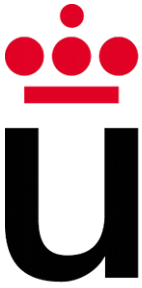


Universidad
Rey Juan Carlos

GRADO EN INGENIERÍA AEROESPACIAL EN AERONAVEGACIÓN

Curso Académico 2022/2023

Trabajo Fin de Grado

Exploration of vision-based control solutions for PX4-driven UAVs

Autora : Laura González Fernández
Tutores : Xin Chen, Alejandro Sáez Mollejo

Trabajo Fin de Grado

Exploration of Vision-Based Control Solutions for PX4-Driven UAVs.

Autora : Laura González Fernández
Tutores : Xin Chen, Alejandro Sáez Mollejo

La defensa del presente Proyecto Fin de Grado/Máster se realizó el día 3 de
de 20XX, siendo calificada por el siguiente tribunal:

Presidente:

Secretario:

Vocal:

y habiendo obtenido la siguiente calificación:

Calificación:

Móstoles/Fuenlabrada, a de de 20XX

Aquí normalmente
se inserta una dedicatoria corta

Acknowledgements

Aquí vienen los agradecimientos...

Hay más espacio para explayarse y explicar a quién agradeces su apoyo o ayuda para haber acabado el proyecto: familia, pareja, amigos, compañeros de clase...

También hay quien, en algunos casos, hasta agradecer a su tutor o tutores del proyecto la ayuda prestada...

Resumen

Aquí viene un resumen del proyecto. Ha de constar de tres o cuatro párrafos, donde se presente de manera clara y concisa de qué va el proyecto. Han de quedar respondidas las siguientes preguntas:

- ¿De qué va este proyecto? ¿Cuál es su objetivo principal?
- ¿Cómo se ha realizado? ¿Qué tecnologías están involucradas?
- ¿En qué contexto se ha realizado el proyecto? ¿Es un proyecto dentro de un marco general?

Lo mejor es escribir el resumen al final.

Abstract

The popular open-source platform PX4 aims to facilitate the programming of unmanned aerial vehicles and their integration with new sensors and actuators and make it approachable for the common developer. This thesis aims to demonstrate how this platform can be used to develop solutions that integrate computer vision techniques and use their input to control the movement of an aerial vehicle, while employing easily-available and affordable hardware with basic specifications. For this purpose, a viable solution is presented that allows a drone to use an onboard camera to identify and keep track of a person in its field of view to follow their movement.

Todo list

Contents

List of Figures

Chapter 1

Introduction

1.1 General context

Unmanned Aerial Vehicles (UAVs), commonly known as drones, have emerged as remarkable aircraft capable of flying without a pilot or operator on board. Originally developed as military technology to safeguard pilots from dangerous missions, UAVs have transitioned into the civilian domain due to advancements in control technologies and the decreasing costs of electronics and sensors. This has led to broader accessibility of UAVs, allowing their utilization in diverse civil applications such as crop monitoring, search and rescue operations, and filmmaking and photography. Traditionally, UAVs have been operated either remotely by a human operator or with limited autonomy through preplanned missions. However, recent trends have indicated a shift towards vision-based control solutions, which offer promising advantages over conventional control methods. Vision-based control harnesses the power of cameras and image processing algorithms to provide real-time feedback and precise control of UAVs.

Compared to traditional control systems that heavily rely on sensors such as accelerometers and gyroscopes, vision-based control solutions offer greater flexibility and robustness. By leveraging the visual input from cameras, these solutions can adapt to varying environmental conditions and handle complex flight scenarios more effectively. Furthermore, vision-based control methods are often more accessible, as they can be implemented on lower-cost platforms, making them appealing to a wider range of users. While the integration of vision-based control solutions into UAVs is a relatively recent development, propelled by the increasing prevalence of artificial intelligence, there is limited availability of fully-developed consumer-ready platforms in the market. However, this does not impede progress in this field, as all the necessary components for constructing such systems are readily available. From the essential hardware required to build a quadcopter, which represents the most common type of UAV, to miniaturized

computers capable of handling complex calculations, and open-source software that can be customized for endless applications, the individual pieces required for constructing and implementing vision-based control solutions are very accessible.

In summary, the rise of UAVs and their integration into various civil applications has paved the way for the exploration of vision-based control solutions. These solutions, utilizing cameras and image processing algorithms, offer enhanced performance, adaptability, and accessibility compared to traditional control methods. Although consumer-ready platforms are still emerging, the necessary components for constructing vision-based control systems are readily available, fueling the advancement of this field.

1.2 The DroneVisionControl project

The project outlined in this thesis is a comprehensive exploration of the tools available for designing and implementing control solutions for the PX4 open-source autopilot platform. The goal is to demonstrate how these tools can be integrated with computer vision mechanisms to achieve vision-based self-guided unmanned aerial vehicles (UAVs). To achieve this objective, a simple application has been developed named Drone Vision Control serving as a foundation for the investigation into the PX4 ecosystem and methods of integrating the necessary hardware and software for UAV control. For the computer vision detection mechanisms, the ready-to-use Mediapipe suite of libraries for real-time image analyses on hand-held devices has been employed to reduce the complexity of the project. Two primary architectures for integrating software and hardware in vision-based scenarios have been proposed, with one scenario involving off-board vision modules and the other integrating onboard camera and vision computation.

The first scenario involves a proof-of-concept control mechanism that translates hand gestures executed in front of an independent camera to flight commands, thereby controlling the movement of the vehicle. This serves as a landing platform to begin development and test the most basic integration between the tools employed. The second scenario builds on the first to integrate the camera and vision computation onboard the vehicle to develop a tracking and following control solution that can track and follow a person standing in the drone's field of view by mirroring their movements in a 3D environment.

To further the development of these control mechanisms, a dedicated development environment was employed. This environment facilitated the design process by allowing a gradual adaptation of the software to match the hardware requirements, starting from a purely simulated flight controller and progressing towards real flight tests in an unmanned vehicle. The key component of this environment is the integration with the AirSim simulation engine. Built upon the powerful 3D computer graphics game engine Unreal,

it provides a platform that ensures secure and comprehensive testing of control solutions that are still in progress and not yet fully reliable.

For this project, all the code has been implemented in Python, running on a companion computer separate from the flight controller board. This deliberate approach ensures that the control solutions remain independent of the hardware components and are not tied to any particular flight stack. Consequently, the results obtained can be applied to any autopilot platform that offers exposed APIs and are not limited to the PX4 ecosystem.

In conclusion, this thesis offers a thorough investigation into the development of vision-based self-guided UAVs, presenting a range of tools and techniques that can be utilized to accomplish this objective.

1.3 Objectives

The main objective of this project is to explore the tools and methods available for designing and implementing control solutions for the PX4 open-source autopilot platform and demonstrate how they can be integrated with computer vision mechanisms to achieve a self-guided UAV that tracks and follows a person's movements.

More specifically, it aims to:

- Research and understand the PX4 platform and its wider ecosystem, including its architecture, capabilities, and available APIs.
- Explore existing computer vision mechanisms and algorithms suitable for vision-based control of UAVs.
- Design and set up a dedicated development environment using the AirSim simulation engine for secure and comprehensive testing of control solutions.
- Employ both software-in-the-loop and hardware-in-the-loop simulation approaches to comprehensively evaluate the control solutions in the simulation engine.
- Develop a proof-of-concept control mechanism that translates hand gestures captured by a camera into flight commands for the UAV.
- Integrate the hand gesture control mechanism with the PX4 platform and test its functionality in a simulated environment.
- Investigate hardware and software requirements for integrating onboard cameras and vision computation in UAVs.

Figure 1.1: Timeline for the development of the project

- Develop a tracking and following control solution that can mirror the movement of a person.
- Test the tracking and following control solution in a simulated environment, integrating the onboard camera and vision computation with the PX4 platform.
- Conduct flight tests on a self-built quadcopter to evaluate the developed control mechanisms, analyzing their performance and reliability in different scenarios.
- Document the findings, methodologies, and results obtained throughout the project and provide recommendations for future improvements.

1.4 Time planning

This project was carried out over a period of approximately one and a half years, taking into account the parallel commitment of working a full-time job. The development process was divided into three distinct phases, totalling approximately 400 hours of dedicated effort. These phases consisted of the following tasks:

- **Phase 1: Proof-of-concept (Oct 2021 - Jan 2022: 61 hours)**
  - Conducted research on the PX4 system (12 hours)
  - Programmed the hand solution (42 hours)
  - Tested the standard drone build (7 hours)
- **Phase 2: Follow solution (Feb 2022 - Aug 2022: 138 hours)**

  - Conducted research on tracking and detection methods (33 hours)
  - Developed the follow solution (80 hours)
  - Tested custom hardware (25 hours)

- **Phase 3: Hardware (Sep 2022 - Feb 2023: 65 hours)**
  - Conducted research on hardware options (15 hours)
  - Developed test tools and refined code (33 hours)
  - Conducted testing of custom hardware and integration (17 hours)

During the initial phase, extensive research was conducted to gain a deep understanding of the PX4 system. This involved studying the architecture, capabilities, and documentation of the platform. Following the research phase, a proof-of-concept control mechanism was developed, enabling the translation of hand gestures captured by an independent camera into flight commands. Subsequently, testing was performed on the standard drone build to verify the basic integration between the control mechanism and the UAV platform.

The second phase focused on the development of an advanced tracking and following control solution. Research was conducted to explore various tracking and detection methods, including computer vision algorithms and techniques. Building upon this research, the follow solution was implemented, enabling the UAV to track and follow a person in its field of view by mirroring their movements in a 3D environment. Custom hardware components, such as sensors and actuators, were tested to ensure compatibility and reliable operation within the follow solution.

In the third phase, research was conducted to explore hardware options for the UAV system, including flight controllers, companion computers, and cameras. The chosen hardware components were integrated, and test tools were developed to enhance the performance and reliability of the control mechanisms. The system's hardware and software integration was thoroughly tested to ensure seamless operation and optimize the control algorithms.

Additionally, a significant amount of time, totalling 133 hours, was dedicated to writing the report, which was carried out concurrently with the research and development work.

This time planning and task distribution allowed for a structured approach to the project, ensuring a comprehensive exploration of the subject matter while accommodating other commitments.

1.5 Thesis layout

The thesis is structured into five chapters, with the three main blocks focusing on each of the three specific aspects of the project mentioned in the previous section: research, development and testing.

The first chapter serves as an introduction to the context in which the project has been developed, providing background information and highlighting the objectives pursued throughout the research. It sets the stage for the subsequent chapters, giving an overview of the project's scope and purpose.

The second chapter discusses the technologies and tools employed in the project and provides a review of the relevant literature and the current state-of-the-art for UAV vision-based control solutions.

The third chapter covers the simulation environments used throughout the project and the architecture of the hardware and software employed in the project. This chapter discusses the design decisions that went into selecting the hardware and software components and describes the overall architecture of the system.

The fourth chapter presents the testing methodology used throughout the project, from initial simulations to flight tests. This chapter provides a detailed account of the testing process and highlights the key findings and insights gained from each phase of testing.

The final chapter concludes the thesis by summarizing the key findings of the project and drawing conclusions about the effectiveness and limitations of the control system. Additionally, this chapter provides suggestions for future development and improvement of the system.

Chapter 2

State of the art

2.1 Literature review

Vision-based control solutions for UAVs on low-cost platforms have garnered significant attention in recent years, driven by the growing demand for cost-effective and efficient aerial applications. This literature review provides an overview of the existing research in this field, highlighting key findings and challenges.

A substantial body of research has focused on developing real-time image processing algorithms to enhance the accuracy and robustness of tracking solutions for UAV applications. For instance, in [1], a computer vision algorithm utilizing optical flow is proposed for tracking ground-moving targets, enabling position measurement and velocity estimation. Similarly, [2] addresses the problem of tracking and following generic human targets in natural, possibly dark scenes without relying on colour information. In [3], several algorithms are developed to create path-following mechanisms that maintain a constant line of sight with the target. These studies attempt to develop solutions that can be applied to any platform regardless of any preexisting autopilot control in the UAVs. They focus on low-level software implementations of complex control theory topics with advanced mathematics. In contrast, this thesis aims to abstract control techniques and computer vision mechanisms to present an easy-to-use platform that combines existing algorithms for robust systems with a lower threshold of knowledge, focusing on the integration between software and hardware.

Due to the fact that the PX4 platform has been available for less than a decade, and despite its recognition as a widely-used standard in the drone industry since 2020, there is limited research done on this ecosystem, and few computer vision projects have focused specifically on this platform. However, some studies demonstrate the possibilities offered by the PX4 software and the Pixhawk flight controllers developed for it. In [4], aerial

images are analyzed in real-time and fed to a deep learning architecture to calculate optimal flight paths. [5] focuses on developing a vision-based precision landing method for quadcopter drones using the Pixhawk flight controller to prevent crashes during landing.

In contrast, other comparable accessible platforms that have been available for a longer period have been explored further on vision-based control projects. Specifically, the Parrot *AR.Drone* camera-enabled quadcopter, which exposes an API that allows control through WiFi from an external offboard computer is the most widely used. In [6], [7], [8], and [9], the *AR.Drone* is utilized for implementing object tracking and following solutions in various environments and conditions, with a particular emphasis on the type of trackers employed to achieve a robust visual following mechanism. In [10], [11], and [12], researchers focus on leveraging the capabilities of the *AR.Drone* as a platform to develop custom control solutions, utilizing embedded navigation and control technology along with low-cost sensors. It is worth noting that the key distinction between the platform used in the mentioned research and the PX4 platform targeted in this thesis is that the *AR.Drone* is offered as a complete low-cost vehicle for developing autonomous guidance and control, whereas the PX4 platform provides a comprehensive environment allowing for enormous customization at each layer of the system.

Another significant advantage of the PX4 platform over other available platforms is its compatibility with a wide range of simulators, enabling the development of complex control systems in diverse environments. This reduces the need for expensive, time-consuming, and meticulous real-world flight testing. Several studies have explored this aspect of the PX4 ecosystem. For example, [13] employs the Gazebo simulator and the PX4 software to develop a system for simulating realistic navigation conditions for obstacle avoidance. In [14], a ROS-Gazebo environment serves as the foundation for designing fault-tolerant controllers capable of recovering from rotor failure. Moreover, [15] aims to integrate a photo-realistic environment simulator with a flight dynamics simulator to achieve full autonomy in the Pixhawk autopilot board. These examples demonstrate the potential applications of the comprehensive PX4 ecosystem for developing control algorithms.

The literature reviewed indicates that the PX4 platform offers unique advantages, such as its extensibility, compatibility with a wide range of simulators, and the ability to personalize each layer of the system. While previous research has primarily focused on low-cost platforms like the *AR.Drone*, there is a clear gap in the exploration of the PX4 ecosystem and its potential for vision-based control solutions.

The studies mentioned in the literature review highlight various applications of vision-based control on the PX4 platform, such as optimal flight path calculation, precision landing, obstacle avoidance, and fault-tolerant control. However, these examples only scratch the surface of what can be achieved with the comprehensive PX4 ecosystem.

Considering the relatively recent emergence of the PX4 platform as a widely-recognized

standard, there is ample room for further research and development. The simulator capabilities of the platform provide a valuable opportunity to explore and refine vision-based control algorithms in a virtual environment, enabling more efficient and cost-effective system development.

By leveraging the PX4 ecosystem, researchers can focus on abstracting complex control techniques and computer vision mechanisms, making them more accessible to a broader range of users. This approach facilitates the development of easy-to-use platforms that integrate existing algorithms and combine them to create robust vision-based control systems. The goal is to lower the knowledge threshold required to implement such systems and enable wider adoption of vision-based control solutions for UAVs.

In conclusion, vision-based control solutions remain a relatively new field within the broader topic of autonomous guidance and navigation for UAVs. While some older low-cost platforms have been extensively researched as the basis for accessible computer vision-driven control systems, there are still numerous unexplored possibilities for applying the simulator capabilities of the PX4 platform to develop vision-based control solutions and leverage modern rendering techniques for simulating complex detection and tracking scenarios. This approach reduces the reliance on demanding real-world flight tests.

2.2 Methodology

This section describes the software programs and libraries employed throughout the development of this project, as well as the hardware used to test the application created.

2.2.1 Software

PX4 autopilot

PX4[1] is a widely-utilized autopilot flight stack designed for unmanned aerial vehicles (UAVs). Developed collaboratively by industry and academic experts, this open-source software benefits from an active global community, ensuring continuous improvements. It has been implemented in C++, a reliable and efficient programming language that gives it versatility and adaptability.

PX4 caters to various vehicle types, including racing drones, cargo drones, ground vehicles, and even submersibles. It accommodates both pre-built and custom-made drones,

[1] https://px4.io/

allowing developers to tailor their UAVs to specific requirements. Furthermore, PX4 supports integration with a range of sensors and peripherals, such as GPS, cameras, obstacle sensors, and more. This flexibility enhances UAV capabilities, ensuring efficient and safe operation in diverse environments. PX4's significance extends within the Dronecode Project[2], a comprehensive drone platform. This project includes essential components like the user-friendly QGroundControl ground station and the reliable Pixhawk hardware, known for its compatibility with PX4. The integration is further facilitated by the MAVSDK library, enabling seamless connections with companion computers, cameras, and additional hardware using the MAVLink protocol. This cohesive ecosystem empowers developers to leverage PX4's full potential and create innovative UAV solutions tailored to specific needs.

Central to PX4's functionality are its flight modes. These modes determine the autopilot's response to user commands and its management of autonomous flight. Offering various levels of assistance, flight modes simplify tasks like takeoff and landing while enabling precise control over flight trajectories and the maintenance of stable positions. PX4's adaptable flight modes ensure flexibility and reliability across a wide range of applications, from capturing aerial photographs to executing intricate autonomous missions.

In the context of this project, PX4 has played a pivotal role as the core component powering various aspects of the system. It serves as the heart of the simulation environment, replicating real-world flight conditions and interactions in a safe and cost-effective manner. Moreover, PX4 acts as the key interface between the high-level commands generated by the developed application and the engine outputs required for precise control of the UAV. It effectively translates the intentions and directives of the application into concrete actions performed by the engines, propellers, and other flight control surfaces, while taking care of maintaining flight stabilization. It implements sophisticated control algorithms and flight dynamics models to maintain stability, responsiveness, and safety during flight operations, thereby freeing the vision-based control application from focusing on low-level implementation.

MAVLink and MavSDK

MAVLink[3] is a lightweight messaging protocol designed as an integral part of the Dronecode Project. In the context of a UAV driven by the PX4 autopilot, the MAVLink protocol plays a crucial role as a standardized communication framework. It enables seamless data exchange between the autopilot and various onboard components, allowing the transmission of telemetry data, commands, and status updates. MAVLink ensures interoperability and facilitates integration with custom-built or third-party components,

[2] https://www.dronecode.org/
[3] https://mavlink.io/en/

enhancing the UAV's capabilities. By providing a reliable and efficient communication interface, MAVLink contributes to safe and coordinated flight operations, empowering developers to create sophisticated UAV systems with the PX4 autopilot at their core.

MAVSDK[4], on the other hand, is a cross-platform collection of libraries that enables seamless integration with MAVLink systems such as drones, cameras, and ground systems. The library handles the underlying MAVLink messaging protocol, abstracting the complexity of message parsing and transmission. Developers can focus on the logic and commands they want to send to the autopilot without worrying about low-level communication details. These libraries offer a user-friendly API for managing one or multiple vehicles, granting programmatic access to crucial vehicle information, telemetry, and control over missions, movements, and other operations. They can be utilized either onboard a drone's companion computer or on the ground via a ground station or mobile device, offering flexibility and convenience in system management.

While primarily implemented in C++, MAVSDK provides wrappers for Swift, Python, Java, and other languages. This project will employ the Python version of the library, allowing the developed application to send high-level commands and receive telemetry information from the autopilot.

QGroundControl

QGroundControl[5] is an open-source ground control station developed by the Dronecode Project, serving as a crucial software component for managing and controlling MAVLink-enabled drones. Its intuitive interface and user-friendly design cater to both professionals and developers. QGroundControl seamlessly connects with PX4 Autopilot flight controllers via wired or wireless connections, facilitating communication with local simulators running the PX4 flight stack.

With QGroundControl, users can efficiently configure and calibrate their flight controllers to ensure optimal performance. It offers a streamlined approach to modifying and tracking configuration parameters, allowing for precise customization of drone systems. Additionally, QGroundControl allows sending essential flight commands such as arming, takeoff, and landing and grants precise control over the drone's movements.

One of the key features of QGroundControl is its map interface, which provides real-time GPS location visualization of the drone. Target waypoints can be defined on the map to plan flight missions, specifying desired speeds and altitudes for each location. This functionality enables the creation of automated flight paths and streamlines mission planning and execution. Figure 2.1 showcases the application interface on a Windows

[4] https://mavsdk.mavlink.io/main/en/index.html
[5] http://qgroundcontrol.com/

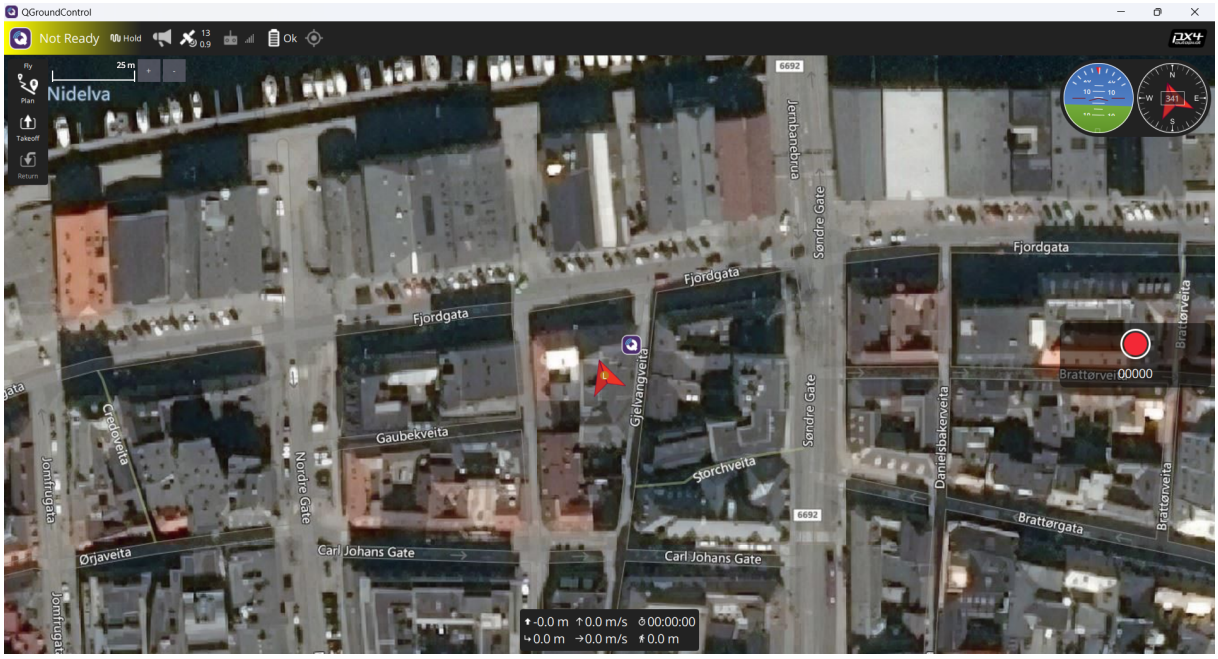


Figure 2.1: Main interface for the QGroundControl program

system.

Unreal Engine and AirSim

Unreal Engine[6] is a versatile 3D computer graphics tool primarily known for its game development capabilities. Initially introduced in 1998 for creating first-person shooters, the engine has evolved over time and is now widely used in various domains such as film, television, and research. It enables the creation of virtual sets, real-time rendering, computer-generated animation, and the development of virtual environments for architecture and vehicle design. Leveraging its real-time graphic generation capabilities, Unreal Engine serves as a powerful foundation for virtual reality applications. The interface of the engine, as shown in Figure 2.2, provides a user-friendly environment for project development.

AirSim[7], released by Microsoft in 2017, is an open-source simulator built on Unreal Engine. Designed for drones, cars, and other vehicles, AirSim supports both software-in-the-loop (SITL) and hardware-in-the-loop (HITL) simulations. It seamlessly integrates with popular flight controllers like PX4 and ArduPilot, allowing for realistic simulations that encompass both physical and visual aspects. Built as an Unreal Engine plugin, AirSim can be easily incorporated into existing Unreal environments. The main objective of AirSim is to provide a platform for AI research, facilitating experiments with deep learning,

[6] https://www.unrealengine.com/en-US
[7] https://microsoft.github.io/AirSim/

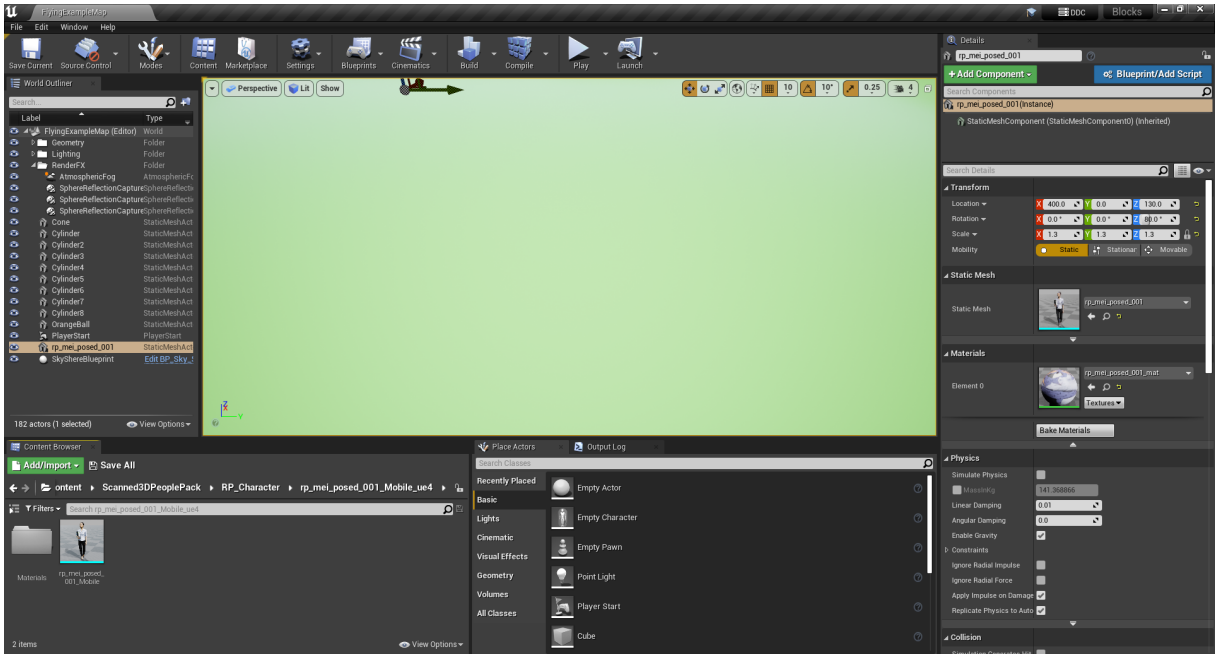


Figure 2.2: Project interface for the Unreal Engine

computer vision, and reinforcement learning algorithms for autonomous vehicles. To achieve this, AirSim offers APIs for data retrieval and vehicle control in a platform-independent manner. As of 2023, Microsoft has announced the development of a new simulation platform, Project AirSim, which will replace the original 2017 AirSim. While the original version will remain accessible to the public, it will no longer receive updates.

The combination of Unreal Engine and AirSim provides a robust framework for developing and testing autonomous systems. Researchers and developers can take advantage of the advanced capabilities of Unreal Engine to create visually immersive virtual environments, while AirSim's integration enables realistic simulations and AI experimentation. This integration offers a valuable toolset for exploring and advancing the field of autonomous vehicle technology.

AirSim plays a crucial role in this project by providing a simulated environment for testing and evaluating a vision-based control solution for tracking and following a person. By integrating AirSim into the project, the developed control algorithm can be applied to virtual drones within the simulated environment. This allows for comprehensive testing of the algorithm in various scenarios and conditions, providing valuable insights into its performance and effectiveness. Through this integration, the project benefits from the convenience and flexibility of simulation-based testing, ultimately leading to the development of a robust and reliable control solution for real-world deployment.

MediaPipe

MediaPipe[8], developed by Google, is an open-source project that provides customizable machine-learning solutions for live and streaming media. It offers a wide range of capabilities, including real-time perception of human pose, face landmarks, and hand tracking. These features enable the development of various applications, such as fitness and sports analysis, gesture control, sign language recognition, and augmented reality effects.

One of the key advantages of MediaPipe is its cross-platform support, allowing it to be used on Android, iOS, desktop/cloud, web, and IoT devices. It is designed to deliver accelerated processing and fast machine learning inference, even on less advanced hardware. This makes it accessible and applicable to a wide range of devices, ensuring that the developed solutions can be deployed on different platforms without sacrificing performance. MediaPipe offers a flexible framework specifically tailored for complex perception pipelines, making it well-suited for tasks that require real-time analysis of visual data.

Its versatility and robustness make it a valuable tool for researchers and developers working on computer vision and machine learning projects, empowering them to create innovative applications that leverage the power of streaming media, like the live feed from a camera onboard a drone. Given the constraints of the onboard hardware for this project, which may have limited processing capabilities, Mediapipe's ability to perform on smaller devices becomes essential.

GitHub

GitHub is an online hosting service that utilizes the Git version control system for software development. It serves as a centralized platform for storing code repositories, making it easy to manage and collaborate on projects. With a massive user base and millions of repositories, GitHub has become the leading source code host. In this project, both the software code[9] and this report[10] are stored in GitHub. This allows for seamless version control and easy access to the project's codebase. Additionally, GitHub allows users to create static websites directly from repositories through GitHub Pages. This feature serves as an ideal solution for creating a landing or presentation page for the project[11]. The page contains an overview of the project's goals and implementation details, accompanied by videos showcasing the tests conducted in Chapter 4.

[8] https://google.github.io/mediapipe/
[9] https://github.com/l-gonz/tfg-giaa-dronecontrol
[10] https://github.com/l-gonz/tfg-giaa-memoria
[11] https://l-gonz.github.io/tfg-giaa-dronecontrol/

OpenCV

OpenCV is a widely used open-source library for computer vision, machine learning, and image processing, offering a rich collection of over 2000 algorithms. It provides powerful capabilities for various image-related tasks, such as capturing images or videos from cameras, extracting valuable information from them, and performing image editing operations.

Developed primarily in C++, OpenCV also offers Python wrappers, allowing users to leverage the flexibility and simplicity of Python while benefiting from the optimized performance of the underlying C++ code. OpenCV-Python integrates seamlessly with the Numpy library, which specializes in efficient numerical operations and adopts a MATLAB-like syntax. This integration enables smooth data exchange between OpenCV and other libraries, like Matplotlib, for convenient graph plotting.

In this project, the OpenCV and Numpy libraries play a crucial role in managing image data and facilitating communication with the MediaPipe library. They are utilized to process image information, illustrate detected landmarks, and enable the annotation of images within the DroneVisionControl application's graphical user interface. Additionally, Matplotlib has been employed to generate the graphs presented in Section 4.2. These libraries collectively contribute to the effective handling and analysis of image-related tasks throughout the project.

2.2.2 Hardware

Pixhawk 4

The Pixhawk 4 is an advanced autopilot module developed by Holybro[12] in collaboration with the Dronecode Project team. It is designed based on the open hardware design of the Pixhawk-project[13] and optimized to run the PX4 flight stack on the NuttX[14] operating system.

Equipped with an integrated accelerometer, gyroscope, magnetometer, and barometer, the Pixhawk 4 can autonomously control unmanned aerial vehicles using the native capabilities of the PX4 flight stack. These built-in sensors provide essential data for flight control and navigation. Additionally, the Pixhawk 4 offers a range of connector sockets, depicted in Figure 2.3. The connectors allow for the expansion of its functionality with external sensors, input/output devices, or a companion computer. These features,

[12] https://shop.holybro.com/
[13] https://pixhawk.org/
[14] https://nuttx.apache.org/

combined with its powerful capabilities, make the Pixhawk 4 an integral component in the project's hardware setup.

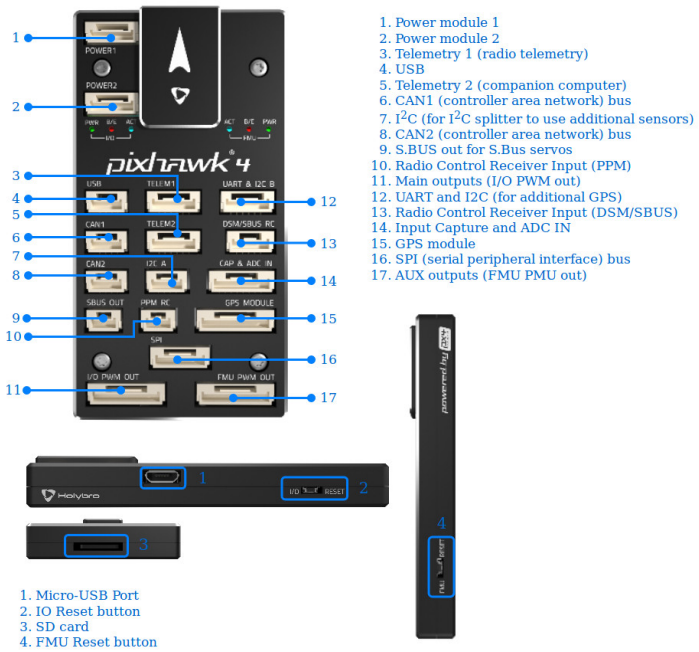


Figure 2.3: Side views and connector map for the Pixhawk 4 autopilot module.

