROS_NAMESPACE`, `ROS_IP` and `ROS_MASTER_URI` are set correctly:

```
1    source /opt/ros/kinetic/setup.bash
2    printf ">>> export ROS_NAMESPACE=drone_1\n"
3    export ROS_NAMESPACE=drone_1
4    printf ">>> export ROS_IP=http://COMPANION_COMPUTER_IP\n"
5    export ROS_IP=COMPANION_COMPUTER_IP
6    printf ">>> export ROS_MASTER_URI=http://MY_PRIVATE_IP_WIFI:11311\n"
7    export ROS_MASTER_URI=http://MY_PRIVATE_IP_WIFI:11311\
```

  replacing `COMPANION_COMPUTER_IP` and `MY_PRIVATE_IP_WIFI` by the private IP addresses of the companion computer and of your offboard computer, respectively.

- **Restart the SSH session**
- **Run MAVROS on the companion computer**
  - Start by following this guide: `https://dev.px4.io/v1.9.0/en/companion_computer/pixhawk_companion.html` and create the `99-pixhawk.rules` file.
  - Reboot the drone
  - In `/opt/ros/kinetic/share/mavros/launch/node.launch`, change the corresponding lines into:

```
<arg name="gcs_url" default="udp://@MY_PRIVATE_IP_WIFI"/>
<arg name="tgt_system" default="1"/>
<arg name="tgt_component" default="0"/>
<arg name="pluginlists_yaml" value="$(find mavros)/launch/
    px4_pluginlists.yaml" />
<arg name="config_yaml" value="$(find mavros)/launch/px4_config.
    yaml" />
```

replacing `MY_PRIVATE_IP_WIFI` by the private IP addresses of your offboard computer.

- Launch the MAVROS node by running the following command on the companion computer:

```
roslaunch mavros node.launch fcu_url:=/dev/ttyPixhawk:921600
```

- The drone is ready to be switched to **OFFBOARD**

It is important to know that for the drone to accept being switched to OFFBOARD, something needs to be published by the offboard computer (i.e. the tuning software needs to be running)

8.3 How to use the software?

8.3.1 Configuration files

The files `config/params_sim.yaml` and `config/params_real.yaml` (located in the `user_interface` package) set up the parameters of the tuning software for simulation (and respectively reality). Those parameters are loaded onto the ROS parameter server upon starting the tuning software and are used by the tuning software's ROS nodes. A list is given below:

- **ROS Topics**
  - Where to publish to command the pose, the attitude
  - Where to subscribe to get the attitude data and the state of the drone (connected, armed, mode)
- **ROS Services**
  - Where to call to get/set PID gains
- **Hover parameters**

  - Hover position (x,y,z coordinates)
  - Hover yaw

- **Step command parameters**
  - Step command's angle
  - Thrust to be given when sending the step command
  - Step command's duration

- **PIDAdvisor's parameters**

8.3.2 How to start the software?

The procedure to start the software depends on whether it must be run in simulation or on a real drone. Both procedures are given below.

Simulation:

- **Run the** `run_sim.sh` file (located in the root folder).
  (The first run might take a few minutes to load the gazebo instance.)
- **Interact** with the user interface.

Reality:

- **Publish the drone's absolute position** through the DroneDome (see Section 4.3.1)
- **Launch the dronedome utility** by running the below command in the root folder:

```
source devel/setup.bash
roslaunch dronedome dronedome.launch server:=gcs_mocap_motive
```

  This lets the drone know its absolute position by remapping the drone's absolute position (acquired through the DroneDome) to the `mavros/vision_pose/pose` topic. You need to have the `VRPN ROS client` package installed. If you do not have it, run the following command:

```
sudo apt-get install -y ros-$ROS_DISTRO-vrpn-client-ros
```

  You need to have the IP address of the computer running the Motive software saved under `gcs_mocap_motive` in your `/etc/hosts` file.
  You need to have the `ROS_IP` set to the IP of your computer to run the above script (in your `.bashrc` typically).

- **Open QGroundControl** and make sure the drone is in **STABILIZED**.

- **Run the** `run_real.sh` file (located in the root folder).
- **Calibrate the hover pose** by running `calibrate hover` in the user interface.
  The drone will automatically go hover at the hover pose (x,y,z and yaw) specified in your configuration file. Running `calibrate hover` sets the hover pose as the current pose.
- **Switch the drone to OFFBOARD** by following the instructions below:
  - **Arm the drone**.
  - **Switch to POSITION** manually through the RC controller. The flight mode of the drone on QGroundControl should indicate POSITION.
  - **Increase the throttle** and **switch to OFFBOARD** manually through the RC controller. he flight mode of the drone on QGroundControl should indicate OFFBOARD. It is important to increase the throttle first, else the drone refuses to switch to OFFBOARD.
- **Interact** with the user interface

8.3.3 How to use the user-interface?

The user interface node is the only node the user has to interact with. It presents itself as a prompt displaying the current PID gains for each attitude rate controller and each attitude controller (see Fig. 43).

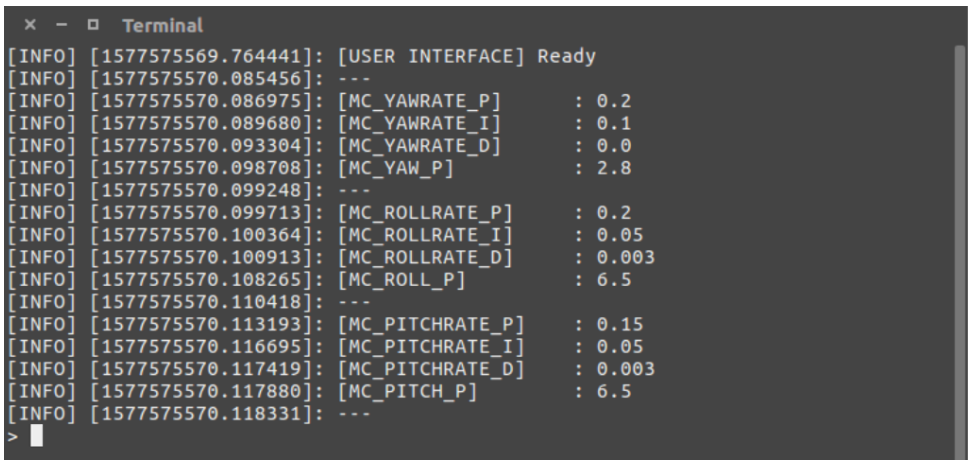


Figure 43: User interface when launching the tuning software

The user is then free to input one of the following commands into the user interface prompt:

- An **action** (i.e. `roll`, `pitch` or `yaw`)

- A **PX4 parameter corresponding to one of the attitude or attitude rate controllers' PID gains** (e.g. `MC_ROLLRATE_P`) (see Tab. 3 and Tab. 4)
- The `calibrate hover` command
- The `apply` command

Each of those commands are explained below.

- **An action (i.e. `roll`, `pitch` or `yaw`)**:

  Inputting an action (e.g. `roll`) into the user interface causes the drone to start rolling (the duration and exact angle of this step command is specified in the configuration file).

