Robotics Lab: Homework 3

Fly your drone

Antonio Polito: https://github.com/P0l1702/
Maria Rosaria Imperato: https://github.com/MariaRosaria1
Gianmarco Ferrara: https://github.com/gianmarco-ferrara

This document contains the Homework 3 of the Robotics Lab class.

Fly your drone

1. **Build your custom multi-rotor UAV**

   (a) In order to simulate our custom UAV, a Neapolitan hexacopter called `pizza_drone`, we created the following:

      i. a folder named `pizza_drone` (located in the `PX4-Autopilot/Tools/simulation/gz/models/` folder) containing the model description, in particular:

      - the `model.sdf` file, defining the physical structure (links, inertia, sticks, ...), collision properties, sensors, and plugins;

```
<?xml version="1.0" encoding="UTF-8"?>
<sdf version='1.9'>
  <model name='pizza_drone'>
    <pose>0 0 .24 0 0 0</pose>
    <self_collide>false</self_collide>
    <static>false</static>

    <link name="base_link">
      <inertial>
        <mass>2.0</mass>
        <inertia>
          <ixx>0.15</ixx>
          <ixy>0</ixy>
          <ixz>0</ixz>
          <iyy>0.15</iyy>
          <iyz>0</iyz>
          <izz>0.25</izz>
        </inertia>
      </inertial>
      <gravity>true</gravity>
      <velocity_decay/>

      <visual name="base_link_visual">
        <pose>0 0 0 0 0 0</pose>
        <geometry>
          <cylinder>
            <radius>0.5</radius>
            <length>0.09</length>
          </cylinder>
        </geometry>
        <material>
          <ambient>1.0 0.0 0.0 1.0</ambient>
          <diffuse>1.0 0.0 0.0 1.0</diffuse>
          <specular>0.5 0.5 0.5 1.0</specular>
        </material>
      </visual>

      <collision name="base_link_collision">
        <pose>0 0 0 0 0 0</pose>
        <geometry>
          <cylinder>
```

```
        <radius>0.5</radius>
        <length>0.09</length>
      </cylinder>
    </geometry>
  </collision>

<visual name="leg_1_visual">
    <pose>0.15 -0.15 -0.12 0 0 0</pose>
    <geometry>
      <cylinder><radius>0.015</radius><length>0.24</length></cylinder>
    </geometry>
    <material>
        <diffuse>0.35 0.6 0.35 1.0</diffuse>
    </material>
  </visual>
  <collision name="leg_1_collision">
    <pose>0.15 -0.15 -0.12 0 0 0</pose>
    <geometry>
      <cylinder><radius>0.015</radius><length>0.24</length></cylinder>
    </geometry>
  </collision>

                                        ...

<link name="rotor_0">
  <pose>0.39 -0.225 0.06 0 0 0</pose>
  <inertial>
  <mass>0.016</mass>
  <inertia>
  <ixx>3.84e-07</ixx>
  <iyy>2.61e-05</iyy>
  <izz>2.64e-05</izz>
  </inertia></inertial>
  <visual name="rotor_0_visual">
    <pose>-0.022 -0.146 -0.016 0 0 0</pose>
    <geometry><mesh><scale>0.85 0.85 0.85</scale>
    <uri>model://x500_base/meshes/1345_prop_ccw.stl</uri>
    </mesh></geometry>
    <material><diffuse>1 1 1 1</diffuse></material> </visual>
  <collision name="rotor_0_collision"><geometry>
  <cylinder>
  <radius>0.16</radius>
  <length>0.01</length>
  </cylinder>
  </geometry>
  </collision>
</link>
<joint name="rotor_0_joint" type="revolute">
<parent>base_link</parent>
<child>rotor_0</child><axis><xyz>0 0 1</xyz><limit>
<lower>-1e16</lower>
<upper>1e16</upper>
</limit></axis></joint>

                                        ...
```

```
    <plugin filename="gz-sim-multicopter-motor-model-system"
    name="gz::sim::systems::MulticopterMotorModel">
      <jointName>rotor_0_joint</jointName><linkName>rotor_0</linkName>
      <turningDirection>ccw</turningDirection>
      <timeConstantUp>0.0125</timeConstantUp>
      <timeConstantDown>0.025</timeConstantDown>
      <maxRotVelocity>1000.0</maxRotVelocity>
      <motorConstant>8.54858e-06</motorConstant>
      <momentConstant>0.016</momentConstant>
      <commandSubTopic>command/motor_speed</commandSubTopic>
      <motorNumber>0</motorNumber>
      <rotorDragCoefficient>8.06428e-05</rotorDragCoefficient>
      <rollingMomentCoefficient>1e-06</rollingMomentCoefficient>
      <rotorVelocitySlowdownSim>10</rotorVelocitySlowdownSim>
      <motorType>velocity</motorType>
    </plugin>

                                                    ...
  </model>
</sdf>
```

- the `model.config` file, containing metadata and versioning information;
- the `meshes` subdirectory, storing the 3D geometry files (`.stl`) for the propellers.

ii. an airframe file named `6001_gz_pizza_drone` (in the `PX4-Autopilot/ROMFS/px4fmu_common/init.d-posix/airframes` folder) configured with the specific mixer and control parameters for the hexacopter frame.

