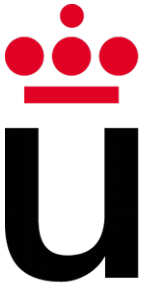


Universidad
Rey Juan Carlos

GRADO EN INGENIERÍA AEROESPACIAL EN AERONAVEGACIÓN

Curso Académico 2023/2024

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 se realizó el día de de 2023, siendo calificada por el siguiente tribunal:

Presidente:

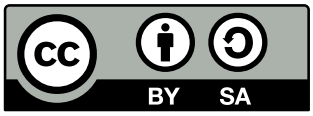
Secretario:

Vocal:

y habiendo obtenido la siguiente calificación:

Calificación:

Fuenlabrada, a de de 2023



Acknowledgements

I would like to take this opportunity to express my deepest gratitude to the many individuals who have played a significant role in the completion of this bachelor thesis.

First and foremost, I want to thank my family for their unwavering support and patience throughout my academic journey. They never doubted me, even when I did it myself. Thank you as well for your help during this project, for being there when I needed a camera person or an extra hand to fly the drone. To my sister, especially, thank you for putting up with me these two years while I inched along this project.

I also extend my thanks to my supervisors for their guidance, expertise, and invaluable feedback throughout the thesis project. Your dedication to my academic growth and your willingness to share your knowledge have been indispensable.

I would also like to express my gratitude to all my friends and colleagues who were by my side during this journey. Your camaraderie and the frequent inquiries about the status of my ”never-ending drone project” pushed me towards the finish line.

This thesis represents the culmination of years of hard work and dedication, and I am grateful for the support of everyone who stood by my side.

Thank you all for being a part of this journey.

Resumen

La popular plataforma de código abierto PX4 surge con el objetivo de facilitar el desarrollo de nuevas soluciones para vehículos aéreos no tripulados y su integración con nuevas tecnologías para favorecer su acercamiento al público general. Esta tesis pretende demonstrar cómo esta plataforma puede ser empleada para desarrollar y evaluar estrategias de control que integren técnicas de visión artificial existentes, utilizando sus resultados para controlar el movimiento de un vehículo aéreo y empleando componentes electrónicos de bajo presupuesto. Para mostrar las capacidades de las herramientas disponibles, se presenta un método de control viable que permite a un dron usar una cámara abordo del mismo para identificar y realizar un seguimiento de una persona en su campo de visión, cuyos movimientos son reflejados por el vehículo.

Abstract

The popular open-source platform PX4 aims to facilitate the development and integration of new solutions for unmanned aerial vehicles, making them approachable for the common developer. This thesis aims to demonstrate how this platform can be used to develop and test control strategies that integrate existing computer vision techniques, using their input to control the movement of an aerial vehicle while employing readily available and affordable hardware components with basic specifications. To showcase the tools available, 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.

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. Initially 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 progress 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 obtained 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 broader range of users. While integrating 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, the most convenient 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 exploring 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 emerges from an 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 (UAVs). To achieve this objective, an application named DroneVisionControl has been developed. It aims to serve 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 project's complexity. Two primary architectures for integrating software and hardware in vision-based scenarios have been proposed, with one scenario involving offboard vision control (offboard configuration) and the other integrating the camera and vision computations aboard the aircraft (onboard configuration).

The first scenario involves a proof-of-concept control mechanism, the hand gesture solution, that translates hand gestures in front of an independent camera to flight commands, thereby controlling the vehicle's movement. This mechanism functions as a starting platform to begin development and test basic integration between the tools employed. The second scenario, the follow solution, builds upon the first to integrate the camera and vision computation onboard the vehicle to develop a tracking and following control solution. This mechanism will track and follow a person standing in the drone's field of view, mirroring their movements in three dimensions.

A dedicated development environment was employed to further the development of these control mechanisms. 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 actual flight tests in an unmanned vehicle. The critical 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 allows secure and comprehensive testing of work-in-progress control solutions to ensure that they become reliable.

For this project, all the code has been implemented in Python and run 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 are not limited to the PX4 ecosystem but can be applied to any autopilot platform that exposes control APIs.

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 evaluate the control solutions in the simulation engine comprehensively.
- 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.

- 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

Figure 1.1: Timeline for the development of the project.

This project was carried out over a period of approximately one and a half years while handling the parallel commitment of working full-time. The development process was divided into three distinct phases, totalling approximately 250 hours of dedicated effort. An additional 250 hours were dedicated to writing the report, which was carried out in parallel with the research and programming work. The development 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 deeper understanding of the PX4 system. This research involved studying the platform's architecture, capabilities, and documentation. Following the research phase, a proof-of-concept control mechanism was developed to translate 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 developing an advanced tracking and following control solution. Research was conducted to explore various tracking and detection methods, including computer vision algorithms and techniques. The follow solution was implemented building upon this research to enable 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, the research explored 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.

This distribution of time and tasks 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 an introduction, a conclusion and 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 introduces 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 reviews the relevant literature and the current state-of-the-art for UAV vision-based control solutions. It also discusses the preexisting technologies and tools employed by the project.

The third chapter covers the simulation environments used throughout the development process and the architecture for the project's hardware and software, discussing the design decisions that went into selecting the individual components.

The fourth chapter presents the testing methodology used throughout the project, from initial simulations to flight tests. This chapter details the results of this testing process and highlights the conclusions and insights gained from each testing phase.

The final chapter concludes the thesis by summarizing the project's key findings and drawing conclusions about the effectiveness and limitations of the control system. Additionally, this chapter provides suggestions for future development and improvements to 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 the fields of tracking and following aboard UAVs and available hardware and software platforms for control, 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 utilising 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 flight control techniques and computer vision mechanisms to present an easy-to-use platform that combines existing algorithms to achieve robust systems, 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 explicitly focused on this platform. However, some studies demonstrate the possibilities offered by

the PX4 software and the Pixhawk flight controllers that run it. In [4], aerial images are analysed 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 longer have been explored further on vision-based control projects. Specifically, the Parrot *AR.Drone* camera-enabled quadcopter is the most widely used. The Parrot platform exposes an API that allows WiFi control from an external offboard computer. In [6], [7], [8], and [9], the *AR.Drone* is utilised for implementing object tracking and following solutions in various environments and conditions, emphasising particularly 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 using embedded navigation, control technology and 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 fully-built low-cost vehicle for developing high-level autonomous guidance and control without concern for hardware interactions. In contrast, the PX4 platform provides a more comprehensive environment, allowing for customisation at each system layer, from the basic sensors to the high-level control software

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. These simulators reduce 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 extensive PX4 ecosystem for developing control algorithms.

In summary, the literature reviewed indicates that the PX4 platform offers unique advantages stemming from its open-source nature, such as its extensibility, compatibility with a wide range of simulators, and the ability to personalise each layer of the system. While previous research has primarily focused on low-cost platforms like the *AR.Drone*, there are many avenues left to explore in the PX4 ecosystem and its potential for vision-based control solutions. The studies mentioned 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 recognised standard, there is ample room for further research and development. The platform's simulator capabilities provide a valuable opportunity to explore and refine vision-based control algorithms in a virtual environment, enabling more efficient and cost-effective 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 already existing 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 [16] is a widely utilised autopilot flight stack designed for 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, or submersibles. It accommodates both stock and custom-made drones, 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 with the Dronecode Project [17], a comprehensive drone ecosystem. The 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 [18] 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, 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 [19], 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 without being concerned with the underlying MAVLink messages.

QGroundControl

QGroundControl [20] 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 PC.

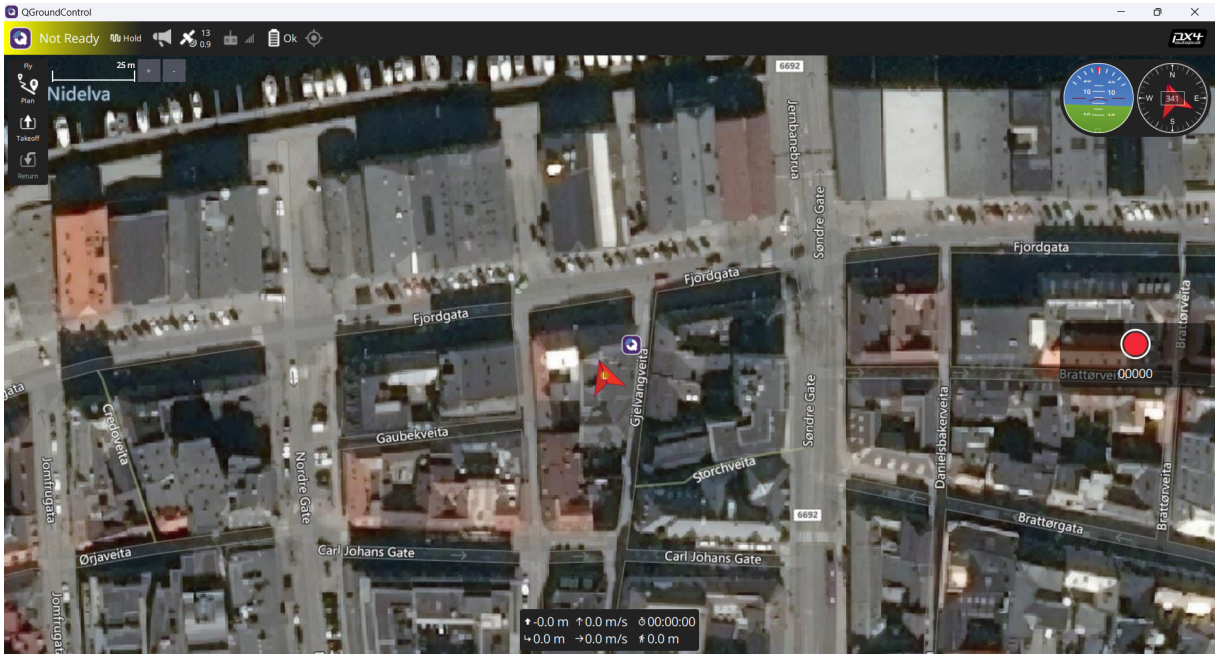


Figure 2.1: Main interface for the QGroundControl program.

Unreal Engine and AirSim

Unreal Engine [21] 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 [22], 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, computer vision, and reinforcement learning algorithms for autonomous vehicles. To achieve this, AirSim offers the `airlib` library 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

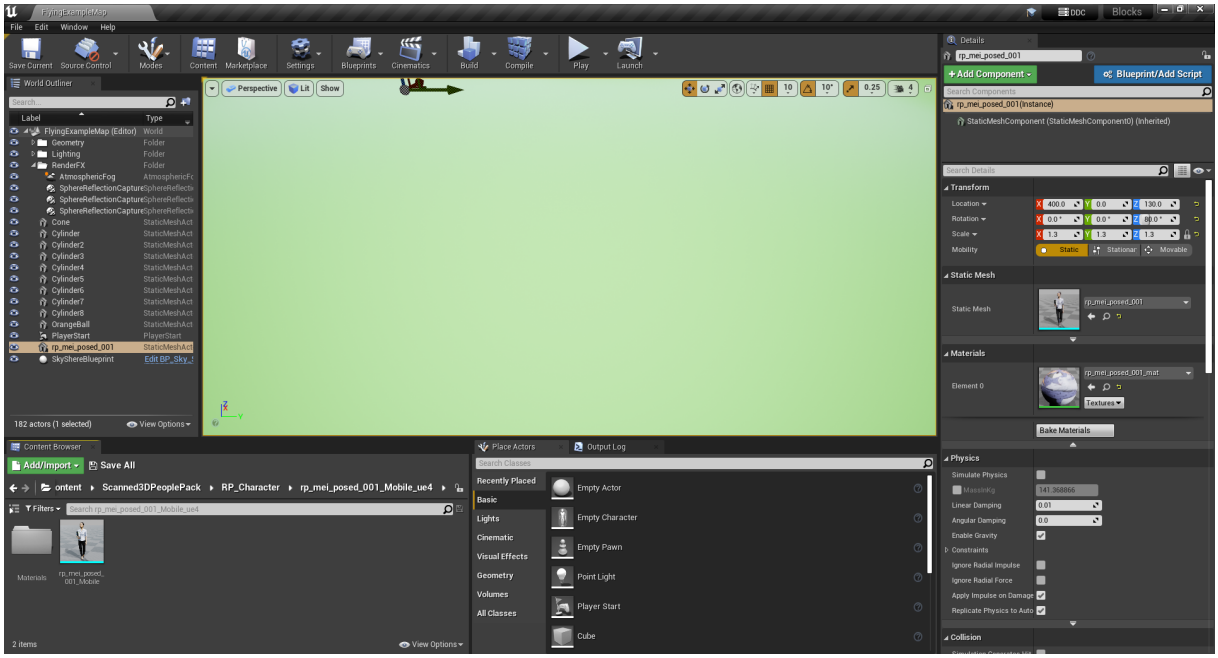


Figure 2.2: Project interface for the Unreal Engine.

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 thorough testing of the algorithms 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 [23], 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. Both the hand and human pose MediaPipe solutions will be used within this project to develop control mechanisms that react to the different positions detected (see Sections 3.4 and 3.5).

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. For this project, both the software code [24] and this report [25] 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. The project's GitHub page [26] contains an overview of the goals and implementation details, accompanied by videos showcasing the tests conducted in Chapter 4.

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 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 and Appendix C. These libraries collectively contribute to effectively handling and analysing image-related tasks throughout the project.

2.2.2 Hardware

Pixhawk 4

The Pixhawk 4 is an advanced autopilot module developed by Holybro [27] in collaboration with the Dronecode Project team. It is designed based on the open hardware design of the Pixhawk project [28] and optimized to run the PX4 flight stack on the NuttX [29] 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. These features make the Pixhawk 4 an integral component in the project's hardware setup. 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. A companion computer is a separate computer connected to the flight controller to allow computationally expensive features like image processing for vision-based control. This project uses the Raspberry Pi 4 as the companion computer connected to the Pixhawk 4.

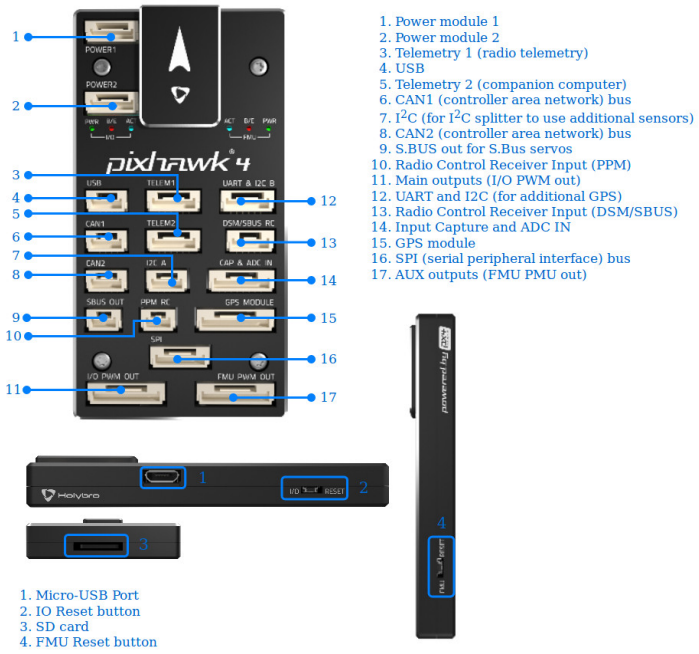


Figure 2.3: Side views and connector map for the Pixhawk 4 autopilot module.

Source: Adapted from *PX4 User Guide* [30].

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 [31], 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.

Figure 2.4: Raspberry Pi 4 Model B

Source: Wikimedia Commons [32].

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

Holybro X500

Figure 2.5: Fully assembled X500 kit.

Source: Adapted from *PX4 User Guide* [30].

The Holybro X500 [33] is a quadcopter designed explicitly by Holybro to be compatible with the PX4 autopilot system. It is supplied as a complete development kit with all the 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, a Radio Control (RC) receiver, and a telemetry radio. The kit does not include a camera for computer vision applications, but the frame leaves some space to add one for this project (see Section 3.2.3). 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 2.5, showcasing the outcome of the assembly process.

Logitech C920 camera

The camera added to the vehicle build to support vision-based control features during flight is the Logitech C920 1080p webcamera [34], chosen for its immediate availability, as it was already in possession. This camera offers a field of view of 78º, 1080p resolution at 30 frames per second and autofocus while weighing 162 grams. In practice, any camera with a contained weight and size that offers enough quality for the pose detection library to function correctly could be used instead to obtain similar results for this project.

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 outside of the simulation environment, focusing on the interaction between the flight controller and the companion computer running the DroneVisionControl application. It discusses two possible hardware configurations: one where the companion computer remains on the ground when the drone flies (offboard configuration) and another where it is placed onboard the vehicle during flight. The key components of the system, including the flight controller, companion computer, and camera, are outlined, along with the methods to exchange information between them. Afterwards, 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 different applications of the PX4 ecosystem. The first one (Section 3.4 - Hand-gesture solution) is designed for the offboard configuration as a proof-of-concept solution where the vehicle is controlled from a ground station by hand gestures. The second (Section 3.5 - Follow solution) demonstrates the applications of the onboard configuration with a person-following solution that controls the drone in flight to maintain a moving person centred 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 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 the virtual 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 follow 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. 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 vehicle's behaviour and executing flight commands. It offers extensive support for simulation, providing developers with a robust environment to test and refine their control solutions.

There are two 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 Linux 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, reducing development time and providing immediate visual feedback on the computer

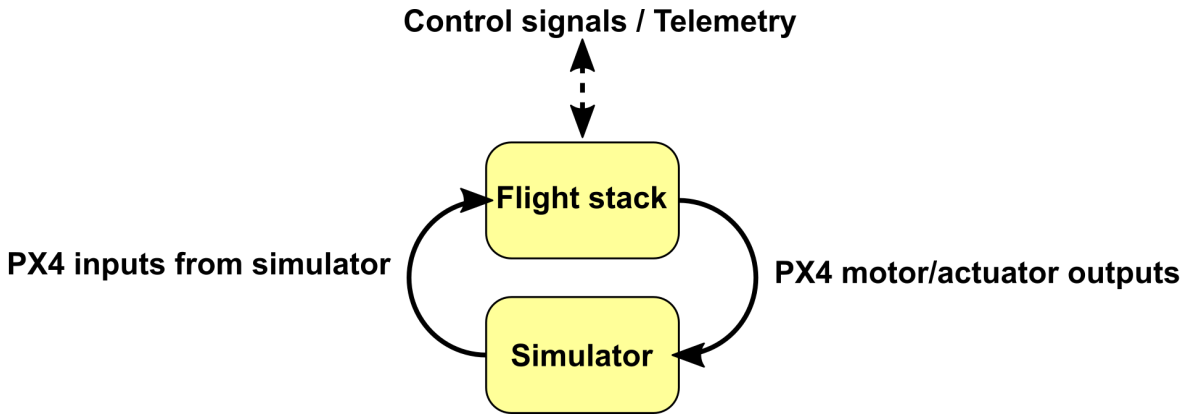


Figure 3.1: Feedback loop during the PX4 simulation.

Source: Adapted from *PX4 User Guide* [30].

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, which interact together through a feedback loop 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.

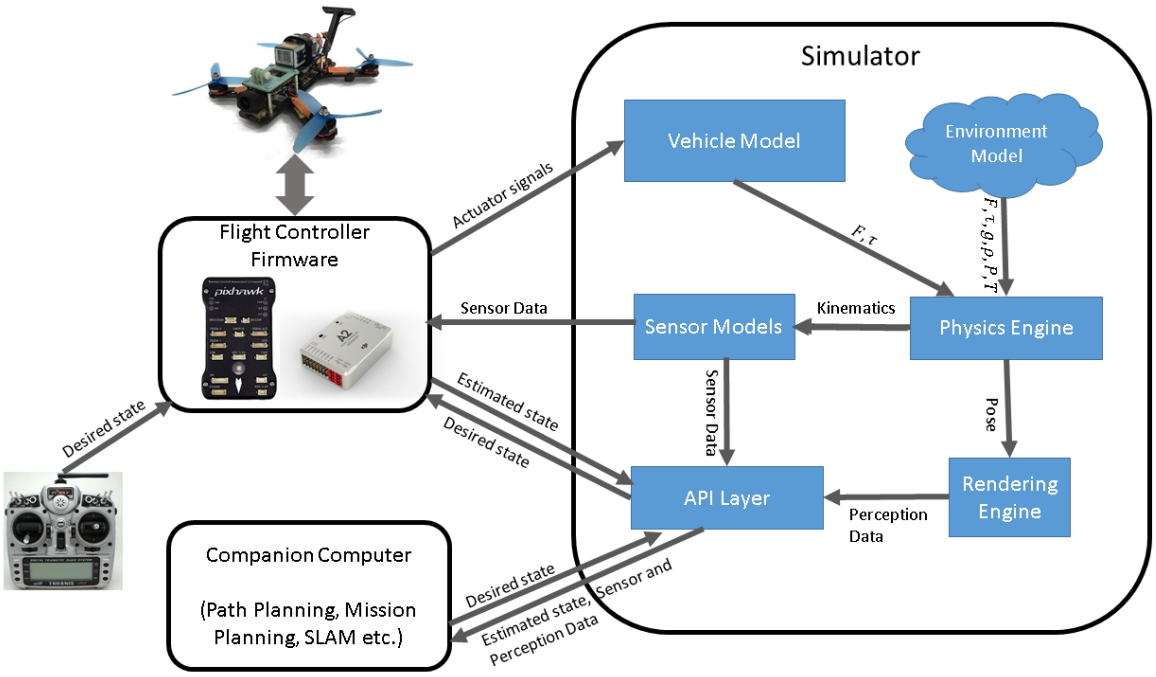


Figure 3.2: High-level overview of how the different components of a simulator interact with the flight stack.

Source: Adapted from *Aerial Informatics and Robotics Platform* [35].

3.1.2 Simulator

The simulator provides 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 directly without going through the flight controller.

In this context, the companion computer is an additional computational device that works together with the flight controller 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 an API that can interact with the flight stack, like the MAVSDK API mentioned in Section 2.2.1. In this project, the communication is done through MAVSDK to isolate the companion computer from the simulator. This separation allows for a straightforward 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 like support for complex simulated worlds, 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 free and paid visual models through its asset library. Its integration with Unreal Engine enables the creation of complex and immersive simulated environments. These features make it especially suited for developing vision-based control solutions, making it the best choice for this project.

3.1.3 Development environment

This section discusses 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 this application is designed to leverage the dedicated processing power of a dedicated 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

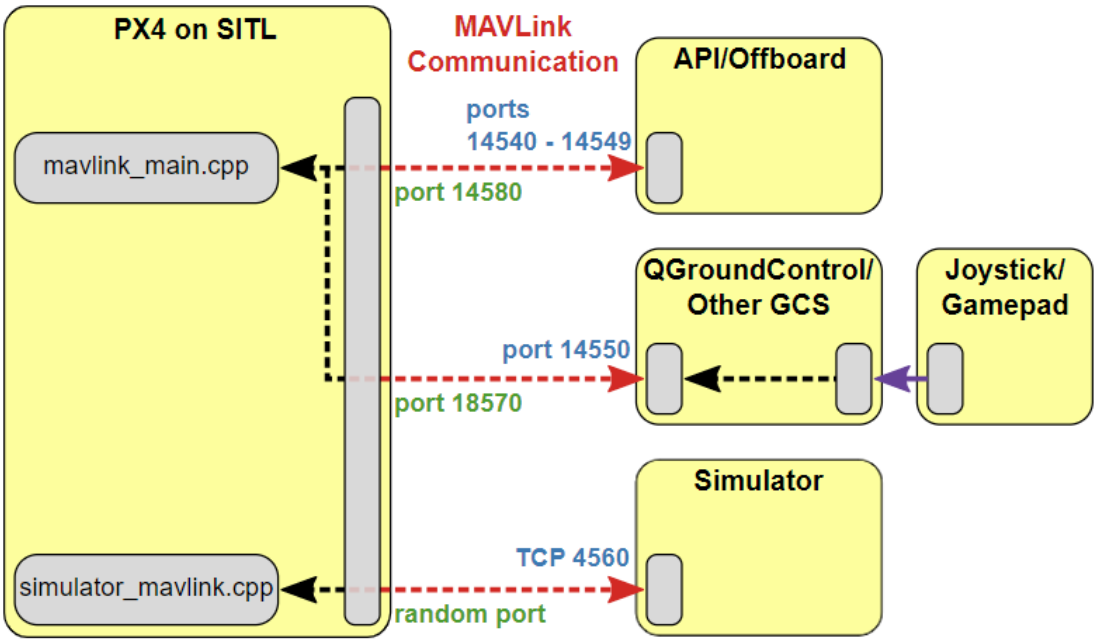


Figure 3.3: Network diagram for the PX4 flight stack during software-in-the-loop simulation. The main MAVLink instance can connect to several API users, like companion computers or the simulator (API layer), and a ground station (QGroundControl). The main communication between PX4 and the simulator is done through a secondary MAVLink instance that transmits sensor data and actuator signals.

