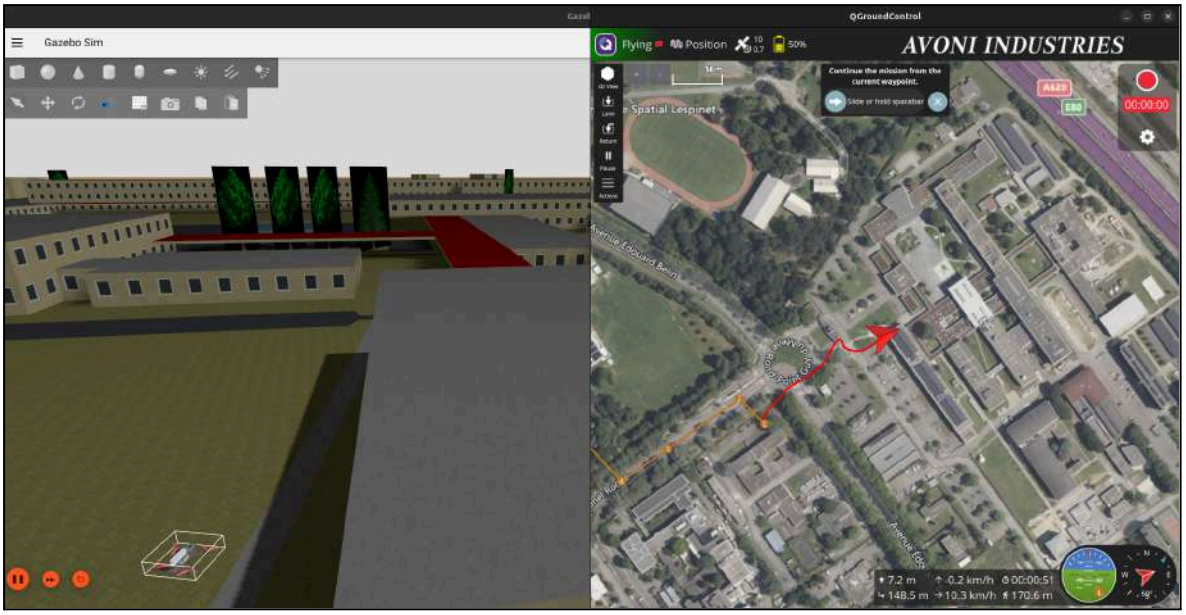
QAV 250

Software (SITL)

April 2026, Toulouse (FRANCE)

Document by Leonardo AVONI, avonileonardo@gmail.com

Part of the QAV250 Project



PX4 SITL autopilot at work. Gazebo (physics engine) is on the left, simulating the QAV250, while QGroundControl (on the right) does the ground station task.

1) Introduction

1.1) Disclaimer

Note: this document will not be as detailed as the previous, since informatic procedures are pretty much machine specific. Meaning that copy-pasting all the commands may not work on the first try.

Here I only list the main operations I did to make the setup work on my Linux machine; but you may need to adapt/modify some of it.

I did not detail the creation of the following:

- Aliases
- Desktop program icons (I like them, if they work well)

Do not hesitate in using ChatGPT, claude.ai, grok.ai, or any other LLM you like to help you get unstuck for informatic issues. And good luck!

1.2) Goal

Having designed then built the real QAV250 drone, the software is the only part missing to make it fly. This report sums-up the tests I did to gain on-hands experience with SITL (simulated flight) software, before tackling the real software that I would be pushing to the Pixhawk flight controller.

The “why” I did first SITL and then real-life was to alleviate a bit the learning curve. Now that the QAV250 is actually flying for real, I noticed that this was a good idea since batteries last for 10-15 minutes, which would quickly limit trial-and-error!

Since we always do more and more software, I decided to tackle everything I could without going in the companion computer part. My goals before going to real QAV250 software were

- Setting up PX4_SITL with Gazebo and QGroundControl
- Planning and executing a simulated mission
- Customize all SITL maps to Toulouse-area maps
- Customize the simulated drone as the one used by me (QAV250 specs)
- Set-up the RC (RadioMaster Pocket) to work with PX4_SITL
- Set-up an onboard camera (simulated)

From a “simulation” point-of-view I already downloaded Liftoff (Velocidrone as alternative), but it’s an FPV simulator. Nice for gettings the hands on, but not the real thing

Current setup is Dell Precision 7550; At the moment, I have Ubuntu 24.04.3 LTS

1.3) Definitions

- **Autopilot:** software that takes care of the flight of the drone. It takes control inputs from the RC, sensor information from the IMU, Gyro, barometer, GPS… and sends appropriate commands to the actuators. PX4 in our case
- **Actuators:** can be the propeller+motors (piloted by appropriate ESCs) and/or servomotors
- **Flight Controller (FC):** embedded hardware board that runs the autopilot software and interfaces with all sensors and actuators; it's Pixhawk 6C mini in our case. Most FCs already include IMU+Gyro+Compass+Barometer sensors.
- **Ground Station:** software running on a PC (e.g. QGroundControl) used to configure, monitor, and command the drone remotely
- **Companion Computer:** an onboard computer (e.g. Raspberry Pi) that runs higher-level tasks alongside the flight controller, communicating with it via MAVLink
- **Simulator, Physics Engine:** software that emulates the drone and its environment, allowing autopilot testing without real hardware; the physics engine models forces, motion, and sensor outputs
- **MAVLink:** lightweight communication protocol used to exchange messages between the autopilot, ground station, and companion computer
- **Flight Mode:** a predefined autopilot behavior (e.g. Manual, Stabilized, Mission) that determines how the drone responds to inputs and manages itself
- **RC:** Remote Controller — the handheld device used by the pilot to send manual control inputs to the drone
- **Attitude:** the drone's angular orientation in space, described by roll, pitch, and yaw
- **.osm:** OpenStreetMap file format storing geographic map data (roads, buildings, terrain) used to generate simulation environments
- **.sdf:** Simulation Description Format — XML file describing a robot or world model (links, joints, sensors, physics) for Gazebo
- **Mission:** a pre-planned sequence of waypoints and commands uploaded to the autopilot and executed autonomously
- **GPS:** Global Positioning System — provides absolute position (latitude, longitude, altitude) by receiving signals from satellites. The more satellites are seen by the sensor, the better precision and accuracy there is for the autopilot to use.
- **Barometer:** pressure sensor used to estimate altitude by measuring atmospheric pressure
- **IMU, Compass, Gyroscope:** the IMU (Inertial Measurement Unit) combines accelerometer and gyroscope data to measure linear accelerations and angular rates; the compass measures heading relative to magnetic north
- **SITL:** Software In The Loop — simulation mode where the autopilot software runs on a PC alongside a simulator, with no real hardware involved

- **Conda environment:** A Python environment (e.g., a Conda environment) is an isolated workspace with its own Python version and packages, preventing conflicts between projects.
- **EKF:** An EKF (Extended Kalman Filter) is an algorithm that estimates a system's state (e.g., position and velocity) by combining sensor data and prediction models.
- **Git:** version control system that tracks changes in code, allowing you to save snapshots, revert, and manage different versions of a project
- **GitHub:** online platform that hosts Git repositories, enabling code sharing, collaboration, and easy downloading of open-source projects (like PX4)
- **Kill switch / Arm-disarm switch:** RC switches that enable or cut motor power. Arming allows motors to spin; disarming stops them immediately. The kill switch is a hard override that cuts power instantly regardless of autopilot state.