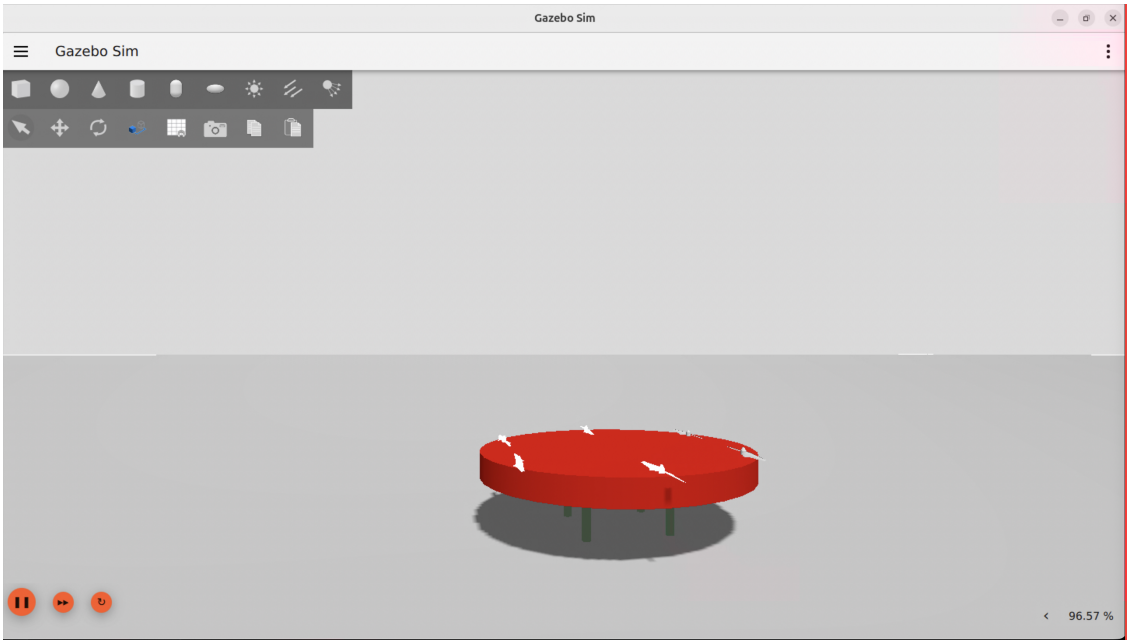

Figure 1: Pizza drone

Finally, in order to run a simulation it is necessary to make the airframe file visible to the `CMAke Lists.txt` file in its folder, `PX4-Autopilot/ROMFS/px4fmu_common/init.d-posix/airframes`. Thus, we added the following line:

```
px4_add_romfs_files(
    ...
    6001_gz_pizza_drone
    ...
)
```

(b) In order to fly our drone in position flight mode we can use the following command from the terminal:

```
make px4_sitl gz_pizza_drone
```