Source: Adapted from *PX4 User Guide* [16].

Holybro X500

The Holybro X500[15] is a quadcopter specifically designed by Holybro to be compatible with the PX4 autopilot system. It is supplied as a comprehensive development kit that consists of essential components for assembly. These components include a durable carbon-fibre twill frame, the Pixhawk 4 flight controller, a power management board, four motors, a GPS module, an RC receiver, and a telemetry radio. The kit is designed for easy assembly, with a build time of approximately 3 hours and no need for specialized tools. The completed quadcopter is depicted in Figure 2.4, showcasing the final outcome of the assembly process.

[15] https://docs.px4.io/main/en/frames_multicopter/holybro_x500_pixhawk4.html

Figure 2.4: Fully assembled X500 kit.

Source: Adapted from *PX4 User Guide* [16].

Raspberry Pi 4B

The Raspberry Pi is a series of single-board computers known for their affordability, compact size, and user-friendly design. The Raspberry Pi Model 4B[16], used in this project, offers improved performance, expanded video output capabilities, and enhanced peripheral connectivity compared to previous models. Despite these advancements, it maintains the same affordable price and compact form factor.

The Raspberry Pi 4B computer comes as a bare circuit board without any additional components like a housing or cooling fan, as shown in Figure 2.5. It features various ports, including USB, HDMI, and Ethernet, as well as built-in Wi-Fi and Bluetooth connectivity. Additionally, it provides a 40-pin GPIO (General Purpose Input/Output) header, allowing direct connection of external devices for expanded functionality. The Raspberry Pi runs the Raspbian OS, a free operating system based on Debian that is specifically optimized for Raspberry Pi hardware. However, it is also compatible with other Linux distributions, which offers flexibility in software choices.

By leveraging the Raspberry Pi 4B as a companion computer to the Pixhawk 4 board, computationally intensive computer vision tasks are offloaded to a dedicated processor. Thanks to its compact size, it can be integrated onboard the vehicle to facilitate real-time processing of a camera feed from an attached camera. This setup enhances the capabilities

[16] https://www.raspberrypi.com/products/raspberry-pi-4-model-b/

of the system by leveraging the Raspberry Pi's processing power for efficient and responsive image analysis during flight operations.

Figure 2.5: Raspberry Pi 4 Model B

Source: Wikimedia Commons [17].

Chapter 3

Design and implementation

This chapter focuses on the design and implementation aspects of developing vision-based control solutions for UAVs within the PX4 ecosystem. It begins by discussing simulation as a safe and cost-effective environment for exploring and refining algorithms without the risks associated with real-world flights. The implementation of simulation in PX4 is explored, covering components such as the simulated flight stack and the simulation engine. The specific simulation and development environment for this project, including system configurations and the use of Unreal Engine and AirSim as the simulation engine, is also examined.

The chapter then moves on to describe the system architecture of the interaction between the flight controller and the DroneVisionControl application running on a companion computer. It discusses two possible hardware configurations: one where the companion computer is situated offboard the vehicle and another where it is placed onboard during flight. The key components of the system, including the flight controller, companion computer, and camera, are outlined, along with their connections. Furthermore, the software architecture of the DroneVisionControl application is described, highlighting the modules that make up the application.

To conclude, two control solutions are presented which demonstrate the capabilities of the PX4 ecosystem. The first one is a proof-of-concept solution where the vehicle is controlled from a ground station by hand gestures. The second is a human-following solution that makes the drone maintain a moving person in its field of view.

3.1 Simulation and development environment

Simulation plays a crucial role in developing vision-based control solutions for UAVs. It offers a safe and cost-effective environment to explore and refine algorithms, test new capabilities, and evaluate system performance without the risks associated with real-world flights. In this context, simulation refers to creating a computer-generated virtual environment that mimics a UAV's real-world conditions and interactions. In this simulation, various aspects of the UAV system, including the flight dynamics, sensor inputs, and environmental factors, are replicated and simulated in a software-based environment.

This section explores how simulation is implemented in the PX4 platform and the development environment built around it for this project to develop the hand-gesture and person-following control solutions mentioned before. The three key simulation components are examined individually: the simulated flight stack, the simulation engine or simulator, and the companion computer for offboard control. The two available simulation modes, Software-in-the-Loop (SITL) and Hardware-in-the-Loop (HITL), are introduced for the flight stack. In contrast, the simulation engine encompasses modelling physical world components and the physics and rendering engines. The communication between the simulator, flight controller firmware, and companion computer is also described. Lastly, the development environment is discussed, covering network configurations and using Unreal Engine and AirSim as the simulation engine for this project.

3.1.1 SITL and HITL simulation

The PX4 platform is a flexible and powerful open-source flight control software stack widely used for developing for UAVs. The platform's flight stack is the core software responsible for controlling the UAV's behaviour and executing flight commands. It offers comprehensive support for simulation, providing developers with a robust environment to test and refine their control solutions.

There are two main simulation modes for the flight stack: Software-in-the-Loop (SITL) and Hardware-in-the-Loop (HITL). SITL simulation enables the flight stack to run on a non-dedicated computer, simulating the flight controller's operating system and allowing for rapid development and testing. On the other hand, HITL simulation executes the simulation firmware on an actual flight controller board, providing a more realistic testing environment. In this mode, the flight stack runs in its native environment while the sensor data and other external inputs are simulated.

Both simulation modes in the PX4 platform offer distinct advantages and are suited for different stages of development. SITL simulation allows developers to iterate quickly,

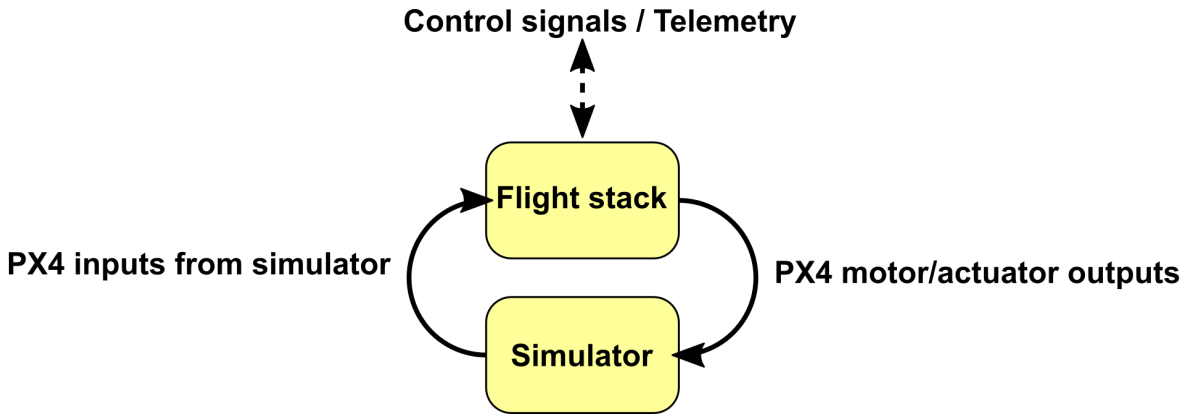


Figure 3.1: Feedback loop during the PX4 simulation.

Source: Adapted from *PX4 User Guide* [16].

reducing development time and providing immediate visual feedback on the computer screen. This mode is particularly beneficial during algorithm exploration and refinement. HITL simulation, on the other hand, offers increased realism by testing on an actual flight controller board. It allows developers to evaluate their vision-based control solutions in a more accurate representation of the real-world environment. This mode is especially valuable when fine-tuning algorithms and assessing system performance.

In the simulation context, two inextricable components exist: the simulated flight stack (running on SITL or HITL mode) and the simulation engine or simulator. The flight stack interacts with the simulator through a feedback loop, enabling a seamless exchange of information between the two components, as shown in Figure 3.1. The feedback loop begins with the simulator generating sensor inputs, such as acceleration measurements or GPS data, based on its internal representation of the simulated world. These sensor inputs are then transmitted to the flight stack. Upon receiving the sensor inputs, the flight stack processes this information and generates response actuator controls, including motor commands or control signals for various UAV components, which are then sent back to the simulator. Finally, the simulator utilises these actuator controls to update the virtual vehicle's position, velocity, and attitude within the simulated world, thereby simulating the UAV's response to the flight stack's commands.

This interaction between the flight stack and the simulator within the feedback loop allows for a dynamic and synchronised simulation experience, mimicking the behaviour of a real-world UAV by providing realistic sensor inputs and simulating the corresponding actuator responses. All communication between the flight stack and the simulator is implemented using the MAVLink messaging protocol and the UDP transport protocol. MAVLink messages serve as a standardised format for transmission, allowing the seamless exchange of information between the two components.

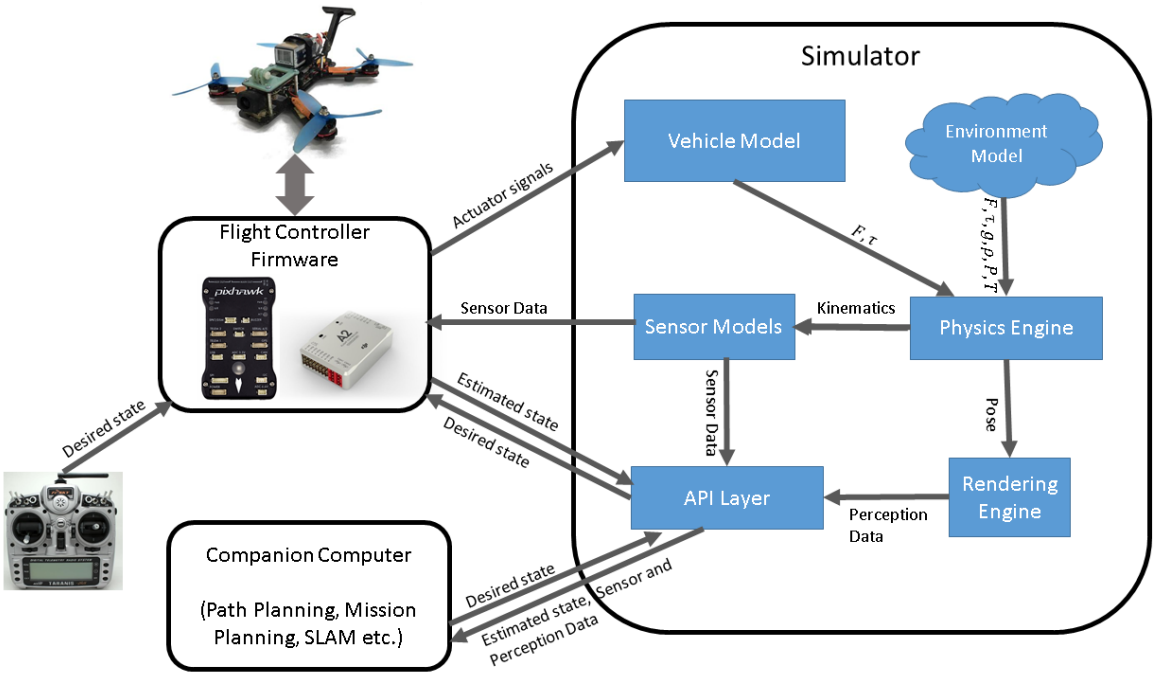


Figure 3.2: High-level overview of how different components of a simulator interact with the flight stack.

Source: Adapted from *Aerial Informatics and Robotics Platform* [18].

3.1.2 Simulator

The simulator plays a crucial role in development by providing visual feedback on the vehicle's reactions to the control software. Inside the simulator, a group of components that model real-world elements work together to make up the complete engine. Figure 3.2 expands upon the diagram in Figure 3.1 to show the high-level overview of how each of these components interacts with each other and with the flight controller firmware (flight stack).

The environment model represents the virtual world where the simulation occurs, providing the simulated surroundings and obstacles for the UAV. It can include terrain and weather conditions that affect flight dynamics, like wind or variable temperatures, which are sent to the physics engine. The vehicle model represents the UAV's physical characteristics, such as its mass, size, aerodynamics, and propulsion system, which are simulated to generate forces and moments based on actuator signals. The sensor model simulates the behaviour and outputs of cameras, lidars, and IMUs, providing realistic sensor data to the flight stack. The physics engine governs the laws of physics within the simulation, calculating forces, torques, collisions, and other physical interactions to ensure that the vehicle and its environment interact realistically. The rendering engine is responsible for rendering the visual aspects of the simulation, allowing for realistic graphics and visualisation. The final component of the simulator is the API layer. It serves

as a communication interface between the simulator and the flight stack, facilitating the exchange of data and control commands through the MAVLink protocol. This API can also connect the simulator to an external companion computer to use the sensor and state data provided.

In this context, the companion computer is an additional computational device that works together with the flight controller firmware and the simulator. It aims to offload specific tasks and computations from the flight controller, enabling more complex algorithms, higher-level control, and real-time data processing. The companion computer allows implementing vision-based control solutions, as it can handle intensive image processing and machine learning. It facilitates the integration of advanced perception, planning, and decision-making capabilities, enhancing the UAV's autonomy and enabling more sophisticated behaviours in simulated environments. The communication between the companion computer and the simulation system can be established by the simulator's own API layer, as shown in Figure 3.2, or through any dedicated API that can interact with the flight stack through MAVLink messages. An example of the latter case would be the MAVSDK API mentioned in Section 2.2.1. This project will use this method to separate the communication between the flight stack and the companion computer from the simulator. The separation allows for a more seamless transition into flight tests, where the real world replaces the simulator component.

There are many options for simulators supported by PX4. The simpler of these is jMAVSim, which can be installed along with the PX4 SITL simulation on a Linux system. It provides a lightweight simulation environment for PX4, allowing for basic testing and evaluation of flight control algorithms. While it may lack some advanced features, jMAVSim is a convenient option for checking that the simulated flight stack has been configured correctly during initial development iterations. A second option for running on Linux is Gazebo. Gazebo is a powerful simulator widely used in robotics and compatible with PX4. It offers more advanced capabilities, such as obstacle avoidance and support for the Robot Operating System (ROS). It provides a realistic physics engine for simulating UAV dynamics and environments and enables the integration of complex sensor models, making it suitable for testing perception and navigation algorithms. However, it is more limited in graphics, which becomes critical when testing control solutions driven by computer vision.

The last option considered to fulfil the requirements of the project is AirSim. This simulator is built as a plugin for the popular Unreal Engine, designed for game development. Unlike jMAVSim and Gazebo, AirSim runs on Windows and leverages the capabilities of a game engine. It provides visually and physically realistic simulations, offering advanced graphics and rendering capabilities. AirSim is particularly advantageous for testing computer vision features as it offers easy access to thousands of visual packages through its asset library. Its integration with Unreal Engine enables the creation of complex and immersive simulated environments. These features make it

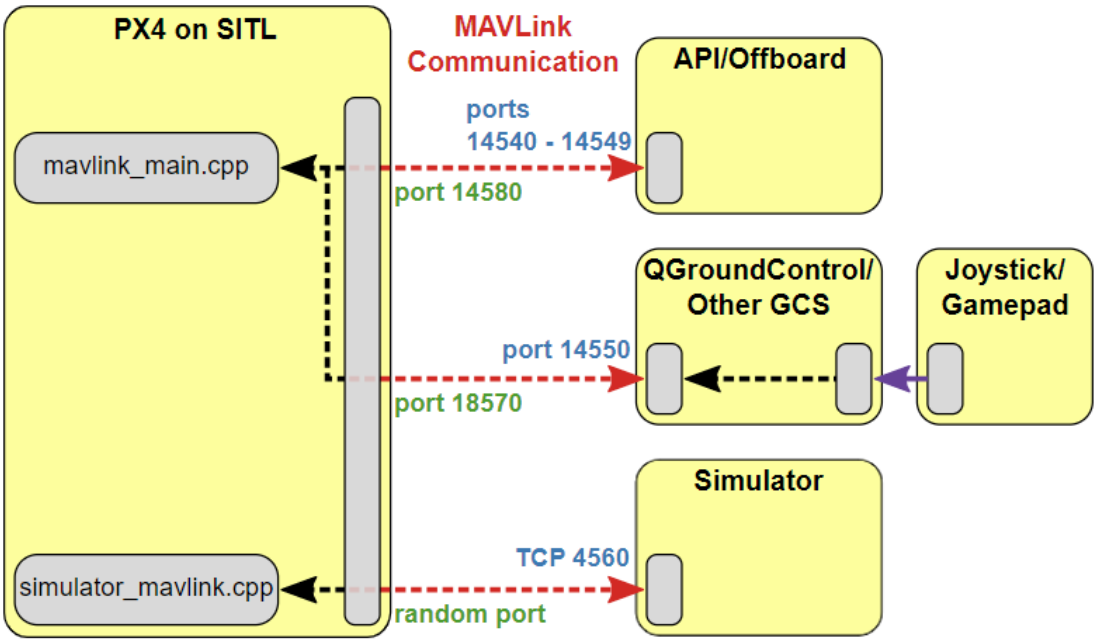


Figure 3.3: Network diagram between the components interconnecting during software-in-the-loop simulation.

Source: Adapted from *PX4 User Guide* [16].

especially suited for developing vision-based control solutions, making it the best choice for this project.

3.1.3 Development environment

This section discussed the configuration adopted in this project to establish connections between the developed control software, the selected AirSim simulator, and the flight stack. It encompasses both SITL and HITL simulation modes, which allow testing the implementation of the DroneVisionControl application within a controlled environment. While the application is designed to leverage the dedicated processing power of a companion computer in real-world scenarios, during simulation, it will primarily operate on the same computer hosting the simulation engine to minimise physical connections between multiple machines.

SITL configuration

The PX4 platform has an established network configuration for SITL simulation, shown in Figure 3.3. The figure's left side shows the two programs that manage MAVLink

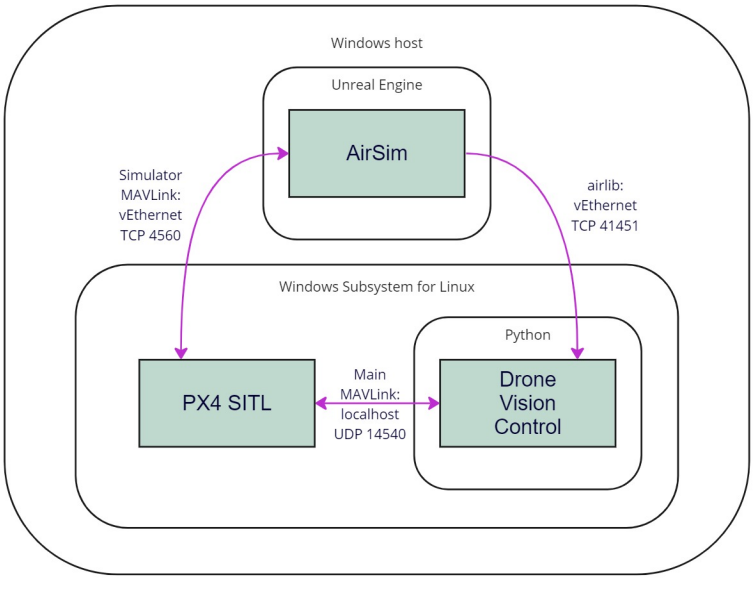


Figure 3.4: Connection diagram of how the three systems interact with each other during SITL simulation.

servers for communicating with components outside the flight stack. The first is the main MAVLink manager, inherent to the PX4 system. It expects connections from offboard control sources, such as a companion computer using an offboard API (MAVSDK) or a ground control station like QGroundControl (2.2.1). These connections are established using the UDP protocol on the port range 14540-14550. By default, port 14550 is reserved for a ground station but can be used by any other offboard application if QGC is not required. The second server runs exclusively during simulations and connects to the simulator program instead through a TCP connection, as it requires higher reliability than the offboard connections.

In this project, the simulator will run on a Windows system, as that is the native environment of the AirSim simulator. The SITL simulation, however, runs best on Linux. To execute both software components simultaneously on a single machine, the Windows Subsystem for Linux (WSL) [19] will be utilised. WSL allows for the execution of a Linux distribution, such as Ubuntu, directly within the Windows operating system, providing a convenient environment for running PX4 SITL alongside AirSim without the need for dual-booting or virtual machines. This configuration ensures that the connections are limited to the machine's internal network, as WSL handles the task of establishing network connectivity between the Windows and Linux environments. It achieves this by setting up a virtual Ethernet bridge and ensuring network traffic is routed appropriately. The complete steps needed to configure the PX4 WSL system are detailed in Appendix A.1.

The complete set of connections necessary between the three individual components

at the transport layer level during SITL simulation is shown in Figure 3.4. The AirSim simulator runs as part of the Unreal Engine application directly on the host Windows system. It connects to the other two components running inside the virtualised Linux subsystem through the established bridge. AirSim and the PX4 flight stack exchange information through the TCP connection to the secondary MAVLink server, as shown in Figure 3.3.

The communication between AirSim and the DroneVisionControl application is facilitated through the API layer depicted in Figure 3.2, which can connect the simulator with a companion computer or other system running additional code. This connection is established using the `airlib` library developed for Python by the creators of AirSim, which allows easy access to the AirSim API. In this project, the connection between AirSim and the application through `airlib` transmits only perception data, specifically camera images of the simulated world. For the transmission of other data, such as the current vehicle state from the simulator to the application and the desired state from the application to the simulator, a direct connection between the flight stack and the application is used instead, as this is the only link that can be maintained in a real-world scenario. This direct link is established through the main MAVLink instance of the flight stack, as illustrated in Figure 3.3.

Given that the application is developed in Python, a versatile programming language compatible with multiple platforms, it can be run either on the central Windows system or within the virtualised Linux subsystem. However, running the application on Linux is more advantageous to avoid the need for the MAVLink server to broadcast to the Windows system.

HITL configuration

Transitioning from SITL to HITL simulation involves shifting from the WSL environment to a physical flight board, specifically a Pixhawk flight controller, to run the flight stack firmware. In HITL simulation, the flight board's motors and actuators are blocked, but the internal software functions fully. The Python execution will also be moved from the Linux subsystem to the Windows host. This transition simplifies the testing process by eliminating the need for additional configurations to establish communication between the flight controller and the internal WSL network, which is isolated from external machines by default.

As the flight stack now runs on separate hardware, it becomes necessary to establish a physical connection between the testing computer and the flight controller for each desired MAVLink channel. This connection can be set up using the debug micro-USB port on the flight controller or any available telemetry ports, which allow serial connections to external systems. These connections can be established, for example, using a serial-to-USB

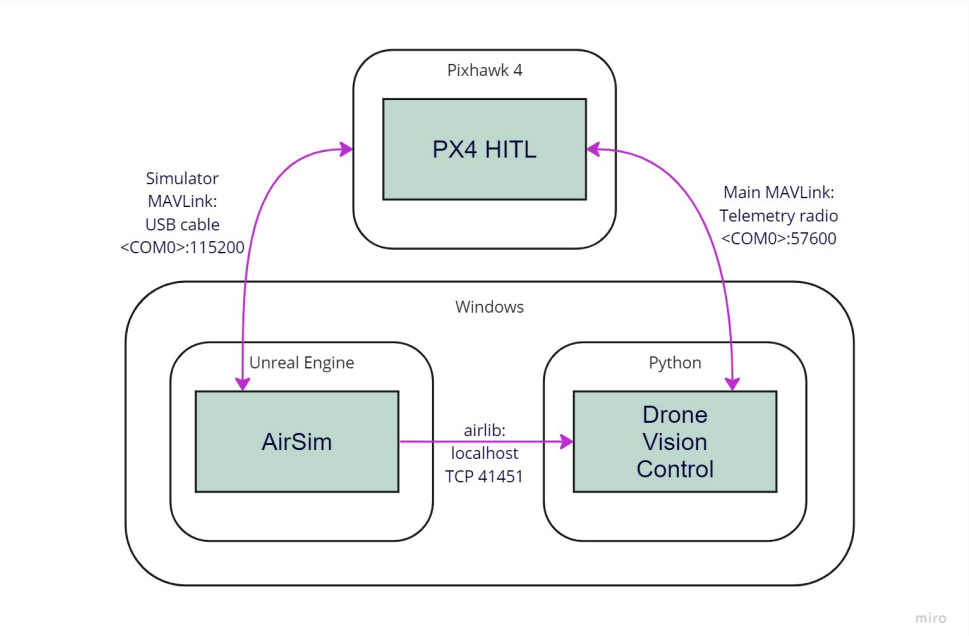


Figure 3.5: Connection diagram of how the three systems interact with each other during HITL simulation.

converter or through a pair of telemetry radios. The simulator and the Python application can communicate independently with the flight controller by using several of these options simultaneously.

Figure 3.5 illustrates the chosen connections for executing tests in HITL mode. The Windows machine hosts both AirSim in Unreal Engine and the Python interpreter running the developed application, which keep their communication through the `airlib` library. In this case, there is no need for the virtual bridge network, and the TCP connection is established through the `loopback` network. The PX4 flight controller is connected to the simulator via a USB to micro-USB cable, configured with a baud rate of 115200. It is also connected to the Python program through a telemetry radio operating at a baud rate of 57600. Both connections on the Windows computer side are attached to USB ports. These ports are accessible via their respective COM addresses, which must be specified in the configuration for AirSim and MAVSDK.

AirSim environment

Microsoft develops the AirSim simulator as a plugin in the computer graphics and game development program Unreal Engine. It implements sensor, vehicle and environment models appropriate for flight simulation while taking advantage of the physics and rendering engines integrated into Unreal. In the engine, projects comprise one or more environments where components like 3D models, cameras, and lighting can be added.

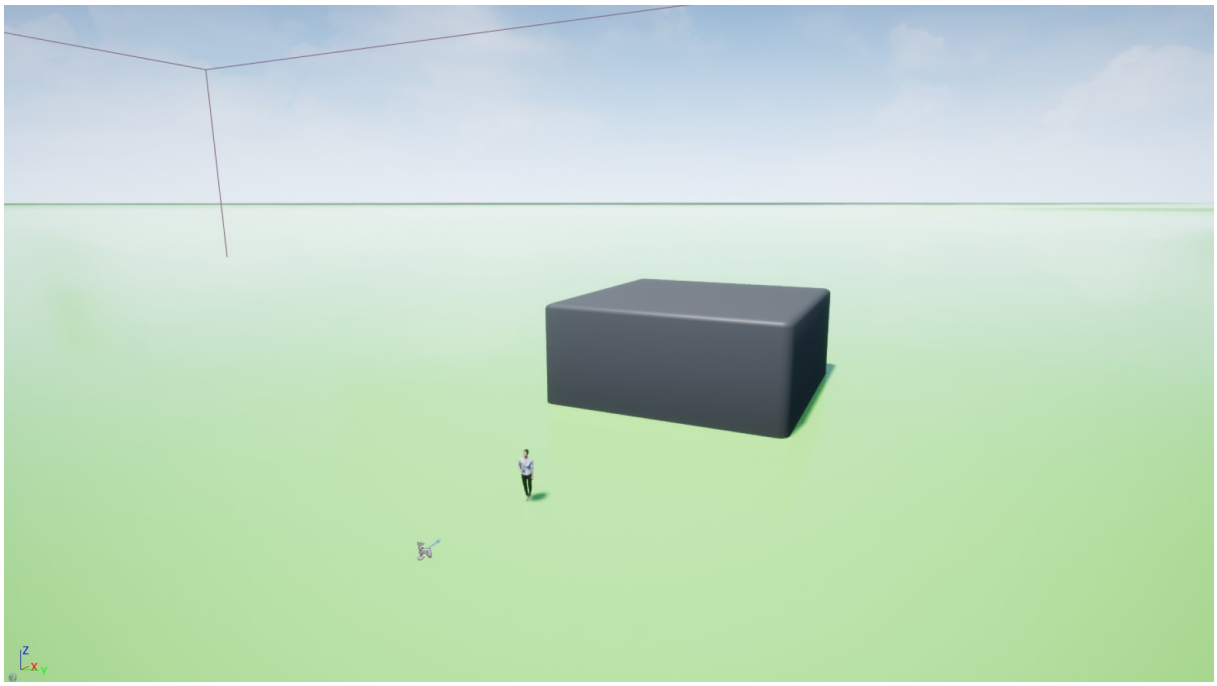


Figure 3.6: Screenshot from the Unreal Engine environment used for testing the computer vision solutions.

The source code for the AirSim project includes a basic Unreal environment with all the minimum components already configured that can be used to test the implementation or as a starting point for more complex environments. It contains several 3D shapes like blocks and spheres and a simple quadcopter to act as the simulated vehicle. Creating custom Unreal environments and running AirSim inside them is also possible by manually adding the built plugin and a vehicle to an existing project.

The environment used in this project for testing the `drone-vison-control` application is derived from the base AirSim environment and can be found on the project's repository[1]. It contains the default quadcopter vehicle, which includes several virtual cameras that allow retrieving images from the vehicle's point of view in the simulated world. The main addition to the environment is the 3D model of a human figure, to be used for testing the human detection and tracking mechanisms in the computer vision solution. This model is part of a free asset library of human models made by Renderpeople [20] obtained from the Unreal Marketplace. Some minor modifications have also been made to the background shapes and colours to provide better contrast for the camera. Figure 3.6 shows an image of the testing environment as seen from the Unreal viewport in Edit mode.

AirSim is compatible with both SITL and HITL simulation modes and several other flight stacks apart from PX4, including AirSim's own internal SimpleFlight flight stack, which is used by default. The plugin must therefore be configured for this project to work with the desired simulation setup (PX4 + WSL + Airsim). This process is explained in

[1] https://github.com/l-gonz/tfg-giaa-dronecontrol/tree/main/data

the AirSim documentation [21] by its developers and in Appendix A.1 for more project-specific details. Appendix A.3 contains the complete settings file used in this project for configuring PX4 with either simulation mode in AirSim, including the parameters that must be individualised for each system.

To start the simulation in AirSim, the following steps must be followed:

1. Attach physical flight controller in HITL mode to simulation computer (HITL only).
2. Start play mode in Unreal.
3. Build and start simulated flight stack in WSL (SITL only).
4. Start companion applications, if any (e.g. DroneVisionControl, QGroundControl).

To build and start the PX4 SITL flight stack for AirSim, a small script can be found in the project source code[2] that will automatically attach to a running AirSim instance if it is executed with the `--airsim` option. To be able to use the physical flight board for HITL simulation, this mode needs to be enabled from the QGroundControl safety configuration. Once the simulation has started, there is no noticeable change between the simulated flight controller in WSL (SITL mode) and the one running in the physical board (HITL mode).

3.2 System architecture

This section describes the architecture of the interaction between the flight controller and the DroneVisionControl application. It outlines the key components of the system along with their connections, discussing two possible configurations: offboard, where the companion computer acts as a ground station, and onboard, where the companion computer and camera are onboard the vehicle. The following sections explore the details of each configuration.

3.2.1 Top-level components

The DroneVisionControl application aims to direct the movement of a UAV by analyzing images captured by a camera. Since the processing power needed to analyse the images exceeds the capabilities of the autopilot flight controller, an additional companion computer is necessary. This companion computer will control the camera and utilise

[2] https://github.com/l-gonz/tfg-giaa-dronecontrol/blob/main/simulator.sh

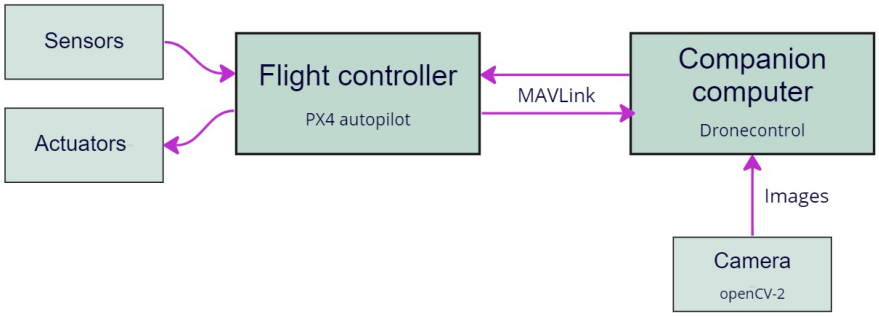


Figure 3.7: Top-level diagram of the hardware/software interactions

machine learning algorithms to extract useful features from the images and convert them into movement directives for the vehicle.

Figure 3.7 illustrates a top-level diagram depicting the key components of the system. The main elements include the flight controller, running the PX4 autopilot firmware, the companion computer hosting the developed application, and the camera responsible for image capture. The camera connects to the companion computer using a USB cable plugged into any available port. The flight controller establishes the communication with the companion computer via the MAVLink protocol described in section 2.2.1, either through telemetry radios or a direct wire connection. The choice of connection type depends on the desired setup of the system.

In the simplest configuration, the companion computer can function as a ground station, directing the vehicle's movement from the ground while the flight controller remains onboard. This configuration is possible when the camera doesn't need to move with the vehicle. In this scenario, wireless communication becomes critical, so it will be established utilizing a pair of telemetry radios. Section 3.2.2 provides a comprehensive guide for this configuration.

Alternatively, when the camera needs to move with the vehicle to capture images from its perspective, the companion computer and camera are placed onboard the vehicle alongside the flight controller. In this case, a direct wired connection between the two is the most suitable option, offering a faster and more stable link. Details of this configuration are provided in section 3.2.3.