Source: Adapted from *PX4 User Guide* [30].

The PX4 platform has an established network configuration for SITL simulation, shown in Figure 3.3. The figure's left side shows the two instances of MAVLink servers running

during simulation to communicate with components outside the flight stack. The first is the main MAVLink instance, 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 (see 2.2.1). These connections are established using the UDP protocol on the port range 14540-14550. Port 14550 is reserved for a ground station by default but can be used by any other external application if QGC is not required. The second server runs exclusively during simulations and connects to the simulator program through a TCP connection instead, 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 flight stack, however, runs best on Linux. To execute both software components simultaneously on a single machine, the Windows Subsystem for Linux (WSL) [36] will be utilised. WSL allows for the execution of a Linux distribution, such as Ubuntu, directly within a host Windows OS, providing a convenient environment for running PX4 SITL alongside AirSim without the need for dual-booting or virtual machines. This configuration simplifies networking on the machine's internal network, as WSL handles the task of establishing the 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 steps needed to configure PX4 on the WSL system are detailed in Appendix A.1.

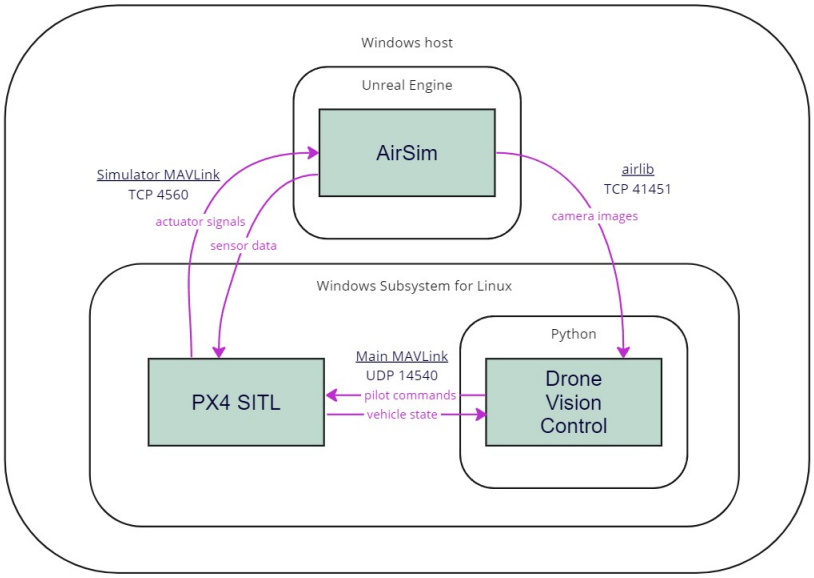


Figure 3.4: Connection diagram of how the three systems interact with each other during SITL simulation.

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, the PX4 flight stack will be used as an intermediary. This way, the DroneVisionControl application does not need to be concerned with whether the flight mechanics are being simulated or determined by sensors in the real world. The link between the developed application and the flight stack is established through the main MAVLink instance, as shown in Figure 3.3 (`mavlink_main.cpp`).

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 or any

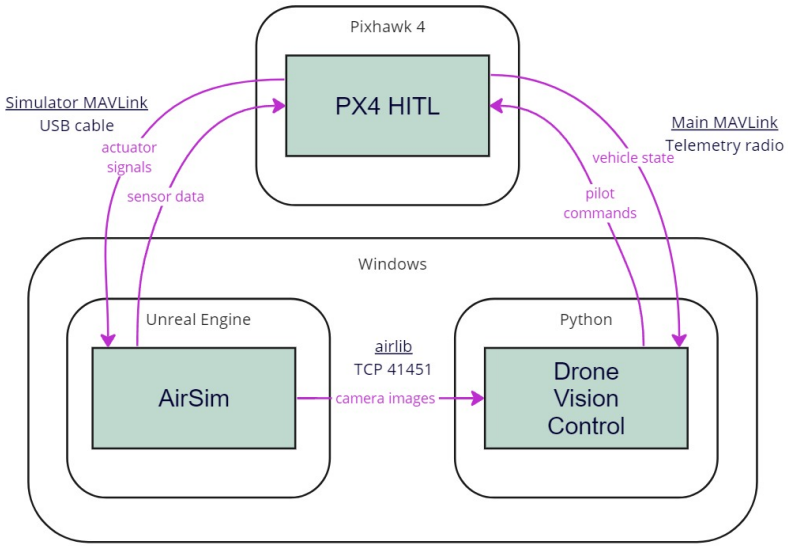


Figure 3.5: Connection diagram of how the three systems interact with each other during HITL simulation.

available telemetry port on the flight controller. To use the telemetry ports to connect to a computer, it is necessary to use either a serial-to-USB converter or 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 to establish separate communication channels.

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

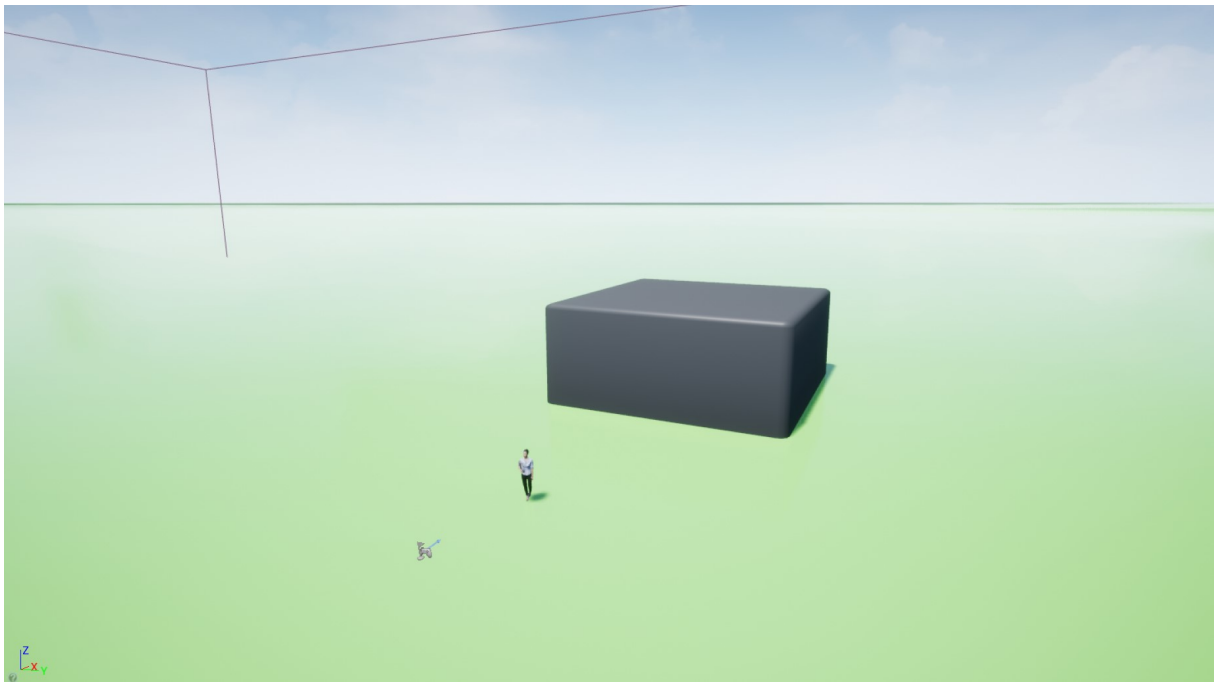


Figure 3.6: Screenshot from the Unreal Engine environment used for testing the computer vision solutions.

rendering engines integrated into Unreal. In the engine, a "project" comprises one or more environments where components like 3D models, cameras, and lighting can be added. 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 DroneVisionControl application is derived from the base AirSim environment and can be found on the project's repository[1]. It contains a test quadcopter vehicle and some 3D shapes used as obstacles. The vehicle model also includes a virtual camera that allows retrieving images from the vehicle's point of view in the simulated world. The main addition to the default environment for this project is the 3D model of a human figure, to be used for testing the person detection and tracking mechanisms in the computer vision solution. The source of this model is a free asset library of human models made by Renderpeople [37] 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

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

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 the AirSim documentation [38] 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 companion computer running the DroneVisionControl application outside of the simulation environment. 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 communicating wirelessly with the flight controller, and onboard, where the companion computer is physically connected with a cable to the flight controller so that they can fly together. The following sections explore the details of each configuration.

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

3.2.1 Top-level components

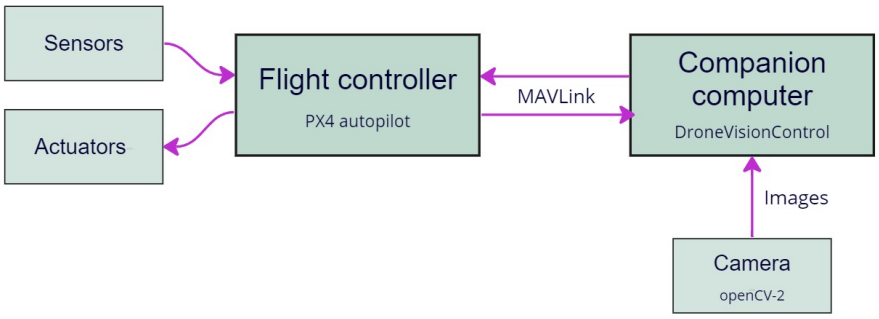


Figure 3.7: Top-level diagram of the hardware/software interactions.

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 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, offboard 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 does not need to move with the vehicle, like in the hand-gesture control solution. In this scenario, the only choice of communication is wireless, 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, like in the follow control solution, the onboard configuration is used. In this scenario, the companion computer with its camera is placed onboard the vehicle alongside

the flight controller. To communicate the two, a direct wired connection is the most suitable option, as it offers a faster and more reliable link than the telemetry radios. Details of this configuration are provided in Section 3.2.3.

The flight controller

The Pixhawk 4 board, designed for the PX4 autopilot, is integral to the project's autonomous flight control. This hardware depends on a set of sensors (gyroscopes, accelerometers, magnetometers, barometers) to determine the vehicle's state and actuators to translate controller outputs into physical movements. Additionally, a GPS or similar positioning system allows for automated flight modes and features like altitude stabilization. In manual flight modes, a Radio Control (RC) system communicates control inputs from a remote unit to a vehicle-mounted receiver. Telemetry radios establish wireless MAVLink connections between ground control stations and PX4-powered vehicles, enabling dynamic parameter adjustments and mission modifications. In actual UAVs, PX4 software runs on the dedicated Pixhawk hardware, whereas in simulated environments, all sensor and actuator components are emulated on the same computer.

DroneVisionControl and the companion computer

The DroneVisionControl application that runs on the companion computer utilizes the Python programming language. Python offers several advantages for projects of this 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 [39] 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 and wireless transmission of images for offboard processing are feasible, they are beyond the scope of this project. 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.

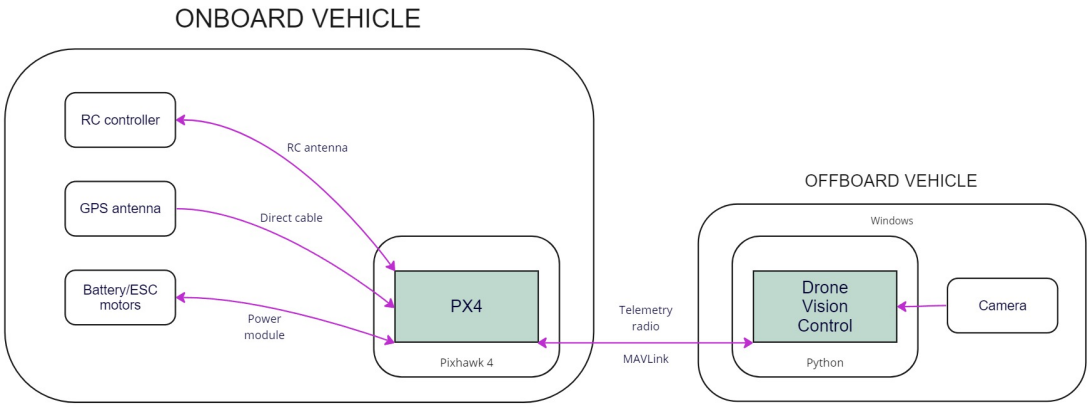


Figure 3.8: Offboard configuration connections.

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, like the DroneVisionControl application, 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.

This project utilizes the Holybro SiK Telemetry Radio [40] for physical tests. These radios are small, lightweight, and cost-effective, offering a range of more than 300 meters (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 and can achieve exchange rates of up to 250kbps. The default baudrate for the connection is 57600.

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.

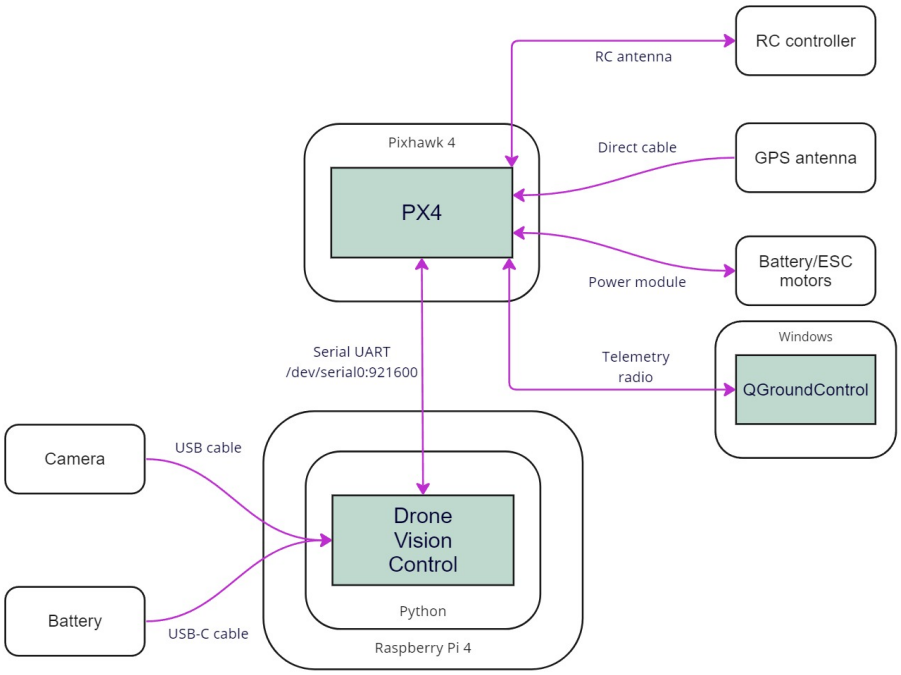


Figure 3.9: Overview of the onboard configuration. All connections are contained inside the vehicle's frame.

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 small enough to fit on the vehicle's frame and 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 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 Pi's officially supported operating system, called Raspbian, is a Debian-based version of Unix, which simplifies the transition from the WSL test environment.

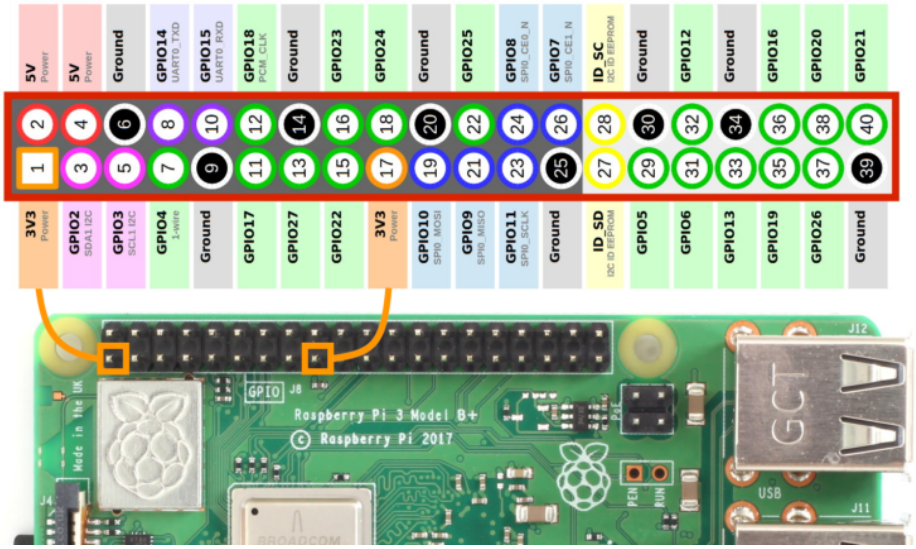


Figure 3.10: The Raspberry Pi microcomputer, with its 40-pin GPIO header marked in red and annotated pinout.

Source: Adapted from *Raspberry Pi GPIO Header with Photo* [41]

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.

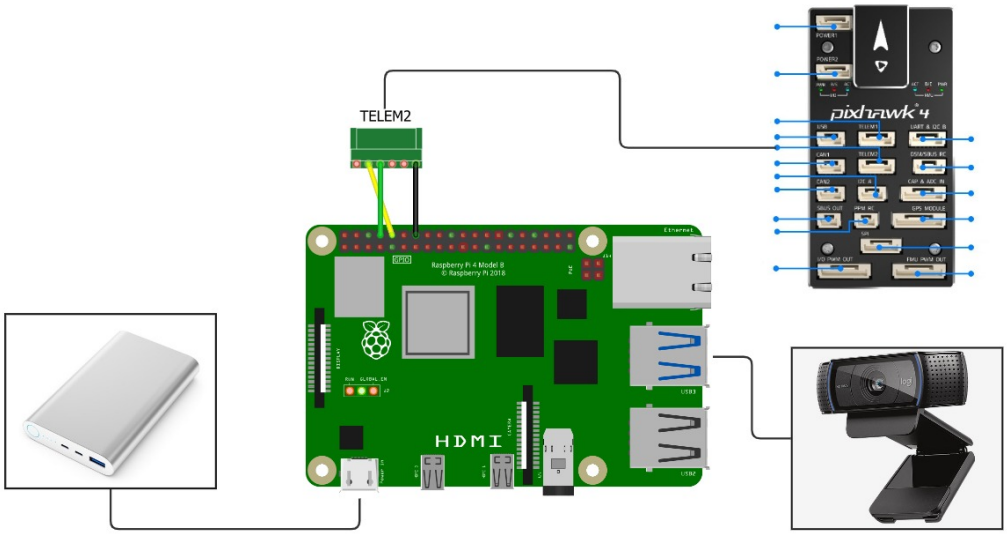


Figure 3.11: A diagram of the wired connections from the Raspberry Pi 4 to the secondary battery and the flight controller (`TELEM2`).

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 [42] 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 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 Chapter 4 is the Logitech C920 webcam described in Section 2.2.2. 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[3].

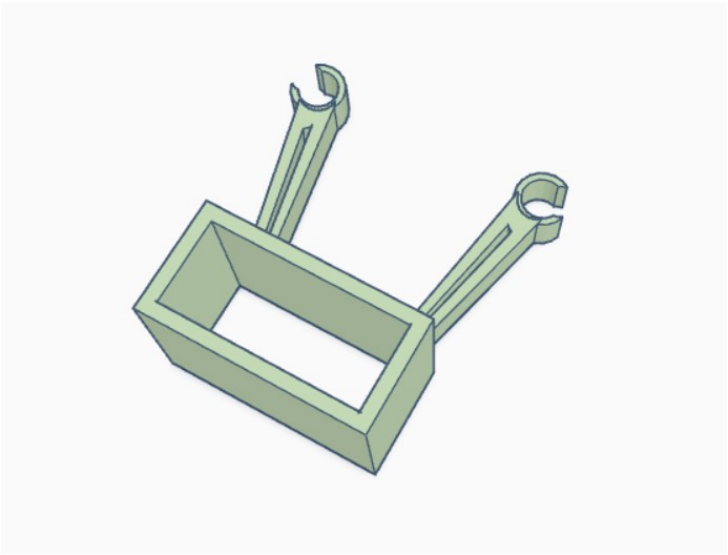


Figure 3.12: 3D model for the camera mount designed for the Holybro X500 frame.

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`. 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 provided in Table 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.

[3] https://github.com/l-gonz/tfg-giaa-dronecontrol/blob/main/data/camera-holder.stl

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 4.3.2.

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 4.3.3, 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

This section presents the software architecture of the DroneVisionControl application. Except when noted as an external library, all the functionality described has been developed specifically for this project. The complete codebase is located in a GitHub repository[4]. 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

[4] https://github.com/l-gonz/tfg-giaa-dronecontrol

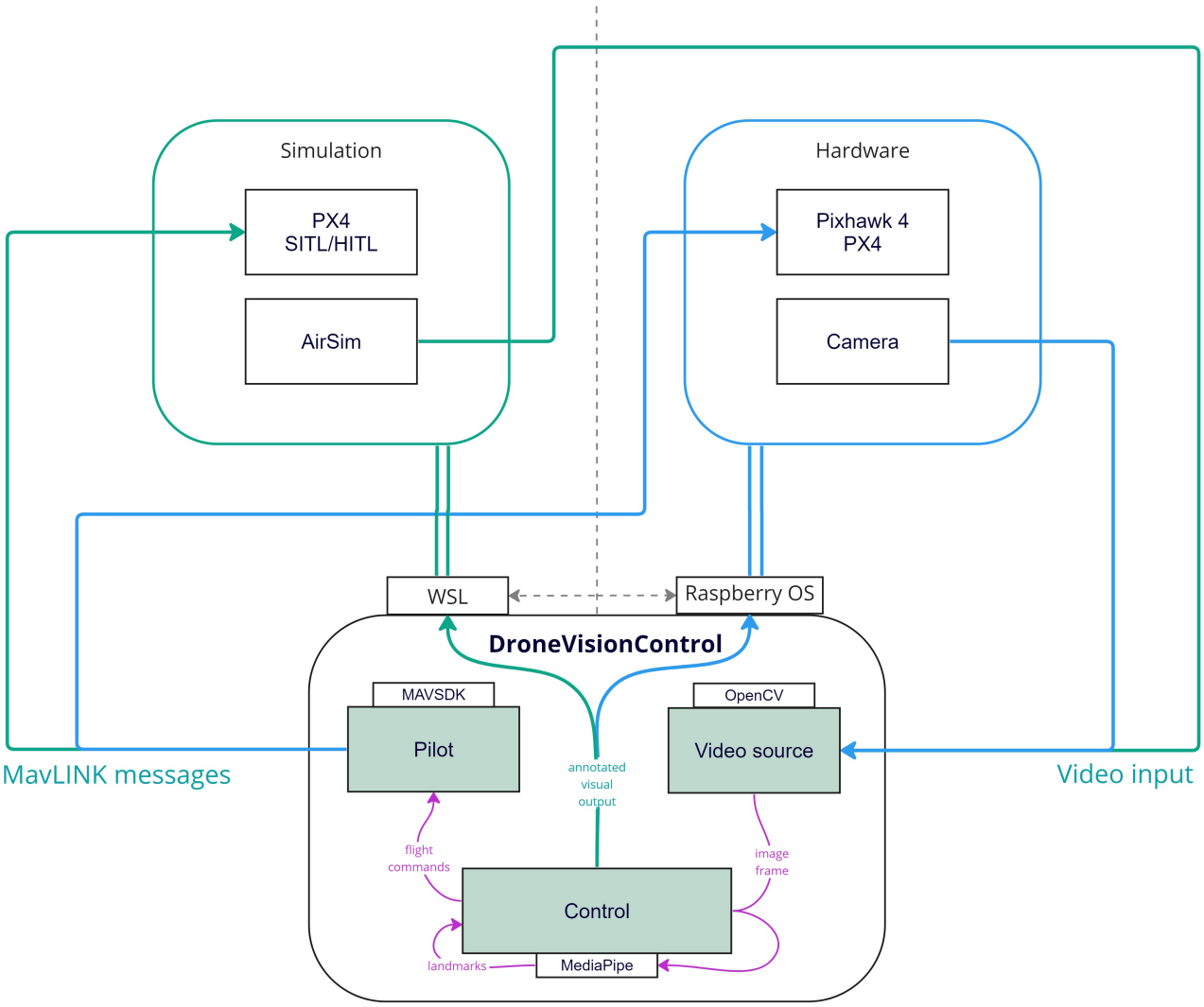


Figure 3.13: Structure of the DroneVisionControl application main modules (green background) and its interactions with the external components (white background). The information flow is shown in purple arrows for internal communication, green arrows for the simulation environment and blue arrows for hardware interactions.

