Rapid Development and Prototyping Environment for Testing of Unmanned Aerial Vehicles

Tiago Alexandre da Silva Oliveira

Thesis to obtain the Master of Science Degree in

Aerospace Engineering

Supervisor(s):	Prof. Pedro Tiago Martins Batista
	Prof. Rita Maria Mendes de Almeida Correia da Cunha

Examination Committee

Chairperson: Prof. José Fernando Alves da Silva
Supervisor: Prof. Pedro Tiago Martins Batista
Member of the Committee: Prof. Alexandre José Malheiro Bernardino

January 2021

Acknowledgments

First, I would like to express my deep gratitude to my supervisors Dr. Pedro Batista, Dr. Rita Cunha, and Dr. David Cabecinhas for their guidance, patience, and continuous support throughout this project. I would also like to thank my fellow aerospace engineering colleague Pedro Trindade for introducing me to the aerial robotics laboratory and for helping me perform my first flights.

To my family and friends, thank you for the encouragement, unconditional support, and for all the good moments shared together.

Finally, I wish to express my gratitude to my girlfriend, Carolina, for reading and reviewing this document and for always being by my side.

This work was supported by the Fundação para a Ciência e a Tecnologia (FCT) through FCT project DECENTER [LISBOA-01-0145-FEDER- 029605], funded by the Lisboa 2020 and PIDDAC programs.

Resumo

Nesta dissertação é projetada e implementada, na ISR Flying Arena, uma plataforma multi-veículo que permite o rápido desenvolvimento e teste de soluções de controlo e navegação para aeronaves não tripuladas. A arquitetura concebida para a plataforma de testes integra: i) um sistema ótico de captura de movimento que determina a posição e atitude dos veículos; ii) computadores offboard responsáveis pelas tarefas de comunicação e pela execução dos programas do utilizador; e iii) múltiplos quadrotores. Com o intuito de providenciar abstração dos veículos e de automatizar a comunicação entre sistemas, foi desenvolvido um conjunto de módulos de software recorrendo a um paradigma de programação por objetos. Estes módulos encarregam-se de todas as rotinas de voo de baixo nível. Um grupo adicional de ferramentas foi também criado para monitorizar voos e gerar os respetivos logs. De forma a possibilitar o teste dos algoritmos implementados antes dos ensaios com veículos físicos, foi concebido um ambiente de simulação configurável, que incluí uma solução para emular o sistema de captura de movimento. A plataforma de testes possibilita a coexistência de quadrotores físicos e simulados, permitindo experiências em condições físicamente inviáveis na ISR Flying Arena. No final foram implementadas com sucesso diversas soluções de controlo, incluindo um algortimo de controlo de formações, validando a arquitetura adotada e atestando a sua robustez e escalabilidade. A plataforma multi-veículo desenvolvida é um elemento chave para a educação e futuras investigações na área da robótica aérea, tendo já sido utilizada no processo de validação experimental das teses de mestrado de outros alunos.

Palavras-chave: ISR Flying Arena, Multirotores, Plataforma de Testes de UAVs, Robótica Aérea, GNC

Abstract

In this dissertation, an indoor multi-vehicle rapid prototyping platform is designed and implemented at the ISR Flying Arena, to support the development and testing of control and navigation solutions for unmanned aerial vehicles. The hardware architecture devised for the prototyping environment comprises: i) an optical motion capture system providing vehicle position and attitude ground-truth; ii) a set of offboard computers managing communication between systems and running user programs; and iii) multiple quadrotors. In order to provide abstraction of the vehicles and to automate the communication between systems through reliable protocols, a set of software modules were programmed using an object-oriented approach. These modules relieve the user from implementing low-level flight routines and communication tasks. An additional group of software tools was also created to allow offboard flight logging and monitoring. With the purpose of enabling testing of the deployed algorithms before experiments with physical vehicles, a fully configurable and easy-to-use simulation environment, including a solution to emulate a motion capture system, was also developed. The devised setup allows for a mixed environment of physical and simulated quadcopters, extending testing to conditions that are physically unfeasible at the ISR Flying Arena. In the end, several control solutions, including a formation-control algorithm, were deployed and tested, validating the adopted architecture and showcasing its robustness and scalability. The created prototyping platform is a key enabler of future research and education in aerial robotics, having already been used in the experimental validation process performed within the scope of the MSc Theses of other students.

Keywords: ISR Flying Arena, Multirotors, UAV Testing Environment, Aerial Robotics, GNC

Contents

List of Tables

List of Figures

Glossary

EKF	Extended Kalman Filter.
FastRTPS	Fast Real Time Publish Subscribe.
GNC	Guidance, Navigation, and Control.
GPS	Global Positioning System.
HITL	Hardware-in-the-Loop.
IMU	Inertial Measurement Unit.
ISR	Institute for Systems and Robotics.
RAVEN	Real-time indoor Autonomous Vehicle ENvironment.
ROS	Robot Operating System.
RTOS	Real Time Operating System.
SITL	Software-in-the-Loop.
UAV	Unmanned Aerial Vehicle.
VTOL	Vertical Take-Off and Landing.

Chapter 1

Introduction

1.1 Motivation

The use and demand of Unmanned Aerial Vehicles (UAVs) have been rapidly increasing across different industries due to their autonomous flying capabilities, onboard decision making power, ability to communicate primary and mission-related information in real-time, possibility to carry different types of payloads, and their aptness to perform hazardous and dull tasks in a vast range of scenarios [1]. Consequently, the research and development of mission-oriented UAVs has grown at an accelerated rate, followed and fueled by consistent advances in air-frame materials, propulsion systems, avionics, sensors, and power sources. These vehicles have become vital in the military for surveillance, reconnaissance, and combat operations, and in the commercial sector for tasks such as aerial imaging, infrastructure inspection, border management, precision agriculture, and product delivery. As a result, the UAV market was estimated at 19.3 billion dollars in 2019 and is projected to reach 45.8 billions by 2025 [2].

To support development of mission-oriented UAVs and to ease academic research of cutting-edge guidance, navigation, and control solutions, universities are designing indoor multi-vehicle dedicated testbeds that surpass weather and daylight constraints. Since an optical motion capture system is mandatory in such testbeds, for indoor position and attitude estimation, these spaces are also suitable for developing machine learning and computer vision algorithms, which can later be incorporated into the vehicles enabling them to do their designated task more efficiently. Precise aggressive maneuvers [3], iterative motion learning [4], and cooperative ball throwing and catching [5] are just a few examples of scientific breakthroughs that were conducted in and benefited from these dedicated spaces. With the recent advances in micro and nano UAVs, flying arenas have also been chosen as the platform for the test of new autonomous formation flight solutions.

Despite being very versatile and crucial for scientific and technological progress in aerial robotics, UAV-devoted testbeds comprise a powerful simulator and a vast amount of dedicated software and hardware whose use requires practice, good fundamentals of computer science, substantial knowledge on the communication protocols, and continued reconfiguration. These aspects form a barrier to their usage that can slow or inhibit research and stagnate innovation [6].

Therefore, the driving force behind this thesis is to encourage more research in aerial robotics and the development of more courses and laboratory classes involving UAVs by not only settling a dedicated testbed at the Institute for Systems and Robotics (ISR), equipping its researchers and students with a powerful tool for extensive testing and validation of new UAV-related algorithms, but by also eliminating its inherent working barriers, by relieving the users of deep Linux and programming knowledge, background in hardware, and by automating all communication between systems.

1.2 Objectives

The main objective of the present work is to design and implement a multi-vehicle rapid prototyping platform for development and testing of navigation and control solutions at the ISR Flying Arena. This prototyping framework must include:

- An experimental indoor environment, consisting of: i) an optical motion capture system providing position and attitude ground-truth; ii) multiple ready-to-fly quadrotors; and iii) a set of external computers handling communication between systems, acting as ground stations, and running the desired user algorithms.
- A fully configurable simulation environment, enabling testing and validation before experiments with physical vehicles. The simulator must have software-in-the-loop (SITL) and hardware-in-the-loop (HITL) capabilities, including also a solution to emulate the motion capture system.

The developed testbed must address scalability and automate the communication between systems.

1.3 Related Work

The first impactful indoor UAV-oriented testbed documented in literature was the MIT RAVEN (Real-time indoor Autonomous Vehicle ENvironment). It was designed in 2005 to overcome the constraints of outdoor test platforms that can only be flown safely during daytime, with good visibility and favorable meteorological conditions, and that require a large support team, which makes the testing process very expensive and difficult to coordinate [7].

The MIT RAVEN enables the rapid prototyping of guidance, coordination, and control algorithms of different types of vehicles, such as autonomous multicopters, fixed-wing drones, and ground-based rovers. An experiment involving the three types of vehicles can be managed by only a single operator, presenting a substantially reduced logistical cost compared to an outdoor test. However, the system has a limit of 10 simultaneous vehicles. This testbed also allows control of the environmental conditions of the indoor space, that can range between ideal to wind induced, by resorting to blowers and fans [8].

The MIT RAVEN architecture is presented in Fig. 1.1. The vehicles are confined to an arena of dimensions 10m $\times$ 8m $\times$ 4m. Their position and attitude are continually and accurately determined by a Vicon MX motion capture system, that yields the data to ground computers - one for each vehicle -

through a reliable Ethernet connection. In this architecture, all computations are done offboard. The position and attitude data are treated by an automatic input process that sends the result to the control module, where the vehicle controller is implemented and the flight control commands are obtained. Finally, the computed commands are sent to the vehicles, at rates higher than 50 Hz, via modified RC transmitters. This is a defining feature of the setup because low cost radio-controlled vehicles can be employed with only minor adaptations, enabling the researchers to take risks during flight testing instead of overprotecting the equipment.

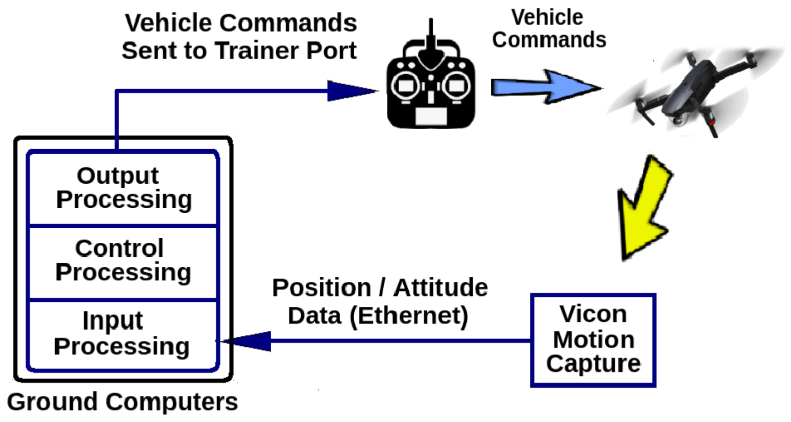


Figure 1.1: Architecture of the MIT RAVEN testbed.

The several control, mission-management, task-allocation and path-planning algorithms validated on RAVEN, in multi-vehicle setups that took under 20 minutes to a single researcher to prepare, prove the efficiency of this testbed [8].

Another well documented aerial robotics platform, essential to a major number of scientific works, is the ETH Zurich's Flying Machine Arena [9]. It was created with a dual-purpose: to be a flexible prototyping environment capable of test, validate, and evaluate new concepts in control and cooperation of UAVs; and to be reliable and robust enough, both software and hardware-wise, for numerous continuous public demonstrations across the globe. Consequently, it comprises not only an indoor facility with dimensions 10m $\times$ 10m $\times$ 10m but also a resilient mobile module of twelve ship-ready cases, weighting 300 kg. Both structures have a modular design so components can be interchanged between them without affecting the remaining systems.

The Flying Machine Arena top level architecture is presented in Fig. 1.2 and, similarly to the RAVEN testbed, comprises global sensing and offboard computing. The position and attitude measurements of the vehicles are provided by the indoor optical motion capture system, typically working at 200Hz, to a station of ground computers. Here, the user code module runs estimation and control algorithms that generate flight commands. Finally, a copilot mode runs a failure detection function that supervises those commands, only allowing the appropriate ones to be sent to the vehicles. All communication is done in a high-frequency, asynchronous way that guarantees nonexistence of delays due to retransmission attempts. To overcome some inconveniences of the absence of synchronization, all sensor data is time-stamped against local hardware clocks.

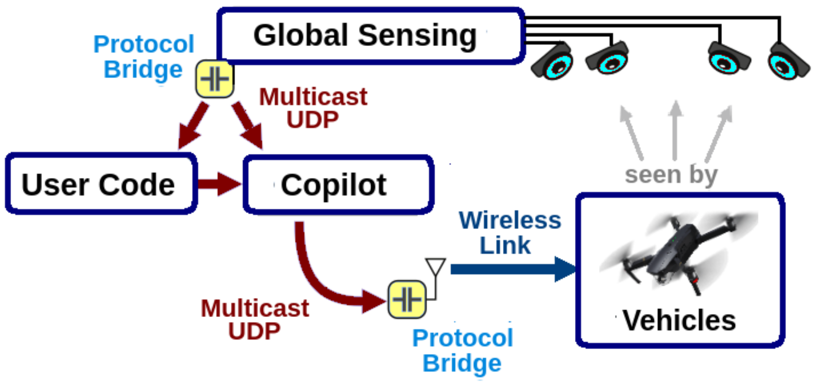


Figure 1.2: Architecture of the Flying Machine Arena.

At a lower level, the Flying Machine Arena reveals a series of interesting features:

- Data logging, managed by an independent process that subscribes to all packets and stores them in a file. All entries are time-stamped and saved in order.
- Playback operation mode, almost indistinguishable from a live test. A process re-sends all the time-stamped entries recorded in the log file to the correct channels, at the appropriate rates.
- Standardize intermodule communication, meaning that every input or output is already corrected, represented in SI units, and is physically meaningful before being sent to another module. This eases debugging and monitoring of the system.
- An onboard microcontroller for each vehicle that generates the control signals to the motors based on the body rates and normalized thrust requested by the offboard modules. There is also a sensor calibration routine implemented in this board that estimates gyroscope bias and propeller efficiency factors when in hover. Additionally, this microcontroller still runs an estimator that relies on the onboard inertial measurements and, periodically, on offboard state estimates, which can be used briefly to land the vehicle in case of loss of connection with the ground computers.
- A simulation environment that reproduces vehicle dynamics by performing numerical integration of the equations of motion. It emulates disturbances and sensor noise, allows the user to add objects such as rackets and balls, and enables the adjustment of the speed of the simulator, to accelerate it during long tests or to slow it down during computational-heavy tasks. It is also possible to configure the seeds of the stochastic processes and, consequently, exactly recreate a simulation - which facilitates breakpoint analysis and debugging.
- Failure modes, guarantying the safety of users and public. If a process in the user mode crashes, the copilot module takes control of the vehicle (see Fig. 1.2), switching to a system default offboard estimator and controller that are constantly running on the background. A low battery failsafe is also present in the copilot module, triggering a landing routine when the battery level is below a predefined threshold. There is also a ”panic command” to shutdown all systems in critical situations.

Research on multi-vehicle transitions, cooperative ball throwing and catching, construction of brick structures, and multi-vehicle dancing have benefited from this indoor platform, proving this architecture is flexible enough for research. The robustness of the Flying Machine Arena and of its mobile module have also been proved, by the more than 100 public demonstrations made in only a single year, across Europe and North America.

Finally, the most modern drone academic research facility in Europe is the Imperial College's Brahmal Vasudevan Aerial Robotics Lab, whose construction was completed at the end of 2017. It was designed to push the university to the lead of the UAV design industry by allowing its researchers to develop, test, and validate a next-generation of flying robots, capable of performing in the most severe environments. With that in mind, the 10m by 6.2m by 5.5m indoor space equipped with 16 Vicon T40 tracking cameras, is capable of simulating different terrains in the air, land, and ocean, and create extreme conditions as fire, smoke, and heat with total security. Aerial construction-bots, equipped with 3D printing technology and capable of helping repair and build structures in natural disaster scenarios, are an example of the type of research projects that are being conducted in this facility [10, 11].

1.4 Contributions

The main contributions of this dissertation are:

- Design of a modular architecture that enables the rapid testing and validation of single-vehicle and multi-vehicle guidance, navigation, and control (GNC) solutions in an indoor environment.
- Implementation of the devised architecture at the ISR Flying Arena, so that it can be used by aerial robotics researchers in the future.
- Implementation of failsafe modes and safety features to protect users and equipment.
- Development of a fully configurable and easy-to-use simulation environment, featuring a solution to emulate a motion capture system.
- A set of software modules that automates the communication between systems through reliable protocols and that automates the low-level tasks required to perform simulations and physical experiments.
- A set of tools for flight logging and monitoring.
- Deployment of GNC algorithms to validate the simulation and the physical experimental environments.
- A digital repository storing the software programs and the documentation [12].

1.5 Thesis Outline

The remainder of the document is structured as follows:

- Chapter 2 presents the architecture of the complete developed system, along with a description of all its components and their interconnections. It also analyses the PX4 autopilot, the core component of the testbed, and explains its advantages and its role in the prototyping environment.
- Chapter 3 describes, in detail, the implementation process of the devised physical tests environment. It also explains how the information provided by the Optitrack cameras is processed and integrated into the testing setup and how the PX4 autopilot was configured, tuned, and merged into the framework.
- In Chapter 4, a fully configurable simulation environment is presented along with the corresponding user interface. The set of processes programmed to automatically launch the simulator with the configurations desired by the user will also be discussed in this stage, coupled with the solution to emulate the motion capture system and a series of strategies to reduce the simulation overhead.
- Chapter 5 presents 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 autopilot. It also introduces the user interface designed to launch offboard programs automatically and a group of tools to assist the user during the tests, such as an automatic log and charting system and global visualization and monitoring tools.
- Chapter 6 describes the set of experiments performed to validate the architecture and the software programs designed for the ISR Flying Arena. In these tests, a set of controllers were deployed and validated, including a formation control algorithm.
- Finally, Chapter 7 states the conclusions and achievements of this work along with a list of possible improvements and additional features for the whole system that can be implemented in future updates.

Chapter 2

System Overview

This Chapter introduces the architecture designed for the rapid prototyping platform. Section 2.1 analyses the PX4 autopilot, the core component of the testbed, and explains its properties and its advantages. Section 2.2 presents the modular architecture adopted for the physical tests environment and for the simulation environment, and describes the communication between systems.

2.1 PX4 Autopilot

UAV autopilot systems allow aerial vehicles to autonomously perform missions requested by an user. Their central process units combine data from inertial sensors such as accelerometers, gyroscopes, and magnetometers with data from external positioning systems like GPS to obtain onboard estimates of the state of the vehicle. This state includes physical vector quantities such as the position, attitude, and linear and angular velocities of the UAV. With this information, the autopilot is capable of running control algorithms to autonomously operate the drone and complete entire missions. UAV autopilots allow manual control of the vehicle in emergencies and have safety modes programmed that are triggered in critical situations, such as when the battery level is dangerously low, to guarantee the security of people and equipment. A user communicates with autopilots via intuitive platforms, so-called, ground control stations, that enable configuration of flight plans and setting of custom parameters.

All mentioned features make autopilots a valuable addition to the traditional aerial robotics testbeds documented in the literature. By automating most flight routines and by abstracting away the low-level communication, actuator control, flight control, and navigation, these systems allow researchers to focus only on the algorithms of their interest. For example, if a group of students is testing and validating a new type of controller, they can use the estimator of the autopilot to obtain the state of the vehicle, without having to spend time implementing one. The same applies to the takeoff and landing procedures. Autopilots also handle communication with sensors and run internal controllers that convert flight commands into actuator signals. Moreover, autopilots can be installed in a vast range of drones, requiring only a quick parameter tuning procedure to adapt to the characteristics of the vehicle. Finally, they have a built-in external connectivity module that substantially eases communication with offboard systems.

In summary, UAV autopilots are commercially available off-the-shelf products that fit and improve aerial robotics testbeds by relieving the users of implementing numerous flight routines, by facilitating communication with offboard modules, and by handling interaction with sensors and actuators. In order to take advantage of all these features, and to design the most complete and user-friendly testing environment possible, it was incorporated a PX4 autopilot in the architecture of the developed system.

The PX4 Autopilot is well suited for the ISR Flying Arena, not only because it is installed in multiple UAVs owned by the Institute for Systems and Robotics but also because [13, 14]:

- It is open source, and as a consequence, its firmware is publicly available, can be modified to perfectly fit the architecture of the tests platform, and can be adapted to specific tests and scenarios. Additionally, it is an economic solution compared to the professional autopilots.
- It provides abstraction of the flight capabilities of the vehicles, which means that it is not necessary to implement specific software for each drone. Moreover, it is configurable, tunable, and compatible with multicopters, fixed wing aircrafts, and VTOL drones. Consequently, it can be implemented in any UAV, independently of its air-frame, which makes the prototyping environment vehicle-agnostic.
- It provides several flight modes with different levels of automation and autopilot assistance. Common but dangerous procedures such as takeoffs and landings can be fully performed by the PX4 autopilot, through the selection of the proper flight mode.
- It comprises core estimators and controllers, already validated by successful flights of thousands of UAVs deployed worldwide, that can be used by researchers to save time and for comparison purposes.
- Its onboard estimators accept external pose input from motion capture systems - a mandatory feature since the ISR Flying Arena is an indoor space and global position sources such as GPS are unavailable or unreliable.
- It has simulation-in-the-loop and hardware-in-the-loop capabilities that enable the simulation and thorough testing of the PX4 autopilot, all of its systems, and their interfaces with outside command software before the field tests.
- It comes with failsafe modes that protect the users and equipment whenever the position estimate is too bad or whenever the vehicle loses connection with an offboard system.
- It is compatible with the ground control station QGroundControl [15] that aides the user in the sensor calibration process, facilitates in-flight monitoring, and enables the users to quickly change internal parameters of the PX4.
- It communicates with exterior modules through a validated messaging protocol called MAVLink, specifically developed for UAV communication. This protocol is available in widely used programming languages such as C++ and Python. Furthermore, it is compatible with a set of robotics libraries and tools called ROS (Robot Operating System).

- It runs a Real Time Operating System (RTOS) that guarantees that the critical flight tasks are completed within a specific range of time. This ensures, for instance, that the motors are actuated in the right moments and that an up-to-date state of the vehicle is always available to the user.

2.1.1 PX4 Architecture

The high-level architecture of the PX4 Autopilot is presented in the diagram of Fig. 2.1. Note that this architecture is divided into two main parts: the **middleware**, a general robotics layer that comprises the storage, external connectivity, and hardware driver blocks, and the **flight stack**, a module responsible for guidance, navigation, and flight control.