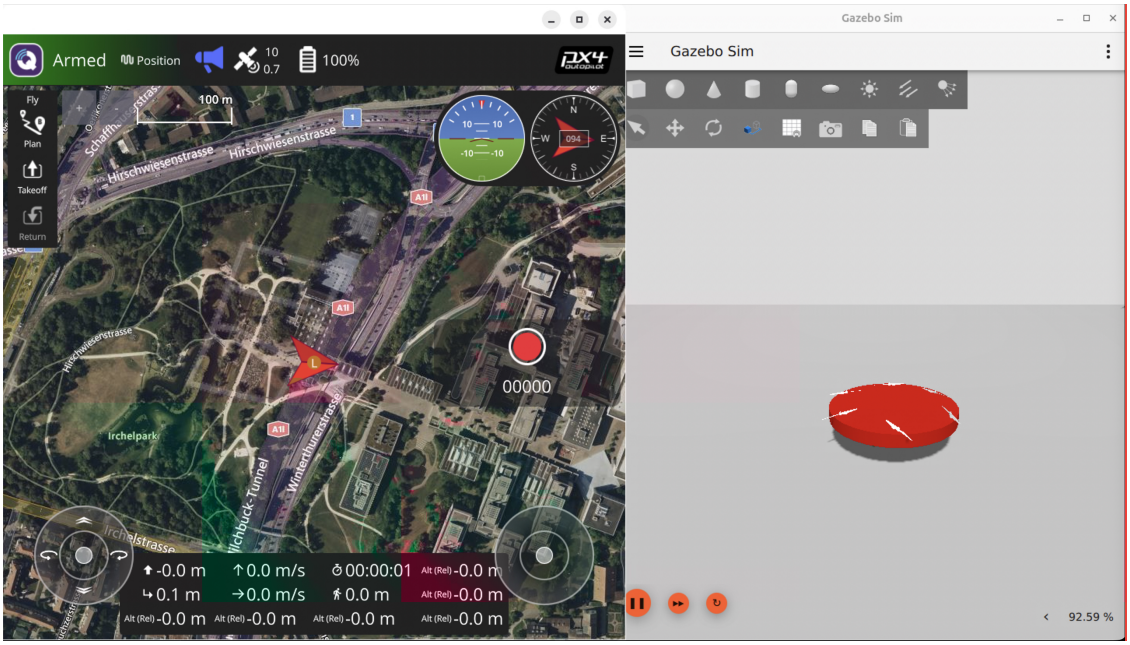

Figure 2: Flying our pizza drone in position flight mode. Video available at link

To plot the actuator outputs during the flight we looked at the `ActuatorOutputs.msg` message in `plotjuggler`. First of all, to make the uORB message visible in ROS we modified the file `dds_topics.yaml` in the folder `PX4-Autopilot/src/modules/uxrce_dds_client/` by adding the following lines in the section of the publication (the actuators outputs are clearly outputs messages to be read):

```
- topic: /fmu/out/actuator_outputs
  type: px4_msgs::msg::ActuatorOutputs
```

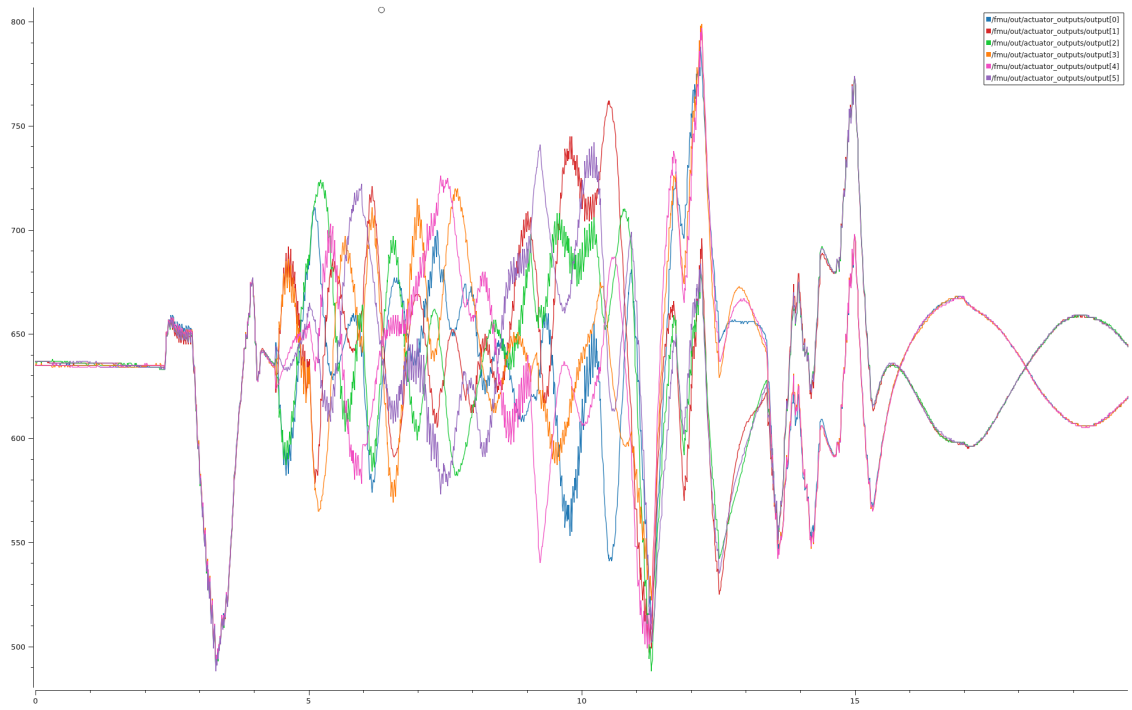

Figure 3: Actuator Outputs

2. **Modification of the `force_land` node**

   (a) It was required to modify the `force_land` node such that if the landing procedure is not completed and the pilot retakes control, even if the threshold is surpassed, nothing will happen. Furthermore, it was suggested to use the uORB message `VehicleLandDetected.msg` to correctly implement this procedure.