The flight controller

The flight controller or autopilot is a circuit board equipped with sensors, and optimized for running the flight stack. The autopilot used in this project is the Pixhawk 4, designed by Holybro for the PX4 firmware. This board depends on a specific set of hardware components to function properly. These components include sensors, which collect environment data for processing, and actuators, which transform the output signals from the controller into movement.

To determine the vehicle's state for stabilization and autonomous control, PX4 relies on sensors such as gyroscopes, accelerometers, magnetometers (compasses), and barometers. These are the sensors that are typically included already integrated with the autopilot's board. Additionally, a GPS or other positioning system is required to enable fully automatic flight modes and assisted features like altitude stabilization or mission planning.

PX4 utilises outputs to control motor speed, flight surfaces (e.g., ailerons and flaps), camera triggers, parachutes, grippers, and other payloads. Brushless motors, controlled by Electronic Speed Controllers (ESCs) connected to the flight controller, rotate the propellers in most PX4 drones. These drones typically employ Lithium-Polymer (LiPo) batteries, which are connected to the system using a Power Module or Power Management Board, providing separate power for the flight controller and ESCs.

For manual control of the vehicle, a Radio Control (RC) system is employed, consisting of a remote control unit that communicates stick and control positions to a receiver on the vehicle. Advanced RC systems can also receive telemetry information from the autopilot, providing feedback to the operator. Telemetry radios offer an alternative means of communication with the autopilot, establishing a wireless MAVLink connection between a ground control station and a PX4-powered vehicle. This enables real-time parameter tuning, flight mode commands, telemetry inspection, and on-the-fly mission changes.

In an actual UAV, the PX4 software runs on dedicated hardware like the Pixhawk 4 mentioned. This hardware includes all the essential sensors for flight as well as interfaces to connect additional actuators and I/O systems (RC, telemetry radio). However, on a simulated environment like the one described in section 3.1, all the hardware components of the sensors and actuators are simulated on the same computer running the flight stack.

DroneVisionControl and the companion computer

The DroneVisionControl application that runs on the companion computer utilizes the Python programming language[3]. Python offers several advantages for projects of this

[3] https://www.python.org/

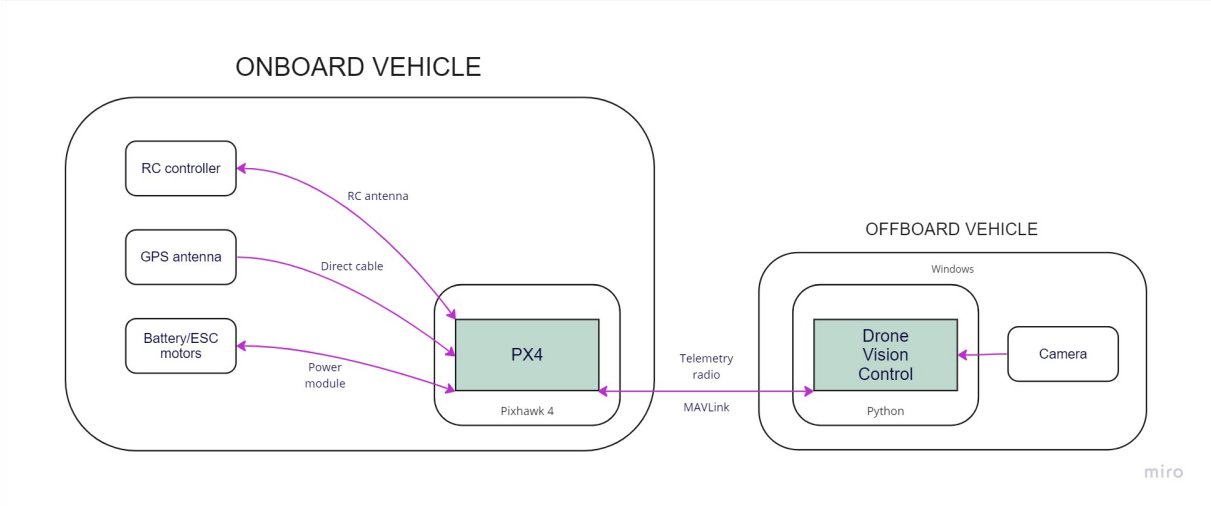


Figure 3.8: Offboard configuration connections

nature, including its high-level, easy-to-use syntax, which results in a more concise code base compared to other languages. Other benefits include Python's versatility and support for object-oriented programming. Moreover, Python has a vast ecosystem of external libraries accessible through its official package manager called `pip`. This package index[4] contains thousands of well-tested utilities, including many designed for machine learning and image processing. Additionally, Python versions of all the necessary libraries for interacting with PX4 via the MAVLink protocol, for object detection and tracking, and for simulation (MavSDK, OpenCV, AirSim, Mediapipe) are available. Being an interpreted language, Python can run seamlessly on any system with Python installed, eliminating the need to compile separate binaries to run in different operating systems.

The following sections will provide an in-depth exploration of the differences between the two configurations mentioned earlier for offboard and onboard companion computers.

3.2.2 Offboard computer configuration

The offboard configuration allows the flight controller to communicate and receive orders from a companion computer that is not physically connected to its hardware, so the latter can remain on the ground while the vehicle flies. This configuration offers several advantages, including a simplified setup without concerns about hardware interactions and power supply to the companion computer during flight. It also allows for the use of a more powerful computer for image processing without adding weight to the vehicle. However, the camera remains connected to the ground computer, limiting the system's real-world applications as the images are not captured from the drone's perspective during flight. While other configurations involving direct camera-to-flight controller connection

[4] https://pypi.org/

and wireless transmission of images for offboard processing are feasible, they are beyond the scope of this project.

In this configuration, the wireless link is established through a pair of telemetry radios. These radios connect to a telemetry port on the flight controller and a USB port on the companion computer. Since the Pixhawk 4 is configured by default to use its `TELEM1` port for this purpose, no additional configuration is needed when using that port. Applications like the QGRoundControl ground station software automatically detect a telemetry radio inserted into any USB port on the host computer and establish the connection to the flight controller. Additionally, software using the MavSDK library can establish a connection by specifying the USB serial port address and the baudrate of the link, usually something similar to `/dev/ttyUSB0:57600` on Linux and `COM1:57600` on Windows.

The radio used for the physical tests in this project is the Holybro SiK Telemetry Radio. It is a small, light and inexpensive open-source radio platform that typically allows ranges of more than 300 meters ”out of the box” (the range can be extended to several kilometres with a patch antenna on the ground). The radios are offered as 915Mhz (Europe) or 433Mhz (US), so they can be used in different regions and comply with the regulations for frequency, hopping channels and power levels. They offer 2-way full-duplex communication through an adaptive TDM UART interface, and their antenna allows for an adjustable 100-mW-maximum output power and -117 dBm receive sensitivity. The link is established by default with a baudrate (max bits per second on a serial channel) of 57600, and it can provide air data rates of up to 250 kbps.

The project utilizes the Holybro SiK Telemetry Radio[5] for physical tests. These radios are small, lightweight, and cost-effective, offering a range of more than 300 meters out of the box (which can be extended with a patch antenna). They operate at either 915MHz (Europe) or 433MHz (US), complying with regional frequency regulations. The radios support two-way full-duplex communication through an adaptive TDM UART interface, with an adjustable maximum output power of 100mW and a receive sensitivity of -117dBm. The default baudrate for the connection is 57600, and the radios can achieve exchange rates of up to 250kbps.

Figure 3.8 provides a summary of the connections required for the offboard computer configuration setup, including the PX4 software, the DroneVisionControl application, their respective hardware platforms, and onboard and ground station peripherals.

[5] http://www.holybro.com/product/transceiver-telemetry-radio-v3/

3.2.3 Onboard computer configuration

The second way of configuring the interaction between the flight controller and the companion computer consists of incorporating both of them together on board the UAV. This is achieved by connecting the flight controller directly to the companion computer using a serial cable. The camera will also be onboard the vehicle, attached to the frame in a way that allows for a practical perspective during flight. This configuration makes it possible to develop new control solutions based on images taken directly from the vehicle, creating a feedback loop that adjusts to maintain a stable output based on the reaction of the vehicle to commands. Figure 3.9 shows a summary of all the connections present in the onboard configuration between the three pieces of software that interact together: PX4, DroneVisionControl and QGroundControl, with their respective hardware platforms and the attached peripherals.

Selecting the appropriate hardware for the onboard computer is crucial in this configuration since the companion computer has to fly along with the flight controller. To be able to take into the air, the computer has to be light enough that its weight can be lifted by the propellers while maintaining adequate battery autonomy. It also needs to be powerful enough that its processor can handle computer vision algorithms. The Raspberry Pi 4 model chosen for this project and shown in Figure 3.10 is one of the most popular small

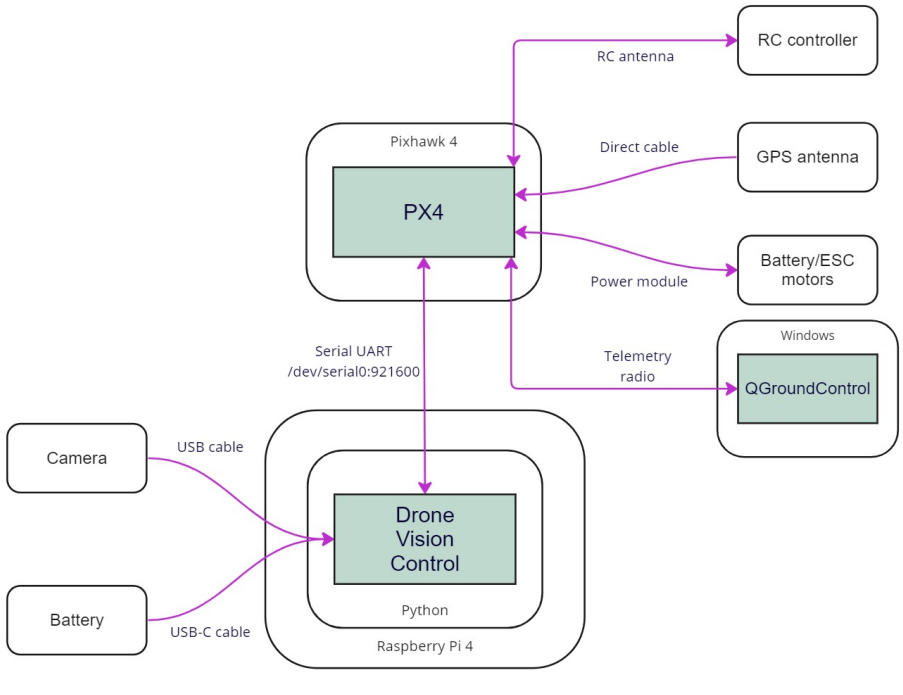


Figure 3.9: Overview of the onboard configuration. All connections are contained inside the vehicle's frame.

computers available in the market at the time, and it is widely used in all kinds of robotics projects both for education and hobbyists. One of the most crucial advantages of using such a platform is the excellent availability of manuals, guides, and other support found on the web. In addition, the Raspberry's officially supported operating system, called Raspberry Pi OS, is a Debian-based version of Unix, which simplifies the transition from the WSL test environment.

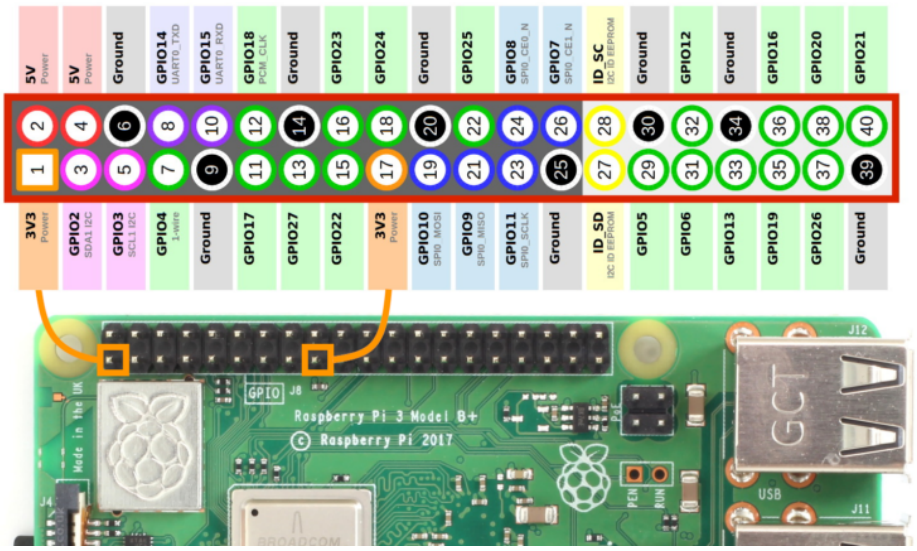


Figure 3.10: The Raspberry Pi microcomputer, with its 40-pin GPIO header marked in red and annotated pinout.

Source: Adapted from *Raspberry Pi GPIO Header with Photo* [22]

Since this computer is designed for integration with hardware projects, it includes a 40-pin General Purpose Input/Output (GPIO) header (highlighted in Figure 3.10) for connecting external devices. This pin header, along with the standard ports in the Raspberry Pi, will be used to implement the three connections to the companion computer required for the onboard configuration, shown in Figure 3.11. The first connection will provide power to the computer, the second will be a connection to the camera that provides images, and the third one will be the telemetry link to the flight controller.

The Raspberry Pi is powered by a 5V input that can be supplied via the USB-C port or specific pins on the GPIO header ("5v Power" on Figure 3.10). In the specific vehicle build of this project, the Holybro PM07[6] power management board supplies 5V to the flight controller and powers the ESCs for the motors. The power management board features two power outputs: one connected to the flight controller's `POWER1` port and an unused one. Initially, attempts were made to power the Raspberry Pi from the main battery by

[6] http://www.holybro.com/product/pixhawk-4-power-module-pm07/

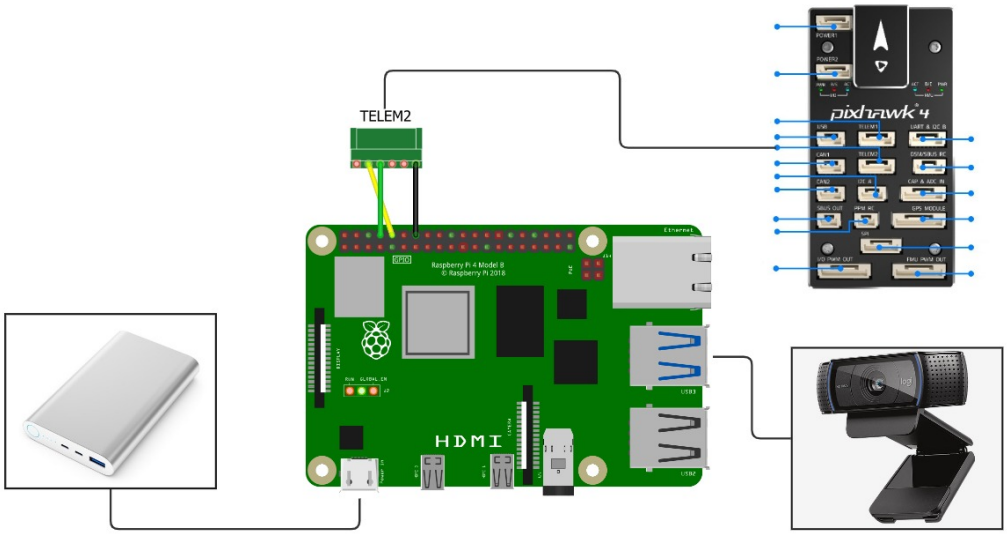


Figure 3.11: A diagram of the wired connections from the Raspberry Pi 4 to the secondary battery and the flight controller (`TELEM2`).

connecting the second output on the power module to the GPIO header's powering pins using a custom connector. However, this resulted in an unstable power supply for the Pi board, causing frequent current dips that affected the companion computer's processing capabilities. To address this, a secondary battery was introduced, providing power to the Raspberry Pi via a USB to USB-C cable. This configuration allows the Raspberry Pi to receive power through its default regulated USB-C port. The drawback is the additional weight of the secondary battery, which also needs to be securely attached to the vehicle's frame during flight.

In contrast to selecting a companion computer, the choice of camera for the onboard system offers greater flexibility. The key considerations are lightweight design and straightforward plug-and-play compatibility with the onboard computer. The camera utilized in the tests outlined in section 4 is the Logitech C920 1080p webcam[7]. Since the Holybro X500 frame doesn't natively support an onboard camera, a custom mount was designed and 3D-printed using PLA plastic. This mount securely attaches the camera to the underside of the vehicle frame's central rods, ensuring stability during flight. The 3D model of the mount is depicted in Figure 3.12, and the print-ready file is available in the project's repository in GitHub[8].

The last wired connection that needs to be established for this configuration is between the flight controller and the companion computer for MAVLink message exchange. This connection will use the secondary telemetry port of the flight controller, `TELEM2`.

[7] https://www.logitech.com/es-es/products/webcams/c920-pro-hd-webcam.960-001055.html
[8] https://github.com/l-gonz/tfg-giaa-dronecontrol/blob/main/data/camera-holder.stl

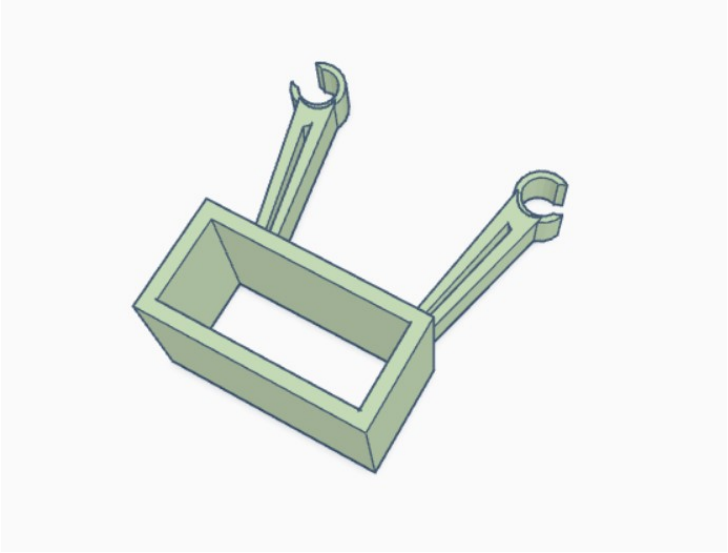


Figure 3.12: 3D model for the camera mount designed for the Holybro X500 frame.

Meanwhile, the telemetry radio will remain connected to the `TELEM1` port to maintain a wireless link. This link can be employed by a ground station (QGroundControl) to override the companion computer's control during flight. The telemetry connections on the flight controller use a 6-pin adapter to attach to the `TELEM` ports. On the Raspberry Pi side, the other end of the connector has three female Dupont wires that connect to the TX/RX UART pins. The pins in the telemetry port are mapped to the corresponding GPIO pins on the Raspberry Pi's header according to the wiring table provided in 3.1. This wiring configuration is also depicted in the diagram in Figure 3.11.

TELEM2		**GPIO header**	
Pin #	*Description*	*Description*	*Pin #*
1	VCC, +5V		
2	TX (out), +3.3V	GPIO15 (RXD0, UART)	10
3	RX (in), +3.3V	GPIO14 (TXD0, UART)	8
4	CTS (in), +3.3V		
5	RTS (in), +3.3V		
6	GND	GND	6

Table 3.1: Mapping between the `TELEM2` port in the Pixhawk 4 board and the Raspberry Pi's GPIO header.

By default, the secondary telemetry port, `TELEM2`, is not enabled for use. Its configuration can be changed through a ground station computer connected to the Pixhawk board by using the Parameters section of the QGroundControl application. The specific parameters that need to be set are discussed in section 4.3.1.

Compared to the default baud rate of 57600 used for the telemetry radio link established earlier, the wired serial connection operates at a faster rate of 921600. This means that data

can be transferred up to 16 times faster through this link. However, it is important to note that the primary limitation on speed for the program lies in the detection and tracking process running on the images. Therefore, a faster link rate does not necessarily result in an overall performance improvement for the solution. In section 4.3.2, different hardware combinations are analyzed to identify any challenges that may impact the program's performance.

Moving beyond the individual hardware components, it is important to compare the use of this onboard configuration with the previously discussed offboard configuration during testing. While the offboard setup allowed for monitoring the program's output by connecting a screen directly to the companion computer on the ground, such a setup is not feasible in the onboard configuration. However, a solution to this limitation is to leverage the Raspberry Pi's WiFi antenna and configure a remote desktop connection. By connecting to this remote desktop from another computer acting as a ground station, real-time monitoring of the camera output and image recognition can be achieved, along with the capability to provide direct input during flight.

3.3 Software architecture

In this section, the architecture of the designed DroneVisionControl application is presented. Figure 3.13 illustrates the main modules of the designed software and their interactions with external libraries. The application comprises three fundamental parts: the **pilot module**, responsible for sending instructions to the flight controller and receiving position and state information through the `mavsdk` library; the **video source module**, which handles image retrieval from various sources and performs necessary image analysis processing; and the **control module**, which facilitates interaction between the other two modules to convert pixel information into position points using the `mediapipe` library, and further into instructions for the pilot.

In the upper part of the diagram in Figure 3.13, the flow of information between the DroneVisionControl application and the external systems is depicted. Green lines represent the path in a simulated workflow, while blue lines indicate the alternative path for a system with actual quadcopter hardware. Purple arrows indicate the input/output of each module within the developed application and how they interconnect. Additionally, smaller utilities have been developed to test the interaction between systems and calibrate different aspects of the control behaviour. These utilities are described in sections 3.3.4 and 3.5.1. A user manual with all the options available in the application can be found in Appendix B.

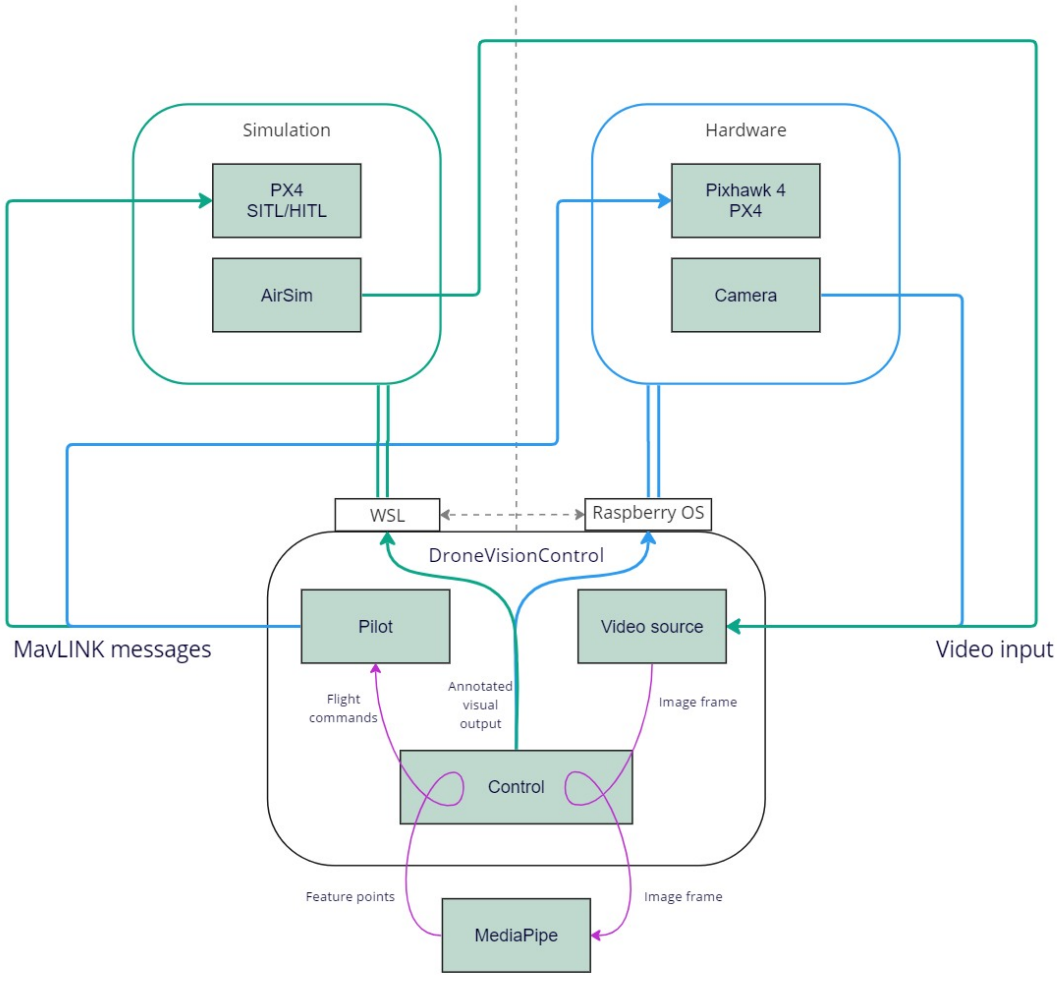


Figure 3.13: Structure of the DroneVisionControl application and its interactions with the necessary additional software for running in a simulation (green) or a vehicle (blue).

3.3.1 Pilot module

The pilot module[9] serves the purpose of providing access to the rest of the application for sending and receiving messages from the PX4 controller through the MavSDK library. This library offers a simple asynchronous API for managing one or more vehicles, allowing programmatic access to vehicle information, telemetry, and control over missions, movements, and other operations. The integration of MavSDK with the pilot module utilizes the Python library `asyncio` [**asyncio**], which enables running coroutines in parallel while waiting for messages provided through MAVLink communication.

To interact with the MavSDK library, all calls need to be written as async functions that await the result of one or more polls to the flight stack. The `asyncio` library provides support for writing concurrent code using the `async/await` syntax, serving as a foundation for various Python asynchronous frameworks used for high-performance network and web servers, database connection libraries, and distributed task queues. It offers a set of high-level APIs to run Python coroutines concurrently and grants complete control over their execution.

The pilot module, integrating MavSDK and `asyncio`, provides functionality to establish a connection to a PX4 vehicle through a physical serial address or a UDP endpoint. During this connection phase, the module will poll for internal information from the flight controller to decide when the system is ready to receive instructions. The MAVSDK library exposes telemetry and other state information through asynchronous generators, which are defined in Python as a convenient way to retrieve data asynchronously. These are accessed with the `async for` syntax.

The second task of the pilot module is to provide a queue for the control module to send actions to be executed in the vehicle sequentially. It implements many basic operations that can be executed in the flight controller, along with error handling and safety checks. These operations include takeoff, landing, return home, and manipulating the vehicle's flying velocity directly by providing speeds in body coordinates. The actions can be executed directly or added to the queue, which is periodically polled to run the actions in the order they are added. The module ensures that each action waits until the previous one has finished and the vehicle is in the desired state before starting the next action. It is also designed to allow a maximum time limit of 10 seconds for each action.

To enable direct control of the vehicle's velocity, a special flight mode defined by PX4, called Offboard mode[10], is required (not related to the offboard configuration described in Section 3.2.2). Offboard mode primarily controls vehicle movement and attitude, adhering to setpoints provided through MAVSDK. This mode relies on position or pose/attitude

[9] https://github.com/l-gonz/tfg-giaa-dronecontrol/blob/main/dronecontrol/common/pilot.py

[10] https://docs.px4.io/main/en/flight_modes/offboard.html#offboard-mode

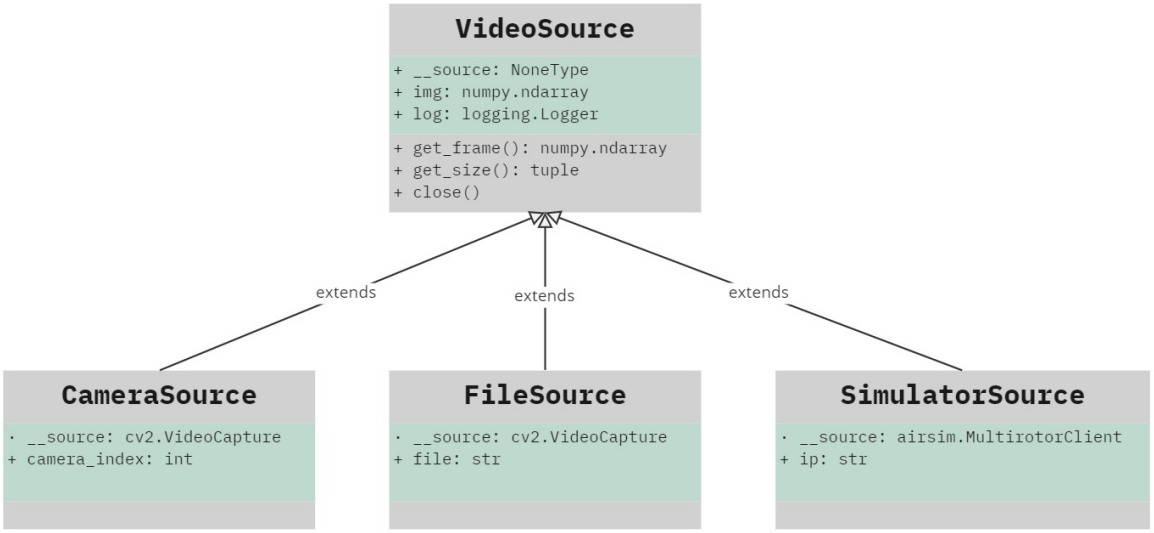


Figure 3.14: Diagram of inheritance on the video source classes available to retrieve image data.

information available to the flight controller, such as through a gyroscope and a GPS antenna. For safety purposes, this mode requires a constant stream of commands to be maintained. If the message rate falls below 2Hz or the connection is lost, the vehicle will come to a halt and, after a timeout, attempt to land or perform other failsafe actions based on the configured parameters. This behaviour is handled internally by the MAVSDK library and made available to the application by the pilot module through a `toggle_offboard_mode` function.

By encapsulating the functionality of sending commands and receiving information from the flight controller, the pilot module provides a robust interface for controlling the vehicle's behaviour and enables seamless integration with other components of the application.

3.3.2 Video source module

The video source module[11] aims to provide a collection of classes to retrieve images from different sources in a manner that allows easy interchangeability without affecting the rest of the application. This design facilitates testing and adaptability to various environments. Three classes of video sources have been implemented: file, simulator, and camera, all of which inherit from the `VideoSource` class, as shown in the diagram in Figure 3.14.

[11] https://github.com/l-gonz/tfg-giaa-dronecontrol/blob/main/dronecontrol/common/video_source.py

The `FileSource` class can open a video file stored on the companion computer and provide frames sequentially until the video is completed. This feature enables the replaying image detection algorithms on previously captured videos using the camera tool described in section 3.3.4. The `CameraSource` class can access a physical camera connected to the computer running the application via USB and provide real-time captured frames. Both the file and camera sources leverage OpenCV's video capture utilities to handle file operations and camera drivers.

The `SimulatorSource` utilizes AirSim's Python library, `airlib`, to communicate with the simulator and retrieve images from a camera object attached to the vehicle model in Unreal Engine. It establishes an automatic connection to the simulator via `localhost`, but it can also be initialized with an IP address to connect to a simulator running on a different computer within the local network. This is particularly useful when the DroneVisionControl runs inside a Linux subsystem (WSL) and the simulator operates on the host Windows system.

3.3.3 Vision control module

The control module encompasses the main logic of the application and is responsible for converting raw images obtained from the video source into commands for the pilot module. Two different types of control solutions have been implemented. The first one is the proof-of-concept control solution described in Section 3.4, operating in the offboard configuration outlined in Section 3.2.2. Its purpose is to translate predefined hand gestures into movement commands for the aerial vehicle. This solution facilitates testing the interaction between all system components in a contained environment by situating the controlling computer outside of the vehicle. The second control system is a follow mechanism, described in Section 3.5, which aims to mimic real-life scenarios where the control algorithms and camera reside onboard the vehicle. It tracks the location of a person detected in the images captured from the drone's perspective and uses that information to calculate velocities necessary for following and keeping the person centred in the drone's view.

Both solutions follow a similar process. Initially, the image is sent to the MediaPipe computer-vision third-party library, described in Section 2.2.1, which extracts the required features from the image in the form of 2D coordinates. Subsequently, various calculations specific to each solution are applied to these coordinates to determine the commands sent to the pilot module. Captured images, detected features and calculated results are communicated to a simple GUI, drawn with the help of the OpenCV library. This interface shows the user all the necessary information about the current state of the running application.

Sections 3.4 and 3.5 provide further explanations of the control modules used in the

two different solutions developed.

3.3.4 Camera-testing tool

In addition to the main modules, several utilities have been included in the DroneVision-Control program to facilitate the development and testing processes of the control solutions. The first tool is available in the `test_camera` module[12] and can be accessed through the command `dronevisioncontrol tools test-camera`. This tool serves multiple purposes, including testing the connection between the computer, the flight stack, and the camera without needing to attach any self-guided control mechanism. It also allows evaluation of the performance of the MediaPipe hand and pose machine learning solutions on real-time images. Additionally, it enables image capture and video recording from a live camera feed for subsequent analysis.

The test tool can be configured through command-line options to use any of the three available video sources (camera, simulator, or video file), connect to a hardware or simulated PX4 flight controller by specifying a connection string or IP, and run on any system acting as a companion computer. Computer vision can optionally be enabled to process incoming images using hand or pose recognition software. While the tool is running, basic commands such as takeoff, landing, or movement along any direction can be sent to a connected vehicle using the keyboard. Appendix B provides a comprehensive breakdown of all the tool's options.