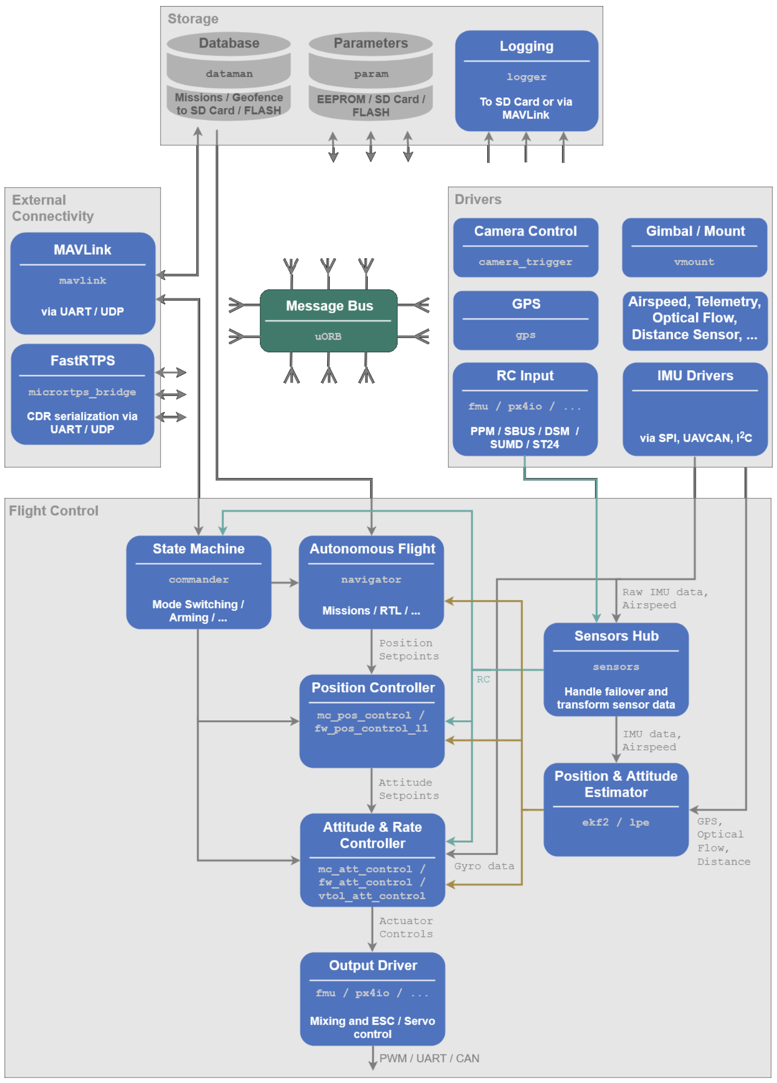


Figure 2.1: PX4 high-level software architecture.

The internal communication between layers, blocks, processes, and threads is done through an asynchronous, publisher/subscriber message bus called uORB, represented by a green box in Fig. 2.1. The uORB messaging system guarantees that all communication is thread-safe. Topic publishing and subscribing can be done from anywhere in the system. Since the uORB message API is user-friendly, researchers can easily develop and run onboard algorithms, which is one more advantage of the PX4.

The communication with offboard systems is handled by the middleware's external connectivity block, primarily through the MAVLink protocol that follows a publish-subscribe scheme and offers lightweight messages and services. The FastRTPS (Fast Real Time Publish Subscribe) communication bridge enables high-performance exchange of uORB messages with offboard systems. It is implemented to offer a direct reliable interface with ROS2. In this thesis, only the MAVLink protocol is used because it is computationally lighter, more stable, tested, and more appropriate to communicate with ground stations than the FastRTPS. The FastRTPS bridge was not considered because, similarly to ROS2, it is in the preliminary stages of development.

In the middleware's storage block runs a logging module that subscribes to the uORB topics that store relevant physical quantities and either saves their values into a log file or sends them via MAVLink to a ground station. The logging rate of each topic is configurable and the system stores not only common variables such as the position and attitude of the vehicle but also more specific ones like vibration levels.

The remaining middleware's block consists in a set of device drivers responsible for decoding the information from the sensors and the peripherals. The PX4 only requires an IMU and a barometer to operate but it also supports fusing of measurements from airspeed sensors, GPS receivers, distance sensors, and optical flow sonars.

The second main part of the PX4 architecture, the flight stack, consists in a set of estimation and control algorithms for multicopters, fixed wing aircrafts, and VTOL drones. Fig. 2.2 presents a closer look of the flight stack.

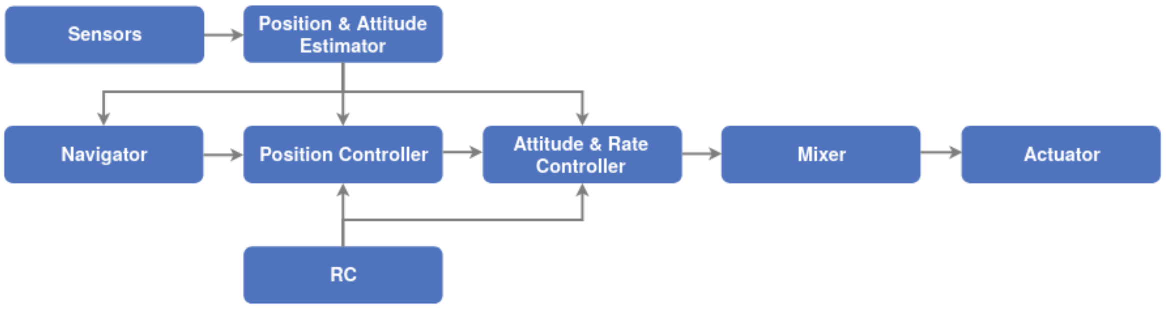


Figure 2.2: PX4 flight stack architecture.

In the position and attitude estimator block, decoded sensor measurements, including the ones received from external positioning sources like the GPS and optical motion capture systems, are fused by an estimation algorithm. The PX4 main estimator consists in a twenty-four-state extended Kalman filter that provides estimates of the three components of the position, velocity, and magnetic field, the four components of the attitude quaternion, the horizontal component of the wind and all the relevant sensor bias. This estimator fuses data in a delayed time horizon (considering the different time delays of each measurement relative to the IMU) and uses a complementary filter to propagate states forward to the current time. This feature enables the combination of low latency measurements of the IMU with the ones received from the indoor optical motion capture system, which are delayed due to the offboard communication overhead.

The navigator block is responsible for the autonomous flight modes. It computes setpoints for the position and attitude of the vehicle based on the mission-plan and on the autonomous procedures such as takeoffs and landings, requested by the user. The control block receives these setpoints as inputs, along with the output state estimates from the extended Kalman filter. The position controller, based on the error between the current position yielded by the estimator and the reference position generated by the navigator, computes attitude and normalized thrust setpoints that will drive the vehicle towards the desired position. Similarly, the attitude controller derives the normalized torque required to rotate the vehicle to the desired attitude. Finally, the mixer translates the normalized thrust and torque into individual actuator commands for each motor and servo, ensuring that some safety limits are not exceeded. The input setpoints for the control blocks can also be provided by an RC transmitter. The PX4 has different position and attitude controllers, as well as different mixing algorithms, depending on whether the vehicle is a multicopter, fixed wing aircraft, or VTOL drone.

The flight stack architecture is flexible and structured in a way that lets researchers use only some of its modules. For example, if they are testing an offboard controller, they can send the desired normalized thrust and torque directly to the mixer, without using the blocks that precede it.

2.2 System Architecture

The architecture of the developed rapid prototyping environment results from combining:

1. The fundamental design and operational mechanism of the indoor testbeds documented in the literature. Extensively described in Section 1.3, these testbeds have three mandatory components: i) aerial vehicles; ii) an optical motion capture system; and iii) ground computers. The pose of the vehicles is yielded by the motion capture system to the ground computers that run estimation and control algorithms. The computed actuator commands are sent to the vehicles via wireless communications. This type of architecture has proved to be efficient by enabling the testing of innumerous scientific works. Therefore, it is used as the basis of the system.

2. The PX4 Autopilot, that adds all the features introduced in Section 2.1. These features include abstraction from the vehicles, onboard estimators, onboard controllers, procedures for autonomous maneuvers, flight modes, processes for decoding data from the sensors, algorithms for encoding data to the actuators, and an external connectivity module that eases all the communication between the vehicle and the ground computers.

3. A dual operation mode of simulation and actual physical flights. This means that user programs running in a ground computer can simultaneous communicate with a PX4 mounted on a physical vehicle and communicate with a PX4 SITL instance “mounted” on a vehicle running on a realistic simulator.

The resulting architecture is represented in Figures 2.3 and 2.4. In Fig. 2.3, a detailed view of the physical/real tests environment is displayed, whereas in Fig. 2.4 a detailed view of the simulation environment is depicted.

2.2.1 Physical tests environment

In the physical/real tests environment, the vehicles are confined to the limits of the ISR Flying Arena, an indoor facility of dimensions 7m $\times$ 4m $\times$ 3m. The arena is equipped with an Optitrack motion capture system composed by eight cameras that provide high-frequency measurements of the position and attitude of the vehicles. These measurements are transmitted to the ground computers via Ethernet. The blue dashed arrow in Fig. 2.3 represents the special cases when there is an onboard companion computer capable of receiving and decoding the Optitrack pose and sending it to the PX4 through a serial port.

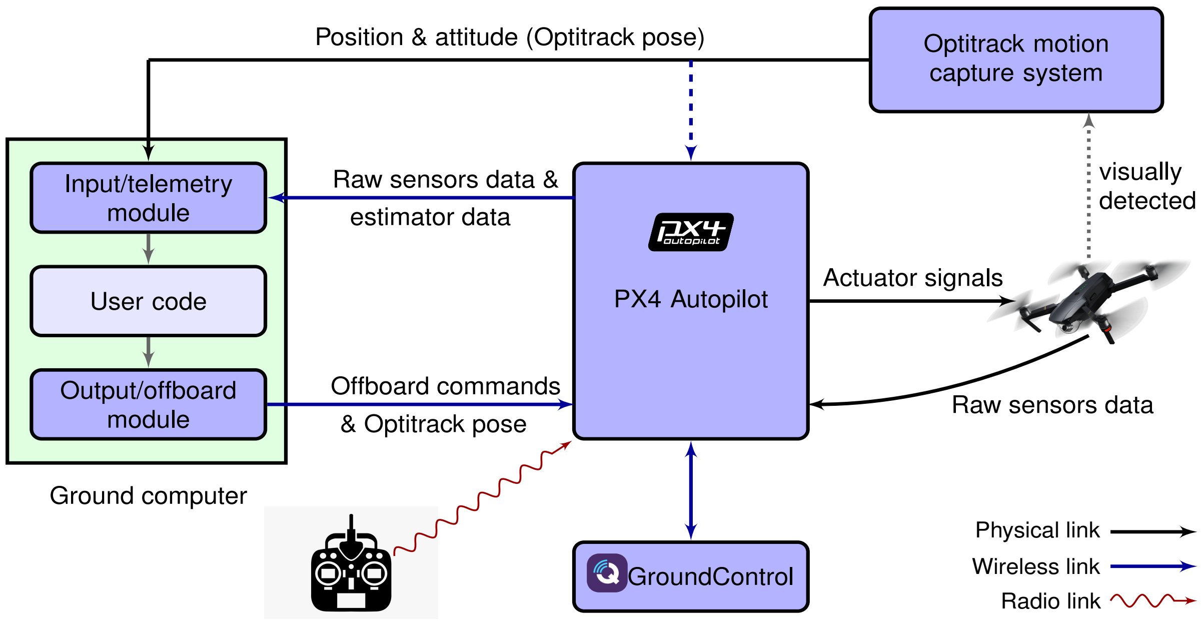


Figure 2.3: Developed architecture for the real environment of the ISR Flying Arena.

The ground computers run offboard guidance, navigation, and control algorithms. These algorithms are implemented in the user code block. To enable the quick and effortless use of the arena, there are two modules assisting this block:

- An input or telemetry module that receives and makes available to the user the position and attitude measurements from the motion capture system, along with the data provided by the PX4 Autopilot. The PX4 provides the raw measurements of the sensors installed on-board the vehicle and the output of its extended Kalman filter. With all this information, and for navigation purposes, the user can decide between implementing their own estimation algorithm or directly use the received estimates. The input or telemetry module runs in a background thread and starts to execute as soon as the ground computer establishes connection with the PX4. It completely automates data reception and data processing.

- An output or offboard module comprising a set of methods that send instructions to the PX4, such as arming commands, takeoff requests, and attitude and thrust references. This module abstracts data sending. The user just needs to call the appropriate methods and the module will send the data to the PX4, according to the MAVLink communication protocol. Note that this module also sends to the PX4, in a background thread, the pose of the vehicle provided by the Optitrack motion capture system and processed by the input module. The PX4 uses this information as an input for its onboard estimator.

By fully automating communication between the ground computers and the remaining systems of the arena, these two modules enable the use of the testing environment by any researcher, regardless of their background in computer science, increasing the focus on the design of offboard algorithms rather than on their implementation details.

The PX4 autopilot can also receive commands from the QGroundControl, an open-source ground control station. The QGroundControl communicates with the PX4 through the MAVLink protocol and can also be used to monitor and modify vehicle parameters. The RC Controller functions as a safety link, enabling the user to land, disarm, and shutdown the vehicle at any moment.

Finally, the PX4 interacts with the vehicle by receiving and decoding the raw measurements provided by the sensors, and by sending individual actuator signals to each motor and servo of the vehicle. The real prototyping environment will be further described and analyzed in Chapter 3.

2.2.2 Simulation environment

The architecture of the simulation environment is presented in Fig. 2.4. The global system is developed with a modular design, where each component has rigorous and well-defined functions and interfaces. Consequently, the simulation environment has the same architecture and interconnections as the real environment. The only difference is that the hardware parts - the PX4 autopilot, the vehicles, and the Optitrack motion capture system - are replaced by equivalent software blocks. These blocks that change when experiments are conducted in the simulation environment, instead of the physical tests platform, are highlighted in red in Fig. 2.4.

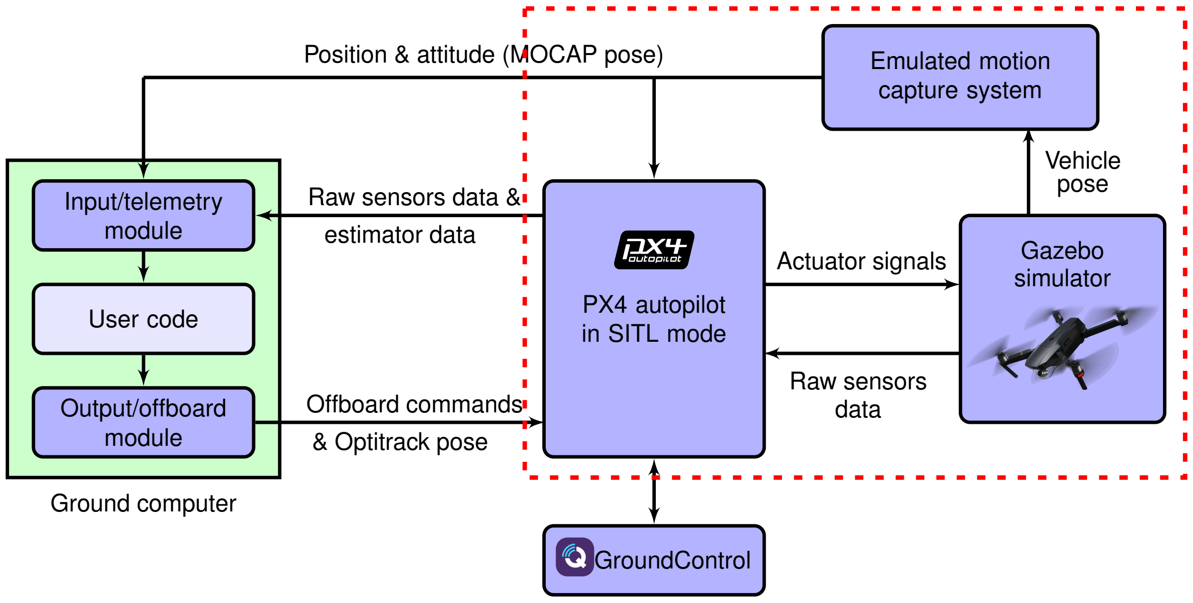

Position & attitude (MOCAP pose)
Emulated motion capture system
Vehicle pose
Input/telemetry module
Raw sensors data & estimator data
User code
Output/offboard module
Offboard commands & Optitrack pose
Ground computer
PX4 autopilot in SITL mode
Actuator signals
Raw sensors data
Gazebo simulator
GroundControl

Figure 2.4: Developed architecture for the simulation environment.

In simulation mode, the behavior of the vehicles is computed by a simulator software, based upon their physical models. Through the actuator signals received from the PX4 SITL instance, the physics engine of the simulator computes the motion of the vehicles and the new sensor measurements. These raw measurements are transmitted back to the PX4. The position and attitude of the vehicles are retrieved from the simulator by a software script that, by adding white Gaussian noise to the retrieved quantities, emulates the measurements generated by a real motion capture system. These noisy measurements are sent to the ground computers and to the PX4. The PX4 autopilot runs in simulation due to its SITL capabilities. The ground computer runs exactly the same modules as in the real tests environment and, due to the modular design adopted, does not need to know whether it is communicating with real or simulated hardware.

The simulated environment will be analyzed in more detail in Chapter 4, which also goes through the choice of the simulator.

2.2.3 Dual mode environment

The designed prototyping environment enables a dual operation mode of simultaneously physical and simulated vehicles. This allows to overcome space and equipment limitations. For instance, for testing a formation control algorithm, part of the drones of the formation can be real and the other part can be simulated. This is possible because the adopted simulator can achieve close to real time performance and the user programs can communicate simultaneously with vehicles of different environments. The implementation of user programs and the scalability of the system will be addressed in detail in Chapter 5.

Chapter 3

Flying Arena

This chapter describes in detail the implementation process of the devised experimental environment. Section 3.1 recalls the overall design adopted for the system. Section 3.2 explains how the information provided by the Optitrack cameras is processed and integrated into the testing setup. Section 3.3 presents some UAVs incorporated into the testing environment and describes how the PX4 was configured and merged into the framework. This section also details the capabilities of the QGroundControl software and how the PX4 and the radio controllers provide solutions to protect the users and the drones in case of failures. Section 3.4 introduces the modules developed to enable effortless offboard control of the vehicles. Finally, Section 3.5 presents a complete overview of the Flying Arena, through detailed diagrams, summarizing all the information presented over the chapter.

3.1 Architecture of the Flying Arena

The overall architecture of the Flying Arena was presented and discussed in Chapter 2. A slightly different representation of that architecture is depicted in Fig. 3.1.

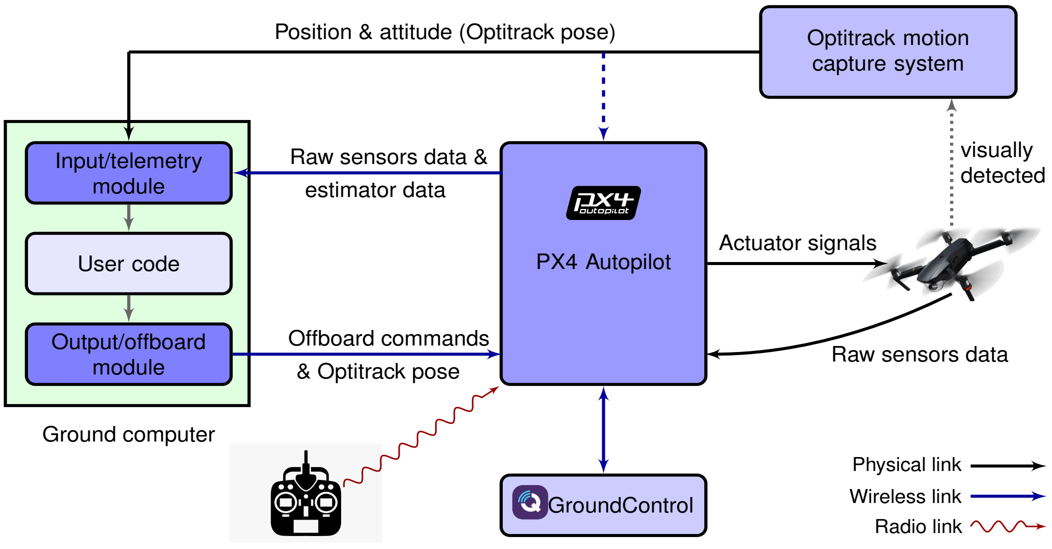


Figure 3.1: Overall architecture of the ISR Flying Arena.

The different shades of blue represent the amount of work that was required to incorporate each block into the setup. The more intense the blue, the longer the time spent setting and coding the block and its interconnections. The figure shows that, although the Optitrack motion capture system and the PX4 autopilot are commercially available products, their inclusion in the system was not immediate. These products required significant and careful configuration along with the development of solutions to decode, convert, and process the information they provide. The input and output modules were the most difficult to implement because they handle and automate communication with the Optitrack and the PX4, according to specific messaging protocols. The detailed implementation of all the blocks represented in Fig. 3.1 is presented over the following sections.

3.2 Optitrack motion capture system

The ISR Flying Arena consists in an indoor test space of dimensions 7m $\times$ 4m $\times$ 2.5m and a set of eight Optitrack motion capture cameras, positioned according to Fig. 3.3. The rapid development and prototyping environment was designed according to the NED (North-East-Down) coordinate system, which is standard in aeronautical applications. The adopted inertial NED coordinate frame, represented in Fig. 3.3, is centered in the arena, at ground level, and has the North axis along the widest side of the arena, the East axis along the shortest side, and the Down axis pointing towards the ground.

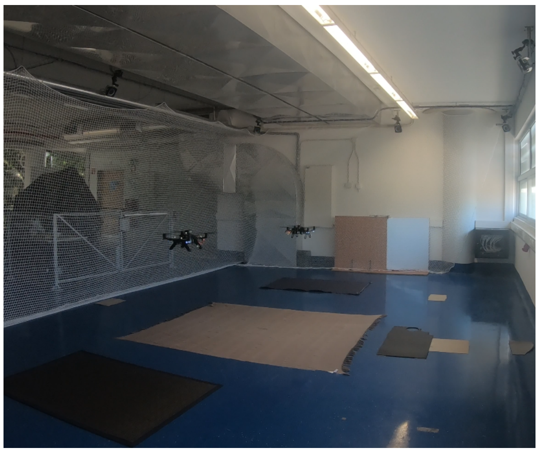


Figure 3.2: ISR Flying Arena.

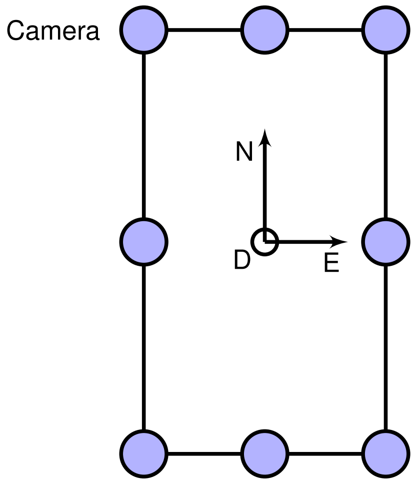


Figure 3.3: Cameras (represented as blue circles) and inertial coordinate frame of the arena.

The Optitrack motion capture system provides high-frequency and low-latency measurements of the position and attitude of the UAVs. It works according to the diagram of Fig. 3.4. First, the Optitrack cameras track special passive markers placed on the body of the vehicles. Then, this tracking data is sent, via Ethernet, to a computer running the Optitrack's Motive software. By feeding the tracking data to its advanced solvers and to its high-level filters, the Motive computes the position and attitude of the vehicles with a positional error less than 0.3mm and a rotational error less than 0.05°. Finally, the Motive sends the position and attitude data to a router that broadcasts it to the local network. Note that the Optitrack system computes the pose data according to a ENU (East-North-Up) inertial frame.

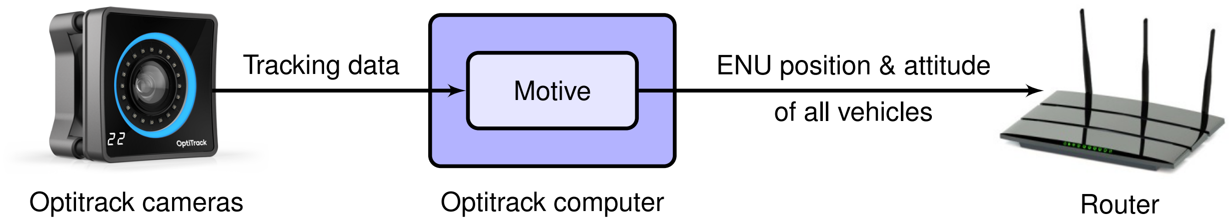


Figure 3.4: Diagram of the information flow of the Optitrack system.

