POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

Programming Environment for Human Drone Interaction: EasyFly

Tesi di Laurea Magistrale in
Computer Science Engineering

Author: **Matteo Plona**

Student ID: 952967
Advisor: Prof. Luca Mottola
Academic Year: 2022-23

Abstract

TODO

Keywords: here, the keywords, of your thesis

Abstract in lingua italiana

TODO

Parole chiave: qui, vanno, le parole chiave, della tesi

Contents

1 | Introduction

In recent years, the rapid advancement of unmanned aerial vehicles, commonly known as drones, has revolutionized various industries and opened up new possibilities for applications ranging from aerial photography and package delivery to search and rescue operations. As drones become increasingly integrated into our daily lives, it is crucial to explore and understand the dynamics of their interaction with humans.

In modern drone applications, the human figure is marginal with respect to the drone. The former usually plays the supervisor role, while the latter performs almost all requested tasks automatically. This significant discrepancy undoubtedly leads to a decrease in interactions between the two.

This thesis will describe EasyFly, an accessible and high-level programming environment for drone applications. The purpose is to provide a programming environment to allow users with different levels of expertise to experiment with drones. Differently from modern applications, our environment allows humans and drones to work closely together, making EasyFly a perfect tool for conducting research in the field of human-drone interactions.

1.1. The Problem: Programming Human-Drone Interactions

Human-Drone Interaction (HDI) is a branch of the more general field of Human-Robot Interaction (HRI), and it can be defined as 'the study field focused on understanding, designing, and evaluating drone systems for use by or with human user [43]'. While the field of HRI offers valuable insights, the distinctive ability of drones to move freely in three-dimensional space, along with their unique shapes, sets HDI as a distinct and independent area of research.

The rapid technological progress in this field has made drones increasingly efficient and autonomous in performing various tasks. While on the one hand, these advancements enable the integration of drones into everyday life, streamlining processes and reducing

the time required for specific activities, on the other hand, modern drone applications do not represent the ideal prototype for conducting research in the field of HDI.

The first limitation of modern drone applications in this discipline's study is that these applications are designed to operate in large environments with minimal human presence. An example can be represented by autonomous delivery drones [41] or 3D mapping applications [36] where the interactions with the human are purposely reduced to the bare minimum.

In these types of applications, interactions between humans and drones are often limited to simple tasks, such as package delivery or crop monitoring. These interactions are often repetitive and lack the diversity and complexity required in research on HDI. Since tasks and interactions are repetitive, users are usually trained to interact with the drone in a specific way. This training can reduce the variability in HDI, making it less suitable for research purposes.

Programming HDI is usually the field's most complex and expensive research phase. It usually requires a specialized team of researchers who can program a custom drone application that addresses the complexity of interactions required. Moreover, the implementation phase is usually the bottleneck of the entire process; every small change to the interaction model can result in days or weeks for implementing the desired behavior.

One of the most significant challenges during the programming of HDI is the testing phase. Modern drones are usually fragile and expensive, while tests are likely to fail. This phase usually introduces a high consumption of resources, both in terms of costs and time needed for repairing the entire setup before another attempt.

1.2. The Solution: EasyFly

To overcome all the issues related to the programming of HDI, we introduce EasyFly, a programming environment for drone applications that address all the research needs in the field of HDI.

To better understand the contribution of EasyFly to this research field, let us take a step back and describe the needs of researchers.

The ideal prototype of a programming environment for researchers studying HDI should address and solve all the problems related to developing drone applications used for research. In particular, this prototype should ultimately reduce the time and costs associated with the development phase and increase the research's effectiveness.

The first characteristic of the ideal prototype is to reduce the level of expertise needed to develop the desired drone application. This feature allows researchers to implement all the required functionality easily without requiring a specialized team of drone programmers. Moreover, this feature would open the doors to a brand-new type of research where the users are questioned to interact with the drone programmed by themselves. EasyFly provides this feature by offering a set of simple operations, which indeed allows the creation of very complex behaviors. In addition to this, EasyFly allows programming in a descriptive fashion; in this way, programs would be self-explaining and easily interpreted by anyone.

As in any other field, research on HDI should be dynamic. In other words, to gather all the possible insights from an interaction, the drone application must rapidly change and adapt to the situation. If the application development cycle is too long, there is the risk of losing many possible opportunities to experiment with possible alternative solutions. The ideal prototype should provide the maximum flexibility in adapting to many possible situations. For this reason, EasyFly has adopted a modular approach for both the hardware and the software components. At any moment, a module (either software or hardware) can be attached or detached to compose the best configuration needed at that specific moment.

Last but not least, facilitating the interaction between the human and the drone should be the primary goal. The ideal prototype should be the first promoter of the interaction. It should offer the best possible condition to allow the two entities to establish an interaction safely and free from any potential bias determined by the programming environment.

To implement EasyFly, we have targeted a specific typology of unmanned aerial vehicles: nano-drones. As the name suggests, nano-drones are simply drones with very small size and weight. Their small size makes them the best choice to facilitate human-drone interaction.

In the first place, nano-drones are less intimidating and intrusive than larger drones, making it easier for researchers to observe how individuals react and interact with them. They also allow minimal disturbance in the observation environment, making them perfect for avoiding any possible noise in the experiment. Given that the drone and the human are supposed to work closely together, any possible malfunction can cause an unexpected drone crash, especially while experimenting with new solutions. It is easy to deduce that the smaller the drone is, the safer the interaction. The last observation is that nano-drones are usually less expensive than bigger ones, allowing researchers to experiment with interactions with multiple drones without affecting their budget.

1.3. The Benchmark: Drone Arena Challenge

In the HDI domain, the research's core part, especially from the computer science perspective, is the experimental phase. During this phase, researchers put their ideas and prototypes to test and assess the practicality of innovative interaction models.

For a programming environment like EasyFly, testing and evaluating in a real research scenario in HDI is essential. The testing in real scenarios can help detect possible weaknesses in the programming environment, allowing for fine-tuning the model.

To best evaluate our EasyFly programming environment, we had the possibility to participate in the Digital Futures Drone Arena project [9]. This project allowed us to perform an in-depth analysis of the impact of using EasyFly while developing human-drone interactions.

Drone Arena is an interdisciplinary research project that aims to create a technological and conceptual platform for interdisciplinary investigations of drones at the intersection of mobile robotics, autonomous systems, machine learning, and Human-Computer Interaction. The project has three inter-related objectives:

1. The constructions of a novel aerial drone testbed that is geared towards application-level functionality rather than low-level control mechanisms.
2. The organization of two challenges where multiple teams are involved and tasked to realize functionality that pushes the state of the art.
3. The conduction of empirical investigation of Human Drone Interactions in the Drone Arena. This includes observation, interviews, and micro-sociological video analysis to inform future competitions in the drone arena and to develop insights from the movement-based explorations of drone piloting.

In these settings, our EasyFly programming environment is focused on the first two objectives of the project. In particular, our programming environment was one of the core parts of the novel aerial drone testbed used for the entire duration of the project. For the second objective of the project, we had the possibility to actively participate in the first of the two challenges organized for the Drone Arena project. During this challenge, we conducted a complete and in-depth evaluation of the impact of using EasyFly; in particular, we compared our programming environment with a simpler and lower-level one.

1.4. State of the Art

The field of HDI is an active and evolving area of research with a focus on improving the ways in which humans and drones interact. It is a multidisciplinary field with two main research areas: the technological and the sociological. Each area focuses on distinct aspects of the interaction between humans and drones.

In the technological area, at the intersection of computer science, mobile robotics, autonomous systems, and machine learning, the key focus is developing and improving the hardware and software components of drones and their interfaces. The main goal of this area is to enhance the capabilities and functionalities of drones to make them more user-friendly and efficient.

In the sociological area, which includes disciplines like social engineering, art, ethics, and political science, the core objective is to understand how the presence and use of drones impact society, individuals, and communities.

The most common drone model used in research in the HDI field is the Parrot ARDrone [43]. Parrot drones provide an easy-to-use software API allowing for quick prototyping, which is likely the reason they are the researcher's first choice. Although the Parrot ARDrone is widely used for research, this model was discontinued by the manufacturer.

Especially in research focused on the sociological area, where researchers usually have less familiarity with programming tools, the prototyping phase is the most complex and time-consuming. EasyFly tries to overcome all the issues related to this phase by creating a simple and flexible programming environment for drone applications. Moreover, it tries to offer a new perspective in the investigation of human-drone interactions where the user plays the role of the programmer.

1.5. Overview

TODO at the end

2 | State of The Art

In this chapter, we will analyze the literature and background works: we will first examine the more general field of Human-Computer Interaction (HCI), passing then to Human-Robot Interaction (HRI) and, finally, Human-Drone Interaction (HDI), what are the purposes, the achievement, and the limitations of this research. Next, we will move to the programming environments for both single drones and swarms of multiple drones, understanding which are the main design paradigms and solutions available.

2.1. Human-Robot Interaction

We live in an era where humans and computers are increasingly in close contact. In the last 40 years, the continuous and exponential growth in computational power and storage capacity, but even more relevant, the huge spread of technologies in every aspect of our lives, has moved the concept of Human-Computer interaction to a central stage of research. With the massive amount of new designs of technologies and systems, this relation grows exponentially in complexity, and the importance of profoundly understanding it is a key aspect of bringing innovation to the next step and, more importantly, avoiding the risk of incurring harmful and undesired situations.

Human-Computer Interaction (HCI) is a discipline concerned with the design, evaluation, and implementation of interactive computing systems for human use and with the study of major phenomena surrounding them [42]. The role of this discipline is to understand in depth the relation between humans and computers, providing the guidelines that allow us to build better user interfaces. On the other side, HCI must also understand the limits and the risks associated with those new interfaces, which should guide governments around the world to regulate and control the expansion of such technologies in a sane manner but avoid limiting their potentiality.

Regarding our work, it is advisable to narrow down the concept of HCI to a subfield of research: Human-Robot Interaction (HRI). Actually, HRI is not only a subfield of the more general HCI; instead, it is a multidisciplinary field with the influence of HCI,

Artificial Intelligence, Natural language understanding, and social science. The primary goal of HRI is to define a general human model that could lead to principles and algorithms allowing more natural and effective interactions between humans and robots [28].

2.1.1. The Interaction Between Human and Robot

The Human is an extremely sophisticated biological system characterized by an impressive, complex, and powerful brain that is the main coordinator and central actor in the whole human system. Given this complexity, for our purpose, we can model it using the Model Human Processor (MPH) proposed by Card, Moran, and Newell in 1983 [23].

MPH describes the Human as composed of 3 subsystems: the perceptual, motor, and cognitive systems. Each of them has a processor and memory. Input in humans occurs mainly through the senses and output through the motor controls of the effectors[26]. Therefore, vision, hearing, and touch are the most important senses in HRI, while fingers, voice, eyes, head and body position are the primary effectors.

On the other hand, the computer is a straightforward machine that processes the input data that it receives into outputs. The processor is the leading actor in the data transformation process: it can perform arithmetic and logic operations very fast. Both input and output data for the computer are encoded binary sequences.

The Interaction, with or without a robot, is a process of information transfer between two (or multiple) systems. The outputs of one system are the inputs of the other and vice versa. In the interaction, it is immediately evident that the human outputs are incompatible with the robot's input. Also, the outputs of a robot are not easily interpretable by a human. This profound discrepancy between the two is the domain of study for HRI; the user outputs need to be translated from body movements and voice to binary sequences, and, on the other way, the robot outputs need to be translated from binary data to images, text, and sounds.

Initially, with computers becoming accessible to the public fifty years ago, command-line interaction was the only option for interacting with a computer. Today, interfaces have evolved into sophisticated forms such as advanced GUI, touchscreens, Augmented Reality, Virtual Reality, and Haptic interfaces. All these new interfaces make one side easier the lives of humans, but on the other side, they have inevitably brought values and ethics in technology design to the forefront of public debate: questions about the goals and politics of human-designed devices, and whether the social interactions of those devices are good, fair, or just [40].

2.1.2. HRI Taxonomy

HRI is a highly heterogeneous field, and understanding the details of interactions between humans and robots is crucial for advancing this dynamic field. To define and better identify the diverse range of interactions between humans and robots, it is important to organize and classify the interactions in a structured taxonomy [44].

Following this taxonomy, HRI can be classified using the following attributes:

Task Type:
It is a high-level description of the task to be accomplished. It sets the tone for the system's design and use. Moreover, it implicitly represents the environment for the robot.

Task Criticality:
It measures the importance of getting the task done correctly in terms of its negative effects should problems occur. To mitigate the subjective nature of this attribute, it can take three values: high, medium, and low.

Robot Morphology:
It can assume three values: anthropomorphic (having a human-like appearance), zoomorphic (having an animal-like appearance), and functional (having an appearance that is neither human-like nor animal-like but is related to the robot's function). It is an essential measure since people react differently based on their appearance.

Interaction Roles:
When humans interact with a robot, they can act in 5 different roles: supervisor (monitor the behavior of a robot), operator (control and modify the behavior of the robot), teammate (works in cooperation together to accomplish a task), mechanic/programmer (physically change the robot's hardware or software), and bystander (needs to understand the robot's behavior to be in the same space).

Type of Human-Robot Physical Proximity:
When interacting with the robot, a human can act at different levels of physical proximity: none, avoiding, passing, following, approaching, and touching.

Decision Support for the Human:
It represents the type of information that is provided to operators for decision support. This taxonomy category has four subcategories: available sensor information, sensor information provided, type of sensor fusion, and pre-processing.

Time and Space:
Divides Human-Robot interaction into four categories based on whether the humans and

robots interact at the same time (synchronous) or different times (asynchronous) and while in the same place (collocated) or different places (non-collocated).

Autonomy Level and Amount of Intervention:
The autonomy level measures the percentage of time that the robot is carrying out its task on its own; the amount of intervention required measures the percentage of time that a human operator must be controlling the robot.

In Table 2.1 is presented the type of interaction that we are targeting in this thesis work using the taxonomy described above:

Attribute	Values
Task Type	Arbitrary interaction with nano drones in a small indoor environment to investigate Human-Drone relation
Task Criticality	Low
Robot Morphology	Functional
Interaction Roles	Mechanic/Programmer or Operator
Physical Proximity	Any value
Decision Support	Available sensors: [proximity (x, y, z), localization (x, y, z), flow (vx, vy), video]
Time and Space	Synchronous and Collocated
Autonomy Level	Any value
Amount of Intervention	Any value

Table 2.1: Categorization of interaction for target applications in our thesis work following the taxonomy proposed by Yanco and Drury [44]

2.2. Human-Drone Interaction

Drones, also known as unmanned aerial vehicles (UAVs), are robots capable of flying autonomously or through different control modalities. Until the early 2000s, drones were complex systems commonly seen in the military world and out of reach for civilians. Modern advancements in hardware and software technologies allow the development of smaller, easier-to-control, and lower-cost systems. Drones are now found performing a broad range of civilian activities, and their usage is expected to keep increasing in the near future. As drone usage increases, humans will interact with such systems more often; therefore, achieving a natural human-drone interaction is crucial.

Human-Drone Interaction (HDI) can be defined as the study field focused on understanding, designing, and evaluating drone systems for use by or with human users [43]. Although some knowledge can be derived from the field of HRI, drones can fly in 3D space, which essentially changes how humans interact with them, making human-drone interaction a field of its own. This field is relatively new in the research community, but in the last few years, the number of publications about HDI has grown exponentially.

2.2.1. The Role of the Human During the Interaction

One of the core topics in the field of HDI is the role of humans during interaction with drones. Depending on the drone's application and its level of autonomy, humans can play different roles when interacting with drone systems.

When the user pilots the drone to accomplish a given task by directly controlling the drone through a control interface, the user is considered an *active controller* of the interaction. In these settings, the user's role is crucial to complete the given task; the drone instead acts as a mere executor of instructions. Examples of this type of interaction are waypoint navigation [31] or artistic exhibitions [27].

The user acts as a *recipient* when he/she does not control the drone, but he/she benefits from interacting with it. An example of this type of interaction are represented by delivery droneshoppe2019droneOSz [18, 41].

Another type of interaction role is when the drone acts as a *social companion* for the user. In this case, the user might or might not be able to control the drone movement, but it holds a social interaction with it. An example of this type of interaction is represented by Joggobot [29], a drone used as a companion for jogging.

The last type of role the user can act when interacting with a drone is the role of *supervisor*. Autonomous drones require users to act as supervisors either to pre-program the drone behavior or to supervise the flight itself in case of emergency. In this case examples can be crop monitoring [25] or aerial photogrammetry [35],

2.2.2. The drone's control modality

Usually, drones expose a control interface that allows users to control their behavior and eventually complete some tasks in the application domain. Each control interface impacts how the pilot interacts with the drone in various aspects, such as training period, accuracy, latency, and interaction distance.

As drones became available to the public, the major drone producers felt the need to change their control modality from the standard remote controller to a more natural and easy-to-use interface. A wide variety of control interfaces are available on the market today, ranging from standard remote controllers to very complex and advanced Brain Controlled Drones [32].

Drone's control interfaces can be classified as follows [43]:

Remote Controller is the standard and most commonly used interface, where the user directly controls the drone movements. This control modality provides low latency and precise control, but on the other hand, it is less intuitive and usable than natural user interfaces. The usability and easiness of this interface strongly depend on the drone's level of autonomy.

