The parameters presented between lines 12 and 16 (or between lines 30 and 34) of the configuration file determine whether or not the **offboard_launcher.sh** program launches each of the tools that support the testing and validation of the user program. These tools will be presented in Section 5.5. For instance, for each vehicle featured in the configuration blocks of Fig. 5.8, the **offboard_launcher.sh** will start a process that automatically logs information produced by the motion capture system, the sensors, and the PX4 autopilot of the vehicle. It should be noted that the **offboard_launcher.sh** program allows to easily run the user program and the support tools across multiple computers. The user only has to configure the code blocks of the **offboard_launcher.sh** file of each computer accordingly.

The Bash code implemented in the second part of the **offboard_launcher.sh** file, responsible for starting each of the processes selected in the configuration blocks, can be found in the digital repository. It should be noted that, before launching an user program that employs MAVROS libraries, the **offboard_launcher.sh** connects the ROS middleware to the vehicle using the px4.launch file [33]. For the MAVSDK modules, the connection to the vehicles is performed in the user programs, when the telemetry methods of the UAV objects start running in the background, through the add_any_connection [34] function of the MAVSDK C++ API and through the system.connect [35] method of the MAVSDK Python API.

5.5 Additional Tools

In this section, a set of software solutions that add new features to the testing environment is presented. These software solutions run in processes that are independent from the user program and from each other, so that an error or unexpected behavior during the execution of one process does not force the other programs to suddenly stop running.

5.5.1 Optitrack pose forwarder

As stated in Chapter 3, whenever a real multicopter does not feature an onboard companion computer, it is necessary to send its Optitrack pose to its PX4 autopilot through a ground computer. Since the methods of the TELEMETRY class subscribe to the position and attitude of the vehicle provided by the Optitrack system, and since the OFFBOARD class features methods for sending that data to the PX4 autopilot, this procedure could have been implemented in the user program. However, this would not the best approach, because if the user program suddenly stopped working, the PX4 autopilot would also stop receiving the Optitrack data and would not produce valid state estimates to safely perform an auto-landing maneuver, when the offboard link loss failsafe was triggered. Consequently, an independent software program called **Optitrack pose forwarder** was implemented for sending the Optitrack pose to the PX4 autopilot of the vehicles that do not feature an onboard companion computer. This program performs the steps extensively described in Chapter 3 - it downsamples, from 180Hz to 60Hz, the position and attitude data of the vehicle published by the Optitrack system in the local network, and sends it to PX4 autopilot using the MAVLink-Router software. To launch this tool, the user only needs to set to 'yes' the **optitrack_pose_forwarder** parameter of the configuration blocks presented in Fig. 5.8.

5.5.2 Relative position sensor emulator

The second supporting tool developed in this thesis consists of a software program that emulates a relative position sensor. This tool provides the position of the drones of an experiment relative to the NED coordinate frame centered on the vehicle where the sensor is installed. The relative position measurements are emulated by subtracting the MOCAP position of the drones of the experiment to the MOCAP position of the vehicle with the sensor, and by adding adjustable Gaussian white noise to the result. The relative position measurements are then published in a topic similar to those employed by the PX4 autopilot to publish sensor readings. Since the TELEMETRY class features a method to automatically subscribe to this topic, the users have access to relative position measurements through the **uav.sen.emu.rel_pos** variable. To run this supporting tool, users only have to add, in the **neighborhood** field of the **offboard_launcher.sh** program presented in Fig. 5.8, the namespace (**ns**) of the drones for which they need to get the relative position. For instance, in the configuration blocks of Fig. 5.8, if users need to emulate a relative position sensor in UAV1 in order to have access to the relative position of UAV2, they must add the namespace of the UAV2 to the **neighborhood** field of line 13. Therefore, the code of line 13 must be **neighborhood+=("uav2")**. This supporting tool is important to the rapid prototyping environment because it proves that sensors not available in the vehicles can be easily emulated, and because it serves as a template for the implementation of other emulated sensors.