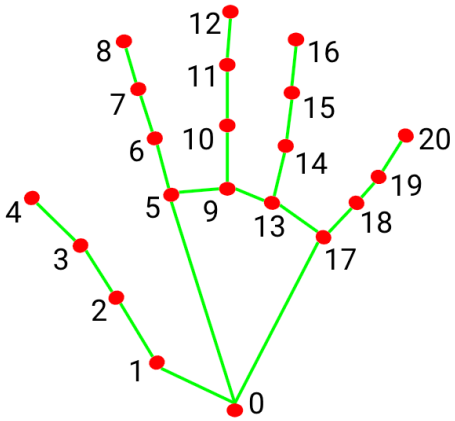
3.4 Proof of concept: hand-gesture solution

This solution was developed to test that the flow of the application works as expected, both in simulation and in actual flight, and that all the systems can interact and establish the required connections with each other. For that reason, it is designed to run with as little setup as possible. Flight tests can be undertaken with the minimal hardware components in the offboard configuration (Figure 3.8).

The control module for this solution is the `mapper` module[13]. It runs on a loop that continuously polls for a new frame from the chosen video source and feeds it to the hand detection functionality provided by MediaPipe [23]. If a hand is detected in the image, 2D coordinates called landmarks are extracted according to the mapping in figure 3.15. These

[12] https://github.com/l-gonz/tfg-giaa-dronecontrol/blob/main/dronecontrol/tools/test_camera.py

[13] https://github.com/l-gonz/tfg-giaa-dronecontrol/blob/main/dronecontrol/hands/mapper.py



0. WRIST
1. THUMB_CMC
2. THUMB_MCP
3. THUMB_IP
4. THUMB_TIP
5. INDEX_FINGER_MCP
6. INDEX_FINGER_PIP
7. INDEX_FINGER_DIP
8. INDEX_FINGER_TIP
9. MIDDLE_FINGER_MCP
10. MIDDLE_FINGER_PIP
11. MIDDLE_FINGER_DIP
12. MIDDLE_FINGER_TIP
13. RING_FINGER_MCP
14. RING_FINGER_PIP
15. RING_FINGER_DIP
16. RING_FINGER_TIP
17. PINKY_MCP
18. PINKY_PIP
19. PINKY_DIP
20. PINKY_TIP

Figure 3.15: Landmarks extracted from detected hands by the MediaPipe hand solution.

Source: Adapted from *Hands - mediapipe* [24].

landmarks are then converted into discrete gestures, such as an open palm, closed fist, or finger pointing in different directions. Each detected gesture is assigned a command, which is queued to the pilot module and executed once the previous commands have been completed.

The conversion from landmarks to gestures is performed by the `gestures` module[14]. Vectors are drawn from the base of each finger to its tip, as well as from the base of the hand (wrist feature) to the base of the fingers. The dot product vector operation is used to calculate the relative angles between each finger and the base of the hand, as shown in figure 3.16. By comparing these angles to a threshold, the module can determine whether each finger is extended or folded and in which direction it points. For example, an open hand gesture is detected when all five fingers are extended.

The program defines several gestures based on the calculated angles. These gestures are shown in Figure 3.17 and detailed in the following list:

- No hand: this happens when no landmarks can be extracted from the image. As a safety feature, the vehicle stops whichever previous commands it had in its queue and goes into Hold flight mode, hovering in the air while maintaining its position.
- Open hand: all five fingers extended, indicating a stop gesture. The drone holds at its current position.
- Fist: all five fingers folded into a fist. The drone arms and takes off if on the ground, or lands if already in the air.
- Backhand: the back of the hand is shown towards the camera, with the thumb pointing upwards and the other fingers pointing to the side. This gesture puts the

[14] https://github.com/l-gonz/tfg-giaa-dronecontrol/blob/main/dronecontrol/hands/gestures.py

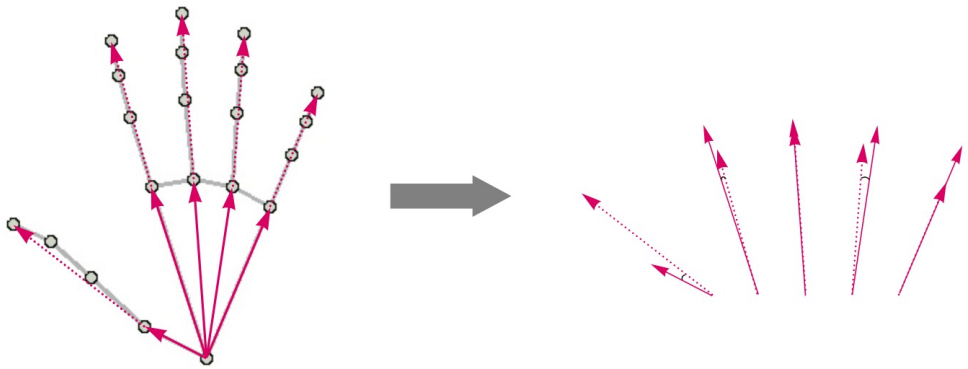


Figure 3.16: Vectors extracted from the detected features are used to calculate reference angles to determine hand gestures.

drone in Return flight mode, where it climbs to a safe altitude and returns to the last takeoff position.

- Index finger pointing up: The index finger is extended and pointing roughly towards the top of the image (within ± 30 degrees). The drone enters Offboard flight mode, allowing direct velocity commands. The drone remains in this mode as long as the finger is extended, and its movement can be controlled with any of the next four commands.
- Index finger pointing to the right: The index finger points to the right of the image (between 30 and 90 degrees from the top). The drone rolls towards its right side at a speed of 1 m/s.
- Index finger pointing to the left: Same as above, but the index finger points to the left of the image. The drone rolls towards its left side.
- Thumb pointing to the right: The index finger is extended up (to maintain Offboard flight mode) while the thumb is folded over the palm, pointing towards the right of the screen. This gesture makes the drone pitch forward at a steady speed of 1 m/s.
- Thumb pointing to the left: Similar to the previous gesture, but the drone pitches backwards when the thumb points to the left of the screen.

The program execution is outlined in figure 3.18. After all the initial parameters have been set, a secondary thread is started to run the pilot queue detailed in 3.3.1, which waits for new commands to be added. The main thread runs a GUI loop that continuously processes gestures calculated from retrieved images and generates actions that are queued for the pilot. It also recognizes user input on the keyboard to control the vehicle directly,

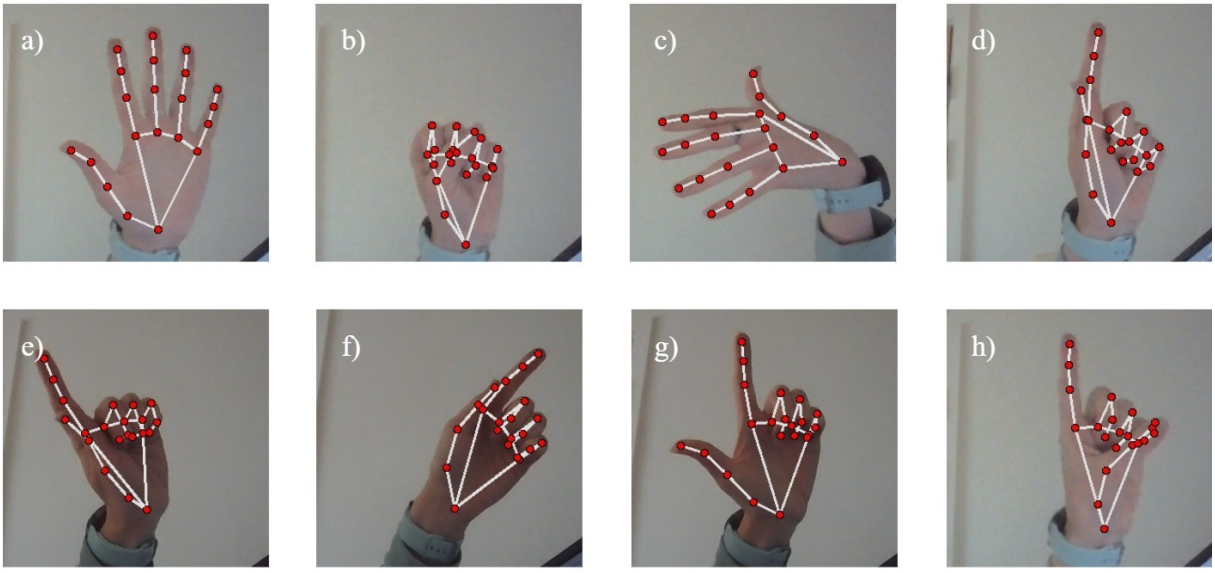


Figure 3.17: Gestures detected by the program to control the drone's movement. a) Open hand, b) fist, c) backhand, d) index point up, e) index point left, f) index point right, g) thumb point left, h) thumb point right

according to the mapping defined in the `input` module[15].

Link here wherever the mapping is detailed (Appendix?)

A complete run of this solution is detailed in Section 4.4.3.

3.5 Final solution: human following

The intention behind developing a UAV control solution that implements tracking and following of humans is to demonstrate the capabilities of the PX4 open-source development platform and its related projects, MAVLink and MAVSDK. The goal is to showcase how complex real-life applications can be designed without relying on expensive proprietary hardware. The follow application requires only a PX4-enabled flight controller installed in an aerial vehicle, a companion computer of appropriate dimensions mounted on board the vehicle, and a camera connected to the companion computer via USB.

In this solution, the vehicle will attempt to find a single person in its field of view and follow their movements by changing its yaw and forward velocity to match horizontal movements and distance changes, respectively. During program execution, the drone can

[15] https://github.com/l-gonz/tfg-giaa-dronecontrol/blob/main/dronecontrol/common/input.py

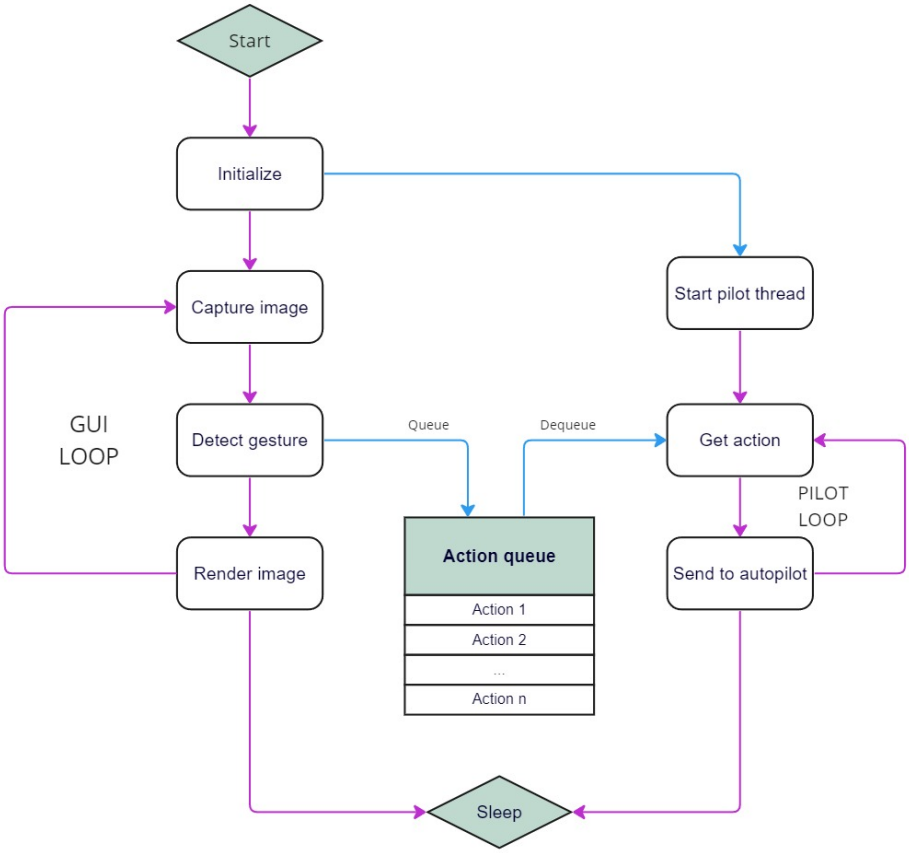
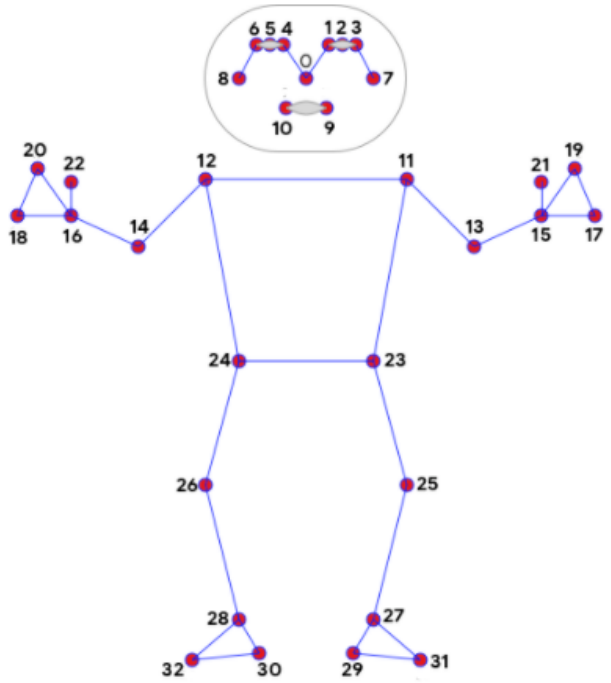


Figure 3.18: Execution flow for the running loop in the hand-gesture control solution.



0. nose
1. left_eye_inner
2. left_eye
3. left_eye_outer
4. right_eye_inner
5. right_eye
6. right_eye_outer
7. left_ear
8. right_ear
9. mouth_left
10. mouth_right
11. left_shoulder
12. right_shoulder
13. left_elbow
14. right_elbow
15. left_wrist
16. right_wrist
17. left_pinky
18. right_pinky
19. left_index
20. right_index
21. left_thumb
22. right_thumb
23. left_hip
24. right_hip
25. left_knee
26. right_knee
27. left_ankle
28. right_ankle
29. left_heel
30. right_heel
31. left_foot_index
32. right_foot_index

Figure 3.19: Landmarks extracted from detected human figures by the MediaPipe Pose solution

Source: Adapted from *Pose - mediapipe* [25]

be controlled via an RC controller, an external ground station application or keyboard input directly to the companion computer through, for example, a remote shell (SSH) or a desktop sharing program.

For safety reasons, the follow mechanism only engages when the flight mode on the vehicle is changed to Offboard mode. This can be done by activating a configured switch on the RC controller. The self-guided control automatically stops if the connection to the computer is lost or if any of the configured fail-safes are triggered, such as low battery, loss of RC or GPS signal, or exceeding predefined pitch and roll limits for a specified time.

In offboard mode with the follow mechanism engaged, the system continuously retrieves images from the onboard camera. These images are processed using the MediaPipe Pose [26] computer vision library to extract pose landmarks in the form of 2D coordinates. Figure 3.19 shows the features extracted by the algorithm and their correspondence to the human body. The extracted landmarks are used to draw a bounding rectangle around the detected person and validate that the received landmarks match the expected pose of a person standing up.

Figure 3.20 shows some examples of the validation checks running on the raw landmarks extracted from an image. To ensure controlled movements, the vehicle will stop and hover if it becomes impossible to detect a person in the image or if the detected features

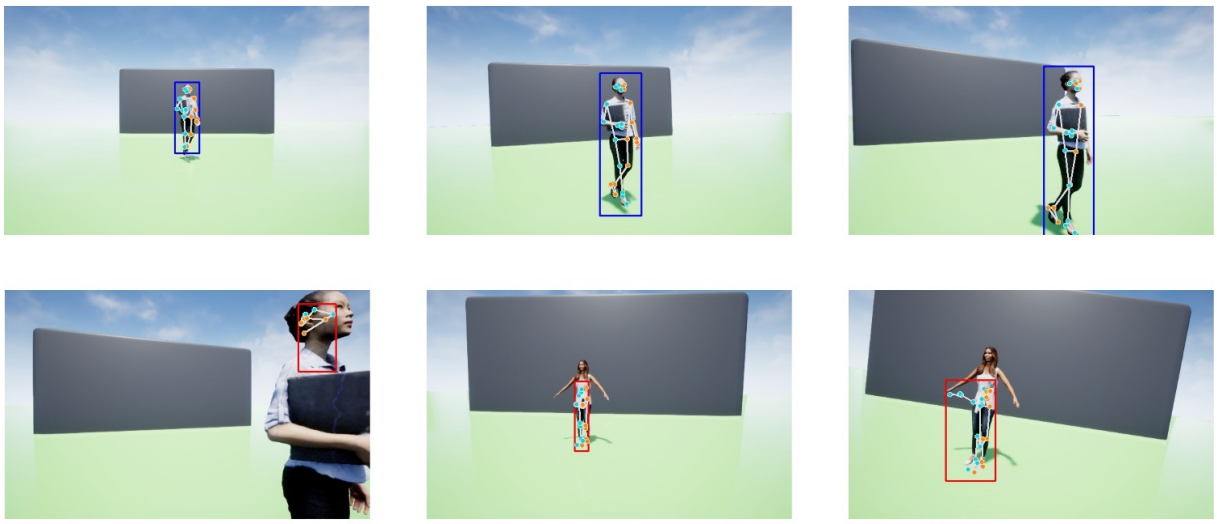


Figure 3.20: Valid versus invalid poses detected by the follow solution

do not match the expected pose. Once a valid bounding box is defined around the target person, its position relative to the camera's field of view is sent to a control mechanism consisting of two independent PID controllers. The theory behind these controllers is explained in section 3.5.1.

The first PID controller is responsible for controlling the yaw velocity of the vehicle to respond to horizontal movements of the person in the image. It aims to keep the person centred horizontally in the field of view by maintaining the x-coordinate of the bounding box's centre at the middle point of the screen. The second PID controller controls the forward velocity of the vehicle to respond to changes in the distance between the followed person and the drone. It adjusts the drone's forward movement to keep the height of the bounding box (as a percentage of the total image height) within a desired range. The target height for the controller needs to be set for each video source and desired distance empirically, as it depends on the camera used. Figure 3.21 shows how the inputs for each controller are extracted from the coordinates of the bounding box detected around the figure.

The follow[16] module's structure is summarized in a diagram in Figure 3.22. In each execution, after connecting to the pilot module with the starting options provided, the main loop runs continuously until the user quits the program. For each iteration, an image is retrieved, pose features are extracted, and a bounding box is calculated. If the vehicle is in Offboard flight mode, the calculated positions are used as inputs for the PID controllers, which generate velocity outputs to be sent to the pilot module. Additionally, manual keyboard control is available to send direct commands to the vehicle.

[16] https://github.com/l-gonz/tfg-giaa-dronecontrol/blob/main/dronecontrol/follow/follow.py

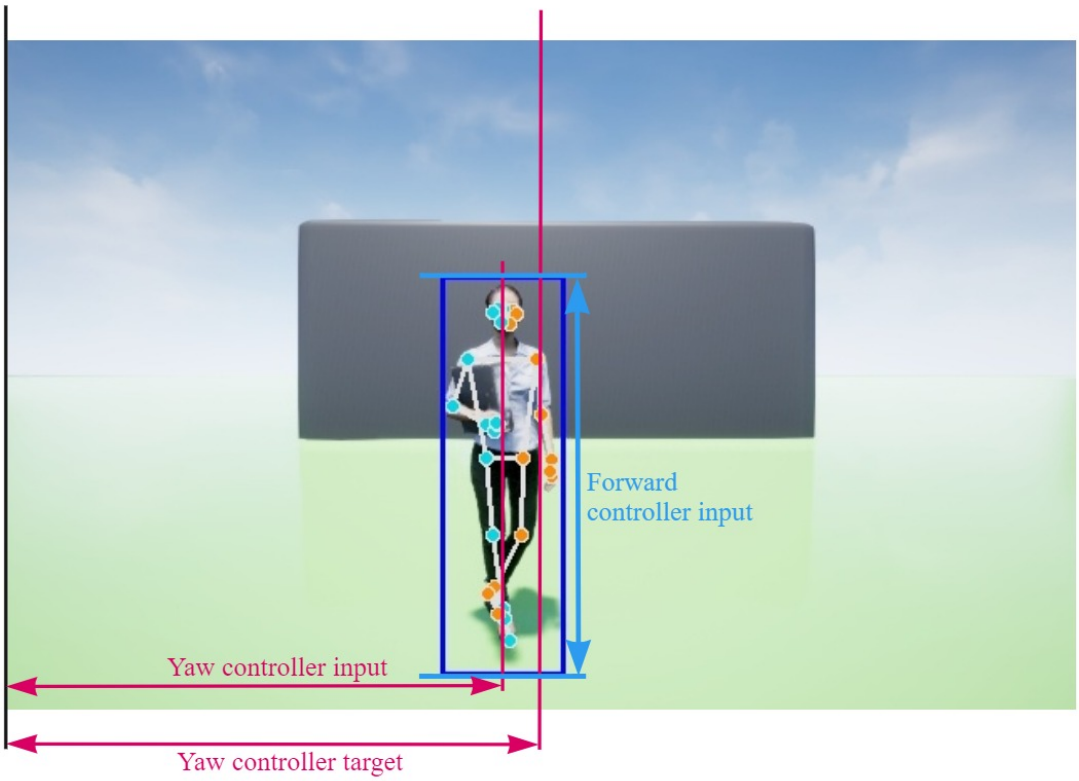


Figure 3.21: Calculation of horizontal position and height of figure from the detected bounding box.

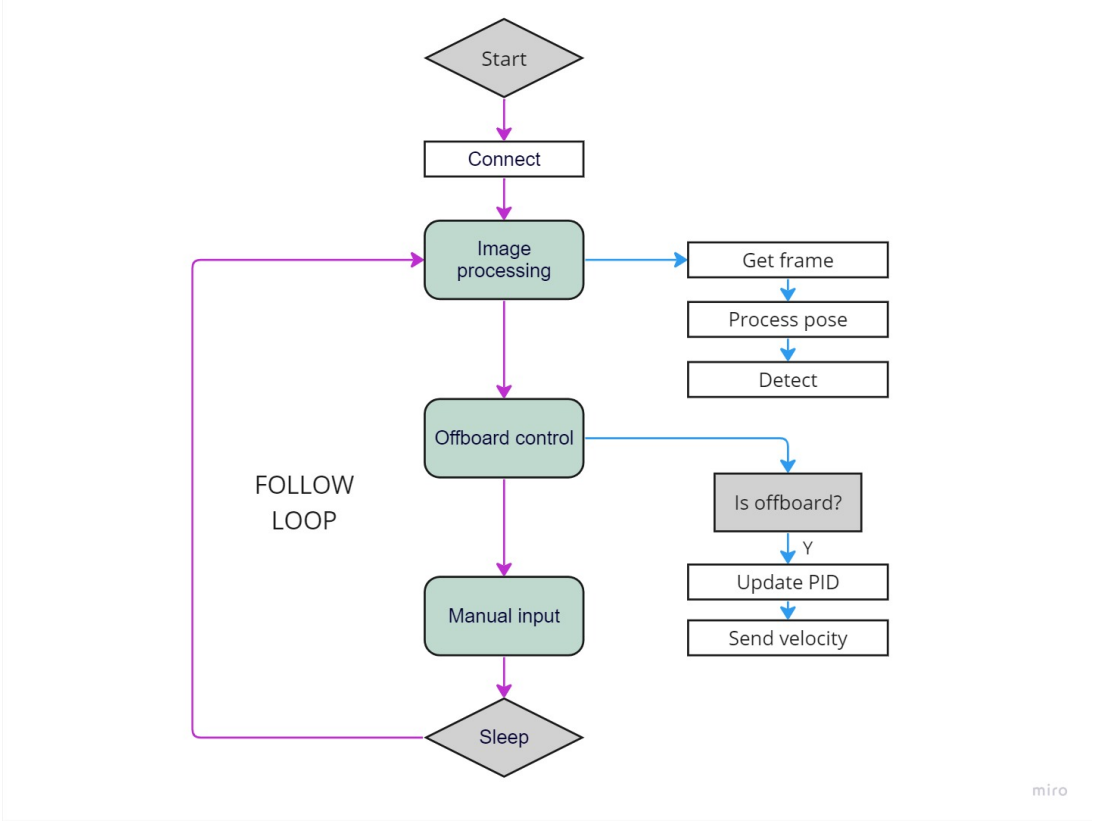


Figure 3.22: Execution flow for the running loop in the follow control solution

A full execution of this solution is shown in flight in Section 4.4.4.

3.5.1 PID tools

A proportional-integral-derivative (PID) is a control loop mechanism commonly used in systems requiring continuously modulated control. It works by continuously calculating an error value from the difference between the received input on a chosen process variable, which in this case is the detected position of a person in the frame, and the desired set point for that variable, a position centred in the frame. From the error value, the output for the controller is calculated according to this equation:

$$u(t) = K_p e(t) + K_i \int e(t)dt + K_d \frac{de(t)}{dt} \tag{3.1}$$

where: $u(t)$ = PID control variable
K_p = proportional gain
K_i = integral gain
K_d = derivative gain
$e(t)$ = error value
de = change in error value
dt = change in time

The PID controllers in this project are implemented with the help of the freely-available `simple-pid` library for Python [27], which supplies all the necessary calculations. It is only necessary to provide the coefficients or tunings (K_p, K_i, K_d) and the set point (target value) for the controller at the start and then update it periodically with the current input value to receive the output velocity. In the DroneVisionControl application, it is the `controller`[17] module the one that interacts with the `simple-pid` library to feed and receive the correct values to the PID controllers calculated from the bounding box around the detected person. Two additional utilities have been developed for tuning and testing the performance of the PID controllers.

The first utility is called `tune_controller`[18]. Tuning a PID controller typically involves testing different combinations of coefficients empirically. The `tune_controller` tool facilitates this process by allowing users to specify a range of potential coefficients and testing the system's response using images retrieved from AirSim and a simulated flight controller (SITL or HITL). The tool sets up a simulated person in an offset position from the target centre, engages the controller with the test values, and plots the detected position input and calculated velocity output on a graph for analysis. After each test, the vehicle returns to the starting position to reset the environment for the next coefficient to be checked. The tool can be executed using the command `dronevisioncontrol tools tune` and can be started with the option `-yaw` or `-forward` to tune a specific controller while deactivating the other one for better visualization. Each coefficient can be tested individually by providing fixed values for the other two parameters.

The second utility is called `test_controller`[19]. This tool is designed to evaluate the performance of an already-tuned controller with different position inputs for the camera. The objective is to assess how the controller reacts to varying relative positions between the vehicle and the person being followed. The tool engages both the yaw and forward controllers simultaneously to simulate real flight conditions. However, it can be run in either yaw or forward mode to vary distances in one of the two axes (left to right and

[17] https://github.com/l-gonz/tfg-giaa-dronecontrol/blob/main/dronecontrol/follow/controller.py

[18] https://github.com/l-gonz/tfg-giaa-dronecontrol/blob/main/dronecontrol/tools/tune_controller.py

[19] https://github.com/l-gonz/tfg-giaa-dronecontrol/blob/main/dronecontrol/tools/test_controller.py

forward to backwards, respectively). The tool measures how the vehicle responds to increasingly larger differences from the target position. The purpose is to ensure that the controller maintains a stable movement that ensures safety throughout the entire flight. The test tool's execution and results are detailed in section 4.2.3 of the project documentation.

3.5.2 Safety mechanisms

The DroneVisionControl application implements a very experimental vision-based guidance system. Therefore, to carry out flight tests in real-life conditions, it is necessary to ensure that there are sufficient safety mechanisms to prevent accidents. These measures include both software-based defences within the application code and safety configurations provided by the PX4 autopilot.

To begin with, the computer vision module in DroneVisionControl verifies the validity of objects detected as a person by examining specific features associated with a standing human figure. It checks that the height of the bounding box is greater than its width and that the features of the head, shoulder, hip, knee, and ankle are detected in the correct order from top to bottom. If any of these checks fail or if no detection output is received, the vehicle immediately enters Hold flight mode. In this mode, any queued velocity commands are discarded, and the drone hovers in its current position. These validation checks are implemented in the `image_processing`[20] module.

In addition to input validation, the application incorporates safety measures related to the PID controllers and the pilot module, which limit the maximum possible velocity of the vehicle at all times during the execution. The PID controllers have output limits set within DroneVisionControl to prevent abnormal velocity commands from being sent to the flight controller. Furthermore, the PX4 autopilot provides a fallback safety measure through its `MPC_XY_VEL_ALL` parameter, which limits the overall horizontal velocity of the vehicle. These limitations ensure that the drone's movements are within a safe and controllable range.

The PX4 autopilot itself offers various safety configuration options, known as failsafes, documented in *Safety Configuration (Failsafes) | PX4 User Guide* [28]. These failsafes detect undesired conditions during flight and include detecting a lost connection to the companion computer while in Offboard mode, a loss of RC transmitter link or GPS position, low battery levels during flight, and unexpected flipping of the vehicle. When any of these conditions are detected, the autopilot triggers a flight mode change to either Hold (hover) or Return (fly back to takeoff position and land), ensuring the safety of the drone and its surroundings.

[20] https://github.com/l-gonz/tfg-giaa-dronecontrol/blob/main/dronecontrol/follow/image_processing.py

The last safety mechanism to mention is not based on automatic detection by the developed software or the autopilot but on active surveillance of the system's behaviour during flight by the operator in control of the vehicle. With an RC controller configured with a switch to control flight mode, it is possible to deactivate Offboard mode at any moment, which will make the vehicle disregard all instructions from the companion computer and assume complete control of the vehicle either through a GPS-assisted mode or fully manual flight. A secondary switch in the RC controller can be configured as a kill switch for a last-resort option to stop all motor outputs immediately. This possibility is most useful when the vehicle is on the ground and cannot manage to take off upwards since there is a danger of breaking the propellers against the ground.

Lastly, operator control plays a vital role in ensuring safety during flight. The operator can use an RC controller with a designated switch to deactivate Offboard mode at any time. This action causes the vehicle to disregard instructions from the companion computer and assume control through GPS-assisted or manual flight modes. Additionally, a secondary switch can be configured as a kill switch, allowing for an immediate stop of all motor outputs when needed. This feature is particularly useful in situations where the vehicle is on the ground and has trouble taking off to reduce the risk of propeller damage if the vehicle flips.

By integrating these safety mechanisms, the DroneVisionControl application aims to mitigate potential risks and ensure safe flight operations under real-life conditions.

Chapter 4

Experiments and validation

This chapter outlines the validation process for each essential component required for the proper functioning of the entire system, ranging from initial simulation tests to comprehensive flight tests conducted on the final guidance solution. The validation process follows the order depicted in Figure 4.1.

The first section involves testing the application in a purely simulated environment. Initially, each part is tested individually, and subsequently, the flight controller simulation of PX4 is integrated with the flight mechanics simulation in AirSim, which receives control commands and the computer vision detection software running on the received images.

Moving on to the second section, a thorough examination is conducted on the follow solution detection and guidance results to ensure that the PID controllers only generate safe velocity commands for any received image input. Additionally, their response should align with the intended movement of the vehicle. To do this, first, each of the controllers is tuned individually with the help of the SITL simulation, and then their joint response is analyzed.

Once the software adheres to the required safety parameters, the third section moves the simulation to HITL mode on the dedicated hardware that will run the actual flights. This hardware includes the dedicated autopilot board and the standalone companion computer connected to it. The main goal is to verify that all connections are functioning as expected and that the devices deliver the required performance.

Lastly, in the fourth and final section, a series of flight tests are carried out, gradually increasing in complexity. These tests span from basic manual control of the vehicle using an RC controller to fully autonomous target-following flight executing in the complete system.