Gesture-based interface is a control modality where the user pilots the drone with body movements. Usually, the drone uses a camera or a Kinect device to extract spatial information and recognize postures. When users are asked to interact with a drone without any instruction, gesture interaction is the primary choice of most users, and this indicates that the training period of this interface is almost close to zero [24]. Compared to other control modalities, gesture-based has a high latency and lower control precision. The flight space for drones that use this control method is sensibly reduced since the pilot needs to be close to the drone during the flight.

Speech-based interface is a control modality where the user pilots the drone using vocal commands. As for gesture-based, this interface is also a natural user interface with a low training period and high usability. They also share the problems of user proximity and high latency of commands.

Touch-based interface is a control modality where the user is requested to control the drone using his hands. The drone usually carries proximity sensors that allow it to receive inputs from the user. It is a natural user interface, and, as the others, it has the same pros and cons.

Brain Computer interfaces allows the user to pilot the drone using brain signals [32]. To enable this type of interface, the pilot must wear some form of BCI headset, the most common being Electroencephalography (EEG) headsets. These devices measure the brain's electrical activity on a human's scalp, which is decoded using machine learning algorithms to control physical systems using brain waves. Compared to the others, it is the most complex control interface and has the highest accessibility for users with disabilities. The problems in using BCI are the poor control quality and higher training period. Further

research in this field will probably lead to more usable and better interfaces.

Interactions can also be combined into *multimodal interfaces*. Integrating different interaction methods can combine the advantages of each; however, it can increase complexity and costs a lot.

2.2.3. Values and ethics in HDI

Drones usually carry cameras, and potentially, they can fly wherever they want; this introduces a lot of issues of privacy [19]. Given the rapid expansion of such technology, governments worldwide have been caught off guard in recent years. Governments tried to quickly create rules and regulations to control the usage of such technology. This rapid regulation has, in most cases, limited drones too much, slowed their expansion, and reduced their potential. Research in ethics and values about HDI is responsible for producing accurate and reliable results that should guide governments in refining and upgrading drone laws.

2.3. Programming Environments

When developing drone applications, the choice of programming environment is crucial to ensure efficient and reliable software. The programming environment is intended as all the resources, hardware, and software used to accomplish the tasks of the application scenario being developed.

As drone technology expanded, many companies working in the drone field started developing and selling their programming environment. Nowadays, most software resources for drone applications are open-source and publicly available. We will dive into the scenario of the drone programming environment, understanding the most common and popular solutions for developing a drone application. We will then analyze the more complex scenario of swarm applications.

2.3.1. Single Drone Programming

Exploring the programming landscape for a single drone involves navigating through specialized tools and frameworks specialized for the development and control of individual drones. Whether managing a custom-built drone or utilizing a commercial off-the-shelf (COTS) model, creating an effective combination of hardware and software is crucial. This section will guide you through essential components and considerations for developing software that governs a drone's flight and functionality.

The development of any drone application can be categorized into two main areas: on-board and off-board the drone.

The on-board area comprises all the hardware and the software that composes the drone itself. As we can see in Figure 2.1, usually the hardware of the drone includes: The chassis of the drone, the motors and propellers, the battery and the power distribution unit, the computing unit and the memory unit, the communication unit and the sensors unit.

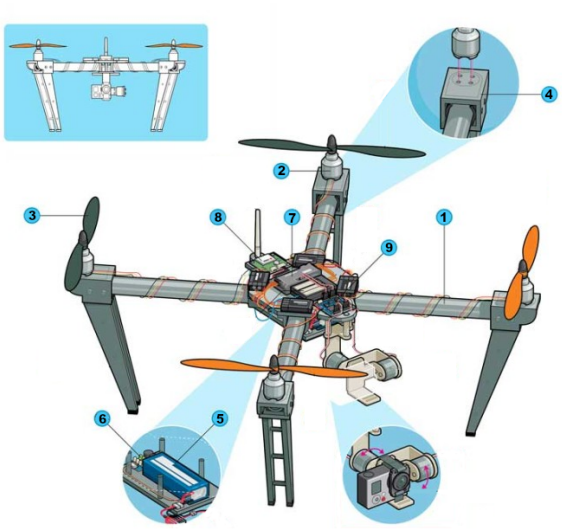


Figure 2.1: The main hardware components of a drone are: 1. Drone's chassis, 2. Motors, 3. Propellers, 4. Motor mount, 5. Battery, 6. Power distribution unit, 7. Computing and memory unit, 8. Communication unit, 9. Sensors unit

The software that runs on the drone, also known as the autopilot software, is usually composed of four main components: the communication unit, the sensing unit, the core control loop unit, and the low-level control unit. In Figure 2.2, we can see how the software components cooperate together to achieve a controllable and stable flight: The Communication Unit receives and decodes commands from the off-board system; the signal is then transformed into power set-points from the Core Unit (control loop) with the help of sensor information. The Sensing Unit gathers information from the environment using sensors, translate sensor readings into readable values, then send this information to the Core Unit and to the Communication Unit to send back telemetry data to off-board systems.

The current landscape of on-board drone solutions ranges from commercial off-the-shelf (COTS) to entirely custom solutions. Commercial producers of drones like Parrot [13], DJI [8], and 3DR [1] usually sell COTS solutions where all the hardware resources are supplied with the software needed to run the drone. Depending on the application, these bundled solutions may not be enough; if this is the case, the developer then needs to manually select each hardware component, control the compatibility with each other, and

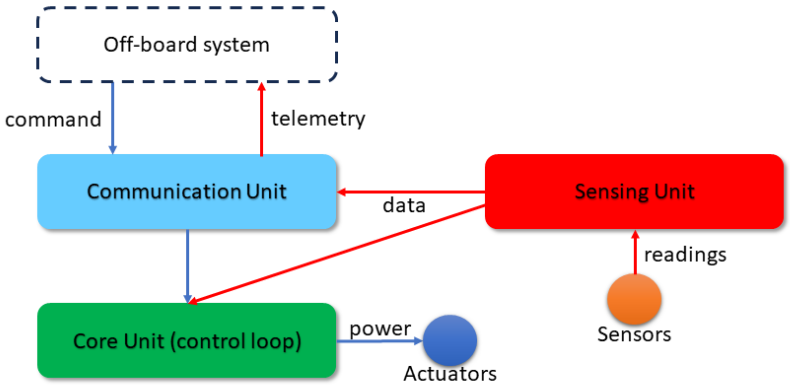


Figure 2.2: The main software components of a drone are: the *Sensing Unit*, the *Communication Unit* and the *Core Unit (control loop)*.

then select (or develop) the software that allows the drone to fly.

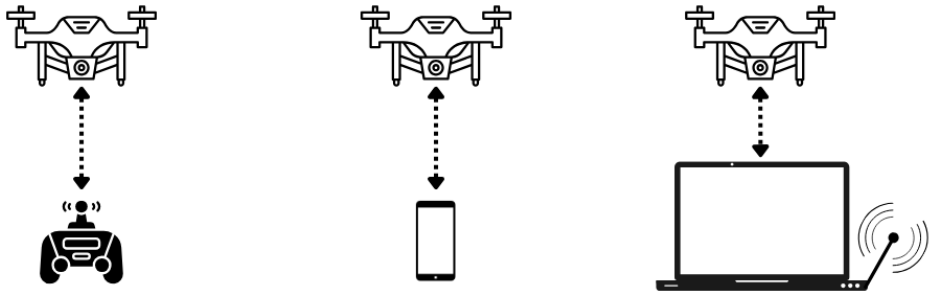
Some producer sell also intermediate solutions [7, 12, 14, 15] between COTS and the completely custom one, these solutions are composed of a microcontroller with usually the basic sensors that compose the Inertial Measurement Unit (IMU) and the autopilot software. These solutions are then extensible with other custom hardware; they provide programming tools to program the behavior of the drone during the flight.

Regarding our setup, the on-board system that we used is a nano drone named Crazyflie 2.1, produced by Bitcraze; it is a COTS solution but with a lot of space for customization for both hardware and software components.

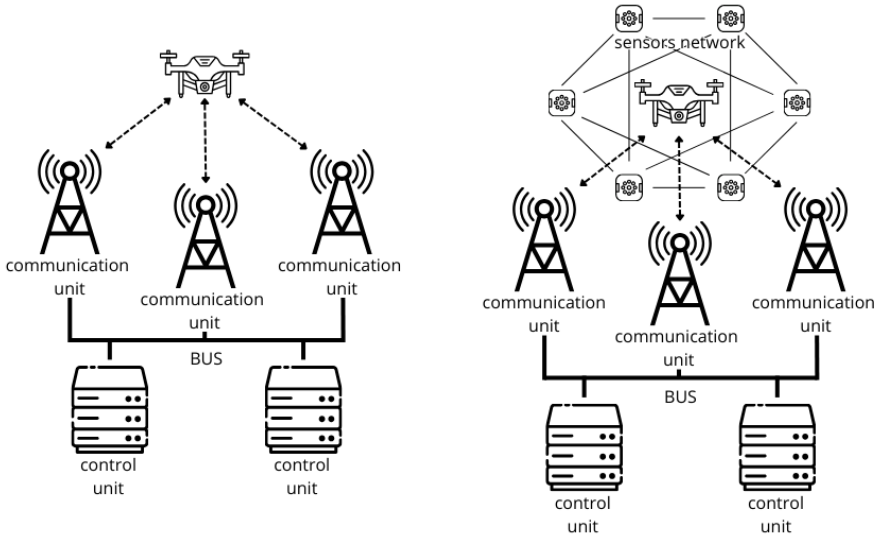
Off-board the drone, the environment is strongly related to the application scenario, and, in particular, it depends on the level of autonomy request for the drone, the flight area dimension, and the complexity of the operation.

Despite the heterogeneity of off-board systems, we can consistently identify two main components in most scenarios: a control unit and a communication unit. In most common situations, control and communication units are hosted on a single device. Example of these devices are remote controls (Figure 2.3a), smartphones (Figure 2.3a) or a computer that acts as base (ground) station 2.3c. When the scenario is more complex and the fight area is very broad, we can have a distributed ground network of control and communication units (Figure 2.3d). Additionally, in combination with a distributed ground network, it can also be deployed a sensor network to gather more information of the drone in its environment(Figure 2.3e). For example, the sensor network can be composed of sensors that collect atmospheric data, allowing for a better knowledge of the environment in which the drone is deployed.

Communication technology and infrastructure are critical topics that can introduce poten-



(a) Remote control (b) Smartphone (c) Base Station



(d) Distributed ground network (e) Distributed with sensor network

Figure 2.3: Off-board ecosystem

tial issues in the off-board environment when it is inadequate for the application scenario. In particular, depending on the flight area's dimension, location, and topography, appropriate technology and communication infrastructure must be deployed to have a properly working drone.

As highlighted in Figure 2.3, the communication infrastructure can be single or distributed. The former is more straightforward to implement and deploy but can be not enough when the flight area is too broad or the topography is irregular; the latter is much more complex but allows for covering all the possible application scenarios.

The most commonly used communication technology in the drone's field are Wi-Fi, radio, Bluetooth, and cellular network [37]. Table 2.2 summarizes the most common communication technologies and their characteristics. Even if the research frontier for drone

communication is mainly focused on cellular networks, in particular the 5G network [39], none of the technologies prevails, but the choice depends on the application scenario. Cellular networks can be a great solution around highly populated areas, while another technology must be considered in rural locations.

Technology	Range	Weight	Complexity	Cost
Wi-Fi	MED [100m]	MED	HIGH	MED
Radio	SHORT-LONG [10-1000m]	LOW	LOW	LOW
Bluetooth	SHORT-MED [<25m]	LOW	MED	LOW
Cellular	LONG [8000m]	LOW	HIGH	MED

Table 2.2: Communication technologies used in drone applications [37]

The software that runs off-board is usually apt to coordinate all the resources of the environment to finally achieve and complete the task needed for the application. When using COTS or intermediate solutions, the vendors usually provide the hardware and software that compose the off-board ecosystem.

Figure 2.4 shows the off-board environment used in this work. It consists of a single base station with a USB dongle radio and an external positioning system (Lighthouse positioning system) composed of two sensors.

A widespread problem encountered while dealing with drones is the testing phase of the application in a real environment. As in any other programming environment, drone programming is not immune to code bugs or hardware problems. Unfortunately, when an error arises, the drone will usually crash; in some unfortunate cases, some parts will break and need to be replaced. Therefore, testing drone applications is a very expensive task both in terms of economic resources and in terms of time.

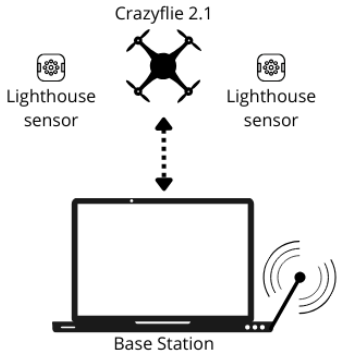


Figure 2.4: EasyFly off-board ecosystem

To overcome this limitation, the main drone producers have developed and distributed a simulation environment that allows testing the software before going into a real scenario. A simulation environment allows running the code written for the planned mission in a graphical simulation that shows the drone performing the tasks. Modern simulation environment [2, 17] usually takes into consideration atmospheric

phenomena like pressure and wind to make the simulation closer to the real deployment environment.

In our work, we built our custom simulation tool, allowing us for a better evaluation of the work itself and, moreover, allowing users of our programming environment to have all the benefits of using a simulation environment.

2.3.2. Swarm Programming

In modern drone applications, where the tasks to achieve are complex, and the application has to be reliable, a common approach is to deploy multiple drones (a swarm) to complete the requested task collectively. This approach is completely different from single drone programming; in fact, swarm programming introduces new challenges and a different approach to achieving the tasks of the application.

Swarm programming is a branch of the more general Swarm Engineering which tries to take advantage of using multiple resources to achieve the application goal with better performance.

Swarm Robotics and, more in general, Swarm Engineering is an emerging discipline that aims at defining systematic and well-founded procedures for modeling, designing, realizing, verifying, validating, operating, and maintaining a swarm robotics system. Taking inspiration from the self-organized behaviors of social animals, it makes use of simple rules and local interactions to design robust, scalable, and flexible collective behaviors for the coordination of large numbers of robots. The inspiration that swarm robotics takes from the observation of social animals (ants, bees, birds, fish, ...) is that starting from simple individuals, they can become successful when they gather in groups, exhibiting a sort of swarm intelligence [22].

In particular, the behavior of social animal groups appears robust, scalable, and flexible. Robustness is the ability to cope with the loss of individuals. In social animals, robustness is promoted by redundancy and the absence of a leader. Scalability is the ability to perform well with different group sizes. The introduction or removal of individuals does not result in a drastic change in the performance of a swarm. In social animals, scalability is promoted by local sensing and communication. Flexibility is the ability to cope with a broad spectrum of different environments and tasks. In social animals, flexibility is promoted by redundancy, simplicity of the behaviors, and mechanisms such as task allocation.

Two great examples in the current literature of swarm engineering are Proto [21] and

Meld [20]. The former is a spatial computing language that allows programming swarms of robots starting from a mathematical model called amorphous medium. The latter is a declarative programming language that uses logic programming to enable swarm programming. Both languages directly take the swarm programming from the point of view of aggregate behaviors, i.e., their approach is to program the entire swarm behavior instead of programming every single component separately.

When swarm robotics is applied in the field of drones, given the high dynamism in the movements of this type of robot, the swarm management and control is much more complex with respect to the single drone, but the capability of the swarm may increase the application's performance.

In the first place, the swarm, compared to the single drone, can provide a higher availability: A single drone has a limited flight time, so its batteries need to be recharged or replaced. A swarm instead can dynamically deploy and retire drones to be always active on the field. In addition, a swarm can also scale when the request increases, e.g., a phenomenon to sense has a peak of occurrence [25]. In the same situation, a single drone application can miss the peak of the phenomena because, for example, it can be stuck at the charging station.

With swarms, the goal of the application is usually defined as a swarm goal. Swarm goals are high-level goals whose achievement is independent of the success or the failure of the single task of a swarm component. The separation between application (swarm) goals and drone tasks allows the application to be scalable and fault-tolerant. Of course, the advantages of the swarm with respect to the single drone hide inside a huge complexity. As the number of components of the swarm increases, the complexity of managing all of them increases a lot as well, introducing a consistent overhead that can affect the application's performance. As in any other engineering problem, we must select and identify the most suitable swarm size for the application to realize.

Depending on the application domain, we can adopt different strategies for coordinating the resources available. We can identify three main programming models that can be applied to swarm programming in the field of drones: Drone-level programming, Swarm programming, and Team-level programming. Drone-level programming is the most straightforward approach; it expects to develop a single application for each component of the swarm, taking into account all the possible interactions between them. This finest grain method allows an entirely independent, customizable, and deterministic single-drone behavior. Since the application has a swarm goal that is not directly related to the single drone's task, with this method, it is usually tricky to use all the swarm resources effi-

ciently to reach the general goal. Moreover, it has been proven [25, 34] that this method is indeed the most complex of the three.

Swarm programming[38], on the other hand, allows writing a single set of rules that are common for all drones, and then every single drone executes that instruction in its local state. The swarm programming model explicitly forbids a shared or global state. This programming model is easier to use and to set up and scale up with multiple drones, but it is challenging to represent tasks that require explicit drone coordination.

Team level programming [34] is a programming model in between swarm and drone level programming, in which users express sophisticated collaborative sensing tasks without resorting to individual addressing and without exposure to the complexity of concurrent programming, parallel execution, scaling, and failure recovery. More generally, Voltron [34] can be viewed as a set of tasks that must be performed by a set of drones subject to particular timing and spatial constraints.

Our work does not address the complexity of swarm programming, although our programming environment, EasyFly, allows for the deployment of multiple drones.