   First of all, by examining the `dds_topics.yaml` file in `PX4-Autopilot/src/modules/uxrce_dds_client/` we ensured that this message is already visible in ROS.

   In the original implementation, the node simply checked if the altitude (z) was greater than 20 meters. If true, it immediately published a `VEHICLE_CMD_NAV_LAND` command. To solve this, a "latch" logic was implemented using the `VehicleLandDetected` uORB message. The specific modifications are detailed below:

   i. New subscription and message inclusion: the header <`px4_msgs/msg/vehicle_land_detected.hpp`> was included. A new subscription to the topic `/fmu/out/vehicle_land_detected` was created to monitor the landing status of the drone.

   ii. State flag introduction: a boolean variable `landing_triggered_` was added to the class. This flag acts as a memory state, indicating whether the forced landing procedure has already been activated during the current flight cycle.

   iii. Modified altitude callback: the condition to trigger a landing in `height_callback` was updated:

   ```
   if(z_ > 20.0 && !landing_triggered_)
   ```

   This ensures that if the landing command has already been sent (`landing_triggered_` is true), the node ignores subsequent altitude readings greater than 20 meters, allowing the pilot to fly freely above the threshold.

iv. Activation logic: inside the `activate_switch` function, immediately after publishing the landing command, the flag `landing_triggered_` is set to `true`. This "locks" the automatic landing system.

v. tReset mechanism: a new callback function, `land_detected_callback`, was implemented. It monitors the `landed` field of the incoming message. The system resets only when the physical landing is confirmed:

```
if (msg->landed && landing_triggered_) {
    landing_triggered_ = false;
}
```

This ensures the protection mechanism remains active until the drone is safely on the ground, satisfying the requirement that nothing happens even if the threshold is surpassed again by the pilot.

(b) To show the objective has been successfully reached it is requested to run a simulation and plot some quantitites. The corresponding topics were recorded in a bag file with the command:

```
ros2 bag record -o test_force_land /fmu/out/manual_control_setpoint
↪ /fmu/out/vehicle_local_position
```

We report:

- the drone altitude during the flight, using the ENU reference frame - meaning that z is pointing upwards - and using the `VehicleLocalPosition.msg` message (already visible in ROS). In particular, the message provides the UAV position according to the NED convention, thus we applied a simple scaling filter (-1) in order to recover the desiderd convention;
- the altitude manual control setpoint, by looking at the `ManualControlSetpoint.msg` message. In particular we plotted the `/fmu/out/manual_control_setpoint/throttle` in order to see when the pilot regains the control of the gas stick;

