Chapter 5

User Programs

This chapter describes in detail the modules and tools developed to help users create and run offboard programs. Section 5.1 recalls the architecture adopted for the ISR Flying Arena and the role of user programs. The communication libraries that can be used by offboard programs to exchange MAVLink messages with the PX4 are presented and discussed in Section 5.2. Section 5.3 describes the object-oriented approach adopted to develop the input and output modules that enable user programs to receive data and send commands to the PX4. Section 5.4 introduces the user interface designed to launch offboard programs automatically. Finally, Section 5.5 presents a set of tools that were developed to: i) produce flight logs; ii) emulate sensors; iii) reproduce the motion of the vehicles in 3D; and iv) plot the time evolution of physical parameters (such as the three components of the position and velocity of the drones) during the experiments.

5.1 Architecture of the ISR Flying Arena

Fig. 5.1 recalls the architecture adopted for the ISR Flying Arena. The modules highlighted in red are the ones covered in this chapter. They are common to both the real and the simulation environments.

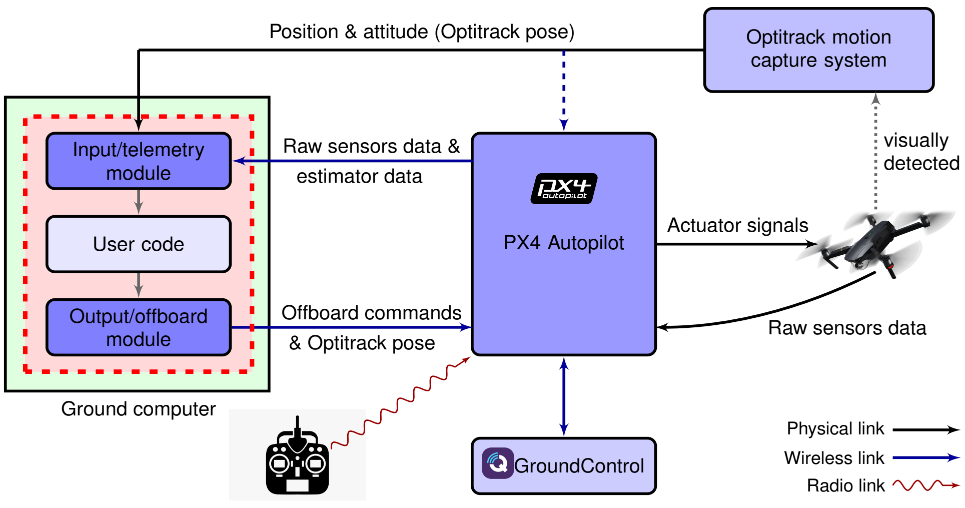

Figure 5.1: Architecture of the ISR Flying Arena, with the modules of the ground computer highlighted.

A user program is composed by the three modules highlighted in Fig. 5.1. However, to perform experiments, a researcher only needs to program the **user code** module with the algorithms of the desired estimators and controllers. The other two modules were developed and coded in this thesis and the researcher just needs to import them. The input or telemetry module features the methods that perform the low-level tasks of subscribing to the information published by the PX4 and by the MOCAP system, and of making that information available to the user in a standardized way. The output or offboard module stores the methods that can be called to send offboard commands and control references to the PX4. Additionally, this module also features the process that sends the external MOCAP data for the PX4 autopilot of the vehicles that do not have a companion computer.

The next chapter presents the communication libraries that were selected to program the input and output modules. For each communication library, a pair of input and output modules were developed. Researchers must employ the pair of modules created from the most advantageous communication library for their experiment. Note that, in the architecture adopted for the ISR Flying Arena, it is advisable to have an independent user program to control each one of the vehicles. Consequently, researchers can employ, for the user programs of different vehicles, pairs of modules coded according to different communication libraries. This shows the flexibility of the developed testing setup.