3 | Tools

The main resource used in this work is Crazyflie, an open platform produced by Bitcraze [3] that offers an ecosystem of products and open-source libraries that allow people to develop new functionality for aerial drones.

The key feature of this platform is that it offers a set of expansion decks that can extend the capabilities of the drone with new sensors. Expansion decks can be mounted on the drones very easily, and they are immediately ready to be used to compose the desired configuration in the application of interest. Given its modularity and high versatility, this platform perfectly fits as a baseline for developing an easy, high-level environment for programming drones.

The aim of this chapter is to present an overview of the Crazyflie platform to provide basic knowledge of its tools and surrounding environment.

3.1. Ecosystem Overview

The Crazyflie platform comprises a set of devices and tools to allow building drone applications. It has a modular architecture that makes it possible to build very versatile systems that are adaptable to many situations. In Figure 3.1, we can see an overview of the ecosystem of the entire platform.

The ecosystem of this platform gravitates around its leading actor: the Crazyflie 2.1 nano drone. In the basic settings, the drone has minimal sensors and actuators that allow it to fly. To empower the drone capabilities, the platform makes available a set of expansion decks that give the drone additional sensors, making it possible to adapt it to many possible situations.

To coordinate the drone's operations, the platform needs a ground station that can be hosted on any computer with a Python script interpreter or a mobile phone with a dedicated App installed.

The communication between the quadcopter and the ground station is handled using a

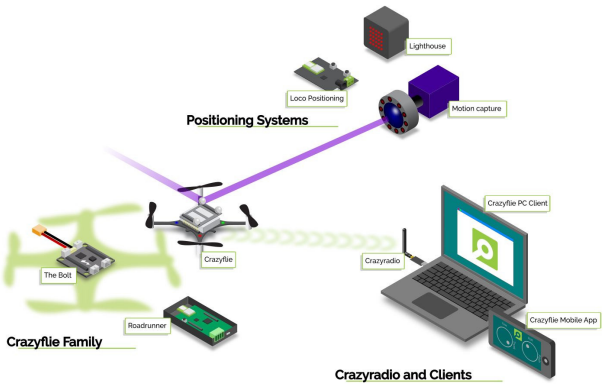


Figure 3.1: This picture represents an overview of the Bitcraze Ecosystem [3]

dongle USB (CrazyRadio) or through Bluetooth when using the mobile App.

The platform also offers multiple absolute positioning systems that allow the drone to have better position estimation in an absolute coordinate system.

3.2. Hardware

In this section, we will provide an overview of the hardware components of the entire Crazyflie platform. We will first analyze the characteristics of the Crazyflie quadcopter used in this work, and then we will briefly introduce the relevant expansion decks. We will then give an overview of the hardware components that compose the absolute positioning system adopted for the work: the Lighthouse positioning system.

3.2.1. The Quadcopter

Bitcraze produces a family of drones with similar hardware and firmware but different sizes and properties. The target for this work is the principal component of this family, the Crazyflie 2.1, a tiny and versatile quadcopter with a solid and modularized design that falls into the category of nano-drones.

As described in Section 1.2, nano-drones are the most suitable typology for conducting investigations around HDI. The modular approach owned by Crazyflie 2.1 constitutes another key advantage in the field of human-drone interactions; the user can easily change the setup to address any possible situation.

This drone has many hardware components hosted on a single, compact, light base. We can identify four main units: Computing Unit, Motor Unit, Sensor Unit, and Power Unit.

The core of the drone is composed of two Micro Controller Units (MCUs): The first is an STM32F4 MCU that handles the main Crazyflie firmware with all the low-level and

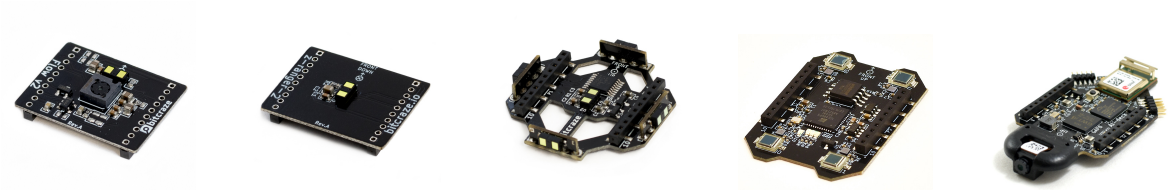
high-level controls. The second MCU, NRF51822, handles all the radio communication and power management.

The motor unit of the Crazyflie 2.1 consists of four Coreless DC motors with plastic propellers fixed at the corners of the base with the help of plastic supports. To control the flight, the drone is equipped with two sensors: a BMI088 sensor, which measures the acceleration along the three coordinates of space plus the angular speed, and a BMP388 sensor, which is a high-precision pressure sensor. The sensor unit of the Crazyflie 2.1, also known as the Inertial Measurement Unit (IMU), is minimal and provides the minimum data that allows the drone to have an almost stable flight.

Its design is robust and simple, easing the assembly and maintenance of its components. Finally, the power unit is constituted by a 240mAh LiPo battery that allows a flight duration of about 7 minutes. The total weight of the drone is 27 grams, and it can lift a payload of 15 grams [4].

3.2.2. Expansion Decks

With only two sensors composing the IMU, the drone has a limited capacity to understand the surrounding environment. To overcome this limitation, the drone can be equipped with additional decks that extend its capabilities in sensing, positioning, and visualization. The platform offers a variety of expansion decks, but for the purpose of our work, only a subset of them has been selected. Figure 3.2 shows the expansion decks selected, particularly those that can enable in some way the human-drones interaction.



(a) Flow deck v2 (b) Z-ranger deck v2 (c) Multi-ranger deck (d) Lighthouse positioning deck (e) AI-deck 1.1

Figure 3.2: Expansion decks of Crazyflie 2.1

Flow deck v2:
The Flow deck (Figure 3.2a) allows the Crazyflie to understand when it moves in any direction. It mounts two sensors: the VL53L1x ToF measures the distance to the ground with high precision up to 4 meters, and the PMW3901 optical flow sensor measures the relative velocity in the x-y plane relative to the ground. This expansion deck is a relative positioning system that lets the drone know its position relative to its take-off point.

Z-ranger deck v2:
The Z-ranger deck (Figure 3.2b) is a simplified and cheaper version of the Flow deck v2. It only measures the distance from the floor up to 4 meters using the usual laser sensor VL53L1x ToF.

Multi-ranger deck:
The Multi-ranger deck (Figure 3.2c) gives the Crazyflie the capability to sense the space around it and react when something is close and for instance, avoid obstacles. This is done by measuring the distance to objects in the following five directions: front, back, left, right, and up with mm precision up to 4 meters, using five VL53L1x ToF sensors.

Lighthouse positioning deck:
The Lighthouse deck (Figure 3.2d) is part of the Lighthouse absolute positioning system (See Section 3.2.4). It comprises four TS4231 IR receivers and an ICE40UP5K FPGA to process the signal received. This expansion deck lets the drone know its position in an absolute coordinate system.

AI-deck 1.1:
The AI-deck 1.1 (Figure 3.2e) extends the computational capabilities and will enable complex artificial intelligence-based workloads to run onboard, with the possibility to achieve fully autonomous navigation capabilities. It mounts an Himax HM01B0 (ultra-low power 320×320 monochrome camera), GAP8 (ultra-low power 8+1 core RISC-V MCU), NINA-W102 (ESP32 module for WiFi communication), and it has 512 Mbit HyperFlash and 64 Mbit HyperRAM memories.

3.2.3. Crazyradio PA

As previously described, the Crazyflie platform expects two nodes of computation: the ground station and the Crazyflie 2.1 itself. To communicate with the Crazyflie 2.1, which has an integrated radio, the ground station needs an external radio dongle. The platform provides a low-latency and long-range USB radio dongle, the Crazyradio PA.

The CrazyRadio PA is based on the nRF24LU1+ from Nordic Semiconductor, and it features a 20dBm power amplifier giving a range of up to 1km (line of sight). The dongle comes pre-programmed with Crazyflie's compatible firmware.

The communication protocol used to communicate is the Crazy Radio Transfer Protocol (CRTP), which is a custom communication protocol of the Crazyflie platform.

3.2.4. Lighthouse Positioning System Hardware

The Lighthouse positioning system is one of the possible solutions that Bitcraze offers to have an absolute positioning system for understanding the drone's coordinates inside the flight space. The hardware of this system is composed of 2 or more HTC-Vive/SteamVR Base Station 2.0. The role of these base stations is to lighten up the flight space with periodic infrared (IR) beams.

Onboard the Crazyflie 2.1, the Lighthouse expansion deck allows it to capture these IR beams thanks to four TS4231 IR receivers. The signal captured is then passed to an ICE40UP5K FPGA that computes the angle of incidence of the IR rays with signal processing. The position and the pose of the Craziflie are then finally computed from the main MCU of the Crazyflie 2.1

3.3. Software Libraries

As previously anticipated, the Crazyflie environment is completed by a set of open-source libraries, publicly available on GitHub, which allow people to program all its components and devices, develop new features, or upgrade existing ones. Each library targets a specific system component and is completely independent of all the others. This section will briefly describe the two main libraries used as a baseline for our work.

3.3.1. crazyflie-lib-python

The crazyflie-lib-python (cflib in short) [6] is a software repository that consists of a Python library for programming scripts that control the behavior of the Crazyflie 2.1. The library provides the base facilities to allow users to define the desired drone behavior in a Python script, abstracting from the low-level control mechanism.

As shown in Figure 3.3, the Python scripts are executed on the ground station. The library contains the code to create communication packets sent through the Crazyradio reaching the Crazyflie, which will eventually execute the commands requested.

3.3.2. crazyflie-firmware

The crazyflie-firmware [5] is a software repository that contains all the firmware of Crazyflie 2.1. The firmware is written in C++ and handles the main autopilot on the STM32F4; it contains the driver of each possible expansion deck and controls all the communication on the opposite side of the cflib.

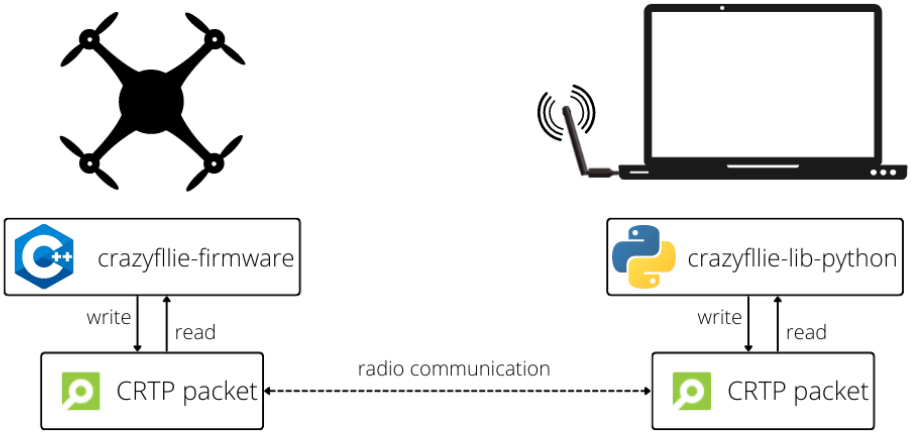


Figure 3.3: The two main software libraries are *crazyflie-firmware* and *crazyflie-lib-python*. The former, written in C++, runs on the drone; the latter, written in Python, runs on the base station. The communication protocol that enables the cooperation of the two is the Crazy Radio Transfer Protocol (CRTP)

3.4. Positioning Systems

Positioning systems represent the core sensing task of every drone application. Knowing the drone's position in the flight space is essential for achieving any possible goal. We can identify two main categories of positioning systems: relative and absolute.

Relative positioning systems are the simplest: starting from the drone's position at take-off, they estimate the position during the flight on the base of the movements made by the drone in the flight space. The measurement of the movements is usually made with the IMU (Inertial Measurement Unit) or with some additional sensors that track the difference in position over time.

The main issue related to this type of positioning system is that the error continually accumulates, and after a certain period of flight, they can lead to consistent mispositioning.

Absolute positioning systems provide coordinates that define the exact location of an object within a specific coordinate system. These systems typically rely on external references or signals from fixed environmental points. The sampling of the position in this type of positioning system is independent of any initial measurements, so the error in the positioning usually does not accumulate over time.

The Crazyflie platform offers multiple positioning systems, both absolute and relative. To select the best system for building our programming environment, we analyzed the following metrics for each possibility that the platform offered:

- Relative or Absolute Positioning System
- Accuracy in sampling
- Cumulative of the error during time

3.4.1. Relative Positioning Systems

The Crazyflie platform offers three relative positioning systems. The first system is represented by the Inertial Measurement Unit, which is provided on the base drone without expansion decks. From our experience, this system has very poor accuracy and cannot be used as the only positioning system of the entire application. However, it contributes its information to obtain the position estimate.

The Z-ranger deck (See 3.2.2) is another positioning system that the platform offers; it provides an estimate only for the z-coordinate with pretty good accuracy, but, as typical for every relative positioning system, it has a high cumulative error during the time.

The Flow deck v2 (See 3.2.2) represents an empowered version of the Z-ranger, a positioning system capable of measuring the distance from the takeoff point for the three coordinates x, y, and z. This last system has a good sampling accuracy and an acceptable cumulative error rate over time. Flow deck v2 is the only relative positioning system that allows the Crazyflie to fly with acceptable precision in the flight space.

3.4.2. Absolute Positioning Systems

The environment provides three different absolute positioning systems solutions with different characteristics, performance, and costs.

The first solution proposed, the Loco Positioning System (LPS), is based on Ultra Wide Band radio that is used to find the absolute 3D position of objects in space. Similarly to a miniature GPS system, it uses a set of Anchors, namely Loco positioning nodes (from 4 up to 8), that act as a GPS satellite, and a Tag, namely, the Loco positioning deck, which acts as a GPS receiver. As indicated in the specification of this system, the accuracy of this system is probably the main limitation and is estimated to be in the range of 10 cm.

The second solution proposed is the Motion Capture System (MCS), which uses cameras to detect markers attached to the Crazyflies. Since the system knows the layout of the

markers on the drone, it is possible to calculate the position and orientation of the tracked object in a global reference frame. It is a very accurate positioning system, but the two main limitations are that the native environment provides only the markers and relies on third-party systems for the entire MCS. Moreover, the position is computed in an external node and needs to be sent to the Crazyflie, increasing the communication load.

The last solution is the Lighthouse positioning system (Lighthouse), the newest introduced in the environment and selected for the work because it overcomes all the limitations of the previous. Lighthouse is an optically-based positioning system that allows an object to locate itself with high precision indoors.

The system uses the SteamVR Base Station (BS) as an optical beacon. They are composed of spinning drums that shine the flight space with infrared beams in a range of 6 meters.

The Crazyflie, on the other hand, with a Lighthouse positioning deck with four optical IR receivers (photodiode), can measure the angle of incidence of the IR beams. Knowing the position and orientation of the BS enables the Crazyflie to compute its position onboard in global coordinates. The knowledge of the position and the orientation of the BS is called system geometry and is composed of a vector in the three dimensions and a rotation matrix.

3.5. State Estimate and Control

All the information given by the positioning systems and by the additional decks are simply data, to become useful, they need to be used cleverly to control the drone's stability and make it possible to fly.

This section will describe the principal software components that act in the control loop, enabling the drone to fly. In the Crazyflie platform, the control loop is managed inside the Crazyflie firmware (See 3.3.2) from the sensor read to the motor thrust. In Figure 3.4, we can see a representation of the control loop of the Craziflie 2.1.

As in any feedback control system, also for the Crazyflie control loop, we can identify two principal components in Crazyflie 2.1: the state estimator and the state control. The synergy between the state estimate and the state control in a control loop is fundamental for achieving accurate and effective system control.

The state estimator is responsible for computing the best possible estimate of the system's current status. This component inside the Crazyflie 2.1 firmware has two concrete implementations with different performances and accuracy:

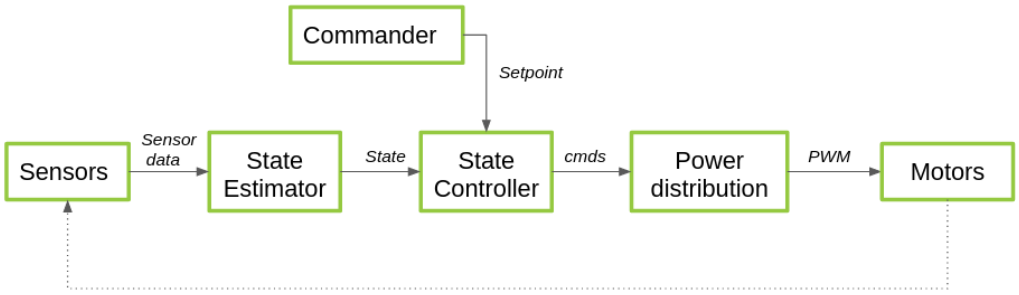


Figure 3.4: The control loop of the Craziflie 2.1 [3]

- Complementary Filter
- Extended Kalman Filter (EKF)

The Complementary Filter is a very lightweight and efficient state estimator. It can only partially use the available sensors; in particular, it uses only data from the IMU and the ToF sensor (Flow deck or Z-ranger). The estimated output is only a portion of the Crazyflie state: the Attitude (roll, pitch, and yaw) and the z coordinate relative to the starting point.