Note about Kalman Filtering: most if not all signals coming from sensors have to be filtered. This is because onboard sensors (PS, IMU, Compass…) are not perfect, so the information they provide has some noise in them

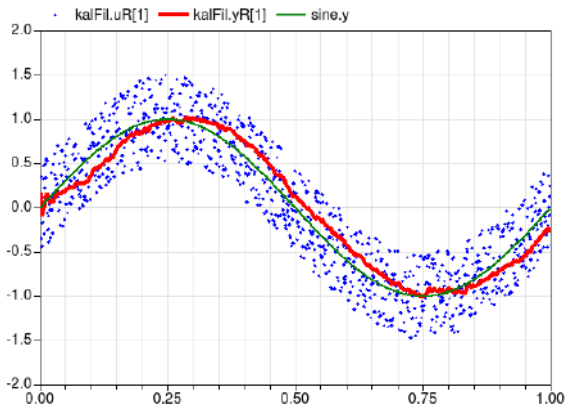


Figure 1: Kalman filter example, from simulationresearch.lbl.gov

For example: suppose the GPS provides XYZ position data as shown to the side with the blue points. Positioning the drone using only the blue points would be hard since the information changes much at each timestep.

An Kalman Filter helps converting the raw information (blue dots) in more accurate information (red curve), that is closer to the true drone position (green curve).

An Extended Kalman Filter mixes informations from several sensors, to provide even better match. In research, this field is called "Estimation" since it concerns ways of estimating real information from real-life sensors (prone to error)

1.4) Additional Resources

The following resources were consulted or identified as useful during this work.

General PX4 SITL setup:

- PX4 SITL setup tutorial: youtube.com/watch?v=pOTH5cZe4vs

OpenStreetMap data sources for custom Gazebo worlds:

- planet.openstreetmap.org — full planet dataset
- download.geofabrik.de — regional extracts
- openstreetmap.org/export — most intuitive; avoid exporting areas that are too large
- extract.bbbike.org — custom area extraction tool

Autonomous and intelligent UAV software (not covered for this project):

- Autonomous Systems Lab Datasets — ETH Zurich
- Intelligent Quads — tutorials and resources for autonomous quadrotor development (GitHub)

2.0) Setting Up PX4_SITL

To get started with PX4, Software-In-The-Loop (SITL), we need a decent computer (I have a Dell Precision 7550, running Ubuntu).

Then, it is good practice to create a folder in the Home repository, called src, and install all the softwares there. For some reason it's good to do that. I called mine src_PX4.

Having done so, you can install in that folder:

- A clone of the PX4 repository;
    - PX4 Autopilot runs on NuttX, Linux, and macOS, and may work on WSL.
    - Before installation, install dependencies with:
      `pip install -r Tools/setup/requirements.txt`
      from `~/src_PX4/PX4-Autopilot`. If using Conda, make sure the packages install into the correct environment (here: Python 3.13.5).
- An install of Gazebo (yes, other physics engines exist, but this is the more common for PX4)
- An install of QGroundControl; runs on Windows, MacOS, Ubuntu, Android

The method for getting started with PX4_SITL is described in the official docs.

2.1) Specific Goals of PX4, QGC and Gazebo

2.1) Role of PX4, QGC and Gazebo

The simulation stack is split across three components, each with a distinct responsibility:

- **PX4 SITL** runs the actual autopilot firmware — control loops, estimators, and control allocation — exactly as it would on real hardware.
- **QGroundControl (QGC)** is the ground station: it handles mission planning, parameter tuning, and telemetry display. It has no flight model of its own.
- **Gazebo** simulates the physical world: aerodynamics, motor thrust, drag, gravity, collisions, and sensor noise.

Two sets of parameters, two different purposes:

- **PX4 CA_* parameters (configured via QGC)** are used by PX4's control allocator (the "mixer"). They define how to decompose a desired force/torque into individual motor commands — e.g. "*I want +0.5 Nm of yaw torque: which motors spin up/down, and by how much?*" This is a mathematical inverse model inside the autopilot.
- **Gazebo SDF motor plugin parameters** (motorConstant, momentConstant, rotorDragCoefficient, …) are used by Gazebo's physics engine. They define what physically happens when a motor spins at a given speed — e.g. "*Motor 3 is commanded at 800 rad/s: what force and torque does it actually produce?*" This is a forward physics model.

The full control loop is therefore:

```
PX4 Controller (desired roll/pitch/yaw/thrust)
   ↓
CA parameters → per-motor PWM/rad/s commands
   ↓
Gazebo SDF motor plugin → forces & torques on airframe
   ↓
Gazebo Physics Engine → simulated IMU, GPS, baro readings
   ↓
PX4 Estimator (EKF2)  ← loop closes here
```

In short: **PX4/QGC parameters drive the controllers; SDF parameters drive the Gazebo physics simulation.**

2.1) Quick Start with PX4

The PX4 documentation is a great way to start understanding what PX4 does and how it work. It's best if you read them rather than me trying to summarise them. Moreover, the common startup method is also detailed from the documentation.

Cloning the repository is done using the following command:

```
git clone https://github.com/PX4/PX4-Autopilot.git --recursive
bash ./PX4-Autopilot/Tools/setup/ubuntu.sh
reboot
```

Having successfully completed the previous commands, one can launch the (default) SITL simulation:

```
make px4_sitl gz_x500
```

That command launches a simulation (SITL) using PX4, and uses the x500 model, using gazebo (gz) as simulator.

Then, it is essential to open QGroundControl, since SITL simulation cannot run without QGC, as the ground control is needed to understand where the drone is and what it should be doing.

A typical view for PX4 SITL is shown in Fig.