5.2 Communication libraries

As stated in Chapter 2, the PX4 autopilot communicates with user programs and ground stations using the lightweight MAVLink messaging protocol. MAVLink follows a publish-subscribe design pattern and is suited for applications with narrow communication bandwidth and for resource-constrained systems, with limited RAM and flash memory. Over the last few years, several high-level communication libraries have been created to allow interaction of offboard programs with MAVLink autopilots [26]. For this thesis, the high-level communication libraries selected, MAVROS C++ [27], MAVROS Python [27], MAVSDK C++ [28], and MAVSDK Python [29], were those that present the most complete communication wrappers, which enable access to a greater number of topics and services. These are also actively maintained, robust, and well-tested libraries that are being used in production environments. Consequently, four pairs of input and output modules were developed in this thesis, one for each of the four communication libraries adopted:

- The first two pairs of input and output modules were programmed using the MAVROS Python and the MAVROS C++ communication libraries. Both libraries are based on MAVROS, a package that provides a communication interface between the ROS middleware and MAVLink autopilots. Since ROS is language-independent, both libraries have the same capabilities, that is, they can access exactly the same topics provided by the PX4 autopilot and can request the same services and offboard control actions. However, since Python is an interpreted language, whereas C++ is compiled, user programs employing the MAVROS C++ input and output modules will tend to be faster and lighter. Compared to the MAVSDK libraries, the MAVROS wrappers are computationally heavier because they employ the ROS middleware. Nevertheless, the MAVROS libraries (unlike the MAVSDK ones) have access to the raw information provided by all sensors, including the motion capture system. Consequently, the input and output modules coded with the MAVROS Python and MAVROS C++ wrappers are the recommended ones for running experiments that employ user-developed estimators. The MAVROS modules can also be used to test controllers that work with the state estimates provided by the EKF of the PX4. However, the MAVSDK input and output modules serve this purpose more efficiently, causing less overhead in the system.

- The other two pairs of input and output modules were programmed using the MAVSDK C++ and MAVSDK Python communication libraries. The MAVSDK APIs are computationally lighter than the MAVROS wrappers and can send the same offboard commands to the PX4 autopilot. However, the MAVSDK libraries do not offer methods to access the raw information provided by all sensors. Moreover, the input modules coded with these APIs are unable to access the data coming from the motion capture system, that in this thesis is obtained using the ROS middleware. Taking all these aspects into account, user programs employing the input and output modules coded with the MAVSDK communication libraries are recommended for testing controllers that use the state estimates provided by the EKF of PX4. This is so because these libraries are computationally more efficient than the MAVROS ones and run on computers with very limited system resources. In contrast, the MAVSDK input and output modules are unsuitable for running experiments that employ user-developed estimators since the MAVSDK APIs do not currently offer solutions to access all the required sensor measurements.

In summary, to test offboard programs that resort to user-developed estimators or that utilize packages of the ROS middleware, it is necessary to employ the input and output modules developed with the MAVROS communication libraries. To test offboard controllers with minimum overhead or in computers with very limited system resources, as the first Raspberry Pi models, it is necessary to use the input and output modules coded with the MAVSDK communication libraries. In the remaining cases, the user can employ any of the libraries, remembering that the different solutions create different levels of overhead in the system, as shown in Fig. 5.2. This figure also highlights the limitations of each library. It is possible to conclude that the four communication libraries employed in this work complement each other and allow the user to choose the programming language, between C++ and Python, that presents the most advantageous tools for its work.

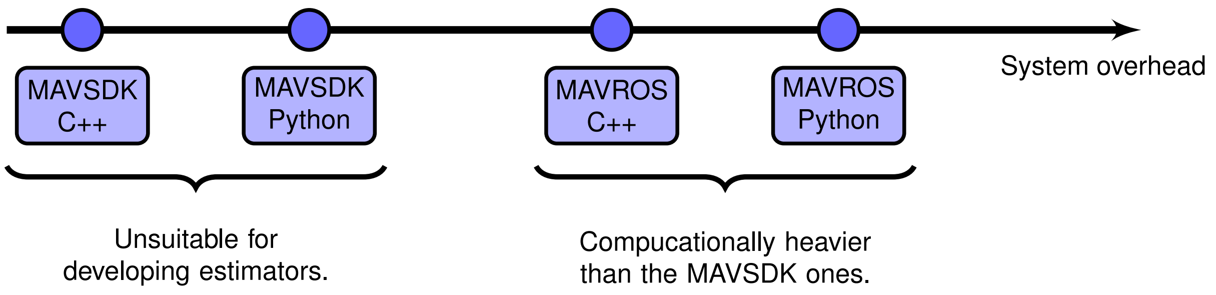
MAVSDK C++
MAVSDK Python
MAVROS C++
MAVROS Python
System overhead
Unsuitable for developing estimators.
Compucationally heavier than the MAVSDK ones.

Figure 5.2: The four different communication libraries represented in increasing order of generated system overhead.