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.1.

3.3.1 Pilot module

The pilot module[5] serves the purpose of providing access to the rest of the application for sending and receiving messages from the PX4 controller through the external 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` [43], 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, defined in Python as a convenient way to retrieve data asynchronously. These are accessed with the `async for` syntax.

The pilot module 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. These commands can be sent to a vehicle once a connection has been established. To enable direct control of the vehicle's velocity, a special flight mode defined by PX4, called Offboard mode [44], is required (not related to

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

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 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.

As an additional feature, the pilot module also implements an asynchronous queue that can be used to execute actions sequentially. The queue is periodically polled to search for newly added commands to execute, ensuring that each action waits until the previous one has finished and the vehicle is in the desired state before starting the next action. Using this queue is optional and depends on the specific behaviour desired from the control solution. All pilot commands can either be executed immediately, interrupting whichever action is being executed at that time, or be added to the queue to be executed at a later point.

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[6] 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.

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

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

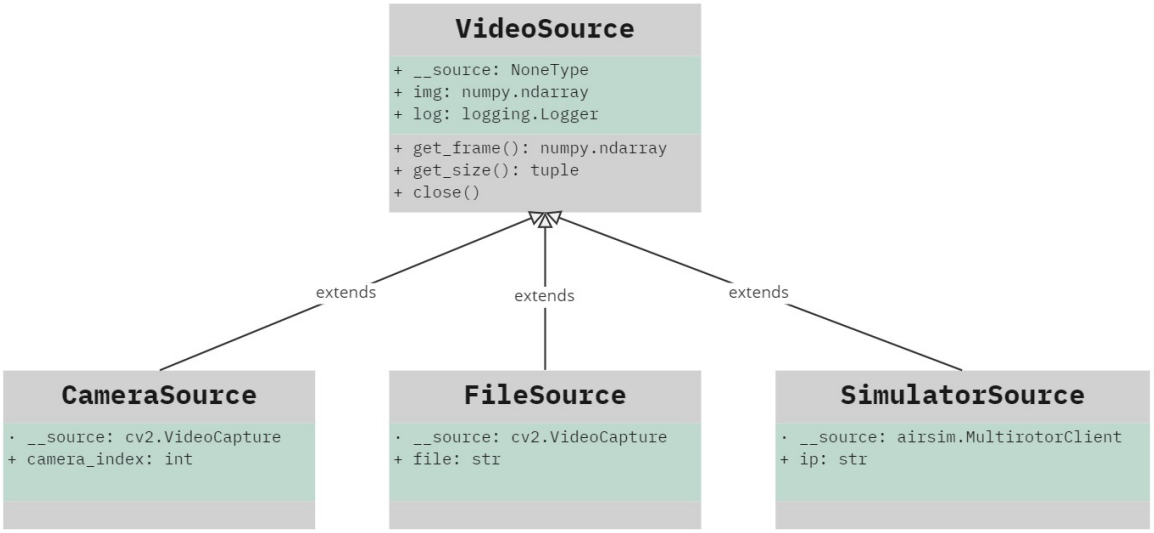


Figure 3.14: Diagram of inheritance on the video source classes available to retrieve image data.

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 that can be divided into three steps:

1. 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. In the hand solution, the coordinates match the features of a detected hand and in the follow solution, the features of a person, as shown in Figures 3.15 and 3.19, respectively.

2. Various calculations specific to each solution are applied to these coordinates. In the hand solution, gestures are extracted from the coordinates through vector calculations, and in the follow solution, a bounding box is drawn around the person to determine its relative position on the field of view.

3. The commands sent to the pilot module are determined based on the input calculated from the coordinates. In the hand solution, each gesture is mapped to a single command, and in the follow solution, velocity commands are extracted from the bounding box position through a PID controller.

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[7] 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

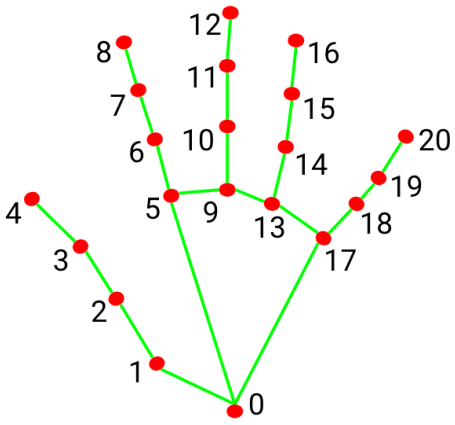
[7] https://github.com/l-gonz/tfg-giaa-dronecontrol/blob/main/dronecontrol/tools/test_camera.py

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.1 provides a comprehensive breakdown of all the tool's options.

This tool will be used extensively during the tests carried out in Chapter 4 to validate individual application components in different scenarios without involving complex control mechanisms.

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 (see Figure 3.8).



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* [45].

The control module for this solution is the `mapper` module[8]. 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 the MediaPipe library [46]. If a hand is detected in the image, MediaPipe returns a series of 2D coordinates called landmarks that identify each joint in the hand as depicted in Figure 3.15. The landmarks received from the external library are converted into discrete gestures, such as an open palm, closed fist, or finger

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

pointing in different directions by the developed `gesture` module[9]. Each detected gesture is then mapped to a command, which is queued to the pilot module and executed once the previous commands have been completed.

The conversion from landmarks to gestures performed by the `gesture` module has been designed as a simple but effective way to identify a limited set of hand positions without relying on complex machine-learning solutions. This conversion works by using the landmark coordinates to define two vectors per finger of the hand. One of the vectors points from the base of the hand (wrist landmark) to the base of each finger (points 1, 5, 9, 13, and 17 in Figure 3.15), and the other vector points from the base of each finger to its tip (points 4, 8, 12, 16, and 20 in Figure 3.15). With the vectors defined, the dot product vector operation is employed to calculate the relative angle of each finger with the base of the hand, as shown in Figure 3.16. Equations 3.1 and 3.2 show example calculations for the index finger.

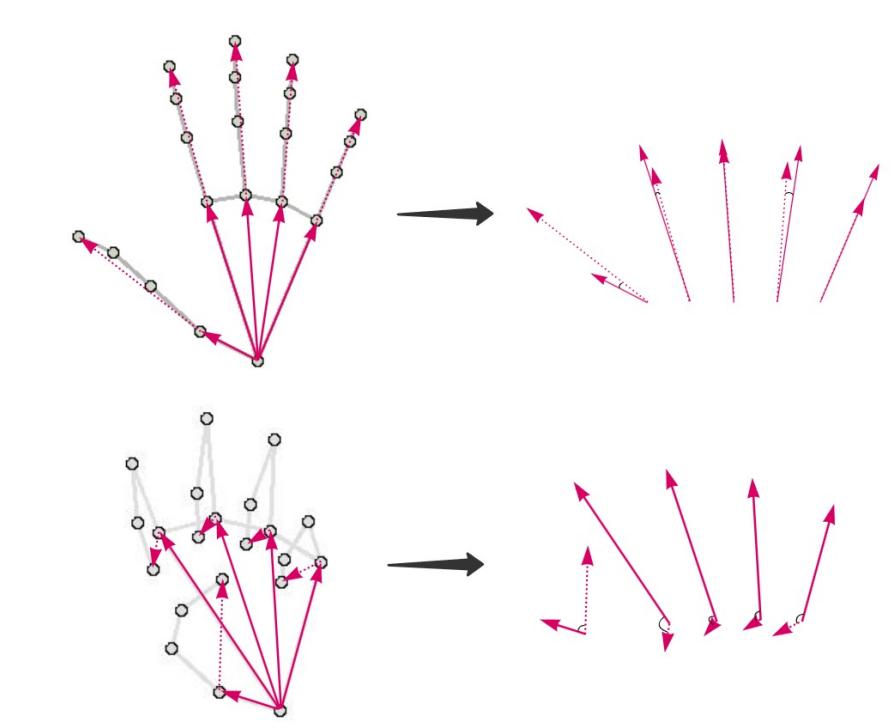


Figure 3.16: Vectors extracted from the detected features on an open and a closed hand are used to calculate reference angles to determine hand gestures.

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

$$\vec{a} = P_5 - P_0 \qquad \vec{b} = P_8 - P_5 \tag{3.1}$$

$$\angle(\vec{a}, \vec{b}) = \arccos\left(\frac{\vec{a} \cdot \vec{b}}{|\vec{a}||\vec{b}|}\right) \tag{3.2}$$

where: $\vec{a}$ = vector for the index base
$\vec{b}$ = vector for the index finger
P_x = coordinates of landmark x according to Figure 3.15

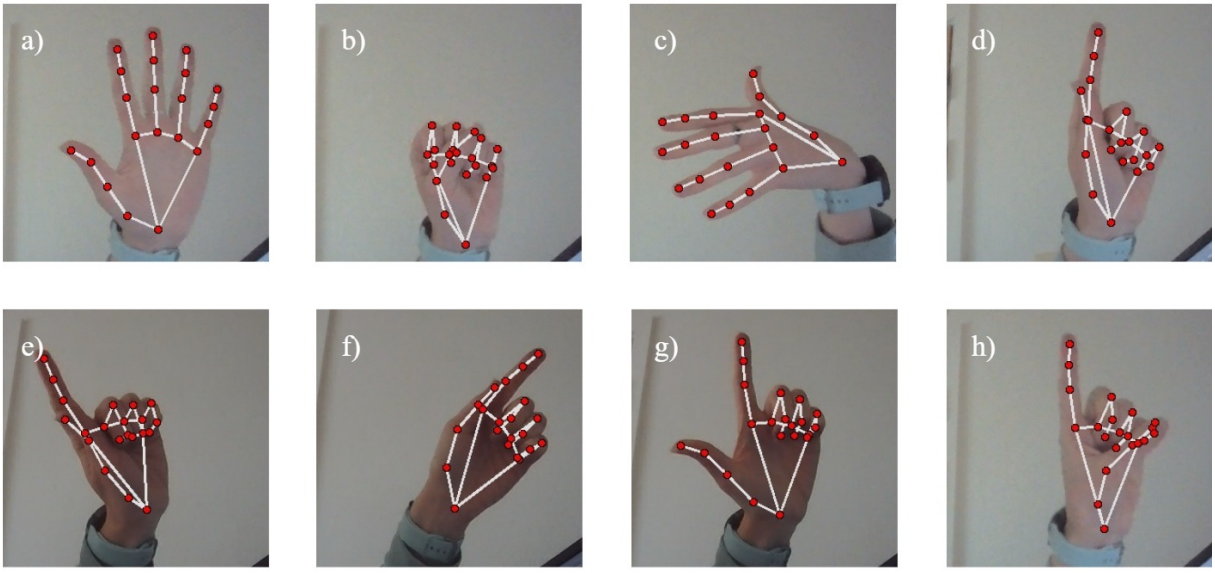


Figure 3.17: Gestures detected by the program to control the drone's movement. a) Open hand, b) fist, c) backhand, d) index point up, e) index point left, f) index point right, g) thumb point left, h) thumb point right

By comparing the calculated angles to different thresholds, the module can determine whether each finger is extended or folded and in which direction it points. The thresholds have been determined experimentally by analyzing the angles calculated for the different fingers for different hand positions. Each possible gesture (see Figure 3.17) is then defined by the direction towards which each finger points, as 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 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 manually, according to the mapping defined in the `input` module[10] and outlined in Appendix B.2. A complete run of this solution is detailed in Section 4.4.3.

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

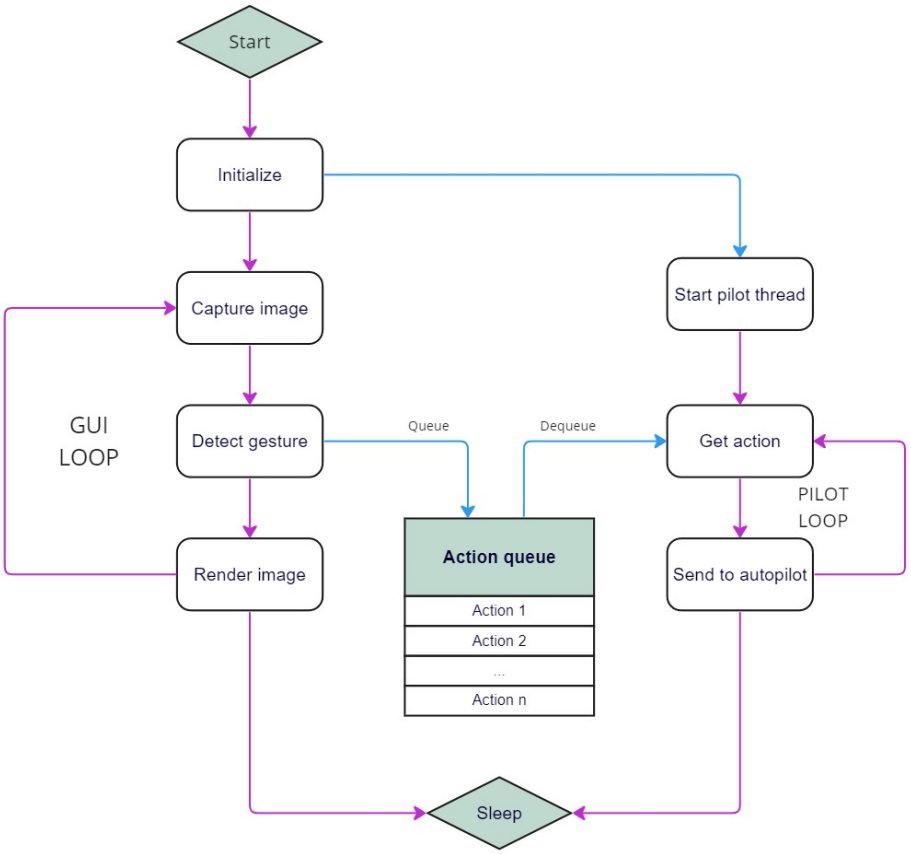


Figure 3.18: Execution flow for the running loop in the hand-gesture control solution.

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 detect the relative position of a 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 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 through a switch in the RC controller configured in QGroundControl for this purpose. The self-guided control also stops automatically if the connection to the computer is lost or any of the available failsafes are triggered, such as low battery or loss of GPS signal. These safety measures are explained in more detail in Section 3.5.2.

When the follow mechanism is engaged, the system continuously retrieves images from the onboard camera. These images are processed using the MediaPipe Pose [47] computer vision library to extract pose landmarks in the form of 2D coordinates. Figure 3.19 shows the features extracted by the external algorithm and their correspondence to the human body.

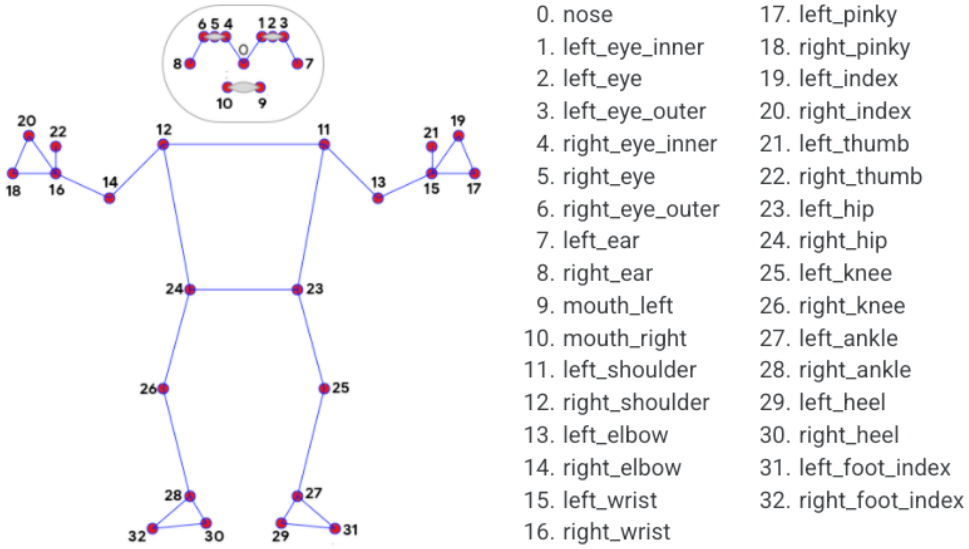


Figure 3.19: Landmarks extracted from detected human figures by the MediaPipe Pose solution.

Source: Adapted from *Pose - mediapipe* [48]

The landmarks received from the MediaPipe library are processed inside the application to draw a bounding box around the detected person and validate that they match the expected pose of a person standing up. The bounding box is calculated as a rectangle with its centre in the midpoint between the minimum and maximum x and y coordinates of the landmarks received and a width and height 10% bigger than the difference between the maximum and minimum coordinates. There are two additional checks to validate that the landmarks extracted from the external library match a person. The first check simply verifies that the height of the bounding box is greater than its width. The second asserts that the features identified with the head, shoulder, hip, knee, and ankle are situated in the correct order from top to bottom of the image. That is, the y coordinate of the head landmark should always be bigger than the y coordinate of the shoulder

landmark, which should in turn be bigger than the y coordinate of the hip landmark. These validation checks are implemented in the `image_processing`[11] module.

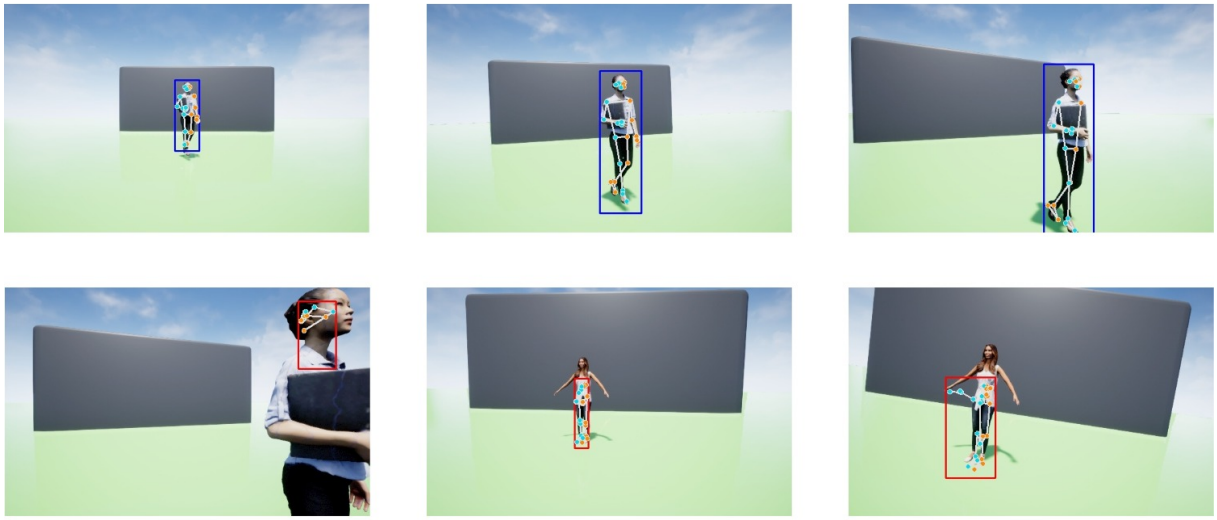


Figure 3.20: Valid (blue) versus invalid (red) poses detected by the follow solution.

Figure 3.20 shows some examples of the validation checks running on the raw landmarks extracted from an image. Once a valid bounding box is defined around the target person, its position in the image is sent to a control mechanism consisting of two independent PID controllers. These control the vehicle's velocity in the yaw and forward direction, respectively. 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 do not match the expected pose (invalid bounding box). The vehicle will resume moving when a valid person is detected again.

As previously mentioned, DroneVisionControl implements two separate PID controllers to control the vehicle's velocity. The first one is responsible for controlling the yaw velocity of the vehicle, responding to movement in the horizontal axis of the camera's field of view to attempt to keep the person centred horizontally in the field of view. To achieve this, the process variable or input fed to the controller is x-coordinate of the bounding box position in the camera image, normalized to the width of the image, and the setpoint is the middle of the screen (0.5). The second PID controller directs 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. Since the perceived height in the image depends on the camera perspective, the setpoint for this controller should be determined experimentally for each video source. The setpoint for the physical camera used for flight test is decided as 0.5 by analysing images taken during flight as described

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

in Section 4.4.2. In the AirSim simulator, the same 0.5 setpoint for the forward controller maintains the vehicle and the person separated by approximately 4 meters.

Figure 3.21 shows how the inputs for each controller are extracted from the coordinates of the bounding box detected around the figure. Section 3.5.1 describes the mechanisms the PID controllers employ to translate the input positions to output velocities for the project.

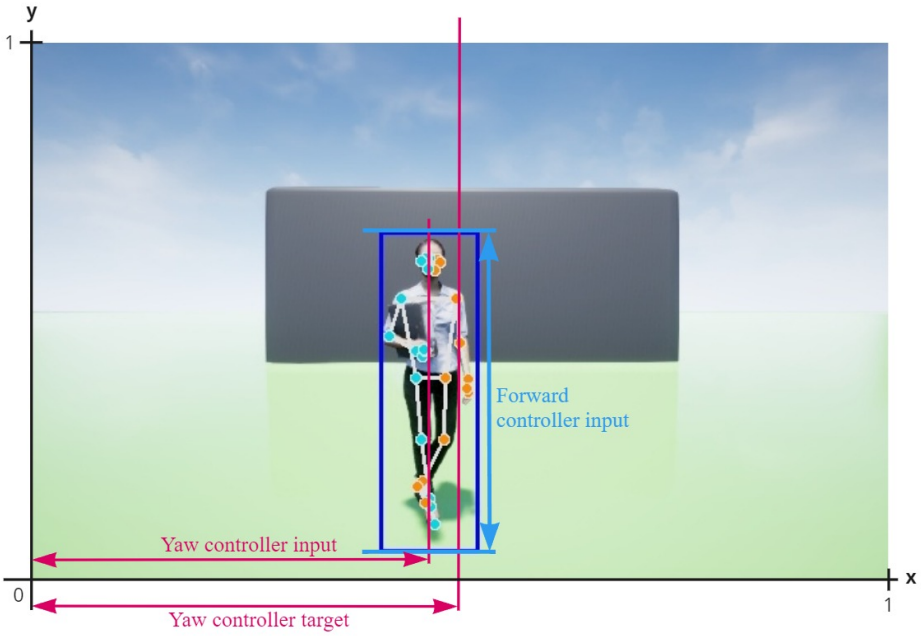


Figure 3.21: Calculation of controller inputs from bounding box drawn around the detected figure.

The structure for follow solution control module[12] 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 bounding box is used as input for the PID controllers, which generate velocity outputs to be sent to the pilot module. Keyboard input is also available to send manual control commands to the vehicle according to the mapping in Appendix B.2. Unlike in the hand-gesture control solution, all the commands sent here to the pilot module (velocity commands and keyboard input) are run immediately instead of queued for execution. A complete execution of this solution is shown in flight in Section 4.4.4.