The Optitrack cameras track the markers by detecting reflected infrared light. Consequently, the user has to mask, before each set of experiments and using the Motive software, all the bright spots in the arena that can be mistaken with passive markers. Additionally, the user has to calibrate the Optitrack motion capture system on a weekly basis, because the calibration accuracy naturally deteriorates over time due to ambient factors, such as fluctuation in temperature. These processes are documented in the digital repository that complements this thesis.

After being broadcasted to the local network, the pose data provided by the Optitrack system needs to be decoded and processed, so user programs and the extended Kalman filter of the PX4 can fuse it with the measurements provided by the inertial sensors to produce estimates for the position and attitude of the UAV. The decoding and processing stages developed are represented in Fig. 3.5 and Fig. 3.6. These solutions are implemented using the Robot Operating System (ROS) middleware and the MAVLink-Router [16], a library that transforms ROS topics into MAVLink streams and routes them to other endpoints, such as the PX4.

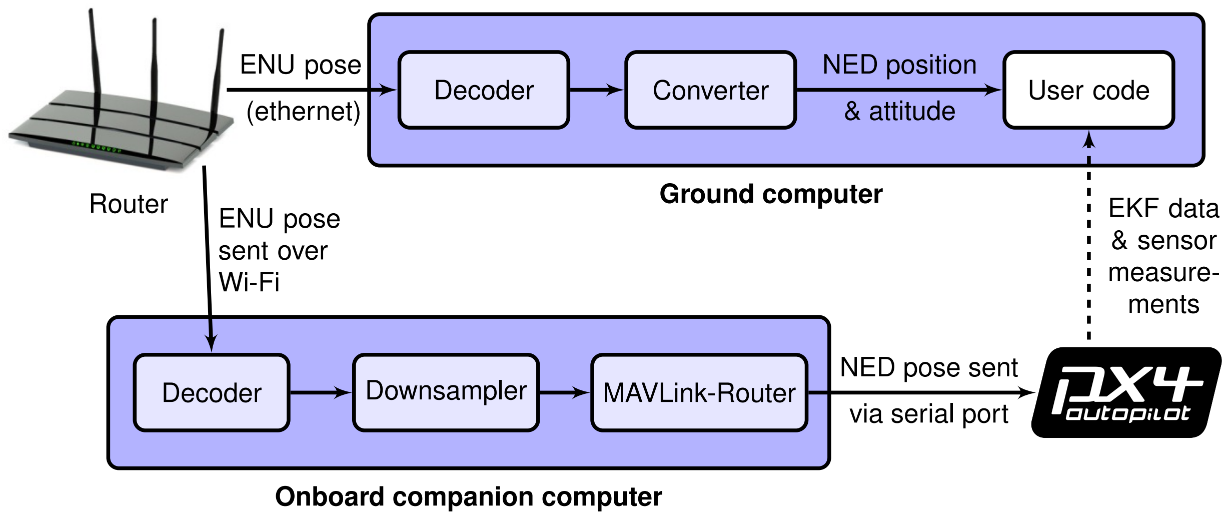


Figure 3.5: Diagram with the processing steps of the Optitrack data in the presence of an onboard companion computer.

The diagram of Fig. 3.5 shows the set of modules developed to deliver the Optitrack data to the user and the PX4, in the cases in which there is an onboard companion computer connected, through a serial port, to the autopilot. On the top part, the position and attitude of the vehicles, in ENU coordinates, is received by the ground computer through an Ethernet link. Then, a decoder block reads this information and publishes it into a ROS topic. The decoder block was built using the VRPN (Virtual Reality Peripheral Network) client, a ROS node that connects to the VRPN server used by the Optitrack system (to stream data to the local network) and exposes the information over a ROS topic. After being decoded, the position and attitude data is converted to the NED coordinate frame, the one adopted for the testing setup, and is finally made available to the user. On the bottom part of Fig. 3.5, the companion computer receives the position and attitude data generated by the Optitrack system via Wi-Fi. The decoder block is equivalent to the one implemented in the ground computer. It was also programmed using the VPRN client that publishes the received pose information into a ROS topic. After the decoding block, the position and attitude data is submitted to a downsampling process. The Optitrack system provides positioning data to the local network at a frequency of 180 Hz. In order to avoid exhausting the bandwidth of the communication channel, which could cause delays in the communication with the PX4 and loss of packets, the downsampling block republishes the pose data into a new ROS topic, dropping two of every three messages received. Therefore, the new ROS topic receives new position and attitude updates at a frequency of 60 Hz. Finally, by using the MAVLink-Router library, the new ROS topic is transformed into a MAVLink stream and is sent, through a serial port, to the PX4 autopilot. During this step, the position and attitude are converted from ENU coordinates, used by the Optitrack system and the ROS middleware, to NED coordinates, used by the MAVLink protocol and the PX4 autopilot. Note that the sensor measurements and the output of the extended Kalman filter of the PX4 are sent, via Wi-Fi, to the ground computer. This gives the users freedom to implement its own estimation algorithms, by fusing the position and attitude data retrieved from the Optitrack system with the sensor measurements received from the PX4, or to simply use the output state estimates of the EKF of the PX4.

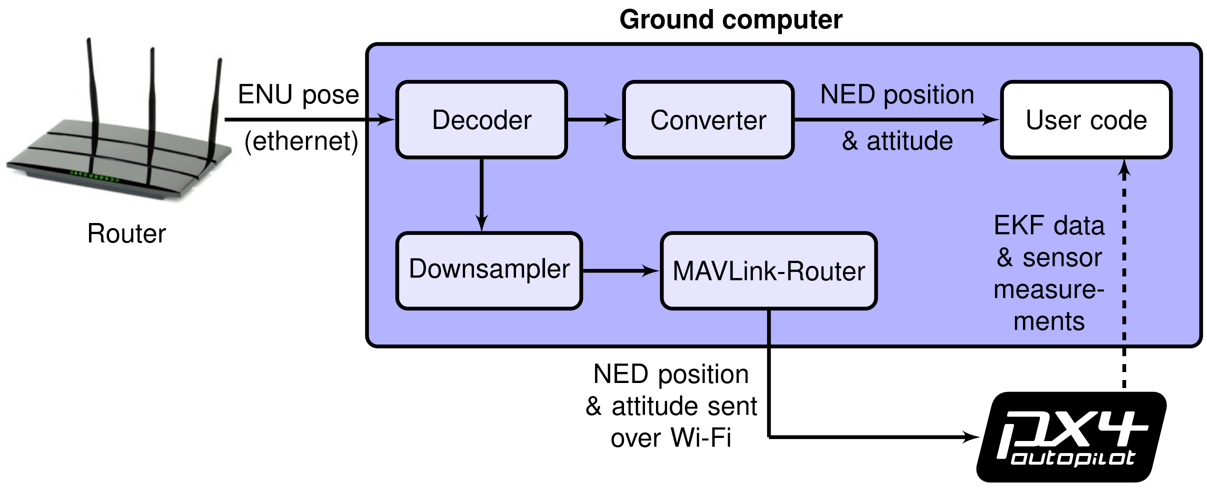


Figure 3.6: Diagram with the processing steps of the Optitrack data in the absence of an onboard companion computer.

The diagram of Fig. 3.6 shows the layout devised to deliver the Optitrack position and attitude of each vehicle to the user programs and to the PX4, in the cases in which there is no onboard companion computer. The blocks adopted are the same ones developed to the setup of Fig. 3.5. The only difference is that all the blocks run on an offboard computer. It should be noted that there is an independent process, for each vehicle, that runs the procedure described in Fig. 3.6. This gives flexibility to the designed setup. For example, if a group of researchers is performing an experiment with multiple UAVs, they can have one ground computer per vehicle, one ground computer for all the vehicles, or a compromise between these two options.

The procedure described over Figures 3.5 and 3.6 is completely automatic. When performing an offboard experiment, the user fills and runs a configuration file, that will be presented and discussed in Chapter 5, that launches in the background the processes that decode and manage the Optitrack data. This enables the users to focus on high-level algorithms. The Optitrack position and attitude of the vehicles, the sensors measurements, and the output of the extended Kalman filter of the PX4 are automatically made available to them in the user code block.

3.3 Quadrotors, PX4, and QGroundControl

In order to perform experiments, it is necessary to integrate vehicles into the testing environment. This is primarily achieved by installing a PX4 autopilot in the vehicles and by configuring it appropriately. As stated in Chapter 2, the PX4 acts as an interface between the offboard modules and the flying capabilities of the vehicle. It decodes data from the sensors, encodes data to the actuators, provides autonomous controllers and estimators, supports safety features, and enables communication with external systems through the MAVLink protocol. In this thesis, two different quadcopters were integrated into the testing setup. The first one was the Intel Aero Ready to Fly Drone [17] exhibited in Fig. 3.7. The second one was Snapdragon, a custom built quadcopter assembled by the researchers of Institute for Systems and Robotics, illustrated in Fig. 3.8. The Intel Aero quadcopter is an example of a vehicle that features an onboard companion computer, whereas the Snapdragon is a lighter drone, that does not have any onboard computer assisting the PX4. Both drones were used in the tests documented in Chapter 6.

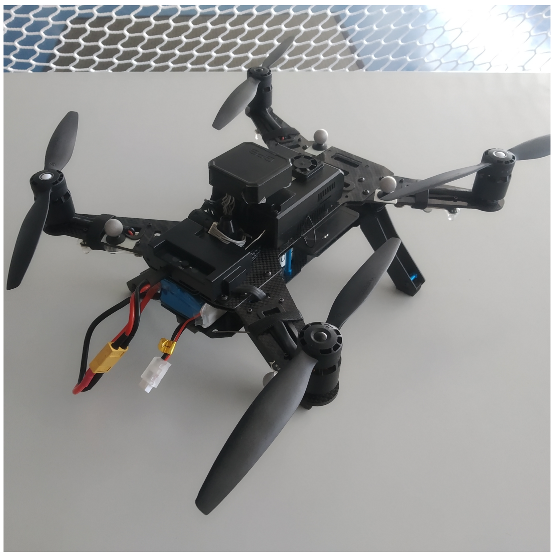


Figure 3.7: Intel Aero Ready to Fly Drone.

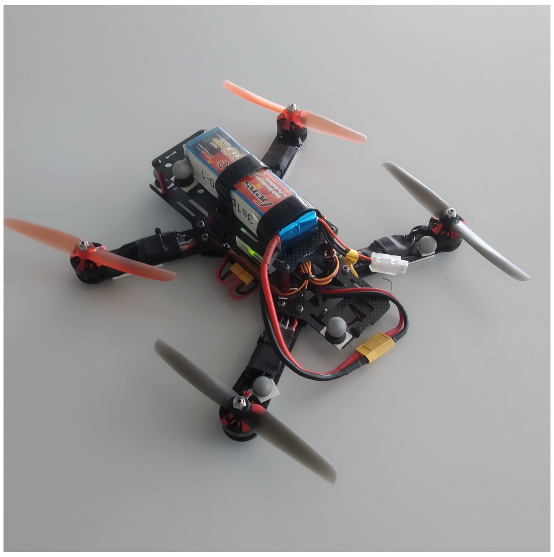


Figure 3.8: Snapdragon quadrotor.

To incorporate these two vehicles into the testing setup, a three-step procedure was followed:

1. In the first step, each PX4 was configured according to the physical properties of the UAV and in conformity with the sensors and avionics installed in the vehicle. This configuration procedure is introduced in Section 3.3.1. It includes the specification of the air-frame of the drone, the calibration of the sensors, the configuration of the failsafe features, and the optimal tuning of the internal controllers of the PX4.
2. In the second step, the extended Kalman filter of the PX4 was tuned according to the properties of the Optitrack system and the communication latency measured. This procedure is presented in Section 3.3.2.
3. Finally, in the last step, a set of modules were developed to enable communication with each PX4 and to automate the reception of odometry updates and the sending of offboard commands to the vehicle. These modules are introduced in Section 3.4 and explored in greater detail in Chapter 5.

3.3.1 Configuring the PX4

The PX4 of the Intel Aero and the Snapdragon quadcopters was configured according to the official guide [18] provided in the PX4 website. The configuration was assisted by the QGroundControl software. Since it was an extensive procedure, only the most relevant steps of the configuration, that are related with important features of the Flying Arena, will be addressed.

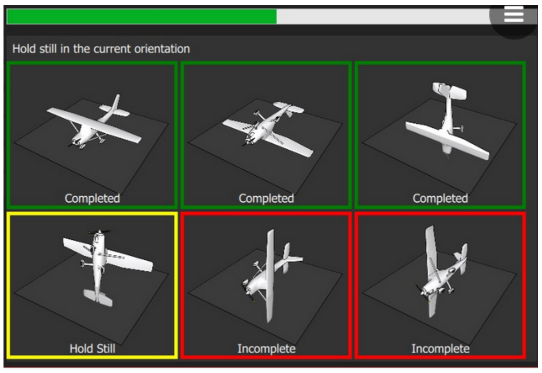
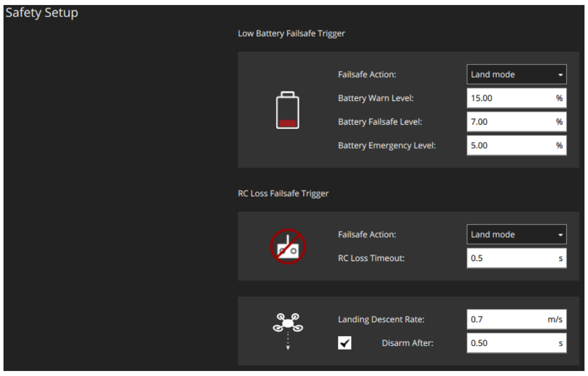


Figure 3.9: Calibration of the accelerometer using the QGroundControl.



Safety Setup

Low Battery Failsafe Trigger

Failsafe Action:	Land mode
Battery Warn Level:	15.00 %
Battery Failsafe Level:	7.00 %
Battery Emergency Level:	5.00 %

RC Loss Failsafe Trigger

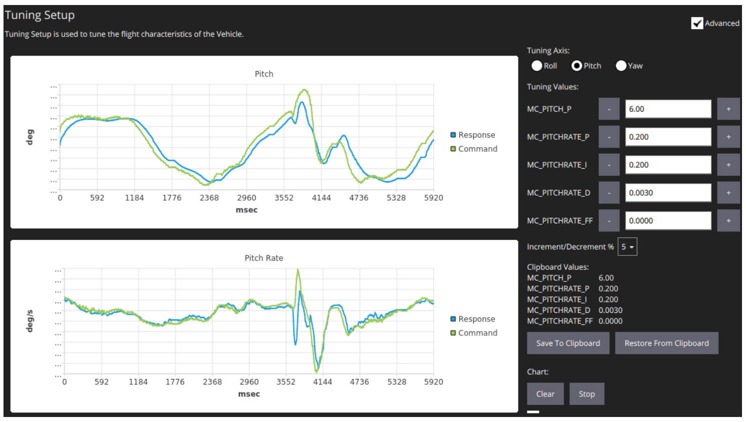
Failsafe Action:	Land mode
RC Loss Timeout:	0.5 s
Landing Descent Rate:	0.7 m/s
✓ Disarm After:	0.50 s

Figure 3.10: Failsafes configuration using the QGroundControl.

After uploading the most recent firmware version of the PX4 to the flight controller board, it was necessary to calibrate the sensors of the vehicle. This was a straightforward process due to a visual calibration setup provided by QGroundControl software. In order to calibrate the inertial sensors, it is only necessary to place and hold the vehicle according to a series of orientations requested by the calibration routine. Fig. 3.9 shows the setup provided by the QGroundControl software to enable the calibration of the accelerometer. The PX4 performs pre-flight sensor quality and estimator checks to verify if there is a good enough position estimate to arm and operate the vehicle. Whenever there is a poor position estimation and a calibration of the sensors is required, the users can resort to this setup to perform it.

Another important aspect that was configured in the PX4 autopilot are the failsafe modes. These safety features protect the user and the equipment when something goes wrong. Whenever a failsafe is triggered, the PX4 performs a pre-selected action, such as transition to hover mode, land the vehicle immediately, or return the vehicle to the home position. Fig. 3.10 presents the safety setup page provided by the QGroundControl to configure the failsafes. The low battery level failsafe was set to warn the user if the battery capacity drops below 15% and to transition to auto-land mode when the battery capacity drops below 7%. The RC loss failsafe is triggered if the communication link with the RC transmitter is lost and was also configured to set the flight mode to auto-land. Whenever the auto-land mode is engaged, the vehicles will vertically descent to the ground at a rate configured to 0.7 m/s. The RC transmitter, shown in Fig. 3.11, is the primarily safety link of the Flying Arena because it provides a last-resort solution to stop the most unpredictable problems through its safety switches configured to immediately put the vehicles in hover mode or shutdown the motors. Some failsafe settings cannot be configured through the QGroundControl safety setup page. These must be configured by editing the internal parameters of the PX4. Examples of this are the position loss failsafe and the offboard loss failsafe. The former is triggered if the quality of the position estimate drops below acceptable levels, whereas the latter is triggered if the offboard communication link is lost during an offboard experiment. These failsafes were configured by editing, respectively, the COM_POSCTL_NAVL and the COM_OBL_RC_ACT parameters of the PX4. If a position loss failsafe is triggered, the PX4 is set to shutdown the motors of the vehicle because a significant drop in the quality of the estimates usually means that the PX4 stop receiving data from the Optitrack and, therefore, the position and attitude estimates are no longer accurate enough to continue the experiment or even land the drone. If an offboard loss failsafe is triggered, the PX4 was configured to activate the auto-land mode. This failsafe usually occurs when the user program that sends offboard commands to the vehicle suddenly stops working. Since there is an independent process sending the Optitrack data to the vehicle, the estimator of the PX4 still produces valid estimates to safely perform an auto-landing maneuver.

Figure 3.11: Radio Controller used in the experiments.



Tuning Setup

Tuning Setup is used to tune the flight characteristics of the Vehicle.

✓ Advanced

Tuning Axis:
○ Roll [● Pitch] ○ Yaw

Tuning Values:

MC_PITCH_P	6.00
MC_PITCHRATE_P	0.200
MC_PITCHRATE_I	0.200
MC_PITCHRATE_D	0.0030
MC_PITCHRATE_FF	0.0000

Increment/Decrement % 5

Clipboard Values:

```
MC_PITCH_P       6.00
MC_PITCHRATE_P   0.200
MC_PITCHRATE_I   0.200
MC_PITCHRATE_D   0.0030
MC_PITCHRATE_FF  0.0000
```

Figure 3.12: PID tuning setup available in the QGroundControl software.

Finally, the internal PID controllers of the PX4 were slightly tuned. This procedure was performed using the tuning setup of the QGroundControl, exhibited in Fig. 3.12. Since the Intel Aero is a commercial product and the Snapdragon was pre-assembled by the Institute for Systems and Robotics, these vehicles were already tuned and flight tested. However, for safety reasons, the controller gains were set to conservative values. Therefore, in this final step, the gains were better adjusted to each quadcopter.

3.3.2 Tuning the extended Kalman filter

To finish the configuration of the vehicles, the extended Kalman filter of the PX4 was tuned according to the properties of the testing setup and the Optitrack motion capture system. First, the sources of position, velocity, and attitude measurements used by the estimator were defined by configuring the bits of the EKF2_AID_MASK parameter. Since the Flying Arena is an indoor environment, a global positioning system such as the GPS is unavailable, so the source of horizontal and vertical position data was set to be the Optitrack. The Optitrack system has the added advantage of providing position information with significantly more precision and higher rates than the GPS. However, the Optitrack system does not provide velocity measurements of the vehicles. Consequently, the estimator was set to not use any source of external velocity data. Finally, two different sources of attitude measurements were configured to be used by the estimator. For roll and pitch estimation, the extended Kalman filter is automatically programmed to depend only on the IMU sensors available onboard, discarding the roll and pitch data retrieved from the Optitrack. This is due to the fact that the inertial sensors measurements are sufficient to produce satisfying low-latency and low-drift roll and pitch estimates. For yaw estimation, the EKF2_AID_MASK parameter was configured so that the estimator uses the yaw measurements provided by the Optitrack and discards the readings of the magnetometers because these are disturbed by the electric motors and affected by magnetic anomalies of the indoor environment. In summary, the extended Kalman filter of the PX4 was configured to use the position and yaw measurements given by the Optitrack motion capture system and is automatically programmed rely on the accelerometers and gyroscopes to produce low-latency and low-drift estimates of the roll and pitch angles.

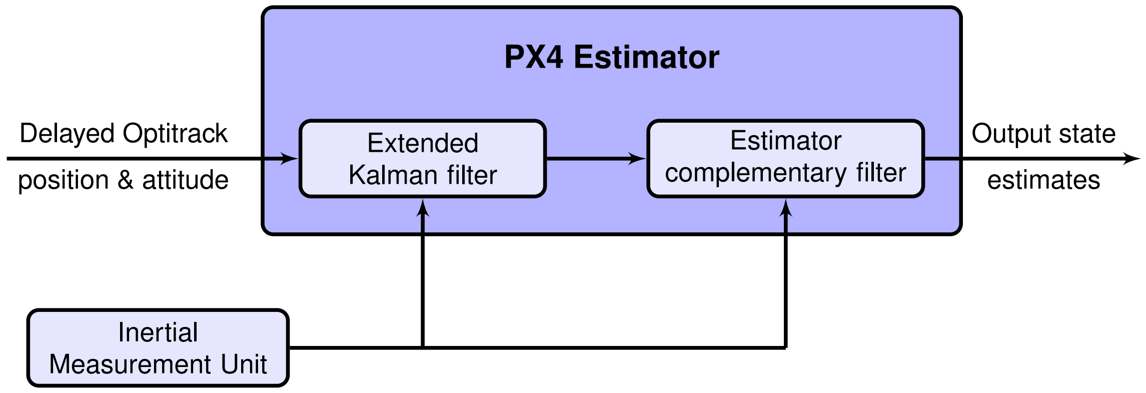

PX4 Estimator
Delayed Optitrack position & attitude
Extended Kalman filter
Estimator complementary filter
Output state estimates
Inertial Measurement Unit

Figure 3.13: Data fusion process of the estimator of the PX4.

To enable the fusion of low-latency measurements of the accelerometers and gyroscopes with the position and yaw measurements received from the Optitrack system, that reach the PX4 with some delay due to communication overhead, the estimator fuses the data in a delayed time horizon. This means that the estimates are computed considering the time delay of the Optitrack data relative to the IMU. Then, an output complementary filter propagates the estimates forward to current time. This procedure is represented in Fig. 3.13. Consequently, it was necessary to tune the extended Kalman filter with the correct time delay between the arrival of the Optitrack and the IMU measurements. A rough estimate of the delay was obtained from the logs by checking the time offset between the IMU and the Optitrack data. Then, this value was further refined by performing a set of experiments with distinct delay values, and by checking the resulting estimator innovations. The time delay obtained, 20ms, corresponds to the one that yielded the smallest innovations. This value was assigned to the EKF2_EV_DELAY internal parameter of the PX4.

3.4 Ground Computers

Once all the steps described in the last sections have been completed, that is, once: i) the PX4 autopilot is adapted to the physical properties of the vehicle; ii) the sensors of the drone are calibrated; iii) the failsafe modes and the radio controller switches are defined; iv) the internal controllers of the PX4 are tuned; v) the Optitrack position and attitude data is automatically being decoded and sent to the PX4 autopilot; and vi) the extended Kalman filter is correctly configured to use the position and yaw data from the Optitrack system, the UAVs are ready to fly. In the devised environment, the vehicles receive offboard control commands from user programs. These programs follow an object-oriented approach and rely on the input and output modules to communicate with the PX4, as shown in Fig.3.14. Each vehicle is represented by an instance of a class, whose attributes store the current state of the vehicle (such as the position, attitude and velocity) and the most recent readings of the sensors. The input or telemetry module consists on a set of methods that, running in background threads, keep the attributes of the class up-to-date. The output or offboard module comprises the methods that send offboard commands and control references to the PX4. Since these modules are extensive and common to both the real and the simulation environment, they will only be explored in detail in Chapter 5.