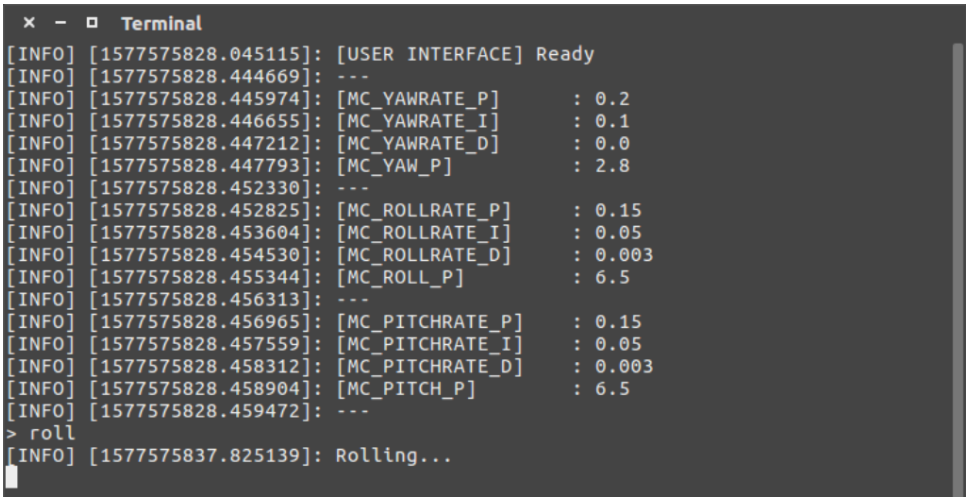


Figure 44: User interface when instructing the drone to roll

- **A PID gain (e.g. `MC_ROLLRATE_P`)**:

  Inputting one of the PID gains (as outlined in Tab. 3 and Tab. 4) allows the user to then specify the new value as a float. This will change the drone's PID gain to the specified value, and then send a step command corresponding to the PID gain that has just been changed (e.g. changing `MC_ROLLRATE_P` will cause the drone to roll).

```
Terminal
[INFO] [1577575569.764441]: [USER INTERFACE] Ready
[INFO] [1577575570.085456]: ---
[INFO] [1577575570.086975]: [MC_YAWRATE_P]      : 0.2
[INFO] [1577575570.089680]: [MC_YAWRATE_I]      : 0.1
[INFO] [1577575570.093304]: [MC_YAWRATE_D]      : 0.0
[INFO] [1577575570.098708]: [MC_YAW_P]          : 2.8
[INFO] [1577575570.099248]: ---
[INFO] [1577575570.099713]: [MC_ROLLRATE_P]     : 0.2
[INFO] [1577575570.100364]: [MC_ROLLRATE_I]     : 0.05
[INFO] [1577575570.100913]: [MC_ROLLRATE_D]     : 0.003
[INFO] [1577575570.108265]: [MC_ROLL_P]         : 6.5
[INFO] [1577575570.110418]: ---
[INFO] [1577575570.113193]: [MC_PITCHRATE_P]    : 0.15
[INFO] [1577575570.116695]: [MC_PITCHRATE_I]    : 0.05
[INFO] [1577575570.117419]: [MC_PITCHRATE_D]    : 0.003
[INFO] [1577575570.117880]: [MC_PITCH_P]        : 6.5
[INFO] [1577575570.118331]: ---
> MC_ROLLRATE_P
[INFO] [1577575576.628296]: [MC_ROLLRATE_P]     : 0.2 => ???
> Enter value: 0.15
[INFO] [1577575584.088377]: [MC_ROLLRATE_P]     : 0.2 => 0.15
[INFO] [1577575584.089226]: Rolling...
```

Figure 45: User interface when setting the `MC_ROLLRATE_P` to a new value

Once the corresponding step command has been sent and the response analyzed, the PIDAdvice are displayed to the user (see Fig. 46).

```
Terminal
[INFO] [1577577625.452897]: [USER INTERFACE] Ready
[INFO] [1577577625.977572]: ---
[INFO] [1577577625.978146]: [MC_YAWRATE_P]      : 0.2
[INFO] [1577577625.978603]: [MC_YAWRATE_I]      : 0.1
[INFO] [1577577625.979061]: [MC_YAWRATE_D]      : 0.0
[INFO] [1577577625.979541]: [MC_YAW_P]          : 2.8
[INFO] [1577577625.980102]: ---
[INFO] [1577577625.982788]: [MC_ROLLRATE_P]     : 2.0
[INFO] [1577577625.983341]: [MC_ROLLRATE_I]     : 0.05
[INFO] [1577577625.983843]: [MC_ROLLRATE_D]     : 0.003
[INFO] [1577577625.984269]: [MC_ROLL_P]         : 6.5
[INFO] [1577577625.984768]: ---
[INFO] [1577577625.985215]: [MC_PITCHRATE_P]    : 0.15
[INFO] [1577577625.985736]: [MC_PITCHRATE_I]    : 0.05
[INFO] [1577577625.986199]: [MC_PITCHRATE_D]    : 0.003
[INFO] [1577577625.986646]: [MC_PITCH_P]        : 6.5
[INFO] [1577577625.987027]: ---
> roll
[INFO] [1577577631.936302]: Rolling...
[INFO] [1577577640.608112]: Plotted step reponse saved in './plots'
[INFO] [1577577640.776687]: ---
[INFO] [1577577640.778306]: [MC_ROLLRATE_P]     : 2.0 (advice: 1.2)
[INFO] [1577577640.781684]: [MC_ROLLRATE_I]     : 0.05 (advice: 15.884725)
[INFO] [1577577640.782602]: [MC_ROLLRATE_D]     : 0.003 (advice: 0.022663)
[INFO] [1577577640.784842]: [MC_ROLL_P]         : 6.5
>
```

Figure 46: PIDAdvice displayed on the user interface after having sent a roll step command to the drone

- **The `calibrate hover` command [IMPORTANT]**

  Inputting the `calibrate hover` command will instruct the tuning software to set the current position (x,y,z coordinate) as well as the current yaw angle as the "hover pose", i.e. the pose sent to the drone when it is "hovering". It is especially important to calibrate the drone's pose when on the rig because the tuning software might otherwise

try to steer the (constrained) drone towards a yaw angle it cannot reach (due to its constraint in yaw), slowly overheating and burning the rotors.