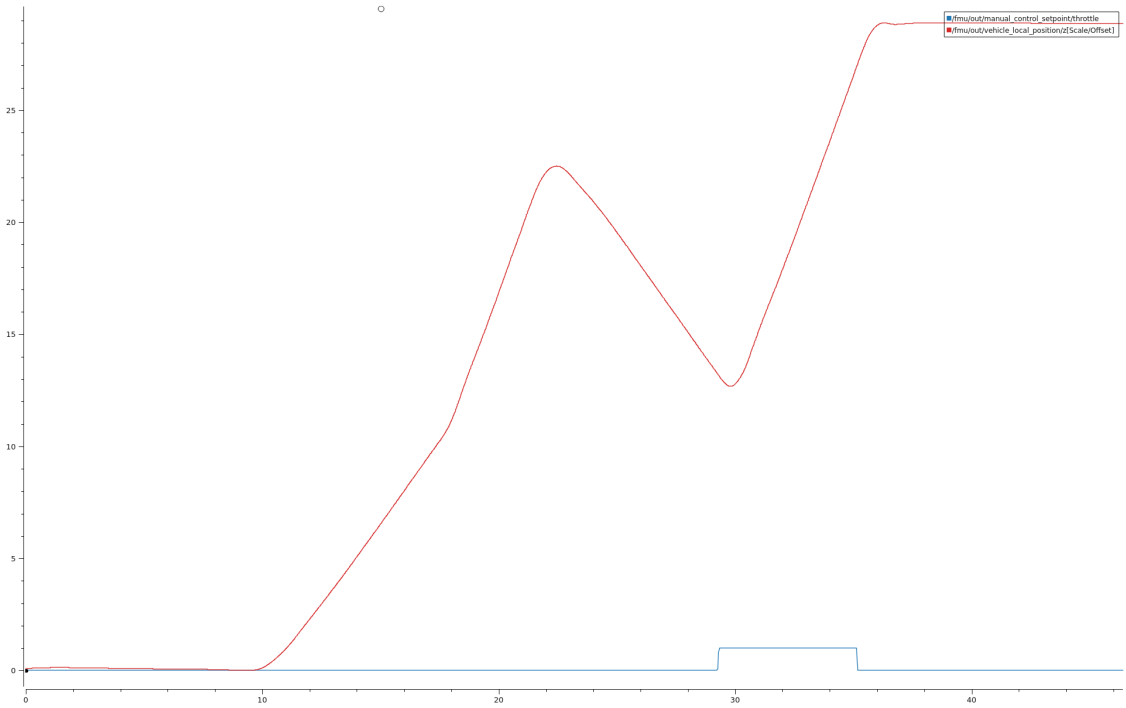

Figure 4: Force Landing procedure. Video available at link

All the previous plots are obtained using `plotjuggler`.

3. **Trajectory planning in offboard mode**

(a) It was required to plan a trajectory with a minimum of 7 waypoints, ensuring the UAV does not stop on the intermediate path waypoints during the trajectory execution (0 m/s only at the end).

The trajectory planner is implemented in the `execute_trajectory.cpp` node. This node is responsible for guiding the UAV through a complex path defined by multiple waypoints using Offboard Control. This implementation utilizes Cubic B-Splines to ensure a smooth, continuous trajectory where the drone does not stop at intermediate points, satisfying the requirement of non-zero velocity during the path execution.

The core logic is divided into three main stages:

i. Trajectory Generation (B-Spline Interpolation): upon initialization, the node defines a matrix of 7 control points (waypoints) relative to the drone's start position. These points include changes in X, Y, Z, and Yaw. To generate the path, the code utilizes the `unsupported/Eigen/Splines` module. Specifically, it uses the `Eigen::SplineFitting::Interpolate` function to create a 3rd-degree (Cubic) B-Spline in 4 dimensions.

ii. Real-time Setpoint Evaluation: the trajectory is parameterized by a normalized time variable $u \in [0, 1]$. At every timer callback:

- The elapsed time $t_{elapsed}$ is updated.
- The spline is sampled using the `spline_.derivatives(u, 2)` method, which returns the position, first derivative (velocity), and second derivative (acceleration) with respect to u.
- Time Scaling: since the spline is parameterized by u but the drone moves in physical time t, the derivatives are scaled by the total mission time T (set to 40 seconds):

$$\mathbf{v}(t) = \frac{d\mathbf{S}}{du} \cdot \frac{1}{T}, \quad \mathbf{a}(t) = \frac{d^2\mathbf{S}}{du^2} \cdot \frac{1}{T^2} \tag{1}$$

- These values are published via the `TrajectorySetpoint` message to the PX4 autopilot.
- Mission Completion and Auto-Landing: a specific logic handles the end of the trajectory. When the normalized time $u \geq 1.0$:
  - The node triggers the `VEHICLE_CMD_NAV_LAND` command to force an automatic landing.
  - A boolean flag `landing_requested_` is set to `true`, which immediately stops the publication of `OffboardControlMode` heartbeats. This is crucial to allow the flight stack to switch internally from Offboard mode to Land mode without conflict.

The original `go_to_point` template was a basic setpoint publisher. The modified `execute_trajectory` node introduces the following key advancements:

- Multi-Waypoint Interpolation: instead of a single target, the node manages a vector of 7 distinct waypoints forming a complex shape.
- Continuous Motion: the original scalar polynomial approach often resulted in stops at every segment. The B-Spline approach allows the drone to pass through intermediate waypoints with non-zero velocity ($v \neq 0$), stopping only at the very end ($u = 1$).
- Full State Control: the node publishes Position, Velocity, and Acceleration setpoints simultaneously (derived from the spline), ensuring tighter tracking performance compared to position-only control.
- Attitude Planning: the planner explicitly interpolates the Yaw angle alongside the Cartesian coordinates, satisfying the requirement to plan attitude setpoints.

(b) To show `pizza_drone` following the planned trajectory we run a simulation (video available at link).

By examining the `dds_topics.yaml` file in `PX4-Autopilot/src/modules/uxrce_dds_client/` we ensured that all messages necessary for the plot are already visible in ROS.