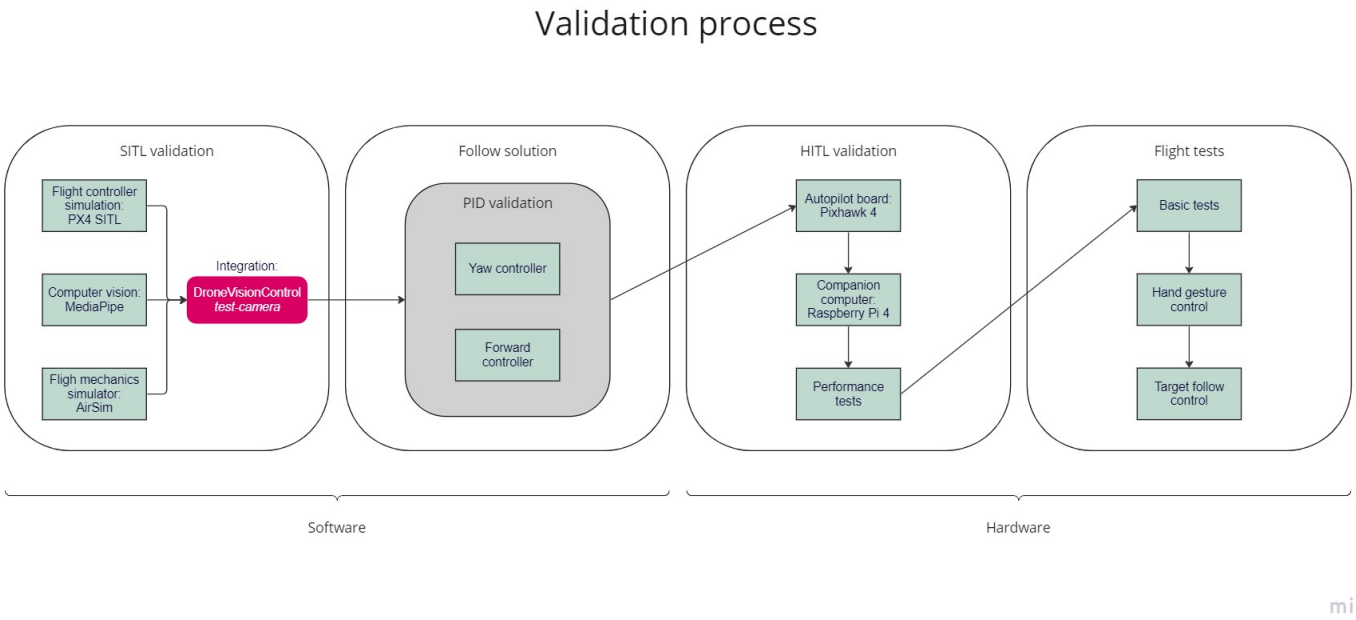


Figure 4.1: Outline for the validation process

4.1 PX4 SITL simulation and validation

Section 3.1 describes the software-in-the-loop simulation mode developed by PX4. This mode will enable testing and validating the correct operation of individual components of the program's architecture in several distinct steps before integrating them into one control flow. These initial tests will ensure that the foundational features of the control algorithms, sending commands to the autopilot and retrieving images for analysis, are dependable. To do that, the following steps will be checked in order:

1. Verify that it is possible to start the simulated flight controller and that it connects to the 3D simulator.
2. Connect the DroneVisionControl program to the flight controller and send basic commands.
3. Retrieve images from a camera and feed them to the computer vision detection utility.

To reduce the amount of configuration needed for the first test, the Gazebo[1] simulator will be used instead of AirSim, since it is already installed during PX4's SITL setup. Gazebo works natively on Linux, so it can run in parallel with the simulated flight stack and the developed application on the same WSL OS. To set up the Linux system, PX4 and DroneVisionControl are installed as detailed in Appendix A.1.

[1] https://gazebosim.org/home

Once both parts are installed, the simulation can be started with the `make px4_sitl gazebo` command or using the `simulator.sh` script found on the project repository[2]. The result can be seen in Figure 4.2, with the user interface and 3D world of the Gazebo simulator on the left side and the PX4 console on the right side. The console can be used to send commands to the vehicle and set configuration parameters for the simulation.

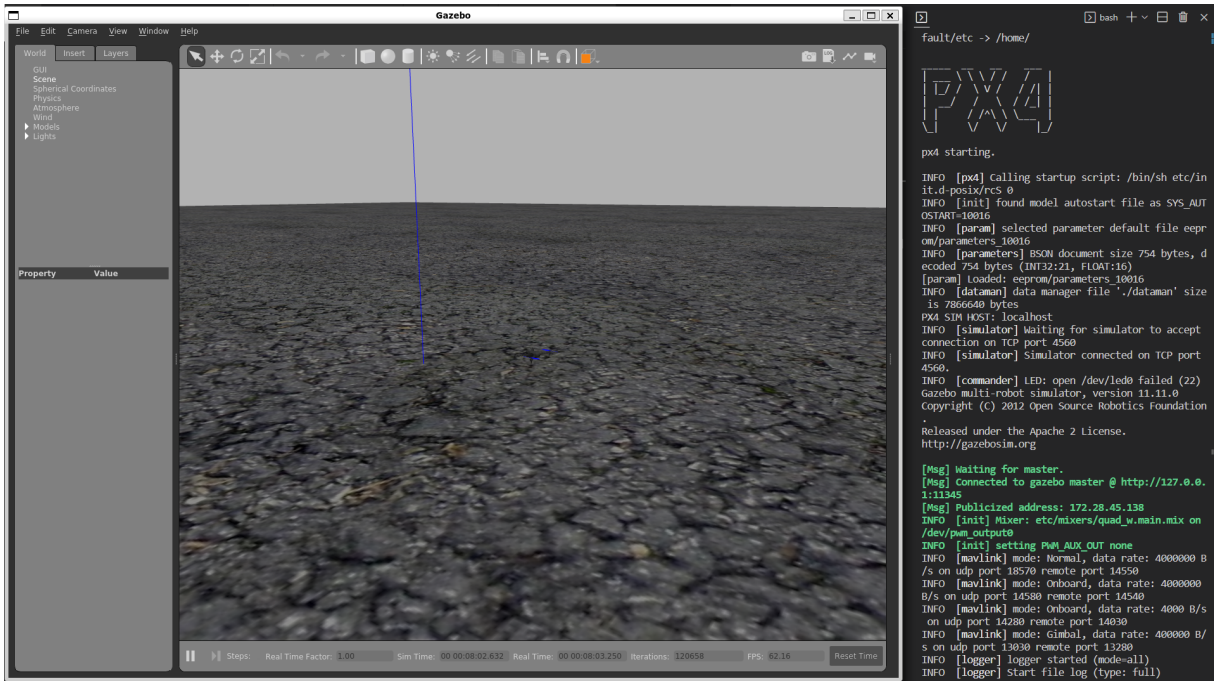


Figure 4.2: Gazebo simulator (left) and output from the PX4 console (right) after PX4's software-in-the-loop simulation is started.

The first command to test is takeoff, which is done by sending `commander takeoff` through the PX4 console. Figure 4.3 shows the simulator's state after the takeoff command, where the vehicle model has climbed to the default takeoff height of 2.5 meters above the ground. The command to land the vehicle again is `commander land`.

The second step is to connect the DroneVisionControl application to the simulation. The `test-camera` utility described in Section 3.3.4 has been developed specifically to test establishing a connection to PX4 SITL and process images without engaging any of the program's control modules. It can be started through the tools section of the application's command-line interface (`dronevisioncontrol tools test-camera`). Once the connection to the simulation is established successfully, movement commands can be sent to the vehicle through keyboard input. For example, pressing the "T" key will trigger takeoff. The result should be the same as sending the `commander takeoff` command through the PX4 console, verifying that the MAVSDK library and the pilot module work as expected.

[2] https://github.com/l-gonz/tfg-giaa-dronecontrol/blob/main/simulator.sh

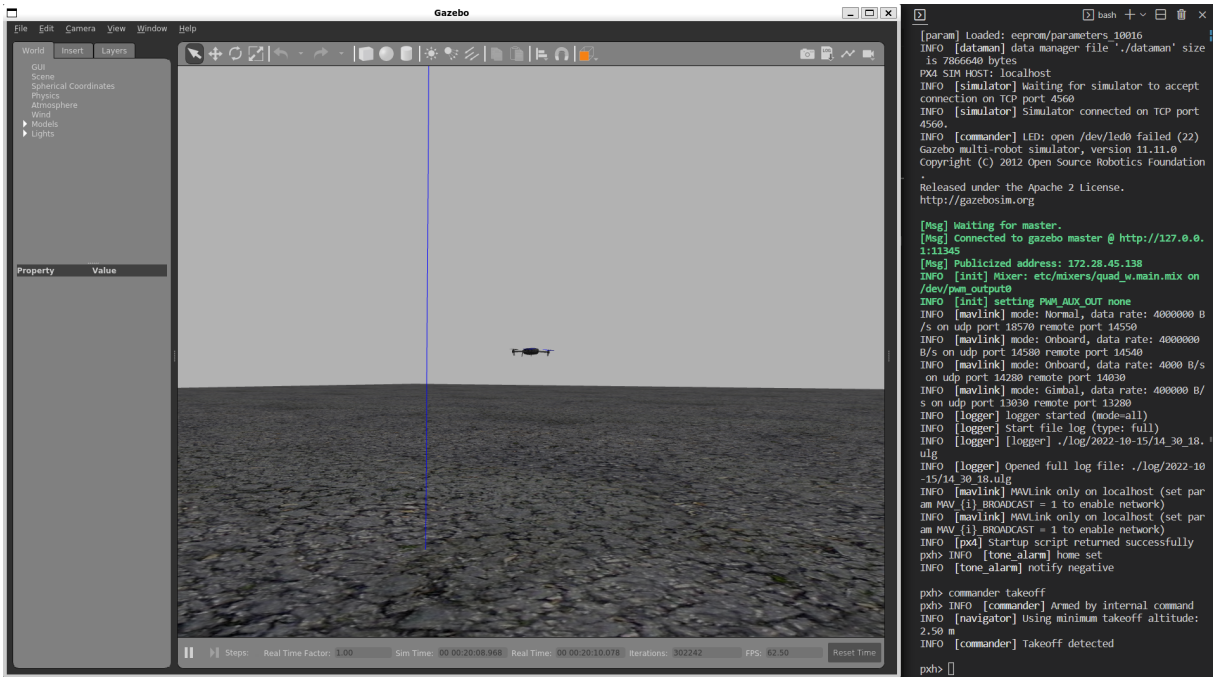


Figure 4.3: Gazebo simulator (left) and output from the PX4 terminal (right) after the takeoff command has been executed.

For the last step of this test, a camera will be added to the system to retrieve images that can be sent onward to the computer vision algorithm. The configuration used in the previous steps, however, needs some modifications, as WSL cannot access hardware devices or USB ports on the host computer. The simplest solution is to migrate the DroneVisionControl application to the Windows system. Since that is the environment chosen for the HITL tests that will be carried out later, it is also beneficial to validate at this point that the application can run without issues on Windows. Appendix A.1 contains the details on configuring the PX4 flight stack simulation to allow connecting to a MAVLink server through a different machine in the local network. Once the application is installed in the Windows system, the same `test-camera` utility can be run with the `-c` option to retrieve images from a connected camera. Additionally, the `-h` and `-p` options can be used to run the hand-detection and pose-detection algorithms, respectively, on the captured images.

Figure 4.4 shows the image and text output of the program when the `test-camera` tool is run with the hand-detection feature activated. On the left side, the detection algorithm tracks the shape of a hand detected in the image and on the right side, the logged information shows the connection being established and keyboard commands being sent to the simulator.

After testing the flight stack, the default simulator and the developed application with its image retrieval, it is time to add the AirSim simulator to the environment.

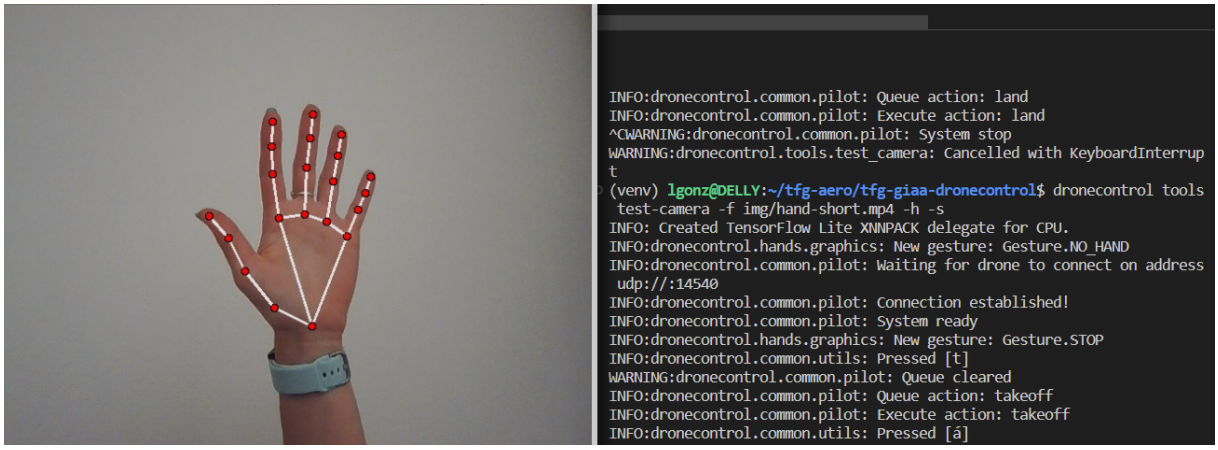


Figure 4.4: Hand detection algorithm running on images taken from the computer's integrated webcam.

4.1.1 PX4 SITL validation with AirSim

The end goal for the development environment is to use the AirSim simulator to take advantage of its 3D-rendering and computer vision capabilities. For this reason, it becomes necessary to validate that the new simulator can run correctly on Unreal Engine, interacting with PX4 as the default Gazebo simulator did. Additionally, all the necessary features for detection, tracking and following should work as expected. These characteristics will be validated in the order below:

1. Verify that the AirSim simulator can start, connect to the PX4 SITL through the WSL virtual network and receive commands from the PX4 terminal.
2. Integrate the previous tests with AirSim by running the hand-gesture control solution described in Section 3.4.
3. Check that the DroneVisionControl program can obtain images from the virtual camera inside AirSim's simulated world.
4. Test the pose recognition algorithms on the images obtained from the AirSim simulation.
5. Check that the follow solution can control the vehicle's velocity directly in PX4's offboard mode.

In the first place, the AirSim simulator needs to be installed in the Windows host. The complete installation process is described in appendix A.1.1. There are specific configuration parameters that have to be set to be able to connect the AirSim simulator in Windows to the PX4 SITL simulation running inside WSL. On the simulator side, AirSim's settings file has to include a line defining the IP address of the network interface to use.

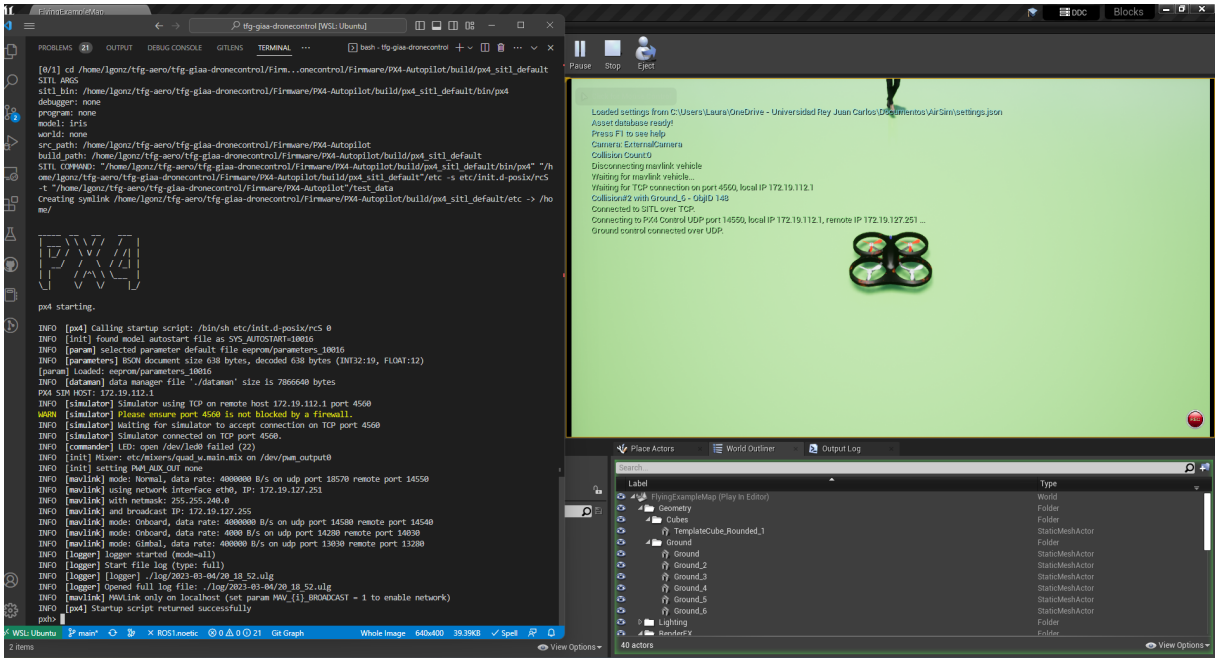


Figure 4.5: AirSim environment connected to PX4 flight stack running in SITL mode.

This parameter, along with the entire configuration file used for SITL testing, can be found in appendix A.3. On the PX4 side, it is necessary to specify that the simulator will attach through a different IP address than `localhost`. This is done by setting the `PX4_SIM_HOST_ADDR` environment variable in the Linux system to the IP address of the Windows host on the WSL virtual network before starting PX4 as follows:

```
export PX4_SIM_HOST_ADDR=[IP-address]
make px4_sitl none_iris
```

These commands start the software-in-the-loop execution, which attempts to attach to an already running simulator listening on the IP address specified and the TCP port 4560, in this case, AirSim. Therefore, every time either PX4 or AirSim stops its execution, both of them have to be stopped and the AirSim simulator restarted first. Figure 4.5 shows the testing environment after the AirSim simulator and the PX4 console have been started successfully. At this point, it is possible to use the PX4 console to send takeoff and land commands to the simulator and observe the 3D model of the vehicle climb into the air.

In the second test, all the previously validated individual components will be integrated with the AirSim simulator by executing the proof-of-concept control solution. Specifically, the DroneVisionControl application will run with the gesture-based control loop enabled. This test will be conducted on the Windows system to be able to rely on images from the physical camera. The complete execution is shown in the video[3] accessible through this link. Additionally, a frame extracted from the video can be observed in Figure 4.6.

[3] https://l-gonz.github.io/tfg-giaa-dronecontrol/videos/test-sitl-hand

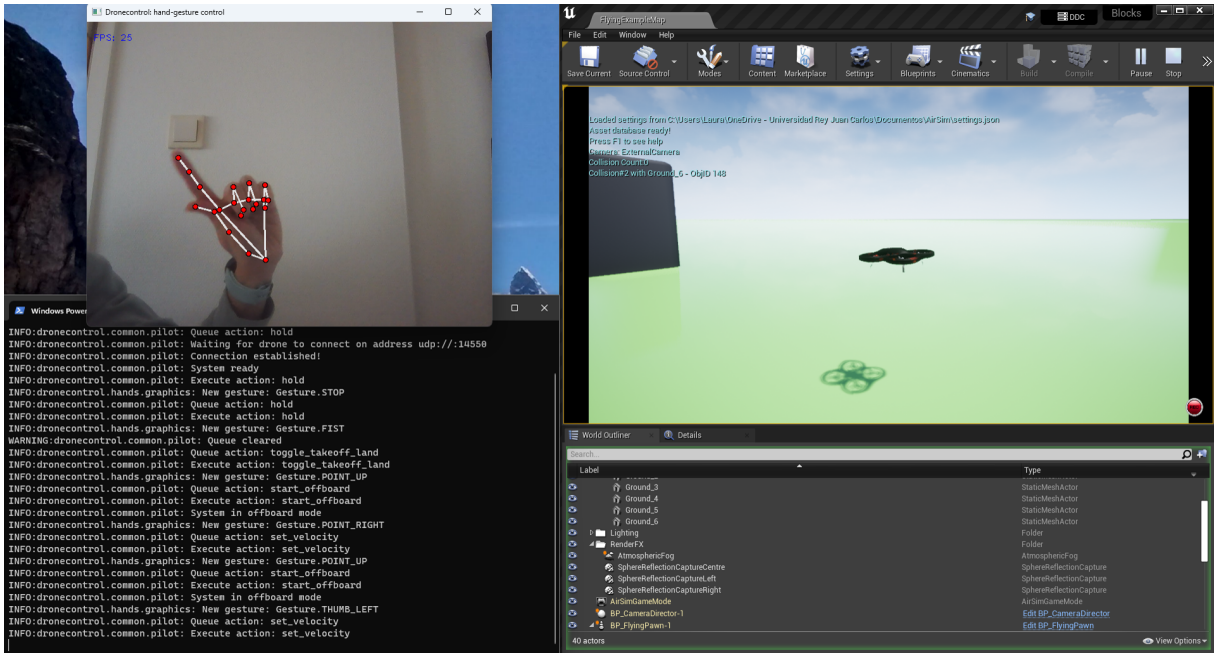


Figure 4.6: Single frame extracted from the video of the full execution of the hand-gesture control solution. Gesture detection is shown on the upper left side of the screen. The lower left side shows the mapping between detected gestures and commands, and the right side shows the vehicle's movement response inside the simulator.

The validation process will now shift its focus towards the tools required to execute the pose detection and tracking mechanism. To assess the detection and tracking of human figures from images captured within the simulator, the camera testing tool provided with DroneVisionControl can be employed once again.

Figure 4.7 demonstrates the output obtained when running the tool with a 3D model of a person positioned in front of the drone within the simulated environment. The following command was executed:

```
dronevisioncontrol tools test-camera --wsl --sim --pose-detection
```

In the image, the computer vision utility successfully detects the key features of the human body, outlining them with a bounding box. Simultaneously, the program's logged output displays two calculated positions in the terminal: the x-coordinate of the midpoint within the bounding box and the percentage of the image height covered by the height of the bounding box. These two values serve as inputs for the PID controllers that drive the follow solution, as described in Section 3.5. The logs can be employed to calibrate the distance at which the drone is to track the person when the control mode is engaged. This is done by setting the simulated vehicle at the target distance from the person model and using the output height percentage as the set point for the forward PID controller.

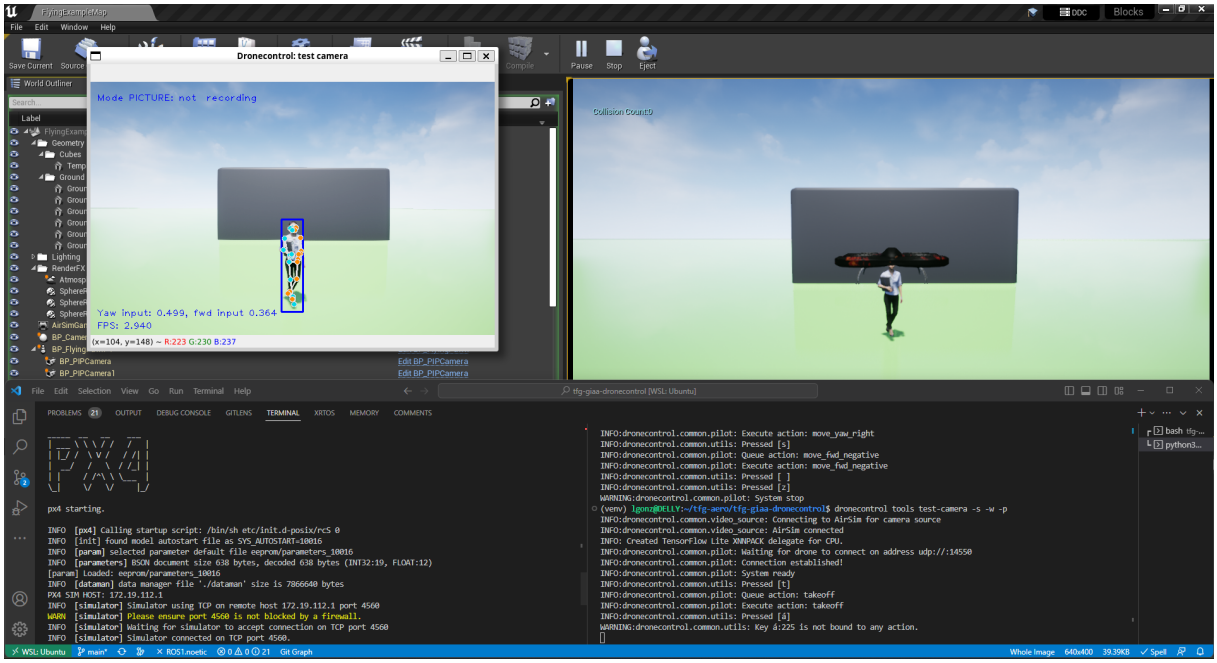


Figure 4.7: AirSim, PX4 and DroneVisionControl applications running side-by-side and connecting to each other

Before using the testing environment to fine-tune the PID controllers in the follow solution based on the vehicle's response, it is essential to confirm that the controllers can appropriately react to changes in the figure's position. This can be achieved by enabling only the proportional term of the controllers with a suitably low magnitude, ensuring slow and smooth movement. The expected result is that the vehicle starts moving forward when the person moves backwards and that it starts turning to the right when the person moves to the right, mirroring the movement in the remaining directions.

To run the follow control program with specific values for the proportional terms of the yaw and forward controllers (10 and 2, respectively), the following command can be used:

```
dronevisioncontrol follow --sim --yaw-pid (10, 0, 0) --fwd-pid (2, 0, 0)
```

4.2 PID controller validation

As mentioned earlier, the person-following mechanism relies on two PID controllers, which serve as the core of the velocity controller. Achieving stable vehicle movement requires tuning the parameters of these controllers to find an appropriate combination.

This section focuses on empirically determining the coefficients for the project's PID

controllers. To accomplish this, a dedicated tuning tool, described in Section 3.5.1 and specifically developed for this purpose, will be utilized. The performance of this tool will be validated using the controller testing tool within the simulated environment, with the flight stack operating in SITL mode. This setup allows for continuous measurement of the PID controllers' response to step movements of a simulated person situated in the 3D world.

Initially, each controller will be independently tuned, enabling the vehicle to move in only one direction at a time. Once the individual tuning is completed, both controllers will be engaged simultaneously to assess their combined response to a variety of inputs. Finally, the complete follow control mechanism will be executed using the obtained tuning parameters to evaluate the final behaviour of the system.

Before calibration, it is necessary to decide on a reference position to set the target values at the controllers. In the world coordinates of the simulated environment, these positions will be x=0 and y=0 for the vehicle and x=600 and y=0 for the person, as depicted in Figure 4.8. From these locations, the person is detected by the vehicles camera centred in the field of view, with a calculated height of 36% of the image height. Consequently, the controllers running within the simulator will have target set points of 0.5 for the yaw controller and 0.36 for the forward controller. The result of these target points is that, when the execution starts, for any changes in the position of the person the controllers will transmit velocity commands to the autopilot to achieve the same relative position between the vehicle and the person as between the reference positions.

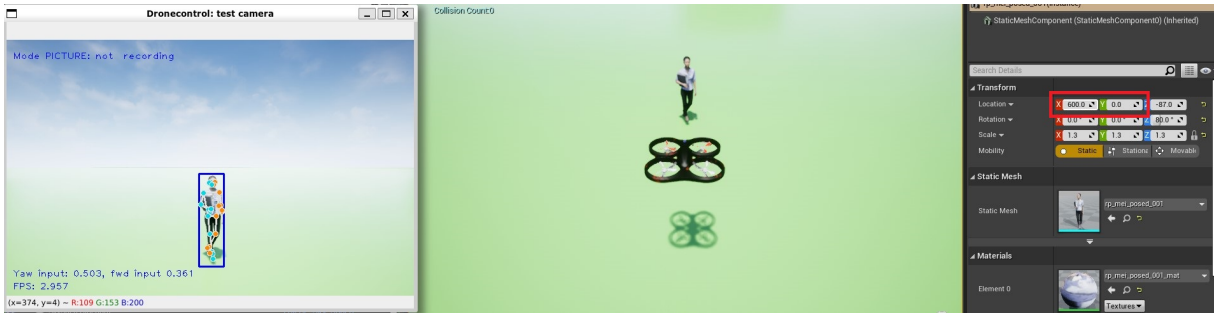


Figure 4.8: Reference position for the yaw and forward PID controllers. From left to right, the panels show the DroneVisionControl application window, the AirSim simulator world view and the world location of the human model in the simulator. The distance between the vehicle and the person is 600 units in the x direction and 0 units in the y direction.

4.2.1 Yaw controller

To determine the correct coefficients for the controller governing the yaw velocity of the vehicle, the target person is positioned slightly to the side within the field of view. This setup ensures that when the controller is engaged, it outputs a rotation of the vehicle

towards that side. Since forward control is not relevant for this test, the target person can be placed closer to the camera to facilitate accurate landmark detection by the pose detection algorithm.

Figure 4.9 illustrates the starting position of the simulated environment before each run. The 3D model representing the person is situated at coordinates (500, 100), which is 100 units to the right of the reference position. In the leftmost panel of Figure 4.9, the DroneVisionControl application displays an input of 0.62 for the yaw controller at this position. The controller will therefore calculate an error of $\epsilon = 0.5 - 0.62$ (set point minus input) at the start of the test.

The controller must then generate a positive yaw velocity to centre the person within its field of view. With a starting offset position to the right, the error is less than zero. However, a positive velocity is required to counteract this error and induce a rightward yaw velocity. Therefore, the coefficients for the yaw controller need to be negative. This ensures that an increased horizontal position results in a positive yaw velocity to decrease it, as indicated by Equation 3.1.

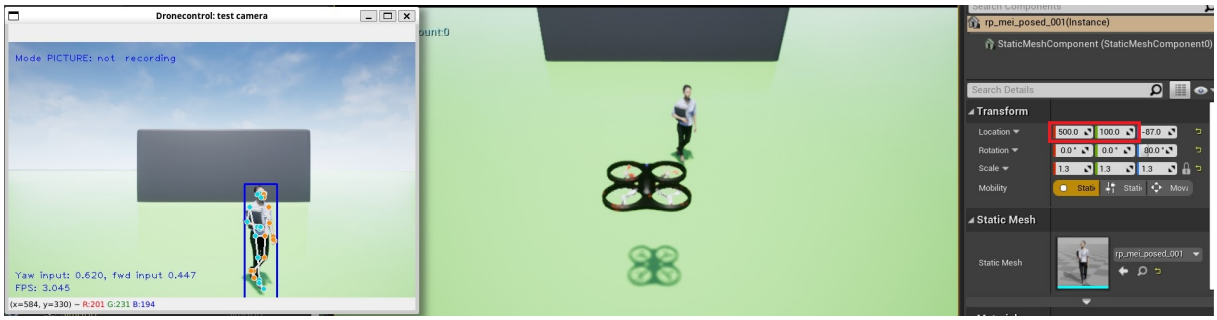


Figure 4.9: Starting position of the simulator for tuning the yaw controller. The human model is situated 500 units forward and 100 units to the right of the vehicle model.

To tune the controller to its correct coefficients, the initial step will be testing different values of K_P while keeping K_I and K_D at zero. Figure 4.10 displays the results of executing the tuning program for the five tested values of K_P, which range from $K_P = -10$ to $K_P = -90$ in increments of 20. The left graph of the figure represents the variation in the horizontal position detected by the camera during the first 30 seconds after activating the controller. The right graph illustrates the yaw velocity outputted by the controller to the pilot module in order to reach the target.

For low values of K_P, the controller causes the vehicle to move slowly towards the target, resulting in a longer time to reach the midpoint position. Conversely, for high values of K_P, the controller attempts to rapidly approach the target, leading to oscillations around the target when it gets close. It is worth noting that the right graph demonstrates how the output velocity of the yaw controller is limited to 5 degrees per second. Thus, even with very high values of K_P, the vehicle will not rotate faster than this limit.

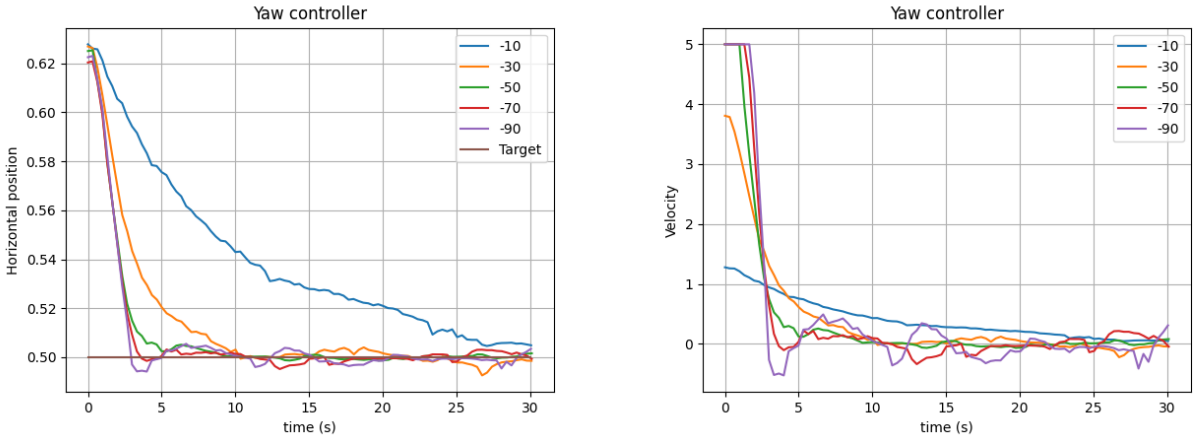


Figure 4.10: Variation of (a) input position and (b) output velocity for different values of K_P and $K_I = 0$, $K_D = 0$ while the yaw controller is engaged.

From the graphs, it can be observed that the most optimal value among those tested is $K_P = 50$. At this value, the distance to the target point decreases rapidly (left graph), while the velocity does not exhibit significant fluctuations around zero (right graph).

The second step involves finding the correct value for K_I. To accomplish this, multiple values of K_I will be tested while keeping K_P at a low value of -20 and K_D at 0. This setup allows for easier observation of the impact of the integral part on the controller.

Figure 4.11 illustrates the behaviour of the input and output of the controller for a sample time of 40 seconds for each tested value of K_I. When $K_I = -1$, progress towards the target is stable and slightly faster compared to when the integral part is not contributing. However, as the magnitude of K_I increases, noticeable oscillations around the target position emerge initially, then gradually diminish over time. For very large values of K_I, approximately from $K_I = -10$ in this case, the vehicle's velocity becomes locally unstable with numerous slight variations in its oscillations.