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

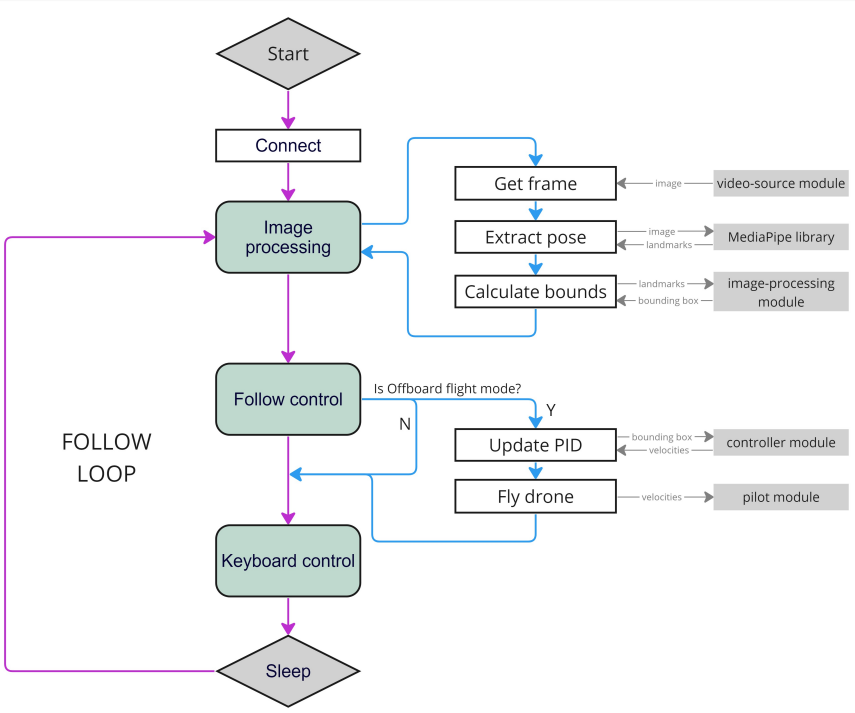


Figure 3.22: Execution flow for the running loop in the follow control solution.

3.5.1 PID controllers

A proportional-integral-derivative (PID) is a control loop mechanism commonly used in systems requiring continuously modulated control. One of its biggest advantages is that it relies only on the response of a single measured process variable, not on knowledge or a model of the underlying process. The controller works by continuously calculating an error value from the difference between the received input on a chosen process variable and the desired set point for that variable. From the error value, the output for the controller is calculated according to Equation 3.3. This output is then fed back into the system as a velocity applied to the vehicle. The change in velocity causes a difference in the detected position, which serves as the next input for the controller, creating a closed control loop.

$$u(t) = K_p e(t) + K_i \int e(t)dt + K_d \frac{de(t)}{dt} \tag{3.3}$$

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 error value is calculated as the difference between the setpoint $r(t)$ and the process variable or input to the controller, $y(t)$:

$$e(t) = r(t) - y(t) \tag{3.4}$$

The output obtained from the controller is composed of three individual components. The proportional component, characterized by the proportional gain K_P, acts proportionally to the current value of the error. On a P-only controller, there is often a remaining control deviation after an equilibrium is reached which causes the set point not to be reached exactly. The integral component, characterized by the integral gain K_I, accounts for past error values, accumulating them over time to eliminate the residual error. The derivative component, characterized by the derivative gain K_D, attempts to estimate the future trend of the error by reacting to its rate of change, increasing or dampening the control as the error accelerates or decelerates. Each mentioned gain must be particularized to obtain an effective PID controller for the system under study.

While a PID controller is defined by three control terms, certain scenarios adapt better to using only one or two of these terms to achieve precise control. In such instances, the unused parameters are effectively set to zero, resulting in what is commonly referred to as a PI, PD, P, or I controller, depending on the specific control actions involved. PI controllers are often employed when the derivative action might be susceptible to measurement noise. Nonetheless, the integral term remains crucial in many instances, as it is pivotal in guiding the system towards its intended target value. The process of deciding the control parameters, or tuning, for this particular application is described in Section 4.2.

The velocities that the controllers output are sent to the pilot module to be communicated to the autopilot through MAVLink messages. The autopilot then uses its own internal controllers to translate these velocities into signals for the propeller motors to run at a specific speed and maintain a smooth flight. This intermediate control process creates a discrepancy between the velocities sent from the DroneVisionControl application and those measured by the vehicle's telemetry and will have to be considered when the control parameters are determined. In practice, it will result in the possibility of implementing more aggressive behaviour on the controllers, as the more complex internal autopilot controllers will work to smooth the final trajectory and avoid sudden movements.

The PID controllers in this project are implemented with the help of the open source `simple-pid` Python library [49], 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 internal `controller` module[13] the one that interacts with the external `simple-pid` library to feed and receive the correct values to the PID controllers calculated from the bounding box around the detected person.

PID tools

An additional utility called `tune_controller`[14] has been developed for tuning and testing the response of the PID controllers. 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 step 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. Each coefficient can be tested individually by providing fixed values for the other two parameters.

3.5.2 Safety measures

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. The software-based safety features used in this project can be divided into those offered by the external PX4 autopilot and those developed as part of the DroneVisionControl application.

The PX4 autopilot offers various safety configuration options, known as failsafes, documented in *Safety Configuration (Failsafes) | PX4 User Guide* [50]. These failsafes detect undesired conditions during flight and include detecting a lost connection to the

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

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

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. The failsafes' failure conditions and actions triggered in response can be configured through QGroundControl, the PX4 console or an offboard API (MAVSDK).

Additionally, operator control can play a vital role in ensuring safety during flight by leveraging QGroundControl to map PX4 commands to switches in an RC controller. This allows the operator to use an RC controller with a configured switch to deactivate Offboard mode at any time during a flight controlled by the follow solution. This action causes the vehicle to disregard instructions from the companion computer and assume control through GPS-assisted or manual flight modes. A secondary switch can also 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.

On the DroneVisionControl side, the computer vision module that interacts with the external pose detection library incorporates several checks to validate that the received results match specific features associated with a standing human figure. Any detections that do not pass these tests are rejected as invalid as described in 3.5. If the detection output is invalid or empty (nothing detected in the image) is received, the vehicle immediately enters Hold flight mode. In this mode, any velocity commands are discarded, and the drone hovers in its current position. When a valid person is detected again, the follow mechanism resumes controlling the vehicle's velocity. In addition to detection validation, the application incorporates output limits for the PID controllers and the pilot module, which limit the maximum possible velocity of the vehicle at all times during the execution.

By integrating the safety features of PX4 with those developed for this project, 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 integrity of the different components in the simulated environment. Initially, there is a component testing section where each part is tested individually. Afterwards comes the integration testing, where the SITL simulation of the PX4 flight controller is integrated with the AirSim environment and the DroneVisionControl offboard application to validate the exchange of control commands and images for the detection software.

In the second section, the simulation tools are employed to tune the PID controllers that drive the behaviour of the follow control solution. The purpose of the tuning process is to find good enough values for the control parameters to be able to validate the vision control solution and determine the limits of the simulation tools. During the process, first, the response of each of the two controllers will be analysed individually to select the appropriate gains. Afterwards, the controllers will be integrated together by applying the chosen values to the follow solution to ensure that they work in a satisfactory manner.

In the third section, once the software is proven to adhere to the established safety requirements, the simulation is moved to the HITL mode, where the software runs on dedicated hardware instead of on the simulation computer. This hardware includes the autopilot board Pixhawk 4 and the Raspberry Pi as its companion computer. The main goal is to verify that all the communication channels function 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

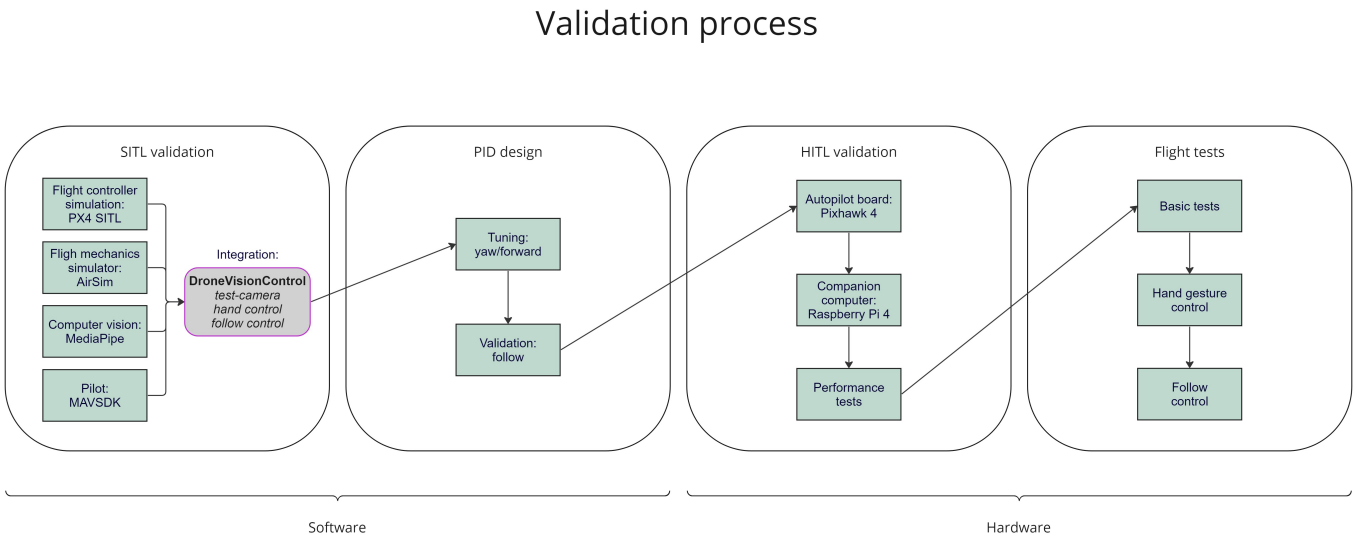


Figure 4.1: Summary of the validation process followed in this chapter.

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.

4.1 PX4 SITL simulation and validation

The software-in-the-loop simulation mode developed by PX4 is described in Section 3.1. The advantage of this simulation method is that it enables testing and validating the correct operation of individual software components of the program's architecture and their correct integration into one control flow before adding further complexity with the dedicated hardware.

4.1.1 Basic functionality tests with Gazebo

To facilitate starting the tests with as few components as possible at a time and reduce the amount of configuration needed, the initial validations will be run with Gazebo [51] as the flight mechanics simulator. This simulator comes by default with the PX4's SITL installation. Gazebo works natively on Linux, so it can run in parallel with the simulated flight stack in SITL mode and the project's software on the same machine without having to be concerned about the networking between Windows and WSL. To set up a Linux machine for these tests, PX4 and DroneVisionControl need to be installed as detailed in

Appendix A.1.

The initial tests conducted in this section will ensure that the foundational features of the control algorithms, sending commands to the autopilot and retrieving images for analysis, are reliable. To do that, the sequence of steps will be as follows:

1. Verify SITL simulation. The simulated flight controller (PX4) and the 3D program (Gazebo) connect to each other. Commands on the PX4 console are visible in Gazebo.
2. Verify pilot module. DroneVisionControl connects to PX4 through the MAVSDK library, and commands are received by the flight controller.
3. Verify video source module (`CameraSource`). Images from a camera are read into the program and displayed.
4. Verify image detection by MediaPipe Hand. Target landmarks are identified on test images by the computer vision detection utility.

Verify SITL simulation

The expected result is that building and starting the PX4 flight stack in a console also starts the Gazebo program, where the drone model is visible and can be controlled by the flight controller executing on the console. This console can be used to send commands to the vehicle and set configuration parameters for the simulation.

Once the required software is installed, the simulation can be started with the `make px4_sitl gazebo` command or using the `simulator.sh` script found on the project repository[1]. 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 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`.

Verify pilot module

The second test will focus on the pilot module to verify whether the DroneVisionControl application can connect to the simulation and send flight commands. The expected result is that the connection is established successfully, and the keyboard inputs sent to the

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

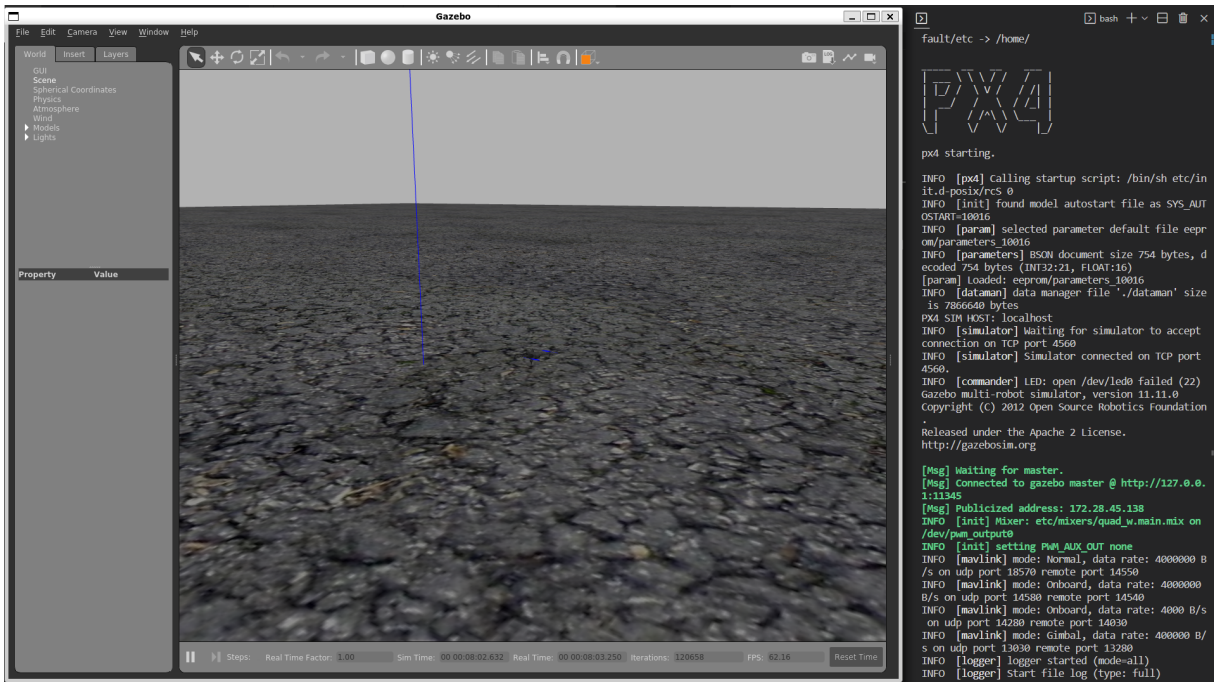


Figure 4.2: Gazebo simulator (left) and output from the PX4 console (right) after PX4's software-in-the-loop simulation is started.

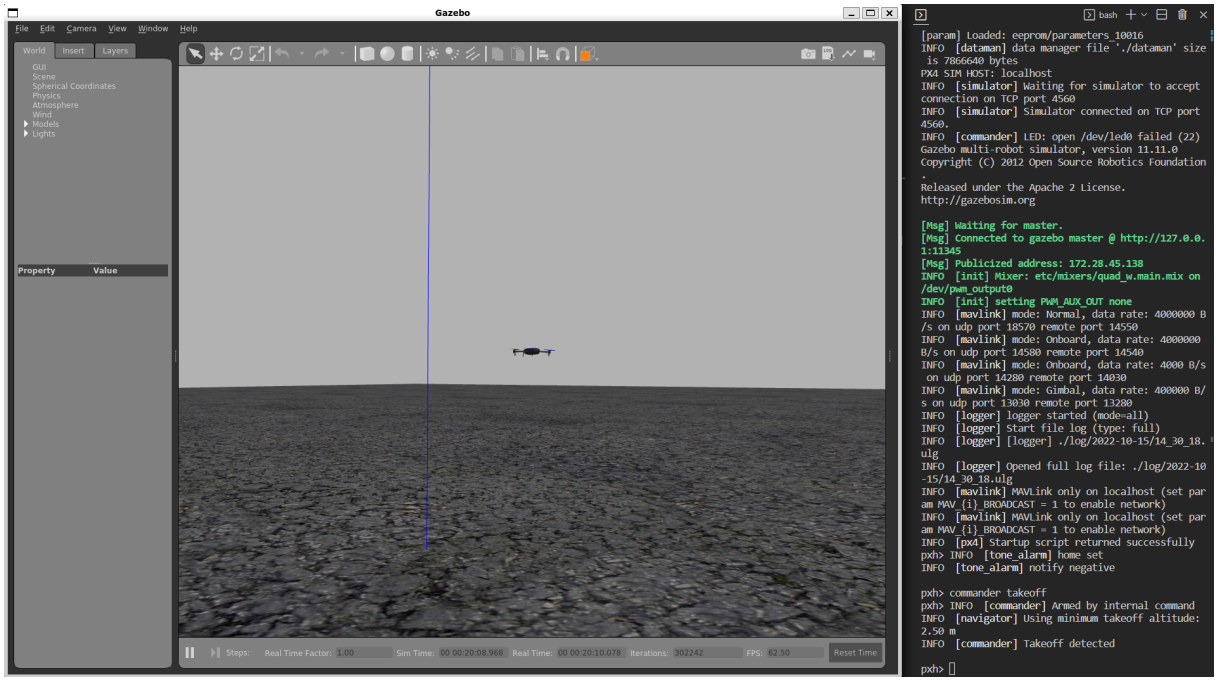


Figure 4.3: Gazebo simulator (left) and output from the PX4 console (right) after the takeoff command has been executed.

application are transformed into commands. These commands are interpreted by PX4 and shown visually in the 3D simulation in Gazebo.

The `test-camera` utility described in Section 3.3.4 has been developed specifically to test different modules without engaging any of the program's control mechanisms. It can be started through the tools section of the application's command-line interface, using the `--sim` option to specify that the test target is the connection to the simulator (`dronevisioncontrol tools test-camera --sim`). Once the connection to the simulation is established successfully, movement commands can be sent to the vehicle through keyboard input. To make this work, the program reads the input and maps it to a command in the pilot module, which is then queued until any previous commands are finished. When it is time to execute the command, the pilot module communicates it to the connected vehicle through the MAVSDK library. In the PX4 console, the logs should show that the command has been received before the vehicle model in Gazebo shows the effect visually.

For example, pressing the "T" key in the console executing the DroneVisionControl application will trigger takeoff. The result should be the same as sending the `commander takeoff` command through the PX4 console. This verifies that the MAVSDK library and the pilot module work as expected.

Verify video source module - `CameraSource`

The goal of this test is to verify that the video source module can retrieve and operate on images taken from a camera connected to the computer. This feature can be tested directly on a standalone Linux OS. However, if the PX4 simulation is running in WSL, as will be needed later for the complete simulation environment, it is necessary to change the configuration. The DroneVisionControl application must be executed from the Windows system instead, as WSL cannot access hardware devices or USB ports on the host computer. 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 used before can be run with the `--camera` option to retrieve images from a connected camera. The expected result is that the program starts and a GUI window is drawn on the screen showing the live images taken from the connected camera. It is possible to save the images from the camera for later analysis in either photo or video format using the spacebar in the keyboard as the trigger. The '<' key changes the capture mode between photo and video.

Verify image detection by MediaPipe Hand

For the last test in this section, the goal is to verify the effectiveness of the image detection mechanisms on the images taken by the camera in the previous step. In this case, the `test-camera` tool can be used with the `-f/--file` option to use a video saved in the computer as the source for the `video-source` module. Additionally, the `-h/--hand-detection` option can be used to run the hand-detection algorithm provided by the MediaPipe library on the source images. The expected result of running these commands is that the application starts, and a window is displayed with the recorded video, with the landmarks detected in the image drawn over the joints of the hand in the correct positions.

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.

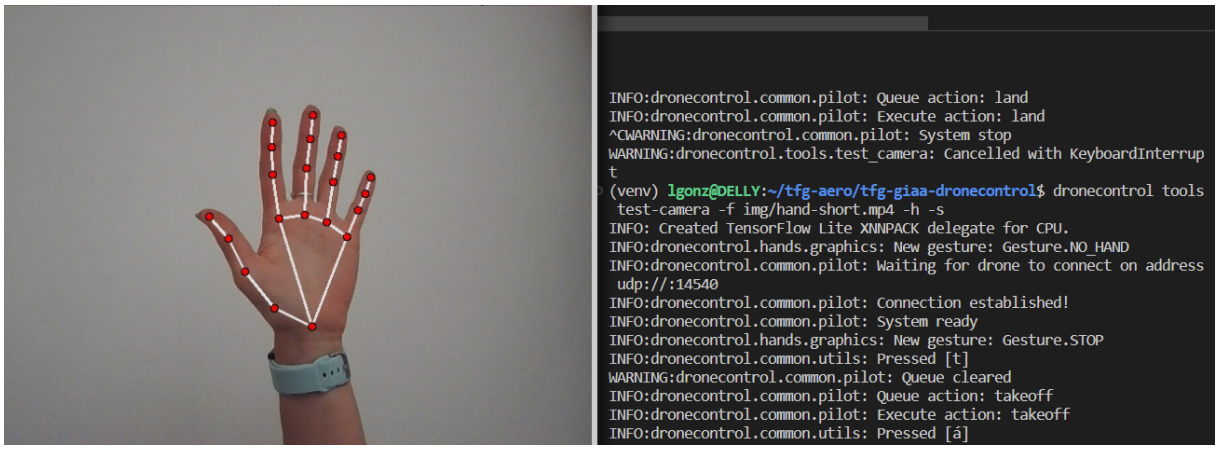


Figure 4.4: Hand detection algorithm running on images taken from the computer's integrated webcam.

After testing the flight stack, the default simulator and the pilot and image modules of the developed application, it is time to add the AirSim simulator to the environment.

4.1.2 System integration tests 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 inside Unreal Engine,

interacting with PX4 as the default Gazebo simulator did. Additionally, all the necessary detection, tracking and following features should work as expected.

To execute the subsequent tests, the AirSim simulator must be installed in the Windows host. Meanwhile, the PX4 flight controller and the DroneVisionControl application will run in a WSL subsystem as described in Figure 3.4. The complete installation process for this setup 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 (the virtual WSL Ethernet adapter). This parameter can be found in Appendix A.3, along with the complete configuration file used in AirSim for SITL testing. On the PX4 side, the flight controller must also be made aware of the network interface to listen to the simulator and started in a specific mode that sets it up to respond to AirSim's attempt to connect. Both of these points are taken care of behind the scenes when starting the PX4 console with the provided `simulator.sh`[2] script with the `--airsim` option.

After the installation is complete, the necessary characteristics will be validated in the order below:

1. Verify SITL simulation in Airsim. The simulator can start, connect to the PX4 SITL through the WSL virtual network and receive commands from the PX4 console.
2. Verify integration with hand solution. The individual modules tested before are integrated together by running the hand-gesture control solution described in Section 3.4.
3. Verify video source module (`SimulatorSource`). Images from the virtual camera in the simulator are read into DroneVisionControl and display AirSim's simulated world.
4. Verify image detection by MediaPipe Pose. Target landmarks are identified on the 3D-model of a person in the simulator by the computer vision detection utility.
5. Verify integration with follow solution. The follow solution can control the vehicle's velocity directly in PX4's offboard mode, reacting to the position of a detected person.

Verify SITL simulation in Airsim

The objective of this test is to confirm that the Unreal Engine test environment containing the AirSim simulator can start and that the PX4 flight controller running in the console finds it and connects to it successfully. The expected result is that, after first pressing play

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

on Unreal and then starting PX4 with the `simulator.sh --airsim` command, the drone model in AirSim can be controlled from the PX4 console in the same manner as with the Gazebo simulator.

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.

