UNIVERSITÀ DEGLI STUDI DI MESSINA

DIPARTIMENTO DI INGEGNERIA

Corso di Laurea Magistrale in Engineering and Computer Science

Industrial IoT

Project:

Drone control and obstacle detection using PX4 and QGroundControl

Students:

Longo Andrea

Musmeci Edoardo

ANNO ACCADEMICO 2023-2024

Contents

Introduction

The objective of this project is to develop an autonomous system for controlling a drone, implemented using ROS 2 as the companion computer, PX4 to simulate the flight controller, and QGroundControl for monitoring. The drone is initially sent on an automated mission toward a predefined waypoint. During the flight, the system can detect any obstacles along the path and, in response, switch the flight mode to "offboard" to perform a dynamic detour. Once the obstacle is avoided, the drone automatically returns to the mission mode and completes the route by reaching the designated final coordinates. To achieve this functionality, a ROS 2 node in Python has been developed, which manages the logic for switching flight modes and controlling the trajectory. The QGroundControl graphical interface is used to monitor and visualize the drone's flight, facilitating the configuration and verification of the mission.

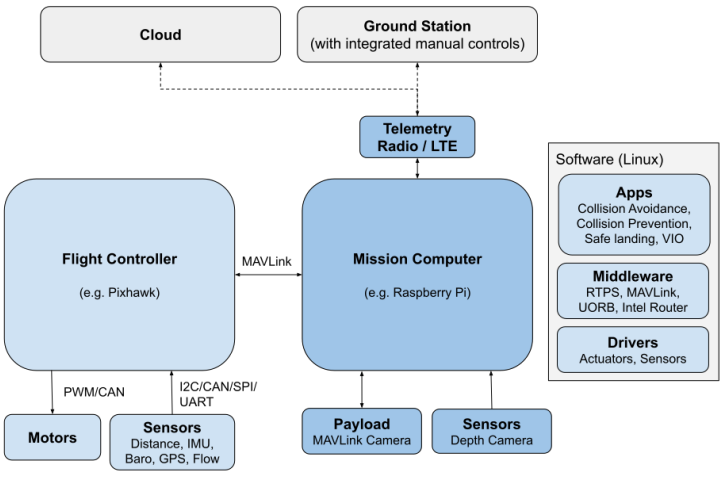

Figure 1: Hardware and Software stack

Chapter 1

Description of the development environment

1.1 PX4-Autopilot e Gazebo Classic

The PX4 firmware can be run in simulation mode, behaving as if it were installed on a real drone's flight controller, but instead communicating with Gazebo. The latter provides a realistic simulation environment in which a drone model can fly and interact with its surroundings. The simulation includes the generation of sensor data from the drone, such as IMU, GPS, and others, which are sent to the PX4 firmware. This allows for testing and developing projects in a controlled environment, replicating real-world scenarios without the need for physical hardware.

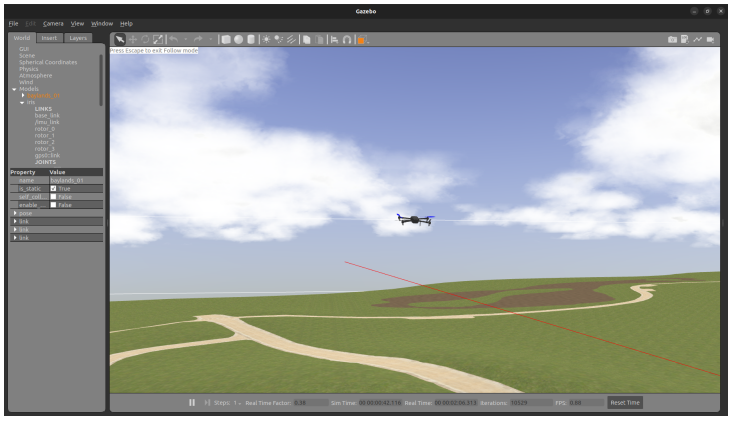

Figure 1.1: Baylands Gazebo World

1.2 Interaction with ROS 2

ROS 2 nodes can interact with PX4 and Gazebo to publish flight commands, receive data from simulated sensors, and implement control and

navigation algorithms. ROS 2 acts as an intermediary between the user (or high-level algorithms) and the PX4 firmware, enabling real-time control of the drone and allowing its trajectory to be adjusted in response to external events or changes in the path.