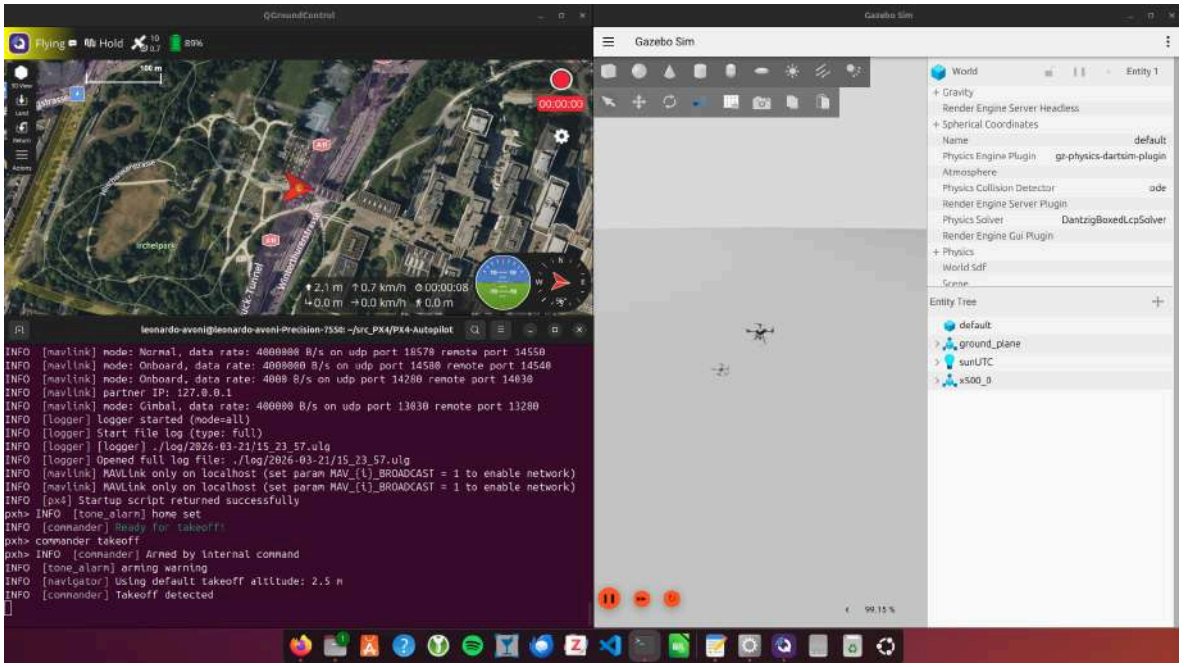


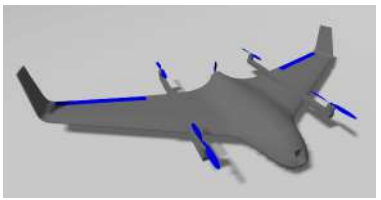
Figure 2: Screenshot example of the typical PX4_SITL view: QGC on the top left, PX4 terminal on bottom left, Gazebo on the right

Once SITL simulation is done, one can close QGroundControl and PX4. To kill all processes, type the following:

```
pkill -f px4
pkill -f gz
pkill -f gazebo
```

Note 1: the SITL simulation was started with the x500 model, however, input arguments depend on the vehicle and/or world we want to launch.

The PX4 software comes with a variety of default vehicles. Among the aerial vehicles, we find the following common models, shown below

		
gz_x500 (quadcopter)	gz_rc_cessna (fixed wing)	gz_standard_vtol (VTOL)
Figure 3: Examples of three flying vehicles already included in PX4		

There are tons of models available; and can be found in the PX4-Autopilot/Tools/simulation/gz/models module of PX4, with different physics methods, different onboard sensors… find what best suits you.

Note 2: the “make” command does several things

- compiles px4, if it’s not already compiled
- launches the PX4_sitl simulation

If changes are done to the software, one has to do make clean, then make px4_sitl gz_x500

Note 3: GitHub stores PX4 code and its version history. If you need a specific PX4 release, make sure to clone that exact version rather than the latest one.

Note 4: on Gazebo, I advise using the Follow mode to force the camera to follow the drone movements. To do so, right click on the drone and select Follow.

2.2) Quick Start with QGC

As with PX4, QGroundControl also comes with a detailed documentation. To be honest, the presente of a Graphical User Interface helps a lot so the doc is not as essential as for PX4. For quick start, the following images are provided so that you are not too lost. For more details, refer to the doc.

About mission design, the generic pipeline is: you first design a mission in QGroundControl Mission Planning Window, then you upload it to the vehicle. During vehicle operation, you can then decide to start the mission

During Mission creation, you also need to define a GeoFence, being a limit you are not supposed to be breaching with the drone.

- In manual flight, as soon as the fence is breached, the vehicle autonomously goes back to the fence limit, keeps altitude and Hold.
- In autonomous flight, using QGC commands, PX4 stops you beforehand and sends you a warning, indicating that current command breaches the GeoFence.
- In autonomous flight, during mission planning, the mission will not be uploaded and a warning will be sent

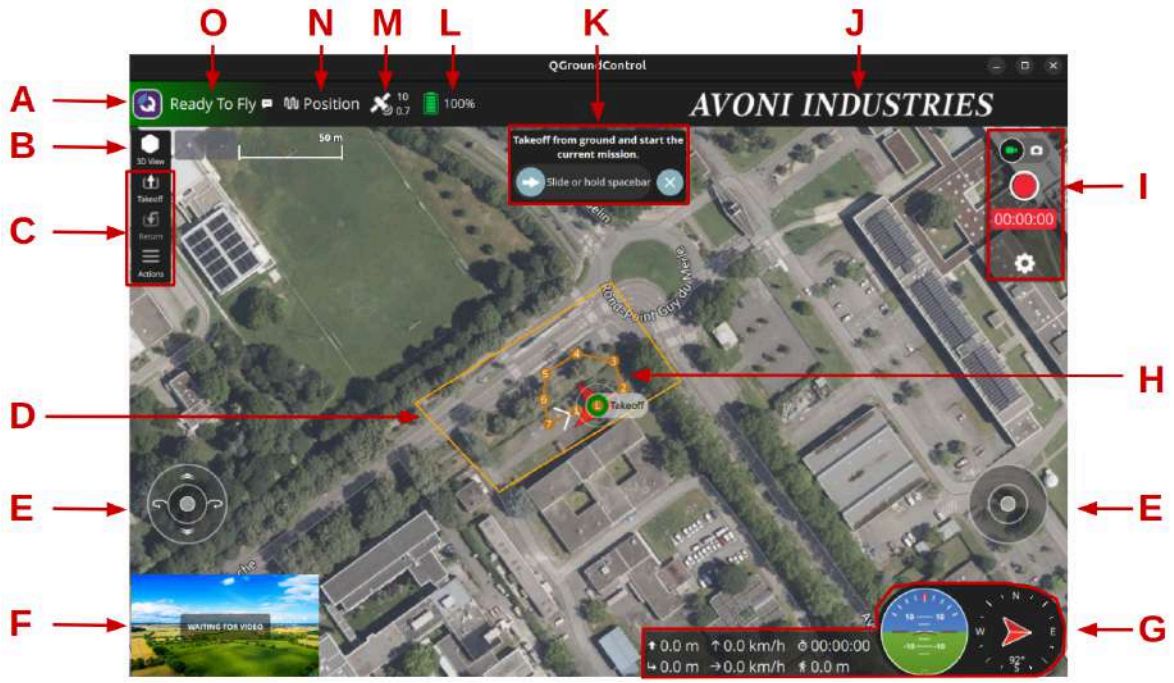


Figure 4: Main QGroundControl screen. Elements A to O are described in Table 1

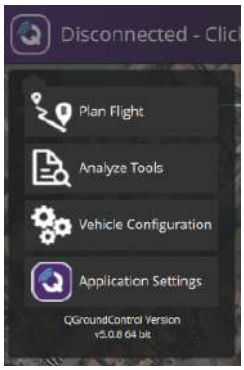


Figure 5: More menus, under QGC top left icon

QGroundControl Q icon (top left) hides four sub-menus

- Plan Flight: used to plan missions
- Analyze Tools: used to debug drone behavior
- Vehicle Configuration: used to set-up onboard PX4 software, do calibrations…
- Application Settings: self-explanatory

Table 1: Description of the various elements in Figure 4			
A	QGC top left icon (hides the menus of Fig. XX2)	**I**	Record Window
B	3D View toggle. Allows to view a 3D rendering of buildings and drone	**J**	Logo (customizable)
C	Command elements. Help directing the drone during flight	**K**	Action Slider (pops-up as a safety, to confirm user actions)
D	GeoFence (not to be exceeded during drone flight)	**L**	Drone Battery Indicator
E	Joysticks (to pilot the drone)	**M**	Satellite count, for GPS sensor
F	Video feed (if any)	**N**	Flight Mode Indicator
G	Compass and Flight Telemetry	**O**	Drone Status
H	Flight Path		

Figure 6: Mission Planning QGroundControl screen. Elements 1 to 9 are described in Table YY

Table 2: Description of the various elements in Figure 6			
1	Mission design tools	**6**	GeoFence setup
2	GeoFence (needed to start planning)	**7**	Mission setup
3	Mission Summary	**8**	Planned Mission
4	Various mission waypoints	**9**	Upload Button (after a mission is created, it must be uploaded to the onboard software)
5	Rally Points Setup (additional loiter/landing points)		

3) Commanding the Drone

Having understood how to start the SITL simulation, we have to be able to control the drone. To do so, there are basically two categories on how to do so:

- Autonomous flight; aka we let the software control entirely how the drone behaves
- Manual flight: we use the joystick (RC connected to the computer) to move the drone around. This part will be covered in Section XX.