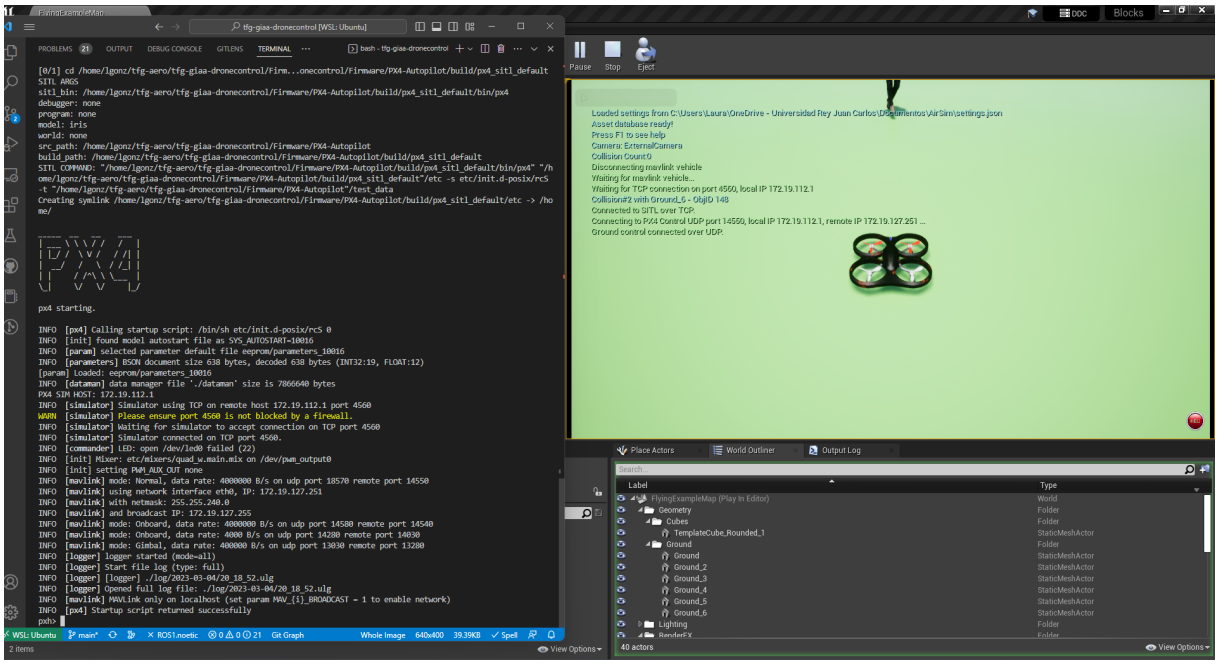


Figure 4.5: AirSim environment (right) connected to the PX4 console (left).

Verify integration with hand solution

In the second test, the goal is to integrate the individual modules tested in the previous steps: the pilot, the external camera and the hand recognition software. This will be achieved through the proof-of-concept hand control solution by running the DroneVisionControl application with the gesture-based control loop enabled. This mechanism is started with the `dronecontrol hand` command. The expected result after the command is executed is that the DroneVisionControl application connects to the PX4 console, and a window is displayed with the output from the external camera. When a hand is shown to the camera, the detection software draws the landmarks over the image. At that moment, the developed control software will attempt to interpret the gesture signalled with the hand and map it to its corresponding command for the vehicle, according to the list in Section 3.4. The complete execution is shown in the video[3] accessible

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

through this link. Additionally, a frame extracted from the video can be observed in Figure 4.6.

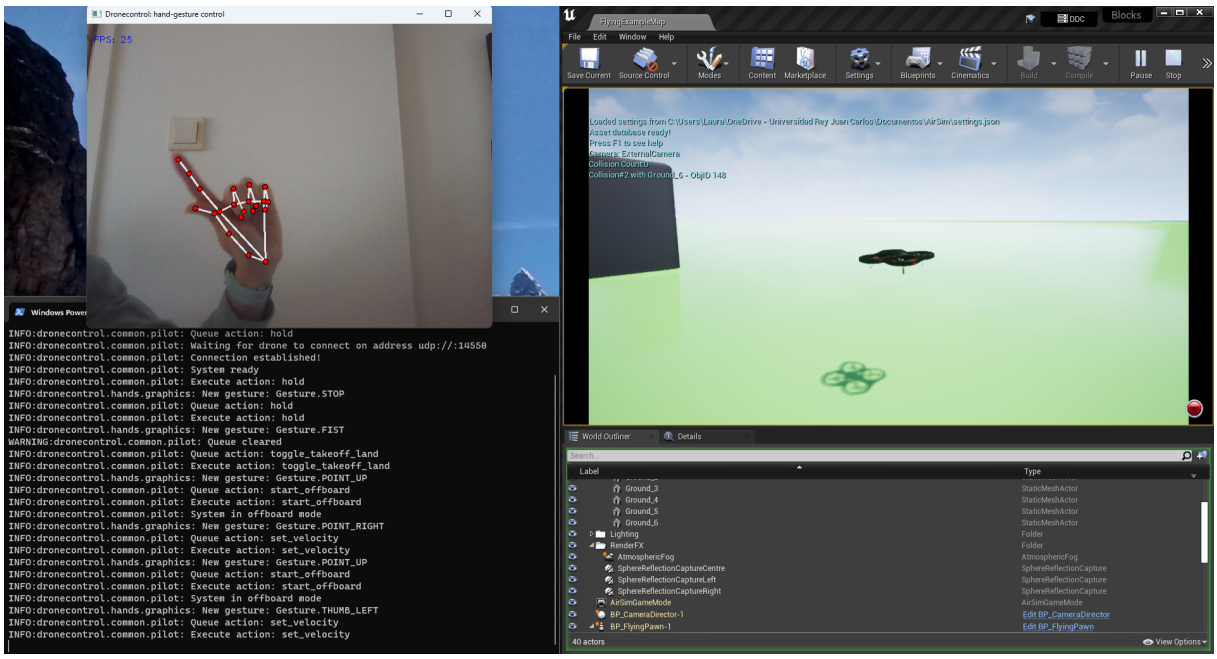


Figure 4.6: Single frame extracted from the video of the full execution of the hand-gesture control solution. Gesture detection is shown on the upper left side of the screen. On the lower left, the console shows the DroneVisionControl output logged during the mapping between detected gestures and flight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

Verify simulator video source and pose detection

The validation process will now focus on the tools required to execute the pose detection and tracking mechanism. The aim is to validate that the video source module can read images from AirSim's virtual camera. These images should be displayed by the DroneVisionControl application and show the person model from the vehicle's perspective so that it can be identified by the MediaPipe Pose detection mechanism. All the landmarks detected in the image should be present and match the correct position in the figure, and a valid bounding box should be visible around it.

To assess the detection and tracking of human figures from images captured within the simulator, the camera testing tool provided with DroneVisionControl can be used again. This time, the `-p/--pose-detection` option can be added to run the pose-detection algorithm provided by the MediaPipe library on the source images.

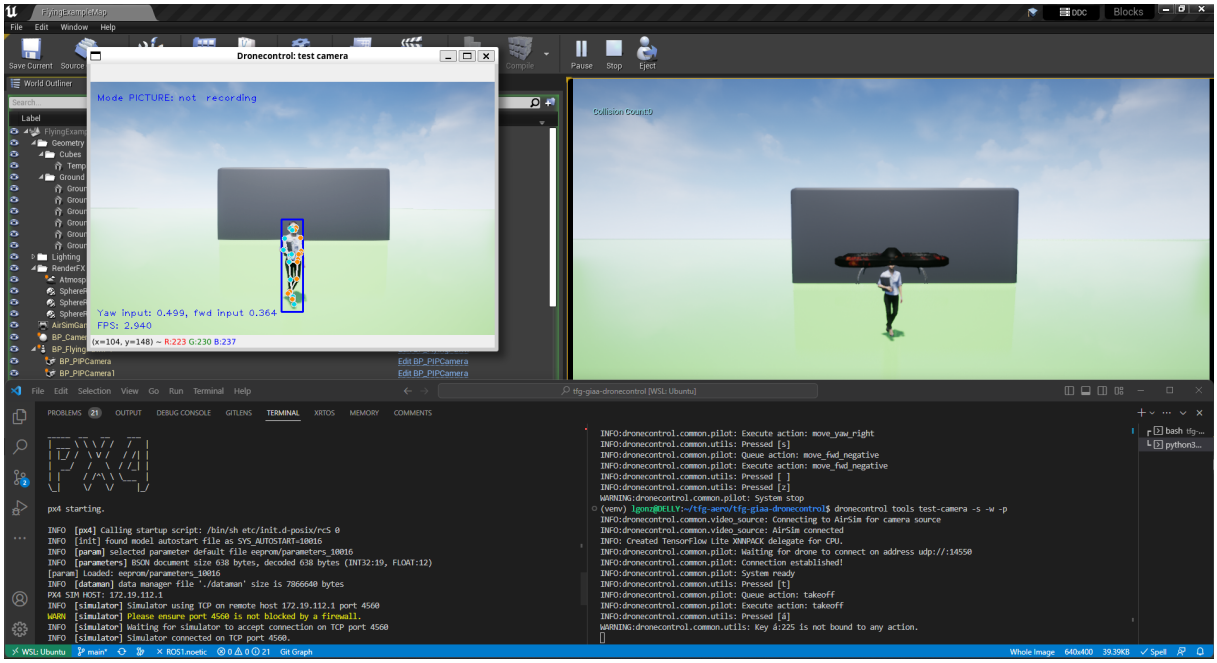


Figure 4.7: AirSim (top half, behind), PX4 (bottom left) and DroneVisionControl (console, bottom right; image window, top left, over AirSim) applications running side-by-side with image retrieval and pose detection working as expected.

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 draws an overlay in the image window with the two calculated positions that will be used as the input for the PID controllers: the normalized x-coordinate of the midpoint within the bounding box (yaw controller) and the percentage of the image height covered by the height of the bounding box (forward controller), as shown in Figure 3.21. Unlike the yaw controller, whose setpoint is set to match the image's centre, the setpoint for the forward controller can be set higher or lower depending on the desired distance for the drone to follow the person when the control mode is engaged. The value of the forward input displayed by the program can be employed to decide on an adequate setpoint by moving the person to the desired distance in the simulator and noting down the measured input detected at that relative position.

Verify integration with follow solution

The last step that will be verified in this section is the control module for the follow solution. The objective is to validate that the AirSim simulator, the pose detection mechanisms and the pilot module can work together to direct the drone's movement based on the position of the detected person. Since the control parameters for the PID controllers directing the movement have not been tuned to obtain adequate values yet, the controllers can be set to run with only the proportional term enabled with a suitably low gain so that the vehicle describes a slow and smooth movement towards the target. This step is essential to confirm that the controllers can react to changes in the figure's position before the simulation environment is used to tune the controllers to the appropriate parameters.