1.3 Control and monitoring via QGC

QGroundControl (QGC) connects to the PX4 firmware via the MAVLink protocol, allowing it to operate both in simulation (SITL), with PX4 running inside Gazebo, and on real hardware (HITL). Through QGC, the user can monitor real-time drone data, such as position and sensor status. This tool provides an intuitive graphical interface for setting up missions and monitoring the system's proper operation.

1.4 MAVROS

MAVROS is a ROS (Robot Operating System) package that acts as a communication bridge between ROS and MAVLink systems like PX4, primarily used for interfacing with drones. This package allows the control and monitoring of the vehicle through MAVLink messages, handling both telemetry data and navigation commands. MAVROS supports a wide range of functionalities, including retrieving vehicle status data (position, velocity, attitude, etc.), setting flight modes, waypoints, and missions.

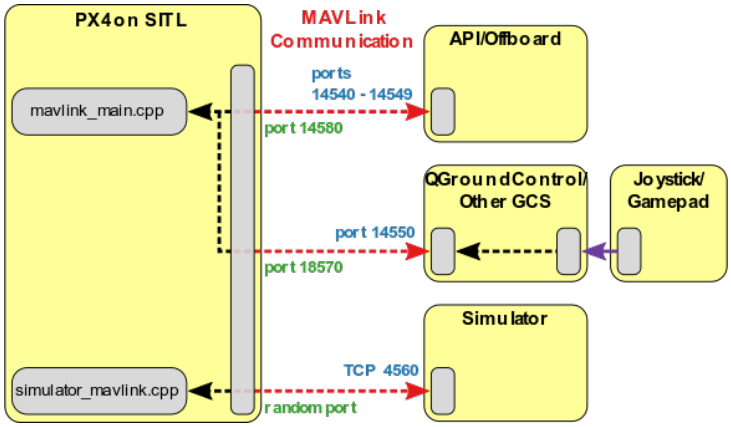

Figure 1.2: Interaction between system's parts

Chapter 2

PX4 and ROS 2 Installation Guide on Ubuntu

This guide describes the steps to install PX4 and ROS 2 on an Ubuntu system.

2.1 PX4 Installation

Requirements:

- Ubuntu 22.04

2.2 Download PX4 Autopilot

Clone the PX4 repository:

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

2.3 Compiling PX4

Once all dependencies are installed, compile the PX4 firmware:

```
cd PX4-Autopilot/
make px4_sitl_default gazebo-classic_iris
```

2.4 ROS 2 Installation

```
sudo apt update && sudo apt install locales
sudo locale-gen en_US en_US.UTF-8
sudo update-locale LC_ALL=en_US.UTF-8 LANG=en_US.UTF-8
```

```
    export LANG=en_US.UTF-8
    sudo apt install software-properties-common
    sudo add-apt-repository universe
    sudo apt update && sudo apt install curl -y
    sudo curl -sSL https://raw.githubusercontent.com/ros/rosdistro/master/ros.key -o /usr/share/keyrings/ros-archive-keyring.gpg
    echo "deb [arch=$(dpkg --print-architecture) signed-by=/usr/share/keyrings/ros-archive-keyring.gpg] http://packages.ros.org/ros2/ubuntu $(. /etc/os-release && echo $UBUNTU_CODENAME) main" | sudo tee /etc/apt/sources.list.d/ros2.list > /dev/null
    sudo apt update && sudo apt upgrade -y
    sudo apt install ros-humble-desktop
    sudo apt install ros-dev-tools
    source /opt/ros/humble/setup.bash && echo "source /opt/ros/humble/setup.bash" >> .bashrc
```