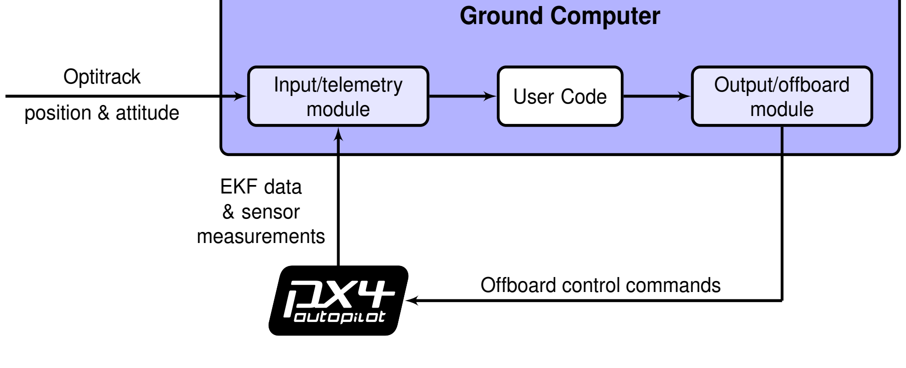


Figure 3.14: Offboard control process of the vehicles.

3.5 Detailed overview

The diagram of Fig. 3.15 aims to summarize the information presented over this chapter by detailing the flow of data between the different modules of the ISR Flying Arena, for vehicles that count with an onboard companion computer.

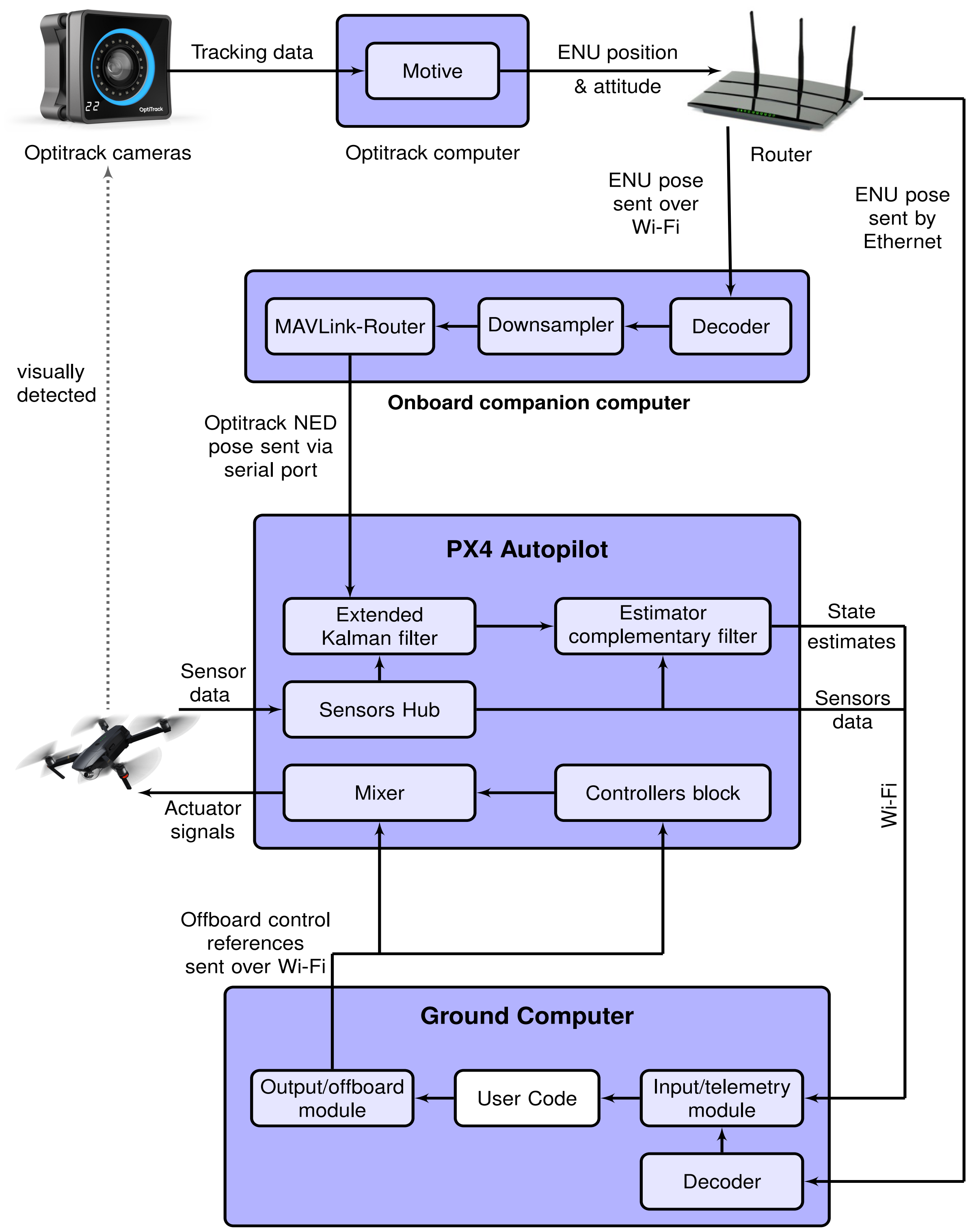


Figure 3.15: Flow of information between the modules of the ISR Flying Arena, for vehicles with an onboard companion computer.

In the absence of an onboard companion computer, the decoder, downsampler, and MAVLink-Router modules have to run in a ground computer, as depicted in Fig. 3.16

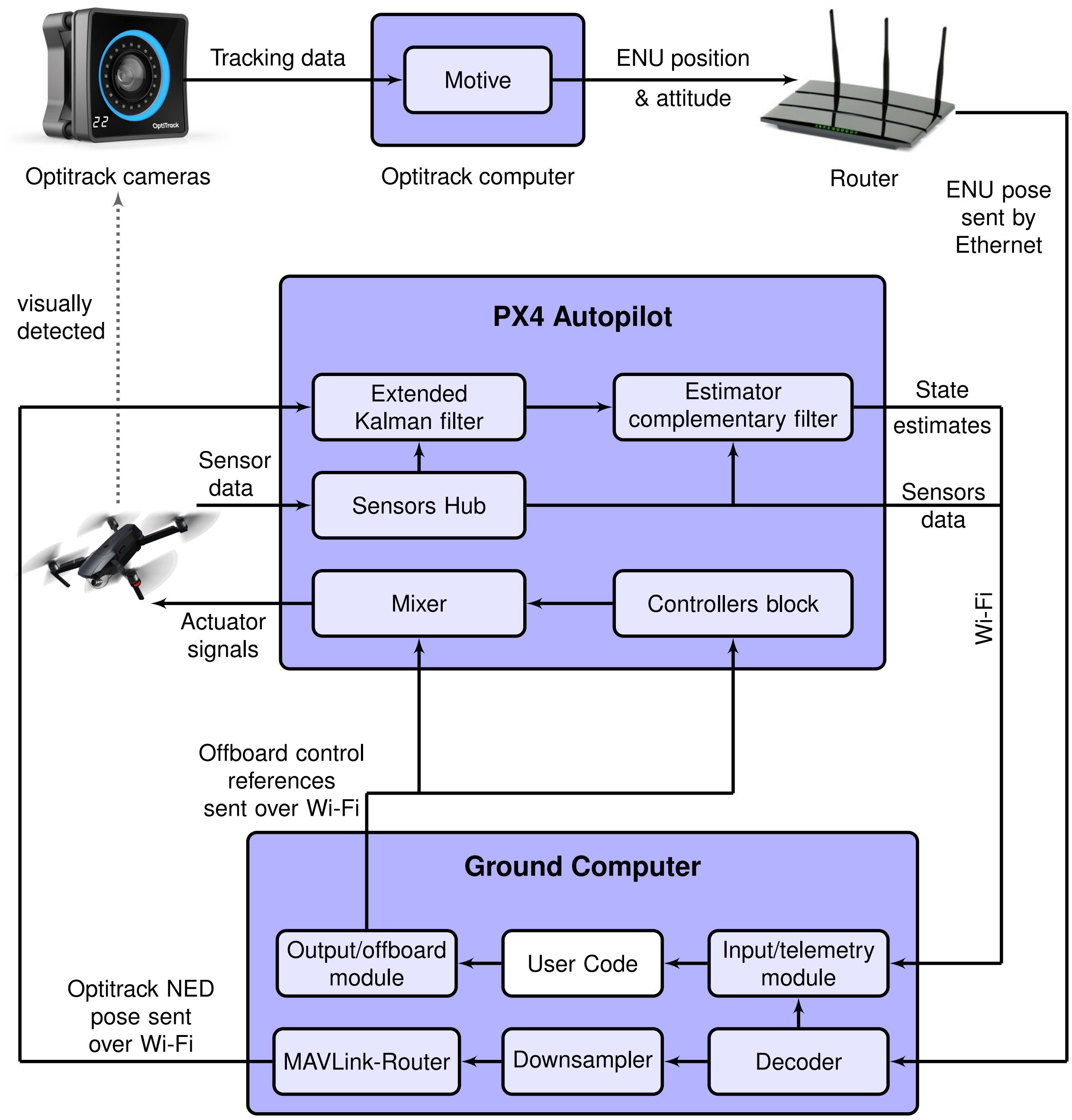


Figure 3.16: Flow of information between the modules of the ISR Flying Arena, for vehicles without an onboard companion computer.

This chapter focused on the configuration of existent commercial hardware components and their integration in the envisioned setup. For this purpose, modules were programmed to decode and process data from the Optitrack system and deliver it to the PX4 of each vehicle. The next two chapters are more focused on the software parts of the Flying Arena. Chapter 4 explores the solutions coded for simulating the testing environment. Chapter 5 presents the solutions coded, in multiple programming languages and using multiple messaging libraries, for automating the communication with the PX4 and the testing of navigation and control solutions.

Chapter 4

Simulation

This chapter describes, in detail, the devised simulation environment. Section 4.1 recalls the overall design adopted for the simulation framework. Section 4.2 introduces Gazebo, the robotics simulator selected for the system. Section 4.3 presents the user interface developed to spawn and simulate vehicles in Gazebo and to run PX4 SITL instances. Section 4.4 describes the software emulation of the physical motion capture system. Section 4.5 discusses solutions to reduce simulation overhead. Finally, Section 4.6 provides a complete overview of the simulation setup, summarizing all the information presented over the chapter.

4.1 Architecture of the simulation environment

The overall architecture of the simulation environment of the Flying Arena was presented and discussed in Chapter 2. A slightly different representation of that architecture is depicted in Fig. 4.1.

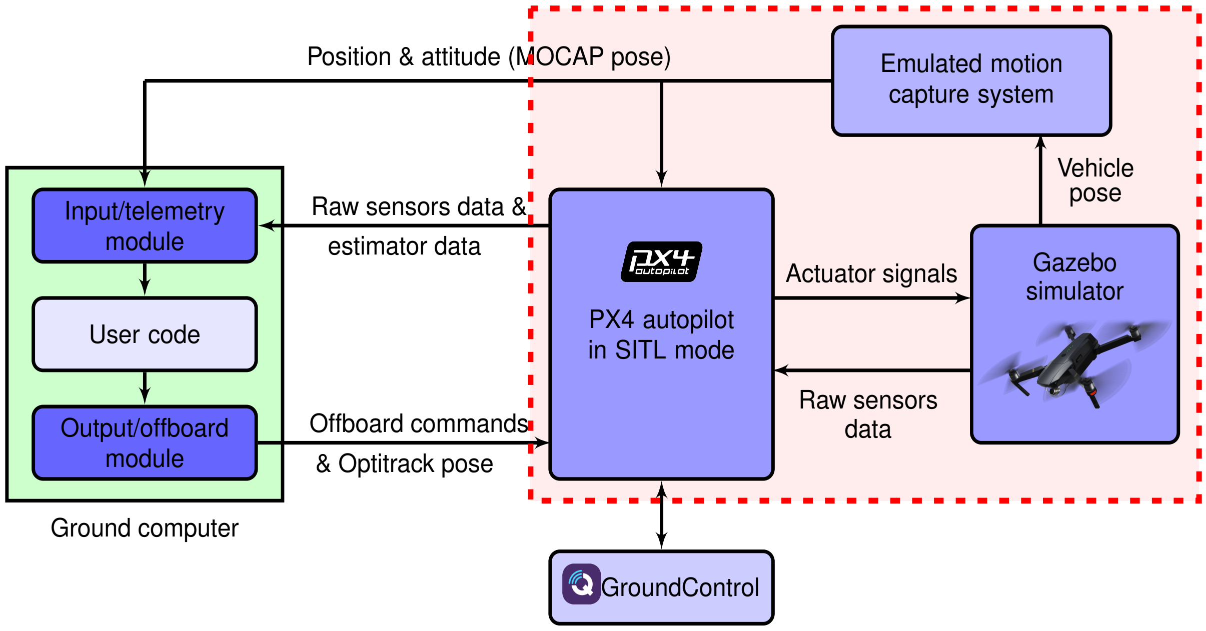

Position & attitude (MOCAP pose)
Emulated motion capture system
Vehicle pose
Input/telemetry module
Raw sensors data & estimator data
User code
Output/offboard module
Offboard commands & Optitrack pose
PX4 autopilot
PX4 autopilot in SITL mode
Actuator signals
Gazebo simulator
Raw sensors data
Ground computer
GroundControl

Figure 4.1: Architecture of the simulation environment.

The blocks inside the red area correspond to the ones that change when experiments are conducted in the simulation environment, instead of the real environment (the actual Flying Arena, with real vehicles and real hardware):

- In the Gazebo simulator block, vehicles are represented as physical entities with configurable dynamic and kinematic properties. The behavior of sensors and actuators is modeled by plugins. Based on the actuator signals received and the sensor noise characteristics, the simulator computes the motion of the vehicles and the new sensor readings.
- In the emulated motion capture system block, a software module retrieves the pose of the vehicles from the Gazebo simulator and delivers it to the user and the PX4. In addition, this module also adds adjustable Gaussian noise to the retrieved quantities, to model the uncertainty of the measurements provided by a real motion capture system.
- In the autopilot block, the PX4 communicates with the simulator to receive sensor data and send actuator signals. The PX4 behaves and has the same capabilities in simulation as when connected to actual vehicles.

The detailed implementation and integration of these three blocks is presented over the following sections. The remaining blocks, common to both the real and the simulation environments, will be discussed in Chapter 5.

4.2 Gazebo simulator

Simulators allow testing of navigation and control solutions in a quick and safe way. Users can interact with a simulated vehicle just as they might with a real one, by using the QGroundControl software or by running offboard programs on the ground computers to send commands and control references to the PX4 autopilot. Before attempting to fly actual vehicles in the ISR Flying Arena, it is recommended to use the simulator to ensure that the estimation and control algorithms work properly and the vehicles behave as expected. This is a key measure to guarantee the safety of users and equipment.

The selected simulator for the devised setup is Gazebo [19], a powerful 3D robotics engine suitable for testing autonomous vehicles. Gazebo was the chosen simulator because: i) it is compatible with the software-in-the-loop capabilities of the PX4 autopilot; ii) it supports all kinds of aerial vehicles, such as multicopters, fixed wing aircrafts, and VTOL drones; iii) it accepts custom models of those vehicles, so that the user can simulate UAVs with dynamic and kinematic properties close to those of the real drones used in the Flying Arena; iv) it offers plugins to simulate the behavior of sensors and actuators; v) it supports multi-vehicle simulation; vi) it is compatible with the ROS middleware, a set of robotics libraries and tools used in this thesis; and finally vii) it is suitable to test object-avoidance and computer vision algorithms, two important research topics at the Institute for Systems and Robotics. The major drawback of the Gazebo simulator is that it is computationally heavy. In multi-vehicle experiments, depending on the number of UAVs simulated, it is necessary to run the simulation in a dedicated computer or even across multiple dedicated computers, for close to real time performance.

Fig. 4.2 shows the message flow between Gazebo and a PX4 autopilot. Gazebo communicates with the PX4 using the MAVLink API [20]. This API defines a set of MAVLink messages that provide raw sensor data from the simulated world to PX4 and receive the actuator signals that will be applied to the motors and controlling devices of the simulated vehicle. A software-in-the-loop build of PX4 uses the simulator_mavlink.cpp [21] module to exchange MAVLink messages with Gazebo. It is important to note that the PX4 autopilot and the Gazebo simulator run in lockstep, which means that they wait on each other for sensor and actuator messages, rather than running at their own speeds. Due to the lockstep feature, it is possible to run the simulation faster or slower than real time without losing messages. If a computer is not powerful enough to run a given experiment at the rate requested by the user, the simulation will run at a slower pace. This is only problematic when performing experiments with simultaneously real and simulated UAVs that interact with each other. In such cases, the simulator must run close to real time. To help achieve a real time simulation rate in experiments with numerous vehicles, solutions were developed, as presented in Section 4.5, to reduce simulation overhead. Moreover, the simulation environment was designed to allow simulation of UAVs across multiple Gazebo instances, dividing the required computing power for the experiment over multiple computers.

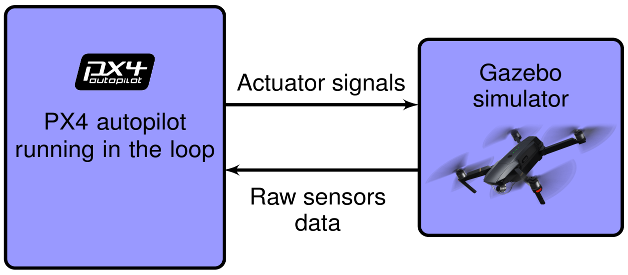


Figure 4.2: Message flow between Gazebo and the PX4.

The installation steps of the Gazebo software are described in detail in the digital repository that complements this thesis. The next section describes how the user can simulate vehicles in Gazebo and run PX4 instances in the loop.

4.3 Simulating vehicles and running the PX4 in the loop

Gazebo can operate as a standalone software or within a ROS environment. Launching Gazebo simulations within ROS is advantageous because this middleware provides a set of services and topics to modify and retrieve information about the state of the simulated world and the simulated vehicles [22]. Since these tools allow software programs to retrieve the position and attitude of the drones from the simulator, which are required for the emulation of the motion capture system, it was decided to launch Gazebo within ROS. The services provided by ROS also allow to use Gazebo as a visualization tool, as presented in Chapter 5.

One of the goals of this thesis is the development of an user interface enabling the quick and effortless use of the simulation framework. Fig. 4.3 presents the user interface developed to launch Gazebo within ROS, spawn vehicles in the simulator, and run PX4 SITL instances. It consists in a simple configuration file, named *gazebo.launch*, that hides the complexity of the simulation launch process.

The *gazebo.launch* file is divided in two different code blocks. The first one, from lines 5 to 8, is responsible for launching Gazebo with ROS. The second one, from lines 15 to 31, is responsible for spawning a vehicle and starting a PX4 SITL instance. For multi-vehicle simulations, multiple blocks of the latter must be created. To launch simulations with this interface, the user just has to fill the arguments of each block with the desired configurations for Gazebo, for the simulated vehicles, and for the PX4 SITL instances. Then, a sequence of software programs operating in the background will automatically launch the simulation, according to the arguments requested. This interface enables the user to benefit from the full capabilities of the simulation environment without prior experience with Gazebo, the ROS tools, or the PX4 firmware. Without the compact interface presented in Fig. 4.3, users would have to extensively edit multiple different files of the PX4 firmware, in order to fully configure the simulation. The next two sections will describe the arguments and the background programs responsible for starting the simulation.

```
 1  <?xml version="1.0"?>
 2  <launch>
 3      <!-- LAUNCH GAZEBO: CHOOSE WHETHER OR NOT TO LAUNCH THE GRAPHICAL INTERFACE.
 4      SELECT THE WORLD. -->
 5      <include file="$(find gazebo_ros)/launch/empty_world.launch">
 6          <arg name="gui" value="true"/>
 7          <arg name="world_name" value="$(find mavlink_sitl_gazebo)/worlds/empty.world"/>
 8      </include>
 9
10      <!-- LAUNCH AN UAV: ASSIGN A UNIQUE NAMESPACE. CHOOSE THE VEHICLE INITIAL
11      POSITION AND INITIAL ATTITUDE. ASSIGN A UNIQUE ID FOR THE DRONE AND FOR THE
12      SYSTEM. ASSIGN A UNIQUE PORT. SELECT THE VEHICLE MODEL. STATE THE FILE WITH
13      THE DESIRED INTERNAL CONFIGURATIONS FOR THE PX4. IF YOU WANT TO CONNECT TO
14      THE DRONE FROM EXTERNAL COMPUTERS, INSERT THE IP ADDRESS OF THOSE COMPUTERS.-->
15      <group ns="uav1">
16          <include file="$(find px4)/launch/drone.launch">
17              <arg name="N" value="0"/>
18              <arg name="E" value="0"/>
19              <arg name="D" value="0"/>
20              <arg name="r" value="0"/>
21              <arg name="p" value="0"/>
22              <arg name="y" value="0"/>
23              <arg name="ID" value="1"/>
24              <arg name="port" value="16001"/>
25              <arg name="vehicle_model" value="iris"/>
26              <arg name="px4_config" value="px4_config"/>
27              <arg name="gcs_1_ip" value="192.168.1.21"/>
28              <arg name="gcs_2_ip" value="192.168.1.22"/>
29          </include>
30      </group>
31  </launch>
```

Figure 4.3: Code of the gazebo.launch file: the configuration file used to launch Gazebo, spawn vehicles in the simulator, and run PX4 SITL instances in the loop.

4.3.1 Launching Gazebo

The developed user interface, that consists in the *gazebo.launch* file presented in Fig. 4.3, launches Gazebo through an auxiliary program called *empty_world.launch* [23], the standard simulation launcher provided by the ROS API for Gazebo. The *empty_world.launch* program accepts multiple arguments that allow the user to start the simulator with different configurations. In order to present an user interface that is as clean and simple as possible, only the most relevant arguments, those that can be advantageous for the user to change depending on the experiment or on the hardware of the computer used for simulation, were selected for the *gazebo.launch* file:

- The **gui** argument determines whether Gazebo will run with or without the graphical user interface. Performing simulations with the graphical user interface enables 3D visualization of the motion and behavior of the vehicles during the experiments. By launching Gazebo without GUI, the user is not able to visualize the drones during the experiments, but the simulation runs faster and uses less system resources. This is a way to run the simulation with less overhead and is the solution to run Gazebo in computers or servers without a dedicated Graphics Processing Unit (GPU). After the simulation, through the generated log files, the user can recreate and visualize the experiment in 3D in more lightweight tools, which will be presented in Chapter 5.
- The **world_name** argument specifies the path to the world file that will be launched by Gazebo. In the world file, it is possible to customize basic features of the simulation, such as physics engine parameters, ground plane textures, and lighting. Since the visualization and physics engine of Gazebo are computationally heavy, the default world adopted for this thesis includes the most simple ground plane and global light source plugins, so as to not generate unnecessary overhead.

In the background, the *empty_world.launch* program runs two different executables. The first is called *gzserver* and the second *gzclient*. The *gzserver* runs the physics engine of Gazebo with the configurations of the world selected by the user, and is responsible for computing the motion of the vehicles and generating sensor data. The *gzclient* runs the graphical interface. The scheme of Fig. 4.4 summarizes the launching process of Gazebo. It is important to note that the user only needs to define the values of the **gui** and **world_name** arguments, the remaining steps of the launching process are automatic.