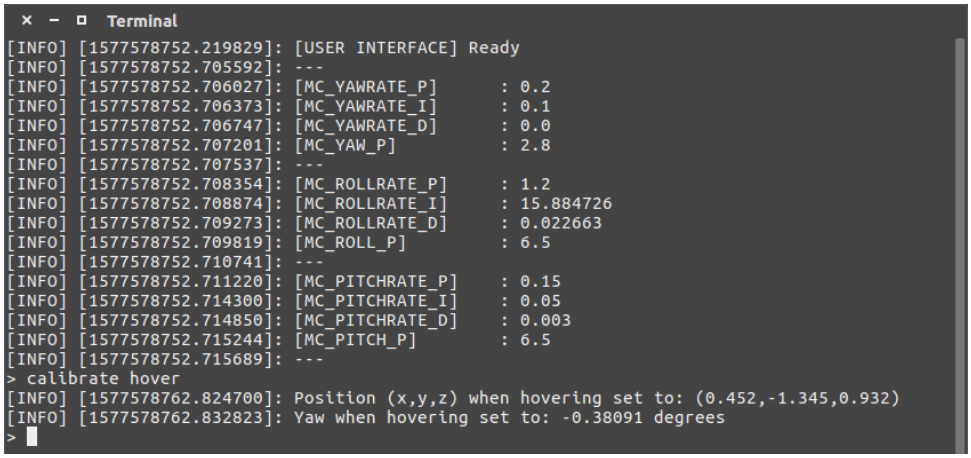


Figure 47: User interface when using the `calibrate hover` command

- **The `apply` command**

  Inputting the `apply` command will change the drone's PID gains to the ones displayed on the screen. This allows for quickly taking advantage of the PIDAdvisor.

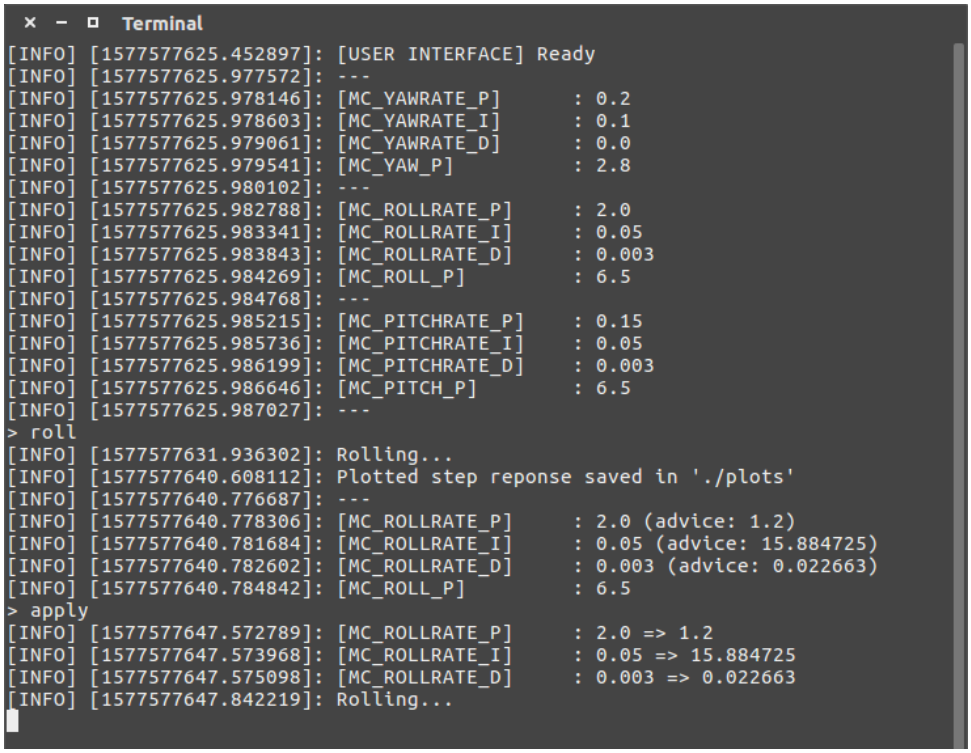


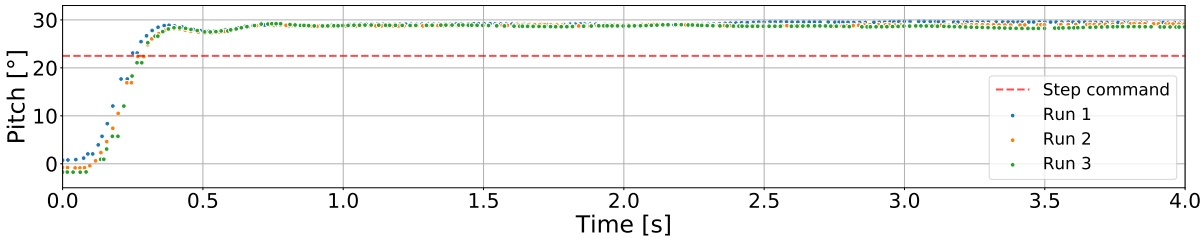
Figure 48: User interface when using the `apply` command

9 RESULTS

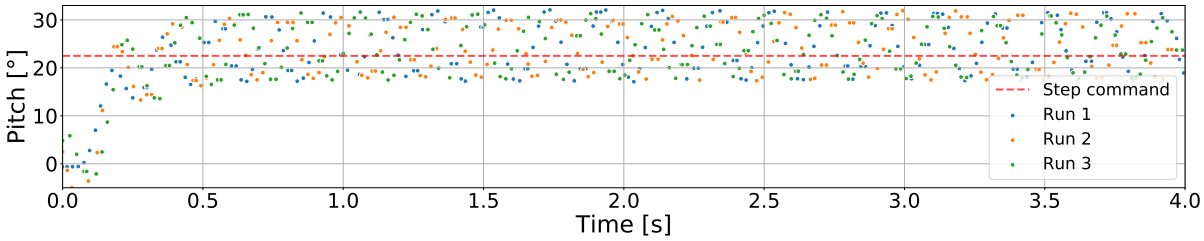
9.1 Tuning with the rig using the Ziegler-Nichols PIDAdvisor

The drone was mounted on the rig to tune its pitch angle. The pitch angle was tuned by sending **22.5° step commands**[7] of **4 seconds duration**. Before starting the tuning process, all three PID gains of the pitch-rate controller were set to 0. The tuning process was as follows:

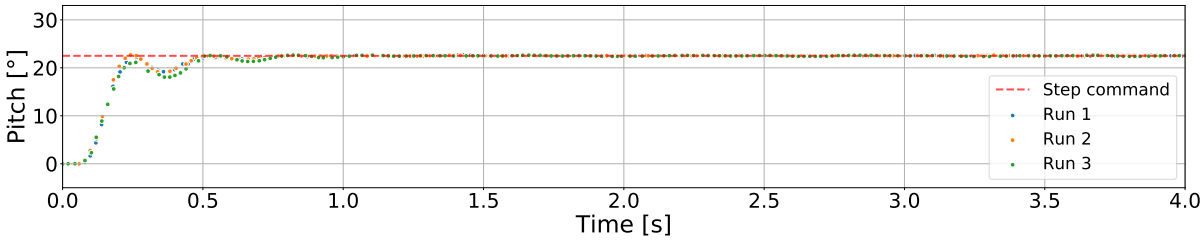
- Slowly increase the P-gain (increments of 0.05) until the drone is able to answer to a step command and come back to hovering (see Fig. 49a).
- Slowly increase the P-gain (increments of 0.01) until stable oscillations were detected by the software (see Fig. 49b).
- Apply the Ziegler gains (as computed by the software, see Tab. 7 and Fig. 49c).