2.5 How to create a ROS 2 workspace:

```
    mkdir ros2_ws
    cd ros2_ws
    ls
    mkdir src
    ls
    colcon buid
    ros2 pkg create project_pkg --build-type ament_python
    cd ros2_ws
    colcon build
    colcon build --packages-select project_pkg
```

Chapter 3

Project description

3.1 Mission Mode

In Mission Mode, GPS coordinates are set, and the waypoint coordinates are then uploaded to the drone using the WaypointPush service. Through the function set_auto_mission_mode, which uses the “SetMode” service, the ”AUTO.MISSION” mode is activated, allowing the drone to take off and head towards the defined waypoint. To verify that the mission mode has been correctly set, the /mavros/state topic is monitored.

```
import rclpy
from rclpy.node import Node
from rclpy.qos import QoSProfile, ReliabilityPolicy, DurabilityPolicy,
    HistoryPolicy, qos_profile_sensor_data
from mavros_msgs.msg import Waypoint, State
from mavros_msgs.srv import WaypointPush, SetMode, CommandBool
from geometry_msgs.msg import PoseStamped
import math


class MissionModeNode(Node):
    def __init__(self):
        super().__init__('mission_mode_node')

        qos_profile = QoSProfile(
            reliability=ReliabilityPolicy.BEST_EFFORT,
            durability=DurabilityPolicy.TRANSIENT_LOCAL,
            history=HistoryPolicy.KEEP_LAST,
            depth=10
        )

        self.get_logger().info("Nodo missione avviato!")

        self.setpoint_timer = self.create_timer(
            0.1, self.publish_position_setpoint)

        self.obstacle_timer = self.create_timer(
            1.0, self.publish_obstacle_position)

```

```
29
30          self.lat = 37.4144411
31          self.lon = -121.9959840
32          self.alt = 10.0
33
34          self.current_state = State()
35          self.setpoint = PoseStamped()
36          self.current_position = None
37
38          self.reference_position = PoseStamped().pose.position
39          self.obstacle_position = None
40          self.scostamento = 20.0
41          self.count = 0
42
43          self.send_mission()
44          self.service_check()
45
46      def send_mission(self):
47          wp = Waypoint()
48          wp.frame = Waypoint.FRAME_GLOBAL_REL_ALT
49          wp.command = 16
50          wp.is_current = True
51          wp.autocontinue = True
52          wp.z_alt = self.alt
53          wp.x_lat = self.lat
54          wp.y_long = self.lon
55
56          wp_push_req = WaypointPush.Request()
57          wp_push_req.start_index = 0
58          wp_push_req.waypoints.append(wp)
59
60          future = self.wp_push_client.call_async(wp_push_req)
61          rclpy.spin_until_future_complete(self, future)
62          if future.result() and future.result().success:
63              self.get_logger().info('Missione caricata con successo!')
64              self.set_auto_mission_mode()
65          else:
66              self.get_logger().error('Errore nel caricamento della
    missione.')
67
68      def set_auto_mission_mode(self):
69          set_mode_req = SetMode.Request()
70          set_mode_req.custom_mode = "AUTO.MISSION"
71          future = self.set_mode_client.call_async(set_mode_req)
72          rclpy.spin_until_future_complete(self, future)
73          if future.result() and future.result().mode_sent:
74              self.get_logger().info('Modalita AUTO.MISSION impostata con
     successo!')
75          else:
76              self.get_logger().error('Errore nell impostazione della
    modalita AUTO.MISSION.')
```