Here we report the time histories of the following quantitites:

- the trajectory into the xy plane

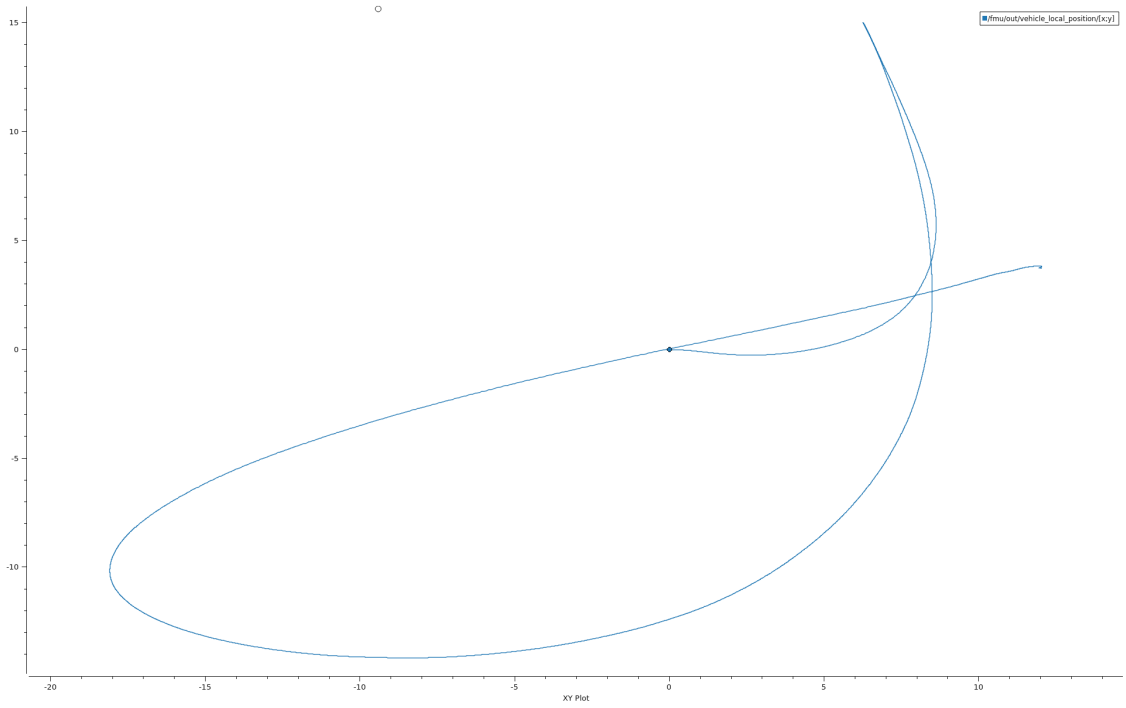

Figure 5: Trajectory into the xy plane