(a) Initial step response after one increase in the P gain (all other gains set to 0)



(b) Gains causing stable oscillations



(c) Ziegler gains

Figure 49: Step responses (**on the rig**) of a 22.5° step command in pitch angle with **different pitch-rate controller gains** (see Tab. 7) (4 seconds duration, 3 runs)

The PID gains of the pitch-rate controller for each of the three steps listed above can be seen in Tab. 7.

[7] The step angle (22.5°) was chosen arbitrarily and deemed common enough during flight to be used as a step angle. The step angle can be specified by modifying the `step_controller/step_angles` parameter in the configuration file (see Section 8.3.1).

Table 7: PID gains of the pitch-rate controller when the drone was first able to come back to hovering after being sent the step command, when the drone started having stable oscillations and after applying the Ziegler gains.

	P gain	I gain	D gain
Initial step response	0.05	0	0
Stable oscillations	0.16	0	0
Ziegler gains	0.096	0.951	0.002

The drone was then unmounted from the rig and flown unconstrained in the DroneDome (see Fig. 50). Given the drone's symmetry, the gains of the roll-rate and pitch-rate controllers were set to be equal. A 22.5° step command (1 second duration) was sent to assess how well it performed outside of the rig. Due to space limitations, the duration of the step command was only of 1 second.

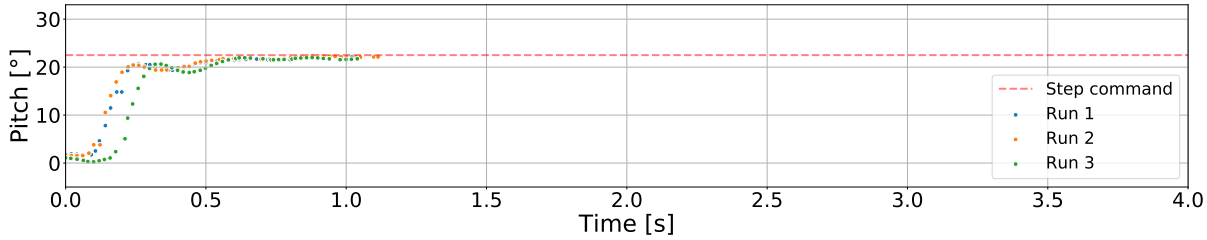


Figure 50: Step response (**unconstrained flight**) of a 22.5° step command in pitch angle with the **Ziegler gains** (see Tab. 7) (1 second duration, 3 runs)

It can be noted that the **step response on the rig is very similar to the step response in unconstrained flight** (see Fig. 49c and Fig. 50). This supports the assumptions that were made during the design of the rig, notably that the change in rotational inertia due to the rig would be negligible. This supports the idea that the step responses obtained on the rig can be generalized to an unconstrained flight's step response.

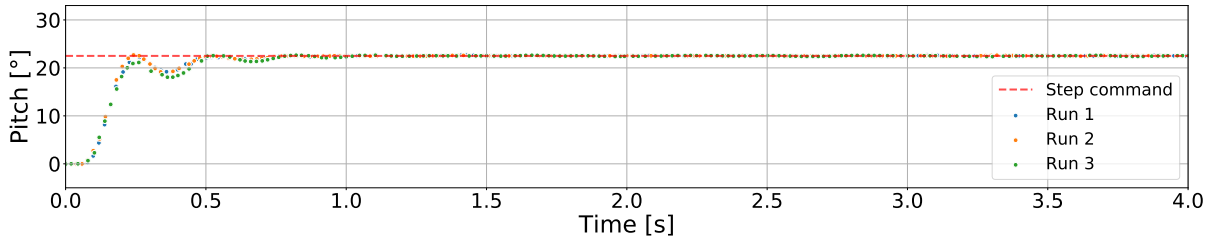
One can also note the **repeatability the step responses**, showing little discrepancies between all 3 runs (except for the stable oscillations step response (see Fig. 49b). The greater discrepancies between the 3 runs of the stable oscillations step response could be due to different initial conditions, given that the oscillations were also occurring when the drone was "hovering"[8].This further highlights the necessity of a test rig to perform the Ziegler-Nichols method on a drone. Additionally, one can note that despite the discrepancies in step responses of the stable oscillations, the period and amplitude of each run's oscillations are roughly identical (0.2 second and 8.7°).

[8] "hovering" is to be understood as "when the drone has a reference in roll/pitch angle of 0°"

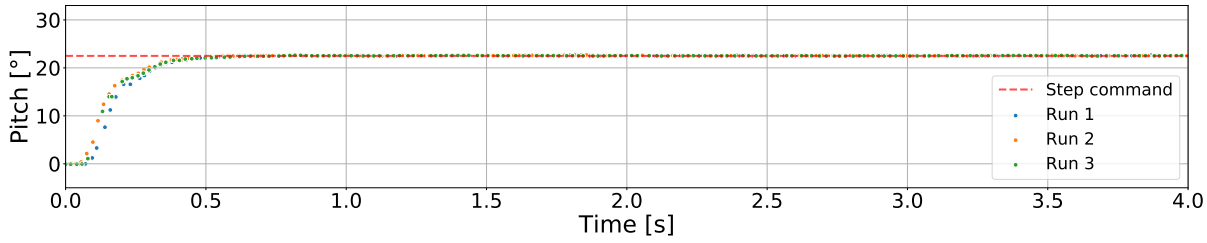
9.2 Hand-tuning a controller's gains

Blindly applying a PIDAdvisor's advice can lead to satisfactory results. However, for a finer tuning, one can further hand-tune the PID gains to obtain a step response that matches the wished step response more closely.

Starting from a step response that is judged satisfactory (obtained through the Ziegler-Nichols PIDAdvisor), the PID gains of the pitch-rate controller were further tuned the through hand-tuning, using the automatically-plotted step responses to assess how well the quad-copter behaves. The result of such a strategy can be seen in Fig. 51 and Tab. 8.