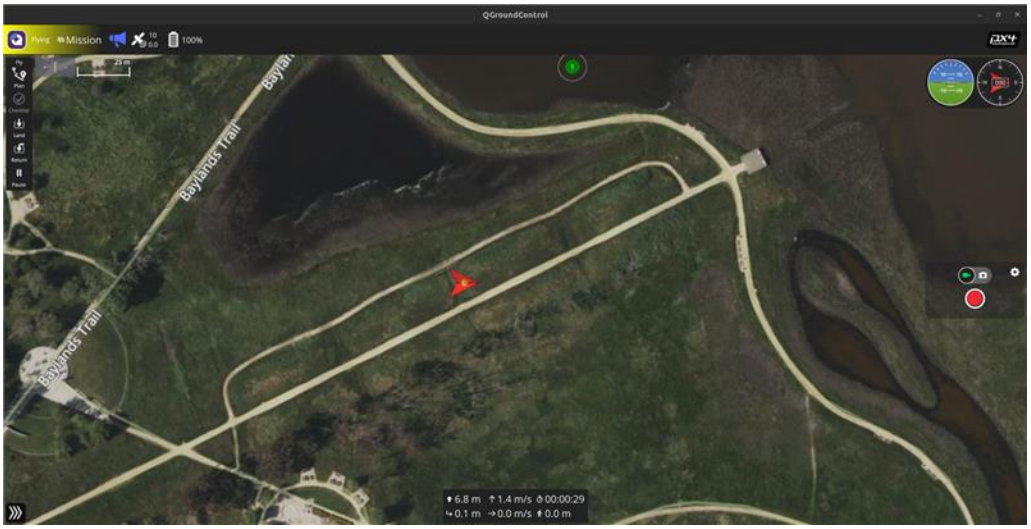

Figure 3.1: Mission Mode

3.2 Drone communication and control via MAVROS

This paragraph describes the communication infrastructure for controlling the drone using ROS 2 and MAVROS, through the publishing and subscribing to various messages, as well as the use of certain services.
Publisher:

- position setpoint: the publisher self.setpoint_pub sends position setpoints to the /mavros/setpoint_position/local topic. This allows the desired position of the drone to be defined.
- obstacle position: self.obstacle_pub publishes the position of any obstacles on the /obstacle_position topic, enabling the drone to avoid them.

Subscriber:

- current position of the drone: self.local_pos_sub is a subscriber that listens to the /mavros/local_position/pose topic, receiving the current position of the drone relative to the launch point. This position is then handled by the self.local_pos_callback function.
- drone state: self.state_sub subscribes to the /mavros/state topic, obtaining the current state of the drone (armed, mission, offboard, etc.). The self.state_callback function handles the received data by updating the drone's state.

- obstacle position: self.obstacle_pos_sub subscribes to the /obstacle_position topic to obtain the position of the obstacle, which is used to avoid collisions. The data is processed by the self.obstacle_pos_callback function.

Client:

- waypoint uploading: self.wp_push_client creates a client for the /mavros/mission/push service, allowing waypoints to be uploaded to the drone's mission. The client continues to search for the service until it becomes available, displaying a waiting message.
- setting flight mode: self.set_mode_client allows changing the drone's mode through the /mavros/set_mode service.
- arming service: self.arming_client connects to the /mavros/cmd/arming service to activate or deactivate the drone's arming, enabling or disabling it for flight.

```
        self.setpoint_pub = self.create_publisher(
            PoseStamped, '/mavros/setpoint_position/local', qos_profile
)

        self.obstacle_pub = self.create_publisher(
            PoseStamped, '/obstacle_position', qos_profile)

        self.local_pos_sub = self.create_subscription(
            PoseStamped, '/mavros/local_position/pose', self.
local_pos_callback, qos_profile_sensor_data)

        self.state_sub = self.create_subscription(
            State, '/mavros/state', self.state_callback, 10)

        self.obstacle_pos_sub = self.create_subscription(
            PoseStamped, '/obstacle_position', self.
obstacle_pos_callback, qos_profile_sensor_data)

        self.wp_push_client = self.create_client(
            WaypointPush, '/mavros/mission/push')
        while not self.wp_push_client.wait_for_service(timeout_sec=1.0)
:
        self.get_logger().info('Servizio /mavros/mission/push non
disponibile, in attesa...')

        self.set_mode_client = self.create_client(SetMode, '/mavros/
set_mode')
        while not self.set_mode_client.wait_for_service(timeout_sec
=1.0):
        self.get_logger().info('Servizio /mavros/set_mode non
disponibile, in attesa...')
```

```
24
25          self.arming_client = self.create_client(
26              CommandBool, '/mavros/cmd/arming')
```