5.5.3 Offboard logger

The offboard logger is a Python program that automatically subscribes to all the information related to the state of a vehicle and logs it to a **CSV** file that can be imported into Matlab or other software for further processing and analysis. This tool also produces plots with the time evolution of all logged physical quantities. The offboard logger enables the rapid validation of control algorithms because the **CSV** file and the plots become immediately available to the user and allow the comparison of the actual and the desired values of a physical quantity, as presented in Fig. 5.9. The PX4 autopilot also has a sophisticated logging system that produces **ULog** files [36]. These files can be converted to **CSV** or can be analyzed in open-source software programs [37]. The PX4 logging system complements the developed offboard logger.

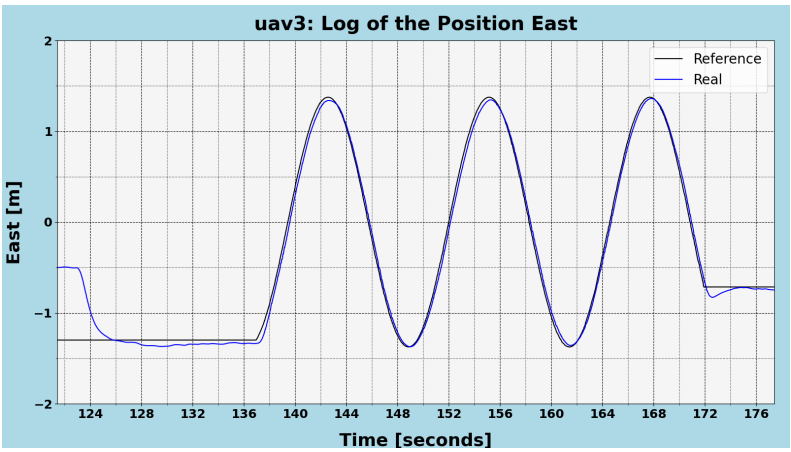

Figure 5.9: Plot of the log of the actual (blue) and desired (black) position east of a drone.

5.5.4 Real-time monitor

The fourth supporting tool developed in this thesis consists of a Python program that enables the user to monitor the state of the vehicles during experiments. For example, Fig. 5.10 depicts the window launched by this software to allow the monitoring of the three components of the actual and desired position of a drone, during the validation of a control algorithm. To launch this software solution, the user needs to set the **real_time_monitor** parameter of the configuration blocks presented in Fig. 5.8, according to the instructions of the documentation available in the digital repository. Note that it is possible to launch several windows for the same drone, each one plotting the components of a different physical quantity. This tool is important for the developed testing framework because it complements and improves the monitoring setup available in the QGroundControl software, in which the user has to monitor the components of all desired variables in a single window with only two subplots.

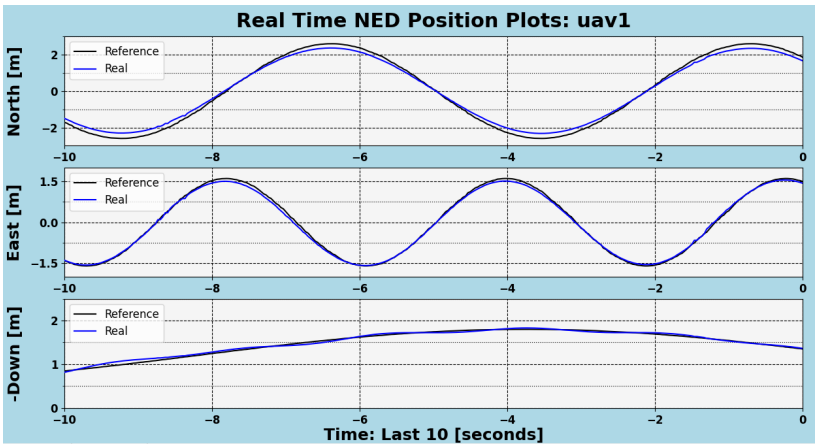

Figure 5.10: Window with the 3 components of the actual (blue) and desired (black) position of the drone.