(a) Ziegler gains



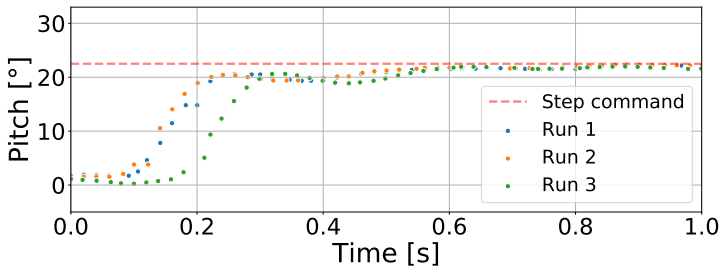
(b) Hand-tuned gains

Figure 51: Step responses (**on the rig**) of a 22.5° step command in pitch angle with **different pitch-rate controller gains** (see Tab. 8) (4 seconds duration, 3 runs)

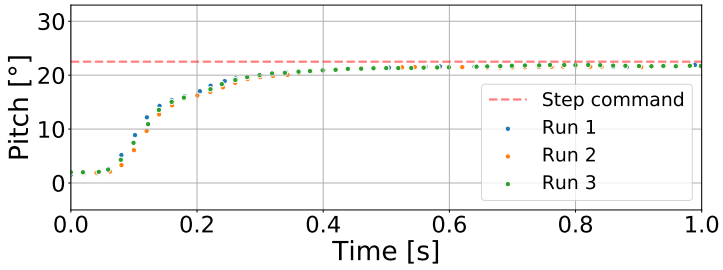
Table 8: PID gains of the pitch-rate controller obtained through Ziegler-Nichols and hand-tuning

	P gain	**I gain**	**D gain**
Ziegler gains	0.096	0.951	0.002
Hand-tuned gains	0.14	0.4	0.002

The drone was then unmounted from the rig and flown unconstrained in the DroneDome. Given the drone's symmetry, the gains of the roll-rate and pitch-rate controllers were set to be equal. A 22.5° step command (1 second duration) was sent to assess how well it performed outside of the rig. Again, due to space limitations, the duration of the step command was only of 1 second. The results are shown in Fig. 52.



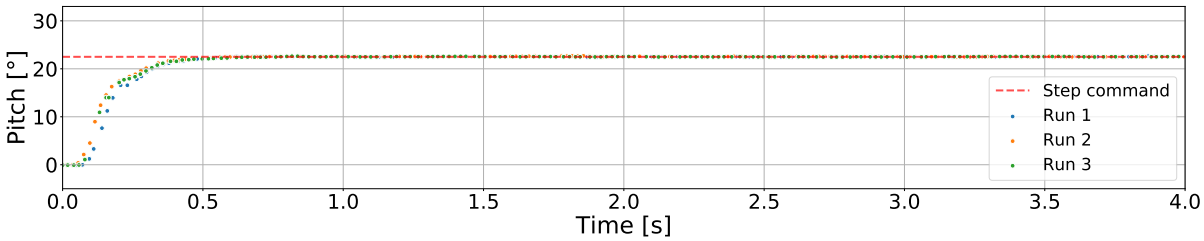
(a) Ziegler gains



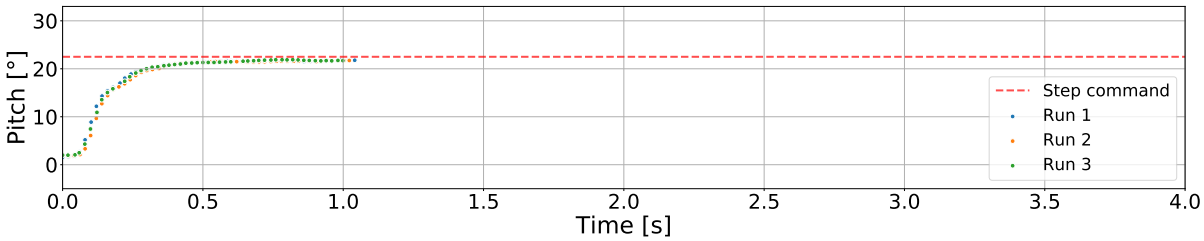
(b) Hand-tuned gains

Figure 52: Step responses (**unconstrained flight**) of a 22.5° step command in pitch angle with **different pitch-rate controller gains** (see Tab. 8) (1 second duration, 3 runs)

As can be seen in Fig. 52, the hand-tuned gains allow the drone to reach the step command in a much more stable manner but in roughly the same time. It is also interesting to note that the step responses on the rig and in unconstrained flight are very similar (see Fig. 53).



(a) On the rig, 4 seconds duration



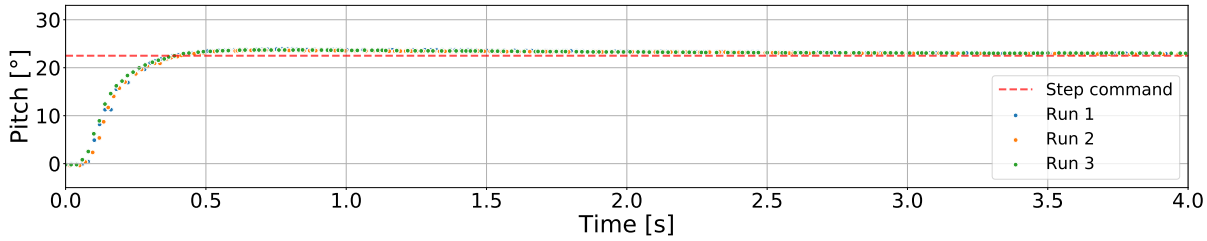
(b) Unconstrained flight, 1 second duration

Figure 53: Step responses (**on the rig** and in **unconstrained flight**) of a 22.5° step command in pitch angle with the **hand-tuned gains** (see Tab. 8) (3 runs)

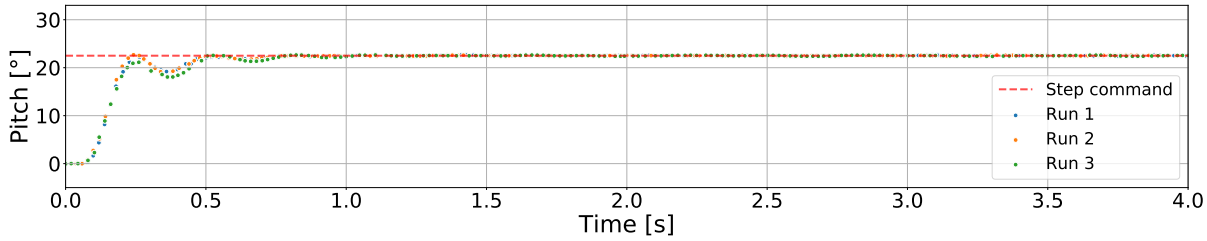
9.3 Comparison with the default PID gains

Let us now compare our results with the step responses obtained when using the default gains. Those PID gains are the default gains (according to PX4) associated with a standard

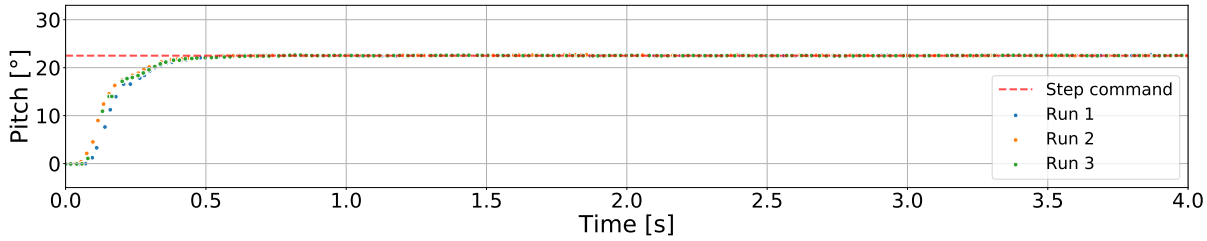
X-shaped quadcopter (see Tab. 9). The step responses were recorded both on the rig and in unconstrained flight (see Fig. 54 and Fig. 55).