It is important to note that, during the coding of the four pairs of input and output modules of the ISR Flying Arena, the same user-interface (name/arguments) was kept for the methods and functions that perform the same task. For example, the method that arms a vehicle is called **arm_drone** in all four output modules, regardless of the communication library that was adopted to program them. This means that the user can code the desired navigation and control algorithms without worrying about which communication library to adopt, because switching to another is an almost instantaneous procedure. For instance, if researchers have developed a program to test a control algorithm using the MAVROS C++ input and output modules, and later decide that it is more advantageous to use MAVSDK C++, they only need to perform the following two steps, as represented in the code of Figures 5.3 and 5.4:

1. First, they must change the include statement shown in line 1 of Figures 5.3 and 5.4, in order to import the MAVSDK input and output modules into the program, instead of the MAVROS ones.

2. Then, since the MAVSDK C++ library does not use ROS tools, they must remove the code of line 4 of Figure 5.3, which creates and initializes a ROS node.

```
1  #include "uav_mavros.h"
2
3  int main(int argc, char **argv){
4      ros::init(argc, argv, "usr_node");
5
6      // Control algorithm
7  }
```

Figure 5.3: Simplified user program employing the MAVROS C++ modules.

```
1  #include "uav_mavsdk.h"
2
3  int main(int argc, char **argv){
4
5
6      // Control algorithm
7  }
```

Figure 5.4: Simplified user program employing the MAVSDK C++ modules.

It is sufficient to perform these two steps, regardless of the complexity of the implemented algorithms. This is because it is not necessary to change the algorithms: although the user is importing different input and output modules, the methods have the same user-visible interface (the same name and arguments) and perform the same well-defined task in both the MAVROS and MAVSDK versions of these modules. The methods only differ in the internal implementation. In the next sections, the object-oriented approach adopted to allow the easy development of user programs will be explained in-depth. There, the methods that were coded for the input and output modules will be also presented.

As a last note, the MAVSDK Python wrappers currently available in the official website do not allow to connect to more than one vehicle simultaneously. With that in mind, the connection methods of this API were changed in this thesis, to allow each computer to communicate with more than one drone at the same time. Therefore, the user must install the MAVSDK Python version found in the digital repository that complements this thesis, instead of the official version. The fact that an open-source communication library, used in commercial vehicles and applications, has been improved in this thesis, gives credibility to the input and output models developed because it shows that a deep level of knowledge has been acquired regarding the communication with the PX4 autopilot.

5.3 Object-Oriented approach

One of the goals of this thesis is to develop a group of software programs that enable the rapid testing of navigation and control algorithms in the ISR Flying Arena. These software solutions should: i) provide a set of functions that allow access to the flying capabilities of the vehicles; ii) handle the communication of user programs with the other systems of the arena; and iii) automate the low-level task of processing the data received. For example, if a researcher needs to test a PID controller, the devised software solutions should automatically connect the user program to the PX4, subscribe to the data provided by the autopilot and the motion capture system, and make this information available to the user in SI units. Additionally, these software programs should also offer a set of functions that allow the sending of control references (such as position, attitude, and thrust setpoints) to the PX4. Thus, the researcher would only have to code the PID control loop and tune the proportional, integral, and derivative gains.

In the devised architecture for the ISR Flying Arena, the software programs described correspond to the input and output modules. Since the MAVROS and MAVSDK communication libraries are available in C++ and Python, two multi-paradigm languages that support object-oriented programming, it was decided to use classes to implement and hide from the user the input and output modules and to, simultaneously, provide abstraction and encapsulation of the vehicles:

- Abstraction consists in handling complexity by hiding all the unnecessary details from the user. That enables the user to develop more complex programs on top of the provided abstraction without the need to understand all the hidden complexity. By implementing classes to provide abstraction of the vehicles, researchers are allowed to quickly and effortlessly perform rapid experiments in the ISR Flying Arena because they only have to code the navigation or control algorithms they intend to test. All the remaining tasks are hidden in the class, implemented in the background, and researchers can take advantage of them without knowing how they work in detail.

- Encapsulation consists in binding together, in a single entity, data and the functions that manipulate that data. By encapsulating in a class the variables that store the state of a vehicle along with the functions that update the values of those variables, we are organizing the code in a clean, logical, and structured way. Moreover, both the data and the functions are kept safe from outside interference and misuse. Encapsulation also allows to change the implementation of the methods of a class at any time, without worrying about the code that uses the class. This property allows, as seen in Section 5.2, to change the communication library used to interact with the PX4, without the need to modify the code of the control and navigation algorithms. This is because when users switch from the MAVROS to the MAVSDK libraries and vice-versa, the only aspect that changes is the implementation of the methods of the class, and not their name or function in the system.