The EKF is a recursive filter that estimates the current state of the Crazyflie based on incoming measurements (in combination with a predicted standard deviation of the noise), the measurement model, and the model of the system itself. It is a step up in complexity with respect to the other estimator; it accepts all the possible sensors' data as input. If enough sensor data are available, it can compute a complete state estimation as output: Attitude, Position, and Velocity in all directions. The choice of which state estimator to use can be forced by the user or automatically set. By default, the firmware uses the lighter Complementary Filter and switches to the EKF if more sensors are available.

The state control uses this estimate to elaborate the actions needed to change the current state to a target state. On the Craziflie's 2.1 firmware, we have three possible alternatives, ordered by complexity:

- Proportional Integral Derivative Controller (PID)
- Incremental Nonlinear Dynamic Inversion Controller (INDI)
- Mellinger controller

3.6. Flight Control

On the other side of the communication channel, the ground station needs a way to interact with the drone and give information on which actions to take to fly in the desired way.

Inside the cflib the one responsible for controlling the flight is a module named Commander Framework. The Commander Framework can be viewed as composed of two layers:

The first layer provides low-level operations that allow writing setpoints and sending them with the custom CRTP protocol.

The second layer, built upon the first, is more abstract and adds some general functionalities, e.g., take-off, land, and move-to.

4 | Communication Framework

In the domain of drone systems, effective communication stands as a central pillar for operational success. Communication with drones refers to the process of transmitting and receiving information between a drone and a remote control device or computer system. It involves the use of various technologies, including Wi-Fi, Bluetooth, and radio frequency (RF) signals, to control the drone and receive real-time data and feedback.

This chapter focuses on a crucial element of our drone programming environment: the Communication Framework. Its purpose is to handle all the communication between the ground station and the flying drone. More in detail, the Communication Framework is a software component hosted on the ground station that allows the user to establish effective communication between the script running on the ground station and the drone.

The Communication Framework manages the communication stream into two separate flows: parameter setting and telemetry logging. The former is the flow of communication that is directed from the ground station to the drone; it is used to set configuration parameters onboard the drone. The latter is the opposite flow of communication, from the drone to the ground station, and it is used to receive all the telemetry data of the drone, allowing for better control.

4.1. The Communication Infrastructure

Before diving into the details of our Communication Framework, it is crucial to understand the underlying communication infrastructure.

As described in Chapter 3, our programming environment is spread into two main devices: the ground station and the drone. The drone's firmware is written in C++ and runs on a small board with limited resources. On the other side, the ground station runs Python scripts that control the drone's behavior. The heterogeneity between these two subsystems increases the complexity of the communication process.

The platform uses a custom network protocol called CRTP to overcome all these challenges. CRTP was designed to allow packet prioritization to help real-time control of the

Crazyflie; in the current implementation, the link guarantees strict packet ordering within a port.

As shown in figure 4.1, the Crayzyflie communication is implemented as a stack of independent layers. Every layer of the communication stack has two parallel implementations, one hosted in the firmware onboard and the other on the ground station's software.

The physical layer is responsible for transmitting packets to and from the Crazyflie.

The link implements safe and ordered packet channels to and from the Crazyflie. This layer abstracts the physical medium and implements one transmitting and receiving packet channel to and from the Crazyflie.

The CRTP layer introduces the concept of a logical packet; each packet measures 32 Bytes and is composed of three parts: port, channel, and payload. The tuple *port:channel* allows the packet to be delivered to the specialized subsystem that receives and processes the message.

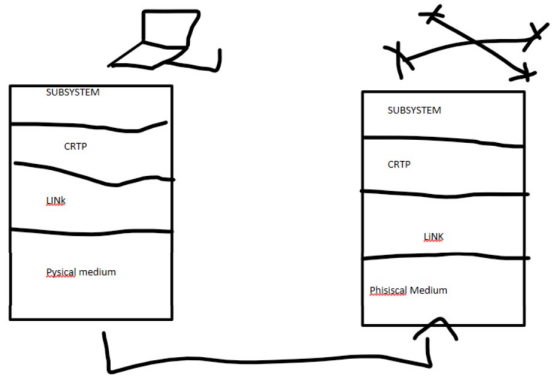


Figure 4.1: The Crayzyflie communication stack is composed of 4 independent layers: Subsystems, CRTP, Link, Physical medium

In the higher-order layer, we can identify four principal subsystems:

- **Memory access** – that manages memory operation on the Crazyflie's physical memories, e.g., trajectory uploading.
- **Commander** – that send/receive control set-points.
- **Parameters** – that manages read/write operations on configuration parameters of the Crazyflie.
- **Data Logging** – that handles logs of telemetry data sent periodically from the Crazyflie to the ground station.

The first two subsystems (memory access and commander) are the simplest, with a straightforward implementation. They expose some methods through an API, and when

called, they craft a packet, fill it with parameters data, and send it to the link layer. On the other side, the same subsystem receives the packet, unpacks the data, and executes the desired function.

Conversely, the parameters and data logging subsystems have some peculiarities that make them more complex. In particular, the parameters subsystem allows performing read/write operations with acknowledgments of the successful operation. The data logging subsystem, instead, must support periodical updates of telemetry data. Moreover, both the two subsystems must handle a variety of variables with different types.

The Communication Framework that we designed and developed is meant to wrap the functionalities of these two complex subsystems and offer the user a more usable and efficient way to handle data logging and parameters.

4.2. The Design of the Communication Framework

The Communication Framework is a software component that provides an easy and accessible way to set parameters onboard the Crazyflie and to log telemetry data. This component is part of the ground station's software and has been designed as a publish/-subscribe system.

Every CRTP data that is routed to the parameter or logging subsystem is managed by the Communication Framework and stored in a central repository. Every other software component or script interested in such data can use the Communication Framework to access the repository and subscribe for value updates.

Given the profound semantic difference between the logging and parameters subsystems, we decided to keep the two concepts separated inside the Communication Framework. For this reason, we developed two different Communication Managers, one for every subsystem. The two implementations, namely the Logging and the Parameters Manager, shares the publish/subscribe design structure, but they slightly differ in the management of the updates in the repository.

4.2.1. The Logging Manager

As described above, the Logging Manager is one of the two implementations of the Communication Framework. In particular, the role of the Logging Manager is to handle the setup and the periodic updates of telemetry data sent from the Crayzyflie.

The internal structure of this component is represented by a values repository where key-

value pairs are stored to maintain the telemetry state. More in detail, each variable, identified by a unique key, represents an entry inside the values repository; the value of the entry is the last updated value of the logged variable.

In a separate repository, the Logging Manager stores the subscribers interested in the variables updates. In this second repository, the component stores a list of references to subscribers for each single variable or set of variables. Each reference consists of a predicate function to allow filtering updates and a callback function to effectively notify the subscriber when the predicate is satisfied with the new value.

To better understand the working principle behind the Logging Manager, we illustrated the complete workflow in Figure 4.2.

Whenever the script running on the ground stations needs to start tracking a variable, the Logging Manager creates or updates a log configuration and sends it to the Crazyflie. The configuration contains information regarding the variable name, the type, and the desired period of updates.

At some point, when the script is interested in reading and getting updated on the variable's values, it creates a callback function, i.e., an action to be executed with the new updated value, and, optionally, a predicate function for receiving updates only when the values satisfy certain constraints.

Then, the script registers these functions to the Logging Manager, specifying the single variable or set of variables of interest. When the logging process starts, the Crazyflie sends periodic messages containing the variable updates; the Logging Manager intercepts those messages and updates its internal values repository. Each time an update occurs, the Logging Manager checks whether any subscriber is watching that variable. If so, for each subscriber, it evaluates the predicate and eventually notifies the subscriber by calling the registered callback.

Given that each variable can have its own size (given the type) and period, the Logging Manager needs to handle different log configurations. Moreover, the single configuration has a size constraint of 26 Bytes given by the size of the CRTP packet, increasing the complexity of the configuration management.

To optimize the number of log configurations created, each time the Logging Manager needs to add a variable, it searches for a suitable existing configuration with enough space to fit it. The Logging Manager only adds a new configuration when strictly necessary, ensuring maximum optimization.

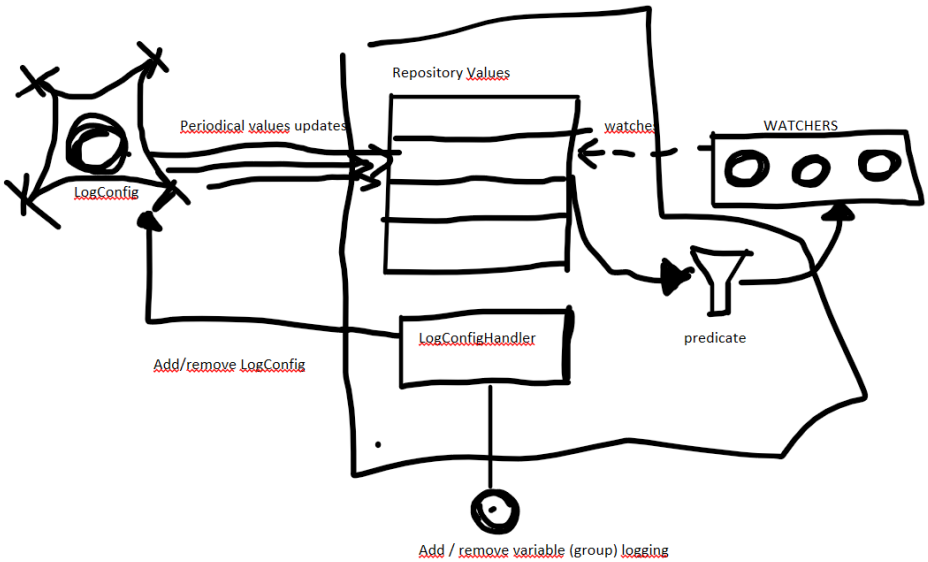


Figure 4.2: Logging Manager's workflows

4.2.2. The Parameters Manager

The other implementation of the Communication Framework, the Parameters Manager, manages the configuration operations of the Crayzyflie. The Crayzyflie 2.1 allows the users to set parameter values to configure the desired setup. An example of a parameter could be the selected state estimator, i.e., Extended Kalman Filter or Complementary Filter (See Section 3.5).

Following the same structure as the Logging Manager, also the Parameters Manager has one value and one subscriber repository where it stores, respectively, the current value for tracked parameters and the references of subscrbers for each variable.

Conversely, variable updates are managed slightly differently with respect to the Logging Manager. In fact, the underlying parameters subsystem does not have a periodic update of values; instead, the updates happen only when the parameter's value is effectively changed onboard. In other words, the Parameters Manager only receives and processes an update when the Crazyflie acknowledges a state change in the parameter's value.

Figure 4.3 represents the operations workflow inside the Parameters Manager. When needed, usually before the takeoff, the script can use the Parameters Manager to read or write configuration values.

Read operations are processed straightforwardly with a synchronous request-response messages exchange. The request contains the name of the variable, and the Crazyflie's re-

sponse contains the current value of the requested parameter.

Write operations, on the other side, are processed asynchronously; first, a request message is sent from the Parameters Manager to the Crazyflie, then, when the Crayzyflie successfully completes the update, it notifies the ground station's Parameters Manager.

Once the Parameters Manager receives a value, it uses the internal publish/subscribe system to notify subscribers.

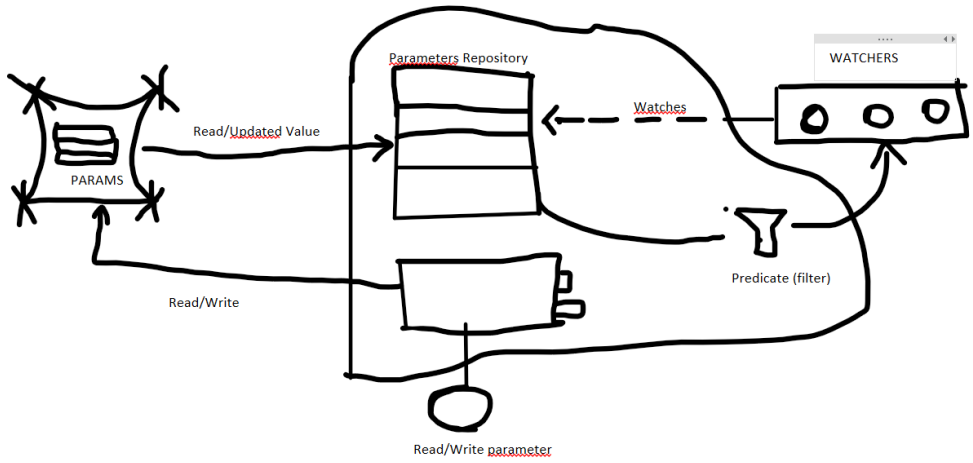


Figure 4.3: Parameters Manager's workflows

5 | Coordination Framework

In any drone application, the drone's ability to react after an event is a key factor for the success of the application itself. In this chapter, we present EasyFly's Coordination Framework, a software component belonging to the ground station system meant to coordinate and orchestrate all the actions of the Crazyflie.

The Coordination Framework acts as a centralized entity that is able to synchronize different parts of the same applications. Moreover, it is possible to coordinate multi-drone applications easily using the Coordination Framework.

In this chapter, we first present the role of this framework and its use cases, providing some concrete examples of possible applications. Then, we will analyze its structure and design principle in detail.

5.1. The Role of the Coordination Framework

Coordination plays a crucial role in the successful implementation of drone applications. Depending on the application type, the coordination of a drone can be seen at two different levels:

- Intra-drone coordination
- Inter-drone coordination

The first level of coordination considers the drone as an individual. In this case, coordination is considered to be the process of reacting in the face of a change in the surrounding environment or the drone's internal state with an ad hoc action properly defined.

In the case of intra-drone coordination, the role of the Coordination Framework is to watch over the current state of the drone (internal state or sensor data). When a specific condition is met, it performs the corresponding action.

A simple use case of the Coordination Framework considering intra-drone communication can be represented by the following:

> For the drone application that we are developing, we need the drone to be maintained not too close to the walls of the flight area. Specifically, it must maintain a safe distance of 0,5 meters from the walls.

In the use case scenario described above, the Coordination Framework will act as a supervisor entity that watches the values read from the lateral proximity sensors of the drone. Whenever the distance is less than or equal to 0,5 meters, it will dispatch the action to move the drone in the opposite direction, bringing the distance from the wall back over the safe distance required.

Conversely, inter-drone coordination applies only to applications where the number of drones is more than one. In these cases, the coordination is considered among individuals belonging to the same swarm.

By synchronizing the actions of multiple drones, coordination allows for the efficient and effective achievement of complex tasks that would be difficult or impossible for a single drone to accomplish alone. In this type of application, the role of the Coordination Framework is to watch over the state of the entire swarm and perform actions whenever a condition on the swarm state is met.

To make this concept of inter-drone coordination clearer, let's consider also in this case a simple use case:

> We need to develop a swarm drone application that performs simple choreography for a demonstration during a fashion event. Our swarm is composed of three drones: D1, D2 and D3. The choreographer designed the choreography: D1 starts flying and slowly increases its height to 1 meter from the ground. After D1 completes its movement, D2 also starts flying and reaches a height of 0.5 meters. When D2 has reached the target height of 0.5 meters, the drones D1 and D3 will slowly meet D2 at 0.5 meters from the ground. Finally, to complete the choreography, land all the drones together.

In this use case scenario, the Coordination Framework needs to track the height of each swarm component. For this reason, the swarm state will be represented by the tuple $\langle D1\ altitude, D2\ altitude, D3\ altitude \rangle$. Whenever the state changes, the Coordination Framework will decide on which actions to take according to the Table 5.1

D1 altitude	D2 altitude	D3 altitude	Action to take
0	0	0	move D1 up to 1 meter
1	0	0	move D2 up to 0.5 meter
1	0.5	0	move D1 down and D3 up to 0,5 meter
0.5	0.5	0.5	land all the drones

Table 5.1: Use case scenario of inter-drone communication

The key advantage of using this approach in both intra and inter-drone communication is that the entire business logic of the application is defined inside the Coordination Framework. This fact implies that the only challenging task that the developer needs to take care of is the definition of the state and the relative actions to take when the state changes. If the state is correctly updated and some external entity properly performs the actions defined, the Coordination Framework will orchestrate all the stuff to achieve the application's goals correctly.

5.2. Structure and Design Principles

The design structure of the Coordination Framework is a publish-subscribe architecture that follows the principle of the observer pattern.

The observer pattern is a design pattern in software engineering that defines a one-to-many relationship between entities, where changes in one entity (the 'observable' or 'subject') are automatically propagated to other entities (the 'observers') that depend on it.

In the observer pattern, the observable stores a list of observers and notifies them automatically whenever it changes state. This way, observers can take action based on the observable's new state without constantly polling it for changes.

Following this design pattern, we designed the Coordination Manager considering four main actors:

- *Observable* – Represents a state that can change over time.
- *Producer* – The script/component that creates the Observable, providing an initial state. Usually, this actor is in charge of updating the state of the Observable during time.
- *Observer* – The script/component interested in watching some Observables' changes. Upon subscribing to the Observable, the Observer is notified by the framework when

the former's state changes.

- *Coordination Manager* – It is the component in charge of managing the entire state of the drone application, allowing Producers to update the value of a portion of the state and notifying Observers whenever this happens.