(a) PX4 default gains (for an X-shaped quadcopter)



(b) Ziegler gains

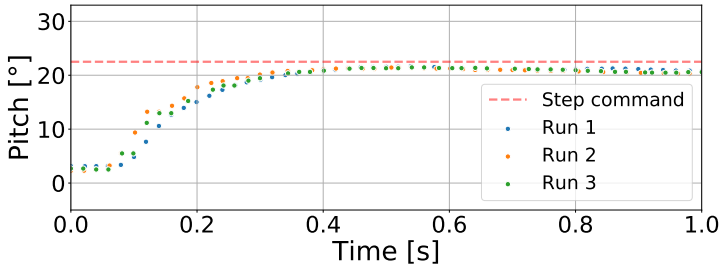


(c) Hand-tuned gains

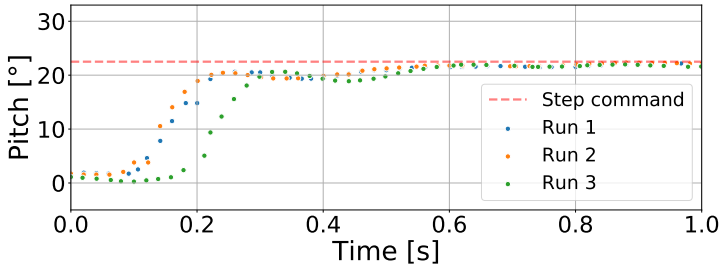
Figure 54: Step responses (**on the rig**) of a 22.5° step command in pitch angle with different pitch-rate controller gains (see Tab. 9) (4 seconds duration, 3 runs)

Table 9: PID gains of the pitch-rate controllers

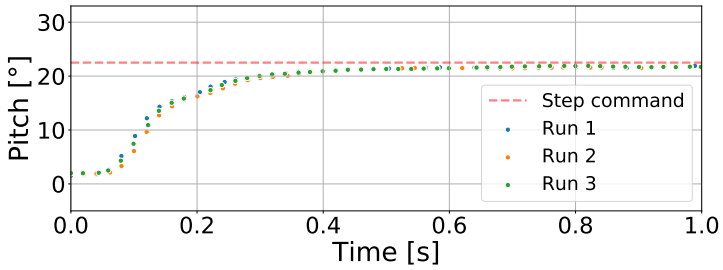
	P gain	I gain	D gain
PX4 default gains	0.15	0.05	0.003
Ziegler gains	0.096	0.951	0.002
Hand-tuned gains	0.14	0.4	0.002



(a) PX4 default gains (for an X-shaped quadcopter)



(b) Ziegler gains



(c) Hand-tuned gains

Figure 55: Step responses (**unconstrained flight**) of a 22.5° step command in pitch angle with different pitch-rate controller gains (see Tab. 9) (1 second duration, 3 runs)

As can be seen in Fig. 55, the gains obtained through hand-tuning perform slightly better than the default gains. The main difference is that the default gains introduce drift, while the hand-tuned gains do not. Indeed, the step response actually reaches the step command and stays there (see Fig. 55c), while the step response with default gains is closest to the step command at roughly 0.5s and then slowly drifts away by a few degrees (see Fig. 55a). This is however a very small difference.

Additionally, the **repeatability of the step responses** (low variance between each run) as well as the **similarity of the step responses obtained on the rig and in unconstrained flight** can be noted (see Fig. 55 and Fig. 54).

9.4 Tuning a drone with another rotational inertia in roll

Let us now inspect how general those results are. The drone's rotational inertia is changed by adding 2 carbon fiber rods (each 22.3cm long and 15g, i.e. 2.5% of the drone's mass) to its frame. Two screws are glued to the extremity of each rod, adding 38g (i.e. 6% of the drone's

mass) to the end of each rod (see Fig. 56). The goal is to change the drone's rotational inertia in roll while leaving the rotational inertia in pitch basically unchanged. The modified drone (as of now referred to as the *weighted drone*) can be seen in Fig. 57.

Figure 56: Carbon fiber rod with two screws glued to one extremity

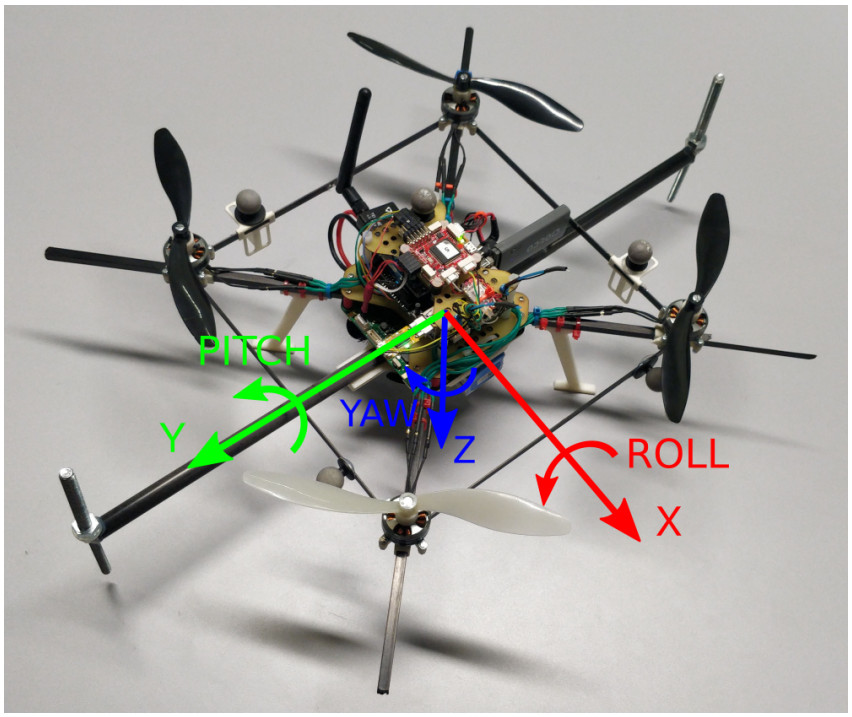
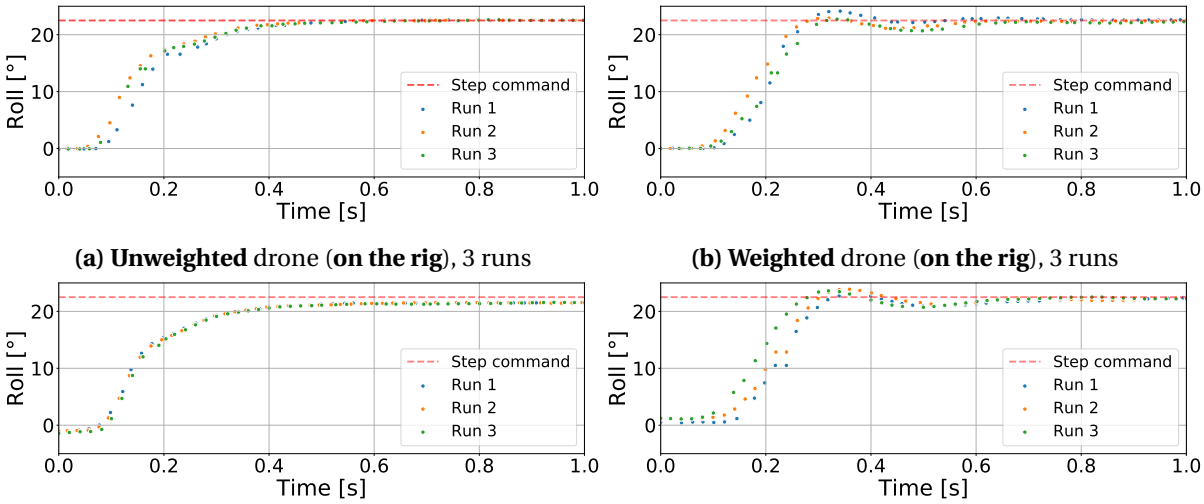