In conclusion, since vehicles are physical entities represented by both data and behavior, it was selected to model them as objects. This allows to take advantage of the properties of the object-oriented programming paradigm to implement the input and output modules and hide their complexity from the user. In the next sections, the classes created with this approach will be presented and analyzed.

5.3.1 UAV class

The first class created in this work was the UAV class. This is the core class of the developed framework and its function is to represent a vehicle. Consequently, the UAV class encapsulates:

1. A set of variables that store the raw measurements of the sensors of the drone, including the position and attitude measurements provided by the motion capture system. This corresponds to the information that users need to code their own estimators.

2. A set of variables that store the current state of the vehicle provided by the extended Kalman filter of the PX4 autopilot. This state can be used for rapid deployment of controllers and to validate the results of the estimators developed by the user, given that the EKF of the PX4 is extensively tested and is employed in commercial products.

3. A set of variables with the last values sent to the actuators (motors and control devices) of the drone. This information is important for debugging.

4. A set of variables with physical properties and the flight status of the vehicle. These parameters inform the user, for instance, if the vehicle is armed, if it is landed, what is the current flight mode, what is the mass of the drone, and how much battery is still available.

5. A set of methods responsible for subscribing to the data provided by the PX4 and the motion capture system, and for keeping the set of variables mentioned in the previous points up to date. This set of methods form the input module.

6. A set of methods that give the user access to the flying capabilities of the drone and allow user programs to send offboard commands and control references to the PX4 of the vehicle. This set of methods correspond to the output module.

In order to organize the code in a logical and structured way and in order to respect the modular design adopted for the ISR Flying Arena, each one of these sets of variables and methods is stored in a different class. Consequently, six new classes were created to support the core UAV class and each one of them has a rigorous and well-defined function. Fig. 5.5 features a diagram that describes all these new classes and how they support and relate to the main UAV class. This diagram shows that there is a clear separation between the different types of variables and methods. For instance, the variables that store the raw measurements of the sensors are stored in a completely different data structure from the one that stores the state of the drone provided by the EKF of the PX4. Similarly, the methods of the input module, that receive and process data, are gathered in an independent class from the one that stores the methods of the output module, that send offboard commands to the vehicle. Moreover, the scheme shows that all information is accessible through the UAV class, given that it inherits all the methods of the TELEMETRY and OFFBOARD classes and that it starts an instance of the SENSORS, EKF, ACTUATORS, and DRONE_INFO classes. This eases the use of the capabilities of the ISR Flying Arena because the user only needs to create an object of the UAV class (instead of an object of each of the other six classes) to have access to the variables and methods that represent and interact with a vehicle.