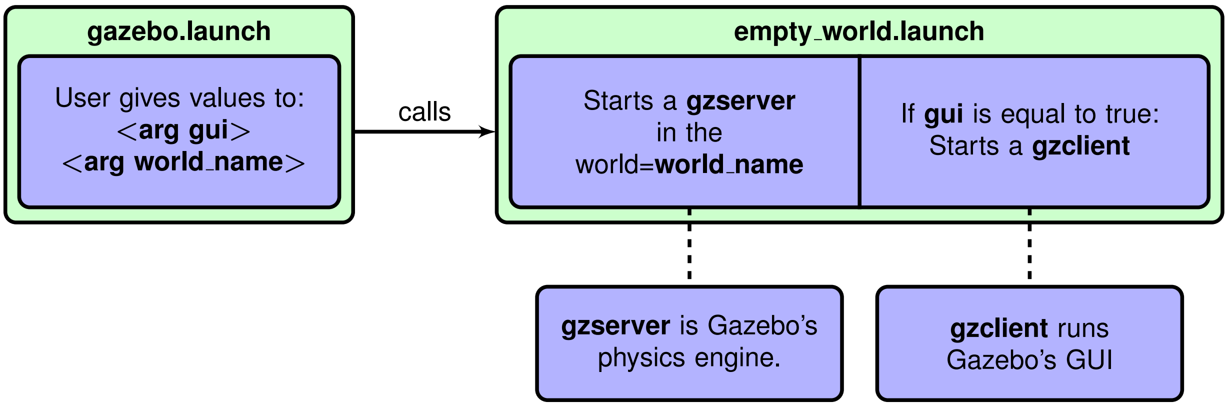

gazebo.launch
User gives values to: <arg gui> <arg world_name>
calls
empty_world.launch
Starts a gzserver in the world=world_name
If gui is equal to true: Starts a gzclient
gzserver is Gazebo's physics engine.
gzclient runs Gazebo's GUI

Figure 4.4: Launching process of the Gazebo simulator.

4.3.2 Launching vehicles and PX4 SITL instances

The second block of the *gazebo.launch* file (Fig. 4.3) is responsible for spawning a vehicle in the Gazebo simulation and starting a PX4 SITL instance for that specific vehicle. To perform these two tasks, a roslaunch XML program named *drone.launch* is called. The *drone.launch* program is a deeply customized version of the recommended roslaunch XML files available in the PX4 firmware [24]. In addition to launching vehicles and PX4 instances, it was also programmed to automatically define the communication settings between the PX4 SITL and the Gazebo simulator and to enable the connection of the simulated drone with multiple offboard computers and offboard QGroundControl stations. The *drone.launch* program uses the following arguments, defined by the user in the *gazebo.launch* file:

- The **N** (North), **E** (East), and **D** (Down) arguments consist in the initial position for the simulated drone, in meters, in the NED coordinate system adopted for the ISR Flying Arena, previously described in Section 3.2. Similarly, the **r** (roll), **p** (pitch), and **y** (yaw) parameters define the initial attitude of the simulated drone, in radians, in the same coordinate frame.
- The **vehicle_model** argument specifies the name of the ROS model used to generate the simulated vehicle. A vehicle model is composed by a XACRO file and a BASE.XACRO file:
  - The XACRO file defines the physical properties of the simulated vehicle, such as its mass, dimensions and moments of inertia. Additionally, this file also contains the parameters that model the behavior of motors and actuators. By editing the XACRO file, the user can generate a simulated drone with dynamic properties similar to those of the real drones used in the Flying Arena. For instance, in the tests described in Chapter 6, the drag coefficient of the simulated drones was reduced from its default value in order to produce identical results to those obtained in the tests performed with the Intel Aero quadcopters, in the actual Flying Arena.
  - The BASE.XACRO file contains the plugins that model the sensors of the simulated vehicle. By editing the BASE.XACRO file, the user can add, modify, and remove plugins. When modifying plugins, the user can tune the uncertainty of the measurements produced by each sensor, by adjusting the bias and the noise density parameters.
- The **ID** argument consists in a positive integer that identifies the PX4 of each drone in a multi-vehicle situation. The **port** argument defines the network port to which user programs have to connect to, for communication with the PX4 of the simulated vehicle. Note that each vehicle must have a unique **ID** and **port**. The **gcs_1_ip** argument defines the local IP address of an external computer, running an user program and/or an instance of the QGroundControl software, that is configured to communicate with the PX4 of the simulated vehicle. The user can add multiple instances of this argument: **gcs_2_ip**, **gcs_3_ip**, **gcs_4_ip**, etc.
- Finally, the **px4_config** argument specifies the file that stores the desired internal configurations for the PX4 SITL instance of the simulated drone. In this file, the user can set the frequency

with which the PX4 publishes each message, define the active failsafe modes, tune the internal controllers of the autopilot, and configure the extended Kalman filter of the PX4. The user can also perform these configuration procedures with the QGroundControl, once the PX4 SITL instance is running. However, it is more convenient to use the **px4_config** file because it only has to be set once and the user can employ it to start a PX4 instance of as many simulated drones as desired. As an example, in the **px4_config** file produced for this thesis, the extended Kalman filter of the PX4 is already set to use the position and yaw measurements given by the motion capture system, as described in Section 3.3.2. This means that the user can perform simulations using this **px4_config** file knowing that the PX4 of each simulated drone will start with the extended Kalman filter already configured.

After receiving these arguments from the *gazebo.launch* file, the *drone.launch* program performs the following steps, represented in Fig. 4.5:

1. First, it generates a URDF model of the simulated drone, through the XACRO and BASE.XACRO files associated with the **vehicle_model** argument selected by the user. This is necessary because Gazebo does not accept vehicles represented by XACRO models. In this thesis, it was decided to work with XACRO files instead of working directly with URDF files because the first result in shorter and more readable models, that are also more easily modified.

2. Then, it launches a ROS node that, using the URDF vehicle model, spawns the drone in the Gazebo simulation. The vehicle is spawned in the position defined by the user in the **N**, **E**, and **D** arguments and with the initial attitude selected through **r**, **p**, and **y** parameters.

3. Finally, it launches a ROS node that starts a PX4 SITL instance for the new simulated vehicle, with the internal configurations defined in the file selected in the **px4_config** argument. This PX4 SITL instance will communicate with user programs through the **port** of the simulation computer selected by the user. The traffic of this **port** is then continuously routed, using the MAVLink-Router tool, to the same **port** of the external computers with the IP address defined in the **gcs_1_ip** and **gcs_2_ip** arguments, so user programs running in those computers can also communicate with the PX4 of the simulated vehicle.

The ROS nodes mentioned in the steps 2 and 3 are created inside the ROS namespace defined in the **group_ns** argument, present in line 15 of the *gazebo.launch* file (see Fig. 4.3). Namespaces allow the coexistence of ROS topics with the same name but that relate to different drones. This is important because every PX4 expects to receive the position and attitude data from the motion capture system over the */mavros/vision_pose/pose* and */mavros/vision/twist* topics, respectively. By running the PX4 node inside a namespace, the PX4 starts to expect receiving the motion capture data over the ***namespace****/mavros/vision_pose/pose* and ***namespace****/mavros/vision/twist* topics, instead. This means that, by defining a unique namespace for each drone in the **group_ns** argument, it is possible to simulate multiple vehicles without their topics names conflicting.

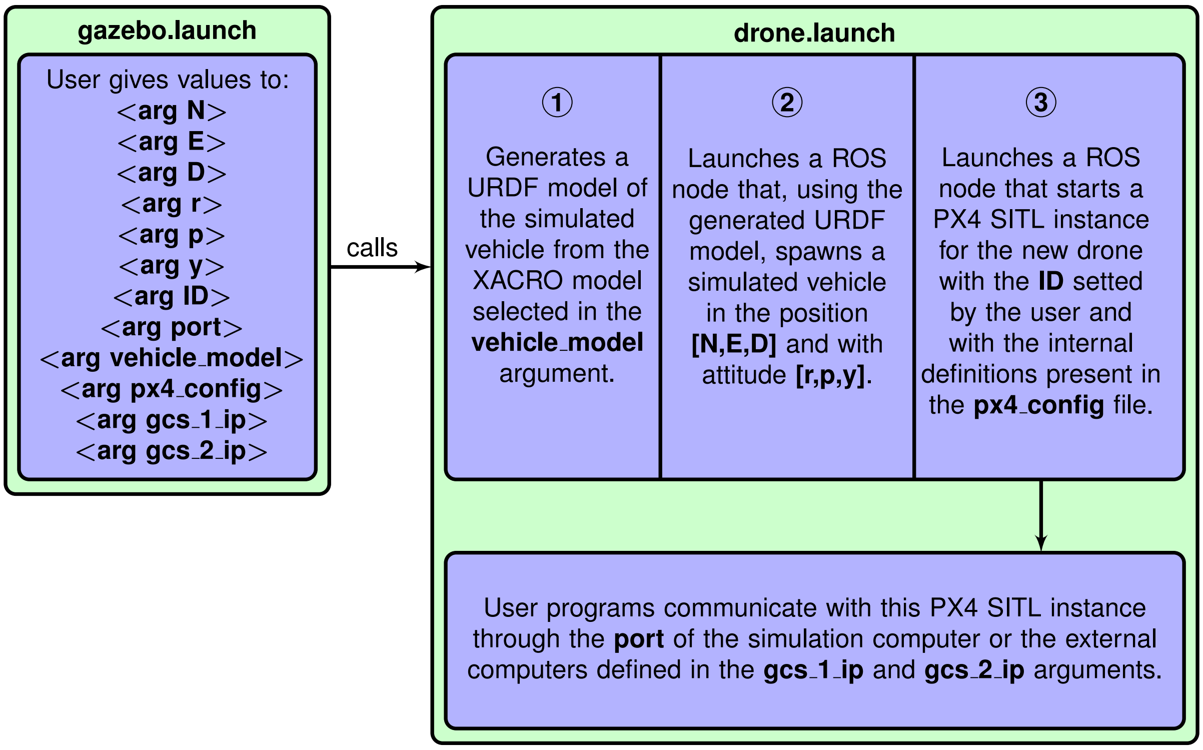

gazebo.launch
User gives values to:
<arg N>
<arg E>
<arg D>
<arg r>
<arg p>
<arg y>
<arg ID>
<arg port>
<arg vehicle_model>
<arg px4_config>
<arg gcs_1_ip>
<arg gcs_2_ip>
calls
drone.launch
① Generates a URDF model of the simulated vehicle from the XACRO model selected in the vehicle_model argument.
② Launches a ROS node that, using the generated URDF model, spawns a simulated vehicle in the position [N,E,D] and with attitude [r,p,y].
③ Launches a ROS node that starts a PX4 SITL instance for the new drone with the ID setted by the user and with the internal definitions present in the px4_config file.
User programs communicate with this PX4 SITL instance through the port of the simulation computer or the external computers defined in the gcs_1_ip and gcs_2_ip arguments.

Figure 4.5: Description of the process of spawning a vehicle and launching a PX4 SITL instance.

As stated in Section 4.2, the Gazebo software communicates with the PX4 SITL instances using the MAVLink protocol. The ports used for the communication are automatically defined in the *drone.launch* program, based on the **ID** argument selected by the user for each simulated drone.

In summary, an user can easily launch simulations through the *gazebo.launch* file. This file is divided into two code blocks. The first one calls the *empty_world.launch* program to start the Gazebo software in the **world** selected by the user. The **gui** argument defines if the graphical user interface of Gazebo will be launched or not. The second code block calls the *drone.launch* program to spawn a drone in the initiated Gazebo simulation and to start a PX4 SITL instance for that simulated drone. For multi-vehicle simulations, multiple blocks of this type must be created. The vehicle spawns in the position selected through the **N**, **E**, and **D** arguments and with the initial attitude selected through the **r**, **p**, and **y** parameters. The PX4 SITL instance starts with the internal configurations defined in the file selected over the **px4_config** argument. User programs can communicate with this PX4 instance through the **port** of the simulation computer or through the **port** of external computers defined in the **gcs_1_ip** and **gcs_2_ip** arguments. Finally, the motion of the vehicle is computed by the physics engine of Gazebo based on the physical properties of the drone and on the characteristics of sensors and actuators defined in the XACRO and BASE.XACRO files associated with the **vehicle_model**. Before spawning the drone, the XACRO files are converted in a URDF model because the first ones are not compatible with Gazebo.

To conclude this section, the graphical user interface that results from executing the *gazebo.launch* file with the exact code shown in Fig. 4.3, is presented in Fig. 4.6. As expected, it features the Iris quadcopter [25], the standard multicopter available for Gazebo, in the origin of an empty world with the most basic light and ground models.

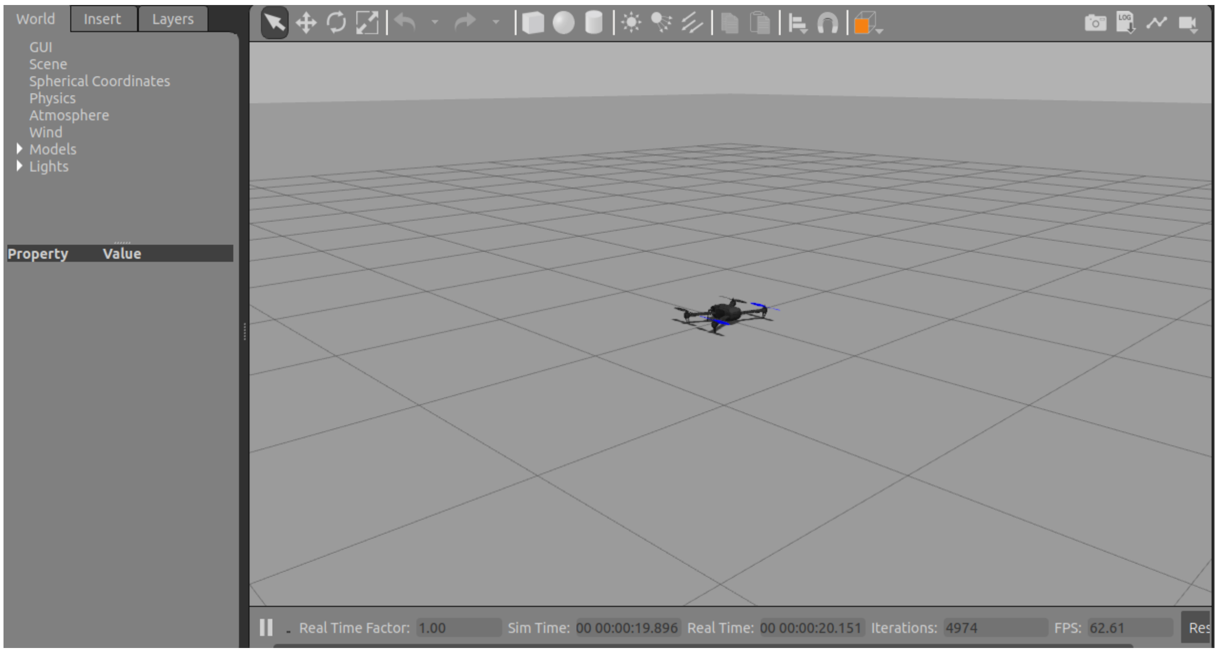


Figure 4.6: Resulting Gazebo's graphical user interface from executing the *gazebo.launch* file presented in this chapter.

4.4 Emulation of the motion capture system

In order to have a simulation environment with the same architecture as the real environment of the ISR Flying Arena, it is necessary to implement a motion capture system capable of providing, to the user programs and to the running PX4 SITL instances, high-frequency and low-latency measurements of the position and attitude of the simulated UAVs. Since there are no official Gazebo plugins or reliable third party tools to emulate a MOCAP system, contrary to what happens for most sensors, it was created a software program using the capabilities of the ROS middleware. More specifically, the developed program was built using rospy [26], the Python client library for ROS.

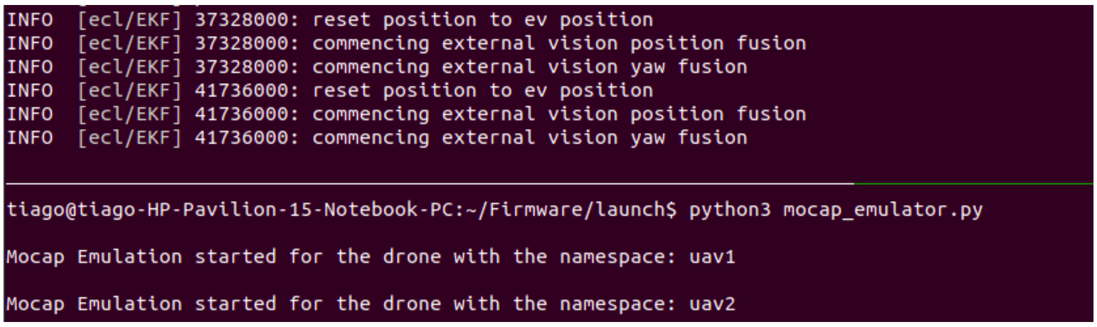
Given the design of the Flying Arena and according to Fig. 4.1, the MOCAP emulator must, as soon as the user launches the simulation, start running a process in the background that continuously retrieves the current position and attitude data of the simulated drones from Gazebo and sends it in the appropriate format to the correct PX4 SITL instances, while also making the data available to the user programs. In light of this, the developed program, *mocap_emulator.py*, performs the following steps:

1. First, it creates a ROS subscriber that listens to the *gazebo/model_states* topic, where the true position and attitude data of every object in the simulation is continuously being published at a frequency of 250 Hz. Every time a new message is published in this topic, a callback routine stores the new pose data of the vehicles, along with their ID, in a global variable. Consequently, this global variable will contain, for each simulated vehicle, a list with the format [**ID**, position, attitude] and each one of these lists will be constantly updated with the most recent pose of the vehicles. In conclusion, in this step, the true position and attitude of the drones are continuously retrieved from the Gazebo simulator and stored in a global variable.

2. Then, it reads the *gazebo.launch* file used to start the simulation and extracts the **ID** and **namespace** assigned by the user to each simulated drone. The **namespace** corresponds to the value of the group_uav argument. This step is indispensable because we need to know which **namespace** is associated with the **ID** of each drone, in order to be able to send the position and attitude obtained in the previous point to the PX4 topics of the correct vehicles. Recall that each PX4 expects to receive the position and attitude data from the motion capture system over, respectively, the ***namespace****/mavros/vision_pose/pose* and ***namespace****/mavros/vision/twist* topics.

3. In the last step and for each simulated vehicle, it creates a thread that runs a set of four ROS publishers. Each thread receives, as arguments, the **ID** and **namespace** of the specific drone it represents. Since the **ID** of the vehicle is known inside the thread, the true position and attitude are easily taken from the global variable that stores the lists in the format [**ID**, position, attitude]. Then, Gaussian white noise is added to the retrieved true position and attitude values, to represent the uncertainty of the measurements provided by a real motion capture system. Finally, based on the **namespace** of the vehicle, a pair of ROS publishers communicates the emulated position and attitude measurements to the PX4 SITL instance, through the topics mentioned in the last step, at a frequency of 60 Hz. The other pair of ROS publishers communicates the emulated position and attitude data to the user programs through two custom topics at a frequency of 180 Hz. These frequency values are the same ones used in the actual ISR Flying Arena. As stated in Section 3.2, user programs receive the pose updates at the work frequency of the Optitrack motion capture system to enable the development of offboard estimation algorithms. The PX4 autopilot receives the position and attitude measurements at a lower rate to avoid exhausting the bandwidth of the wireless communication channel, which could cause delays and loss of packets. Since only the MOCAP position and yaw measurements are used by the extended Kalman filter of the PX4 - to estimate the linear dynamics and correct the IMU drift, which are tasks performed at a low frequency - this downsampling procedure does not affect the performance of the EKF.

Before starting experiments in the simulation environment, the user should wait for the messages indicating that the MOCAP emulator is working and that the EKF of each PX4 is successfully using the external position and yaw measurements. Figure 4.7 shows an example of a terminal with these messages, for a simulation with two drones.



```
INFO  [ecl/EKF] 37328000: reset position to ev position
INFO  [ecl/EKF] 37328000: commencing external vision position fusion
INFO  [ecl/EKF] 37328000: commencing external vision yaw fusion
INFO  [ecl/EKF] 41736000: reset position to ev position
INFO  [ecl/EKF] 41736000: commencing external vision position fusion
INFO  [ecl/EKF] 41736000: commencing external vision yaw fusion

tiago@tiago-HP-Pavilion-15-Notebook-PC:~/Firmware/launch$ python3 mocap_emulator.py

Mocap Emulation started for the drone with the namespace: uav1

Mocap Emulation started for the drone with the namespace: uav2
```

Figure 4.7: Terminal with two panes, showing the messages confirming that the EKF of each PX4 (in the top pane) and the *mocap_emulator.py* program (in the bottom pane) are successfully working.

To conclude this section, it is highlighted the fact that the emulation of the motion capture system is an automatic process, independent from the user. This meets the goal of designing an intuitive and straightforward simulation environment. The only reason that could lead the user to modify the MOCAP emulator would be to adjust the standard deviation of the Gaussian white noise added to the true position and attitude of the drones. To ensure that this is easily achievable, the standard deviation value is defined in a YAML configuration file.

4.5 Reducing simulation overhead

The major drawback of the Gazebo simulator and, therefore, the developed simulation environment, is that it is computationally heavy. A researcher using their personal computer to run the Gazebo software will discover that, for multi-vehicle experiments, the simulation evolves slowly, at a rate much lower than real time. This can be a problem when consistently performing long tests because it can significantly delay research. Moreover, executing simulations in real time is important when, for example, users are testing control algorithms for UAV formations, in which part of the drones of the formation are real and flying in the actual ISR Flying Arena, and the other part is being simulated in the Gazebo software. Since the control signal computed for the real drones will depend on the position or velocity of the simulated ones, and vice-versa, the tests will only be successful if the simulation is running in real time and not at a slower rate.

In order to allow users to run Gazebo at the most convenient rate, a set of solutions that make simulations computationally lighter are presented below. Some of these solutions have already been briefly mentioned throughout this chapter.

- The first recommended solution is to run the simulations without the graphical user interface of Gazebo. This makes Gazebo start faster and use less system resources. The disadvantage of this solution is that the user cannot visualize the motion of the vehicles during the simulation. To overcome this downside, lightweight visualization tools were created in Python, in Matlab, and using the graphical user interface of Gazebo but without employing its physical engine. These tools will be presented in detail in Chapter 5. The Python visualization tool is the computationally lightest one and allows the user to monitor the 3D position of the vehicles during the simulation. However, the 3D representation of the drones is too simplistic and it does not provide information about the attitude of the vehicles. The Matlab and the Gazebo-based visualization tools produce a better 3D representation of the vehicles and include attitude information. However, these can only be used once the simulation is finished, since they reproduce the data saved in the log files.
- The second recommended solution is to remove from the vehicle models all sensor plugins that are not being used during the experiments. The plugins that model the behavior of the sensors are found in the BASE.XACRO file. In the vehicle models used in this thesis for validation of control solutions, the GPS, the magnetometer, and the barometer plugins were removed, because the extended Kalman filter of the PX4 autopilot uses the data from the motion capture system as external

position and yaw sources. If these plugins were still present during the experiments, the physics engine of Gazebo would have to keep computing new measurements for these sensors, creating unnecessary overhead in the simulation. When a sensor plugin is removed, it is usually necessary to change some internal settings of the PX4. For example, when the magnetometer and barometer plugins were removed, it was necessary to change the SIS_HAS_BARO and SIS_HAS_MAG flags because, otherwise, the PX4 pre-flight tests would not pass, and it would not be possible to arm the vehicle.

- The third recommended solution consists of using a dedicated computer for the simulation or even dividing the simulated drones over multiple different computers. This solution distributes the computational power required for the simulation over several machines. It also takes advantage of the design adopted for the ISR Flying Arena, which allows user programs to communicate with any drone regardless of whether it is real or simulated on any of the computers of the local network. When conducting experiments with drones simulated across multiple computers, it is advisable to analyze the logs and reproduce them in the visualization tools developed in this thesis to make sure that no collisions have occurred.