Figure 57: Weighted drone

The quality of the weighted drone's unconstrained flight when using the best PID gains obtained so far (i.e. the hand-tuned gains obtained in Section 9.2, see Tab. 10) is assessed before mounting the drone onto the rig. The step responses can be seen in Fig. 58.



(a) Unweighted drone (**on the rig**), 3 runs

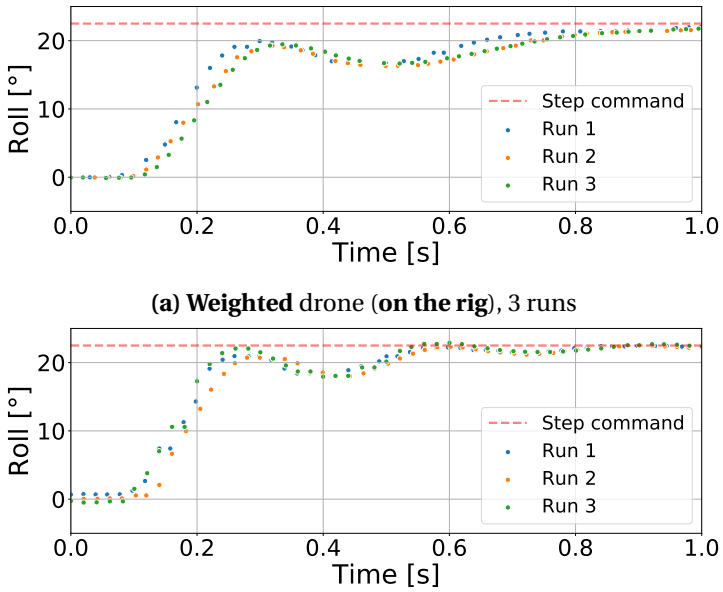
(b) Weighted drone (**on the rig**), 3 runs

(c) Unweighted drone (**unconstrained flight**), 3 runs

(d) Weighted drone (**unconstrained flight**), 3 runs

Figure 58: Step responses (22.5° step command, 1 second duration) in roll angle with the **hand-tuned gains from the unweighted drone** from Section 9.2 (see Tab. 10)

The tuning of the roll angle is then done using the Ziegler-Nichols PIDAdvisor, as outlined in Section 9.1. Once stable oscillations were reached, the Ziegler gains were applied and the quality of unconstrained flight was assessed (see Fig. 59)

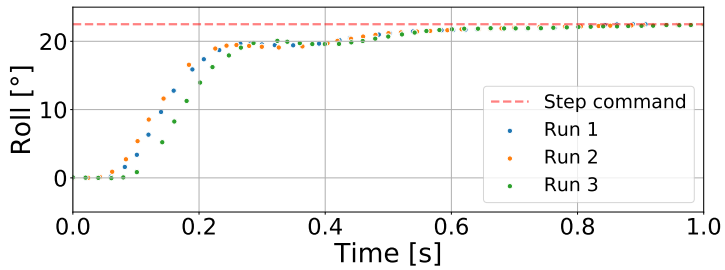


(a) Weighted drone (**on the rig**), 3 runs

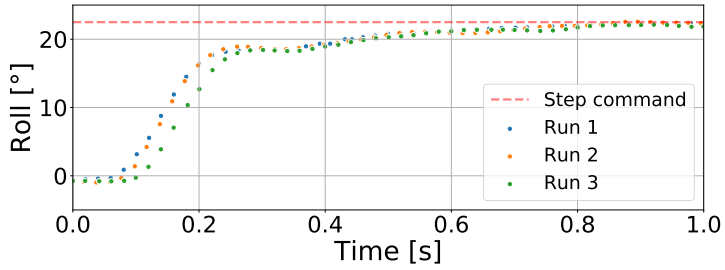
(b) Weighted drone (**unconstrained flight**), 3 runs

Figure 59: Step responses (22.5° step command, 1 second duration) in roll angle with **Ziegler gains** (see Tab. 10)

The gains were then hand-tuned by inspecting the step responses, in order to get the best controller possible. The result of this hand-tuning can be seen in Fig. 60.



(a) Weighted drone (**on the rig**), 3 runs



(b) Weighted drone (**unconstrained flight**), 3 runs

Figure 60: Step responses (22.5° step command, 1 second duration) in roll angle with **hand-tuned gains** (see Tab. 10)

Table 10: PID gains of the pitch-rate controller

	P gain	**I gain**	**D gain**
Gains hand-tuned for the unweighted drone	0.14	0.4	0.002
Ziegler gains	0.192	1.860	0.005
Hand-tuned gains	0.25	1.86	0.006

First, the difference in step response between the weighted and unweighted drone can be observed (see Fig. 58). This was expected given that the drone's dynamics have been changed by the added weight. While the weighted drone's step response (see Fig. 58d) could be considered acceptable, it is important to show that the tuning methods works with a drone with different dynamics.

By following the Ziegler-Nichols PIDAdvisor's advises, PID gains allowing the drone to fly were achieved (see Fig. 59). This shows that, from scratch and by only following the advice computed by the PIDAdvisor, the tuning software manages to find gains satisfactory enough to allow unconstrained flights. From that point one, PID gains with a smoother step response (no overshoot) were achieved through hand-tuning. (see Fig. 60)

As above, the **repeatability of the step responses** (low variance between each run) as well as the **similarity of the step responses obtained on the rig and in unconstrained flight** can be noted (see Fig. 58, Fig. 59 and Fig. 60).

10 CONCLUSIONS

In this project, a tuning platform helping the user throughout the tuning process of a multicopter was developed. This platform takes the shape of a tuning software used conjointly with a physical rig. The tuning software provides an interface for the user to interact with the drone and handles the analysis of the drone's behavior. The rig ensures that the drone is constrained in position without impairing its orientation, allowing for a safe assessment of the drone's step responses.