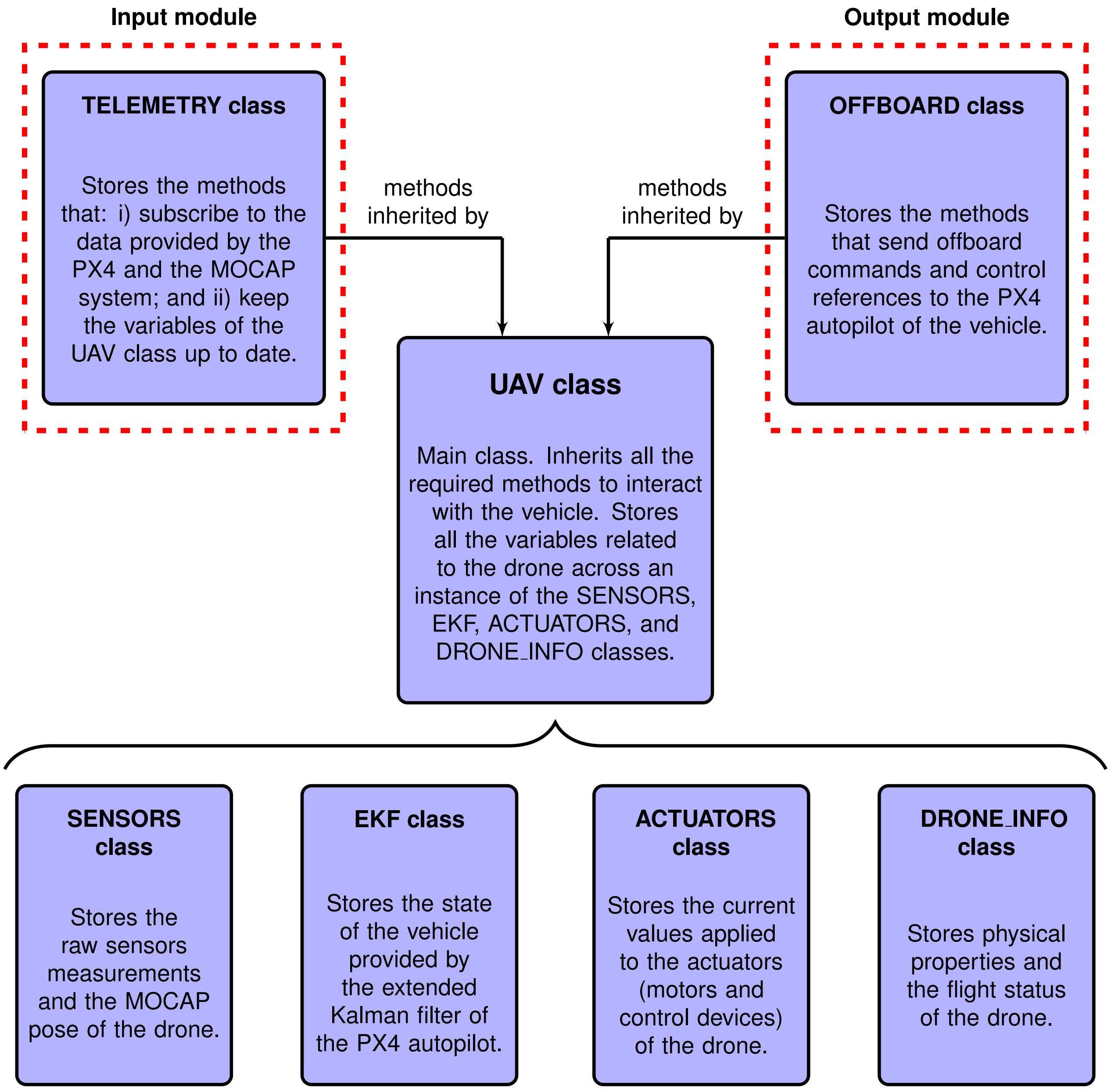
Input module
Output module
TELEMETRY class
Stores the methods that: i) subscribe to the data provided by the PX4 and the MOCAP system; and ii) keep the variables of the UAV class up to date.
methods inherited by
methods inherited by
OFFBOARD class
Stores the methods that send offboard commands and control references to the PX4 autopilot of the vehicle.
UAV class
Main class. Inherits all the required methods to interact with the vehicle. Stores all the variables related to the drone across an instance of the SENSORS, EKF, ACTUATORS, and DRONE_INFO classes.
SENSORS class
Stores the raw sensors measurements and the MOCAP pose of the drone.
EKF class
Stores the state of the vehicle provided by the extended Kalman filter of the PX4 autopilot.
ACTUATORS class
Stores the current values applied to the actuators (motors and control devices) of the drone.
DRONE_INFO class
Stores physical properties and the flight status of the drone.

Figure 5.5: Description of the classes developed to help users create offboard programs.

With the purpose of clearly exposing how the UAV class was implemented and how it enables users to rapidly test and validate control and navigation solutions, the constructor of this class is presented in Fig. 5.6. The exhibited code corresponds to the MAVROS Python version of the constructor. It is important to note that, since two different programming languages were selected for this thesis, the SENSORS, EKF, ACTUATORS, and DRONE_INFO classes had to be coded twice, once in Python and once in C++. Similarly, since the users have the option of employing four distinct communication libraries (MAVROS C++, MAVROS Python, MAVSDK C++, and MAVSDK Python), the constructor of the UAV class, as well as the methods of the TELEMETRY and OFFBOARD classes, were implemented four times, each time according to a different communication API.

```python
1  class UAV (TELEMETRY, OFFBOARD):
2  
3     def __init__(self, drone_ns, mass, radius, height, num_rotors, thrust_curve):
4        """
5        Constructor of the UAV class. Starts a background thread responsible for keeping
6        all the variables of the UAV class up to date. Awaits until the connection with
7        the drone is established.
8  
9        Parameters
10       ----------
11       drone_ns : str
12          ROS namespace where the data from the PX4 and the MOCAP system is encapsulated.
13       mass : float
14          Mass of the drone.
15       radius : float
16          Radius of the drone.
17       height : float
18          Height of the drone.
19       num_rotors : int
20          Number of rotors of the drone.
21       thrust_curve : str
22          Thrust curve of the drone.
23       """
24       self.sen = SENSORS()
25       self.ekf = EKF()
26       self.act = ACTUATORS()
27       self.info = DRONE_INFO()
28 
29       self.info.drone_ns = drone_ns
30       self.info.mass = mass
31       self.info.radius = radius
32       self.info.height = height
33       self.info.num_rotors = num_rotors
34       self.info.thrust_curve = thrust_curve
35 
36       self.telemetry_thread = threading.Thread(target=self.init_telemetry, daemon=True)
37       self.telemetry_thread.start()
38 
39       print("\nConnecting to the drone...")
40       while self.info.is_connected==False:
41          rate=rospy.Rate(2); rate.sleep()
42       print("Connection established!")
```

Figure 5.6: MAVROS Python code of the constructor of the UAV class.