- the altitude

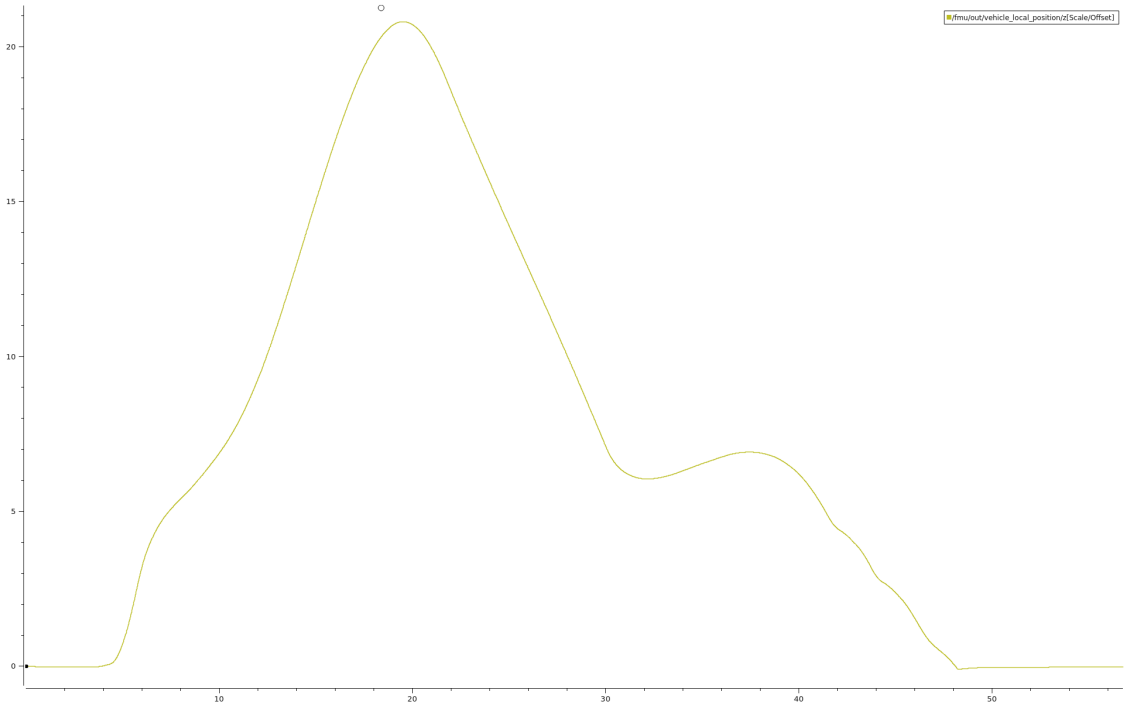

Figure 6: Drone altitude

- the yaw angle

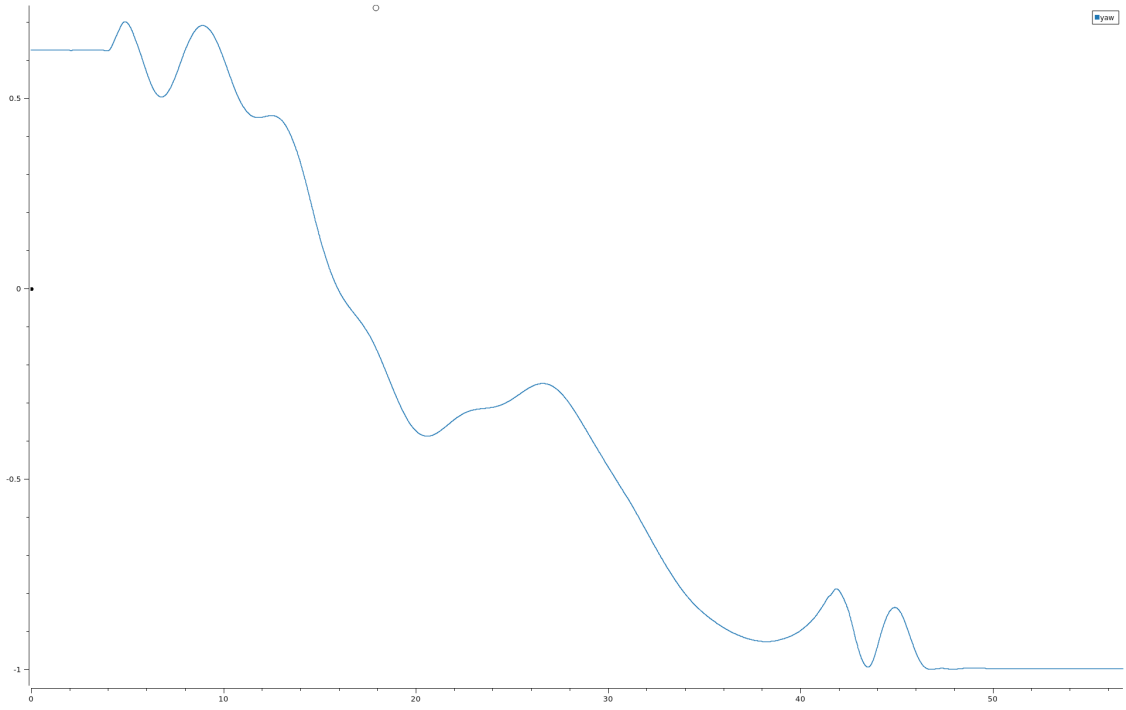

Figure 7: Yaw angle