For the autonomous flight, there are at least two possibilities with current setup:

- Either we send commands from the PX4 shell to tell the drone what to do (refer to Table 3)
- Or we tell the drone the actions to do from the QGC screen. For example, we click on a point on the map, and tell the drone to go there; or we tell the drone to land…

While the second method is intuitive using QGC Graphical User Interface, the commands for the first method must be told, as is done below:

3.1) PX4 Terminal Commands

Table 3: Various PX4 shell commands (non-exhaustive list)	
`help`	List all available commands.
`commander status`	Current state (armed, mode, health).
`commander arm /` `commander disarm`	Arm → enables motors Disarm → stops motors
`commander land`	Auto land at current position.

`commander safety on`	Enable safety (motors blocked).
`commander stop`	Stops commander module (⚠️ rarely used).
`navigator`	Handles autonomous flight
`uorb top`	Shows active topics + message rates (not contents).
`param show`	List all parameters.
`mavlink status`	Shows connections (QGC, simulator, etc.).
`<module> start / stop / status`	Manage PX4 modules (start, stop, check status). 👉 Used for debugging internal components.
`commander check`	Run preflight checks (sensors, GPS, battery).
`commander takeoff`	Auto takeoff to default altitude.
`commander mode position`	Switch to Position mode (GPS/local position hold).
`commander poweroff`	Shutdown vehicle (in SITL: stops simulation)
`commander transition`	VTOL only: switch between multicopter ↔ fixed-wing.
`listener vehicle_local_position`	Shows position (x, y, z, velocity).
`listener -n 1 <topic>`	Shows <topic>, but a single print
`param show SIM_GZ_EC_FUNC3`	Show a specific parameter

3.2) PX4 Multicopter Flight Modes

The general way for commanding the drone flight is to make it behave according to a selected flight mode:

- For manual flight mode (in blue in the table below), RC commands are interpreted by the autopilot to determine the drone flight
- For autonomous flight modes (in purple in table below), drone flight is determined autonomously according to a certain mission, or according to PX4 terminal inputs, or MAVLink messages

The following table sums-up flight modes for Multicopter drones. The detailed description is [available in the docs.]

Table 4: Various PX4 Multicopter flight modes. Some modes require GPS signal, other IMU attitude stabilization, other a Barometer altitude indication.
Manual flight modes are indicated in blue, autonomous flight modes are indicated in purple.

Mode Name	Requirements:			Description
	GPS?	IMU?	Barometer?	
Position	✓	✓	✓	Holds position, moves according to operator
Position Slow	✓	✓	✓	Same as Position, but slower
Altitude (AltCtl)	✗	✓	✓	Maintains altitude and attitude; can drift due to IMU drift, wind, perturbations…
Altitude Cruise	✗	✓	✓	Maintains altitude, but also maintains non-zero attitude when the RC sticks are released
Stabilized (or Manual)	✗	✓	✗	Same as Altitude, but altitude is not controlled anymore
Acro	✗	✗	✗	the RC commands not the attitude, but the angular velocities of the drone. Hard to fly without training
Hold	✓	✓	✓	Holds current position in the sky. Compensates for perturbations
Return (RTL)	✓	✓	✓	Returns and lands to a launch point
Takeoff	✓	✓	✓	Takes off from current point
Land	✓	✓	✓	Lands at current position
Orbit	✓	✓	✓	Performs orbits (circles) around current position
Follow Me	✓	✓	✓	Follows a certain target position, specified via MAVLink
Offboard	✓	✓	✓	Delegates drone control to a companion computer

4) Custom World Setup

After working for a bit with the default map (centered in Zurich) and the default gz_x500 drone, the next step was understanding how to do the following

- Map building in QGC, 2D view
- Map building in QGC, 3D view
- Map building in Gazebo

In all three cases, I wanted a simulation world based in the ISAE-Supaero/ENAC region of Toulouse (France).
The reason for that was not to do something fancy, or to flex about France (not particularly relevant) but most importantly because ENAC and ISAE-Supaero is the neighbourhood I currently live and work in, hence I know what it looks like and I have a physical grasp of how far things are with respect to one another.
Meaning that I can better understand the drone position, and for example “how fast it’s going” because when doing so, I will basically be thinking “hmm yes it’s going slightly faster than me on my bike, or ok it’s more or less going walking speed…” Having a desert map does not give you that insight

Of course, I recommend you trying to build a map correspondent to an area you are familiar with.

3.1) Custom World, QGC Settings

Before we do anything, we specify also the default position; so that we will be able to locate the drone at startup around Toulouse area GPS
To change the default location:

- On Gazebo: check on the right the “Spherical Coordinates” tab, and set the gps coordinates you most like.
- You can navigate to the `PX4-Autopilot/Tools/simulation/gz/worlds/default.sdf` world that you are using (it’s a file that lists all the specs of the world you are using)

Having this, we have a default gazebo world (the default, empty, simulation environment, BUT with the updated location in terms of GPS coordinates (in my case 43.564905, 1.478764)

For QGC, the maps are updated automatically since QGC uses the web to download local maps automatically. HOWEVER, the 3D items, like buildings are not automatically updated. This can be easily done by downloading an .osm file from https://www.openstreetmap.org/export and specify it in the imports of QGC (Q icon on top

left corner -> Application Settings -> Fly View -> 3D View): toggle “Enable, fill-in the path for the selected .osm file and you are good

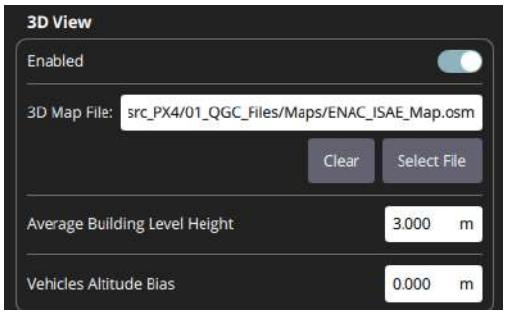


Figure 7.1: Selector within QGC parameters for custom osm selection

Figure 7.2: Resulting 3D view in QGC. Notice the presence of 3D buildings in the background

3.2) Custom World, Gazebo Settings

To obtain a custom world for Gazebo to use, different from the default.sdf world,, things are more tricky. In fact, adding geographic features to a model can reveal challenges:

- **Method A:** One can try to extract google maps files and images, and recreate a realistic videogame-grade world. This method uses blender and takes forever to do. Moreover, as I discovered later, the heavier the world, the harder Gazebo takes to run it
- **Method B:** Another method is a simplification of method A, where we combine the osm file from OpenStreetMap, as well as an elevation .tiff image and a 2D map of the area. Assembly of the three elements had to be done in Blender. This seemed like a good tradeoff but it was not; Blender was crashing a lot (maybe it’s my Ubuntu blender install, maybe it’s something else. When I tried it, I used a combination of BlenderGIS, OSM .osm map, and OpenTopography tif depthmap, from https://portal.opentopography.org/rasterOutput?jobId=rt1774696922651 . Still, I never managed to export something from Blender; and was already spending too much time
- **Method C:** the most “automatizable” and quick: there is a tool called OSM2World. Having downloaded it on Ubuntu, I fed-it the previously extracted ISAE-Supaer/ENAC .osm file, and after a few seconds, the program showed a finished look of the ISAE + ENAC campus. Yes, the colors are messed up, but it’s because of how fast it took to generate this. Moreover, a shady world is better than no world

Note: the .osm file (OpenStreetMap file) contains informations about streets, railroads, highways, buildings, rivers… that may be present in real life.