From the analysis of the definition and the constructor of the UAV class, presented in Fig. 5.6, it is possible to infer the following:

- First, that the UAV class inherits the methods of the TELEMETRY and OFFBOARD classes through the multiple inheritance statement of line 1. In the same way, from lines 24 to 27, it is possible to recognize the initialization of the objects that store, in a modular way, the information provided by the PX4 autopilot and the motion capture system. In other words, the first point that can be inferred is how the structure presented in the diagram of Fig. 5.5 is coded.

- Second, that the user, to initialize an instance of the UAV class, needs to assign values to a set of arguments required by the constructor. These arguments include a set of physical properties of the specific multicopter model to be used, such as its mass and dimensions. In this thesis, for each of the multicopter models available, a subclass of the UAV class could have been created, in which the physical parameters of the multicopters were already pre-assigned. For instance, a subclass called INTEL_AERO could have been created, with the physical properties of this quadrotor model already defined. However, this approach was not selected because it would have significant disadvantages compared to using only the UAV class and its parameterized constructor. The first drawback is that users would have to create a new subclass whenever they would want to use a new multicopter model. The second drawback would be that, whenever an implemented algorithm was applied to a different multicopter model, the user would have to change, in the code, the subclass employed. This is not a problem when using the UAV class and its parameterized constructor, because the physical properties of the the multicopter are passed as command line arguments (as it will be further explained in Section 5.4) and, therefore, the code developed is independent from the multicopter used. This also facilitates the process of moving from simulation to real testing. The last disadvantage would be that, when using subclasses with the physical properties of the vehicles already pre-assigned, users would not develop the habit of introducing and double-checking the values of these arguments. Consequently, whenever users would change the battery, propellers, or landing gear of the drone, which could mean a change in the mass and dimensions of the vehicle, they would be prone to forget to change the variables that store the values of these physical quantities, that would be hidden and encapsulated inside the hypothetical subclass. This small mistake could ultimately cause a collision with another vehicle or with the physical limits of the arena.

- Finally, that the methods of the TELEMETRY class start automatically in the background as soon as an object of the UAV class is declared. This is due to the code of lines 36 and 37 of Fig. 5.6, where a thread that initiates the methods that keep the variables of the UAV class up to date is launched. This is an important feature because it means that the complexity of the TELEMETRY class is hidden from the users and they can take advantage of it without having to understand how the communication libraries work and how the data received is processed. Thus, using the UAV class to develop offboard programs is straightforward. After declaring an object of the UAV class, users can develop their algorithms knowing that the variables related to that drone are up to date and available in the SENSORS, EKF, ACTUATORS, and DRONE_INFO objects and knowing that, whenever they need to send commands and control references to the drone, they only have to call the respective method of the OFFBOARD class, that is inherited by the UAV class.

To conclude this section, it is disclosed that, in the constructor of the UAV class of the MAVSDK modules, an object of the SENSORS class is not instantiated, in order to prevent researchers from employing these communication libraries when testing algorithms that use raw sensors measurements. As stated in Section 5.2, the MAVSDK modules have limited access to the raw sensors readings and to the motion capture system data, so they are not suited for testing such algorithms. In fact, the MAVSDK modules were specifically developed for researchers testing, with minimum overhead and on any computer regardless of its resources, algorithms that only employ the state estimates provided by the EKF of the PX4 autopilot. It is recalled that the code developed by users to implement navigation and control solutions is independent from the communication library used, because the methods that perform a given function have the same header in the modules of all four communication modules. Therefore, whenever users employ the MAVSDK modules in unsuitable situations, they can switch to the MAVROS modules by simply modifying two lines of code, as already demonstrated in Section 5.2. Since the MAVROS modules have access to the information published by all systems of the ISR Flying Arena and enable the use of ROS tools, they are the best option for testing the majority of the applications.