A similar effect can be observed to a lesser extent in Figure 4.12, where measurements were taken for $K_P = -50$ and $K_D = 0$. In this graph, with $K_I = -1$, the controller reaches the target position approximately 3 seconds faster, exhibiting similar oscillations in velocity compared to exclusively using of the proportional term (Figure 4.10, for $K_P = -50$).

In the final step, the tuning process focuses on the derivative part of the controller. Multiple values of K_D have been tested in conjunction with the chosen $K_P = -50$, both without any integral part ($K_I = 0$) and with $K_I = -1$. This combination aims to determine the impact of the derivative part on the controller and verify whether the integral part chosen in the previous step can work alongside it to enhance the controller's response to a changing input.

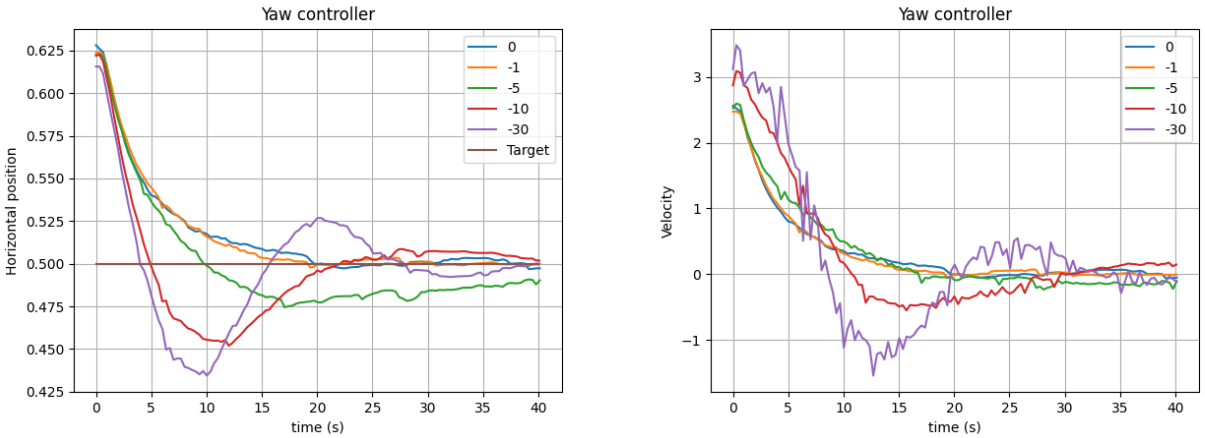


Figure 4.11: Variation of (a) input position and (b) output velocity for different values of K_I and $K_P = -20$, $K_D = 0$ while the yaw controller is engaged.

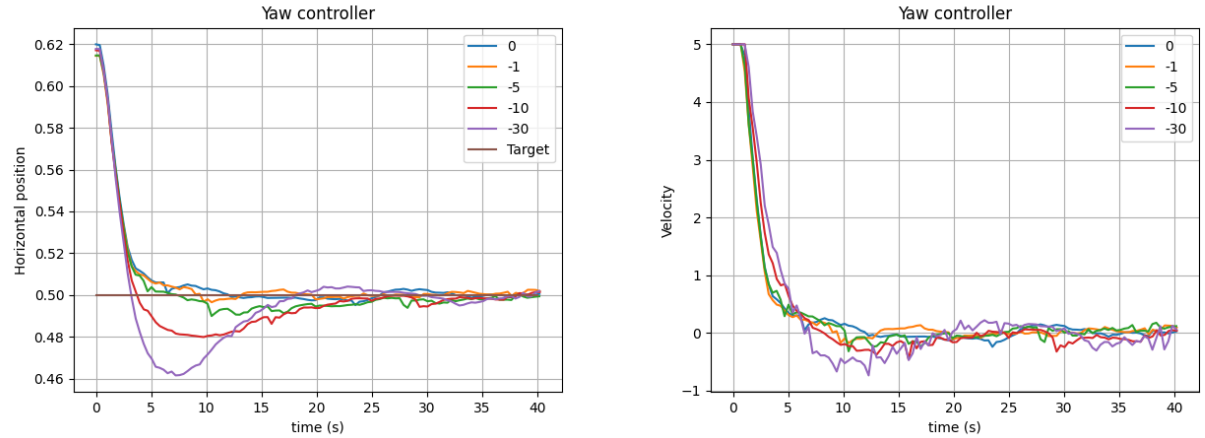


Figure 4.12: Variation of (a) input position and (b) output velocity for different values of K_I and $K_P = -50$, $K_D = 0$ while the yaw controller is engaged.

Figures 4.13 and 4.14 depict the evolution of the position detected by the computer vision system and the velocity outputted by the controller for a sample time of 40 seconds. Figure 4.13 corresponds to the case where $K_I = 0$, while Figure 4.14 represents the case with $K_I = -1$, both with $K_P = -50$.

In all the tested scenarios, the iteration with $K_I = -1$ (Figure 4.14) demonstrates better convergence towards the target positions compared to their respective K_D values for $K_I = 0$. This indicates that the integral part has been appropriately chosen. Moreover, it is worth noting that adding any amount to the derivative part does not yield any visible benefit in the step response of the controller. The curve that stabilizes first on the target position is still the one with $K_D = 0$. Additionally, the velocity graph produces very similar results between $K_D = 0$ and $K_D = -5$.

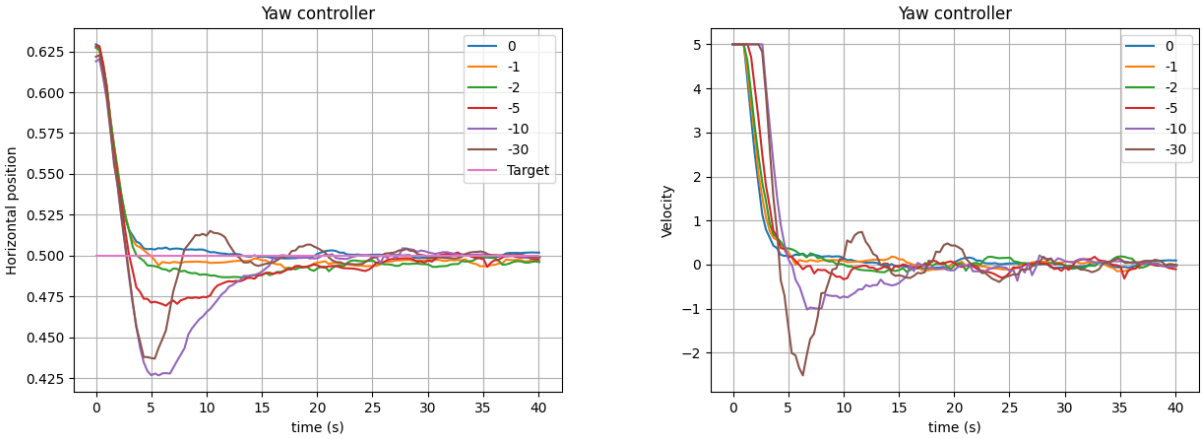


Figure 4.13: Variation of (a) input position and (b) output velocity for different values of K_D and $K_P = -50$, $K_I = 0$ while the yaw controller is engaged.

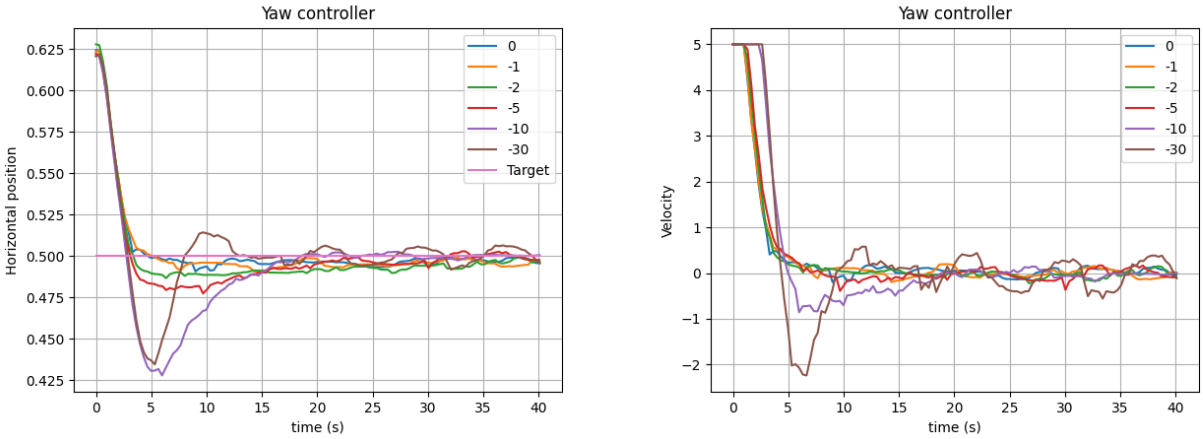


Figure 4.14: Variation of (a) input position and (b) output velocity for different values of K_D and $K_P = -50$, $K_I = -1$ while the yaw controller is engaged.

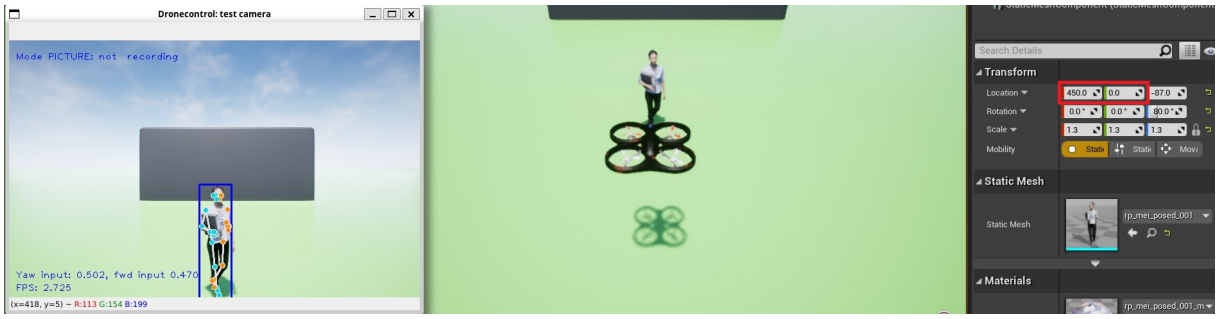


Figure 4.15: Starting position of the simulator for tuning the forward controller. The human model is situated 450 units forward and centred from the vehicle position.

Based on these observations, the final coefficients for the yaw controller will be determined as $K_P = -50$, $K_I = -1$, and $K_D = 0$. Consequently, the controller will effectively function as a PI (proportional-integral) controller rather than a complete PID (proportional-integral-derivative) controller. A recording of the whole tuning process for the yaw controller can be seen in this video[4].

4.2.2 Forward controller

The tuning process for the forward controller follows a similar procedure as the one used for the yaw controller. The initial setup for tuning involves positioning the figure closer to the vehicle than the reference position and ensuring it is centered in the vehicle's field of view. Figure 4.15 illustrates this starting configuration, with the figure located at the (450,0) position in the simulated world.

At this position, the input to the forward controller is 0.47, indicating that the bounding box around the detected figure occupies 47% of the camera's field of view height. Consequently, the controller's response should be a negative forward velocity that moves the vehicle away from the target person, thereby reducing the perceived figure height. Since a negative output velocity directly reduces the input at the entrance of the controller, the coefficients for this PID controller should be positive. This is in contrast to the yaw controller, where the feedback loop was inversely proportional (a positive velocity decreased the input to the yaw controller).

To begin the tuning process for the forward controller, the starting coefficients should be selected as it was done with the yaw controller. However, in this case, the forward velocity needs to be smaller than the yaw velocity to prevent destabilization of the camera caused by rapic pitch angle changes induced by the forward movement. Therefore, the coefficients to test for the forward controller will be reduced by one order of magnitude compared to the yaw controller.

[4] https://l-gonz.github.io/tfg-giaa-dronecontrol/videos/tune-yaw-controller

Initially, values ranging from $K_P = 1$ to $K_P = 9$ in increments of 2 are tested, with $K_I = 0$ and $K_D = 0$, for a sample time of 30 seconds. The resulting curves generated by the controller for these coefficients are shown in Figure 4.16. It is observed that the forward controller is generally more unstable compared to the yaw controller due to the influence that fast forward and backward movements have on the pose detection algorithm.

This is particularly visible at the start of each test as slightly different heights are detected in the image from the camera even though the vehicle is in the same position relative to the human figure. The high sensitivity of the detection mechanism creates an aditional effect, especially for the bigger K_P values tested. As the vehicle begins its movement back to the target position, the pose detection mechanism captures a slightly different perspective of the followed person, leading to increases in the detected height. This increase is reflected in the small spikes at the beginning of the detected height graph on Figure 4.16, despite the fact that the velocity graph shows a negative velocity, indicating that the vehicle is moving backward away from the person.

To mitigate this effect, it is important to keep the output forward velocity small in the controller. The right graph of Figure 4.16 also demonstrates that for high values of K_P, the output velocity increases significantly, reaching the maximum velocity limit of 0.4 $^m/_s$ set for the forward controller. A value of K_P up to 3 provides a rapid descent without significant oscillations around the target height, making it a suitable choice for the final controller.

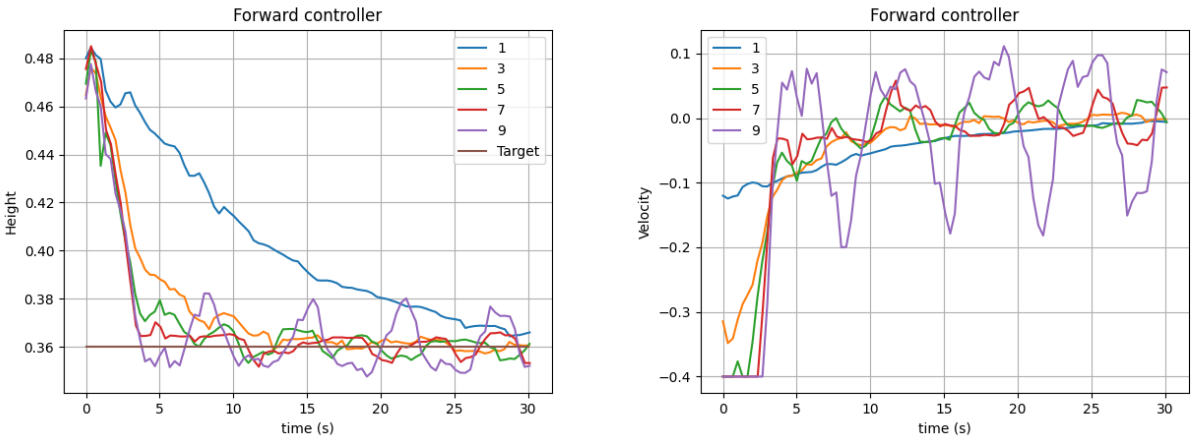


Figure 4.16: Variation of (a) input height and (b) output velocity for different values of K_P and $K_I = 0$, $K_D = 0$ while the forward controller is engaged.

The tests for K_I and K_D in the forward controller are depicted in Figures 4.17 and 4.18, respectively. Increasing the coefficient for the integral part leads to initial overshooting of the target position, followed by stabilization and approximation to the target. However, since overshooting is not desirable in this application due to safety concerns, a value of $K_I = 0$ is chosen for the forward controller.

In the case of the derivative component, all tested values of K_D lead to increased oscillations in the system. Therefore, it is decided to exclude the derivative part from the forward controller. As a result, the final coefficients for the forward controller will be $K_P = 3$, $K_I = 0$, and $K_D = 0$, making it simply a proportional controller without integral or derivative components.

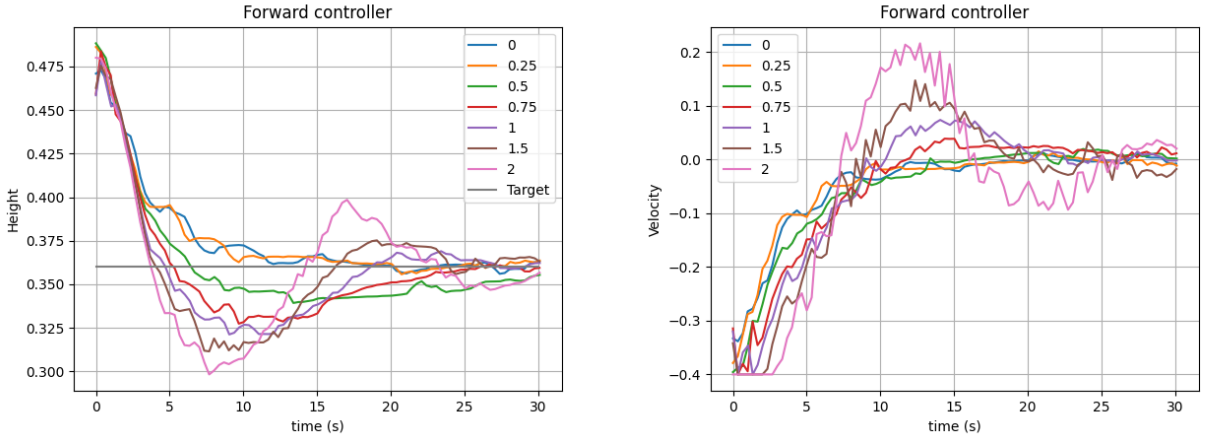


Figure 4.17: Variation of (a) input height and (b) output velocity for different values of K_I and $K_P = 7$, $K_D = 0$ while the forward controller is engaged.

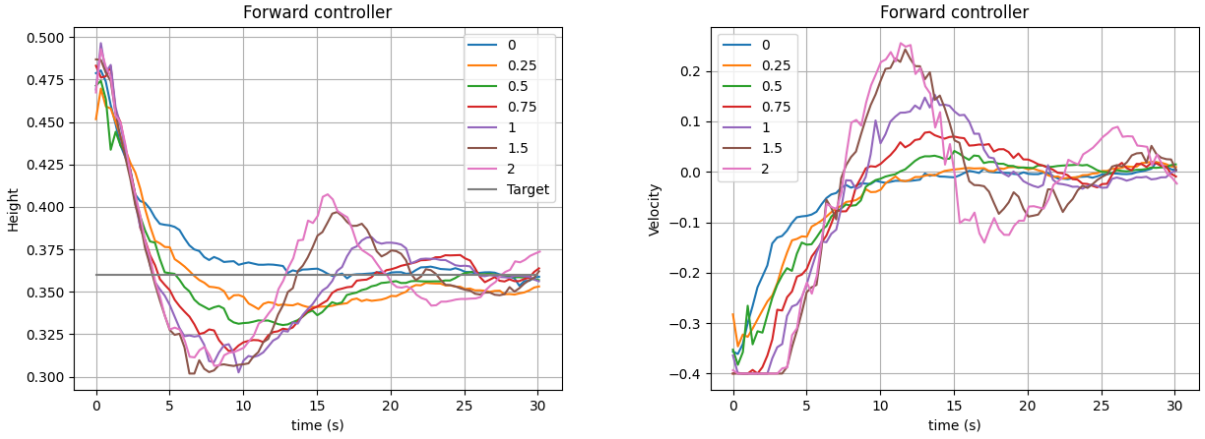


Figure 4.18: Variation of (a) input height and (b) output velocity for different values of K_D and $K_P = 7$, $K_I = 0.5$ while the forward controller is engaged.

4.2.3 PID tuning validation

The final validation of the tuning obtained for the controllers will be performed using the `test-controller` tool described in Section 3.5.1. The goal is to check the step response of the controllers for different starting distances and validate their performance when

engaged simultaneously. For the first test, the starting distances will vary along the y-axis, and for the second one, they will vary along the x-axis.

In the first test, the positions will vary along the y-axis, meaning that the figure will move from left to right in the field of view of the vehicle. The y-coordinates to be tested will range from -150 to 150 units in increments of 50, while the x-coordinate of the figure in the simulated world will remain fixed at $x = 500$.

The results of the first run are shown in Figure 4.19. The y-coordinates tested range from -150 to 150 units in increments of 50, while the x-coordinate of the figure in the simulated world remains fixed at $x = 500$. These changes in position mean that the figure moves from left to right in the field of view of the vehicle, following a line parallel to the lateral axis of the vehicle. To counteract this movement, both the yaw and forward controllers need to engage to reach the reference position.

The plots in Figure 4.19 depict the changes in the normalized horizontal distance and normalized figure height detected by the person recognition algorithm during the time it takes for the vehicle to reach the target distance from the human figure for each tested start position. The target is considered reached when the error is less than 2% and the output speed at the controller is less than 10% of the maximum value. The full execution of the `test-controller` tool with varying lateral positions can be seen in the video found in the project's page[5].

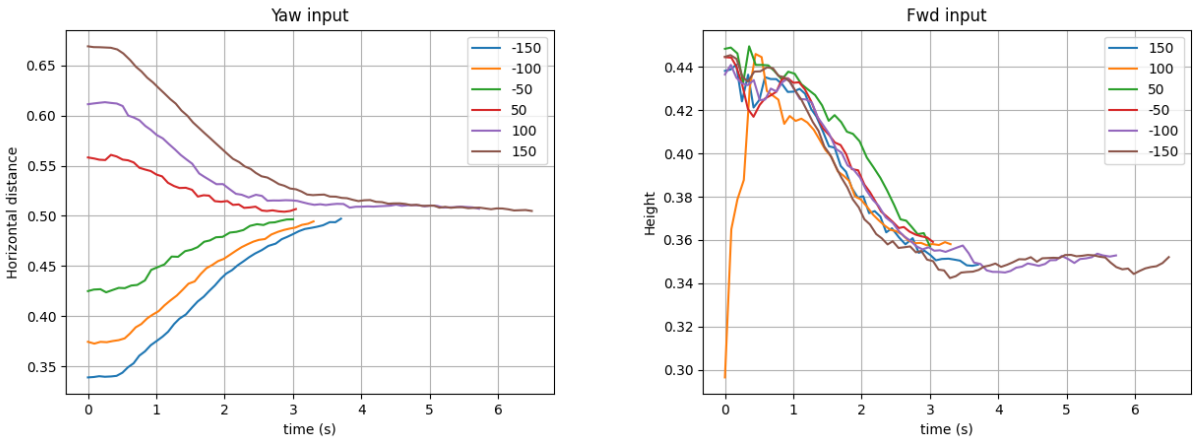


Figure 4.19: Changes over time in detected horizontal position and height as input for the controllers with different starting positions in the y-axis

During execution, the yaw controller introduces a negative yaw velocity when the figure is on the left half of the camera's field of view and a positive yaw velocity when the figure is on the right half, aiming to achieve a target horizontal distance of 0.5 (centred in the camera

[5] https://l-gonz.github.io/tfg-giaa-dronecontrol/videos/test-yaw-controller

image). On the other hand, the forward controller outputs a negative forward velocity to reduce the detected height of the figure from around 0.44 to the target value of 0.36.

Looking at Figure 4.19a, it can be observed that most of the time is spent initiating the movement towards the target. Once the vehicle starts moving, there is not much difference between the -50 and the -150 steps in the time it takes to reach the target position, with the former taking around 3 seconds and the latter taking approximately 3.6 seconds.

In Figure 4.19b, the trajectories appear quite similar since the starting distance to the target is the same for all the cases. For each run, the controller guides the vehicle to move backwards, ensuring that the figure stays sufficiently far away, resulting in a decrease in the detected height. Additionally, a detection anomaly can be observed in Figure 4.19b. For the starting position at $y = 100$ (yellow line), there is a brief initial period where the detected height is very small due to a detection error in the first frames processed by the computer vision algorithm. However, within half a second, the detection stabilizes, and the controller successfully guides the vehicle to the target position without significantly impacting the time taken or the final position. This demonstrates that the controllers are capable of recovering from detection errors without compromising the vehicle's movement.

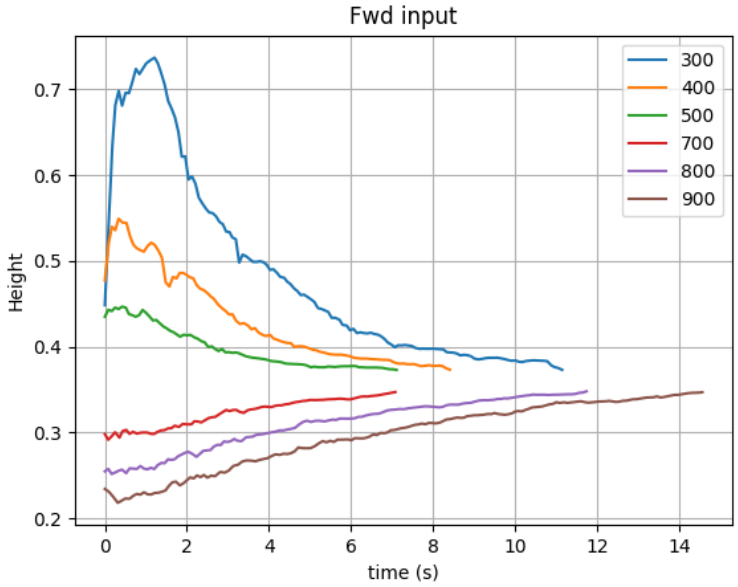


Figure 4.20: Changes over time in detected height as input for the forward controller with different starting positions in the x-axis

To further validate the performance of the forward controller, the `test-controller` tool can be used with varying positions along the x-axis. This means that the tests are conducted with the human figure at different distances along the longitudinal axis of the vehicle, closer and further away than the reference distance. Throughout the process, the figure is kept centred in the camera's field of view (y position remains 0). Therefore, in this

scenario, the yaw controller does not need to be considered.

Figure 4.20 illustrates the changes over time in the input to the forward controller for each starting position as the vehicle moves towards the target position. The graph highlights significant differences in how the controller responds to positions closer or further away from the target distance. When the person is very close to the vehicle, there are substantial differences in detected heights for minor changes in longitudinal distance, leading to the rapid movement of the vehicle away from its start position. Conversely, when the person is further away from the vehicle than the target, the same differences in distance are associated with minor differences in detected height. As a result, the controller determines a smaller velocity output compared to the cases when the person is closer to the vehicle. Consequently, it takes a longer time for the vehicle to reach the target position.

Notably, even when the person is so close to the vehicle that part of their figure falls outside the camera's field of view, the pose detection mechanism functions well enough to estimate the person's continued position outside the image. This can be seen on Figure 4.20 for the case of $x = 300$, where the detected height starts lower than it should be and gradually increases as the full person starts to fit in the camera's field of view.

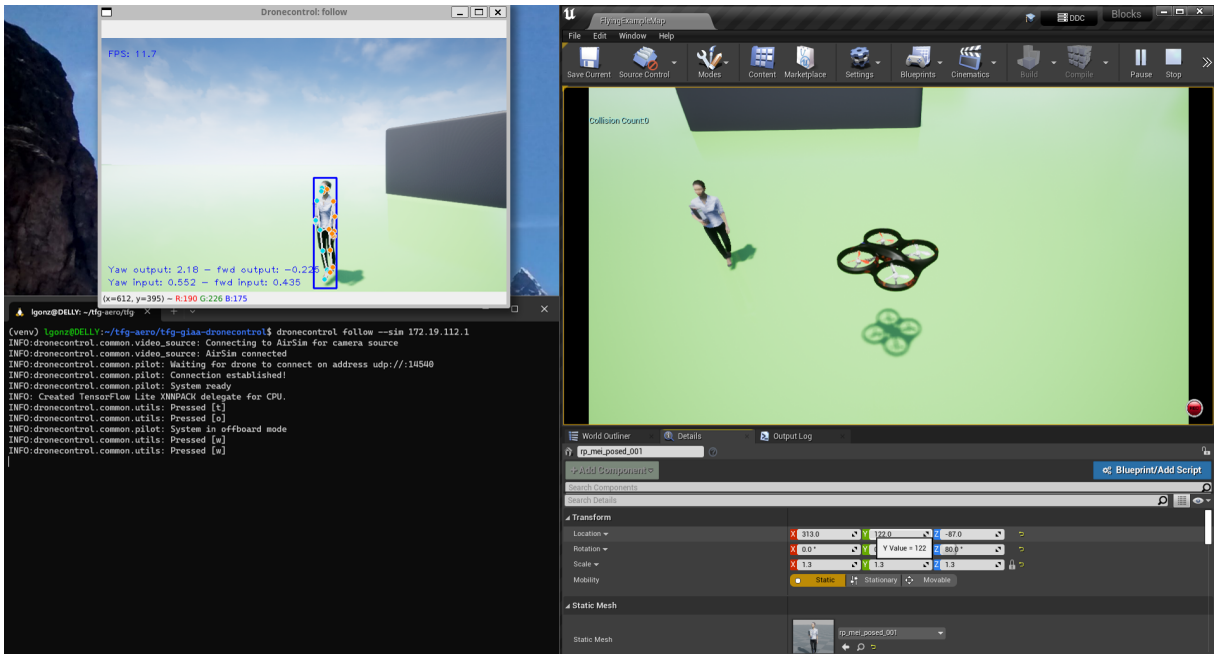


Figure 4.21: Single frame from the video showing the movement of the drone in response to changes in the position of the tracked person

Overall, the results of the validation process indicate that the tuned controllers perform effectively, and accurately follow the person across different starting positions, even when detection errors occur. They successfully respond to variations in the person's distance and

angle from the vehicle, allowing for accurate tracking and movement towards the target relative position.

The selected coefficients for the controllers can now be applied to the complete follow solution in order to assess the expected performance of the vehicle in real flights. To provide a visual demonstration, a video showcasing the drone's movement using these parameter values can be accessed here[6]. Additionally, Figure 4.21 displays a frame extracted from the video, giving a glimpse of the drone's behaviour during the follow operation.

4.3 PX4 HITL simulation and validation

This section will delve into the practical implementation of the hardware-in-the-loop (HITL) mode using QGroundControl, Pixhawk 4 board, and the AirSim simulator. The objective is to transition from using a simulated version of the flight stack running on Linux (SITL) to executing the PX4 software natively on a physical Pixhawk board with simulated input and output. This simulation mode allows for real-time testing and validation of the system by integrating physical hardware with simulated environments.

Achieving a seamless interaction between the simulator, board, and external control applications requires several configuration steps and setting up wired connections. This configuration will be explored with the goal of achieving a system that can run the developed control solution, replicating the same behaviour demonstrated in the previous section. In the first instance, the DroneVisionControl application will be run from the simulation computer, as previously demonstrated. However, after the performance of the flight board is validated, it will transition to running on a separate companion computer, the Raspberry Pi 4.

The steps for setting up the HITL simulation environment include configuring the flight board through QGroundControl, configuring AirSim to connect to a physical board, and adding new communication channels for additional control mechanisms (DroneVisionControl and RC). QGroundControl provides a specific quadcopter HITL airframe configuration, which initializes the board with all the necessary parameters to activate the simulation mode. The details of these parameters can be found in Appendix A.2. Configuring the board using QGroundControl is a straightforward process. Simply connecting the Pixhawk 4 board to the computer through its debug Micro-USB port will make QGroundControl automatically detect and establish a connection with the board.

On the AirSim side, enabling the simulator to work in HITL mode requires making modifications to its configuration file. Specifically, the option to accept serial connections needs to be enabled. This file is described in detail in Appendix A.3. It is important to note

[6] https://l-gonz.github.io/tfg-giaa-dronecontrol/videos/test-sitl-follow

that both QGroundControl and AirSim cannot simultaneously establish a connection to the Pixhawk board through the same USB port. Only one of them can be active and connected to the board at any given time. Consequently, QGroundControl must be shut off while conducting the simulation to allow AirSim to establish the necessary communication with the board.

To test the complete system configuration for HITL, as described in Section 3.1 and outlined in Figure 3.5, the Pixhawk board requires an additional communication channel dedicated to the MAVLink exchange with the DroneVisionControl application. This channel is achieved by adding a telemetry radio to the board, which will connect to a counterpart radio on the simulation computer. This wireless link, along with the already existing USB connection, will provide the two separate MAVLink channels needed to complete the environment. As the AirSim simulator requires a higher update rate compared to the DroneVisionControl application, it is important to keep the Pixhawk to AirSim connection on the wired link. The DroneVisionControl application, on the other hand, sends and receives commands from the Pixhawk at a lower rate, as it depends on the results of the slower computer vision algorithms. The data transmission rate of the radio link is, therefore, sufficient for this application.

Since the flight stack now runs on a physical controller, it becomes possible to attach an RC antenna to the `PPM RC` port of the board. This antenna allows the vehicle to be flown using an independent RC controller[7]. By configuring the switches in the RC unit for flight mode change or as a kill switch, additional tests can be carried out to verify the developed safety features described in Section 3.5.2, like interrupting autonomous flight upon flight mode changes or upon loss of signal from the RC controller.

After all the necessary connections are set up, and the AirSim simulator started, the control program can be started with the following command:

```
dronevisioncontrol follow --sim --serial COM[X]:57600
```

The `COM[X]:57600` section describes the serial connection to the telemetry radio, where the COM port number will vary depending on the particular USB port to which the telemetry radio is connected, and the baudrate specified is 57600.