3.3 Service verification and drone state monitoring

The following code defines a series of methods used for the control and management of the drone's autonomous flight. In particular, the service_check function monitors the availability of the arming service (arming_client) and the mode setting service (set_mode_client). The drone is armed only when both services are available. The state_callback method is triggered whenever a status message is received, updating the current state of the drone and logging any mode changes. Finally, the local_pos_callback updates the current position of the drone.

```
1     def service_check(self):
2         while not self.arming_client.wait_for_service(timeout_sec=1.0)
    or not self.set_mode_client.wait_for_service(timeout_sec=1.0):
3             self.get_logger().warn('Attendo i servizi...')
4         self.get_logger().info("Servizi disponibili")
5         self.arm_drone()
6
7     def state_callback(self, msg):
8         if msg.mode != self.current_state.mode:
9             self.current_state = msg
10            self.get_logger().info(f"Stato attuale: {self.current_state
    .mode}")
11
12    def local_pos_callback(self, msg):
13        self.current_position = msg.pose.position
```

3.4 Offboard Mode and obstacle detection

During the flight in "AUTO.MISSION" mode, the drone monitors the /obstacle_position topic, where coordinates simulating the presence of an obstacle are published. As it approaches the obstacle, the drone switches to "Offboard" mode using the change_mode function and moves according to position setpoints published on the /mavros/setpoint_position/local topic. Once the drone reaches the setpoint coordinates, it returns to mission mode to continue towards the waypoint. The obstacle_pos_callback function is responsible for updating the position of obstacles that the drone must monitor to avoid collisions. In the publish_position_setpoint function, the drone's current position is checked to determine if an obstacle is present. If an obstacle is detected within a 5-meter radius, the drone updates its reference position and changes its flight mode to "Offboard." The found_obstacle function calculates the distance between the drone's current position and that of the obstacle. If the distance is less than 5 meters, it indicates that an obstacle has been found, and the drone's reference position is updated accordingly. Finally, the reached_setpoint function checks whether the drone has reached the setpoint coordinates by comparing its current position with the expected setpoint, using a predefined tolerance of 1 meter. If the drone is within this tolerance, the setpoint is considered reached, and a log message is recorded to indicate the achievement of the goal.