- The final alternative is only recommended for experienced users and consists of reducing the performance of the physics engine solver of Gazebo. There is a natural trade-off between the performance of the solver and the amount of system resources used. Thus, if the user recognizes that the physics engine is over-performing for the particular experiment being carried out, it can decrease, for example, the number of iterations performed by the solver in order to increase the time rate of the simulation.

4.6 General overview

The simulation environment allows testing of navigation, guidance, and control solutions in a safe and non-compromising way. The development of this environment required the emulation of a motion capture system, the integration of Gazebo (a software capable of simulating the dynamics of vehicles represented by complex customizable models), and the integration of the PX4 firmware, as shown in Fig. 4.8.

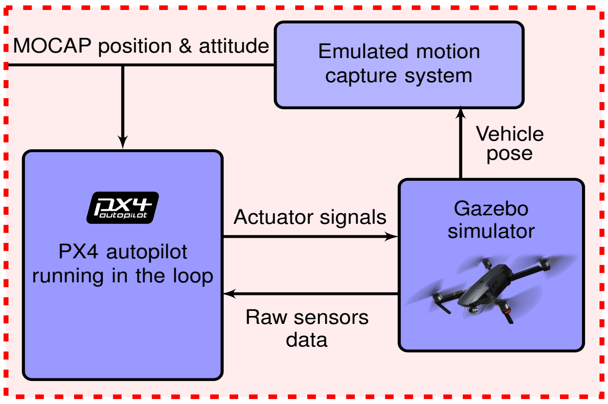

MOCAP position & attitude
Emulated motion capture system
Vehicle pose
PX4 autopilot running in the loop
Actuator signals
Raw sensors data
Gazebo simulator

Figure 4.8: Blocks implemented and integrated in the simulation environment.

Throughout this chapter, the set of programs developed to start simulations automatically, according to the settings desired by the user for Gazebo, for the vehicles, and for the PX4 SITL instances were presented. A new and final representation of these programs is depicted in Fig. 4.9. To launch a simulation, the user only needs to perform two steps: assign values to the arguments of the *gazebo.launch* file and run the *simulator.sh* executable. The *simulator.sh* is a bash program that automatically initiates the *gazebo.launch* and the *mocap_emulator.py* programs, which can be a complicated procedure for users that are less experienced with ROS.

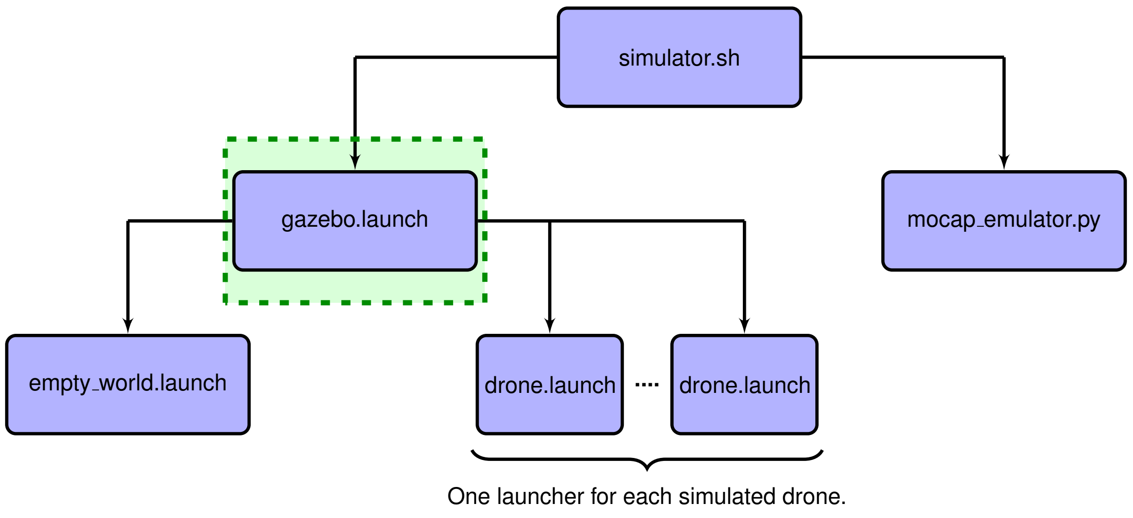


Figure 4.9: Launching process of the simulation environment.

In a brief and final summary of the remaining simulation launching process, the *empty_world.launch* program starts Gazebo in the requested world, the *drone.launch* program spawns a vehicle in the simulation and starts a PX4 SITL instance for that vehicle, and the *mocap_emulator.py* is the Python solution developed to emulate the motion capture system.

The next chapter presents the modules developed to allow user programs to easily perform experiments in both the real and the simulation environments.

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.

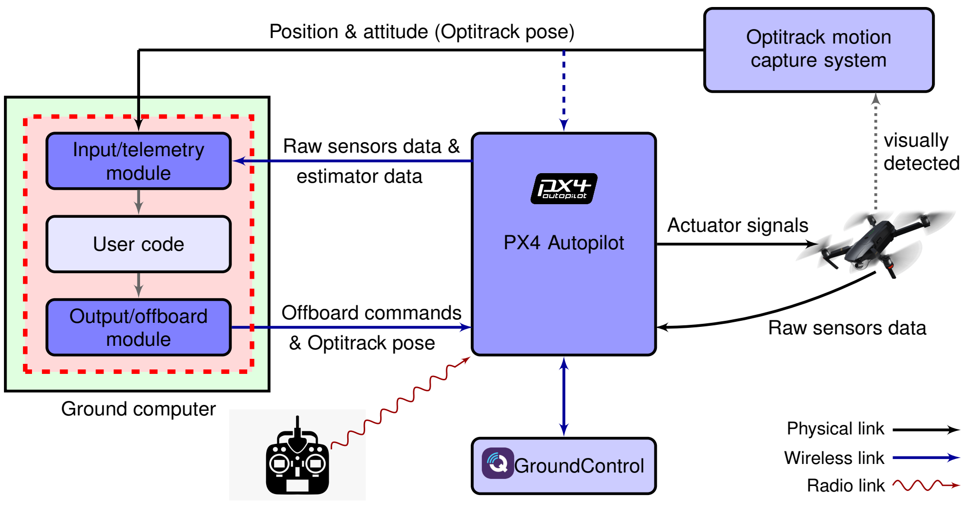


Figure 5.1: Architecture of the ISR Flying Arena, with the modules of the ground computer highlighted.

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 [27]. For this thesis, the high-level communication libraries selected, MAVROS C++ [28], MAVROS Python [28], MAVSDK C++ [29], and MAVSDK Python [30], 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.

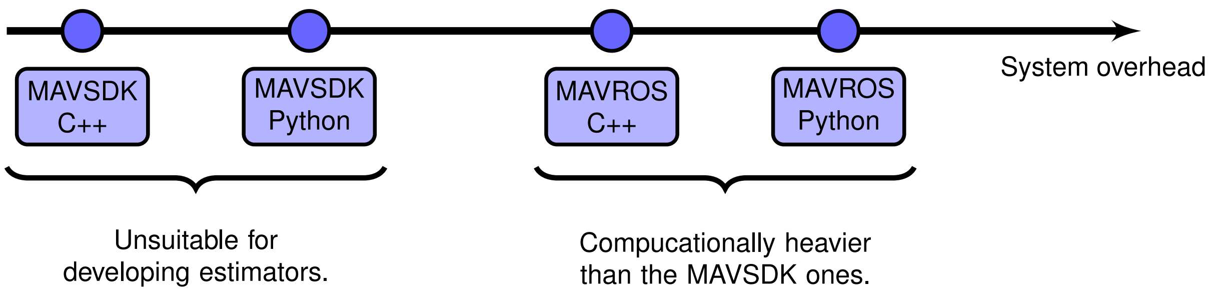

MAVSDK C++
MAVSDK Python
MAVROS C++
MAVROS Python
System overhead
Unsuitable for developing estimators.
Compucationally heavier than the MAVSDK ones.

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.