5.5.5 Visualization tools

The last supporting tools developed in this thesis consist of a set of lightweight software programs that reproduce, in 3D, the evolution of the position and attitude of all drones of an experiment. These programs enable the user to monitor the 3D pose of the vehicles during the tests, by subscribing in real-time to their pose information, and also after the tests, by reproducing the pose data saved in the log files.

The visualization tools are crucial for the designed testing framework because, by reproducing the 3D motion of the vehicles during and after the experiments, they enable the user to run simulations without the graphical user interface of Gazebo and still monitor the behavior of the vehicles. This results in computationally lighter simulations. The user can also run the Gazebo software and the visualization tools in different computers, dividing the computational power required to run and monitor a simulation across two different machines. Additionally, these tools also enable the user to visualize, in a single window, the 3D motion of all the drones of the experience, regardless of whether they are real or being simulated in any of the computers of the local network.

Figures 5.11, 5.12, and 5.13 present the three visualization tools developed. The Gazebo-based tool employs the graphical user interface of Gazebo in a lightweight way, that is, significantly reducing the

frequency in which vehicles are plotted and eliminating unnecessary animations, such as the motors rotation. The Python tool features a simple 3D representation of the drones that do not provide information about the attitude of the vehicles, so that it does not generate excessive overhead, which is important because this tool was created especially to be employed in experiments with hundreds of vehicles, that are too computationally heavy to be reproduced by the other two visualization programs.

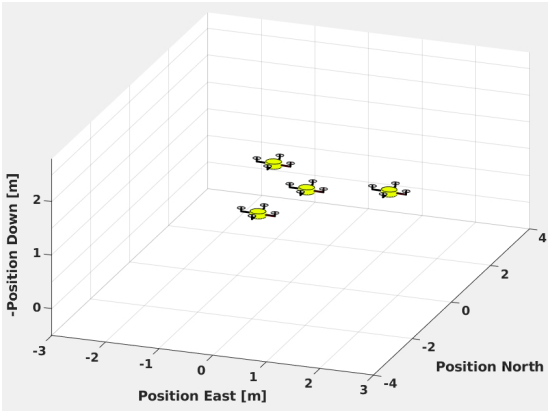

Figure 5.11: Matlab visualization tool plotting, in a single window, a formation of four vehicles, two of them simulated in one computer and the other two simulated in another computer.

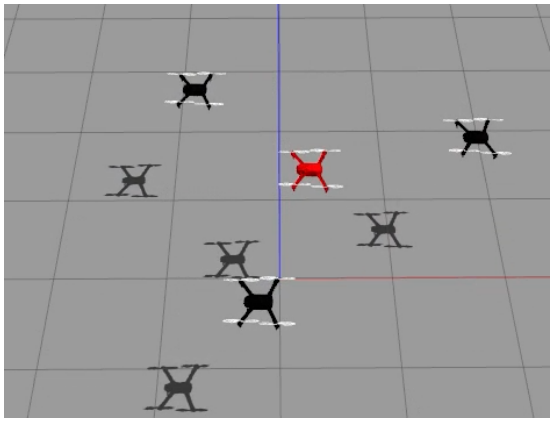

Figure 5.12: Gazebo-based visualization tool plotting, in a single window, a formation of four vehicles, one of them real and the other three simulated.

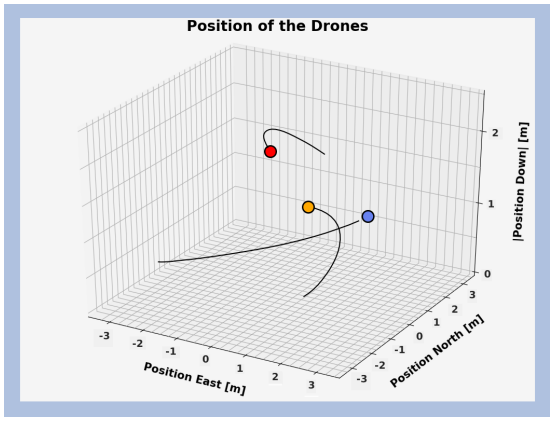

Figure 5.13: Python visualization tool showing the motion of three drones.