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)
```

The results show that the vehicle starts moving forward when the person moves backwards and turns to the right when the person moves to the right, mirroring the movement when the person moves in the opposite direction.

4.2 PID controller design

To implement the person-following mechanism effectively, a pair of PID controllers is employed to derive velocity outputs from position data obtained through image detection. These controllers are defined by their control parameters: the proportional (K_D), integral (K_I) and derivative (K_D) gains, as shown in Equation 3.3. To optimize their performance, it is necessary to fine-tune the controllers by selecting appropriate values for these parameters. These values will be determined through empirical experimentation, acknowledging that obtaining theoretical optimal values through this method, as discussed in 3.5.1, is not feasible. Nevertheless, it remains the most straightforward approach to achieve a satisfactory enough performance from the closed-loop system to validate the follow control solution. For each parameter, several values will be explored with the aim of striking a balance between a more aggressive controller (characterized by larger gains) for faster control response and a more robust controller (with smaller gains).

The procedure will be as follows. First, the sign of the process gain is selected. For the controller to operate as expected, a positive output should result in an increase in the input. If the system works in the opposite way, the feedback will lead to unstable behaviour where

the process variable grows exponentially. This behaviour can be fixed by inverting the sign of the output of the controller before it is fed back into the process.

The second step will focus on selecting the control parameters mentioned before. In the initial phase, the controller operates exclusively as a pure P-controller, with both the I-portion and D-portion deactivated, thereby isolating the proportional gain (K_P). To find an optimal value for this parameter, different values of K_P are tested systematically. Starting with a relatively low gain to ensure gradual changes in the process variable, the values are progressively increased until a distinct overshoot is observed, which should quickly diminish without noticeable oscillations.

Utilizing a P-only controller often leaves a residual control deviation, preventing the setpoint from being precisely reached. To address this residual error, it becomes necessary to introduce an integral gain that gradually compensates for the error over time. During this phase, the proportional gain remains set at the previously determined value, while the integral gain is gradually increased until the control deviation is adequately mitigated. Once again, an overly aggressive setting should be avoided to prevent undesirable oscillations.

A derivative gain can be incorporated to dampen the initial overshoot in the process variable to enhance the controller's performance further. This is accomplished by following a similar incremental approach as before. A suitable starting point for the derivative gain is typically around one-tenth of the integral gain [52].

The testing environment

The outlined tuning method will be applied separately to the yaw and forward controllers, with flight control enabled exclusively in one direction at a time. For this purpose, the custom tuning tool developed for this project and detailed in Section 3.5.1 will be employed to expedite the testing of various controller parameter values and their subsequent comparison. This tool allows designating which controller (yaw or forward) is enabled for testing and which parameter values will be subject to iteration. In each test, two of the parameters will be set with fixed values, while the third parameter is set sequentially to different values. For each of the values in the sequence, the new tunings are applied to the controller and its step response is plotted.

To capture the step response of the controller under focus, a deliberate offset is introduced between the vehicle and the person model within the simulation environment. This offset positions them away from the reference position at which the controller's input aligns with its setpoint. In the coordinate system of the simulation environment (AirSim), with the vehicle located at $x = 0, y = 0$ on the ground plane, the reference position for the person model is $x = 420, y = 0$. Shifting the person model from that position along

either the y-axis or the x-axis will trigger a step response in the yaw or forward controller, respectively. Figure 4.8 shows the reference position for the controllers.

At the end of this process, the outcomes are visualized through a series of graphs generated by the program. These graphs offer insights into the controller's input and output over time and the actual changes in position and velocity recorded by the autopilot telemetry during the test. The specific telemetry variables displayed vary according to the particular controller under analysis. The graphs associated with the yaw controller show the heading and yaw speed, while those related to the forward controller show the position along the forward axis and ground speed.

Given the potential noise introduced by the image detection mechanisms, it is often more advantageous to fine-tune the controllers by examining the changes in the sensor-measured positions rather than the ones detected by computer vision mechanisms. This approach is preferred as the internal autopilot controller aids in smoothing out the resulting curves, making it easier to see trends and oscillations.

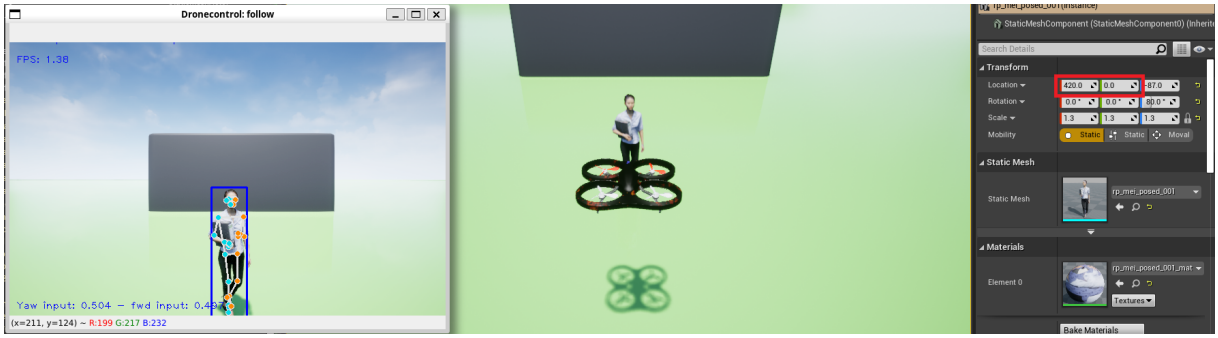


Figure 4.8: Reference position for the yaw and forward PID controllers. From left to right, the panels show the DroneVisionControl application window, the AirSim simulator world view and the world location of the human model in the simulator. The distance between the vehicle and the person is 420 cm in the x direction and 0 in the y direction.

4.2.1 Yaw controller

The initial focus of the tuning process is on the yaw controller. As mentioned earlier, the first step entails determining the sign of the process gain. In this case, the process variable, or input to the controller, corresponds to the normalized position of the detected person in the horizontal axis of the camera's field of view. Here, a value of 0 denotes the person's location at the left edge of the field of view, while a value of 1 signifies their position at the right edge. When the controller generates a positive output, it results in a positive yaw velocity, causing the vehicle to rotate in a rightward direction. In response, the person within the camera's field of view shifts to the left, reducing the input to the controller. Given that positive outputs should lead to increasing inputs, the sign of the output velocity needs

to be inverted to prevent undesired exponential growth.

A starting position must be established following the sign's determination and preceding any parameter testing. This starting position introduces an offset for the target person model relative to the reference position illustrated in Figure 4.8, provoking a step response within the controller. This offset will be 100 cm along the y-axis for the yaw controller. Given that the vehicle is positioned at the origin in the simulation environment, the person model should be situated at coordinates $x = 420, y = 100$ before starting the tuning process, as shown in the rightmost side of Figure 4.9. In the leftmost panel of Figure 4.9, the DroneVisionControl application displays an input of 0.64 for the yaw controller at this position. The controller will, therefore, calculate an error of $e(t) = 0.5 - 0.62$ (setpoint minus input) at the start of the test.

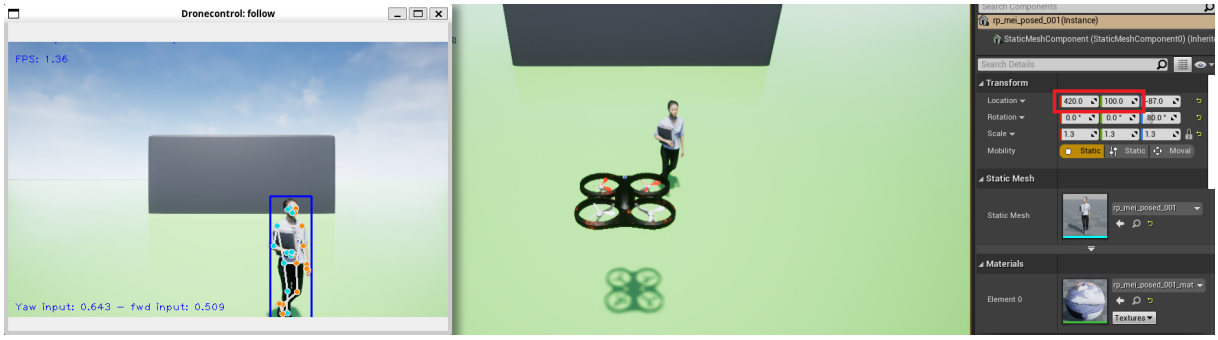


Figure 4.9: Starting position of the simulator for tuning the yaw controller. The human model is situated 420 cm forward and 100 cm to the right of the vehicle model.

Proportional component

The proportional gain is the first parameter that needs to be tuned. The values chosen to test for K_P range from 25 to 150 in steps of 25. To have a P-only controller during the test, the K_I and K_D components are set to 0. The results of the test are shown in Figures 4.10 and 4.11. The yaw controller's input and output are plotted over time in the first figure (4.10). On the left side, the input is represented by the calculated error (setpoint minus input position), and on the right side, the output is the velocity sent to the flight controller in degrees per second. The second figure (4.11) depicts the corresponding telemetry measurements during the test, with the left graph showing the heading in degrees retrieved from the flight controller and the right graph showing the yaw velocity in degrees per second. The effects of the internal PX4 control mechanisms can be appreciated when comparing these two figures. Thanks to them, the noise introduced by the image detection features is not transmitted to the vehicle's trajectory but smoothed out.

Following the tuning process, the value selected should present a small overshoot initially and then smooth out without undue oscillation. In Figure 4.11a, the values

between 75 and 125 present the initial overshoot, and from $K_P = 125$ the oscillations start to be more noticeable. Therefore, the value selected for the proportional gain will be $K_P = 100$.

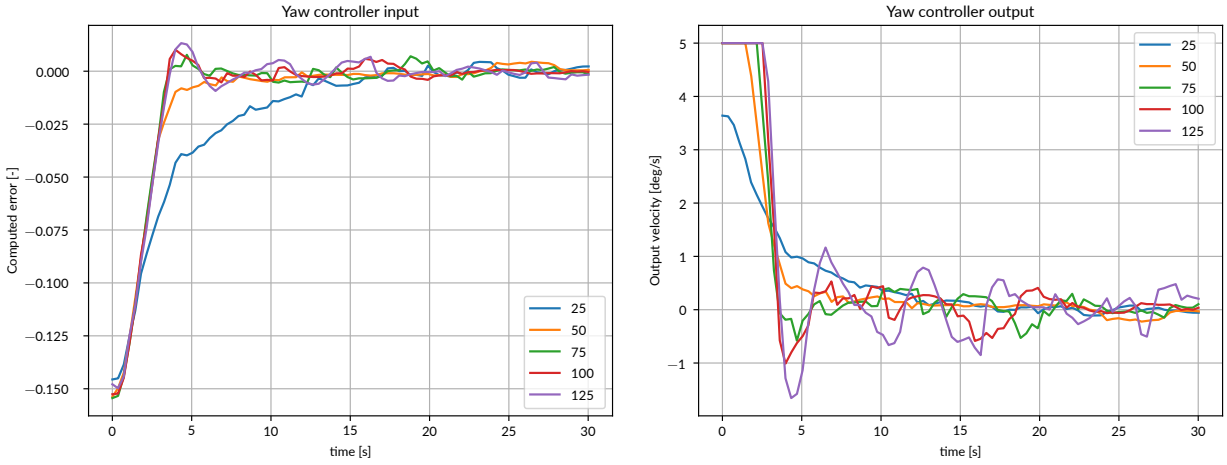


Figure 4.10: Variation of (a) computed error and (b) output velocity for different values of K_P and $K_I = 0$, $K_D = 0$ while the yaw controller is engaged.

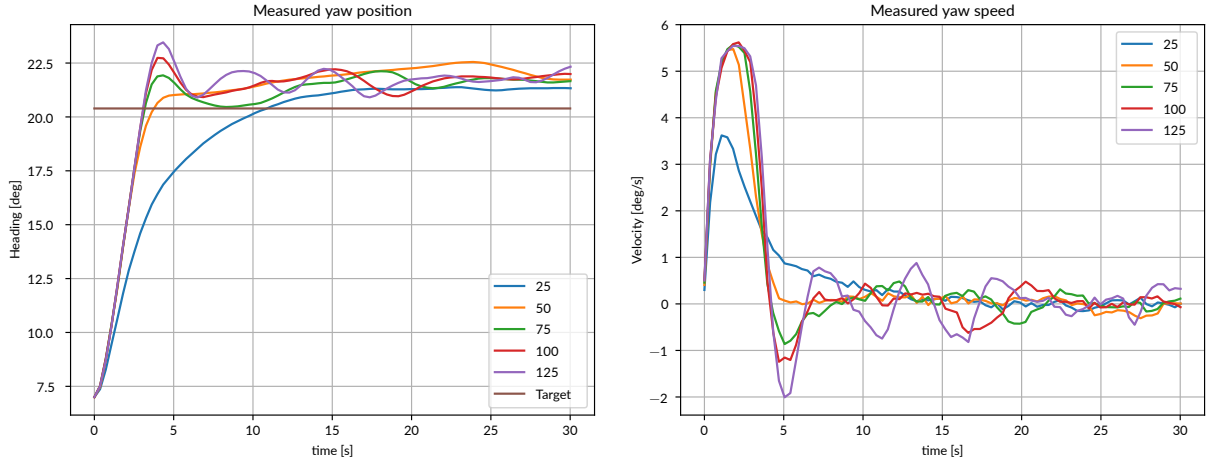


Figure 4.11: Variation of (a) measured yaw heading and (b) measured yaw velocity for different values of K_P and $K_I = 0$, $K_D = 0$ while the yaw controller is engaged.

Integral component

The measured positions from the P-only controller (Figure C.4a) show a remaining deviation from the target heading after the controller reaches an equilibrium. This common occurrence in P-controllers is solved by adding an integral component. The gain for this component can be decided in the same manner as the proportional part. In this case, the

K_P value will be set to 100, as decided in the previous section, and the K_D will remain at 0. Thus, the controller is being tested as a PI-controller. The values chosen for K_I for the test range from 0 to 50 in steps of 10. The results for the computed error and the measured vehicle heading obtained by running the tuning tool with those values can be seen in Figure 4.12. The additional velocity graphs produced by the tool can be consulted in Appendix C.2. Looking at Figure 4.12, it can be noted that for K_I of 30 or more, the final heading ends up very close to the target, eliminating the remaining deviation detected in the previous section. From these values, the selected one will be $K_I = 40$, as it offers the best balance between reaching the target as fast as possible and not showing undesirable oscillation after the initial overshoot.

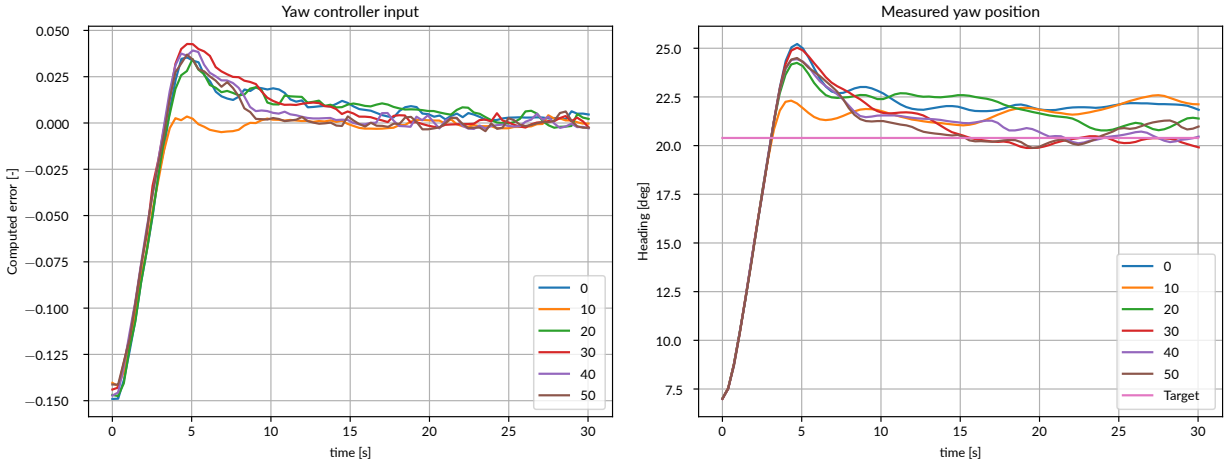


Figure 4.12: Variation of (a) computed error and (b) measured yaw heading for different values of K_I and $K_P = 100$, $K_D = 0$ while the yaw controller is engaged.

Derivative component

The last component in a PID controller is the derivative part. This component can help improve the performance by dampening oscillations in the controller output, thus allowing the selection of a higher K_P that would otherwise be possible. However, noise in the input signal to the controller (like the one generated by the image detection mechanisms) is known to degrade the derivative action since spikes in the input cause significant contributions from the derivative part [53]. To observe the effect of this component, the tuning tool has been run with a fixed value of $K_P = 100$ and $K_I = 40$. The results for the computed error and the measured vehicle heading can be seen in Figure 4.13. The additional velocity graphs produced by the tool can be consulted in Appendix C.2. For all of the tested values between 5 and 80, the performance of the controller clearly decreases, creating more oscillations the higher the value of K_D. There are ways of getting around the noise limitations to take advantage of the derivative part, like adding filters to the controller's input [54]. However, for this particular application, it is enough to settle for a

PI-controller and leave out the derivative part altogether.

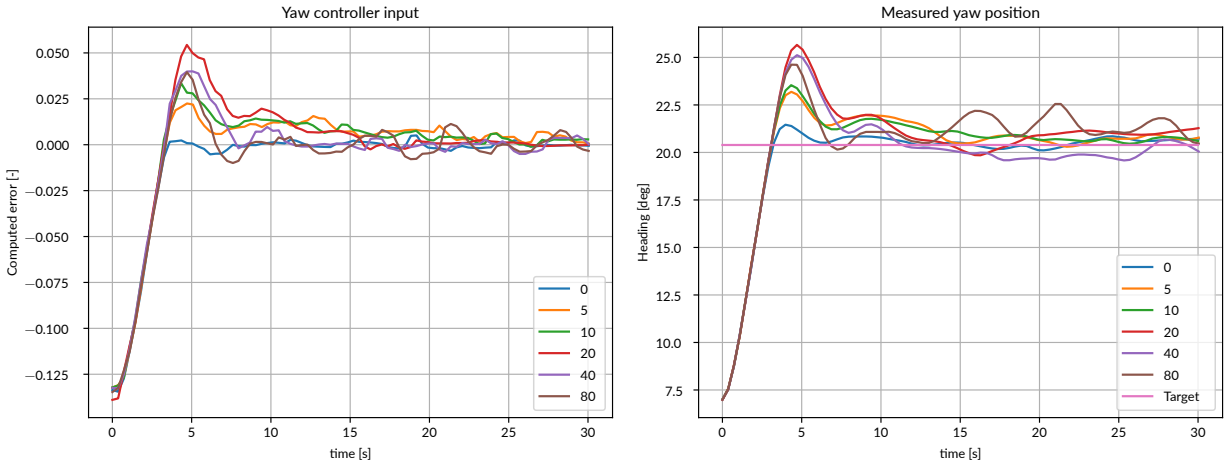


Figure 4.13: Variation of (a) computed error and (b) measured yaw heading for different values of K_D and $K_P = 100$, $K_I = 40$ while the yaw controller is engaged.

Then, the final values chosen for the yaw controller's parameters are $K_P = 100$, $K_I = 40$ and $K_D = 0$. These values constitute only a rough estimate of adequate component gains that produce a good enough controller response for the follow solution to be validated. The trial-and-error tuning method employed also showcases an additional application for the simulation environment offered by PX4 and Airsim. There exist many computational methods of higher complexity that can obtain more optimal parameters for the controllers, but they fall out of the scope of this project.

4.2.2 Forward controller

Once the yaw controller has been tuned, the focus shifts to the forward controller. The tuning process for the forward controller mirrors that of the yaw controller. When examining the process gain in this context, it becomes evident that a positive output generated by the controller induces a positive velocity in the forward direction, bringing the vehicle closer to the target person. The input variable is determined by the height of the detected person within the camera's field of view, normalized to the height of the field of view itself. This value increases as the drone approaches the target. Consequently, a positive output velocity naturally leads to an increasing input to the controller, indicating that the controller gain is already set to the correct sign.

Regarding the start position for the environment to tune the forward controller, in this case, an offset is required in terms of the distance between the person and the vehicle along the x-axis. To achieve this offset, the person model will be positioned at coordinates

$x = 320, y = 0$, while the vehicle remains at the origin within the simulated world. This chosen starting position, which sets the person and the vehicle closer together than at the reference point, will result in an initially negative velocity output as the vehicle moves away from the person. Figure 4.14 illustrates this starting position within the simulator. In the leftmost panel of Figure 4.14, the DroneVisionControl application displays an input of 0.69 for the yaw controller at this position. The controller will, therefore, calculate an error of $e(t) = 0.5 - 0.69$ (setpoint minus input) at the start of the test.

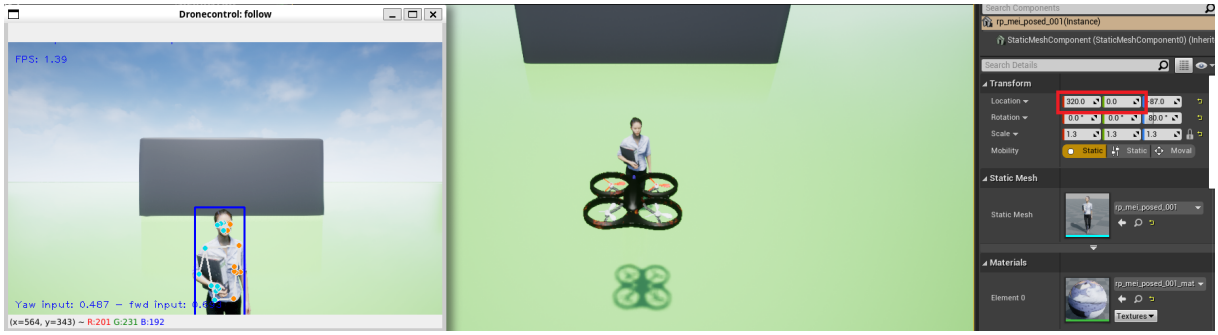


Figure 4.14: Starting position of the simulator for tuning the forward controller. The human model is situated 420 cm forward and centred from the vehicle position.

The process to select the component gains for the forward controller runs similarly to the yaw controller. The plotted outputs from the tuning utility are contained in Appendix C.2. Based on those graphs, the selected values for the parameters are $K_P = 4$, $K_I = 1$ and $K_D = 0$.

4.2.3 PID controller validation

The chosen values for the controllers can now be applied to the complete follow solution to get a clearer picture of the expected performance of the vehicle during flight tests. In this test, the follow application is started, and the drone is made to take off and switch flight mode to offboard control. After the person is detected by the pose mechanisms, the 3D-model is moved around in the simulated world to observe how the vehicle reacts. The expected result is that the movement is appreciatively more responsive than on the follow test run with the non-tuned P-only controller at the end of Section 4.1.2 (*Verify integration with follow solution*). To provide a visual demonstration, a video showcasing the test described can be accessed here[4]. Additionally, Figure 4.15 displays a frame extracted from the video, giving a glimpse of the drone's behaviour during the follow operation.

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

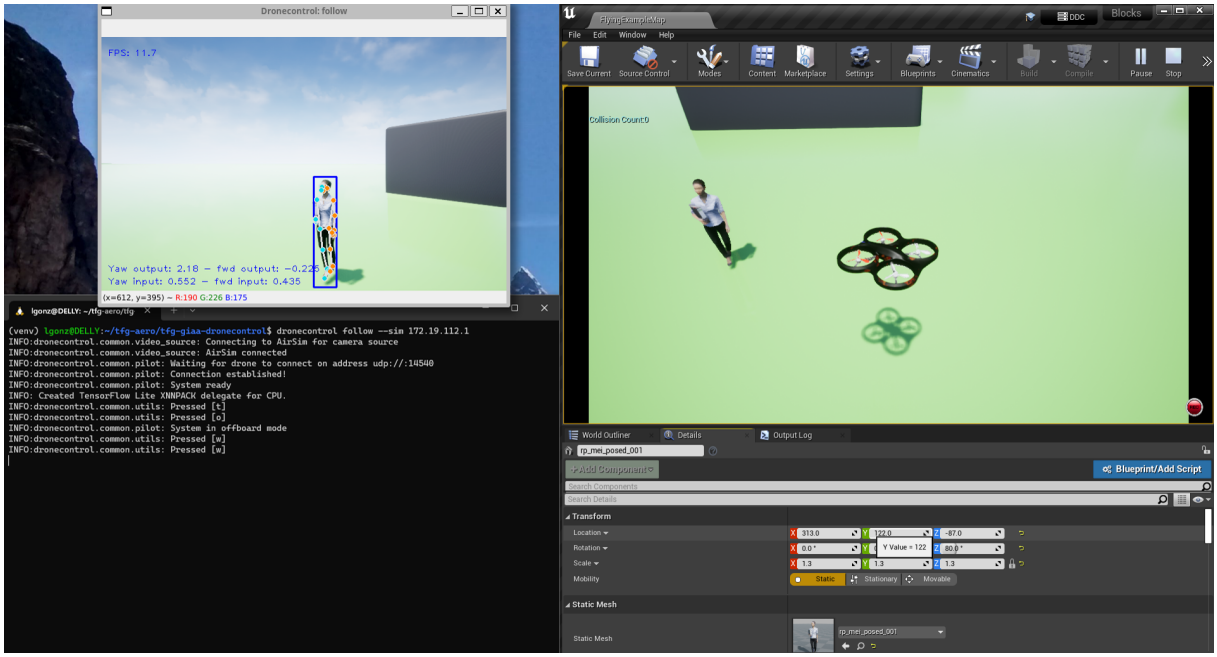


Figure 4.15: Single frame from the video showing the movement of the drone in response to changes in the position of the tracked person.

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 to achieve a system that can run the developed control solution, replicating the same behaviour demonstrated in the previous section. In the first part, the DroneVisionControl application will be run from the simulation computer, as outlined in Figure 3.5. In the second part, after the performance of the flight board is validated, the execution of the DroneVisionControl application will be moved to a separate companion computer, the Raspberry Pi 4. This dual approach allows testing both the offboard and onboard configurations.

4.3.1 HITL validation with simulation computer (offboard configuration)

The purpose of this section is to test the offboard configuration on the ground before any flight tests, with the PX4 flight stack running on the dedicated autopilot board and the DroneVisionControl application running on the simulation computer. To run the flight stack on the Pixhawk board without flying, it is necessary to use the HITL simulation mode, where the flight mechanics will still be simulated by AirSim. 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 modifying 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 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 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 [55]. 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.

4.3.2 HITL validation with Raspberry Pi (onboard configuration)

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 same hardware as during flight tests to allow the identification of potential hardware performance issues. The tests mimic the onboard computer configuration outlined in Figure 3.9, with the DroneVisionControl application running on an onboard companion computer.

Figure 4.16 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, the operating system of the Raspberry needs to be set up for the DroneVisionControl application. A detailed explanation of the complete installation process, along with all the necessary libraries and dependencies for the Raspberry Pi, is included in Appendix A.2. 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 [56], an open-source implementation of a Microsoft Remote Desktop Protocol server that is compatible with the Raspberry OS.

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

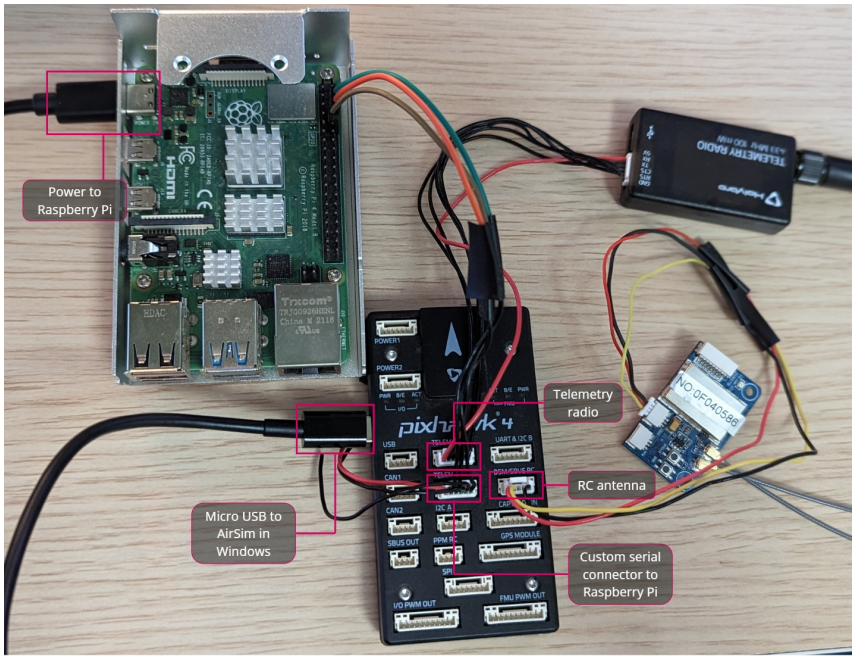


Figure 4.16: Pixhawk 4 board connected to a Raspberry Pi running the DroneVisionControl application and a Windows computer running the AirSim simulator. The setup includes a telemetry radio for QGroundControl and an RC receiver for manual control.

1. Power supply. Capacity to function when powered by a battery.
2. Serial connection. Pixhawk to Raspberry Pi and Pixhawk to the simulation computer.
3. DroneVisionControl software. Ability to run the DroneVisionControl application and its dependencies.
4. Performance of computer vision algorithms with limited processing power.

Verify power supply

The first step is to verify that the Raspberry Pi can be powered by a secondary battery connected to the board's power port by a USB-C cable instead of the standard AC power supply. Through this battery, sufficient power must be provided to the Raspberry Pi to maintain enough processing speed for the image processing software and to power the camera connected to the board. The performance differences between powering the Raspberry board with the AC supply and the battery are detailed in Section 4.3.3. At this point, it is enough to check that the `test-camera` utility can run at an acceptable frame rate with either hand or pose detection enabled on previously recorded images from the camera or simulator.

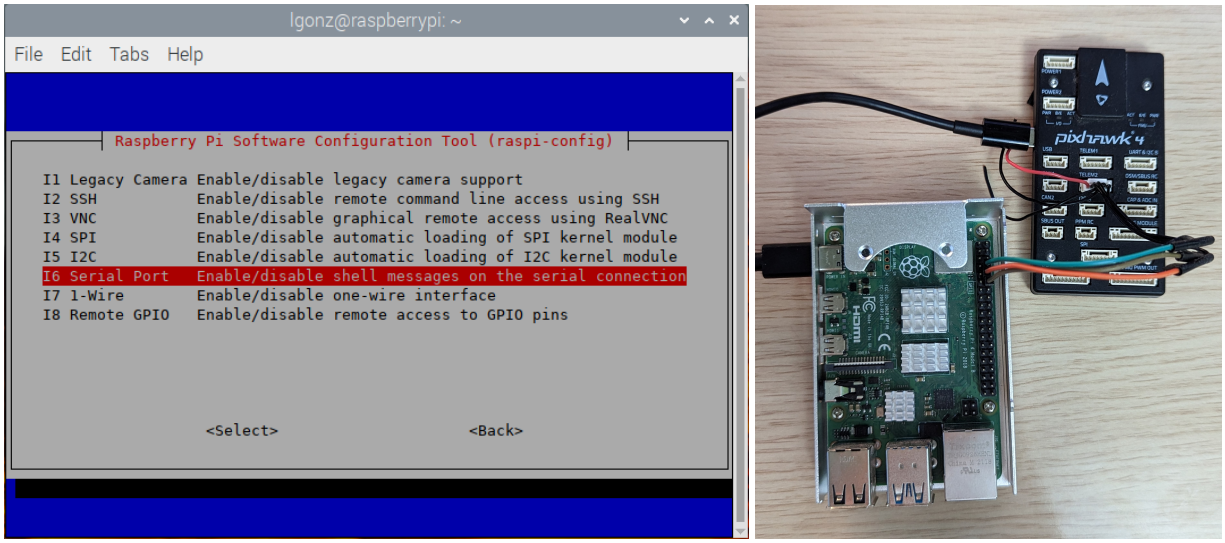


Figure 4.17: a) Picture of Raspberry's `raspi-config` and b) close-up of Pixhawk to Pi cable connection.

Verify serial connections

The most important connection that needs to be verified is the MAVLink communication channel between the flight controller and the onboard computer. The custom connector described in Table 3.1 is used for this purpose, with the `TELEM2` port on the Pixhawk board connected to the TX/RX UART pins on the Raspberry Pi's GPIO header. Additionally, configuration is required for the flight controller and the companion computer. For the PX4 flight controller, only the `TELEM1` port of the Pixhawk is configured for telemetry radio. An additional MAVLink channel must be configured in QGroundControl by setting the correct values to the parameters listed in Table 4.1.

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.

On the Raspberry Pi side, the serial port is configured by default to exchange shell messages. This needs to be disabled using the `raspi-config` command-line utility and selecting the following steps: Interface options -> Serial Port -> Disable login shell, enable serial port hardware (see Figure 4.17). After making these changes, the `/dev/serial0` address can be used to communicate with the device at the baud rate configured in QGroundControl.

The remaining connections serve to communicate the Pixhawk board and the

simulation computer. Two different channels are needed to be able to exchange information with QGroundControl and AirSim at the same time. The connection to AirSim will be implemented through the development-only micro-USB port on the Pixhawk board and the connection to QGroundControl through the telemetry radio. Due to the substantially lower throughput of the telemetry radio in comparison with the USB cable, using the telemetry radio to communicate between the Pixhawk board and AirSim will result in inferior performance since there is a more significant amount of messages transferred than between QGroundControl and the Pixhawk.

Verify DroneVisionControl software

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
```

This command connects DroneVisionControl to PX4 via the hardware address `/dev/serial0` at a baudrate of 921600. Meanwhile, the images from the simulator's virtual camera are sent to the Raspberry Pi over the local area network. The `<AirSim host IP>` parameter defines the IP the application will connect to to receive these images.

The result from the execution is shown in Figure 4.18. 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 movements of the vehicle as it responds to the instructions from the flight controller that listens to the companion computer.

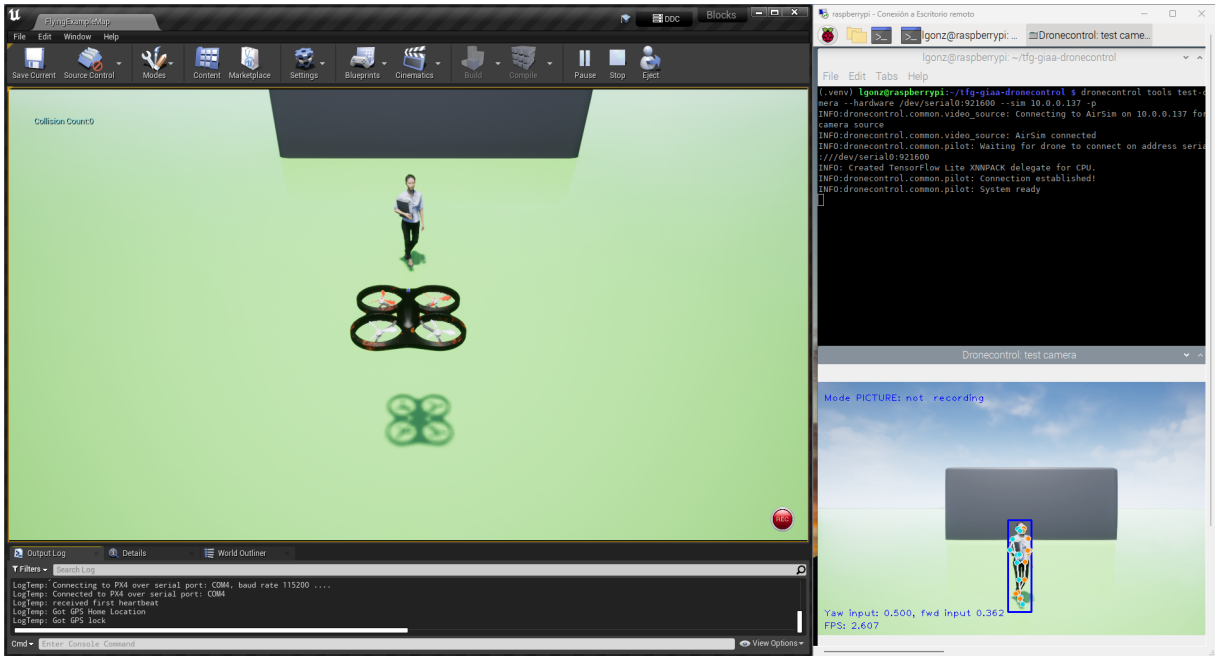


Figure 4.18: Left: AirSim simulator on Windows host. Right: RPi desktop with DroneVisionControl application and pose output.

4.3.3 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 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 (overhead from the async execution). Figure 4.19 illustrates the time allocation for each task during an average run of the follow solution with PX4 running in SITL mode and AirSim as the simulator. DroneVisionControl was also run on the simulation computer. The time consumption was obtained by measuring the time at the start and end of each task, calculating the time difference between them and averaging across every iteration of the main loop. The utility for this process can be found in the measure function inside the

application's utils module[5].