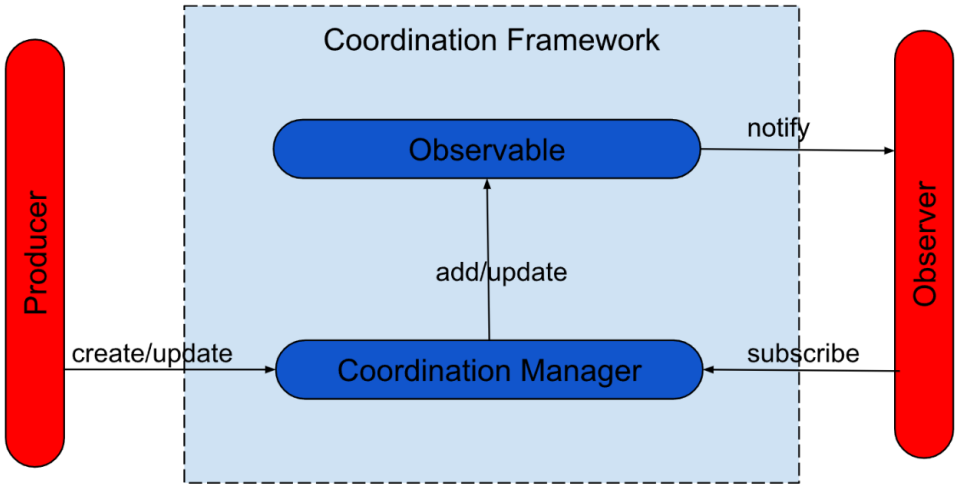


Figure 5.1: The architecture of the Coordination Manager comprises two internal entities: the Observable and the Coordination Manager. Conversely, Producers and Observers are external to the framework and interact with it.

As presented in Figure 5.1, the core of the Coordination Framework is composed only of the Observable and the Coordination Manager entities. The Producer and the Observer are entities considered outside the frameworks and only act as utilizers of its functionalities.

5.2.1. The Observable

In our Coordination Framework, the Observable can be considered as the basic block of the entire structure. In the core, the Observable holds a piece of information known as the state of the Observable.

The state can be as complex as desired, and most importantly, it can change over time. Moreover, the state is atomic and cannot change partially; in other words, regardless of the complexity of the state, every update must provide complete information.

The Observable, on its own, is meaningless. It becomes valid only when an external entity (observer) is interested in watching the state and being notified when some update occurs.

For this reason, the Observable stores a list of references, known as subscriptions, to the external entities interested in watching the value of the state. In the standard observer

pattern, the subscription consists of a callback function that the Observable uses to notify the Observer of the new value. In our implementation, we decided to add some additional information to the subscription to increase the functionality of the pattern. In particular, the subscription is a tuple composed of the following parts:

- *Action* – A callback function to be called whenever the state of the Observable changes
- *Condition* – A predicate function that filters the values before notification. It is a function of the updated state of the Observable and decides whether to notify the Observer or not.
- *Context* – It is an optional object to be passed as a parameter to the Action. It can contain some helpful information to allow the Observer to take the action. When subscribing, the Observer can provide an arbitrary Context value; the provided Context will be stored in the subscription and sent back with each notification.

In the current implementations of the Coordination Framework, we have developed two types of Subsctipions, one synchronous and the other asynchronous. As the names suggest, the synchronous subscription will block the execution of the Observer until the state is updated and the condition is satisfied. Conversely, using the asynchronous subscription, the Observer will continue executing and be called back later when the value changes and the condition is met.

The standard observer pattern provides only the asynchronous notification. We decided to extend the pattern with the synchronous one also because, sometimes, it is useful to check that the drone application is in the desired state before continuing the execution.

5.2.2. The Coordination Manager

The Coordination Manager is the orchestrator of the entire Coordination Framework. Its role is to manage the whole application's Observables, creating and updating them when requested by some external Producer.

Its core is a global repository of Observables, representing the whole application state. Inside this repository, the Observables are identified by a unique name.

As shown in Figure 5.1, the Coordination Manager represents the facade of the entire Coordination Framework; external entities like Producers and Observers are forbidden to manipulate the values stored inside the Observables directly. This constraint allows centralized control, avoiding any possibility of an inconsistent state.

To better understand the working principle of the Coordination Framework, we can show in action the Coordination Manager explaining more in detail how the use case scenario described at the beginning of the chapter can be easily solved with this approach.

The scenario is the following:

> For the drone application that we are developing, we need the drone to be maintained not too close to the walls of the flight area. Specifically, it must maintain a safe distance of 0,5 meters from the walls.

The first thing to do to solve this problem using the Coordination Manager is to define the State of the application. In particular, we can identify four Observables:

- Left Observable
- Right Observable
- Front Observable
- Back Observable

Our state is then represented by the following:

```
State = {
    "Observable_Left": "Distance from Left wall",
    "Observable_Rigth": "Distance from Right wall",
    "Observable_Front": "Distance from Front wall",
    "Observable_Back": "Distance from Back wall",
}
```

Once we have defined the state and managed the update of such a state, then we can define the second part of the Coordination Manager. To do this, we can set up 4 Observers, one for each Observable, to move the drone away from the respective wall. In particular, for each Observer, we can define an Action like the following:

```
def move_away_left_wall_action( actual_distance, contex):
        context.move_drone_right(0.5 - actual_distance)

def move_away_rigth_wall_action( actual_distance, contex):
        context.move_drone_left(0.5 - actual_distance)

def move_away_front_wall_action( actual_distance, contex):
        context.move_drone_back(0.5 - actual_distance)
```

```
def move_away_back_wall_action( actual_distance , contex ):
        context.move_drone_front(0.5 - actual_distance)
```

Moreover, since we want to perform the action only if the distance from one wall is less than 0.5 meters, we can specify for all the Observer the following Condition function:

```
def closer_enough_condition(actual_distance):
        return actual_distance < 0.5
```

With this approach, the main advantage is that the entire business logic of the application is defined inside the Coordination Framework. As the solution to the use case shows, this framework simplifies the task for a user, letting them focus on the definition of state and reactions instead of trying to manage all the coordination between the parts.

Another advantage is that the resulting code is clean and can easily be read and understood also by users with less programming experience.

On the other hand, when the number of Observables and Observers starts to become very big, the constraint of handling everything in a single and centralized component will surely introduce some overhead.

6 | Extended Crazyflie

In the previous chapters, we analyzed two core parts of our EasyFly programming environment: the Communication and the Coordination frameworks. At this point, these two frameworks can appear isolated and useless for reaching our final objective of creating a simple programming environment for drone applications in HDI research.

To leverage the real potentiality of these two frameworks, we need a principal actor that links all the pieces together and transforms our EasyFly from being a software library to a complete programming environment for drone applications.

Here is where the Extended Crazyflie (ECF) comes into play: the simplest but most effective component of EasyFly. This component is a centralized unit that wraps and links together all the functionalities of our programming environment. Moreover, it directly connects to a Crazyflie 2.1 and is used as a unique interface to interact with and control the drone's behavior.

In this chapter, we will present the Extended Crazyflie component, analyzing in detail its structure and the main functionalities. We will conclude by presenting a possible solution for the use case scenario described in the previous chapter (see Secion5.2.2), by using this approach.

6.1. The Structure and Design Principles of Exteded Crazyflie

Learning a new programming environment for drone applications can be really challenging. Usually, when used for the first time, the setup phase of a programming environment is very complicated and full of issues.

Since EasyFly is intended to be a simple and accessible programming environment for drone applications, the learning curve must be as fast as possible. For this reason, we have been focused on two main objectives:

- Automation of all processes that are usually performed in most drone applications

- Centralization to a single component for all the functionalities provided inside the programming environment

In particular, the first objective should allow a beginner developer to set up the environment quickly and be ready to develop in a few minutes. By simply knowing the address of the Crazyflie 2.1, the developer should be able to connect and control it without any effort.

The second objective should allow a beginner developer to know all the functionalities they can use to control the drone by simply looking at the API of one single component. Whatever is accessible from that component is all they need to control the Crazyflie in the desired manner.

The ECF component structure is simple and follows the facade design pattern: The Facade pattern is a design pattern in software engineering that provides a simplified interface to a complex system of classes, interfaces, and objects. It is commonly used to hide the complexity of a system; it presents a simple and unified interface and perfectly fits our needs.

To effectively implement the facade pattern, the ECF component initializes and keeps the references of the other components of the programming environment, such that any part of the EasyFly is accessible via the ECF component.

As shown in Figure 6.1, the ECF component is composed of three parts:

- The Communication Framework
- The Coordination Framework
- A set of additional modules

The frameworks discussed in the previous chapters represent the first two sections of the structure. In particular, we included one instance for each framework's manager in the ECF component. We have one Logging and one Parameters Manager for the Communication Framework, while for the Coordination Framework, we have one Coordination Manager.

The last section of the structure is represented by a set of Modules, each of which is in charge of introducing additional functionalities to the environment. The type of modules instantiated is automatically determined by the ECF component and depends on the current configuration mounted onboard the Craziflie 2.1 (see Section 6.1.1). We will enter more in detail about such modules in the following sections; for now, let us consider these modules as an encapsulated set of functionalities.

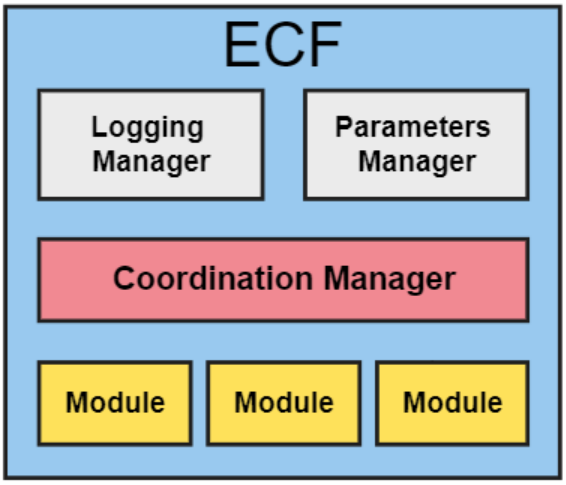


Figure 6.1: Extended Craziflie component structure

The successful implementation of the facade design pattern allows for a centralized interface, accomplishing the second objective previously stated. Still, the Extended Crazyflie does not include any automation processes, allowing users to eliminate the expensive setup phase and lowering the learning curve.

To understand how the ECF accomplishes this other objective, we need to take it a step further and analyze better how the structure described above is created when a new instance of the ECF is created. In Figure 6.2, we can see the sequence of the operations performed at the component's initialization. As we anticipated, the ECF is directly linked to a Crazyflie 2.1, so before getting an instance of the main component, we need to know the address (uri) of the Crazyflie that we desire to connect.

The first thing the component does is initialize the communication protocol drivers from the ground station to the drone and instantiate the connection between the two. After that connection is established, the ECF creates a unique instance for the Parameter Manager, the Logging Manager, and the Coordination Manager and stores them in its local storage.

To initialize the software sensing used for estimating its state (see Section 3.5) with a clean initial state, resetting the estimators before taking off is usually helpful. Indeed, by default, the ECF uses the Parameters Manager to reset the state estimate right after the creation of the Managers.

The last operation in the initialization phase is the creation of the additional modules. Since the modules creation process is automatic and depends on the current configuration of decks onboard, the ECF needs to know the current configuration. To do so, it uses the Parameters Managers; by reading some parameters, it can easily determine which decks are mounted onboard.

Each module can be viewed as a simple component, so for each module needed, the ECF creates a new instance of the module's component and stores it in its local storage. The ECF stores its modules in a local repository so that each module's instance can be retrieved straightforwardly by knowing the module's type.

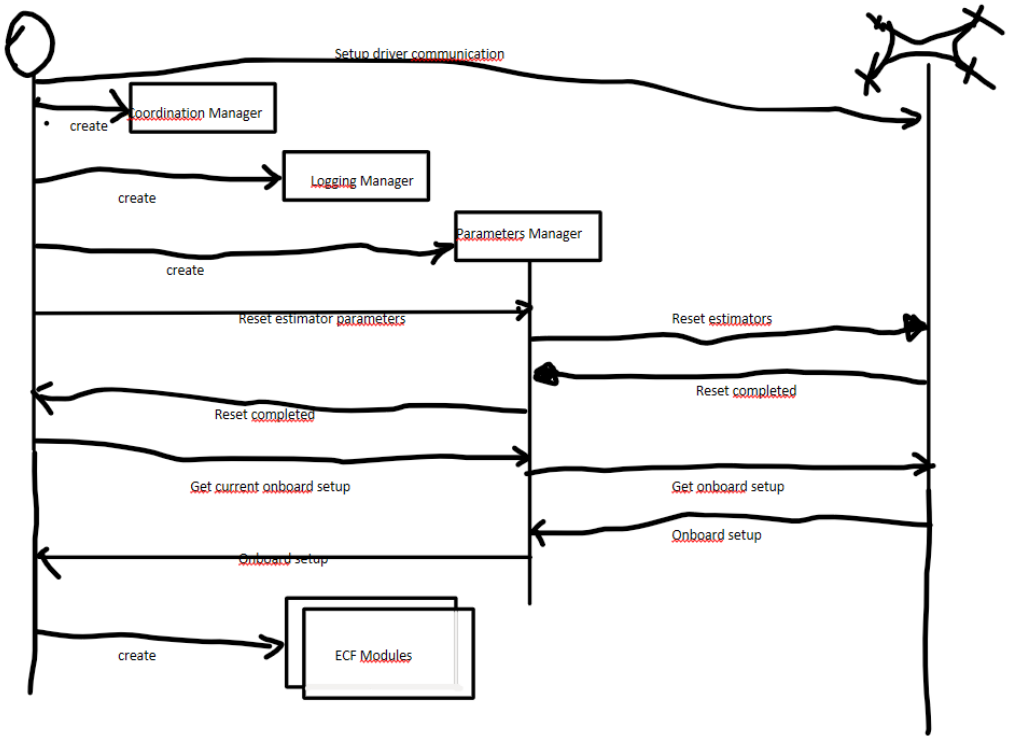


Figure 6.2: The sequence diagram of the initialization of the Extended Craziflie shows the operations performed to have a working instance of the component

6.1.1. Extended Crazyflie Modules

As we have seen, the Extended Crazyflie component instantiates a set of modules, each of which can be viewed as a container of functionalities. So far, it is still unclear what such modules are, what their structure is, what those added functionalities are, and last but not least, which type of modules the EasyFly programming environments have.

In this section, we will deeply understand ECF Modules, and we will answer all of these questions. Before entering into the details of modules, let's step back and understand the origin of the idea behind those modules. To make this, think back to the use case scenario solved in Section 5.2.2, and more generally, think of the process we proposed to develop a drone application easily using EasyFly.

The steps that we followed were:

- Define the state of the application
- Setup the Coordination Framework to handle that state
- Setup the Logging Manager to update correctly that state
- Define how and when reacting to state changes

Apart from the first step, which is usually the most complex and depends mostly on the application scenario, we can say that the other steps are made of identical operations, almost independently from the application. Apparently, the first step, which defines the state, is indeed profoundly diverse across different applications. But if we analyze the possible states that the application can use, we can notice that the state is usually composed only of internal and sensed data; resulting in a very limited set of variables. Ultimately, we will use the same state or a union of "standard" parts for most of the applications.

For example, the estimated position information is always crucial in almost any drone application. Hence, we could define a component that automatically creates a state inside the Coordination Manager and keep it updated.

An ECF Module is a component that defines a *standard state*. Using the Communication and Coordination frameworks, it updates this state and makes it available through a *standard Observable* inside the Coordination framework.

By using standard modules, we reduced and focused (for most of the application) the development process to a single task: defining how and when reacting to state changes. The structure of an ECF module is shown in Figure 6.3: it has a list of state variables, each of which is continuously updated every time a new value is available. So, at any point, these variables contain the last value known read from the Crazyflie for that variable.

As the last fundamental component, the ECF Module can include some utility functions strictly related to the state they define. In the next chapter (see Chapter 7), we will better analyze ECF modules and their utility functions.

To answer the last question: what type of ECF Modules exist in EasyFly? We need to consider valuable states to become *standard states*.

Since our state can be defined either by internal state or by external sensed data, we decided to create the following Modules:

- Internal State
  - State Estimate Module

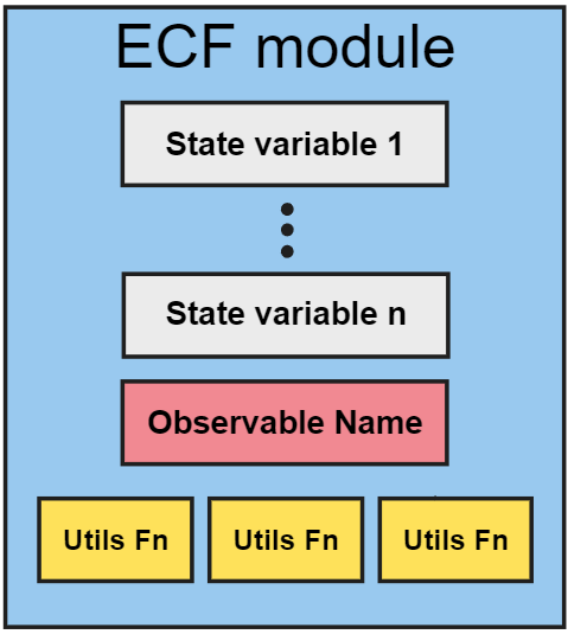


Figure 6.3: Extended Craziflie Module internal structure

  - Battery Module

- Sensed Data State
  - Multiranger Module
  - Height Module
  - Lighthouse Module
  - Ai Deck Module

6.1.2. Use Case Scenario's Solution

We analyzed in detail what ECF is and what its structure is. Moreover, we described the purpose and the functioning of its core part, the ECF Modules.

At this point, we will show a possible solution to the same use case scenario we solved in Chapter 5.2.2, this time using the functionalities of the ECF component. We purposely selected the same example that is already solved to see how much complexity the ECF handles, hiding it from the developer.