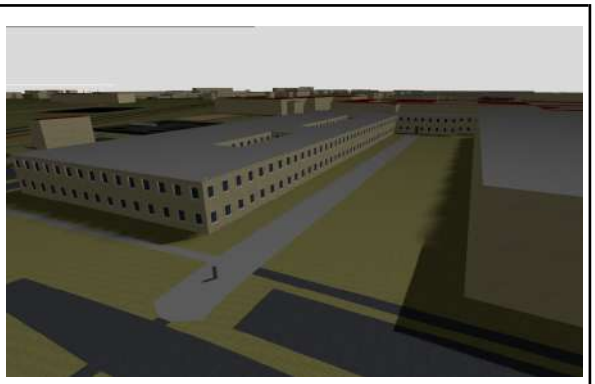
Figure 8: Typical view of OSM2World, to be used for Method C

Once satisfied with the model, I exported it as an `.obj` file for use in Gazebo. Three issues were addressed during the process:

- **Ground plane:** Added a separate ground plane 0.5m below the OSM-generated model (in orange), ensuring the drone always has a solid surface to land on even when operating far from the ISAE-ENAC area.
- **Axis alignment:** Verified that the OSM2World-generated map uses the correct axis convention (Z-up vs Y-up) expected by Gazebo.
- **GPS consistency:** Confirmed that the same GPS location is displayed in both QGC and Gazebo.

Note: this is not necessarily an optimized procedure. In particular, spending more time in OSM2World maybe can lead to better views; and maybe obj is not the most eGazebo-efficient file format. Still, for the moment it worlds (but those are things that can be improved).

Google Maps View



Gazebo view, using OSM2World

Figure 9: ISAE Building at 43.56609833025355, 1.4743481856831777, seen in Google Maps and in the OpenStreetMap -> OSM2World -> Gazebo

With OSM2World, dimensions are not the most accurate. Namely, the two buildings in the figure should be having the same height; they do not through OSM2World (7.5m in OSM2World against 11m in real life). For the moment this is acceptable since our main concern is the drone, but upgrades/edits in the world-to-gazebo pipeline should be made in the future if more realistic data is seeked.

Also: note that current Gazebo model is heavier than default empty model. Maybe loading a more HD model would make its loading time even higher; but maybe there are options to combine both OSM data and photogrammetry data (automatically I mean) that provide an .obj map for Gazebo that is more defined than current one.

5) Creating a Custom Vehicle

5.1) Initial Model Setup

Note: All files described below were created by copying the equivalent `gz_x500` files, then modifying and renaming them accordingly.

Setting up a custom vehicle for Gazebo requires two things on the PX4 side: an **airframe file** and a **model folder**.

Airframe file Create a file named `4030_gz_qav250_leo` in:
`PX4-Autopilot/ROMFS/px4fmu_common/init.d-posix/airframes/`

This file defines the vehicle's generic settings and its default world (here changed to ENAC_ISAE). Also add the filename to the CMakeLists.txt in the same directory, then recompile PX4 to apply the changes:

```
cd ~/src_PX4/PX4-Autopilot
make clean && make px4_sitl
```

Model folder Recompiling alone is not enough — you also need to define the physical model. Create a folder named `qav250_leo` in:
`PX4-Autopilot/Tools/simulation/gz/models/`

`organised as follows:`

```
├── model.config
├── model.sdf
├── materials/
│   └── textures/          # texture images (.png)
```

```
├── meshes/                  # 3D geometry files (.stl, .dae)
└── thumbnails/              # preview images
```

`model.config` and `model.sdf` describe the model, while the `meshes/` folder holds the 3D geometry files. The result is shown in Figure 10.

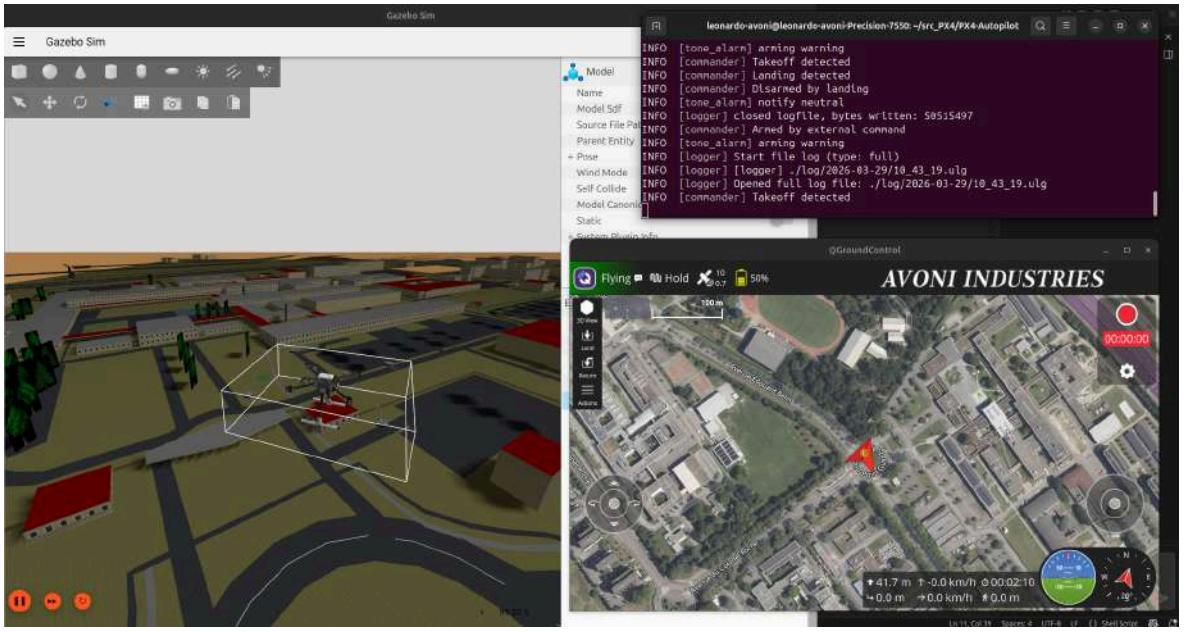


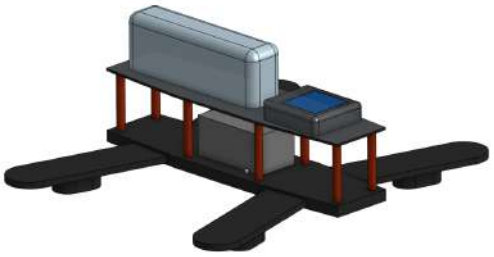
Figure 10: PX4_SITL + Gazebo + QGC. Yes, I also took the time to create my own logo, that we can easily include in QGC from the Application Settings. Note how the drone propellers were actually moved from the frame

5.2) Model Customization

The drone model .sdf file is composed of a connected group of meshes. There are for example the frame mesh, and the various propellers mesh. In the example shown above, I managed to move the center of rotation of the propellers away from the frame. Of course it's not physical, but it can be done.
The current model is rigid, I don't know if flexible Gazebo models can be made.