[7] https://docs.px4.io/main/en/config/radio.html

4.3.1 PX4 HITL validation with Raspberry Pi

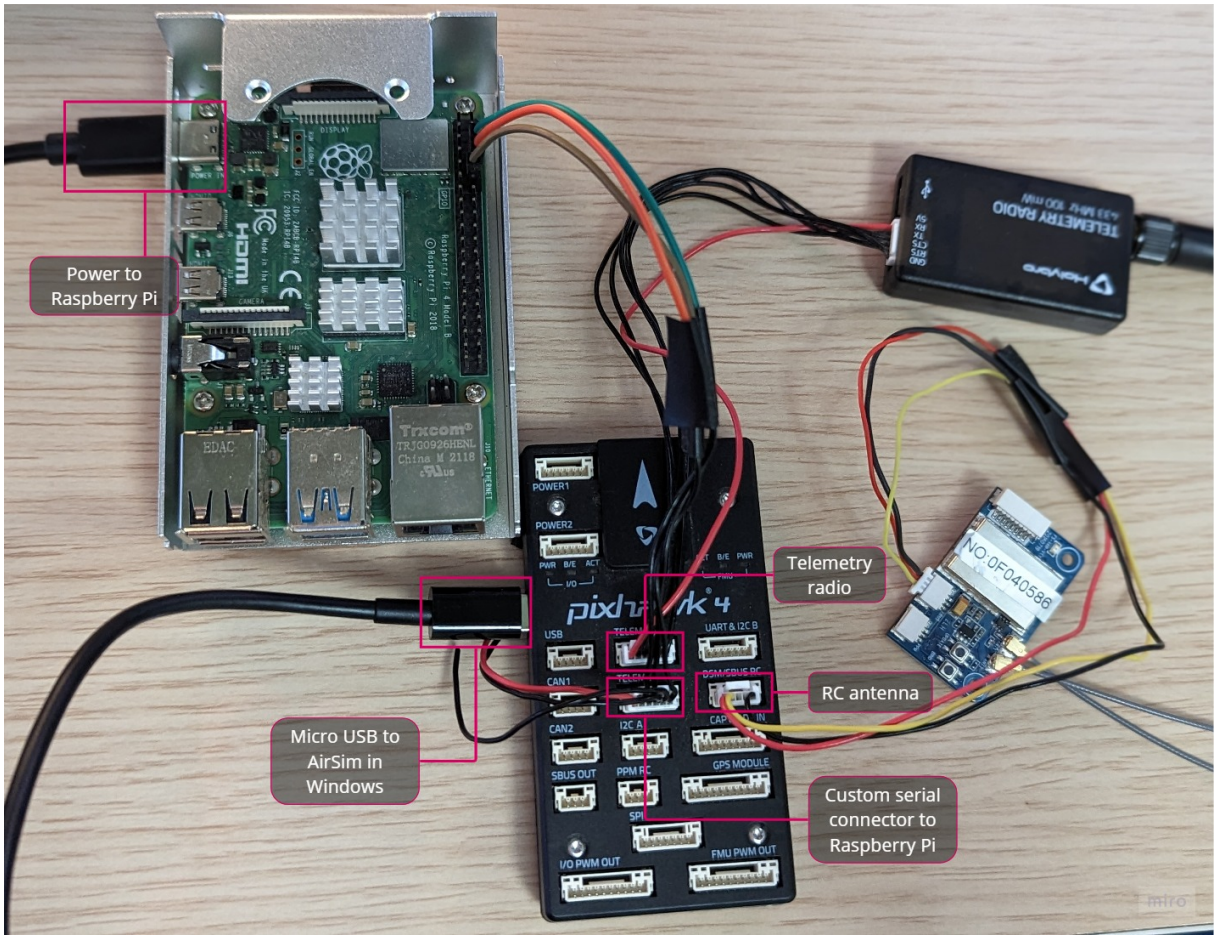


Figure 4.22: Pixhawk 4 board connected to a Raspberry Pi running the DroneVisionControl application and a Windows computer running the AirSim simulator. The setup includes a telemetry radio for QGroundControl and an RC receiver for manual control.

The next crucial step in transitioning from a fully simulated environment to real flight is to establish a connection between the future onboard computer, the Raspberry Pi 4, and the Pixhawk flight controller. This connection enables conducting more realistic tests using the exact hardware that will control the drone.

To ensure successful progress towards autonomous flight, several key characteristics of the Raspberry Pi must be addressed:

1. Capacity to function when powered by a battery.
2. Stability of the serial connection to the Pixhawk board.
3. Ability to run the DroneVisionControl application and its dependencies.

4. Connection to an external camera

5. Performance of computer vision algorithms with limited processing power.

Figure 4.22 provides an overview of the connections used for HITL testing with the Raspberry Pi. The main addition from the previous section is the direct connection between the Pixhawk board and the Raspberry Pi's I/O pins. While the inclusion of the telemetry radio is not strictly required in this scenario, it enables maintaining a simultaneous connection to QGroundControl on the ground station or simulation computer, facilitating better oversight and monitoring.

Before any tests can begin, it is necessary to set up the operating system of the Raspberry. A detailed explanation of the complete installation process, along with all the necessary libraries and dependencies for the Raspberry Pi, is included in Appendix **??**. To conveniently control the Raspberry Pi during the installation, a remote desktop connection is recommended. This allows for transmitting screen contents, as well as mouse and keyboard input, over a local network. Thus, accessing the Pi's desktop from the ground station computer becomes feasible, even during flight. One available option to achieve this is XRDP[8], an open-source implementation of a Microsoft Remote Desktop Protocol server that is compatible with the Raspberry OS.

The initial hardware connection to test is the power supply to the Raspberry Pi. As discussed in Section 3.2.3, the chosen method for powering the companion computer in the onboard configuration is through a secondary battery. To establish this connection, a USB to USB-C cable will be used, connecting the battery to the Raspberry Pi's power port. The purpose of testing the power supply at this stage is to ensure its suitability before it becomes critical to power the onboard computer on the vehicle during flight. This power source must provide sufficient power to maintain the processor running at an appropriate speed and also supply power to the external camera, which will be connected to the Pi's board and draw power from it.

Another important connection is the MAVLink communication channel between the flight controller and the onboard computer. The custom connector described in Section 3.2.3 is used for this purpose, with one end attached to the `TELEM2` port on the Pixhawk board and the TX/RX pins on its other end connected to the UART pins on the Raspberry Pi's GPIO header. For this serial connection to function, additional configuration is required for both the flight controller and the companion computer. In QGroundControl, the Pixhawk must be configured to enable a secondary MAVLink channel. By default, only the `TELEM1` port, which is used by the telemetry radio, is configured. The necessary parameters to modify and their corresponding values are listed in table 4.1.

On the Raspberry Pi side, the serial port is initially configured for use as a terminal

[8] http://xrdp.org/

Parameter name	Value
MAV_1_CONFIG	TELEM2
SER_TEL2_BAUD	921600

Table 4.1: PX4 parameters that require configuration to enable MAVLink communication through the secondary telemetry port.

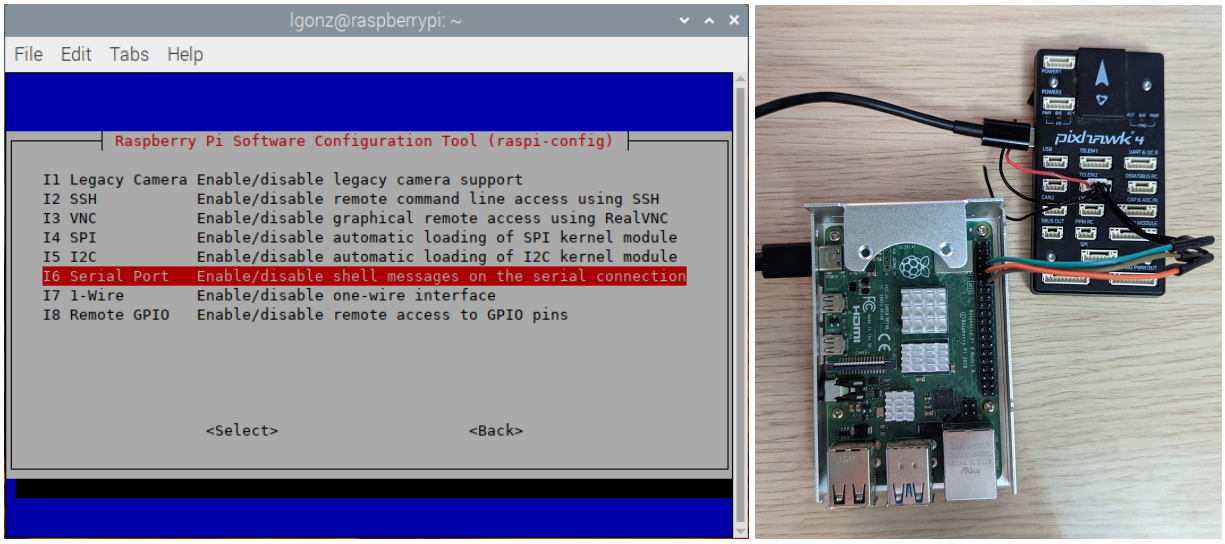


Figure 4.23: a) Picture of Raspberry's `raspi-config` and b) close-up of Pixhawk to Pi cable connection

rather than a hardware serial port. This can be rectified using the `raspi-config` command-line utility by selecting the following steps: Interface options -> Serial Port -> Disable login shell, enable serial port hardware. After making these changes, the `/dev/serial0` address can be used to communicate with the device at the baud rate configured in QGroundControl. Figure 4.23 displays the `raspi-config` tool and the connector used for the serial connection between the flight controller and the companion computer.

The remaining connections with the ground station's AirSim and QGroundControl are set up through the development-only micro-USB port on the Pixhawk board and the telemetry radio, respectively. To validate the complete configuration, the test camera utility will be used by executing the following command in the Raspberry Pi:

```
dronevisioncontrol tools test-camera --hardware /dev/serial0:921600 --sim <AirSim host
↪  IP> --pose-detection
```

The result from the execution are shown in Figure 4.24. On the right side, the remote connection to the Raspberry Pi's desktop is displayed, showing the output of the DroneVisionControl program running the pose detection algorithm on the images received from the simulator. On the left side of the figure, the AirSim simulator renders the

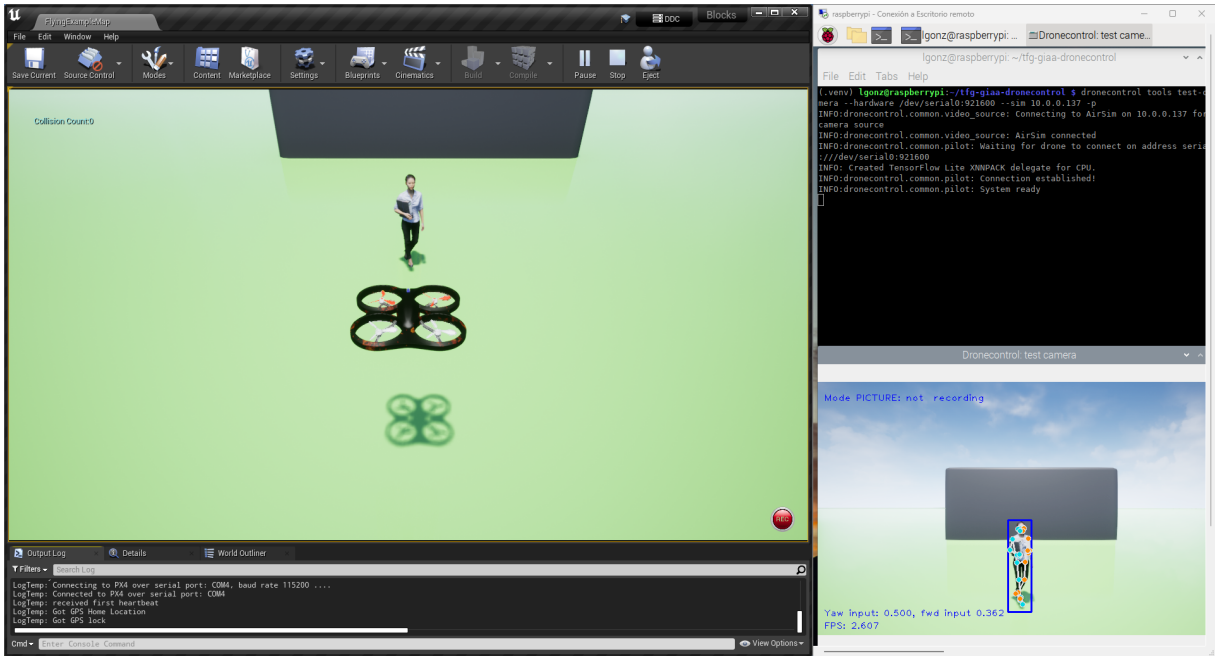


Figure 4.24: Left: AirSim simulator on Windows host. Right: RPi desktop with DroneVisionControl application and pose output.

movements of the vehicle as it responds to the input from the flight controller and the companion computer.

4.3.2 Performance analysis

One crucial question that remains to be answered before the vehicle can take flight using this hardware and software configuration is whether the Raspberry Pi 4b's modest processor, a quad-core ARM Cortex-A72 64-bit SoC running at 1.5GHz, can handle the detection and tracking algorithms with sufficient performance to achieve similar results to those obtained with simulated hardware (SITL) and respond effectively to real-time movement. To address this matter, the average time spent by the program on each task in the running loop can be calculated and analyzed under different scenarios. This analysis will help estimate the maximum speed at which the algorithm can track a person.

Based on the running loop for the follow solution described in Section 3.5 and shown in Figure 3.22, the processing cost can be divided into several distinct areas that can be measured independently: image processing, offboard control, manual input, and released thread (sleep). Figure 4.25 illustrates the time allocation for each task during an average run of the follow solution using simulated hardware (PX4 running in SITL mode with AirSim). The time measurements were obtained by calculating the time difference between the start and end of each statement and averaging across every iteration of the main loop.

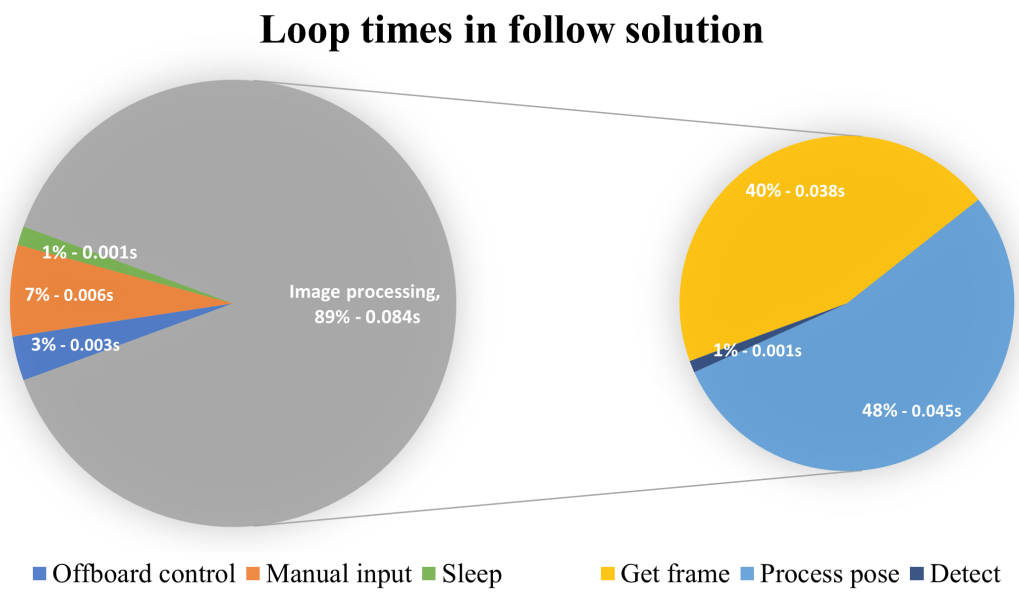


Figure 4.25: Average percentages of a loop spent by each task in the follow solution with their corresponding average absolute times in seconds.

The most time-consuming task is image processing, which accounts for approximately 89% of the total loop time. This task is where the most significant differences in performance can be expected between the simulated hardware and the solution running on the Pixhawk 4 + Raspberry Pi combination. To gain further insight into the time allocation within the image processing task, the work can be further divided into three subtasks:

1. **Get frame**, which involves requesting a new frame from the video source.
2. **Process pose**, which entails sending the frame to the MediaPipe library for detection and tracking.
3. **Detect**, which involves calculating the bounding box coordinates and determining whether it represents a valid pose.

By analyzing the performance of these subtasks, we can gain a better understanding of the individual components contributing to the overall image processing time.

The right graph of Figure 4.25 depicts the additional division of the image processing task, with the subtasks taking 40%, 48%, and 1% of the total loop time, respectively. The average time for each iteration of the loop is 0.095 seconds, resulting in an average performance of 10.5 frames per second (FPS).

Similar measurements were conducted for different hardware combinations, employing varying degrees of simulation while running the follow solution in offboard mode and

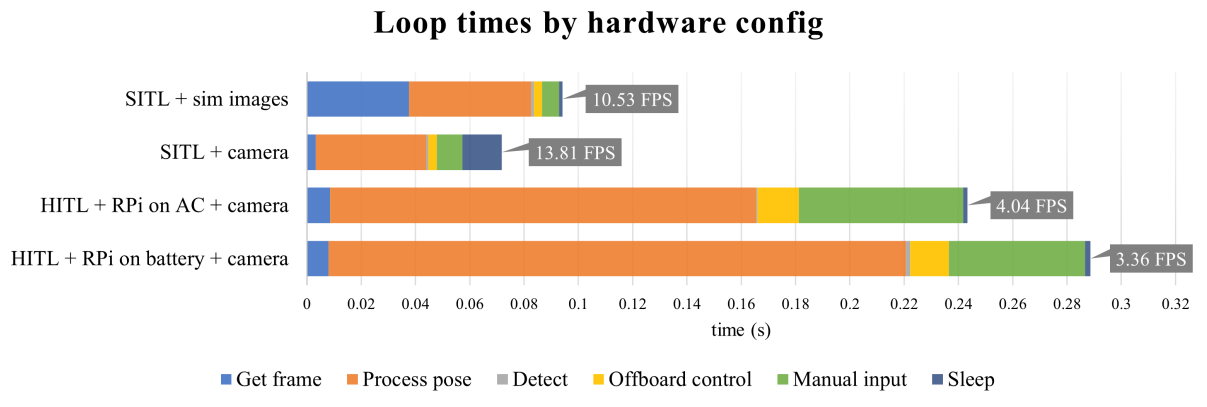


Figure 4.26: Average FPS and time spent on each task per iteration of the follow solution for the different hardware configurations.

connected to the AirSim simulator. The hardware combinations tested include:

1. All simulated hardware: PX4 in SITL mode + DroneVisionControl on the simulation computer + images from the AirSim simulator as the video source.
2. Simulated hardware with real images: PX4 in SITL mode + DroneVisionControl on the simulation computer + images from an attached camera as the video source.
3. Test hardware with AC power supply: PX4 in HITL mode on Pixhawk 4 + DroneVisionControl on Raspberry Pi + images from an attached camera as the video source.
4. Test hardware with battery: PX4 in HITL mode on Pixhawk 4 + DroneVisionControl on Raspberry Pi powered by a battery + images from an attached camera as the video source.

Figure 4.26 presents the average measurements obtained for all the analyzed hardware combinations. In the first test, it is observed that retrieving each new frame from the AirSim simulator takes significantly more time compared to using an external camera. This discrepancy arises from performance limitations in the simulation running on Unreal Engine, which will not be a factor once the simulation engine is no longer used. In the second test, replacing the simulator images with the feed from an external camera results in faster image retrieval and the highest performance among all the tests, averaging 14 FPS with peaks exceeding 20 FPS.

In the third and fourth tests, the image processing calculations are performed on the onboard Raspberry Pi computer, leading to a 400% to 500% increase in the time required for pose processing. Additionally, there is also a noticeable difference in performance observed for the Raspberry's processor between powering the computer through the AC

power supply and through an external battery. This difference stems from the fact that the former supplies 3A of current to the board and the latter only 2A.

These measurements provide insights into the board's expected behaviour during actual flights, with the fourth test (hardware with battery) representing the closest approximation to real flight conditions. From the results, it is expected that a performance of approximately 3 FPS can be sustained during flight, which gives a time between frames of approximately 0.3 seconds. Considering that the camera's field of view for flight tests covers approximately four meters at the target follow distance, the person being tracked should be able to move at a speed of 3-4 m/s with the drone maintaining line of sight.

4.4 Quadcopter flight tests

After validating the performance and safety of the control algorithms in the simulated environment, the next step involves conducting flight tests with a fully-built physical UAV. This final phase of the validation process aims to assess the performance of the developed software with all the previously analysed components together in a real quadcopter during flight. To achieve this, first, the base vehicle will be constructed using the chosen development kit. Subsequently, all additional components required for this project, such as the companion computer and camera, will be integrated into the base frame. Once the vehicle can successfully fly with the full payload under remote control, the developed control solutions will be tested.

The initial test will run the hand-control solution to verify that the autopilot can receive flight commands from an offboard computer outside of the simulation. Next, the follow mechanism will be started to confirm that the companion computer can function in flight as well as it did during the simulation tests.

The exact steps that will be executed one after the other to ensure that safety is maintained during the whole process are as follows:

1. Assemble the quadcopter with its basic components.
2. Attach the custom payload.
3. Conduct a test flight using only the remote control and factory autopilot while monitoring through QGroundControl.
4. Perform a flight using the custom software from an offboard computer, utilizing the `test-camera` tool.
5. Conduct a flight using the `test-camera` tool from the onboard computer.

6. Perform a flight using the custom hand-gesture control solution from the offboard computer.

7. Conduct a flight using the custom follow solution from the onboard computer.

4.4.1 Build process

The chosen vehicle for this project is the Holybro X500, specifically designed to be compatible with PX4. The PX4 documentation[9] provides detailed instructions on how to build the vehicle using its Development Kit. Figure 4.27 illustrates all the components required to construct the complete vehicle.

Figure 4.27: Development kit for the Holybro X500.

Source: Adapted from *PX4 User Guide* [16].

Once the standard parts are assembled, the custom additions can be integrated into the remaining space within the frame. The Raspberry Pi companion computer will be positioned between the autopilot and GPS antenna. This placement facilitates a convenient connection between the autopilot and the Raspberry Pi's I/O pins using short cables, preventing excessive wire clutter within the frame.

[9] https://docs.px4.io/main/en/frames_multicopter/holybro_x500_pixhawk4.html

During flight, the Raspberry Pi will be powered by a dedicated external battery, which supplies power through a 2-ampere USB port. This port will be connected to the Raspberry Pi's original power cable, utilizing its USB-C power supply socket. As explained in Section 4.3.1, this power supply is sufficient to operate the connected camera and run the developed software with satisfactory performance. The battery will be positioned beneath the autopilot, as depicted in Figure 4.28.

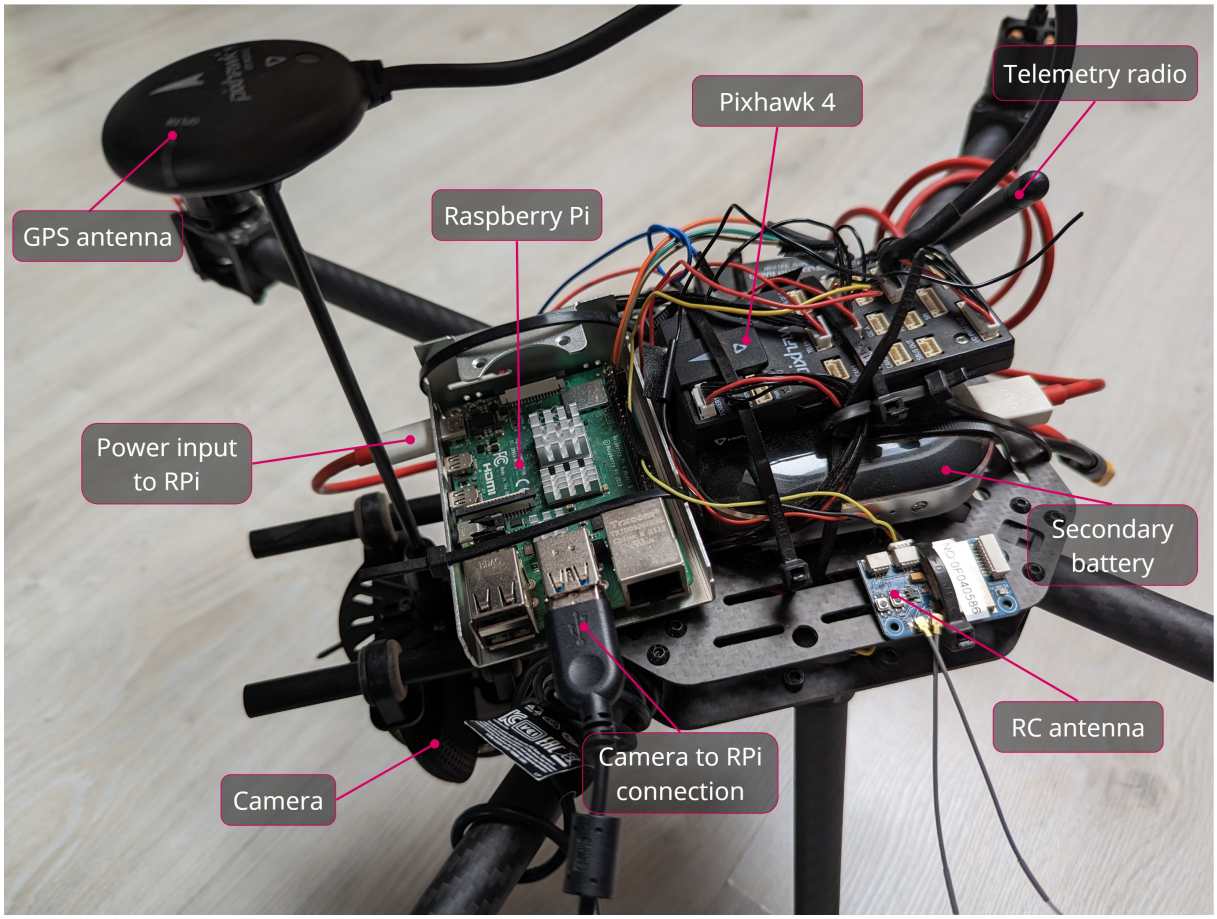


Figure 4.28: Complete build of the quadcopter with the main components highlighted.

To ensure the camera is securely mounted on the vehicle's frame, the custom-built support system described in Section 3.2.3 will be utilized. The camera holder will be attached to the slide bars beneath the main frame, positioning the camera's weight as close to the centre of mass as possible behind the GPS platform. The main battery, responsible for powering the engines and autopilot, and located on the underside of the carbon frame, can be shifted along the forward axis to balance the added weight from the companion computer, its battery and the camera so that the centre of gravity falls approximately in the centre of the vehicle. Figure 4.29 shows the vehicle's underside with the attached camera and the strap for holding the main battery.

Figure 4.29: Underside of the vehicle, with the supports for holding the main battery and the camera in place.

Once the vehicle construction is complete, additional installation and calibration steps are necessary before it can be flown. These steps, outlined in the build instructions, should include disabling any previously activated simulation modes in the vehicle configuration. Furthermore, the `MAV_1_CONFIG` parameter must be set to `TELEM2`, as explained in Section 3.2.2. Calibration of all the onboard and externally attached sensors specific to this build is also required. The QGroundControl ground station application provides a configuration screen with the necessary calibration tools for vehicle setup, as depicted in Figure 4.30. The vehicle configuration can be performed either by directly connecting the flight controller to a computer via the micro-USB port or wirelessly by connecting the companion telemetry radio to the computer running QGroundControl.

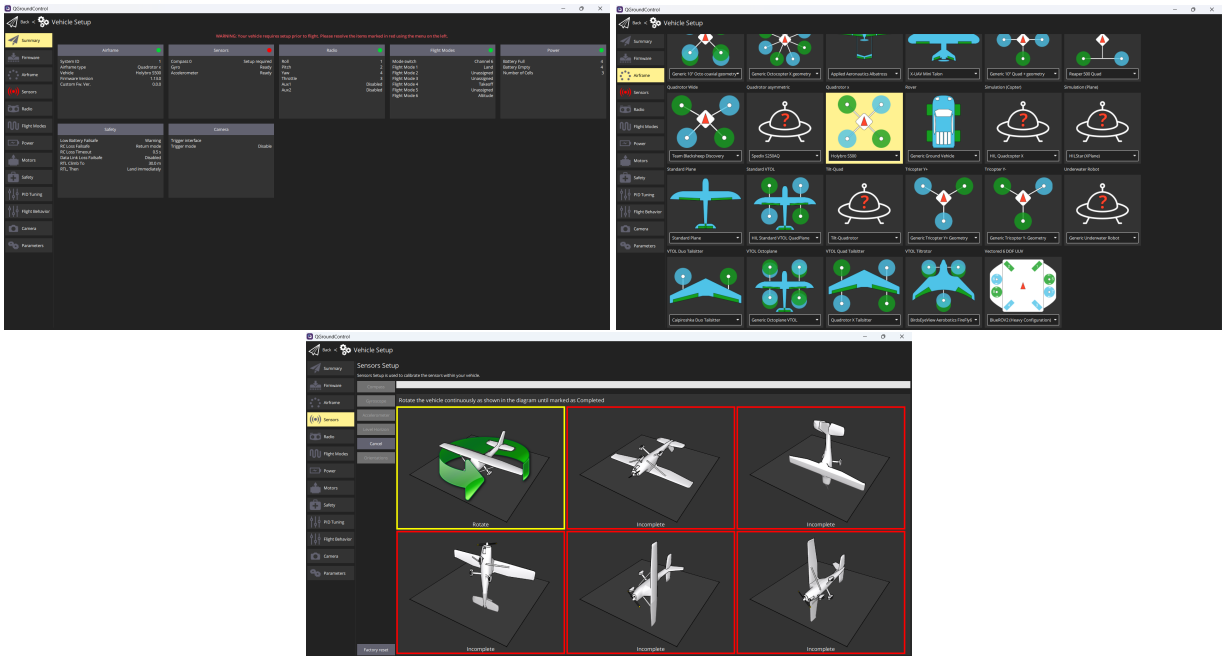


Figure 4.30: Screenshot from the QGroundControl calibration and setup tools used to configure the vehicle

4.4.2 Initial tests

Baseline flight with factory software

Once the vehicle is fully configured, testing the assisted takeoff and landing can be conducted using the RC controller and QGroundControl. At this stage, the drone should be capable of maintaining stable flight using autopilot-assisted flight modes, such as Position Mode. In Position Mode, the roll and pitch sticks in the controller control the vehicle's acceleration over the ground in the forward/backward and left/right directions, respectively, relative to the vehicle's heading. The throttle stick controls the ascent and descent speed. When the sticks are centred, the vehicle actively maintains its position in 3D space, compensating for wind and external forces. This semi-manual mode can serve as a safe means to verify the functionality of the factory autopilot.

Through QGroundControl, it is possible to map different switches on the RC controller to various autopilot commands. For the test, a two-position switch will be mapped to the armed/disarmed state in order to control the quadcopter's engine startup. A three-position switch will be assigned to change the vehicle's flight mode between the landing, takeoff and position modes. By shifting between the available switch positions during flight, the primary autopilot modes can be tested. Any other flight modes or triggers that need to be used can be set directly through the QGroundControl interface during flight.

To initiate the flight, the main battery is connected to the power module socket. This action powers up the autopilot, GPS antenna, telemetry radio, and RC receiver. Subsequently, QGroundControl is launched on a computer connected to the second telemetry radio via USB. If all connections have been established correctly, the ground station application will automatically connect to the vehicle and display its position on a satellite map. Similarly, turning on the RC controller establishes its connection with the vehicle, provided they have been correctly paired together as outlined in the build instructions guide. Once all wireless connections are established, the drone can take off by switching to the armed state, followed by selecting the takeoff flight mode. While the drone is airborne, switching to the position flight mode enables direct control through the joysticks on the controller.

Offboard computer flight with test tool

The second test flight aims to verify that the custom software (DroneVisionControl) can wirelessly transmit takeoff and landing commands from the offboard computer using a MAVlink channel. The channel will be established by taking advantage of the developed `test-camera` tool through the telemetry radio. During this flight, it is important to note that the QGroundControl application cannot be connected to the vehicle as it will interfere with the telemetry radio channel used by the DroneVisionControl application. Consequently, the RC controller will serve as a backup in case of any issues with the software. The controller can be used to switch flight modes and override inputs from the DroneVisionControl application, providing manual control if necessary.

Since the DroneVisionControl application will arm the vehicle on its own by sending a takeoff command and to ensure safety during flight, the two-way switch on the controller will be mapped on all subsequent tests to the command to cut off power to the engines. This command can be valuable in exceptional situations where it is necessary to protect the vehicle or the surrounding area, such as if the autopilot were to destabilize during takeoff and landing or there was a complete loss of control during flight.

To initiate the test, the main battery is reconnected to the power module. Then, the following command is executed on the offboard computer, depending on whether it is desired to be run on a Windows or Linux machine.

```
Windows: dronevisioncontrol tools test-camera -r COM<X>:57600
Linux: dronevisioncontrol tools test-camera -r /dev/ttyUSB0:57600}
```

After successfully establishing the connection with the vehicle, the computer keyboard can be used to control the flight. Pressing the T key triggers takeoff, while the L key initiates the landing process. The O key sets the autopilot to offboard flight mode, enabling it to

Figure 4.31: Terminal output from the `test-camera` tool running on an offboard computer and image of the drone flying in response

receive velocity commands. Subsequently, the WASD keys will control the forward and sideways movement of the vehicle, and the QE keys will adjust its yaw rotation.