In the next sections, a brief analysis of the classes that support the main UAV class will be carried out. For a more detailed insight into these classes, the documentation available in the digital repository can be consulted.

5.3.2 TELEMETRY class as the input module

The devised architecture for the ISR Flying Arena, which has been discussed throughout this thesis, features an input module responsible for receiving and processing the data provided by the PX4 autopilot and the motion capture system. Fig. 5.1, presented at the beginning of this chapter, recalls this architecture and the purpose of the input module. In the object-oriented approach adopted to enable the rapid development of offboard navigation and control solutions, the input module corresponds to the TELEMETRY class. The methods of the TELEMETRY class, inherited and launched by the objects of the UAV class as soon as they are initialized, subscribe to the information published by all systems of the ISR Flying Arena and make this data accessible to the user as variables of the SENSORS, EKF, ACTUATORS, and DRONE_INFO classes, which correspond exactly to the responsibilities defined for the input module.

In a lower-level, the methods of the TELEMETRY class are implemented according to the two following main steps:

- First, the capabilities of the communication APIs are used to subscribe to the telemetry updates provided by the PX4 autopilot and the MOCAP system, and to associate a callback function that, whenever a new message is received, processes the new information. For example, in the case of the MAVROS APIs, for each important ROS topic provided by the PX4 and the MOCAP system, a ROS subscriber is declared. These subscribers associate a callback function to each ROS topic. Consequently, whenever a new message is published in one of those topics, the associated callback function is invoked with the new message as the argument.
- Then, in the callback functions, the physical quantities contained in the new message are converted, if necessary, to SI units and to the NED coordinate system adopted for the ISR Flying Arena. The converted values are finally stored in the corresponding variables of the SENSORS, EKF, ACTUATORS, or DRONE_INFO objects.

It should be noted that the TELEMETRY class, just like the OFFBOARD class, is a product of the modular design adopted for the ISR Flying Arena. These two classes only exist so that the methods of the UAV class that receive information are stored in a different and independent data structure from the methods that send information to the PX4 autopilots, as required by the fundamentals of a modular architecture. Therefore, they should not be interpreted as superclasses of the UAV class, but as assistant data structures that guarantee the logical organization of the code.

5.3.3 OFFBOARD class as the output module

The devised architecture for the ISR Flying Arena, recalled in Fig. 5.1, comprises also an output module responsible for providing solutions for sending offboard commands, control references, and the MOCAP pose (when there is not an onboard companion computer) to the PX4 autopilot of the vehicles. In the object-oriented approach adopted for this thesis, the output module corresponds to the OFFBOARD class. This class comprises a set of methods, inherited by the UAV objects, that automate the procedure of sending data and instructions (such as arming commands, takeoff requests, and attitude and thrust references) to the PX4 autopilot of the drones, which correspond to the exact responsibilities established for the output module.

In a lower-level, the methods of the OFFBOARD class are implemented as described in the following points:

- First, the information to be sent to the PX4 autopilot is converted into a data structure (a message) compatible with the communication library selected by the user. In this step, the information is also converted, if it corresponds to a physical quantity, to the default units and to the default coordinate frame employed by the communication API.
- Then, using the capabilities of the selected communication library, the message is sent to the appropriate topic of the PX4 autopilot, as a MAVLink stream.

As an example, to send a position setpoint using the MAVROS APIs, it is necessary to convert the position passed by the user in NED coordinates to the ENU coordinate system adopted by ROS. Then, this position is integrated into a PoseStamped message [30]. Finally, a ROS publisher sends the PoseStamped message, that contains the position setpoint, to the appropriate topic of the PX4 autopilot.

The methods of the OFFBOARD class are essential for the rapid development of navigation and control solutions in the devised setup because they relieve users from implementing these time-consuming tasks of converting information into messages and sending them to the PX4 autopilot according to the norms of a communication library. If researchers instantiate an object of the UAV class called **drone1**, they can arm the drone simply by calling the corresponding class method through the code **drone1.arm_drone()**. This method will automatically create and send the message with the arming command to the PX4 autopilot of the vehicle, relieving the user of implementing those tasks. Another great advantage of using the methods of the OFFBOARD class is that they were carefully tested during the development of this thesis, which guarantees that the user programs will not stop during execution due to an error of implementation in the communication functions.

5.3.4 SENSORS, EKF, ACTUATORS, and DRONE_INFO classes

This section features a brief description of the classes that store, in a structured way, the set of variables that contain the data related to a vehicle.