I decided to call the model `gz_qav250_leo` and start customizing it for it to resemble the QAV250 drone.

To do so,I changed mass and inertia properties defined inside the .sdf file.
Then I created custom .stl models for both the frame and the propellers

	
QAV250 drone model (without motors). Model available on OnShape	Tri-blade 5.1'' propeller, design rescaled from this design; files for CW and CCW propellers are here
Figure 11: geometries used for the QAV250 drone model	

After assembling the new propellers and new frame together, I obtained the following view in Gazebo

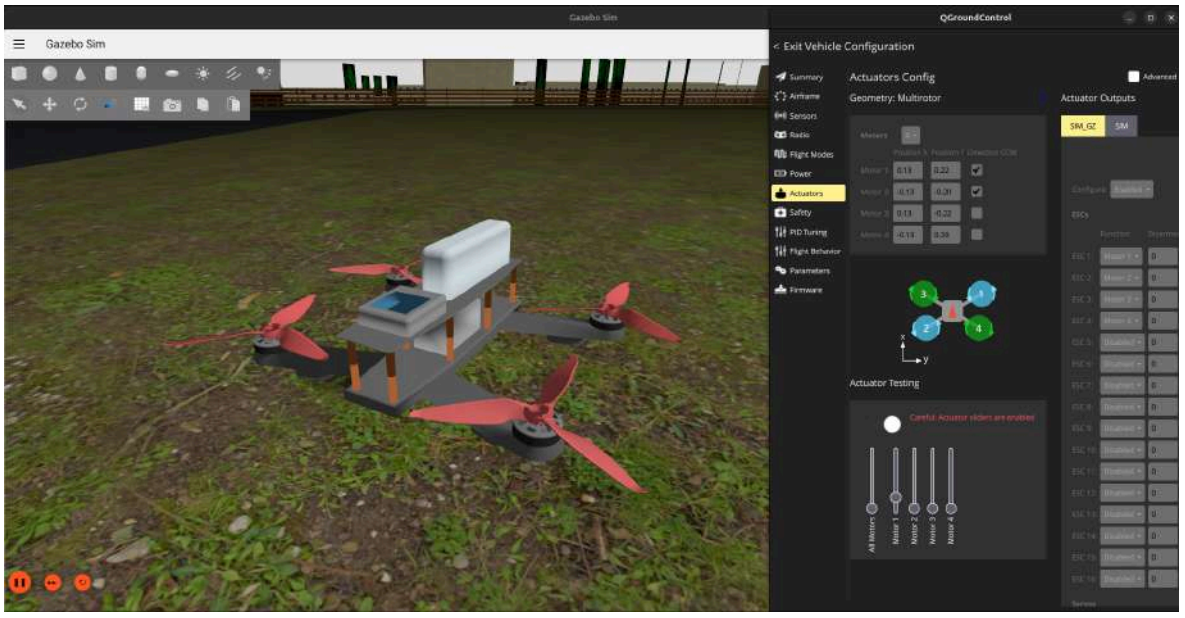


Figure 12: Gazebo view of the simulated QAV250 drone. On the right is the QGC Vehicle Configuration view

Note: there is a VERY IMPORTANT difference between the model used in Gazebo, and the setup configuration to be done in PX4.
PX4 configuration can be accessed from terminal line commands in the pxh> shell, but it is just easier to modify things from QGC -> Vehicle Configuration window.
The idea, as explained in Section XX, is the following:

- The Gazebo model is there to simulate the behavior of the drone in the external environment.
- The PX4 setup is there for the onboard controllers and flight planning to use.

Modifying the drone controller parameters can be done inside QGC -> Vehicle Settings -> PID Tuning. Changing the PID parameters leads to different drone performance, for example the drone getting more oscillations. An example of the QGC PID window is shown in Figure 13.

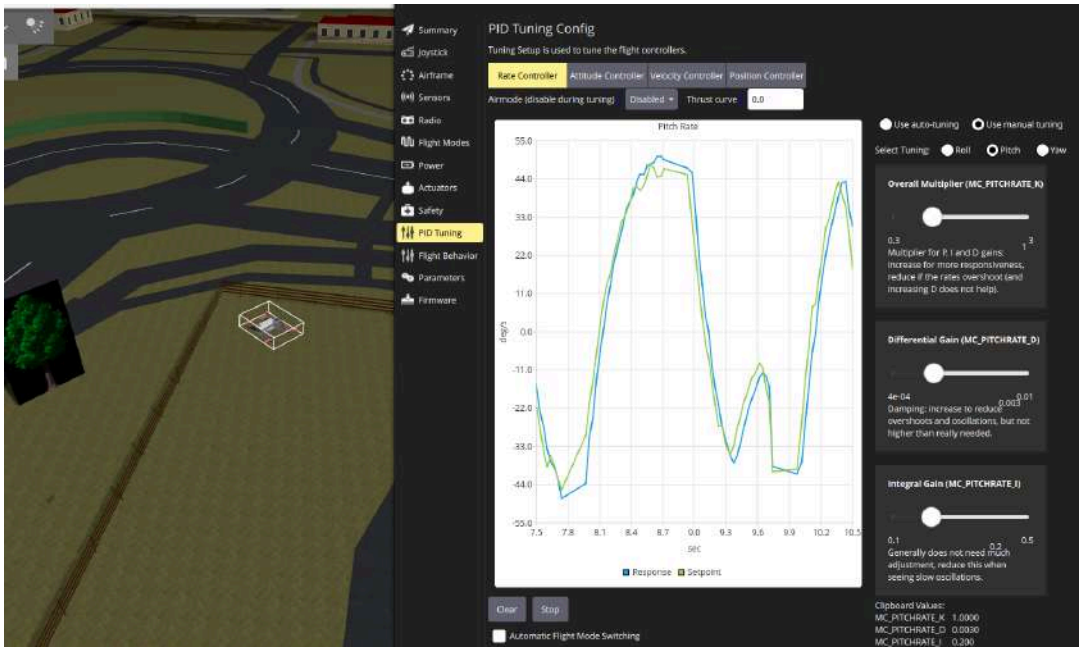


Figure 13: Gazebo view of the simulated QAV250 drone. On the right is the QGC Vehicle Configuration view

5.3) Issues and Debugging

When building the custom QAV250 model, I noticed the following issues:

- Once the model was done, and flight was attempted, the drone seemed to shake quite a bit. Reducing the motor constants in the .sdf model seemed to solve the issue. This is reasonable since initial sdf file was taken from x500 drone, which is larger and with more powerful motors. It is possible that excessive motor power destabilizes the system.
- I had a lot of trouble trying to set-up the switches of the RC too perform PX4 tasks (arm, disarm, flight mode toggle…). More details in Section XX.
- Drone orientation was set according to the real build, hence Yaw 180° (because of the flight controller position). This meant building the sdf model accordingly, performing proper motor allocation in QGC -> Vehicle Configuration -> Actuators. However, using Yaw 180° also led to Gazebo view to be oriented towards the drone rear all the time. Since this is just simulation, I decided to revert and use Yaw 0° instead (default)