For simplicity, we rewrite the specification of the example:

> For the drone application that we are developing, we need the drone to be maintained not too close to the walls of the flight area. Specifically, it must maintain a safe distance of 0,5 meters from the walls.

If we analyze the requirements, we can say that the only information we need to know to control the drone is the distance from the wall. For that purpose, we can use the Multiranger deck, which will give us the distance from the nearest obstacle in the four directions (see Section 3.2.2). Before going straight forward to the implementation, we should briefly introduce the module that we will use in this example: the Multiranger module. The Multiranger module is an ECF module which defines a Sensed Data State as follows:

```
State = {
    "left": "Distance from nearest obstacle on the left",
    "right": "Distance from nearest obstacle on the right",
    "front": "Distance from nearest obstacle on the front",
    "back": "Distance from nearest obstacle on the back",
    "up": "Distance from nearest obstacle above"
}
```

For this state, the Multiranger Module will create and keep updated an observable in the Coordination Manager. We can take advantage of this Observable and subscribe to it to react when the distance goes below 0.5 meters. Here is the code of the example:

```
from cflib.positioning.motion_commander import MotionCommander
from extension.decks.deck import DeckType
from extension.extended_crazyflie import ExtendedCrazyFlie

def safe_distance(multiranger_state : dict, mc : MotionCommander):
    if multiranger_state['left'] < 0.5:
        mc.right(0.5 - multiranger_state['left'])
    if multiranger_state['right'] < 0.5:
        mc.left(0.5 - multiranger_state['right'])
    if multiranger_state['front'] < 0.5:
        mc.back(0.5 - multiranger_state['front'])
    if multiranger_state['back'] < 0.5:
        mc.forward(0.5 - multiranger_state['back'])

with ExtendedCrazyFlie('radio://0/80/2M/E7E7E7E7E7') as ecf:
    with MotionCommander(ecf.cf) as mc:
        ecf.coordination_manager.observe(
                observable_name= ecf.decks[DeckType.bcMultiranger]
                                    .observable_name,
```

```
            action= safe_distance,
        context= [mc],
    )
```

The code is organized into two main parts; the first part (*lines 6-14*) is a function that contains the core logic of the application. If we analyze the signature, the function is an Action of the Coordination Framework: it takes as input the state of one observable and a MotionCommander object (the context) that is used to act on the Crazyflie effectively.

Four *if* statements in the function's body check whether the distance from the nearest obstacle in one direction is below 0.5 meters. If that is the case, the function will use the Motion Commander object to move the drone in the opposite direction, bringing the drone back to a safe distance from the obstacle.

The second part (*lines 17-23*) instead represents the only setup needed to create a working application. It comprises 2 *with* statements and a single function call.

Line 17 will create a new instance of Extended CrazyFlie and assign it to the variable *ecf*. As we know, all the setup operations are performed by creating the ECF component. Since we are mounting a Multiranger deck, it will also automatically create an instance of the Multiranger Module. The user with a single line of code is ready to use all its functionalities.

Line 18 will create and initialize an instance of the Motion Commander. The Motion Commander is a functionality offered by the base cflib (see Section 3.6), which exposes some functionalities that allow controlling the drone flight.

In line 19, there is a call to the Coordination Manager to set up a subscription to the multiranger observable. In particular, we specified an action, the *safe_ distance* function, and a context, the Motion Commander's instance *mc*. In this way, every time the Multiranger Module updates the state, the *safe_ distance* function will be called with the new state and the instance of the Motion Commander; if the function finds anomalies in the distance, it will apply a correction.

As we can see, the resulting code is clean and easy to understand, and all the complicated setup operations are handled automatically under the hood. The Extended CrazyFlie component is a tool with much potential that allows the developer to start with the proper setup by writing a single line of code. Moreover, with standard ECF Modules, the developer can focus on the application's core business logic and reduce the skills needed to program an application and the time necessary to complete it effectively.

7 | ECF Modules

As described in the previous chapter, ECF Modules are the most essential part of the EasyFly programming environment. In this chapter, we will analyze each ECF Module available in the EasyFly extension in detail, describing the purpose, main functionalities, and implementation details for each. We will first analyze the Modules with an internal state and then move to the Modules with a sensed data state.

7.1. State Estimate Module

Regardless of the purpose, a crucial aspect of operating a drone is having a deep understanding of the drone's position, velocity, and acceleration. These parameters are vital for ensuring the drone's stability and safe navigation. Accurate knowledge of the drone's position and speed enables its operator to control its movements and avoid obstacles or hazards in its path. Furthermore, understanding acceleration is crucial in achieving precision and accurate control of the drone's movements. State estimate accuracy depends on how much information the control loop can use. In particular, the decks that contribute more to the estimation process are the Lighthouse positioning Deck and the Flow Deck V2.

As the name suggests, the State Estimate Module is an ECF Module that manages the estimated variables of the Crazyflie 2.1. The full state of the State Estimate ECF Module is:

```
State = {
    "x" : "Position on the x-axis from the origin",
    "y" : "Position on the y-axis from the origin",
    "z" : "Position on the z-axis from the origin",
    "vx" : "Velocity on the x-axis",
    "vy" : "Velocity on the y-axis",
    "vz" : "Velocity on the z-axis",
    "ax" : "Acceleration on the x-axis",
    "ay" : "Acceleration on the y-axis",
```

```
    "az" : "Acceleration on the z-axis",
    "roll" : "Roll in rad",
    "pitch" : "Pitch in rad",
    "yaw" : "Yaw in rad",
    "rateRoll" : "Roll rate in rad/s",
    "ratePitch" : "Pitch rate in rad/s",
    "rateYaw" : "Yaw rate in rad/s",
}
```

Because the control loop onboard the Crazyflie 2.1 runs at 500Hz, a new estimation is computed every two milliseconds; these variables can rapidly change over time. We need to update the state as frequently as possible to ensure that the information provided is accurate and precise. For this reason, we selected the lowest possible sampling period for the Communication Framework, which is 10 milliseconds.

In this ECF Module, the utility functions allow the user to record the state variable and, at the end of the flight, plot the recorded data. As with any other telemetry data, it is usually vital for a user to have the possibility to analyze that information after the flight, especially in the development phase of the application. For this reason, the utility functions that we implemented will allow a user to analyze and compare data from multiple application runs.

7.2. Battery Module Module

When dealing with drones, battery management is an important factor that must always be considered. Usually, due to its weight, the battery has minimal capacity, especially on drones with small dimensions.

An application can be perfectly developed, but it can easily fail if it does not consider the limitation of resources like the battery. To help the developer with this task, we created another ECF Module that manages battery-related information and keeps them updated.

Given this, the full state of the Battery ECF Module is:

```
State = {
        "pm_state" : "Battery power management state",
        "voltage" : "Battery voltage in V",
        "battery_level" : "Estimated battery level in percentage",
}
```

The first variable of the state, *pm_state*, represents the state in which the power management of the drone is in. The state can be one of the following:

- Battery – The drone is on and using its battery.
- Charging – The drone is plugged into the power supply.
- Charged – The drone has completed the recharge, and its battery is fully charged.
- Low Power – The drone needs to be recharged.
- Shutdown – The drone is off.

With the other two properties of the state, *voltage* and *battery_level*, the developer can determine with high precision the recharging phase.

Since battery management strictly depends on the application to be developed, it is hard to find an implementation that satisfies all the possible usage. For this reason, we decided to leave the implementation of the decision processes for determining the outside of the Battery Module. The module is responsible only for keeping the information in the state consistent and updated.

7.3. Multiranger Module

The Multiranger deck is an accessory board for the Crazyflie 2.1. The deck is designed to provide the drone with range-sensing capabilities, allowing it to detect objects and obstacles in its environment.

The Multiranger deck features four integrated ultrasonic sensors, which can be used to measure distances in a range of up to 4 meters. These sensors emit high-frequency sound waves that bounce off of nearby objects and return to the deck, allowing the deck to calculate the distance to the objects based on the time it takes for the sound waves to return. Multiranger deck is a powerful tool for adding range sensing capabilities to the Crazyflie 2.1, enabling it to perform a range of applications such as mapping, autonomous navigation, and obstacle avoidance.

As any other ECF Module, also the Multiranger Module has a state that it manages. The state is composed of the 5 range measurement, one for each direction that it supports:

```
State: {
    "front" : "Distance␣from␣the␣nearest␣obstacle␣on␣the␣front␣(up␣to␣4m)
    "back" : Distance from the nearest obstacle on the back (up to 4m)",
␣␣␣␣"right"␣:␣Distance␣from␣the␣nearest␣obstacle␣on␣the␣right␣(up␣to␣4m)"
```

```
    "left" : Distance from the nearest obstacle on the left (up to 4m)",
    "up" : Distance from the nearest obstacle above (up to 4m)",
}
```

Given the wide range of applications the Multiranger deck fits, we decided to implement a simple implementation of two use cases of this deck directly inside the module:

- Object Tracking
- Obstacle Avoidance

With the Object Tracking behavior, the drone searches for an object to track when it starts. As soon it detects an object inside a configurable range, that object becomes the guide to follow. The drone with this behavior will try to follow whenever the tracked object moves in space.

The velocity needs to change depending on the distance from the object. If the object is on the perimeter of the activation range, we need to move fast towards it, trying not to lose the tracking. On the other hand, when the object is closer to the safe range, the velocity can be slowed since the object is closer to the drone.

When the tracked object stops standing at a fixed point, the drone will get closer to a safe distance that will never be violated.

In summary, as we can see in Figure 7.1, the drone with an Object Tracking behavior has two ranges: The first range, which is the activation range, is used to determine what the tracked object is. The second range, the safety range, is used to prevent collision with the tracked object when it stands at a fixed point.

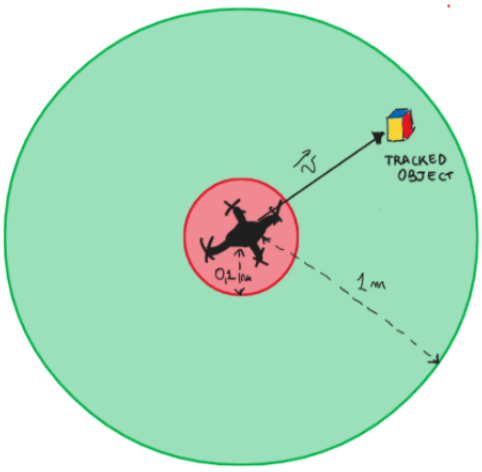


Figure 7.1: Multiranger Object Tracking behavior

The Obstacle Avoidance behavior is the opposite behavior with respect to object tracking. In this case, the activation range is used to determine an obstacle to fly away from.

When an obstacle is detected on the perimeter of the activation range, the drone will slowly start moving in the opposite direction. If the obstacle is dynamic and gets closer, the drone will increase its speed to try to escape from the obstacle.

In this behavior, the safety limit is used for landing the drone. If the obstacle gets too close, the only thing the drone can do to avoid the collision is landing.

7.4. Height Module

As we have seen in section3.2.2, both the Flow deck V2 and the ZRanger deck V2 mount a sensor that allows the drone to measure its height from the ground. Like the sensors equipped by the Multiranger deck, this sensor enables the height measurement of up to 4 meters, with very high precision. We designed the height module to manage the information provided by this sensor to simplify height-related tasks in drone applications.

The height module is an ECF module that manages the drone's height. It holds a simple state that consists of a single variable: the height.

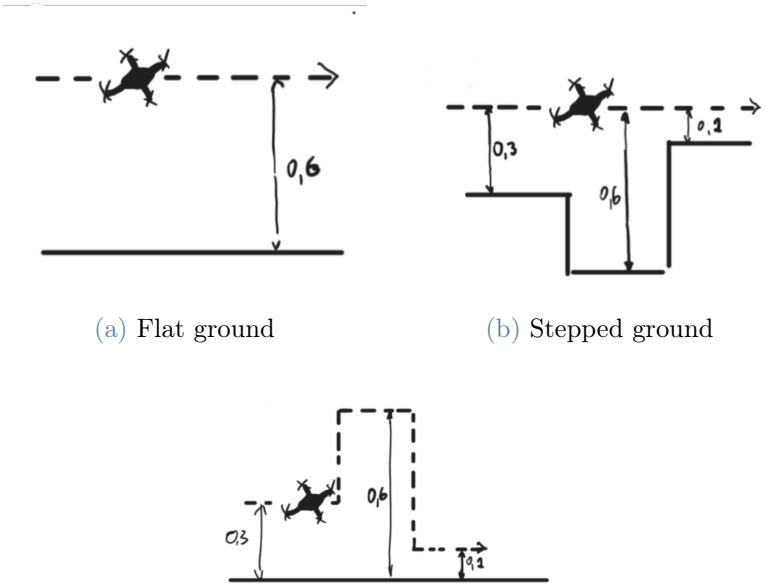
```
State: {
    "height" : "Height␣from␣the␣ground␣(up␣to␣4m)"
}
```

When either the Flow deck V2 or the ZRanger deck V2 are attached to the drone, the control loop will use the information provided by the sensor to estimate the internal state of the drone.

At first glance, this contribution seems valuable and harmless; in reality, this is true only when the ground surface is almost flat. When the ground surface is uneven, the measurement of the height sensor will act as a noise to the state estimation process, especially if the ground presents a stepped surface.

To better understand the uneven floor problem, let's consider the example shown in Figure 7.2.

In the first Figure 7.2a, the drone is flying over a flat ground, so the height sensor will contribute to the state estimate with the sensed value of 0.6 meters, which helps the State Estimator to provide a better estimate of the state (at least for the variable related with the height of the drone).



(a) Flat ground

(b) Stepped ground

(c) State estimator perspective of stepped ground

Figure 7.2: Uneven floor problem

In the second Figure 7.2b, the drone flies over a stepped ground. The height sensor will provide the State Estimator with a variable height by moving over this surface. The measured height will be either 0.6 or 0.1 meters, depending on the position.

Since the State Estimator is not aware of the conformation of the ground, the information perceived is the same as if the ground was flat and the drone was rapidly changing its height, as shown in Figure 7.2c. The rapid change in the state estimate is not expected and can produce some instabilities in the flight.

To better understand the phenomena, we run some experiments with the following settings: We placed a Crazyflie 2.1 with a height sensor attached in front of a cube-shaped obstacle with a height of 0.5 meters. We then write a script to take off the drone at 0.7 meters altitude and proceed above the obstacle. After passing the obstacle, the drone was supposed to land.

As expected, as soon as the drone flies over the obstacle, the height sensor deck passes from a measure of 0.7 meters to a measure of 0.2 meters. This (unwanted) rapid change always makes the drone crash.

To bypass this problem, we added a parameter that turns the height sensor's contribution to the state estimate on or off. This way, when the application's environment has flat ground, the developer can enable the contribution and better estimate the drone's height.

Conversely, when the ground is uneven, the developer can avoid the problem described above by simply disabling the contribution through the parameter.

To simplify the developer's life, we implemented a utility function inside the height module that allows setting the parameter with a simple function call.

7.5. Lighthouse Module

The Lighthouse ECF Module is the ECF Module associated with the Lighthouse Deck. As described in Section 3.4.2, the Lighthouse deck is a part of the absolute positioning system Lighthouse. The Lighthouse positioning system consists of two (or more) base stations emitting infrared light signals in a sweeping pattern. The Lighthouse deck picks up these signals, which are mounted on the Crazyflie 2.1 drone. The deck then uses the received signals to calculate its precise position and orientation in 3D space.

This Module does not have a meaningful state. All the information that would belong to its state is technical and not applicable to any application. Given this, the Lighthouse ECF module is an exception in the standard ECF Module implementation: it neither handles nor maintains a State.

A natural question can be: if the Lighthouse ECF Module does not hold a state, why is it useful? To answer this question, we need a little bit of background on the functioning of the Lighthouse positioning system: The Lighthouse positioning system is the most accurate and efficient positioning system that the whole Crazyflie platform provides. The problem related to this positioning system is the complex configuration that it needs to work.

The computation of the position in 3D space by knowing only the sweep angles of IR beams is not so direct. Indeed, to work correctly, the Lighthouse system needs a very accurate configuration based on multiple measurements taken in the flight space.

In fact, before using the Lighthouse, we need to estimate the environment's geometry. The system's geometry estimates the position and orientation of the two Lighthouse Base Stations with respect to the origin of the absolute positioning system. To perform this estimation, the developer needs to measure the Geometry in multiple points of the flight space, and then by averaging them, he can get the best possible estimate. The more accurate the Geometry estimation, the more precise the positioning system.

To answer the previous question, the Lighthouse ECF module exists and is extremely useful to overcome this limitation while using this positioning system. The Lighthouse

ECF Module is an ECF Module without state; it comprises numerous utility functions that help configure the Lighthouse positioning system.