Figure 4.31 illustrates the output displayed in the computer's terminal window, showcasing the connection process, the sent velocity commands, and the camera output from the offboard computer. Additionally, a video capturing the entire process can be accessed in the project's hrefhttps://l-gonz.github.io/tfg-giaa-dronecontrol/videos/flight-test-offboardwebsite[10].

[10] https://l-gonz.github.io/tfg-giaa-dronecontrol/videos/flight-test-offboard

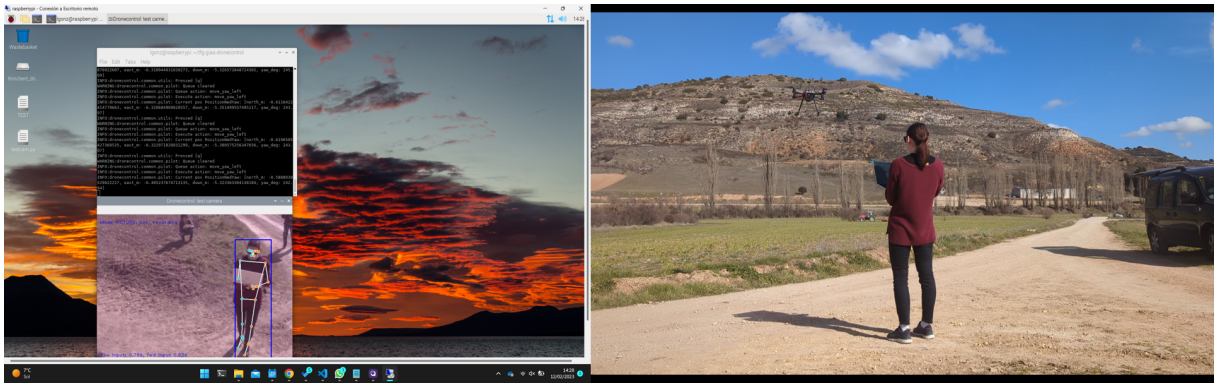


Figure 4.32: Pose detection algorithm running on images taken during flight

Onboard computer flight with test tool

The third and final test flight in this section aims to confirm that the custom software can send takeoff and landing commands through a cabled MAVlink channel from the onboard computer. Additionally, it ensures that the onboard camera can capture a clear image of the vehicle's field of view during flight.

For this test, the same tool used in the previous test will be executed on the Raspberry Pi, utilizing the wired serial link between the onboard computer and the Pixhawk autopilot board. As the tool will now use the camera connected to the companion computer, positioned to look down on the pilot, pose detection can be activated on the received images.

To initiate the flight test, the main battery needs to be attached to the power module and the secondary battery to the Raspberry Pi. Once the onboard computer has started, it will be operated through a remote desktop connection via WiFi. Using this connection, a terminal window can be opened on the desktop to execute the following command:

```
dronevisioncontrol tools test-camera -r /dev/serial0:921600 -p
```

Unlike the telemetry radio flight, this test employs a serial connection running at a baud rate of 921600, matching the configured baud rate on the `TELEM2` port of the Pixhawk board. The `-p` option enables pose detection in the output images. A video documenting the entire process can be accessed on this link[11]. Additionally, an image extracted from the video can be seen in Figure 4.32.

[11] https://l-gonz.github.io/tfg-giaa-dronecontrol/videos/flight-test-onboard

4.4.3 Hand gesture control

During the basic flight tests, all the connections and individual components of the software were validated in realistic flight scenarios. Now, the focus shifts to integrating the piloting system with the results of image recognition to test the developed vision-based control solutions.

The first solution to be tested in flight is the hand-gesture guidance system, which runs on an offboard computer without performance constraints and doesn't rely on battery power. The setup for this test will be the same as the second test flight, described in Section 4.4.2, using the telemetry radio for the serial link and disregarding the onboard companion computer. Once the autopilot board is powered up, the control solution is initiated by executing the following command, where <device> is replaced with the appropriate COM port or TTY device to which the telemetry radio is connected, depending on the platform.

```
dronevisioncontrol hand -s <device>:57600
```

Once the pilot connection is established, the image from the computer's webcam will be displayed on the screen with an outline highlighting any detected hand. To begin controlling the vehicle, the open palm gesture should be shown to the camera. Closing the hand into a fist will initiate takeoff, and pointing up with the index finger will activate the offboard flight mode. Moving the index finger right or left will cause the vehicle to mirror the movement, while moving the thumb right or left will make the vehicle move forwards and backwards, respectively. At any point during the test, displaying an open hand will cause the drone to hover in its current position. Losing sight of the controlling hand will also trigger the hovering mode.

A video showcasing the entire process can be accessed on this link[12]. Additionally, an image extracted from the video can be seen in Figure 4.33.

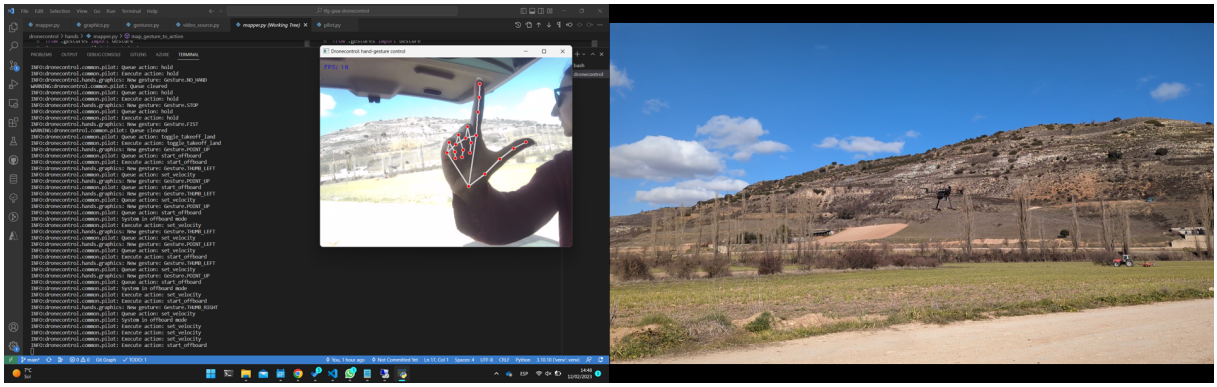


Figure 4.33: Image taken during flight controlled by the hand-gesture solution. The vehicle is moving forward.

[12] https://l-gonz.github.io/tfg-giaa-dronecontrol/videos/flight-test-hand

4.4.4 Target detecting, tracking and following

The last test to conduct is for the follow control solution. In this section, the companion computer will run the follow program with the aim of tracking and following a moving target outside of simulation. The setup for this test will be the same as the third test flight, described in Section 4.4.2. Without the need for a wireless telemetry connection on the DroneVisionControl side, the telemetry radio can be utilized to track the vehicle's path using the QGroundControl application on a secondary offboard computer. The control application will be started with the following command:

```
dronevisioncontrol follow -s /dev/serial0:921600
```

This video[13] showcases the process of the vehicle taking off (T key) and activating offboard flight mode (O key) to initiate the tracking of a detected figure. Figure 4.34 presents an image extracted from this video.

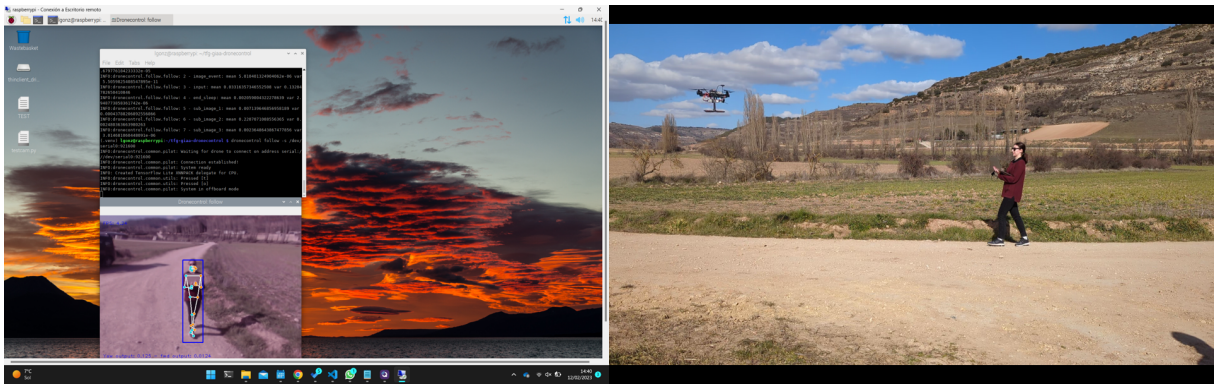


Figure 4.34: Terminal and image output of the DroneVisionControl follow solution running on the Raspberry Pi

During the flight test, the application running on the Raspberry Pi achieves a maximum frame rate of around 6 FPS when the follow mechanism is active and approximately 8 FPS when it is disabled by switching out of offboard flight mode. In practical terms, this means that the person being tracked by the drone needs to move relatively slowly to ensure that the camera does not lose sight of them before the control loop can send the command to the vehicle to move to the previously detected position. However, for a proof-of-concept scenario, this performance is acceptable.

At the end of the program execution, the average loop time and average runtime for each task in the control loop are displayed in the terminal. Based on the measurements obtained during the test flight, the average frame rate is calculated to be 3.58 FPS. Figure

[13] https://l-gonz.github.io/tfg-giaa-dronecontrol/videos/flight-test-follow

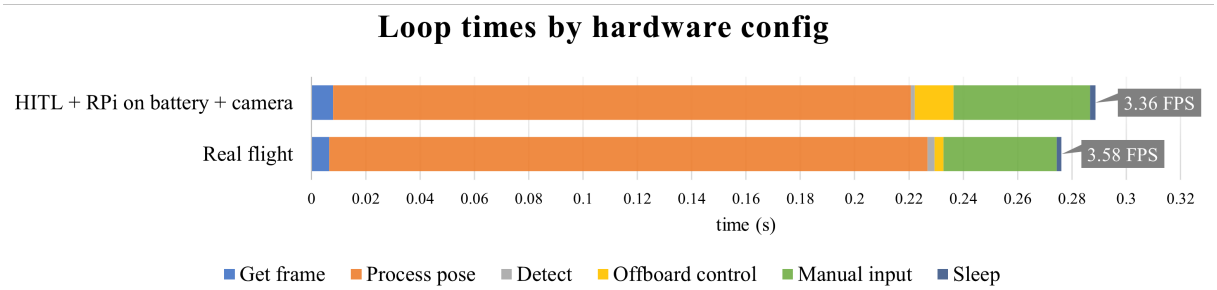


Figure 4.35: Average loop time for each individual task in the follow control solution and total frame rate for realistic simulation versus actual test flights.

4.35 compares these measurements with those analyzed in Section 4.3.2, particularly with the test configuration selected to most closely resemble actual flight conditions (autopilot board running on HITL mode and the companion computer powered by the secondary battery). Very similar results are recorded for both tests, confirming the simulation configuration employed as a valid method of obtaining realistic results without complex or costly flight tests.

Overall, this test flight verifies the effectiveness of the follow control solution and its ability to track and follow a moving target during a real flight scenario, as well as the validity of the simulation process to obtain realistic results.

Chapter 5

Conclusions

This chapter analyses the results obtained from the project, as well as the obstacles that arose during the development and testing phases. Afterwards, the proposed initial objectives are examined for completion, and the lessons learned during the project are exposed, along with some suggestions for future work in the field.

5.1 Evaluation of objectives

The main objective of this project was to illustrate how the PX4 platform could be used to develop vision-driven control mechanisms for UAVs. This was achieved by building and presenting a complete development environment that can be used for each stage in the process of creating new control solutions, from the concept phases to running tests in the target hardware.

For the more specific objectives, in the first place, the tools and systems developed by the Dronecode project, which includes PX4, MavSDK and QGroundControl, were leveraged in cooperation with the DroneVisionControl written code to reduce the workload of the project by making use of preexisting and stable technologies.

Second, to show the minimum requirements needed to develop an individual control solution, a combination of low-cost and accessible hardware was employed to test all the developed software and take the project out of the simulation environment and into a real drone capable of flying. Thanks to the platform's modular nature, all the individual pieces are also interchangeable with other hardware based on budget or availability as long as it can run the same software as the one chosen for the tests in this project.

Third, it has been demonstrated how the development environment, all the presented

existing tools and the chosen hardware can be tied together to create viable vision-driven mechanisms for UAVs, by developing two distinct control solutions for two different computer-camera-autopilot configurations that show the flexibility of the system and some of the possibilities it offers. The viability of these solutions was further validated by carrying out test flights in real-life conditions outside of simulation environments.

Fourth, the testing process presented in chapter 4 introduces a systematic approach to validating the system that can help carry any new software from the earliest phases of testing to the final flight tests while maintaining safety and reliability by progressively introducing new parts on top of the components already validated, for the smallest possible divisions of both the software and the hardware.

Finally, a UAV of the quadcopter type was built to carry out the test flights mentioned before by using a development kit commercialised for easy assembly and already designed to work with the PX4 platform. This made it possible to dedicate the most time to developing the desired control solutions without being concerned with the low-level electronics involved in controlling the engines or the basic flight mechanics like takeoff and landing.

5.2 Lessons learned

5.2.1 Applied knowledge

During the development of this project, it became necessary to apply many of the different pieces of knowledge acquired through the course of the bachelor's degree. The following subjects have been especially relevant to provide the necessary experience to complete the project:

1. **Fundamentals of Programming and Computer Science**. This is the first introduction to computer science and programming in the degree. It provides the foundational knowledge of how to write good code that is the basis of this project's development.
2. **Systems and Circuits**. This course serves as an introduction to electricity and circuits. This information has been useful in understanding how the drone's electronics are powered and how they work together, especially in implementing the custom connectors between the Raspberry Pi and the Pixhawk board.
3. **Architecture of Computer Networks / Telematic Systems**. These two subjects deal with the protocols that make up computer communications. The knowledge they provide on how the UDP, TCP and IP protocols work together, as well as regarding the overall communication between computer networks, has been invaluable in the

process of integrating the three separate computers that make up the simulation environment developed for the project (autopilot board, companion computer and simulation computer) and making them communicate with each other in a single network.

4. **Aerospace Engineering**. The lessons learned in this course offered an introduction to the field of aerospace in general and UAVs in particular. It was especially relevant to acquire the basic knowledge of how drones stay in the air and how their movement in their 3 axes is controlled through the four propellers in a quadcopter, which is the basis for the autopilot in this project.

5. **Operating Systems**. This subject provided much of the necessary knowledge to set up and work with Linux operating systems, which has been needed in this project both for the PX4 SITL simulated autopilot and for the configuration of the Raspberry Pi board as a companion computer onboard the aircraft.

6. **Engineering of Information Systems**. This course offered important tools to handle version control in any software project and, more specifically, to work with the GitHub platform to host the code safely and keep track of issues.

7. **Command and Control Systems**. This subject deals with the analysis and design of control systems, including PID controllers like the ones used to regulate the output velocity of the vision-guided follow system implemented in this project. It provided insight into important concepts like feedback loops, stability and error analysis.

8. **Telematic Services and Applications**. The main takeaway from this subject was the experience it provided in using the Python programming language in complex projects, including package management with `pip` and how to use some of the most common external libraries for Python.

   Other lessons obtained during the course of the bachelor's that cannot be pinpointed to an isolated subject but to the combination of many related pieces in the training itinerary, and that has been invaluable for the development of this project, include knowledge on how to investigate and read technical documentation, how different communication systems work and how to use them to their best advantage, or how to manage the development of complex projects.

5.2.2 Acquired knowledge

Throughout this project's development, many challenges have required expanding on the knowledge mentioned in the previous section and acquiring new competencies to find solutions to the problems that surfaced. These are some of the main aptitudes developed:

- Python skills: how to organise projects with multiple submodules and create packages from own code, as well as experience in several standard libraries:
  - Asynchronous programming in Python with the `asyncio` library.
  - Matrix and vector handling with the `numpy` library.
  - Plotting customisable graphs in different UI platforms with the `matplotlib` library.
- Knowledge of open-source UAV autopilots, including the main options available, the tools they offer, how they work and how they allow developing new functionality for their platform.
- Computer vision: knowledge of the main tasks of classification, localisation, landmark detection, and tracking, types of algorithms available and how they are trained in big datasets and how to select one based on the required accuracy and available performance.
- OpenCV: how to use the multi-platform library to analyse and manipulate images from different sources.
- Raspberry Pi board and Raspberry OS: how to work with serial connections from general I/O pins, how the OS differs from other Linux variants and its limitations, and how to improve the performance of Python code to adapt to the limited processing power.
- Using Unreal Engine for physics simulation, employing pre-made plugins for the engine and customising environments, besides working with 3D models to simulate and test computer vision scenarios.
- Practical application of a PID to a control problem and how to use trial-and-error to calibrate it for an experimental system with an unknown transfer function.
- 3D-modelling in Tinkercad and 3D-printing with PLA plastic to design and build a camera holder adapted to the drone frame.
- Using Miro for drawing different kinds of diagrams and image layouts.

5.3 Future work

The work presented in this bachelor's thesis focused on presenting several basic vision-based control solutions for the PX4 autopilot to show the system's capabilities. Therefore, there are numerous avenues for further research and development in this area that could expand on the work presented here.

- Integration with more sensors: In this work, only a single camera was used as the input source for the computer vision algorithm. In future work, it would be beneficial to investigate the integration of other sensors, such as lidar, radar, or multiple camera systems, to improve the performance of the computer vision algorithm.

- More sophisticated computer vision algorithms: The algorithms used in this thesis employ basic image processing techniques to make up a detector-tracker ML pipeline. However, there is still room for improvement in the sophistication of computer vision algorithms. Future work can investigate the use of deep learning-based algorithms, such as convolutional neural networks, to improve the accuracy and robustness of the system.

- Exploration of alternative control algorithms: A simple proportional-integral-derivative (PID) controller was used in this work to control the drone's position. However, several alternative control algorithms, such as model predictive control or neural network-based controllers, could be explored.

- Multi-drone control: The focus of the project was on controlling a single drone. However, in real-world scenarios, multiple drones may need to be controlled simultaneously. Future work can investigate using multi-drone vision-based control algorithms to enable the coordinated control of multiple drones.

- Application to other domains: While the focus of this work was on drone control, the PX4 autopilot is compatible with other ground and water-based vehicles, so the proposed computer vision solutions could be applied to other domains, such as robotics or autonomous vehicles.

Overall, the proposed vision-based control solutions for PX4 autopilots are a promising approach that can be further improved through future research. The areas mentioned above provide several directions for future work that can lead to more accurate, robust, and adaptive control solutions that react to their environment and that can contribute to providing autonomous flight on multiple application fields.

Acronyms

API Application Programming Interface. 22

FPS frames per second. 80

GPIO General Purpose Input/Output. 35

GUI Graphical User Interface. 42

HITL Hardware-in-the-Loop. 20

IMU Inertial Measurement Unit. 22

OS Operating System. 56

QGC QGroundControl. 25

RC Radio Control. 31

SITL Software-in-the-Loop. 20

TCP Transport Control Protocol. 25

UDP User Datagram Protocol. 25

WSL Windows Subsystem for Linux. 25

Appendix A

Installation manuals

This appendix https://github.com/l-gonz/tfg-giaa-dronecontrol

A.1 SITL: Development environment

This section describes the process of installing all the necessary applications to set up a development environment to run PX4's SITL simulation in a system similar to the one described in section 3.1. The instructions assume a computer running Windows 10/11 as the operating system and Windows Subsystem for Linux installed. PX4 details the installation steps of their source code for several platforms in their documentation [29], where Ubuntu is the recommended platform. To make it easier, the DroneVisionControl repository contains a small shell script that aggregates all the steps and installs all the dependencies with the folder structure that the project expects. This includes installing and setting up PX4 and QGroundControl and creating a virtual environment for the project, installing the Python packages and the DroneVisionControl application.

To run the script, simply clone the repository, navigate to the project folder and execute:

```
./install.sh
```

To test the installation of PX4, execute the following line (requires a graphic interface for WSL):

```
./simulator.sh --gazebo
```

To test the installation of DroneVisionControl, execute:

```
dronevisioncontrol tools test-camera -s -c
```

Camera input is not supported from within WSL, but it should be possible to control the simulated drone with the keyboard.

The next step is to install DroneVisionControl in the Windows machine to be able to access an integrated or USB camera. It requires having Python already installed. First, clone the project repository, navigate to the folder and set up the virtual environment:

```
pip install virtualenv
virtualenv venv
venv\Scripts\activate
```

Then install DroneVisionControl:

```
pip install -r requirements.txt
pip install -e .
```

Additionally, to make the simulated PX4 application broadcast to the Windows machine on port 14550, it is necessary to edit the file `px4-rc.mavlink` located in `Firmware/PX4-Autopilot/etc/init.d-posix` on the project folder in WSL. To the mavlink start command for the ground control link on line 14, append -p to enable broadcasting:

```
mavlink start -x -u $udp_gcs_port_local -r 4000000 -f
```

To test the installation, with the PX4 simulator still running in WSL, execute:

```
dronevisioncontrol tools test-camera -s -c
```

It should now be possible to both control the drone with the keyboard and obtain images from a camera attached to the computer, as well as run the hand control solution in the simulator.

A.1.1 Installation of AirSim

To run the PX4 software-in-the-loop simulator in AirSim and test the follow solution, first, install Unreal Engine at version 4.27 at least from the Epic Games Launcher[1] and open the

[1] https://www.unrealengine.com/download

environment found in the `data` folder of the repository. To use AirSim with a different Unreal environment, follow the guide in the AirSim documentation[2] After starting play mode in Unreal for the first time, a `settings.json` file will appear in an AirSim folder in the user's Documents folder. The contents of this file need to be replaced with the configuration file found in section A.3 to be able to interact with PX4 running inside WSL, selecting the correct value for `UseSerial` and exchanging the `LocalHostIp` in the file for the IP of the Windows machine in the virtual WSL network. This IP can be obtained by typing `ipconfig` in the Windows command prompt and looking for the IPv4 address under "Ethernet Adapter vEthernet (WSL)".

After the settings have been set, restart play mode in Unreal; the output log should show a message saying "Waiting for TCP connection on port 4560, local IP <Windows-IP>". It is now possible to build and start PX4 by executing in WSL:

```
./simulator.sh --airsim
```

If the IP has been set correctly in the AirSim settings, PX4 and the simulator will find each other correctly and connect. The test-camera tool can be run either from Linux within WSL or from Windows by executing:

```
dronevisioncontrol tools test-camera -s -w
```

The follow control solution can be run from Linux with:

```
dronevisioncontrol follow --sim 172.19.112.1
```

and from Windows with:

```
dronevisioncontrol follow --sim -p 14550
```

A.2 HITL: Installation on a Raspberry Pi 4

Write this part

Install DroneVisionControl

[2] https://microsoft.github.io/AirSim/unreal_custenv/

Set up XRDP to connect from Windows PC https://linuxize.com/post/how-to-install-xrdp-on-raspberry-pi/

Set up UART serial connection in RPi: https://discuss.px4.io/t/talking-to-a-px4-fmu-with-a-14119?page=2

A.3 AirSim configuration file

Listing A.1 provides the configuration file used for AirSim. The file includes various settings and parameters tailored to the specific requirements of the project.

The `PX4` section contains the configuration settings for the PX4-type flight controller. It tells the simulator to connect to the specific external flight stack. The `LockStep` parameter is set to true to indicate synchronized step execution between the simulator and the flight controller and the rest of the connection settings replicate the network configuration described in Figure 3.3.

The `Parameters` section includes specific PX4 parameters relevant only for simulation, such as disabling safety checks for RC control and the start world location of the vehicle. These parameters can be customized based on the desired behaviour and performance of the flight controller.

The only settings that need to be edited depending on the individual machine and current simulation mode are the `LocalHostIp` and `UseSerial`. The `LocalHostIp` should be replaced with the IP of the Windows host for the WSL virtual network interface to allow communication with the Linux subsystem. This is only necessary for SITL mode. The `UseSerial` parameter determines the communication channel with the flight controller and is set to either `true` for HITL simulation mode through USB or `false` for SITL simulation mode to use UDP instead.

The `CameraDefaults` section defines the settings for the camera images that can be retrieved through the `airlib` library from external code.

```
{
    "SettingsVersion": 1.2,
    "SimMode": "Multirotor",
    "ClockType": "SteppableClock",
    "Vehicles": {
        "PX4": {
            "VehicleType": "PX4Multirotor",
            "LockStep": true,
            "UseSerial": "<SITL: false, HITL: true>"
            "UseTcp": true,
            "TcpPort": 4560,
            "ControlIp": "remote",
            "ControlPortLocal": 14550,
            "ControlPortRemote": 18570,
            "LocalHostIp": "<Windows-IP>",
            "Sensors":{
                "Barometer":{
                    "SensorType": 1,
                    "Enabled": true
                }
            },
            "Parameters": {
                "NAV_RCL_ACT": 1,
                "NAV_DLL_ACT": 0,
                "COM_RCL_EXCEPT": 7,
                "LPE_LAT": 47.641468,
                "LPE_LON": -122.140165
            }
        }
    },
    "CameraDefaults": {
        "CaptureSettings": [
            {
                "ImageType": 0,
                "Width": 640,
                "Height": 400
            }
        ]
    }
}
```

Listing A.1: AirSim's `settings.json` file, located in the computer's Documents folder, with settings required for configuring this project.

Appendix B

Command-line interface of the application

```
Usage: dronevisioncontrol tools test-camera [OPTIONS]

Options:
  -s, --sim TEXT        attach to a simulator through UDP, optionally provide
                        the IP the simulator listens at
  -r, --hardware TEXT   attach to a hardware drone through serial, optionally
                        provide the address of the device that connects to PX4
  -w, --wsl             expects the program to run on a Linux WSL OS
  -c, --camera          use a physical camera as source
  -h, --hand-detection  use hand detection for image processing
  -p, --pose-detection  use pose detection for image processing
  -f, --file TEXT       file name to use as video source
  --help                Show this message and exit.


Usage: dronevisioncontrol tools tune [OPTIONS]

Options:
  --yaw / --forward     test the controller yaw or forward movement
  --manual              manual tuning
  -t, --time INTEGER    sample time for each of the values to test
  -p, --kp-values TEXT  values to test for Kp parameter
  -i, --ki-values TEXT  values to test for Ki parameter
  -d, --kd-values TEXT  values to test for Kd parameter
  -h, --help            Show this message and exit.


Usage: dronevisioncontrol tools test-controller [OPTIONS]

Options:
  --yaw / --forward  test the controller yaw or forward movement
  -f, --file TEXT    file name to use as data source
  -h, --help         Show this message and exit.
```

```
Usage: dronevisioncontrol hand [OPTIONS]

Options:
  -i, --ip TEXT       pilot IP address, ignored if serial is provided
  -p, --port INTEGER  port for UDP connections
  -s, --serial TEXT   connect to drone system through serial, default device
                      is /dev/ttyUSB0
  -f, --file PATH     file to use as source instead of the camera
  -l, --log           log important info and save video
  -h, --help          Show this message and exit.


Usage: dronevisioncontrol follow [OPTIONS]

Options:
  --ip TEXT           pilot IP address, ignored if serial is provided
  -p, --port TEXT     pilot UDP port, ignored if serial is provided, default is
                      14540
  --sim TEXT          run with AirSim as flight engine, optionally provide ip
                      the sim listens to
  -l, --log           log important info and save video
  -s, --serial TEXT   use serial to connect to PX4 (HITL), optionally provide
                      the address of the serial port
  -h, --help          Show this message and exit.
```

References

[1] J.E. Gomez-Balderas et al. 'Tracking a ground moving target with a quadrotor using switching control: Nonlinear modeling and control'. In: *Journal of Intelligent and Robotic Systems: Theory and Applications* 70.1-4 (2013), pp. 65–78. DOI: 10.1007/s10846-012-9747-9.

[2] V. Bevilacqua and A. Di Maio. 'A computer vision and control algorithm to follow a human target in a generic environment using a drone'. In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 9773 (2016), pp. 192–202. DOI: 10.1007/978-3-319-42297-8_19.

[3] R. Rysdyk. 'UAV path following for constant line-of-sight'. In: 2003. DOI: 10.2514/6.2003-6626.

[4] A.M. Moradi Sizkouhi et al. 'RoboPV: An integrated software package for autonomous aerial monitoring of large scale PV plants'. In: *Energy Conversion and Management* 254 (2022). DOI: 10.1016/j.enconman.2022.115217.

[5] R.I. Naufal, N. Karna and S.Y. Shin. 'Vision-based Autonomous Landing System for Quadcopter Drone Using OpenMV'. In: vol. 2022-October. 2022, pp. 1233–1237. DOI: 10.1109/ICTC55196.2022.9952383.

[6] R. Bartak and A. Vykovsky. 'Any object tracking and following by a flying drone'. In: 2016, pp. 35–41. DOI: 10.1109/MICAI.2015.12.

[7] A. Chakrabarty et al. 'Autonomous indoor object tracking with the Parrot AR.Drone'. In: 2016, pp. 25–30. DOI: 10.1109/ICUAS.2016.7502612.

[8] J. Pestana et al. 'Vision based GPS-denied Object Tracking and following for unmanned aerial vehicles'. In: 2013. DOI: 10.1109/SSRR.2013.6719359.

[9] K. Haag, S. Dotenco and F. Gallwitz. 'Correlation filter based visual trackers for person pursuit using a low-cost Quadrotor'. In: 2015. DOI: 10.1109/I4CS.2015.7294481.

[10] A. Hernandez et al. 'Identification and path following control of an AR.Drone quadrotor'. In: 2013, pp. 583–588. DOI: 10.1109/ICSTCC.2013.6689022.

[11] J.J. Lugo and A. Zell. 'Framework for autonomous on-board navigation with the AR.Drone'. In: *Journal of Intelligent and Robotic Systems: Theory and Applications* 73.1-4 (2014), pp. 401–412. DOI: 10.1007/s10846-013-9969-5.

[12] P.-J. Bristeau et al. 'The Navigation and Control technology inside the AR.Drone micro UAV'. In: vol. 44. 1 PART 1. 2011, pp. 1477–1484. DOI: 10.3182/20110828-6-IT-1002.02327.

[13] J. García and J.M. Molina. 'Simulation in real conditions of navigation and obstacle avoidance with PX4/Gazebo platform'. In: *Personal and Ubiquitous Computing* 26.4 (2022), pp. 1171–1191. DOI: 10.1007/s00779-019-01356-4.

[14] T. Chen et al. 'A Pixhawk-ROS Based Development Solution for the Research of Autonomous Quadrotor Flight with a Rotor Failure'. In: 2022, pp. 590–595. DOI: 10.1109/ICUS55513.2022.9986633.

[15] D.M. Huynh et al. 'Implementation of a HITL-Enabled High Autonomy Drone Architecture on a Photo-Realistic Simulator'. In: 2022, pp. 430–435. DOI: 10.1109/ICCAIS56082.2022.9990214.

[16] *PX4 User Guide*. The Linux Foundation. URL: https://docs.px4.io/main/en/ (visited on 13/01/2023).

[17] Michael H. („Laserlicht"). *Raspberry Pi 4 Model B - Side*. Wikimedia Commons. URL: https://commons.wikimedia.org/wiki/File:Raspberry_Pi_4_Model_B_-_Side.jpg.

[18] Shital Shah et al. *Aerial Informatics and Robotics Platform*. Tech. rep. MSR-TR-2017-9. Microsoft Research, 2017.

[19] *Windows Subsystem for Linux Documentation*. Microsoft Learn. URL: https://learn.microsoft.com/en-us/windows/wsl/ (visited on 05/04/2023).

[20] Renderpeople. *Over 4,000 Scanned 3D People Models*. URL: https://renderpeople.com/ (visited on 16/01/2023).

[21] Microsoft. *Build on Windows - Airsim*. URL: https://microsoft.github.io/AirSim/build_windows/ (visited on 16/01/2023).

[22] Matt Hawkins. *Raspberry Pi GPIO Header with Photo*. URL: https://www.raspberrypi-spy.co.uk/2012/06/simple-guide-to-the-rpi-gpio-header-and-pins/ (visited on 14/01/2023).

[23] Fan Zhang et al. *MediaPipe Hands: On-device Real-time Hand Tracking*. 2020. DOI: 10.48550/ARXIV.2006.10214.

[24] Google LLC. *Hands - mediapipe*. URL: https://google.github.io/mediapipe/solutions/hands.html (visited on 14/01/2023).

[25] Google LLC. *Pose - mediapipe*. URL: https://google.github.io/mediapipe/solutions/pose.html (visited on 14/01/2023).

[26] Valentin Bazarevsky et al. *BlazePose: On-device Real-time Body Pose tracking*. 2020. DOI: 10.48550/ARXIV.2006.10204.

[27] Martin Lundberg ("m-lundberg"). *simple-pid*. Version 1.0.1. PyPi. URL: https://pypi.org/project/simple-pid/ (visited on 20/01/2023).

[28] PX4 team. *Safety Configuration (Failsafes) | PX4 User Guide*. Dronecode foundation. URL: https://docs.px4.io/main/en/config/safety.html (visited on 21/01/2023).

[29] PX4 team. *Setting up a Developer Environment (Toolchain) | PX4 User Guide*. Dronecode foundation. URL: https://docs.px4.io/main/en/dev_setup/dev_env.html (visited on 18/03/2023).