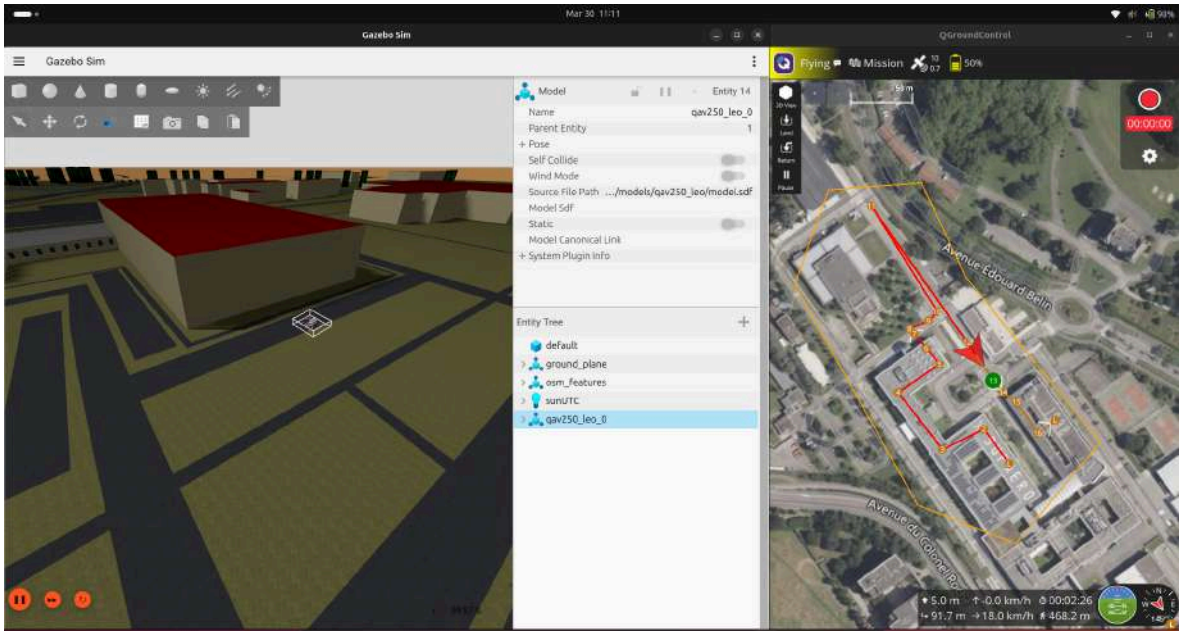


Figure 14: PX4_SITL + Gazebo with a mission planning ongoing. The orange contour is the geofence (if the drone exceeds it it goes in failsafe)

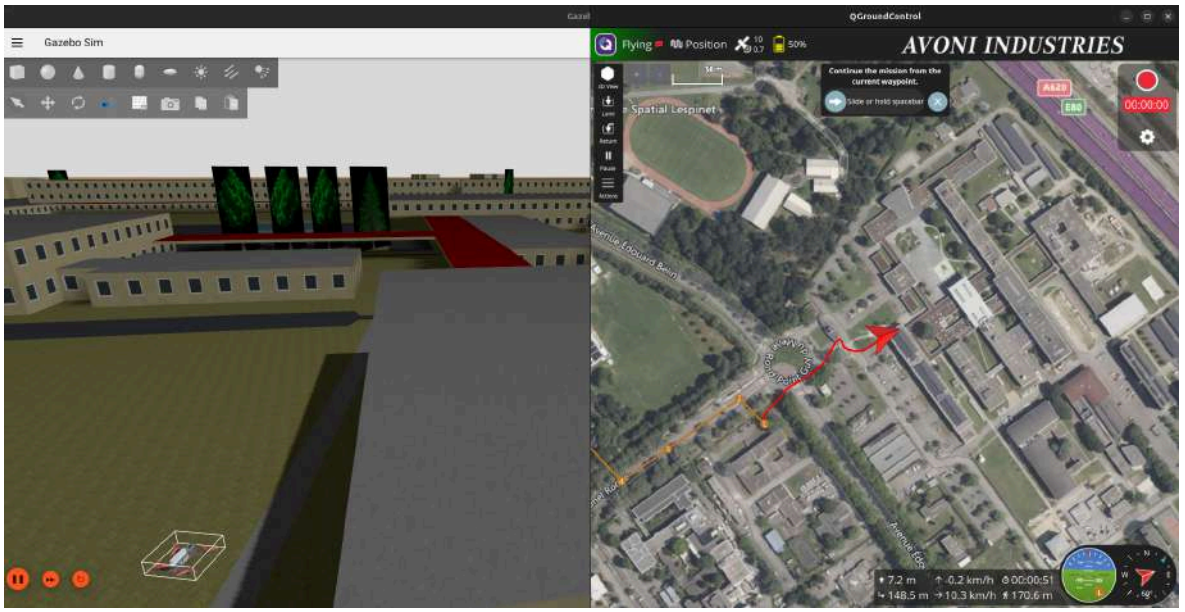


Figure 15: Drone flying forward, with Gazebo view also looking forward. I guess the gazebo views corresponds to some extent to the vehicle orientation

6) RC SITL Control

6.1) RC Setup for SITL

While at the end of this report, RC control was actually core of the overall PX4_SITL experience. In fact, an RC is an excellent was to send directly drone commands throughout drone flight.

On my case, I was working with the RadioMaster Pocket Crush ELRS, and had quite a series of issues, summarized here. More details on the RC can be found here, and setup guide can be found here.

The RC comes with two USB-C ports. The lower one is for charging, the upper one is to connect to the computer via USB. The RC must have charged batteries to operate properly.

I had already setup the RC to use for the Liftoff flight simulator. The idea here was connecting it to the PX4_SITL+QGC+Gazebo. Note that doing so is not strictly the same as using the RC with the drone and ground station in real life:

- For SITL, RC sends commands to QGroundControl, which sends the requested drone movement to the drone via MAVLink
- For real-life drone, RC sends commands to the drone directly, without passing by QGC; which knows drone position and telemetry through MAVLink.

Since SITL simulation runs on the same computer, the difference compared to real RC behavior is not huge; apart for the fact that the RC input is interpreted in the QGC -> Vehicle Configuration -> Joystick instead of QGC -> Vehicle Configuration -> Radio.

However, if one were to fly the real drone by sending commands from a QGC Joystick (a second RC, or the on-screen joysticks from Figure XX) range limitation can happen, as well as delay and loss of signal.

The following issues were found when trying to use the RadioMaster Pocket Crush as Joystick for QGC (Ubuntu setup)

- Missed proper recognition of the left vertical stick (typically the throttle), similar to here. Custom definition of Ubuntu Joystick input had to be done to fix the problem.
- Missed recognition of switches of the RC as Joystick buttons. This was identified to also be due to missing USB configuration settings for such a small RC. Ideally, I would have liked to configure an arm/disarm switch, a hold position switch, a killswitch and a mode selection switch. However I was reaching the limit of EdgeTX RC software, QGroundControl and Ubuntu, so I let it go.
- I also tried to update EdgeTX firmware to check if more USB options were available, but the situation stayed the same.