- The SENSORS class comprises the set of variables that store the raw measurements of the sensors of the drone, as well as the set of variables that store the position and attitude measurements provided by the motion capture system. In order to respect the modular design adopted for the software modules of the ISR Flying Arena, new classes were created to store the information of each sensor. For example, the data provided by the inertial measurement unit, by the GPS sensor, and by the motion capture system are stored in objects of the IMU, GPS, and MOCAP classes, respectively. As a result, users know precisely what is the source of the information they are working with. Accessing the data stored in these classes is straightforward. For instance, after successfully instantiating an object of the UAV class named **drone1**, the raw acceleration measured by the IMU is available in the **drone1.sen.imu.acc_body** variable and the position of the vehicle according to the motion capture system is stored in the **drone1.sen.mocap.pos** variable.

- The EKF class contains the set of variables that store the current state of the vehicle provided by the extended Kalman filter of the PX4 autopilot. Since, in this case, all information comes from the same source, it was not necessary to create support classes. In order to give examples of how users can access the information stored in the EKF class, consider an object of the UAV class called **drone1**. The position of the vehicle provided by the internal estimator of the PX4 is available in the **drone1.ekf.pos** variable, the attitude in Euler angles is available in the **drone1.ekf.att_euler** variable, and the same attitude represented as a quaternion is stored in the **drone1.ekf.att_q** variable.

- The ACTUATORS class contains only two arrays. The first one, that for an object of the UAV class named **drone1** corresponds to the **drone1.act.active** variable, indicates the working motors and servos of the vehicle. The second one, that corresponds to the **drone1.act.output** variable, consists in the normalized values (0 to 1 or -1 to 1) applied to those motors and servos.

- The DRONE_INFO class comprises a set of variables with physical properties and the flight status of the vehicle. Once again, it is simple to access the information provided by the class. After instantiating an UAV object called **drone1**, the mass of the vehicle is available in the **drone1.info.mass** variable, the remaining battery is stored in the **drone1.info.battery** variable, and the current flight mode of the PX4 autopilot is saved in the **drone1.info.flight_mode** variable.

Over the examples presented above, only a part of the variables comprised in each class were introduced. A complete and descriptive list of all the variables can be consulted in the documentation available in the digital repository that complements this thesis. The next section explains how the user, through the properties of the objects of the UAV class, can easily and effortlessly code offboard programs.

5.3.5 Example of an user program employing the UAV class

In order to show that the UAV class completely hides the complexity associated with the communication and with the use of all the systems of the ISR Flying Arena, Fig. 5.7 presents an example of a simple offboard program that sends position and yaw setpoints to a PX4 autopilot. This user program commands the vehicle to climb one meter and move one meter North and East from its initial position, while also commanding the drone to keep its initial yaw. This example employs the MAVROS Python communication library. It is recalled that the rapid development and prototyping environment was designed according to a NED (North-East-Down) coordinate system.

```
 1  import sys
 2  import rospy
 3  import numpy as np
 4  from uav_mavros import UAV
 5
 6
 7  if __name__ == "__main__":
 8
 9      rospy.init_node(’example_py’, anonymous=True)
10
11      uav = UAV(sys.argv[1], sys.argv[2], sys.argv[3], sys.argv[4], sys.argv[5],
12                sys.argv[6])
13
14      uav.start_offboard_mission()
15
16      pos = uav.ekf.pos + np.array([[1],[1],[-1]])
17      yaw = uav.ekf.att_euler[2]
18      time = 10
19
20      uav.set_pos_yaw(pos, yaw, time)
21
22      uav.auto_land()
```

Figure 5.7: Code of the example.py program.

This example starts with the required libraries and modules being imported and with a ROS node being initialized. This ROS node is necessary because the MAVROS Python API is being used. Then, in line 11, an object of the UAV class, named **uav**, is declared. The arguments with which the **uav** object is instantiated are passed through the command line, which allows this program to be independent from the multirotor used. The declaration of the **uav** object, in line 11, causes the methods that communicate with the PX4 autopilot and with the motion capture system to be launched in the background. These background methods keep the variables of the **uav** object up to date. This means that all the complexity associated with receiving and processing data is hidden from the users behind a single line of code.

The method shown in line 14 arms the drone and changes the flight mode of the PX4 autopilot to ’offboard’. Therefore, it prepares the PX4 of the vehicle to receive references of position, velocity, attitude, thrust, etc. This single method relieves the user from creating a set of messages containing the commands for arming and changing the flight mode, and from sending these messages to the respective topics of the PX4 autopilot, according to the MAVROS Python communication library.

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.