In the code of lines 16 and 17, the initial position and yaw of the vehicle are retrieved from the **uav.ekf.pos** and the **uav.ekf.att_euler** variables that store, respectively, the current position and attitude of the drone provided by the extended Kalman filter of the PX4 autopilot. The position and the initial yaw of the vehicle could have also been retrieved from the **uav.sen.mocap.pos** and **uav.sen.mocap.att_euler** variables, that store the raw measurements of the position and attitude of the drone provided by the motion capture system. This shows that the data of the UAV class is easily accessible and is also stored in an intuitive way. For instance, in this case, it is easy to distinguish whether the pose comes from the extended Kalman filter or from the MOCAP system.

Finally, the method that sends a position and yaw setpoint to the PX4 is called in line 21, and the method that triggers the auto-land mode is called in line 23. The PX4 autopilot disengages from offboard mode if it does not receive offboard commands at a frequency of, at least, 2Hz. Consequently, the methods that send setpoints or control commands to the PX4, such as the **uav.set_pos_yaw** method, feature an internal loop that continuously creates offboard messages with the control references desired by the user and sends them according to the protocol in use. Once again, the methods of the UAV class work as an interface between the user program and the PX4, hiding and relieving the user from implementing low-level communication tasks.

In summary, the example of Fig. 5.7 proves that users, by employing the UAV class developed in this thesis, are able to easily and effortlessly code and test their navigation and control algorithms in the ISR Flying Arena. The methods of this class perform all the communication tasks required to carry out an experiment, and the details of their implementation are hidden from the user. Consequently, users can employ and benefit form the services of the methods of the UAV class without knowing how they work in detail or how they were implemented. This results in compact offboard programs, independent from the multicopter used, such as the example of Fig. 5.7. This example is much more readable and easy to implement than the official takeoff and land examples [31, 32] and does not require knowledge of the communication libraries. In Chapter 6, which is the chapter dedicated to testing and validating all the work developed, examples of more complex user programs implemented in this thesis will be presented.

## 5.4 Launching an user program

In order to help users launch their offboard programs, which can be challenging especially when adopting the MAVROS communication libraries, a Bash program was created. This Bash program, called **offboard_launcher.sh**, automatically starts all the required processes to perform an experiment, further reducing the difficulty involved in using the real and the simulation environments of the ISR Flying Arena. The **offboard_launcher.sh** program is divided into two parts. In the first part, users define the physical properties of each vehicle and the offboard programs and tools they want to run. In the second part, the Bash code that launches the selected user programs and tools is implemented. The second part is hidden from the user. Fig. 5.8 exhibits the first part of the **offboard_launcher.sh** program, that works as a configuration file. For each vehicle that users want to include in the experiment, it is necessary to create a block of code like the one featured between lines 1 and 16 (or between lines 19 and 34) of Fig. 5.8.

```
1   ### UAV1 ###
2   ns+=("uav1")
3   fcu_url+=("udp://:16001@")
4   mass+=("1.52")
5   radius+=("0.275")
6   height+=("0.100")
7   num_rotors+=("4")
8   thrust_curve+=("iris")
9
10  program_name+=("example.py")
11  program_args+=("")
12  optitrack_pose_forwarder+=("no")
13  neighborhood+=("no")
14  offboard_logs+=("yes")
15  real_time_monitor+=("no")
16  visualization_tool+=("no")
17
18
19  ### UAV2 ###
20  ns+=("uav2")
21  fcu_url+=("udp://:15003@")
22  mass+=("1.3")
23  radius+=("0.295")
24  height+=("0.222")
25  num_rotors+=("4")
26  thrust_curve+=("intel_aero")
27
28  program_name+=("example.py")
29  program_args+=("")
30  optitrack_pose_forwarder+=("no")
31  neighborhood+=("no")
32  offboard_logs+=("yes")
33  real_time_monitor+=("no")
34  visualization_tool+=("no")
```

Figure 5.8: Configuration part of the offboard_launcher.sh program.

After creating a configuration block for each vehicle, users must start by carefully completing each one of the fields presented between lines 2 and 8, that are related to the network address and to the physical properties of the drones. In the example of Fig. 5.8, it is possible to clearly spot the UDP address and the physical attributes of an Iris and of an Intel Aero quadcopters. Users must adapt these values to the airframe they want to fly. Then, through the arguments presented between lines 10 and 16, users must select the offboard programs and/or supporting tools to be launched, for each vehicle, by the **offboard_launcher.sh** program. The **program_name** argument takes the path to the user program with the navigation and control algorithms to be tested. In the code of Fig. 5.8, the example.py file presented in Fig. 5.7 was selected as the user program of both quadcopters. Note that it is also possible to select different programs for each vehicle, or to not select any user program at all, in order to only run the supporting tools. The **offboard_launcher.sh**, when launching an user program, sends to it the parameters selected in lines 2 to 8, as command line arguments. These arguments should be used, in the user program, to instantiate the UAV object that represents the vehicle, as shown in the code of Fig. 5.7. If the developed algorithm features other unique arguments related to the model of the quadcopter, such as controller gains, they can also be passed through the command line via the **program_args** field.

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.