Since a lot of back-and-forward was done to test the RC as joystick, it was convenient to have a terminal line tool for testing, hence jstest.

6.2) EdgeTX Firmware Update

I updated EdgeTX using the **Bootloader method** (connecting via USB, no SD card removal needed), following the official EdgeTX manual.

I downloaded `bw128x64.zip`, the correct firmware for the RadioMaster Pocket, and updated from version 2.8 to **2.12**. The current version can be checked from the text file in the SD card's home folder.

Note: `edgetx-cpn-linux-v2.12.0.zip` (also available for Windows and macOS) allows you to simulate your radio on a PC — not covered here.

7) MAVLink and Camera+Gimbal

7.1) MAVLink and MAVSDK

MAVSDK is a Python SDK that lets you control PX4 drones (or any MAVLink vehicle) programmatically.

```
pip install mavsdk
pip install aioconsole
```

MAVSDK-Python is built around `asyncio`, meaning most commands use `await`. The standard Python shell does not support `await` directly — `aioconsole` provides an async-friendly shell (`apython`) that does.

Basic usage example:

```
from mavsdk import System
drone = System()
await drone.connect()      # INFO [mavlink] partner IP: 127.0.0.1
await drone.action.arm()
await drone.action.takeoff()
await drone.action.land()
await drone.action.disarm()
```

MAVSDK exposes an object-oriented API on top of MAVLink, allowing commands such as fetching telemetry, executing missions, holding altitude, or navigating to a GPS location. More examples are available at the [MAVSDK-Python repository](#).

A template script grouping several commands together (for demonstration purposes) is available [here](#).

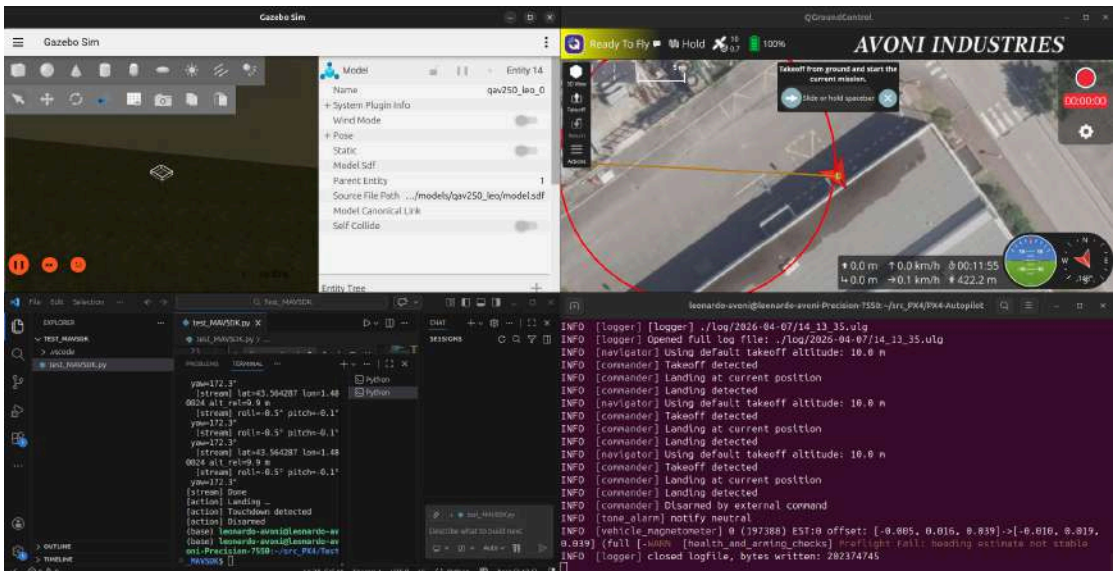


Figure 16: MAVSDK python seetup: top line contains Gazebo and QGC; bottom line shows the terminal and a VSCode Python script running

7.2) Camera and Gimbal

Before moving forward, I wanted to try to place some kind of camera on the QAV250; and while I was at it I thought it was a good idea to add a gimbal.
The motivation behind it is that it helps the camera to always point in the proper direction, even though the drone has to change its attitude to move around. Some examples in Figure 17.

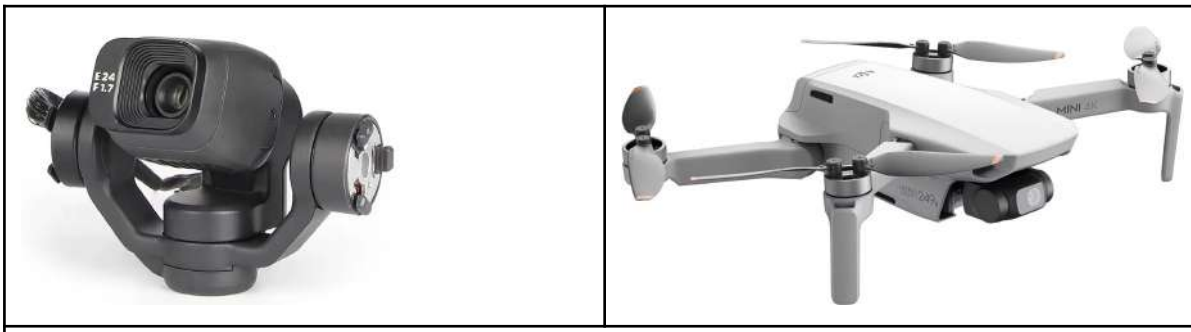


Figure 17: Typical DJI gimbal and drone. Credits: DJI

Instead of coding everything from zero, PX4's gazebo modules includes the `x500_gimbal`; a version of the x500 drone that includes a camera gimbal.

Concerning the camera feed from Gazebo to QGC, it can be set-up via QGC -> Application Settings -> Video -> Source: UDP h264 video stream. The model and QGC view are shown below.

Figure 18: x500_gimbal PX4+Gazebo model

Figure 19: Working video stream in QGC, of the view from Gazebo

Even though I managed to include the camera + gimbal in qgc, there is still work to do on the stabilization of the gimbal: at the moment it does not self-stabilize it.

Moreover, at the moment, the gimbal "can" be controlled by "click-to-point" but it moves in an unexpected way. Probably the euler angles mapping is not made properly.

We consider for the moment that as a first software implementation, it is good enough. Now we try to flash real drone software.

Note: current SITL gimbal camera implementation takes the video feed from Gazebo and sends it to QGC through UDP communication; which is ok since everything takes place on the same computer. In reality, video has to go from the drone to the QGC on the ground control station (computer). This part is generally addressed via VTx/VRx elements.

8) Conclusion

We here conclude this first part about autopilot softwares. Currently, we managed to understand the following points:

- Workflow of PX4+QGC+Gazebo
- Addition of MAVLink via MAVDSK
- Joystick commands, either from RC or from QGC on-screen sticks
- Mission planning
- PX4 pxh> commands
- Various flight modes behavior (quadcopter)
- Custom Gazebo World, according to real-life specs
- Custom Gazebo drone model, according to real-life specs
- Custom drone PX4 setup and simulated flight
- Camera integration

In the future, the following points can be addressed

1. Fixed-Wing model setup
2. MissionPlanner + ArduPilot SITL
3. Flightgear simulator (alternative to Gazebo, but with less support)
4. Companion Computer setup
5. Better gimbal behavior
6. ROS2 integration
7. Computer vision