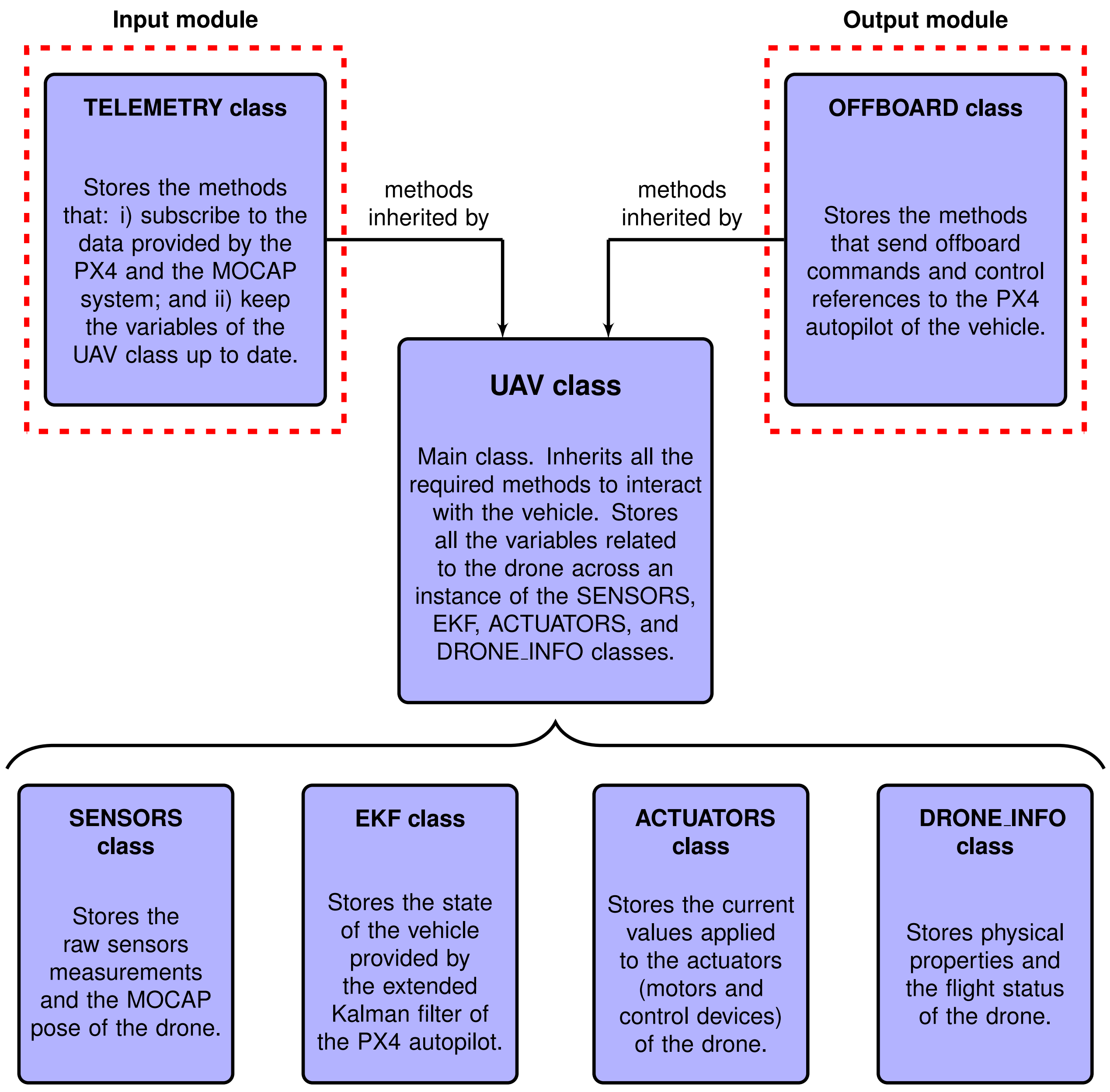

Input module
TELEMETRY class
Stores the methods that: i) subscribe to the data provided by the PX4 and the MOCAP system; and ii) keep the variables of the UAV class up to date.
methods inherited by
methods inherited by
Output module
OFFBOARD class
Stores the methods that send offboard commands and control 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.

```
 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 [31]. 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 [32, 33] 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 [34]. 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 [35] function of the MAVSDK C++ API and through the system.connect [36] 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 [37]. These files can be converted to **CSV** or can be analyzed in open-source software programs [38]. The PX4 logging system complements the developed offboard logger.

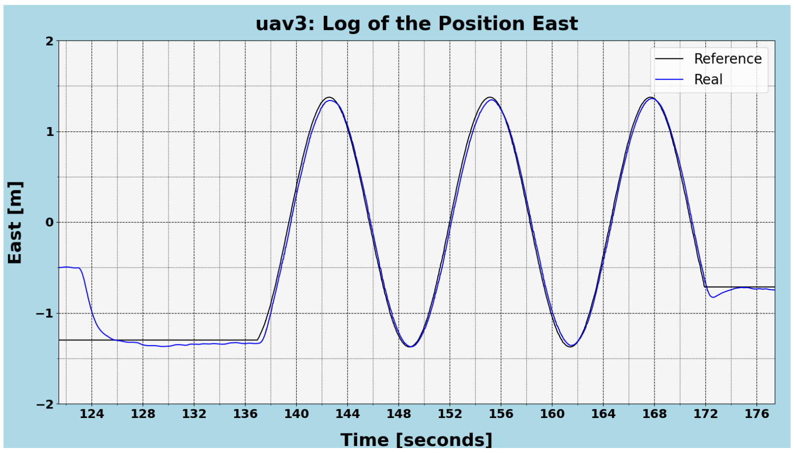


Figure 5.9: Plot of the log of the actual (blue) and desired (black) position east of a drone.

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.

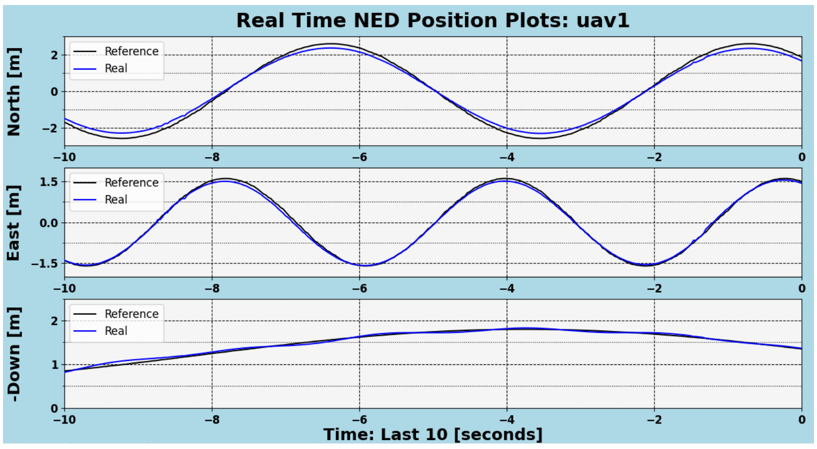


Figure 5.10: Window with the 3 components of the actual (blue) and desired (black) position of the drone.

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.

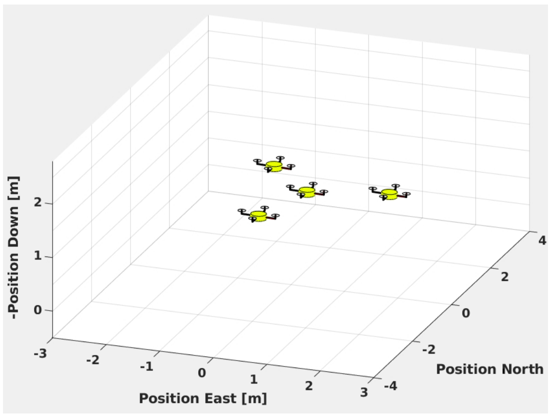


Figure 5.11: Matlab visualization tool plotting, in a single window, a formation of four vehicles, two of them simulated in one computer and the other two simulated in another computer.

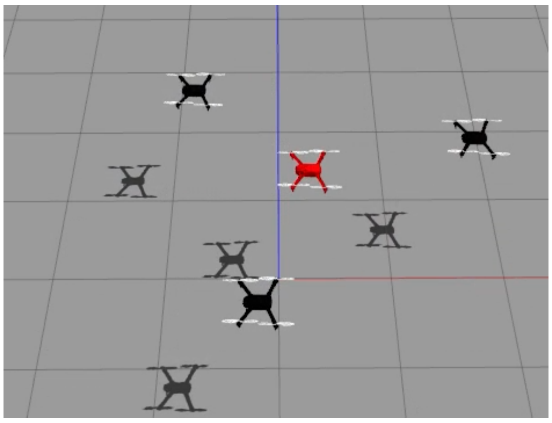


Figure 5.12: Gazebo-based visualization tool plotting, in a single window, a formation of four vehicles, one of them real and the other three simulated.

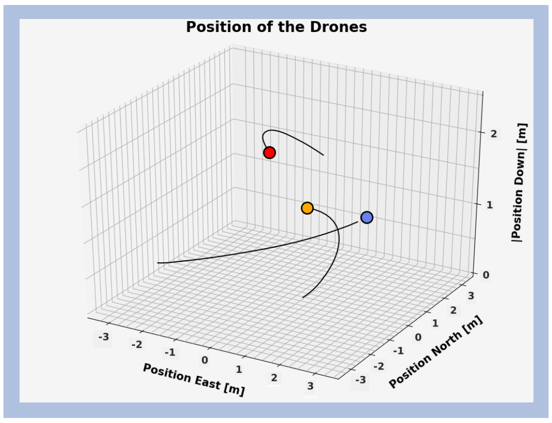


Figure 5.13: Python visualization tool showing the motion of three drones.

Chapter 6

Tests and Results

This chapter presents the set of experiments performed to validate the architecture and the software programs designed for the ISR Flying Arena. Section 6.1 features the tests conducted to confirm that the failsafe modes behave as expected, ensuring the safety of researchers and equipment. Sections 6.2 and 6.3 exhibit the results of the implementation of offboard controllers using multiple communication libraries, modules, tools, computers, and vehicles, attesting that the developed setup is flexible, scalable, and enables the rapid deployment of GNC solutions. Additionally, a group of experiments created for public demonstrations is also presented. Section 6.4 presents the results of the deployment of a formation-control algorithm, evidencing that the ISR Flying Arena is suitable for testing multi-vehicle control solutions that depend on real and simulated aerial vehicles interacting with each other. Finally, Section 6.5 shows that the created setup is already being used by other students to rapidly and effortlessly validate their researches, which is solid evidence that the goals of this thesis have been met.

6.1 Failsafe modes

The first set of tests aimed to verify that the failsafe modes of the PX4 autopilot were correctly configured. These safety features, described and set in Section 3.3.1, protect the users and the equipment of the ISR Flying Arena when an unexpected event occurs. In order to attest the proper configuration and functioning of the failsafe modes, the following experiments were conducted:

- Low battery failsafe test - performed in order to check if the PX4 autopilot of the vehicles transitions to auto-land mode when the battery capacity drops below a defined threshold.
- RC loss failsafe test - carried out to verify if the PX4 autopilot of the vehicles transitions to auto-land mode when the communication link with the RC controller is lost.
- Position loss failsafe test - executed to check if the vehicles disarm/shutdown when the quality of the position estimate drops below acceptable levels. As explained in Section 3.3.1, in this particular case, the vehicles were configured to not switch to auto-land mode since the position and attitude estimates are no longer accurate enough for an autonomous landing maneuver.

- Offboard loss failsafe test - performed to ensure that the PX4 of the vehicles switches to auto-land mode when the offboard communication link is lost. This usually happens when the user program fails, due to an error, and stops communicating with the vehicle.

Additionally, another set of tests was also carried out to guarantee that the users can stop the experiment at any time, even if none of the above failsafe modes was activated. This is usually required when the GNC algorithm implemented by the user is not behaving as expected and, consequently, the stability of the drones is compromised or the vehicles are at risk of collision with each other, with the walls, or with the Optitrack cameras. With this in mind, the following experiments were conducted:

- QGroundControl test - performed to check if the QGroundControl successfully communicates with the PX4 and if researchers can use it to auto-land or shutdown the vehicles at any moment.
- RC controller test - executed in order to verify that the RC controller has been correctly installed and that the users can auto-land or shutdown the vehicles, at any moment, using the RC controller.
- Dedicated software tool test - carried out to guarantee that the user can auto-land or shutdown the vehicles, at any moment, using the dedicated software tool developed in this thesis. The advantage of this tool over the QGroundControl station and the RC controller is related to scalability: regardless of the number and type of drones employed, the users only need to click on a button provided by the software tool to immediately stop the experiment; they do not need to manage multiple RC controllers or select the drones they want to land one by one in the QGroundControl.

The described tests were performed and successfully completed by all vehicles available in the ISR Flying Arena. These tests were also carried out in the simulation environment, to ensure that the safety features of the simulated vehicles are consistent with the ones adopted for the real drones. The careful testing and validation of the failsafe modes, in conjunction with the use of the simulation environment to ensure that the GNC algorithms were working properly, made it possible to perform over 300 flights in the ISR Flying Arena without any accident and without damaging any equipment.

6.2 PX4 position controller

Once the safety conditions were guaranteed, it was viable to conduct more sophisticated tests on the architecture, features, modules, and tools developed for the ISR Flying Arena. The first experiment consisted of a multi-vehicle situation in which two drones track a set of desired setpoints using the internal position controller of the PX4 autopilot. This test allowed to establish a high level of confidence in the developed setup before advancing to more complex and demanding trajectories and before introducing custom controllers in the loop. It also enabled to ensure that the position controller of the PX4 autopilot is stable and that it can be used by researchers to rapidly validate navigation/estimation algorithms without having to invest time implementing a control solution. This experiment was repeated multiple times in order to test both the real and the simulation environments, and in order to test all software modules developed. Table 6.1 features the description of each test repetition.

		UAV1		UAV2	
Test Number	Test Trajectory	Type	Module	Type	Module
1	Setpoints Tracking	Simulated	MAVROS Python	Simulated	MAVROS Python
2	Setpoints Tracking	Simulated	MAVROS C++	Simulated	MAVROS C++
3	Setpoints Tracking	Simulated	MAVSDK Python	Simulated	MAVSDK Python
4	Setpoints Tracking	Simulated	MAVSDK C++	Simulated	MAVSDK C++
5	Setpoints Tracking	Simulated	MAVROS Python	Simulated	MAVROS C++
6	Setpoints Tracking	Simulated	MAVSDK Python	Simulated	MAVSDK C++
7	Setpoints Tracking	Real	MAVROS Python	Real	MAVROS Python
8	Setpoints Tracking	Real	MAVROS C++	Real	MAVROS C++
9	Setpoints Tracking	Real	MAVSDK Python	Real	MAVSDK Python
10	Setpoints Tracking	Real	MAVSDK C++	Real	MAVSDK C++
11	Setpoints Tracking	Real	MAVROS Python	Real	MAVSDK Python
12	Setpoints Tracking	Real	MAVROS C++	Real	MAVSDK C++
13	Setpoints Tracking	Simulated	MAVROS Python	Real	MAVROS Python
14	Setpoints Tracking	Real	MAVROS C++	Simulated	MAVROS C++
15	Setpoints Tracking	Simulated	MAVSDK Python	Real	MAVSDK Python
16	Setpoints Tracking	Real	MAVSDK C++	Simulated	MAVSDK C++
17	Setpoints Tracking	Simulated	MAVROS Python	Real	MAVSDK C++
18	Setpoints Tracking	Real	MAVROS C++	Simulated	MAVSDK Python

Table 6.1: Description of the tests performed with the setpoints tracking trajectory.

These experiments test all four software modules described in Chapter 5 - that were programmed using different communication libraries - in both the real and the simulation environments. Figures 6.1 and 6.2 exhibit two drones performing the setpoints tracking test in the Gazebo simulator and in the ISR Flying Arena, respectively. Note that the user can employ different communication libraries to control each of the drones of an experiment, as demonstrated, for instance, in tests 5 and 6. Note also, through tests 13 to 18, that each experiment can have simultaneously real and simulated vehicles. Figures 6.3 and 6.4 depict the two drones of test 13 in the same graphical window, proving that the created visualization tools enable the user to monitor vehicles from different environments in the same display.

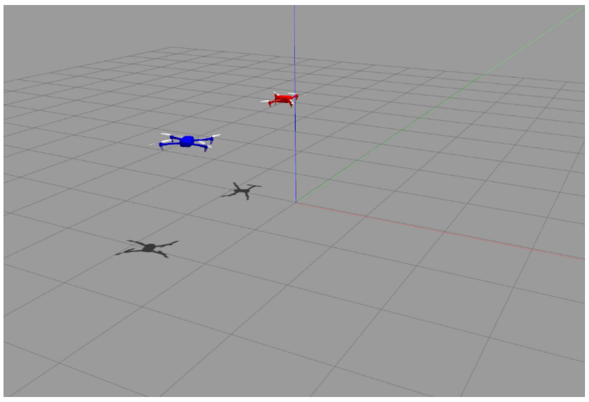


Figure 6.1: Two drones performing the setpoints tracking test in the simulation environment.

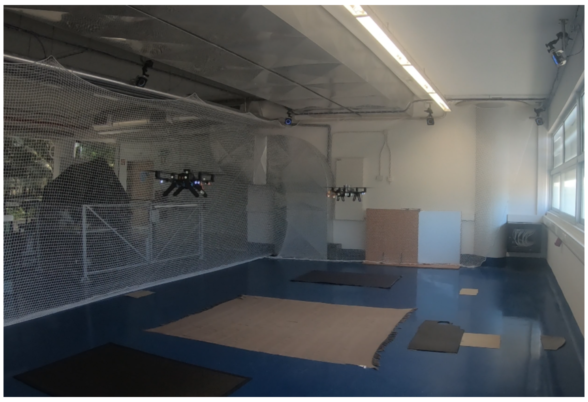


Figure 6.2: Two drones performing the setpoints tracking test in the ISR Flying Arena.

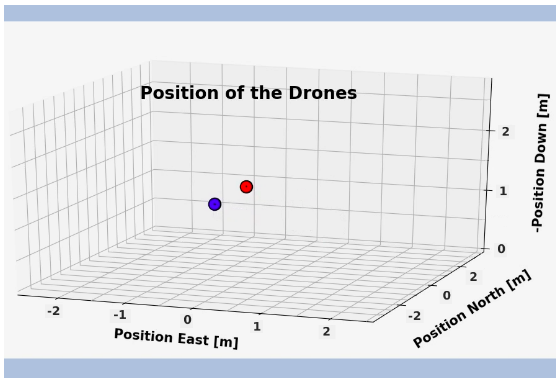


Figure 6.3: Python visualization tool showing the drones of test 13 in the same window.

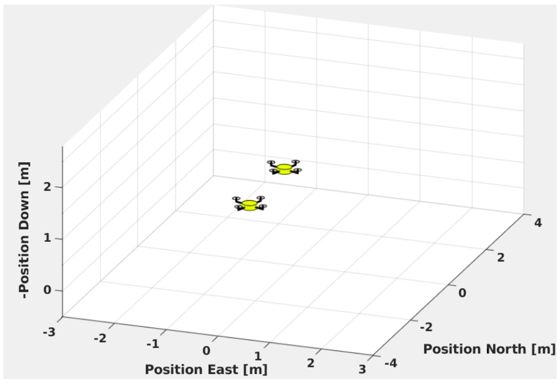


Figure 6.4: Matlab visualization tool showing the drones of test 13 in the same window.

Finally, it is presented in Fig. 6.5 the evolution of the desired, simulated, and real altitude of the UAV1 with time, during the setpoints tracking test. The responses obtained in the real and simulated environments are almost coincident, which demonstrates the importance of the simulator for a first validation of the GNC algorithms before advancing to tests with real vehicles. It is important to note that the response of the drones is independent of the communication library used. The plot of Fig. 6.5 was generated with the tools developed in this thesis, presented in Section 5.5.

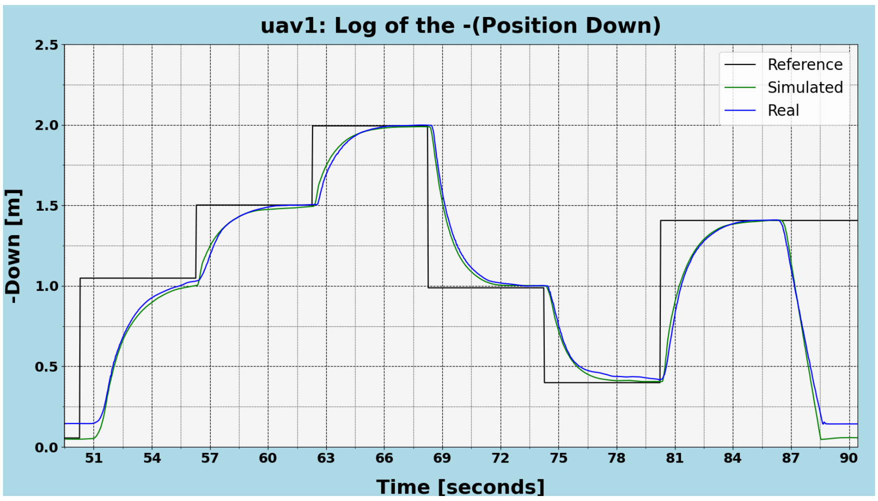


Figure 6.5: Evolution of the altitude of the UAV1 with time in the setpoints tracking test.

The fact that all tests documented in Table 6.1 have been carried out with success, proves that the systems of the ISR Flying Arena work as designed. The position and attitude of the vehicles generated from both the Optitrack and the emulated MOCAP System are indeed reaching the PX4 autopilots, that successfully merge this data with the IMU measurements. Similarly, these tests attest that user programs and the PX4 autopilot are, in fact, exchanging commands and telemetry information. This is true for the modules of all four communications libraries (MAVROS C++, MAVROS Python, MAVSDK C++, and MAVSDK Python). These experiments help validate the architecture, the configuration process, and the programs created for both the real and the simulation environments, exposed in Chapters 3 and 4 respectively, as well as the set of software modules responsible for allowing communication between the user programs and the PX4 autopilot of each vehicle, presented in Chapter 5.

It should be noted that the first six tests documented in Table 6.1 were performed and successfully completed twice. In the first time, the two drones were simulated in the same computer. In the second time each drone was simulated in a different machine. This proved that the designed setup enables to distribute the computational power necessary for a multi-vehicle simulation across several computers. In a similar way, the tests 7 to 12 of Table 6.1 were successfully conducted twice. In the first time the user programs were running in the same computer whilst in the second time, the user programs were running in different machines. This confirmed that the designed setup also allows to run user programs across different computers.

To conclude this section, an extra set of tests were carried out to improve the tuning of the internal position controller of the PX4 autopilot. These tests are described in Table 6.2 and employ the trajectories used in the remaining experiments of this chapter. All tests were completed with success as shown in Fig. 6.6, that presents the obtained response for the evolution of the north position of the UAV1 with time, in test number 28. This means that users can safely employ the internal position controller of the PX4 to validate estimation algorithms, in the same way that users testing control solutions can safely depend on the extended kalman filter of the PX4 autopilot to obtain estimates of the state of the vehicles.

		UAV1	
Test Number	Test Trajectory	Type	Module
19	Lissajous Curve 1	Simulated	MAVROS Python
20	Lissajous Curve 1	Real	MAVROS Python
21	Lissajous Curve 2	Simulated	MAVROS C++
22	Lissajous Curve 2	Real	MAVROS C++
23	Lissajous Curve 3	Simulated	MAVSDK Python
24	Lissajous Curve 3	Real	MAVSDK Python
25	Letters Demonstration	Simulated	MAVSDK C++
26	Letters Demonstration	Real	MAVSDK C++
27	Formation Leader	Simulated	MAVROS Python
28	Formation Leader	Real	MAVROS Python
29	Estimator Validation	Simulated	MAVROS C++
30	Estimator Validation	Real	MAVROS C++

Table 6.2: Description of the remaining tests performed with the position controller of the PX4 autopilot.

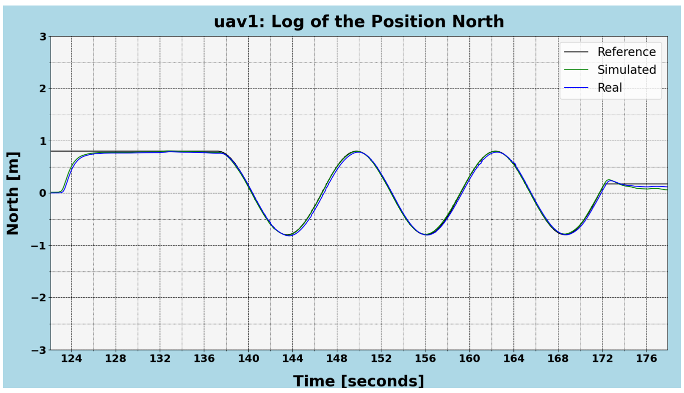


Figure 6.6: Evolution of the north position of the UAV1 with time in the formation-control test.

6.3 Custom PID trajectory tracking controller

This section aims to prove that the created setup enables the rapid implementation and test of control solutions. For this purpose, a classic PID trajectory tracking controller was deployed, tuned, and tested in a set of fast and demanding trajectories. The implemented position controller acts as an outer loop controller, since it provides attitude and thrust references to the attitude controller of the PX4 autopilot, analyzed in Section 2.1.1. A descriptive list of the experiments carried out with the deployed control algorithm can be found in Table 6.3. Note that each experiment is repeated several times in order to allow the testing of all software modules, tools, and configuration files created for the ISR Flying Arena.

		UAV	
Test Number	Test Trajectory	Type	Module
31	Lissajous Curve 1	Simulated	MAVROS Python
32	Lissajous Curve 1	Real	MAVROS Python
33	Lissajous Curve 1	Simulated	MAVSDK Python
34	Lissajous Curve 1	Real	MAVSDK Python
35	Lissajous Curve 2	Simulated	MAVROS C++
36	Lissajous Curve 2	Real	MAVROS C++
37	Lissajous Curve 2	Simulated	MAVSDK C++
38	Lissajous Curve 2	Real	MAVSDK C++
39	Lissajous Curve 3	Simulated	MAVROS Python
40	Lissajous Curve 3	Real	MAVROS Python
41	Lissajous Curve 3	Simulated	MAVROS C++
42	Lissajous Curve 3	Real	MAVROS C++
43	Letter F	Simulated	MAVSDK Python
44	Letter F	Real	MAVSDK Python
45	Letter L	Simulated	MAVSDK C++
46	Letter L	Real	MAVSDK C++
47	Letter Y	Simulated	MAVROS Python
48	Letter Y	Real	MAVROS Python
49	Word FLY	Simulated	MAVROS Python

Table 6.3: Description of the tests performed with the PID position tracking controller.

The trajectories of tests 31 to 42 consist of sinusoidal parametric equations known as Lissajous curves [39], that are frequently employed in strategies of aerial surveillance. These selected trajectories are also aggressive (requiring from vehicles roll and pitch angles of $\pm 20^{\circ}$ for successful trajectory tracking) and visually appealing so that they can be used in public demonstrations of the ISR Flying Arena. Figures 6.7 and 6.8 exhibit two quadrotors performing the first Lissajous trajectory in the Gazebo simulator and in the ISR Flying Arena, respectively. Figures 6.9, 6.10, and 6.11 show the top view of each of the Lissajous curves adopted while Fig. 6.12 features the altitude profile of these three paths. The points in these plots represent the set of positions covered by the vehicles during the tests. The plots were generated through the Python visualization tool after correctly setting up its YAML configuration file.

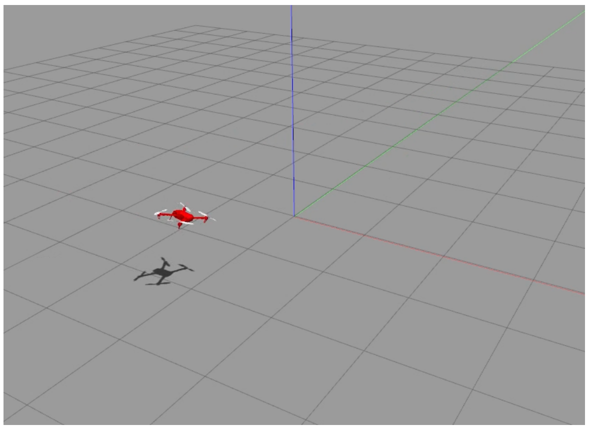


Figure 6.7: Quadcopter performing the Lissajous curve 1 test in the simulation environment.

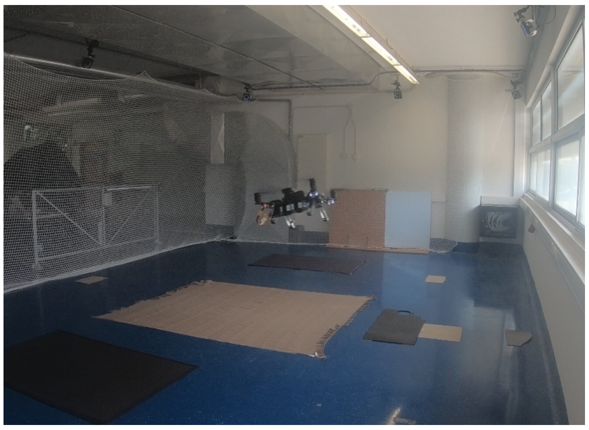


Figure 6.8: Quadcopter performing the Lissajous curve 1 test in the ISR Flying Arena.

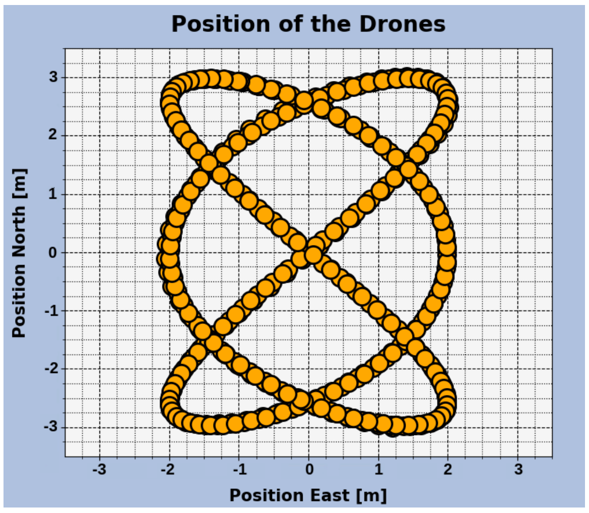


Figure 6.9: Top view (or North-East view) of the Lissajous curve 1.

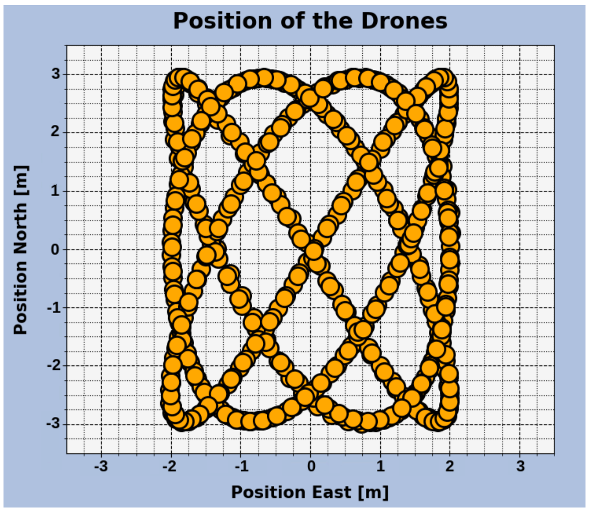


Figure 6.10: Top view (or North-East view) of the Lissajous curve 2.

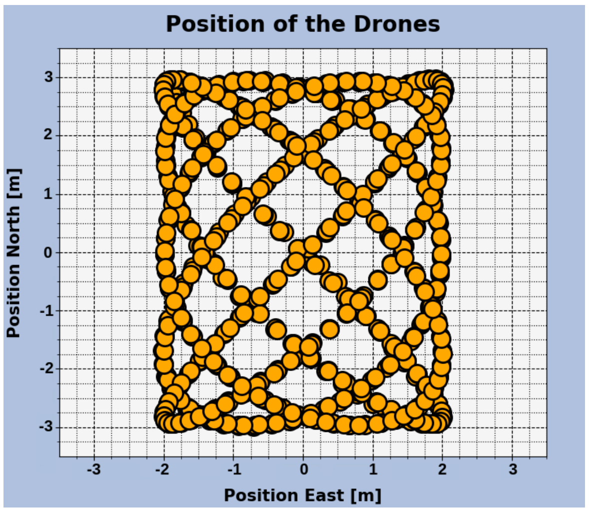


Figure 6.11: Top view (or North-East view) of the Lissajous curve 3.

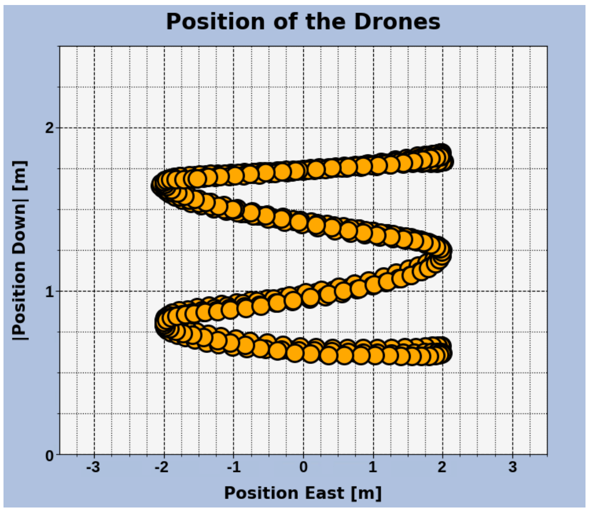


Figure 6.12: Altitude profile adopted for the Lissajous trajectories.

Figures 6.13 and 6.14 present the evolution of the north position and the altitude with time, respectively, for the drone of test 36, whereas Fig. 6.15 exhibits the evolution of the pitch angle with time for the vehicle of test 40.

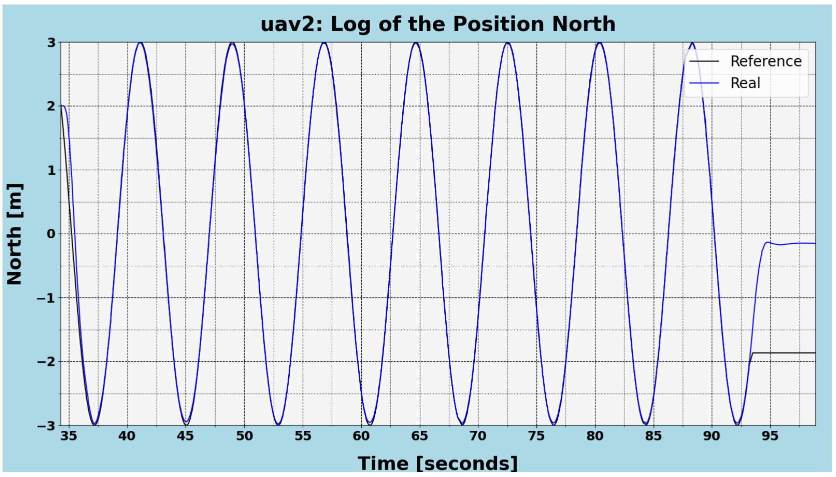


Figure 6.13: Evolution of the north position with time for the UAV of test 36.

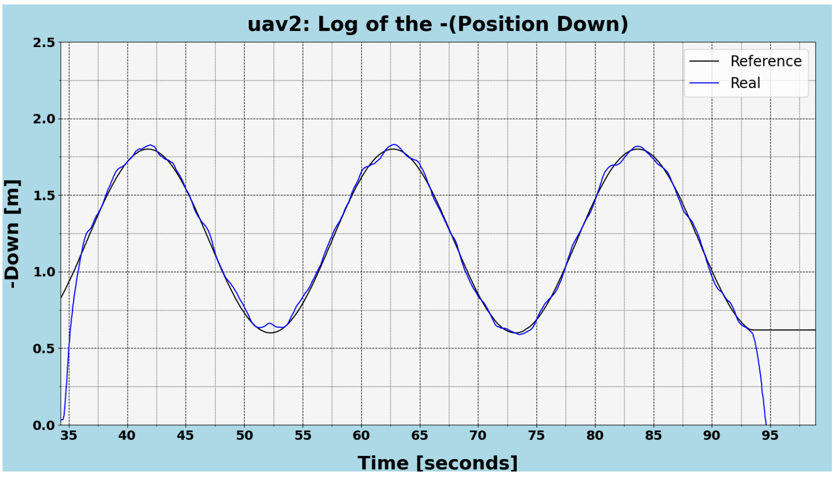


Figure 6.14: Evolution of the altitude with time for the UAV of test 36.

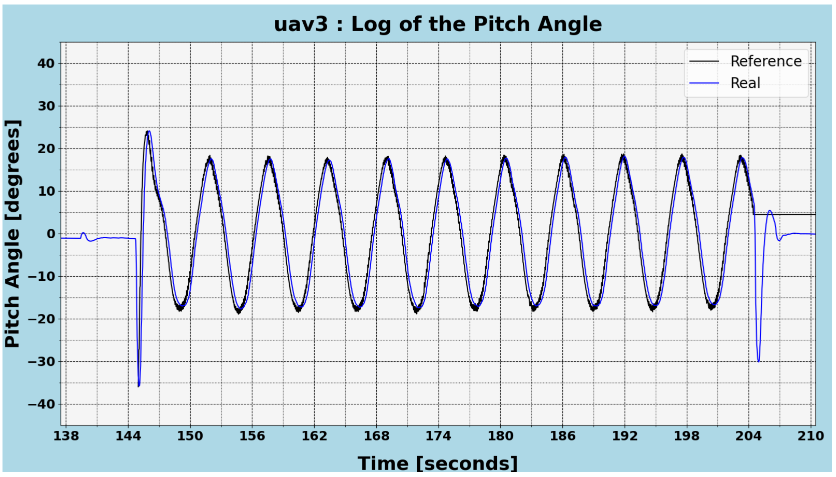


Figure 6.15: Evolution of the pitch angle with time for the UAV of test 40.

Through the analysis of the experiments described in Table 6.3, which were all completed with success, and through the study of Figures 6.9 to 6.15, it is possible to draw the following conclusions:

- First, the created setup allowed to rapidly deploy and test the PID position tracking controller in both the real and the simulation environments, using different software modules. The position controller was also able to track all trajectories. The implementation and validation of the controller was a fast process because, for each test, it was only necessary to create a user program with the control algorithm and the trajectory. The remaining systems of the ISR Flying Arena were already programmed and configured, and were easily reused, which saved several weeks of work.

- Second, the software tools presented in Section 5.5 not only work as intended but also helped in the rapid validation process of the position controller. These tools provided CSV flight logs and plots with the evolution of the state of the vehicle with time as soon as the experiments were completed (Figures 6.13 to 6.15 are examples of these plots), which allowed to immediately evaluate the performance of the controller.

- Finally, that the attitude controller of the PX4 autopilot is stable for both the simulation vehicles and the drones available at the ISR Flying Arena, which allows future users to safely employ them in their projects.

The developed position controller was also employed in tests 43 to 49. These experiments consisted of slow trajectories with letter shapes, mostly intended to be reproduced in public demonstrations. Through these tests it was possible to validate the scalability of the adopted architecture. Fig. 6.16 showcases almost 50 vehicles performing a trajectory that spells the word FLY, in the simulation environment. Due to space and equipment limitations, it was not possible to carry out this exact experiment in the ISR Flying Arena. It was only possible to execute each letter individually with one and then two real vehicles. Nevertheless, these tests were important as another proof of the robustness of the architecture and the software modules developed in this work.