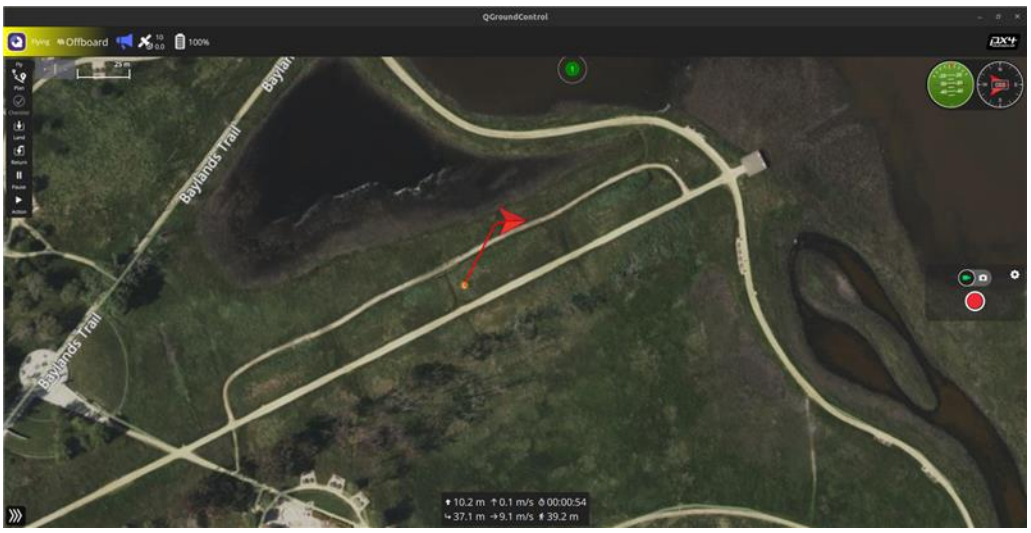

Figure 3.2: Offboard mode

3.4 Offboard Mode and obstacle detection

```
    def obstacle_pos_callback(self, msg):
        self.obstacle_position = msg.pose.position

    def publish_position_setpoint(self):
        if self.current_position is None or self.obstacle_position is
    None:
            return

        if self.found_obstacle() and self.count==0:
            self.count +=1
            self.get_logger().info("Ostacolo rilevato!")
            self.reference_position = self.current_position
            self.change_mode("OFFBOARD")

        self.setpoint.pose.position.x = self.reference_position.x +
    self.scostamento
        self.setpoint.pose.position.y = self.reference_position.y
        self.setpoint.pose.position.z = self.reference_position.z

        self.setpoint_pub.publish(self.setpoint)

        if self.reached_setpoint():
            self.set_auto_mission_mode()


    def found_obstacle(self):
        if self.current_position is None or self.setpoint is None:
            return False

        distance = math.sqrt(
            (self.obstacle_position.x - self.current_position.x)**2 +
            (self.obstacle_position.y - self.current_position.y)**2 +
            (self.obstacle_position.z - self.current_position.z)**2
        )
        if distance < 5.0:
            self.reference_position = self.current_position
            return True
        return False


    def publish_obstacle_position(self):
        obstacle_msg = PoseStamped()
        obstacle_msg.pose.position.x = 15.0
        obstacle_msg.pose.position.y = 30.0
        obstacle_msg.pose.position.z = 10.0
        self.obstacle_pub.publish(obstacle_msg)

    def reached_setpoint(self):
        if self.current_position is None or self.setpoint is None:
            return False

        tolerance = 1.0
        dx = abs(self.current_position.x - self.setpoint.pose.position.
    x)
        dy = abs(self.current_position.y - self.setpoint.pose.position.
    y)
        dz = abs(self.current_position.z - self.setpoint.pose.position.
    z)
```

```
54
55          if dx < tolerance and dy < tolerance and dz < tolerance:
56              self.get_logger().info("Setpoint raggiunto!")
57              return True
58          return False
```