The Lighthouse ECF Modules provide three main utility functions, each of which corresponds to a geometry estimation process:

- *simple_geometry_estimation*
- *multi_bs_geometry_estimation*
- *automatic_geometry_estimation*

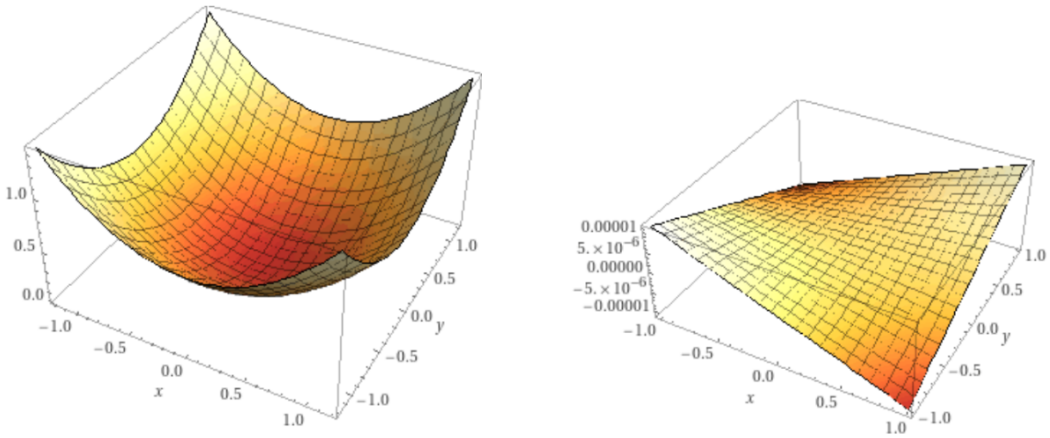
The first estimation process, *simple_geometry_estimation*, is a very primitive and quick approach to the Geometry estimation process. It consists of a single measurement in one position in the 3D space. When this utility function is called, the ECF Module asks Crazyflie 2.1 for a single estimation of the position and orientation of the Lighthouse Base Stations. When this information is received, the Module sets the measurement point as the origin of the 3D coordinate system. After that, it uploads this configuration back to Crazyflie 2.1.

A problem related to this type of estimation is that the error in the estimate is non-constant in the flight space. As evident from Figure 7.3a, it increases proportionally to the distance from the origin. In this case, the resulting position estimate in the 3D space around the origin would be accurate. Still, as far as we move from the origin, the error in the drone position estimate will increase, possibly resulting in instabilities or even crashes. This estimation process is intended only for quick tests where the drone position does not deviate too much from the origin.

To overcome this significant limitation, we developed a more accurate estimation process: *multi_bs_geometry_estimation*. As shown in Figure 7.3b, this estimation process tries to make the error in the estimate constant across the entire flight space. Moreover, this estimation method also supports the presence of multiple Lighthouse Base Stations (more than 2) that significantly increase flight space.

This estimation method consists of multiple executions of the first estimation process. All the recorded samples are averaged to build the most accurate geometry estimation sent to the Crazyflie 2.1.

The only drawback of this last estimation process is that it requires human help while doing the process. The problem is that the user moving the drone around the drone in the fourth phase is interfering with the estimation process. The user's body can block the IR beams from some base stations, invalidating lots of samples that otherwise would



(a) Error distribution in the flight space while using the *simple_ geometry_ estimation*

(b) Error distribution in the flight space while using the *multi_ bs_ geometry_ estimation*

Figure 7.3: Error distribution for Lighthouse

have been useful to compute a better geometry estimation.

The last estimation process, the *automatic_ geometry_ estimation*, is the automation of the previous process. The user can configure the execution as he wants, and then the ECF Module (with the help of the Flow Deck v2) performs the sampling process by autonomously moving the drone around the flight space.

The *automatic_ geometry_ estimation* is the most complex but most powerful estimation process.

7.6. AI deck Module

The AI Deck is an accessory board designed to provide on-board artificial intelligence (AI) capabilities to the Crazyflie 2.1, allowing it to perform advanced computational tasks without needing external processing.

The AI Deck is equipped with a powerful microcontroller unit (MCU) and a field-programmable gate array (FPGA), enabling high-performance computing and direct machine learning inference on the drone. The MCU on the AI Deck can run complex algorithms and execute machine learning models. The FPGA complements the MCU by accelerating certain computations and enhancing the overall processing capabilities.

The AI Deck can communicate directly with the ground station via WiFi thanks to the NINA unit mounted on the deck. The AI Deck, by default, runs a program on its MCU, which makes the video streaming of the camera mounted with the deck available on the

WiFi communication channel. For this work, we adopted this standard configuration of the AI Deck without implementing any algorithm for object tracking or anything similar.

The AI Deck ECF Module is an ECF module that allows one to connect, view, and run algorithms on the video streamed by the AI Deck itself. When this ECF module is detected and created, it automatically tries to connect to the WiFi network generated by the deck.

In particular, the Module consists of the following utility functions:

- *record*
- show_recording
- *run_ai*

The utility function *record* allows the user to record from the video stream a video in mp4 format. This functionality is significant when the user is in the developing phase of an AI algorithm. In this case, the user can record videos and test/train their models using a flight recording.

The show_recording utility simply pops up a video player with the last unsaved recording, or if the path to the filename is provided, it shows the recording located at that path.

When the developer has completed its AI algorithm, the user can use the *run_ai* utility to run that algorithm on the real-time streaming from the camera and perform the actions needed to accomplish the goal of the application to be developed.

The *run_ai* function takes in input the algorithm (a python function) and an arbitrary list of arguments that can be used to take the actions.

This function looks very similar to a Subscription in our Coordination Manager with an Action (the algorithm) and a Context (the arguments) to an Observable that maintains the video stream.

Even if we could use our Coordination Framework, we decided to keep this particular implementation outside of it on purpose for two main reasons: First, we did not want to overload the Coordination Framework with heavy data like images to make the complete state lighter. Secondly, we thought that a direct implementation of the *run_ai* function would have been much easier to use in this case with respect to the standard subscription mechanism across the Coordination Framework.

8 | Conclusions and future developments

The evaluation of a drone programming system is crucial to assess its effectiveness, performance, and suitability for specific applications. This chapter aims to provide a comprehensive evaluation of EasyFly. The baseline used to compare our programming environment is the basic Crazyflie python library (cflib)

We divided the evaluation process into two main topics: qualitative and performance analysis.

In the first step, we will give a qualitative analysis of EasyFly and an understanding of the strengths and leaks of our programming system. For this analysis, we participated in the challenges organized for the Digital Futures Drone Arena project [9]. During this challenge, groups of students and enthusiasts were asked to complete an obstacle run using Crazyflie 2.1 nano drones.

Then, we will move to performance analysis of EasyFly in a real example implementation, comparing it with a similar implementation written using only the core cflib. This evaluation aims to assess the ease of implementing a drone application with EasyFly and the consequent reduction in complexity in the scripts. On the other hand, we will demonstrate that the performance of an application developed with EasyFly is almost the same as the implementation using the core cflib offered by Crazyflie.

Given the high complexity of performance evaluation of drone applications, we designed and implemented a simulation environment for EasyFly. This simulation environment allowed us to conduct a precise performance analysis without deviating from the real environment's performance.

8.1. Qualitative Analysis

In the HDI domain, the research's core part, especially from the computer science perspective, is the experimental phase. During this phase, researchers put their ideas and prototypes to test and assess the practicality of innovative interaction models.

For a programming environment like EasyFly, testing and evaluating in a real research scenario in HDI is essential. The testing in real scenarios can help detect possible weaknesses in the programming environment, allowing for fine-tuning the model.

To best evaluate our EasyFly programming environment, we had the possibility to participate in the Digital Futures Drone Arena project [9].

The Drone Arena project planned two challenges where groups of students and enthusiasts with different backgrounds were asked to complete a specific task using drones. These challenges aim to allow researchers to collect data and perform studies about human-drone interactions.

The EasyFly programming environment was part of the inaugural challenge in Sweden in June 2022 [10]. In this challenge, five groups of students and enthusiasts with knowledge of computer science were asked to program Crazyflie 2.1 nano drones to complete an obstacle course. There were two main alternatives to write the scripts: the standard cflib or the EasyFly programming environment.

The challenge was distributed over three days; the first two days were dedicated to development and testing, and the last day was dedicated to the final challenge.

On the first day, we gave the groups a brief presentation on the basic knowledge of the Crazyflie 2.1 platform. We also described the main components and working principles of the EasyFly programming environment.

Then, we provided the groups with a Python project, where inside was the original cflib, our EasyFly programming environment, and a folder of examples for both. From then on, the groups were free to develop and test the code with real Crazyflie 2.1 in a sample stage, with minimum support from the research team.

This particular setting of the challenge allowed us to see and measure the impact of using EasyFly on a group with minimum knowledge of the Crazyflie platform. On the final day of the challenge, 2 out of 5 teams decided to participate in the final using our EasyFly programming environment.

The approach for all the groups was to use a Multiranger deck and perform a hand-driven

flight through the obstacle course. The main differences were how the groups implemented the mechanism that allowed them to push or pull the drone using their hands.

By analyzing the code that all the groups produced, we noticed two main aspects of the usage of the EasyFly programming environment: The groups that started using EasyFly from the beginning developed more complex solutions. The code of the groups that used the programming environment looked clean and simple, even if the solution they implemented was more complex.

Although we expected more adhesion on EasyFly, we noticed that almost all the groups had a look at the code of the programming environment, in particular inside the utility functions of the ECF modules. In their final code, almost every group used the utility functions of the Multiranger ECF Module, either directly or indirectly.

In conclusion, the challenge of the Drone Arena project allowed us to understand that the entire EasyFly programming environment, particularly the Coordination Manager, can seem too complicated at first glance and was a little more complex than we thought. The groups that decided not to use the programming environment decided on purpose to develop a more simple and sometimes effective solution.

8.2. Performance Analysis

EasyFly is a programming environment that aims to ease the development of drone applications, reaching the same solution with less effort, both in terms of time to write the solution and expertise required to implement the solution. Even if simplicity remains the first objective, performance plays an important role in the EasyFly programming environment. If the programming environment is easy to use but, in the end, results in inefficiency in terms of performance, it automatically becomes useless.

To assess the performance of EasyFly, we compared it with the standard implementation of Crazyflie's cflib across two real example applications used in HDI research. The first application scenario takes place in an artistic exhibition between a human performer and a drone. The second application scenario is an obstacle course similar to the Drone Arena inaugural challenge [10].

In this analysis, we will demonstrate that the expected loss in performance with respect to the basic cflib implementation is shallow compared to the reduction of program complexity.

8.2.1. EasyFly Simulation Environment

Measuring performance in drone applications is a very hard task, and usually, the performance of the applications can vary highly across different runs. When running the application in a real environment, the performance of the applications can usually vary highly across different runs. Given the fact that the conditions of the environment influence performance, the only possible strategy to measure performance is to execute and sample the same scenario multiple times and then average the results. Of course, this approach can have a high cost in terms of time and resources consumed.

To simplify the evaluation process and collect more precise data, we developed a simulation environment for EasyFly. In the simulation environment, the drones are replaced by a virtual copy that replicates the drone's movements in a controlled simulation. The simulation allows the reduction of the influence of the environment on performance measurements to zero, enabling precise performance measurement with a single run. This approach drastically reduces the costs of the analysis. A possible drawback of a poorly accurate simulation environment is that the performance can deviate from the real application performance.

In our simulation environment, we create a virtual Crazyflie 2.1 that emulates a real Crazyflie 2.1. Upon receiving the commands from the ground station, the virtual Crazyflie computes and updates its internal state (position, velocity, acceleration, and some sensor measures) and sends back data. In the current implementation of the EasyFly simulation environment, the virtual Craziflie emulates the control-related commands, logging, and parameter features.

As shown in Figure 8.1, the virtual Crazyflie communicates with the ground station's script using a UDP channel. Usually, the ground station script and the simulated Crazyflie run on the same machine, so the UDP communication is local.

The script of the ground station is entirely independent of Crazyflie's software. The execution and performance of the ground station's script are not affected by whether the Craziflie is virtual or not. This separation guarantees that the performance measured with EasyFly's simulation environment does not deviate from the real environment.

8.2.2. Application Scenario Evaluated

To best evaluate the performance of any system, it is crucial to select the appropriate application scenario. In particular, it is essential to test the system in a scenario that is as close as possible to an actual application of this system.

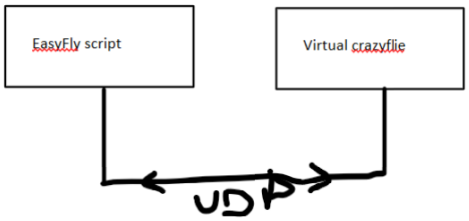


Figure 8.1: Structure of EasyFly simulation environment

Given that our programming environment is designed to build drone applications for HDI research, we selected two representative application scenarios from the literature on HDI.

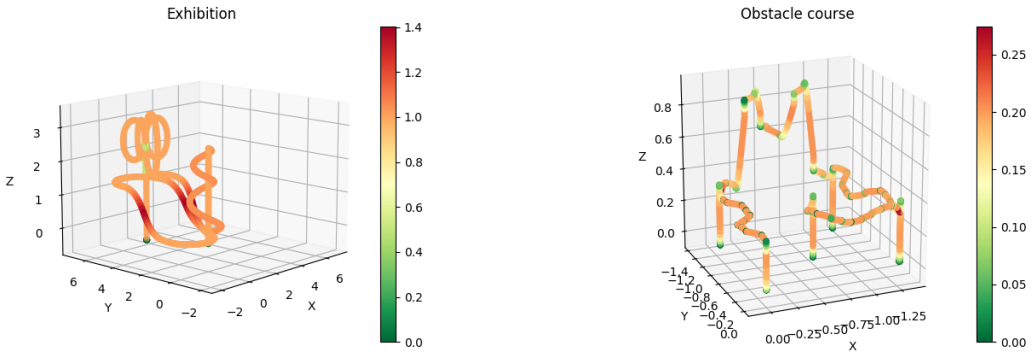
The first application takes inspiration from recent research conducted in the field of HDI [27]. This research explored how ethicality is shaped in the interaction between a choreographer, a performer, and a choir of five drones performing together on the opera stage. In our simplified version of this scenario, we reduced the drones to a single drone. Moreover, the choreography designed for the drone is determined before the performance. In other words, our drone should follow a predetermined path on the stage, exhibiting with the performer. The drone should also implement an obstacle avoidance mechanism to avoid any collision on the stage. In Figure 8.2a, we can see the trajectory followed by the virtual Crazyflie during the performance measurement of this application.

The second application we selected for performance analysis is an obstacle course similar to the one adopted for the inaugural challenge of the Digital Futures Drone Arena project [10]. In this application, we based the implementation of the algorithm on an elevation model of the environment. In other words, we modeled the obstacle course stage with a grid where every cell contains the elevation at that coordinate. We then implemented a script to navigate safely in the provided environment. We also adopted the obstacle avoidance mechanism in this application to avoid unpredicted crashes. In Figure 8.2b, we can see the trajectory followed by the virtual Crazyflie during the performance measurement of this application.

8.2.3. Tools and Metrics for Measuring Performances

As previously anticipated, our core objective is to produce a simple programming environment; given this, the most critical analysis is around the complexity of the applications. Another important aspect to keep under control is the execution performance

To evaluate our programming environment, we indeed targeted two main categories of performance metrics: complexity and execution performance.



(a) Frist application scenario: drone exhibition

(b) Second application scenario: drone obstacle course

Figure 8.2: Application scenarios' trajectories

The first category, complexity, is composed of a set of metrics that tries to evaluate the complexity of a program. The category is composed of the following metrics:

- Lines of Codes (LOC)
- Cyclomatic Complexity (CC)
- Halstead Metrics (HAL)

The LOC metric is the most simple measure of the complexity of a program. As the name suggests, it is the count of lines of code of the program under analysis.

The Cyclomatic Complexity is a Complexity analysis method proposed by Mccabe in 1996; it evaluates the complexity by measuring the number of linearly independent paths through a piece of code [33].

The Halstead Metrics was introduced by Maurice H. Halstead in the 1970s. They are based on the analysis of the number of unique operators and operands in a program, as well as their occurrences [30]. The five main metrics proposed by Halstead are:

- **Program Vocabulary** (η): The total number of unique operators and unique operands in the program. It is given by the formula: $\eta = \eta_1 + \eta_2$ where η_1 is the number of distinct operators, and η_2 is the number of distinct operands.
- **Program Length** (N): The total number of operator occurrences and operand occurrences in the program. It is given by the formula: $N = N_1 + N_2$, where N_1 is the total number of operators and N_2 is the total number of operands.
- **Program Volume** (V): A measure of the size of the program. It is given by the

formula: $V = N \log_2 \eta$.

- **Program Difficulty** (D): Represents the complexity of the program. It is given by the formula: $D = \frac{\eta_1}{2} \cdot \frac{N_2}{\eta_2}$.
- **Effort** (E): Represents the effort required to understand and develop the program. It is given by the formula: $E = D \cdot V$.

To collect all these measurements, we used *radon* [16], a Python library that computes all such metrics using static analysis on the source code of the scripts.

The second category of metrics, the execution performances, is a set of metrics that measures the resource consumption of the scripts. The category is composed of the following metrics:

- Memory (RAM) consumption
- Average CPU consumption
- Max CPU consumption
- Network load

These metrics provide valuable insights into the efficiency and resource utilization of the scripts. Memory consumption indicates how much random access memory is being used. Average CPU consumption gives an idea of the overall CPU usage over a period, max CPU consumption highlights the peak CPU usage, and network load measures the amount of data transmitted over the network during script execution.

Analyzing these metrics is crucial for identifying potential bottlenecks and ensuring resource utilization does not constrain our programming environment.

The machine used to run the performance analysis mounts an AMD Ryzen 7 5700U with Radeon Graphics with 8GB of RAM. To measure the memory consumption, we used *memory-profiler* [11], a Python library for collecting RAM usage while executing a Python script. We used the *Performance Monitor* tool for Windows for the CPU metrics. Finally, the simulation environment automatically computes the network load by summing all the CRTP received and sent.

8.2.4. Results Analysis

This section will present the complexity and performance analysis results aimed at evaluating our EasyFly programming environment.