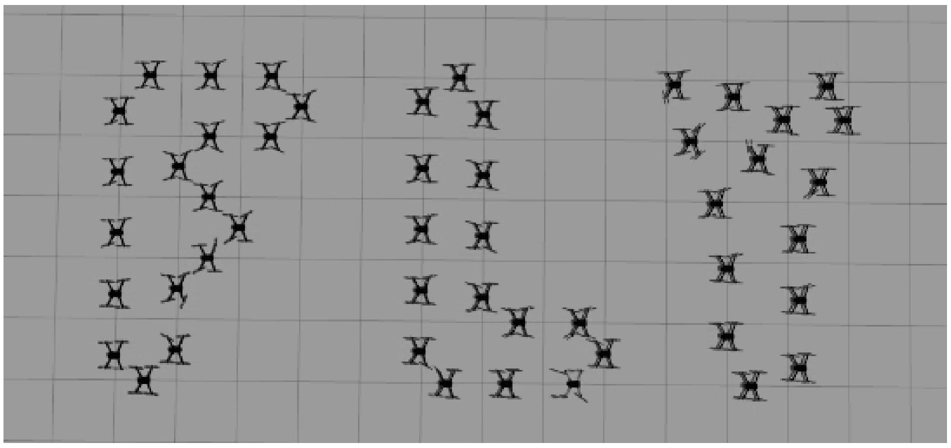


Figure 6.16: Set of quadcopters performing the test number 49 in the simulation environment.

6.4 Formation-control algorithm

In order to prove that the ISR Flying Arena is also suitable for testing complex multi-vehicle control solutions that require the emulation of sensors which are not available in the multicopters, a formation-control algorithm [40] was deployed and tested. The formation topology adopted is exhibited in Fig. 6.17, where arrow direction indicates flow of information/measurements. The vehicle 1 is the formation leader whereas the remaining vehicles are followers. In the devised experiment, the leader tracks a time-varying trajectory and the followers orbit around him. Since the control algorithm for the followers depends on their position and velocity relative to other drones of the formation (for instance, the control algorithm of vehicle 2 depends on its position and velocity relative to drone 1 and 4, as indicated by the arrows), it was necessary to emulate relative position and velocity sensors. The emulation of sensors was already described in Section 5.5.2. Table 6.4 features the description of the tests performed with this time-varying formation.

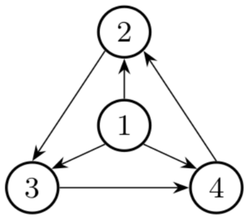


Figure 6.17: Adopted formation topology.

Test Number	Test Trajectory	UAV1 (Leader)	UAV2	UAV3	UAV4
50	Time-varying formation	Simulated	Simulated	Simulated	Simulated
51	Time-varying formation	Real	Simulated	Simulated	Simulated
52	Time-varying formation	Real	Real	Simulated	Simulated
53	Time-varying formation	Simulated	Simulated	Real	Real

Table 6.4: Description of the tests performed with the time varying formation.

Fig. 6.18 shows the quadcopters of test 50 performing the experiment in the simulation environment, proving the scalability of the created testbed.

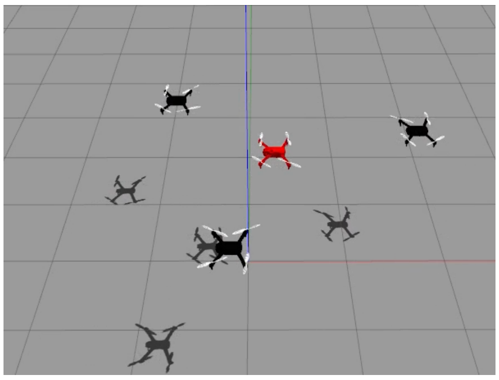


Figure 6.18: Quadcopters performing the time varying formation trajectory in simulation.

Figures 6.19 and 6.20 show the two simulated and the two real vehicles of test 52 performing the experiment. This test demonstrates that the created setup enables the validation of the formation control algorithm (which depends on the relative position and velocity between vehicles) with half of the drones of the formation flying in the Arena and the other half being simulated in the Gazebo software. This property allows researchers to validate GNC solutions in simulation/mixed-environment when the actual sensors are not available and to extend testing to conditions that are unfeasible in the actual flying arena. During and after the experiment, users can monitor these four vehicles in the same graphical window using the developed Gazebo-based or Matlab-based visualization tools, as shown in Figures 6.21 and 6.22. Additionally, Fig. 6.23 attests that it is possible to visualize the two real drones (exhibited in blue) in the same Gazebo instance where the other drones (exhibited in black) are being simulated. The drones in visualization mode are easily identified because their rotors are always straight and static.

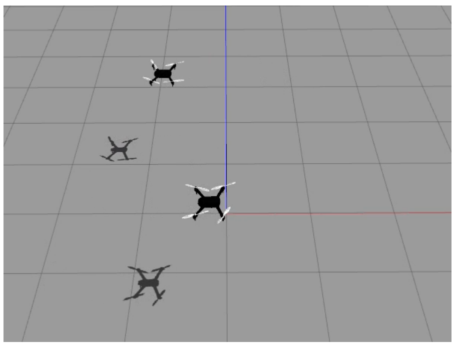


Figure 6.19: Simulation of the drones 3 and 4 of test number 52.

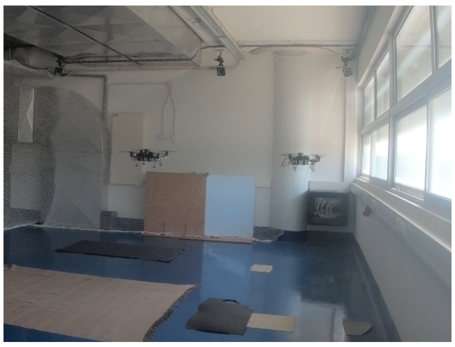


Figure 6.20: Vehicles 1 and 2 of test number 52 flying in the ISR Flying Arena.

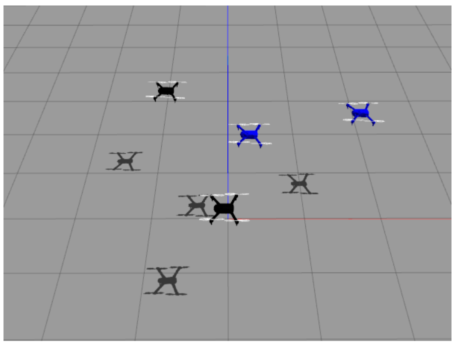


Figure 6.21: Gazebo-based visualization tool displaying the drones of test 52 in the same window.

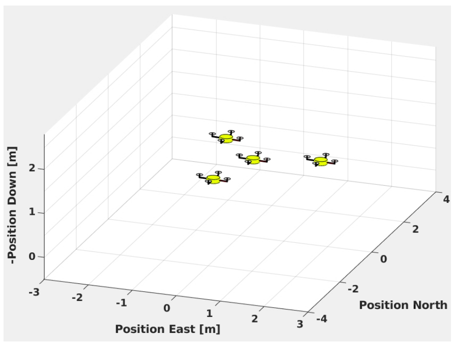


Figure 6.22: Matlab-based visualization tool showing the drones of test 52 in a single window.

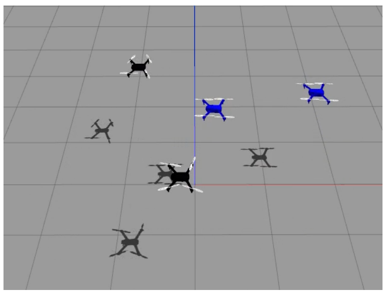


Figure 6.23: Gazebo window with two drones being simulated and two drones in visualization mode.

Figures 6.24 to 6.27 exhibit the results obtained in test 52. The real quadcopters are represented in orange and blue. The orange vehicle is the leader and the remaining drones are the followers. Fig 6.24 presents the convergence of the vehicles from their initial position to their formation position. Figures 6.25 to 6.27 prove that, after the initial convergence, the followers successfully kept the formation while orbiting around the moving leader. The results were similar for all experiments documented in Table 6.4. The plots of Figures 6.24 to 6.27 were obtained using the developed Python visualization tool, which demonstrates the important role of the supporting tools presented in Section 5.5 for the rapid validation of GNC algorithms at the ISR Flying Arena.

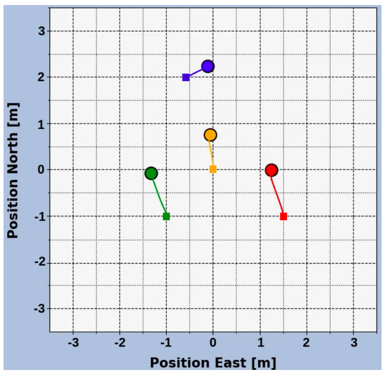


Figure 6.24: Formation movement at $t = 10s$.

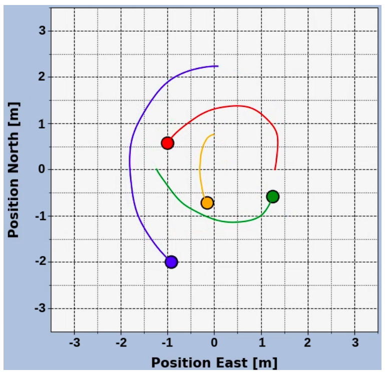


Figure 6.25: Formation movement at $t = 20s$.

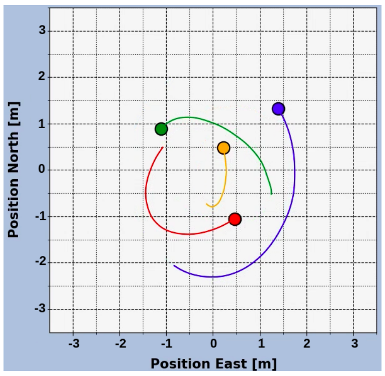


Figure 6.26: Formation movement at $t = 30s$.

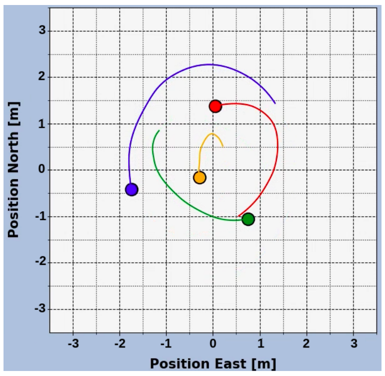


Figure 6.27: Formation movement at $t = 40s$.

These results match the ones obtained in the original research [40] and prove that it is possible to successfully implement and validate formation control algorithms using the framework designed in the scope of this thesis. These experiments also prove that the software programs developed to emulate the relative position and velocity sensors work as designed, enabling researchers to use them as templates for the emulation of other sensors.

In summary, the tests performed with the time-varying formation reinforced that the architecture adopted for the ISR Flying Arena is robust, scalable, and flexible, allowing experiments with real and simulated quadcopter interaction. Moreover, these experiments tested every module, program, tool and configuration file developed, which, in addition to the experiments successfully carried out in the previous sections, confirm that the created software can be safely employed by other researchers to rapidly deploy and test GNC solutions.

6.5 Supporting other researchers

Once the testing stage was completed and the ISR Flying Arena prototyping framework was deemed robust enough, it was successfully applied in the validation process of GNC solutions by other students, in the course of their theses works. This is solid evidence that the main objective of this work (design and implement a multi-vehicle rapid prototyping platform for development and testing of GNC solutions) was successfully achieved.

6.5.1 Validating a navigation system

The first research work whose testing and validation process was facilitated by the created setup consisted of a navigation system based on distance measurements [41]. In order to prove, in an experimental environment, that the navigation algorithm converged, it was necessary to track a specific trajectory previously designed by the researcher with an acoustic transponder attached to the vehicle. It was also necessary to generate detailed flight logs with the measurements provided by the Optitrack system, by the EKF of the PX4, and by the inertial sensors.

By combining the testing setup, the software modules, and the tools developed in this work, it was possible to automate all low-level tasks required to carry out experiments in the ISR Flying Arena, and, within just two hours, the researcher was able to: i) attach the acoustic sensor to the drone; ii) create a landing site to smooth the landing of the vehicle and thus not damaging the sensor; iii) calibrate the Optitrack system; iv) create a user program for tracking the desired trajectory; v) simulate the trajectory tracking; and vi) conduct several experiments at the ISR Flying Arena with a physical vehicle. The flight logs were automatically generated by the offboard logger tool. Without the support of the framework developed in this thesis, the validation of the navigation algorithm in an experimental environment would be a time consuming effort.

6.5.2 Validating control algorithms

The second thesis research assisted by the prototyping framework was related to predictive control strategies for aggressive parcel relay maneuvers using drones [42]. By using the systems and software programs created, configured, and described in Chapter 3, that prepare the Optitrack system and the physical vehicles of the ISR Flying Arena for offboard experiments, and by using the complete software environment presented in Chapter 4, the researcher was able to immediately conduct simulations and tests with physical drones in the ISR Flying Arena using his own computer. Since the student had previously developed the user programs with the control algorithms to be tested, he decided not to use the software solutions presented in Chapter 5. Nevertheless, the system and programs employed still spared him several weeks of work. This particular situation shows the importance of the modular architecture adopted for the Arena, which gives users the freedom to only employ the modules that they are interested in.

Chapter 7

Conclusions

This chapter concludes the thesis by presenting the main results achieved, as well as the envisioned steps for future work.

The goal of this dissertation was to design and implement a multi-vehicle rapid prototyping platform for development and testing of guidance, navigation, and control solutions. With that in mind, a modular architecture was devised and tailored for the tests setup that comprised an Optitrack system providing indoor position and attitude ground-truth, a set of ground computers for running the GNC solutions, and multiple quadcopters equipped with PX4 autopilots. The devised testbed addressed scalability and featured a simulation environment for testing the deployed algorithms before experiments with physical vehicles.

Once the architecture of the tests platform was defined, software programs that routed the pose of the vehicles from the Optitrack system to the PX4 autopilots and to the ground computers were successfully developed. Then, the extended Kalman filter of the PX4 was configured to fuse this data with the measurements provided by the IMU, the failsafe modes of the vehicle were set up, and the internal controllers of the PX4 were tuned. After this stage, the physical vehicles were ready to fly.

Subsequently, a fully configurable simulation environment was created using the Gazebo software. In order to produce a simulation setup as similar as possible to the physical Arena, a software solution that emulates the Optitrack system was programmed. One of the most significant contributions of this thesis was the development of a single configuration file that launches the simulation with all the settings desired by the user for the simulated world, for the vehicles (including their dynamics, motors properties and onboard sensors) and for the PX4 autopilot. This configuration file spares significant time to users whenever they need to edit the simulation.

Afterwards, a set of software modules were programmed, using an object-oriented approach and four different communication libraries, to automate all the low-level tasks related to the communication between user programs and the PX4 autopilots. The multiple communication solutions grant flexibility to the tests platform because some modules employ the ROS middleware, which offers valuable tools for robotics applications, while others are lightweight, which enables them to run in small onboard computers with limited resources. An offboard logger program, emulated sensors, and a set of visualization and monitoring tools were also created at this point to enhance the capabilities of the prototyping platform.

Finally, both the real and the simulation environments were validated by performing over 50 different tests, where several control solutions, including a formation-control algorithm, were successfully deployed. These tests demonstrated that the architecture of the ISR Flying Arena is robust, successfully addresses scalability, and enables experiments simultaneously involving physical and simulated vehicles, overcoming space and equipment limitations. The created setup was also used in the experimental tests and validation process performed within the scope of the MSc Theses of two other students. The developed tests platform allowed for the validation of a navigation system in experimental environment in only two hours, sparing several weeks of work to the researcher. Ultimately, it was proved that the created prototyping environment successfully met the goals proposed for this project by enabling the rapid deployment and test of GNC solutions.

7.1 Future Work

In order to improve the ISR Flying Arena the following future steps are envisioned:

- Incorporate micro UAVs in the tests platform to overcome the small dimensions of the physical space. Given the size of the vehicles currently available, it is not safe to simultaneously use more than two or three physical drones in the experiments. Although it is possible to replace physical vehicles with simulated ones, as demonstrated in the formation-control tests, the simulation environment is computationally heavy and prone to delays.
- Create new software modules according to the FastRTPS (Fast Real Time Publish Subscribe) communication bridge, which enables high-performance exchange of uORB messages with offboard systems and offers a direct reliable interface with ROS2. The FastRTPS bridge was not considered for this work because, similarly to ROS2, it is in the preliminary stages of development. Due to the modular architecture adopted for the ISR Flying Arena, the future integration of these new communication modules will be straightforward.
- Implement the GNC algorithms directly in the PX4 Autopilot, for reduced latency. New software programs could be created for automatically subscribing to the EKF and to the sensors data, according to the low-level uORB messaging protocol used by the PX4 autopilot for communication between internal processes.

Bibliography

[1] R. Austin. *Unmanned Aircraft Systems: UAVS Design, Development and Deployment*. Wiley, 2010.

[2] Markets and Markets. Unmanned Aerial Vehicle (UAV) Market, Oct 2019. URL `https://www.marketsandmarkets.com/Market-Reports/unmanned-aerial-vehicles-uav-market-662.html`. Accessed: July 2020.

[3] F. L. Mueller, A. P. Schoellig, and R. D'Andrea. Trajectory generation and control for precise aggressive maneuvers with quadrotors. *The International Journal of Robotics Research*, Jan 2012.

[4] D. Mellinger, N. Michael, and V. Kumar. Iterative learning of feed-forward corrections for high-performance tracking. *International Conference on Intelligent Robots and Systems*, Oct 2012.

[5] R. Ritz, M. W. Müller, M. Hehn, and R. D'Andrea. Cooperative quadrocopter ball throwing and catching. *International Conference on Intelligent Robots and Systems*, Oct 2012.

[6] M. Schmittle, A. Lukina, L. Vacek, J. Das, C. P. Buskirk, S. Rees, J. Sztipanovits, R. Grosu, and V. Kumar. OpenUAV: A UAV Testbed for the CPS and Robotics Community. *International Conference on Cyber-Physical Systems*, Apr 2018.

[7] J. P. How, J. Teo, and B. Michini. Adaptive Flight Control Experiments using RAVEN. *Yale Workshop on Adaptive and Learning Systems*, 2008.

[8] J. P. How, B. Behihke, A. Frank, D. Dale, and J. Vian. Real-time indoor autonomous vehicle test environment. *IEEE Control Systems Magazine*, Mar 2008.

[9] S. Lupashin, M. Hehn, M. W. Mueller, A. P. Schoellig, M. Sherback, and R. D'Andrea. A platform for aerial robotics research and demonstration: The Flying Machine Arena. *Mechatronics Journal, Volume 24, Issue 1*, Feb 2014.

[10] Imperial College London, Aerial Robotics Lab Facilities. URL `https://www.imperial.ac.uk/aerial-robotics/facilities/`. Accessed: July 2020.

[11] Imperial College London, Engineering News, Oct 2017. URL `http://www.imperial.ac.uk/news/182375/uk-world-leader-drone-tech-says/`. Accessed: July 2020.

[12] ISR Flying Arena - Digital Repository. URL `https://tiagoalexnd@bitbucket.org/dsorglobal/tiagooliveira.git`.

[13] PX4 Software Overview. URL https://px4.io/software/software-overview/. Accessed: July 2020.

[14] PX4 Autopilot User Guide. URL https://docs.px4.io/master/en/. Accessed: July 2020.

[15] QGroundControl User Guide. URL http://qgroundcontrol.com/. Accessed: July 2020.

[16] MavLink Router. URL https://github.com/mavlink-router/mavlink-router. Accessed: August 2020.

[17] Specifications for the Intel Aero Ready to Fly Drone. URL https://www.intel.com/content/www/us/en/support/articles/000023272/drones/development-drones.html. Accessed: August 2020.

[18] Basic PX4 Configuration. URL https://docs.px4.io/v1.9.0/en/config/. Accessed: August 2020.

[19] Gazebo: Robot simulation made easy. URL http://gazebosim.org/. Accessed: August 2020.

[20] MAVLink API: Simulator Messages. URL https://mavlink.io/en/messages/common.html#SIM_STATE. Accessed: August 2020.

[21] PX4 simulator.cpp file. URL https://github.com/PX4/Firmware/blob/master/src/modules/simulator/simulator_mavlink.cpp. Accessed: August 2020.

[22] Gazebo: ROS Communication. URL http://gazebosim.org/tutorials/?tut=ros_comm#Prerequisites. Accessed: August 2020.

[23] Gazebo ROS Package: empty_world.launch file. URL https://github.com/ros-simulation/gazebo_ros_pkgs/blob/jade-devel/gazebo_ros/launch/empty_world.launch. Accessed: August 2020.

[24] PX4 simulation launchers. URL https://github.com/PX4/Firmware/tree/master/launch. Accessed: August 2020.

[25] 3DR Iris quadcopter. URL http://www.arducopter.co.uk/iris-quadcopter-uav.html. Accessed: August 2020.

[26] Rospy, the Python client for ROS. URL http://wiki.ros.org/rospy. Accessed: August 2020.

[27] MAVLink Wrapper/Developer APIs. URL https://mavlink.io/en/about/implementations.html#mavlink-wrapperdeveloper-apis. Accessed: September 2020.

[28] MAVROS Package Summary. URL http://wiki.ros.org/mavros. Accessed: September 2020.

[29] MAVSDK C++ Library. URL https://mavsdk.mavlink.io/develop/en/cpp/. Accessed: September 2020.

[30] Python wrapper for MAVSDK. URL https://github.com/mavlink/MAVSDK-Python. Accessed: September 2020.

[31] PoseStamped message. URL http://docs.ros.org/melodic/api/geometry_msgs/html/msg/PoseStamped.html. Accessed: September 2020.

[32] MAVROS offboard control example. URL https://dev.px4.io/master/en/ros/mavros_offboard.html. Accessed: September 2020.

[33] Example: Takeoff and Land. URL https://mavsdk.mavlink.io/develop/en/examples/takeoff_and_land.html#source_code. Accessed: September 2020.

[34] PX4.launch file. URL https://github.com/mavlink/mavros/blob/master/mavros/launch/px4.launch. Accessed: September 2020.

[35] MAVSDK C++ connection function. URL https://mavsdk.mavlink.io/develop/en/api_reference/classmavsdk_1_1_mavsdk.html#classmavsdk_1_1_mavsdk_1a229888e2931c16d11edbed07b03174d4. Accessed: September 2020.

[36] MAVSDK Python: System class. URL https://github.com/mavlink/MAVSDK-Python/blob/master/mavsdk/system.py. Accessed: September 2020.

[37] Flight Log Analysis. URL https://dev.px4.io/master/en/log/flight_log_analysis.html. Accessed: September 2020.

[38] Log Analysis Tools. URL https://docs.px4.io/master/en/log/flight_log_analysis.html#analysis-tools. Accessed: September 2020.

[39] A. Borkar, A. Sinha, L. Vachhani, and H. Arya. Application of Lissajous curves in trajectory planning of multiple agents. *Autonomous Robots*, 2019.

[40] P. Trindade, R. Cunha, and P. Batista. Distributed Formation Control of Double-Integrator Vehicles with Disturbance Rejection. *Proceedings of the 21st IFAC World Congress*, July 2020.

[41] J. Franco. Sistema de navegação baseado em medidas de distância. Master's thesis, Instituto Superior Técnico da Universidade de Lisboa, Jan 2021.

[42] J. Pinto. Model predictive control strategies for aggressive parcel relay maneuvers using drones. Master's thesis, Instituto Superior Técnico da Universidade de Lisboa, 2021.