The tuning software was shown to effectively assist the user in the tuning process, reliably achieving stable PID gains for an X-shaped quadcopter. The experiment was repeated on an unbalanced quadcopter, whose inertia was changed by attaching weights to it. Its attitude-rate controller was then successfully tuned through the tuning platform.

The `PIDAdvisor` framework enables the automatic analysis of the step response and can yield PID gains allowing for stable flight while requiring few user inputs, as was shown through the use of the Ziegler-Nichols `PIDAdvisor`. Through the help of the automatically plotted step response, the user can further tune the PID gains if needed.

The rig allows for the study of the step response in roll and pitch while remaining adaptable to different drone morphologies. Additionally, the discrepancy between the step response on the rig and step response when flying unconstrained was shown to be negligible.

The final goal would be the publication of both the software and hardware parts, in order to let other people in the drone community benefit from our efforts.

11 FUTURE WORKS

The `PIDAdvisor` framework is yet to be fully exploited, e.g. by implementing other PID tuning strategies. Additionally, the use of machine learning techniques could be investigated to further develop the autonomous tuning provided by the tuning software.

The tuning software currently only focuses on the tuning of the attitude and attitude rate controllers. One could build on top of the current state of the software to use it for tuning the position and velocity controllers. Another improvement could be the development of a GUI for the user interface, allowing for a more user-friendly experience.

When it comes to the rig, it currently requires the *drone part* to be designed specifically for the drone. The development of a more versatile part would be an improvement. Additionally,

the current rig only allows for roll (or pitch) tuning, and requires the user to physically interact with the drone to switch between those two. This interaction takes time and the development of a 3DOF rig that is both adaptable and that does not temper with the drone's dynamics would prove itself to be very valuable.

Finally, the tuning platform should be tested on more unconventional drone morphologies, as to better understand the limitations of the tuning platform.

ACKNOWLEDGMENTS

I would like to thank Fabrizio Schiano for his guidance throughout this project. He was a great mentor, steered me towards what needed to be studied and actively supported me during all stages of this project, be it the development of the tuning software, the design of the rig, the experiments in the DroneDome or the writing of this report.

I would also like to thank Przemek Kornatowski for his help regarding the design of the rig, Anand Bhaskaran for his insights during the preparation of the mid-semester presentation, Olexandr Gudozhnik for his help concerning the hardware and Fabian Schilling for his DroneDome utility, which made using the OptiTrack very straightforward. Finally, I would like to thank Harry Vourtsis for his two drone batteries, which greatly facilitated the real-life experiments, as well as Hauke Maack, for our German chats.

I thoroughly enjoyed this semester project, the creative liberties I was given as well as the support I received from the LIS and its collaborators.

REFERENCES

[1] Judy Scott and Carlton Scott. "Drone delivery models for healthcare". In: *Proceedings of the 50th Hawaii international conference on system sciences*. 2017.

[2] Paolo Tripicchio et al. "Towards smart farming and sustainable agriculture with drones". In: *2015 International Conference on Intelligent Environments*. IEEE. 2015, pp. 140–143.

[3] Guoru Ding et al. "An amateur drone surveillance system based on the cognitive Internet of Things". In: *IEEE Communications Magazine* 56.1 (2018), pp. 29–35.

[4] Javier Irizarry, Masoud Gheisari, and Bruce N Walker. "Usability assessment of drone technology as safety inspection tools". In: *Journal of Information Technology in Construction (ITcon)* 17.12 (2012), pp. 194–212.

[5] Atheer L Salih et al. "Flight PID controller design for a UAV quadrotor". In: *Scientific research and essays* 5.23 (2010), pp. 3660–3667.

[6] PX4. *Multicopter PID Tuning Guide*. 2019. URL: `https://docs.px4.io/v1.9.0/en/config_mc/pid_tuning_guide_multicopter.html` (visited on 12/31/2019).

[7] Srikanth Govindarajan et al. "Design of Multicopter Test Bench". In: *International Journal of Modeling and Optimization* 3.3 (2013).

[8] M. Santos et al. "Experimental Validation of Quadrotors Angular Stability in a Gyroscopic Test Bench". In: *2018 22nd International Conference on System Theory, Control and Computing (ICSTCC)*. Oct. 2018, pp. 783–788.

[9] Eureka Dynamics. *FFT Gyro product page*. 2019. URL: `https://eurekadynamics.com/fft-gyro/` (visited on 10/09/2019).

[10] Samir Bouabdallah, Pierpaolo Murrieri, and Roland Siegwart. "Design and Control of an Indoor Micro Quadrotor". In: *IEEE International Conference on Robotics & Automation*. 2004.

[11] João Gutemberg B. Farias Filho et al. "Modeling, Test Benches and Identification of a Quadcopter". In: *2016 XIII Latin American Robotics Symposium and IV Brazilian Robotics Symposium (LARS/SBR)* (2016), pp. 49–54.

[12] Tayebi Abdelhamid and McGilvray Stephen. "Attitude Stabilization of a VTOL Quadrotor Aircraft". In: *IEEE Transactions on Control Systems Technology*. Vol. 14. 3. 2016.

[13] Yushu Yu and Xilun Ding. "A Quadrotor Test Bench for Six Degree of Freedom Flight". In: *Journal of Intelligent & Robotic Systems* (2012). Ed. by Springer.

[14] Fernando Dos Santos Barbosa. "4DOF Quadcopter: Development, modeling and control". MA thesis. São Paulo: Escola Politécnica of Universidade de São Paulo, 2017.

[15] Zhaohui Cen et al. "Robust Fault Diagnosis for Quadrotor UAVs Using Adaptive Thau Observer". In: *Journal of Intelligent & Robotic Systems* (2014). Ed. by Springer.

[16] Xunhua Dai et al. “Unified Simulation and Test Platform for Control Systems of Unmanned Vehicles”. In: *ArXiv* abs/1908.02704 (2019).

[17] Xi Chen and Liuping Wang. “Step Response Identification of a Quadcopter UAV Using Frequency-sampling Filters”. MA thesis. Melbourne, Australia: RMIT University, 2015.

[18] Gazebo. *Gazebo*. 2019. URL: http://www.gazebosim.org/ (visited on 12/24/2019).

[19] ROS. *ROS*. 2019. URL: https://www.ros.org (visited on 12/24/2019).

[20] Morgan Quigley et al. “ROS: an open-source Robot Operating System”. In: *ICRA 2009*. 2009.

[21] PX4. *PX4*. 2019. URL: https://px4.io/ (visited on 12/24/2019).

[22] L. Meier, D. Honegger, and M. Pollefeys. “PX4: A node-based multithreaded open source robotics framework for deeply embedded platforms”. In: *2015 IEEE International Conference on Robotics and Automation (ICRA)*. May 2015, pp. 6235–6240.

[23] PX4. *Controller Diagrams*. 2019. URL: https://dev.px4.io/v1.9.0/en/flight_stack/controller_diagrams.html (visited on 12/31/2019).

[24] J. G. Ziegler and N. B. Nichols. “Optimum Settings for Automatic Controllers”. In: *Transactions on ASME*. Vol. 64. 1942.