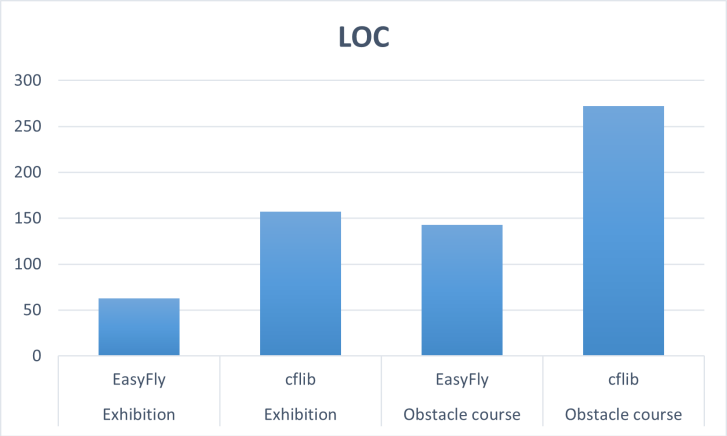


Figure 8.3: Lines of Code

As described in Section 8.2.2, we selected two application scenarios that best represent our target application. Since our primary goal is to produce a simple programming environment that researchers of HDI can use, this evaluation tries to asses the simplicity of the environment while not losing too much in performance.

As we have seen in Section 8.2.3, we divided the evaluation into two categories:

- Complexity analysis
- Performance analysis

The complexity analysis represents the core part of the evaluation and is composed of three metrics:

- Lines of Codes (LOC)
- Cyclomatic Complexity (CC)
- Halstead Metrics (HAL)

Looking at the results, it is crystal clear that our programming environment, EasyFly, is much less complex than the original cflib we selected as a baseline for comparison. Starting from the most straightforward metric, LOC, we can see in Figure 8.3 that the results are, in both scenarios, halving the count. EasyFly, with respect to cflib, allows you to write the same application with half of the code.

By looking at the second metric, CC, we can see, also in this case, the same trend. Moreover, when comparing the CC across the two different scenarios, it is evident that the performance gap between cflib and EasyFly increases with complexity. In Fact, in the second scenario, the obstacle course, the total CC computed is 50% less than the cflib's

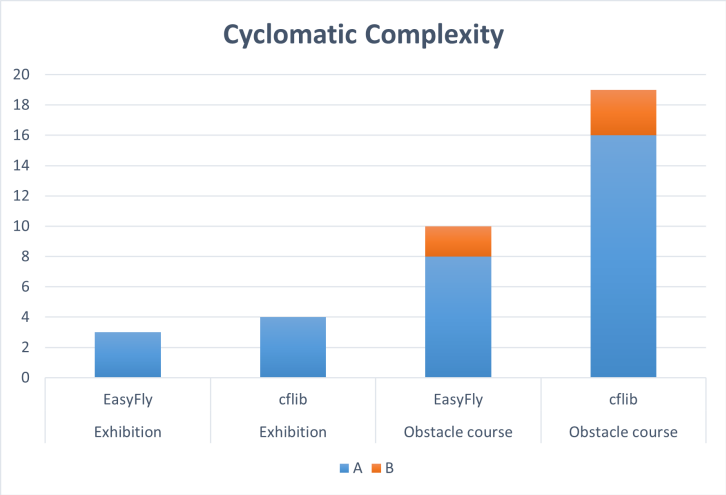


Figure 8.4: Cyclomatic Complexity

implementation.

The last metric for complexity, HAL, by looking at the results in Table 8.1, confirms what we previously said for LOC and CC. In particular, in this case, it is much more evident that the performance gap between cflib and EasyFly increases with complexity. Here, for the Effort (E), we observed a gap of over 55% for the second scenario.

In the second part of the evaluation, the performance analysis, we expect a slight loss in performance due to the overhead introduced by the EasyFly main components. In fact, by looking at Table 8.2, we can see that the CPU percentage utilization is generally less when using the standard cflib with respect to EasyFly.

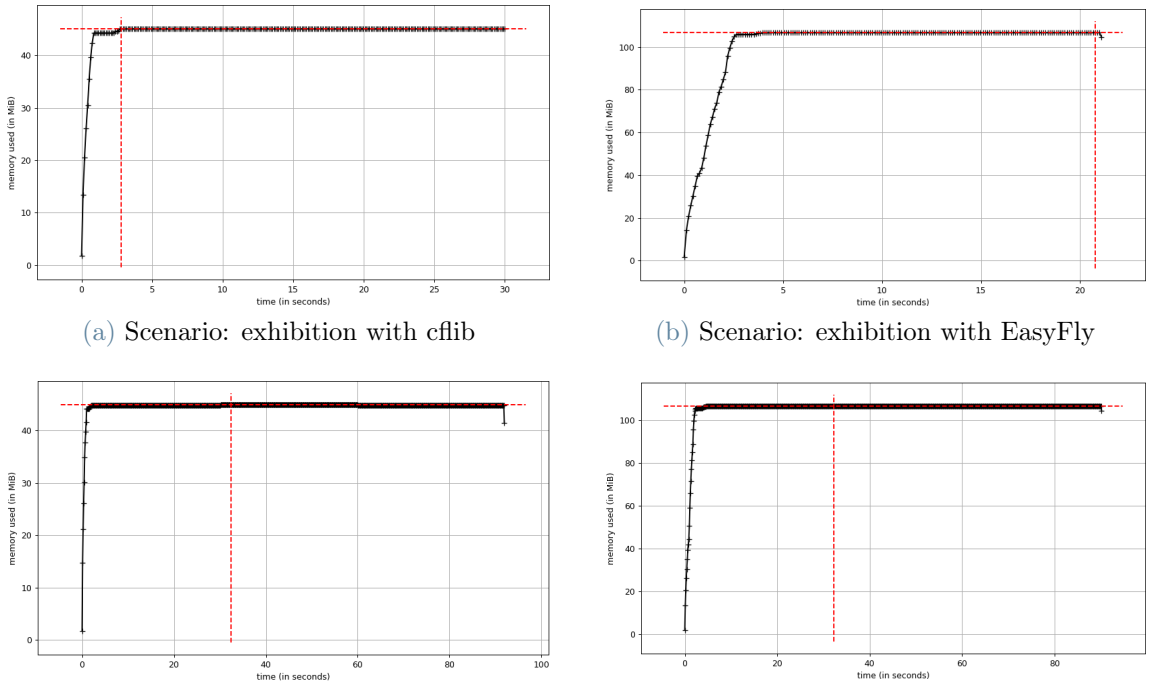
The reason for this increase in CPU utilization is that EasyFly handles most things automatically. An important observation needs to be around the average CPU consumption. In this measurement, we can see that the metrics for EasyFly are not so far from the standard cflib, and, in the first more straightforward scenario, it is also better. These metrics indicate that EsayFly is more flexible and better adapts resource utilization across different situations.

Scenario	η	N	V	D	E
Exhibition with cflib	33	45	226.99	4.04	916.72
Exhibition with EasyFly	38	48	251	3.0	755.70
Obstacle course with cflib	171	296	2195.68	8.74	19187.76
Obstacle course EasyFly	108	168	1134.82	7.66	8696.31

Table 8.1: Halstead Metrics

Scenario	CPU Average (%)	CPU MAX (%)
Exhibition with cflib	0.570	12.829
Exhibition with EasyFly	0.266	16.449
Obstacle course with cflib	0.540	18.423
Obstacle course EasyFly	0.625	22.340

Table 8.2: CPU consumption



(a) Scenario: exhibition with cflib

(b) Scenario: exhibition with EasyFly

(c) Scenario: obstacle course with cflib

(d) Scenario: obstacle course with EasyFly

Figure 8.5: Memory (RAM) usage

The memory consumption metric is a weakness for EasyFly. In particular, it is clear from the results shown in Figure 8.5 that the standard cflib is consuming less than 50% of EasyFly for the same application scenario. Also in this case, the overhead is due to all the infrastructure needed to manage most of the tasks automatically. Even if the memory consumption metric is a weakness for EasyFly, we can anyway learn two important facts: The first is that memory consumption is in the order of hundreds of MiB; hence, any modern machine can handle such memory load with any problem. The second fact is that by increasing the complexity of the application scenario, the gap between the cflib and EasyFly is constant; this suggests that the memory consumption will never drift from the performance of the standard cflib.

The last metric for the performance evaluation is the network load. As previously de-

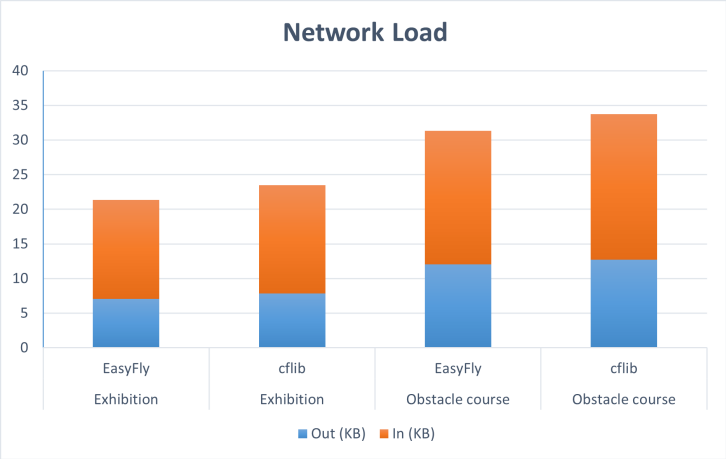


Figure 8.6: Network load

scribed, the simulation environment helped greatly in retrieving such metrics. In particular, the simulation execution automatically computes the incoming and outgoing network load. In Figure 8.6, we can see the result of this computation, where we pointed out the outgoing and the incoming traffic load in KB for every application scenario. The point of view of the measurement is that of the ground station. Out-packages have the direction from the ground to the drone, and In-packages follow the other direction. Thanks to the optimization made by the Communication Framework hosted on EasyFly, we can observe a slightly better result for the network load metric. In particular, Easyfly automatically optimizes the logging task, reducing the cost of communication.

In conclusion, as expected, we have observed that our programming environment, EasyFly, is significantly reducing the complexity of its scripts; On the other hand, the performance metrics are kept on the same level with respect to the standard cflib.

8.2.5. Future Developement

This thesis work mainly focused on drone application with a single drone deployment. This limitation clearly does not fit modern HDI research's needs.

For this reason, a possible future development around the EasyFly programming environment is the possibility of deploying multiple drones without introducing useless complexity in the development phase. In the actual implementation of the standard cflib there is a possibility of running scripts for swarm of drones.

EasyFly should extend this basic feature by integrating it within its Coordination Manager, allowing for inter-drone communication.

Another key point of future development should be around the simulation environment. For the purpose of this work, we used the simulation environment mainly to evaluate performance. In our opinion, allowing the user to leverage the potentiality of such a system should be critical.

Testing with drones is costly; hence, having the possibility to rely on a solid simulation environment is essential. A future version of our simulation environment should include the possibility of utilizing all the available sensors inside the simulation. Moreover, multiple drone deployments should be allowed inside the simulation to support swarm programming.

Bibliography

[1] 3dr. URL http://3dr.com/.

[2] Dij flight simulator. URL https://www.dji.com/it/simulator.

[3] Bitcraze. URL https://www.bitcraze.io/.

[4] Crazyflie 2.1, . URL https://www.bitcraze.io/products/crazyflie-2-1/.

[5] crazyflie-firmware, . URL https://github.com/bitcraze/crazyflie-firmware.

[6] crazyflie-lib-python, . URL https://github.com/bitcraze/crazyflie-lib-python.

[7] Cubepilot. URL https://www.cubepilot.com/.

[8] Dij. URL https://www.dji.com.

[9] Drone arena, . URL https://www.dronearena.info/.

[10] Drone arena, . URL https://www.dronearena.info/?page_id=25.

[11] Memory profiler. URL https://pypi.org/project/memory-profiler/.

[12] Navio2 emlid. URL https://docs.emlid.com/navio2/.

[13] Parrot. URL https://www.parrot.com.

[14] Pixhawk. URL https://pixhawk.org/.

[15] Px4 autopilot. URL https://px4.io/.

[16] Radon. URL https://radon.readthedocs.io/en/latest/index.html.

[17] Sphinx. URL https://developer.parrot.com/docs/sphinx/index.html.

[18] Wing drones. URL https://wing.com/.

[19] C. Anderson. How i accidentally kickstarted the domestic drone boom. *Wired Magazine*, 22:2012, 2012.

[20] M. P. Ashley-Rollman, S. C. Goldstein, P. Lee, T. C. Mowry, and P. Pillai. Meld: A declarative approach to programming ensembles. In *2007 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 2794–2800. IEEE, 2007.

[21] J. Bachrach, J. Beal, and J. McLurkin. Composable continuous-space programs for robotic swarms. *Neural Computing and Applications*, 19:825–847, 2010.

[22] E. Bonabeau, M. Dorigo, and G. Theraulaz. *Swarm intelligence: from natural to artificial systems*. Number 1. Oxford university press, 1999.

[23] S. Card, T. MORAN, and A. Newell. The model human processor- an engineering model of human performance. *Handbook of perception and human performance.*, 2 (45–1), 1986.

[24] J. R. Cauchard, J. L. E, K. Y. Zhai, and J. A. Landay. Drone & me: an exploration into natural human-drone interaction. In *Proceedings of the 2015 ACM international joint conference on pervasive and ubiquitous computing*, pages 361–365, 2015.

[25] K. Dantu, B. Kate, J. Waterman, P. Bailis, and M. Welsh. Programming micro-aerial vehicle swarms with karma. In *Proceedings of the 9th ACM Conference on Embedded Networked Sensor Systems*, pages 121–134, 2011.

[26] A. Dix. Human–computer interaction: A stable discipline, a nascent science, and the growth of the long tail. *Interacting with computers*, 22(1):13–27, 2010.

[27] S. Eriksson, K. Höök, R. Shusterman, D. Svanes, C. Unander-Scharin, and Å. Unander-Scharin. Ethics in movement: Shaping and being shaped in human-drone interaction. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*, pages 1–14, 2020.

[28] D. Feil-Seifer and M. J. Mataric. Human robot interaction. *Encyclopedia of complexity and systems science*, 80:4643–4659, 2009.

[29] E. Graether and F. Mueller. Joggobot: a flying robot as jogging companion. In *CHI'12 Extended Abstracts on Human Factors in Computing Systems*, pages 1063–1066. 2012.

[30] T. Hariprasad, G. Vidhyagaran, K. Seenu, and C. Thirumalai. Software complexity analysis using halstead metrics. In *2017 international conference on trends in electronics and informatics (ICEI)*, pages 1109–1113. IEEE, 2017.

[31] M. Hoppe, M. Burger, A. Schmidt, and T. Kosch. Droneos: A flexible open-source

prototyping framework for interactive drone routines. In *Proceedings of the 18th International Conference on Mobile and Ubiquitous Multimedia*, pages 1–7, 2019.

[32] K. LaFleur, K. Cassady, A. Doud, K. Shades, E. Rogin, and B. He. Quadcopter control in three-dimensional space using a noninvasive motor imagery-based brain–computer interface. *Journal of neural engineering*, 10(4):046003, 2013.

[33] T. Mccabe. Cyclomatic complexity and the year 2000. *IEEE Software*, 13(3):115–117, 1996.

[34] L. Mottola, M. Moretta, K. Whitehouse, and C. Ghezzi. Team-level programming of drone sensor networks. In *Proceedings of the 12th ACM Conference on Embedded Network Sensor Systems*, pages 177–190, 2014.

[35] F. Nex and F. Remondino. Uav for 3d mapping applications: a review. *Applied geomatics*, 6:1–15, 2014.

[36] F. Nex and F. Remondino. Uav for 3d mapping applications: a review. *Applied geomatics*, 6:1–15, 2014.

[37] G. Pantelimon, K. Tepe, R. Carriveau, and S. Ahmed. Survey of multi-agent communication strategies for information exchange and mission control of drone deployments. *Journal of Intelligent & Robotic Systems*, 95:779–788, 2019.

[38] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, A. Y. Ng, et al. Ros: an open-source robot operating system. In *ICRA workshop on open source software*, volume 3, page 5. Kobe, Japan, 2009.

[39] A. Sharma, P. Vanjani, N. Paliwal, C. M. W. Basnayaka, D. N. K. Jayakody, H.-C. Wang, and P. Muthuchidambaranathan. Communication and networking technologies for uavs: A survey. *Journal of Network and Computer Applications*, 168:102739, 2020.

[40] K. Shilton et al. Values and ethics in human-computer interaction. *Foundations and Trends® in Human–Computer Interaction*, 12(2):107–171, 2018.

[41] S. R. R. Singireddy and T. U. Daim. Technology roadmap: Drone delivery–amazon prime air. *Infrastructure and Technology Management: Contributions from the Energy, Healthcare and Transportation Sectors*, pages 387–412, 2018.

[42] G. Sinha, R. Shahi, and M. Shankar. Human computer interaction. In *2010 3rd International Conference on Emerging Trends in Engineering and Technology*, pages 1–4. IEEE, 2010.

[43] D. Tezza and M. Andujar. The state-of-the-art of human–drone interaction: A survey. *IEEE Access*, 7:167438–167454, 2019.

[44] H. A. Yanco and J. Drury. Classifying human-robot interaction: an updated taxonomy. In *2004 IEEE international conference on systems, man and cybernetics (IEEE Cat. No. 04CH37583)*, volume 3, pages 2841–2846. IEEE, 2004.

List of Figures

List of Tables

Acknowledgements

Here you might want to acknowledge someone.