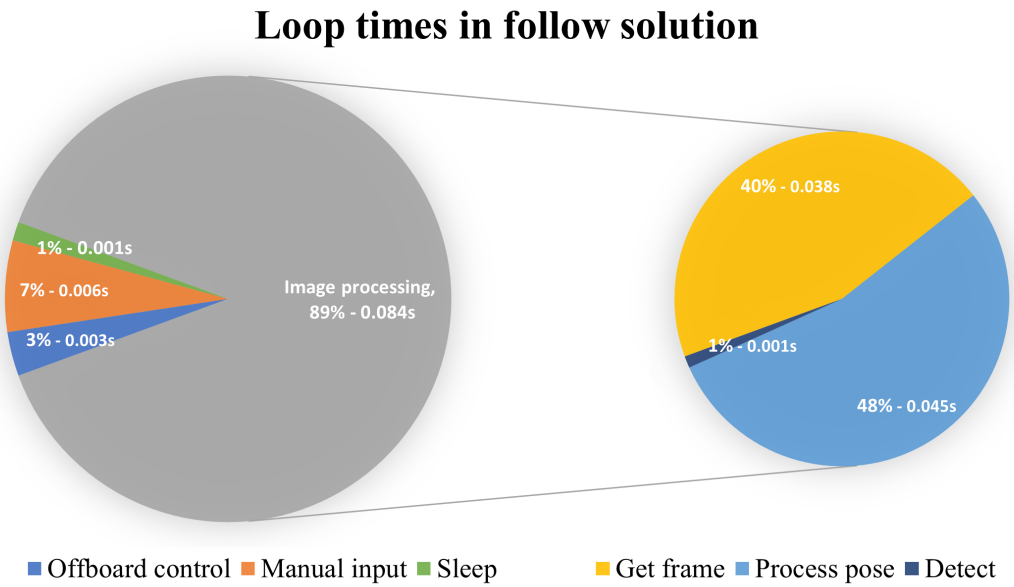


Figure 4.19: Average percentages of a loop spent by each task in the follow solution with their corresponding average absolute times in seconds (SITL + AirSim configuration).

The most time-consuming task is image processing, which accounts for approximately 89% of the total loop time. 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.

As shown in 4.19 (right), the subtasks take 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 connected to the AirSim simulator. The hardware combinations tested include:

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

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 powered by AC + 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.

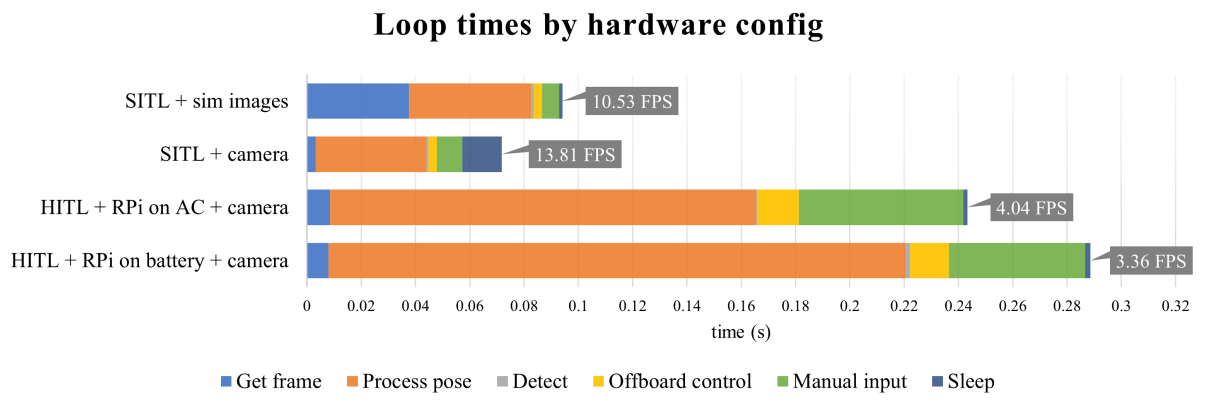


Figure 4.20: Average FPS and time spent on each task per iteration of the follow solution for the different hardware configurations.

Figure 4.20 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. This performance is satisfactory enough for this project, but there is room for improvement by, for example, optimizing the image processing algorithm or using more powerful hardware for the onboard companion computer.

4.4 Quadcopter flight tests

After validating the performance 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. The developed control solutions will be tested once the vehicle can successfully fly with the entire payload under remote control.

The initial test will run the hand-control solution to verify that the autopilot can receive flight commands from an offboard computer outside 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 guaranteed during the whole process are as follows:

1. Assemble the quadcopter with the custom payload.
2. Baseline flight with factory software. Conduct a test flight using only the remote control and factory autopilot while monitoring through QGroundControl.
3. Offboard computer flight with `test-camera` tool. Perform a flight controlled by the DroneVisionControl software running on an offboard computer (laptop).
4. Onboard computer flight with `test-camera` tool. Conduct a flight controlled by the DroneVisionControl software running on the onboard computer (Raspberry Pi).
5. Hand-gesture control with offboard computer. Control the drone with hand gestures by running the hand-gesture control solution on the offboard computer.

6. Follow control with onboard computer. The drone flies following a target person by running the follow control solution on 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[6] provides detailed instructions on how to build the vehicle using its Development Kit.

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 (see Figure 4.21). 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. During flight, the Raspberry Pi will be powered by a dedicated external battery, separate from the main battery powering the motors, which supplies power through a 2-ampere USB port. This port will be connected to the Raspberry Pi's USB-C power supply socket with a charging cable. As explained in Section 4.3.2, the power supply by this secondary battery 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 4.21.

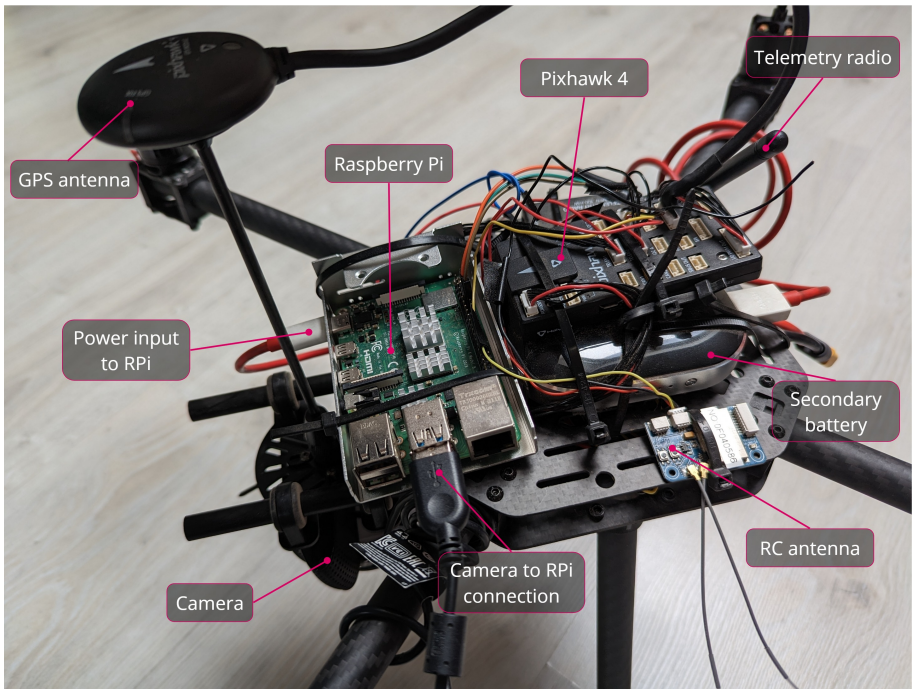


Figure 4.21: Complete build of the quadcopter with the main components highlighted.

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

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 (see Figure 4.22). 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 4.22: Underside of the vehicle, with the supports for holding the main battery and the camera in place.

Once the vehicle is constructed, additional installation and calibration steps are necessary before it can be flown. This configuration can be performed through QGroundControl by connecting the flight controller to a computer either directly via its debug micro-USB port or wirelessly via the telemetry radio. The calibration steps are outlined in the Holybro X500 build instructions and are required to ensure that all the sensors attached to the autopilot function correctly. The QGroundControl configuration screens shown in Figure 4.23 depict some of these steps. Besides the standard calibration, two additional PX4 settings must be set in the Pixhawk board for the DroneVisionControl software to work. First, the HITL simulation mode needs to be disabled in the QGroundControl safety settings and, second, the `TELEM2` port should be enabled to communicate with the Raspberry Pi, as explained in Section 3.2.2.

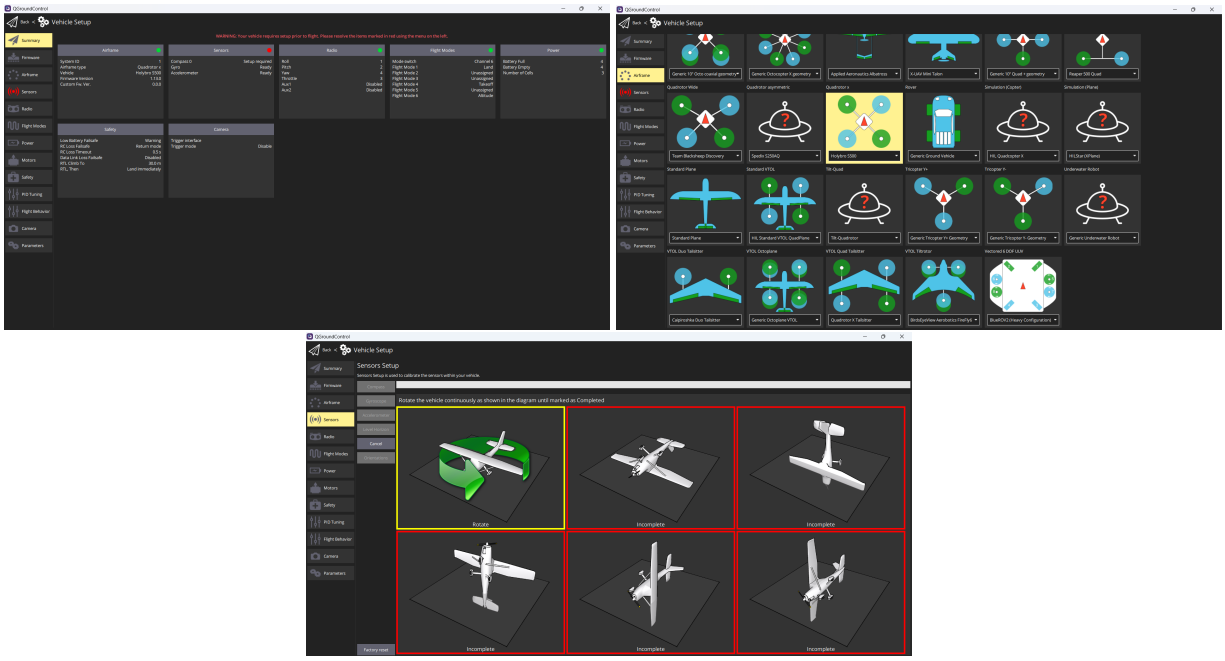


Figure 4.23: Screenshots from the QGroundControl calibration and setup tools used to configure the vehicle. From the top left: a) overview screen of the state of the vehicle components, a warning indicates that some sensors must be calibrated; b) airframe selection screen; c) gyroscope calibration screen detailing the process.

4.4.2 Initial tests

Baseline flight with factory software

Once the vehicle is fully configured and calibrated, 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 activated 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 a laptop used as an offboard computer. The MAVLink channel between the autopilot and the computer will be established through the telemetry radio by the developed `test-camera` tool. It is important to note that, during the flight, the QGroundControl application must be disconnected, as only one application can use the telemetry radio at a time. Consequently, the RC controller will serve as the backup control mechanism 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 in the same way as in the previous test. Since the DroneVisionControl application automatically arms the vehicle when sending a takeoff command, the two-position switch can be repurposed as a kill switch for all subsequent tests. This allows the operator to cut all power to the engines to protect the vehicle or the surrounding area if the autopilot were to malfunction. Some cases when this command can be useful include the vehicle trying to take off sideways because of a poorly balanced payload or a person getting too close to the propellers when the engines are running.

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

After successfully establishing the connection with the vehicle, the computer keyboard can be used to control the flight to test sending commands from DroneVisionControl. 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 receive velocity commands. These are sent with the WASD keys to control the forward and sideways movement of the vehicle and the QE keys to adjust its yaw rotation. At any point during the test, any input from the RC controller will activate manual control, and the drone will stop listening to the control application.

Figure 4.24 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 website[7].

Figure 4.24: Terminal output from the `test-camera` tool running on an offboard computer and image of the drone flying in response.

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

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 the cabled MAVlink channel to the autopilot from the onboard computer (Raspberry Pi). Additionally, it ensures that the onboard camera can capture images during flight that are clear enough for the pose detection software to detect and track a person in its field of view.

For this test, the same tool used in the previous test will be executed on the Raspberry Pi mounted onboard the vehicle. 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 is attached to the power module and the secondary battery to the Raspberry Pi. Once the onboard computer has powered up, it can be operated through a remote desktop connection via WiFi. Using this connection, a console can be opened on the Raspberry's desktop to execute the following command:

```
dronevisioncontrol tools test-camera --hardware /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[8]. Figure 4.25 shows the Raspberry Pi's desktop with pose detection running on the images taken from onboard the vehicle, as transmitted to the ground station laptop via WiFi.

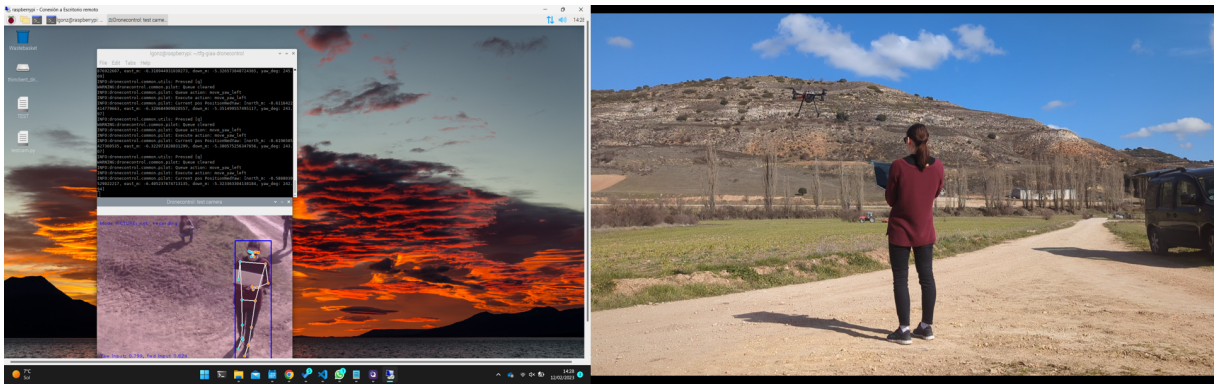


Figure 4.25: On the left, the onboard computer controlled from the ground station laptop runs pose detection while flying. On the right, the drone and the detected person are seen from a different angle

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

4.4.3 Hand gesture control with offboard computer

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 OS of the offboard computer.

```
dronevisioncontrol hand --serial <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. The open palm gesture (hold command) can be shown to the camera to test that the hand is correctly recognised without triggering any action. 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 forward and backward, 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[9]. Additionally, an image extracted from the video can be seen in Figure 4.26.

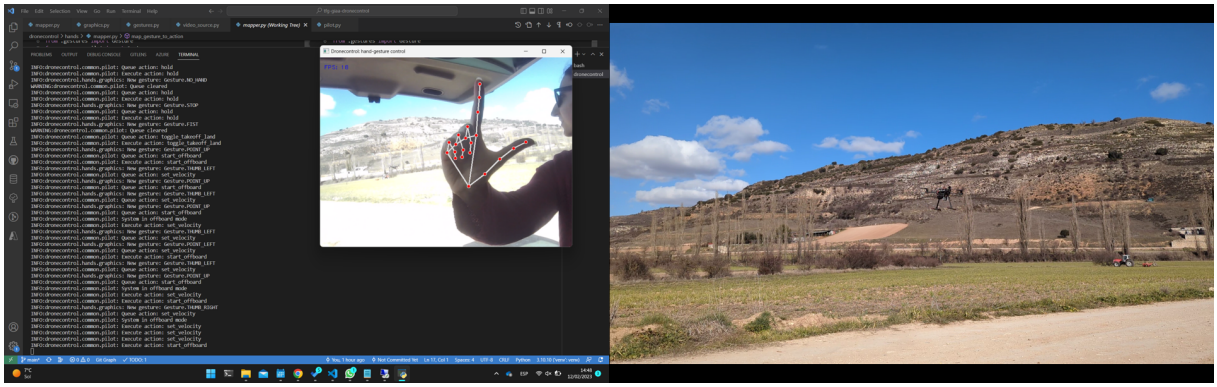


Figure 4.26: Image taken during flight controlled by the hand-gesture solution. The vehicle is moving forward.

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

4.4.4 Follow control with onboard computer

The last test to conduct is for the follow control solution. In this section, the onboard computer will run the control application to test detecting, tracking and following a moving person in a realistic scenario. The setup for this test will be the same as in the third test from Section 4.4.2, with the application running on the Raspberry Pi, controlled over WiFi from the ground station laptop. Unlike the test for the hand solution, DroneVisionControl does not need a wireless telemetry connection to connect to the Pixhawk. Consequently, the telemetry radio can be used to monitor the vehicle with the QGroundControl application while the flight is taking place. To ensure safety during the test, a second person must monitor the flight prepared with the RC controller to intervene and pilot the aircraft if the autonomous control fails in any way.

The control application will be started by connecting over WiFi to the Raspberry Pi desktop and opening a console to execute the following command:

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

The complete execution of this test has been recorded in this video[10] as shown in Figure 4.27. During the test, the vehicle is made to take off through the Raspberry Pi console by pressing the T key and to initiate following the detected person by pressing the O key to start offboard flight mode. Afterwards, when the person moves around in the field, the vehicle follows closely to maintain the person in its field of view. At the end of the test, the offboard mode is deactivated by pressing the O key again to check that the drone stops following the person and holds its position before landing the aircraft with the L key.

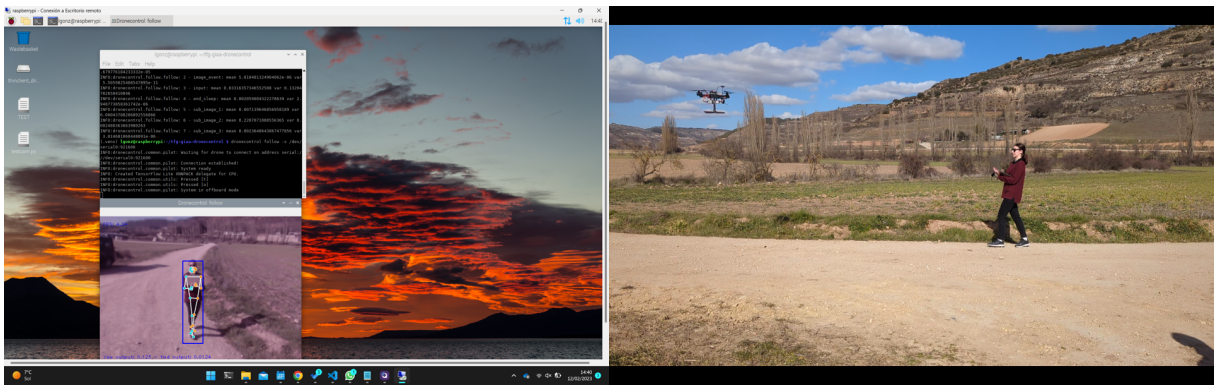


Figure 4.27: On the left, console and image output of the DroneVisionControl follow solution running on the Raspberry Pi. On the right, outside perspective of the flight test taking place.

In the recorded process, it can be appreciated that there are moments when the software

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

stops recognising the person in the image. In these cases, the vehicle noticeably stops moving and hovers in the air until the person is recognized again.

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 4.28 compares these measurements with those analyzed in Section 4.3.3, 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.

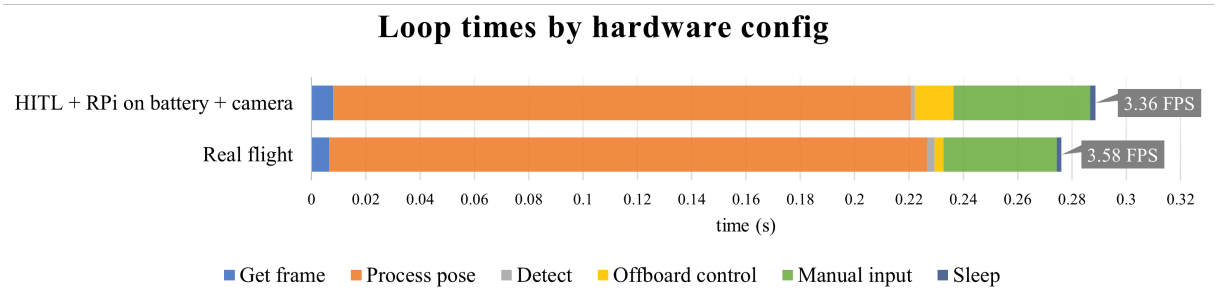


Figure 4.28: Average loop time for each individual task in the follow control solution and total frame rate for realistic simulation versus actual test flights.

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 summarizes the key conclusions obtained from the project and 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.

The main accomplishment of this thesis has been to achieve a complete end-to-end vision-based control solution that can make a UAV fly following a target person. From the test results exposed in Chapter 4, it can be seen that the designed software and hardware manage to work together to obtain the desired control of the vehicle, with good enough results to use the control mechanisms on real flight tests. Responsiveness is maintained during the flights, and changing conditions like vision loss are handled appropriately.

The second element to highlight is the testing process followed to validate the developed solutions, made possible by integrating the PX4 platform with the designed development and testing environment. The focus of this process has been to maintain a systematic step-by-step approach to testing that can be employed to validate each individual component of the system in isolation before combining them together.

The main drive of this emphasis on thorough testing has been to ensure safety at every point of the process, with a focus on the most potentially dangerous situations, the flight tests outside of simulation. Thanks to the unit tests on the components and the contingency measures implemented to regain control if errors occur, the potential for undesired scenarios is mitigated satisfactorily.

This systematic approach also facilitated identifying and isolating any conflictive elements during development. Some examples of obstacles whose origin was made easy to pinpoint by incremental testing include setting up thresholds and target setpoints experimentally for the hand and follow control solutions or discovering performance issues

from powering the onboard companion computer from the same power supply as the drone engines. A trial-and-error approach was likewise used to determine the most appropriate configuration for the different tools and the best communication channels for integrating components.

5.1 Evaluation of objectives

Casting back to the introduction for this project, the main objective was to illustrate how the tools in the PX4 ecosystem could be used to develop vision-driven control mechanisms for UAVs. This was achieved by building and presenting a complete development environment that integrates those tools into a platform that acts as the foundation for each stage of the process, from the concept phases to running tests in the target hardware.

In particular, the tools and systems developed by the Dronecode Project, which includes PX4, MavSDK and QGroundControl, were integrated with the designed DroneVisionControl application to reduce the workload of the project by making use of preexisting stable technologies. To show the minimum requirements to develop a complete control solution, low-cost and accessible hardware was employed to test all the developed software. This made it possible to dedicate the most time to developing the desired control solutions without being concerned with low-level electronics and control theory.

Additionally, it has been demonstrated how the development environment, the presented existing tools, and the chosen hardware can be connected 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.

Finally, the testing process presented in chapter 4 fulfils the objective of presenting 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 components.

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:

- 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.
- Safety-oriented testing methodology, including strategies for systematic validation of application components, performance analysis on software and hardware components and design of safety mechanisms to prevent accidents in loss of control situations.
- 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.
- Python skills: how to organise bigger projects with multiple submodules and create custom packages, as well as experience in several common 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.

  - Analysing and manipulating images from different sources with the OpenCV library.

- 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.

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 sensor fusion techniques to integrate other sensors, such as lidar, radar, or multiple camera systems, to improve the performance of the computer vision algorithm and allow for more precise tracking. This could reduce the noise in the input to the PID controllers and achieve a more optimal tuning, at the price of increasing the complexity of the detection mechanisms.
- More sophisticated computer vision algorithms: The algorithms used in this thesis employ basic image processing techniques to make up a detector-tracker machine learning pipeline. As an alternative to more sensors, 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 and mitigate the detection errors in the current 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 to achieve a better movement response in low-performance hardware.