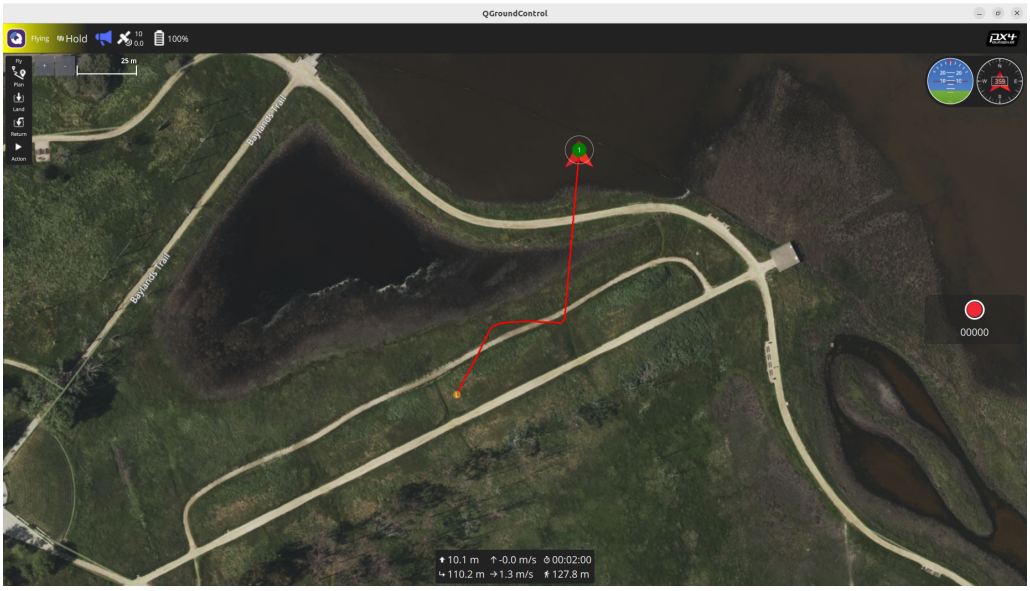

Figure 3.3: End of mission

3.5 Flask version

An additional version of the project was implemented using Flask. The latter is a lightweight web framework for Python that allows for the easy creation of web applications. In this case, Flask is used to create an API that listens for incoming POST requests at the /update_coordinates endpoint. The API expects to receive data in JSON format containing the coordinates, specifically the X and Y values. Once the coordinates are received, they are modified: the X coordinate is shifted 20 meters to the right, while the Y coordinate remains unchanged. The new coordinates are returned as a JSON response, allowing the API to interact with the ROS 2 node by updating the setpoints.

```python
from flask import Flask, request, jsonify

app = Flask(__name__)


@app.route('/update_coordinates', methods=['POST'])
def update_coordinates():
    data = request.get_json()

    x_coord = data.get('X')
    y_coord = data.get('Y')

    print(f"Ricevute coordinate: X {x_coord}, Y {y_coord}")


    meters_to_shift = 20.0
    new_x = x_coord + meters_to_shift
    new_y = y_coord
    return jsonify({"new_x": new_x, "new_y": new_y})

if __name__ == '__main__':
    app.run(debug=True, host='0.0.0.0', port=5000)
```

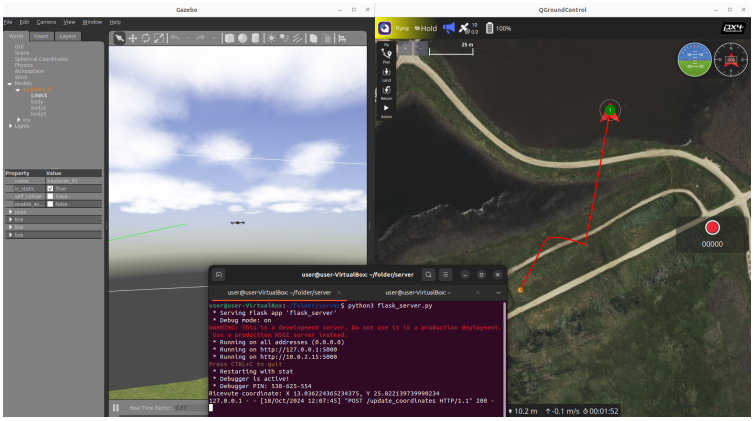

Figure 3.4: Mission with server flask

Chapter 4

Conclusions

In this project, a system has been developed that can dynamically manage the flight of an autonomous drone, thanks to the integration of PX4, ROS 2, and QGroundControl. The drone, initially programmed to complete a predefined mission, is designed to detect obstacles along its path and modify its flight trajectory by switching to "offboard" mode. After avoiding the obstacle, the drone can return to mission mode and successfully complete the journey to its destination. The ROS 2 node developed in Python handles the flight logic through the MAVROS library, offering a high degree of flexibility and control. The use of QGroundControl has facilitated real-time monitoring of the drone, providing an intuitive graphical interface to visualize the drone's position and sensor status, as well as to send commands and missions. Thanks to this approach, the system can be easily tested and optimized in simulated environments before being implemented on real hardware, thereby ensuring a safer and more efficient development cycle.

4.1 Future works

In our implementation, the drone's coordinates relative to the launch point are saved after detecting the obstacle. These coordinates are modified via the Flask API and used as position setpoints. A possible modification could involve updating the GPS coordinates and using them to create a new waypoint.