- the UAV velocity

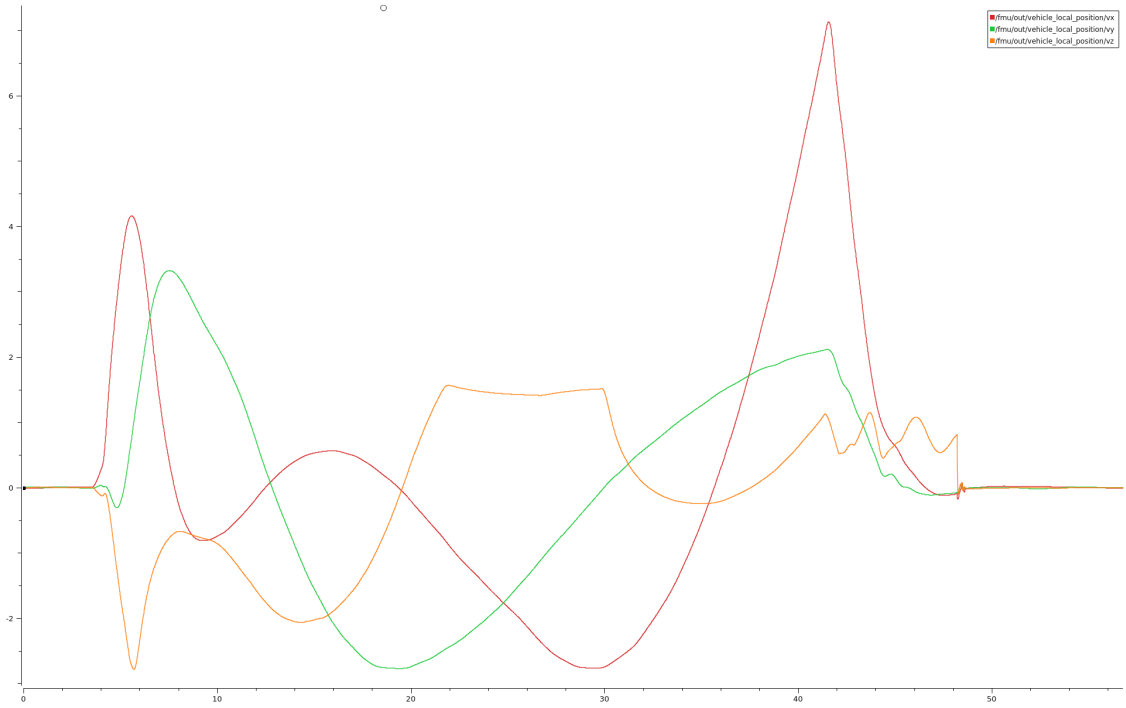

Figure 8: UAV velocity

- the UAV acceleration

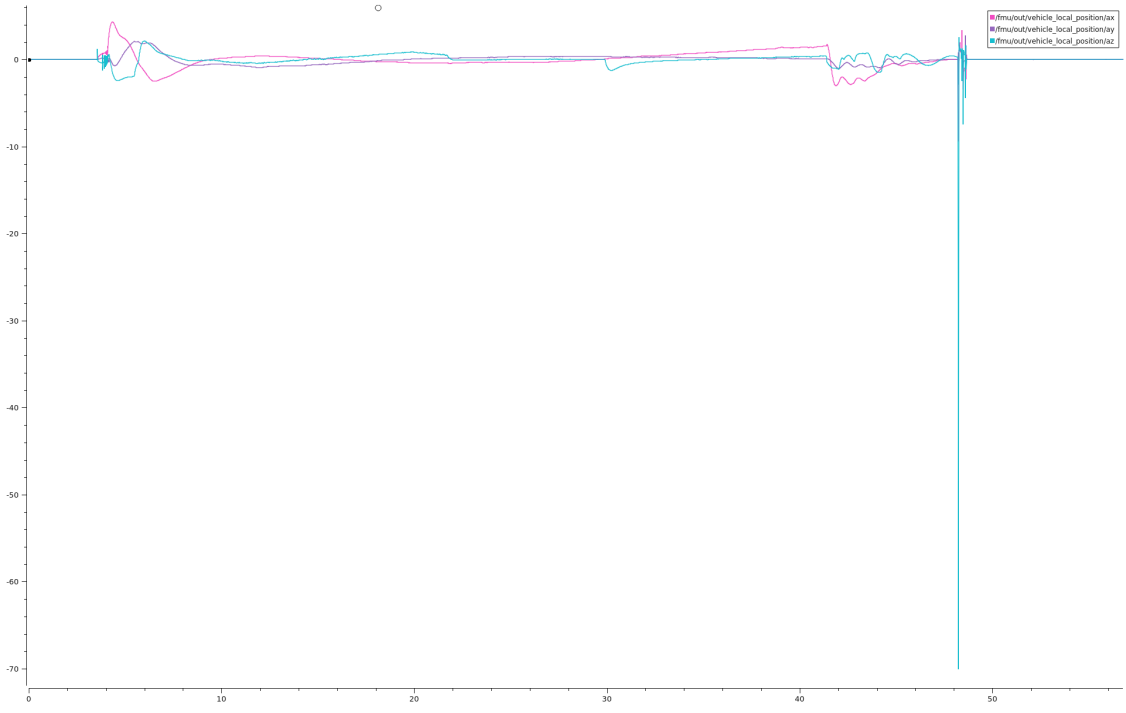

Figure 9: UAV acceleration