- Design of a complete Graphical User Interface: the command-line interface offers limited control over the variables in the implemented solutions. Adding a graphical interface would make the program more user-friendly and accessible.
- Multi-drone control: The project focused 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 with a centralized companion computer to enable coordinated control in response to different perspectives of the environment from 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. 34

GPS Global Positioning System. 17

GUI Graphical User Interface. 42, 97

HITL Hardware-in-the-Loop. 20

IMU Inertial Measurement Unit. 22

OS Operating System. 17

PID proportional-integral-derivative. 51

QGC QGroundControl. 25

RC Radio Control. 18

SDK Software Development Kit. 11

SITL Software-in-the-Loop. 20

TCP Transport Control Protocol. 25

UAV Unmanned Aerial Vehicle. 1

UDP User Datagram Protocol. 25

WSL Windows Subsystem for Linux. 25

Appendix A

Installation manuals

This appendix describes the installation process for the tools employed throughout this project.

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 [57], 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 project repository[1], 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):

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

```
./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[2] and open the environment found in the `data` folder of the repository. To use AirSim with a different Unreal environment, follow the guide in the AirSim documentation[3] 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 Appendix 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

For the Raspberry Pi, the operating system of choice for the tests carried out in this project is the native Raspberry OS, but the configuration and results will be similar if using any

[2] https://www.unrealengine.com/download
[3] https://microsoft.github.io/AirSim/unreal_custenv/

other Linux distribution. The main requirement for the OS is that it counts with a graphical interface so that the output of the DroneVisionControl application can be visualised.

The installation of the DroneVisionControl application is done the same way as in the WSL system. The most significant difference with the SITL configuration comes from the setup of the Pixhawk board to run the hardware-in-the-loop simulation. The changes in configuration are done through the QGroundControl application, which can be connected to the board through a micro-USB cable or wirelessly through the telemetry radios. The steps to enable the HITL simulation are as follows:

1. Configure the airframe for simulation through the Vehicle Setup screen (Vehicle Setup -> Airframe -> HIL Quadcopter X).
2. Enable HITL simulation in the Safety settings (Vehicle Setup -> Safety -> HITL enabled).
3. Enable a new MAVLink channel on the secondary telemetry port (Vehicle Setup -> Parameters -> MAV_1_CONFIG.

Some additional steps need to be taken to enable a UART serial connection through the RX/TX pins on the GPIO header of the Raspberry Pi. A description of this process can be found in [58]. Other helpful documentation for setting up XRDP on the Raspberry Pi to start a remote desktop from a standalone PC can be found on [59].

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.

A.4 Library versions

The following versions of the external libraries have been employed throughout the project:

PX4 v1.12.3 Unreal Engine v4.27.2 AirSim v1.7.0 MAVSDK v1.2.0 MediaPipe v0.8.10 OpenCV-Python v4.5.4.60 simple-pid v1.0.1

```
{
    ”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

Application interface

B.1 Command-line interface

```
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 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.
```

B.2 Keyboard control

Table B.1 presents the options available for manual control of the drone through the keyboard on the companion computer running the DroneVisionControl application. These commands are available for all the test tools and control solutions (test-camera, hand, follow) as long as a MAVLink connection is established.

Z	Quit
K	Kill switch
H	Return home
0	Stop (Hover)
T	Take-off
L	Land
O	Toggle Offboard flight mode
W	Pitch forward
S	Pitch backward
D	Roll right
A	Roll left
Q	Yaw left
E	Yaw right
1	Up
3	Down
Space	Take picture / start video
<	Change picture saving mode (Photo <> video)
R	Reset image processing

Table B.1: Keyboard to drone command mapping.

Appendix C

PID tuning graphs

C.1 Yaw controller

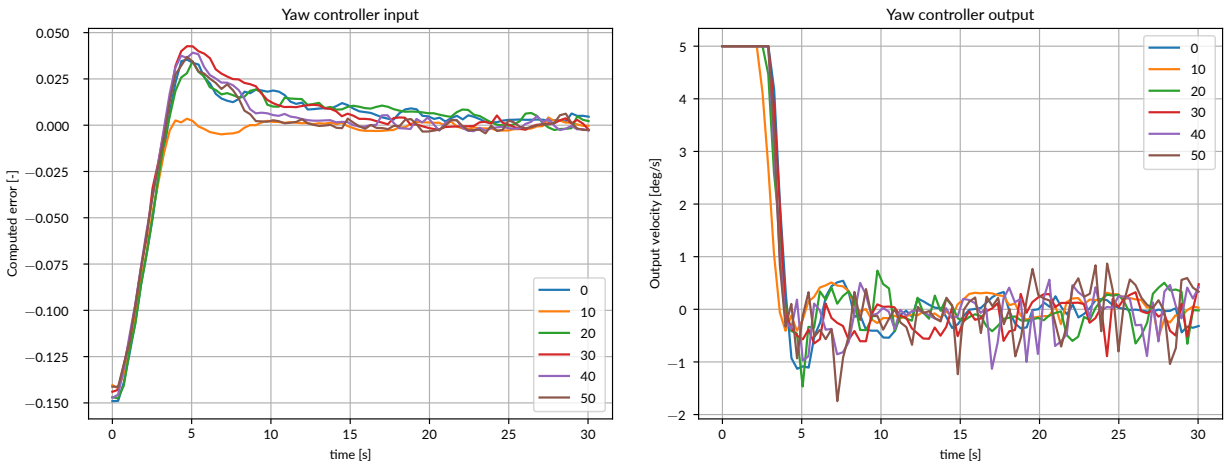


Figure C.1: Variation of (a) computed error and (b) output velocity for different values of K_I and $K_P = 100$, $K_D = 0$ while the yaw controller is engaged.

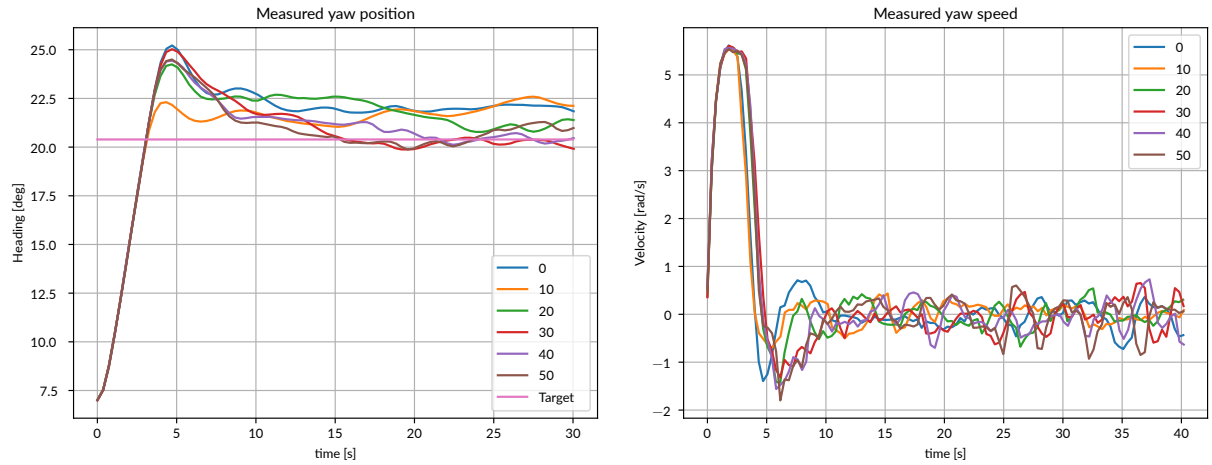


Figure C.2: Variation of (a) measured yaw heading and (b) measured yaw velocity for different values of K_I and $K_P = 100$, $K_D = 0$ while the yaw controller is engaged.

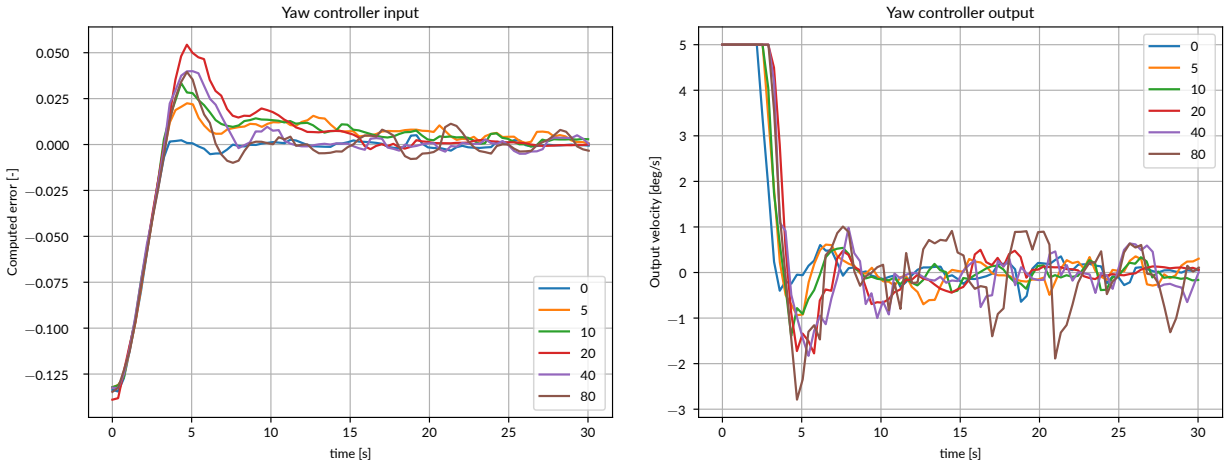


Figure C.3: Variation of (a) computed error and (b) output velocity for different values of K_D and $K_P = 100$, $K_I = 30$ while the yaw controller is engaged.

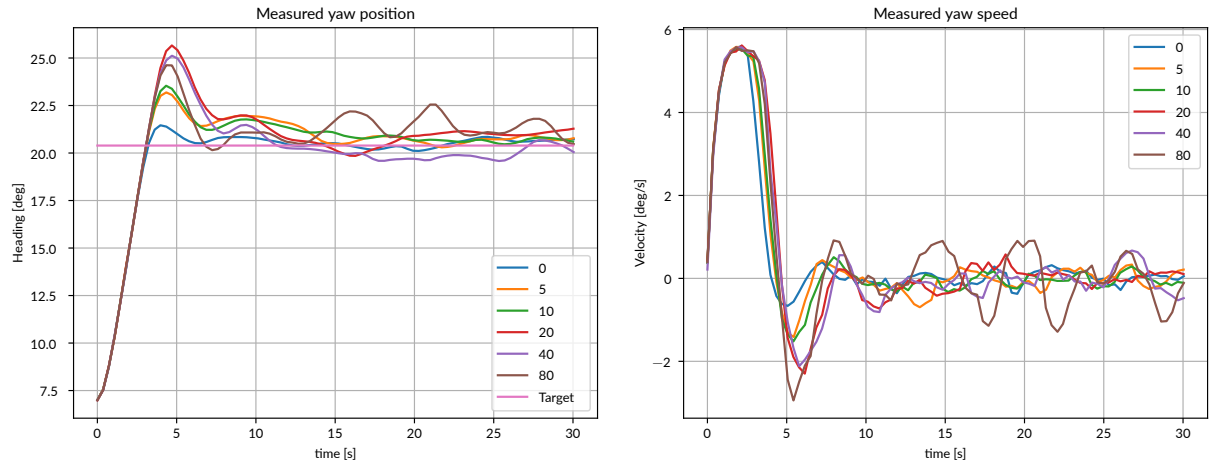


Figure C.4: Variation of (a) measured yaw heading and (b) measured yaw velocity for different values of K_D and $K_P = 100$, $K_I = 30$ while the yaw controller is engaged.

C.2 Forward controller

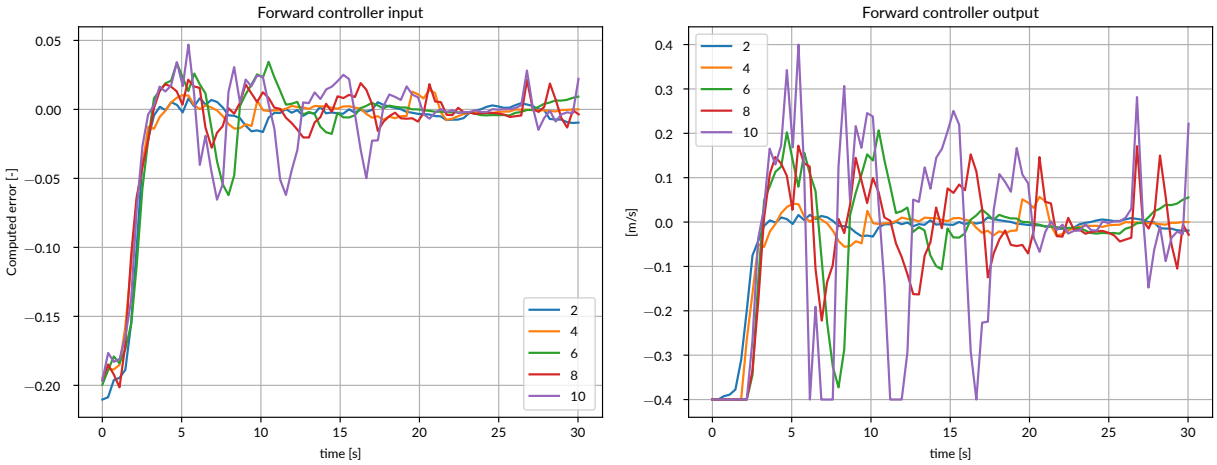


Figure C.5: Variation of (a) computed error and (b) output velocity for different values of K_P and $K_I = 0$, $K_D = 0$ while the forward controller is engaged.

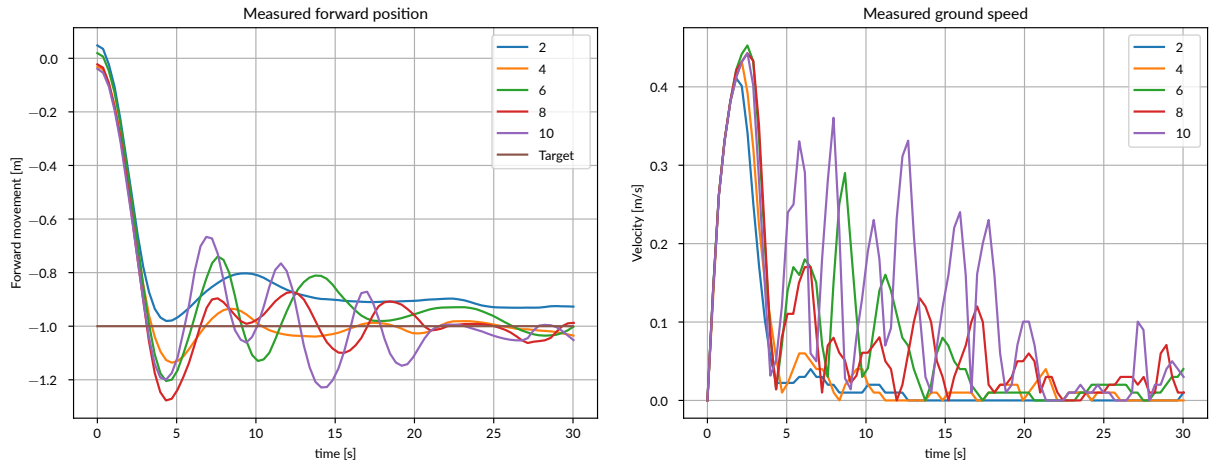


Figure C.6: Variation of (a) measured forward position and (b) measured absolute ground velocity for different values of K_P and $K_I = 0$, $K_D = 0$ while the forward controller is engaged.

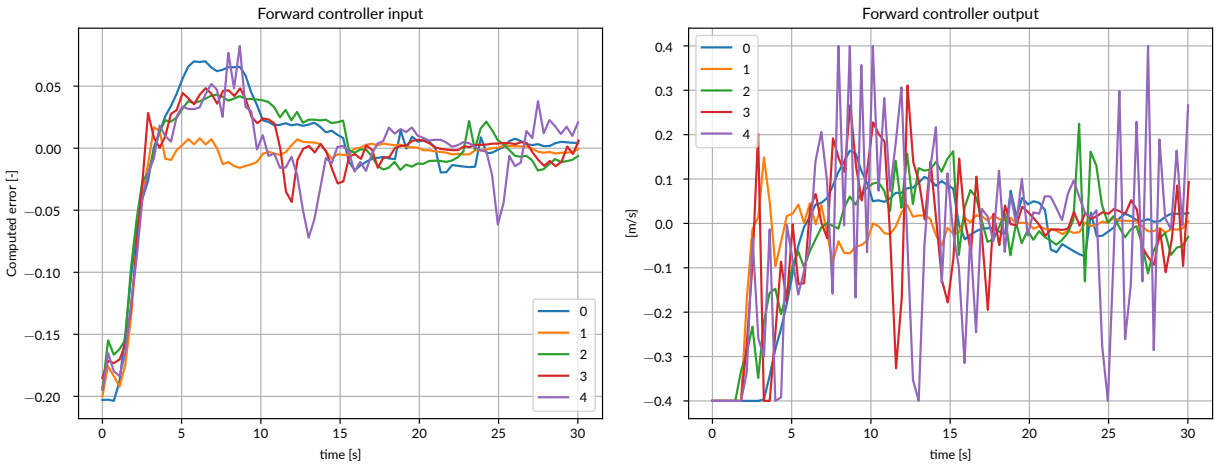


Figure C.7: Variation of (a) computed error and (b) output velocity for different values of K_I and $K_P = 4$, $K_D = 0$ while the forward controller is engaged.

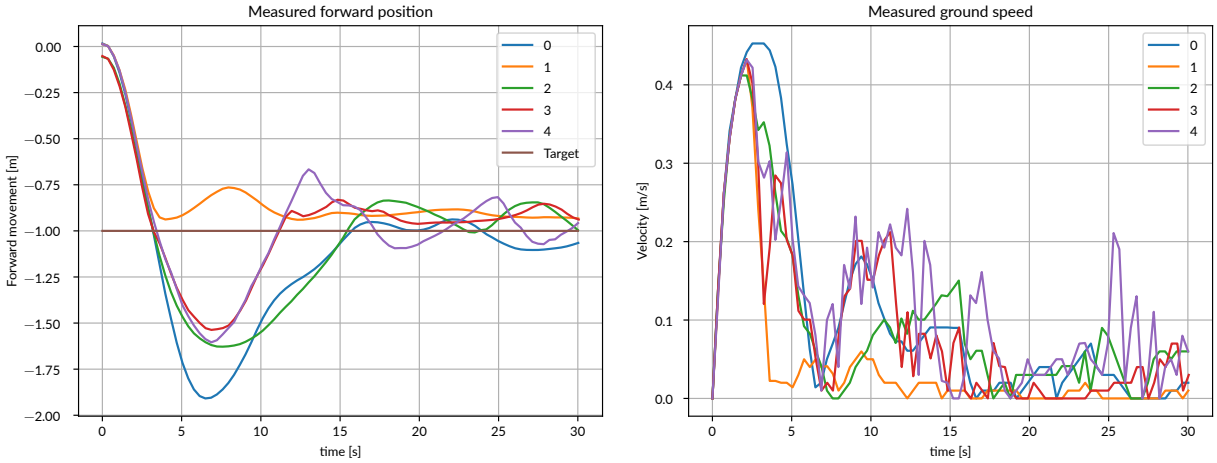


Figure C.8: Variation of (a) measured forward position and (b) measured absolute ground velocity for different values of K_I and $K_P = 4$, $K_D = 0$ while the forward controller is engaged.

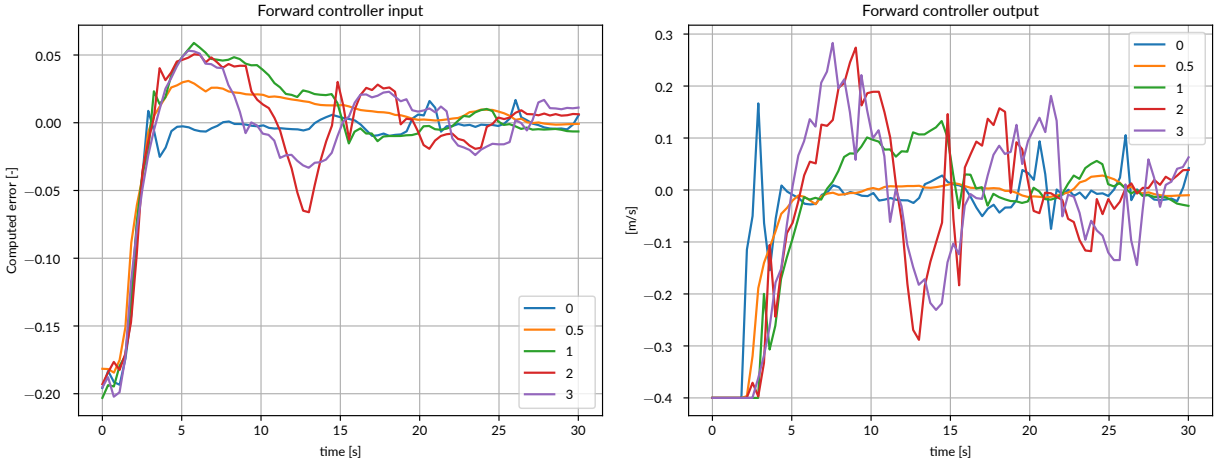


Figure C.9: Variation of (a) computed error and (b) output velocity for different values of K_D and $K_P = 4$, $K_I = 1$ while the forward controller is engaged.

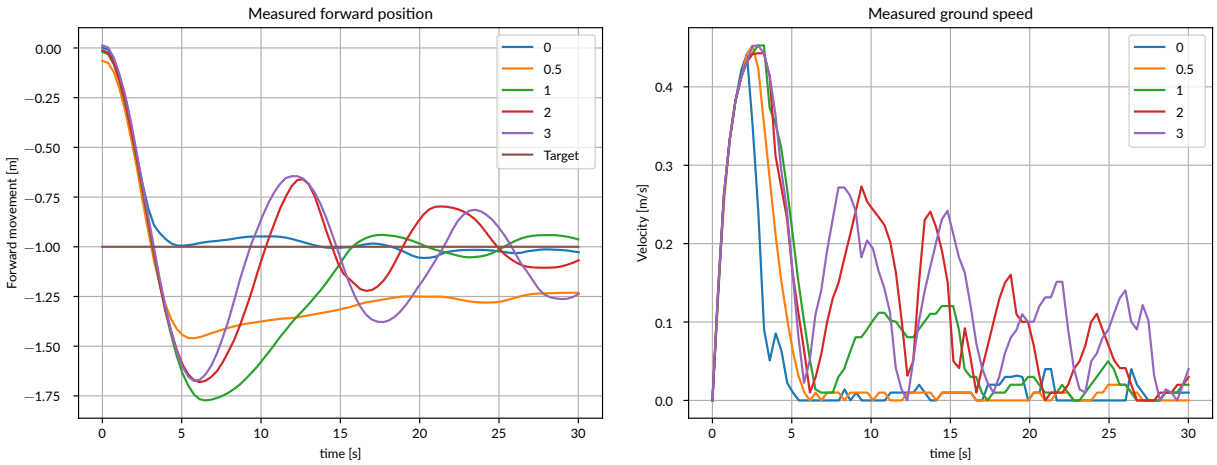


Figure C.10: Variation of (a) measured forward position and (b) measured absolute ground velocity for different values of K_D and $K_P = 4$, $K_I = 1$ while the forward controller is engaged.

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] *Open Source Autopilot for Drones - PX4 Autopilot*. URL: https://px4.io/ (visited on 16/02/2023).

[17] *The Dronecode Foundation*. URL: https://www.dronecode.org/ (visited on 16/02/2023).

[18] *Introduction · MAVLink Developer Guide*. URL: https://mavlink.io/en/ (visited on 16/02/2023).

[19] *Introduction · MAVSDK Guide*. URL: https://mavsdk.mavlink.io/main/en/index.html (visited on 16/02/2023).

[20] *QGC - QGroundControl - Drone Control*. URL: http://qgroundcontrol.com/ (visited on 16/02/2023).

[21] *The most powerful real-time 3D creation tool - Unreal Engine*. URL: https://www.unrealengine.com/en-US (visited on 16/02/2023).

[22] *Home - AirSim*. URL: https://microsoft.github.io/AirSim/ (visited on 16/02/2023).

[23] *MediaPipe - Google for Developers*. URL: https://developers.google.com/mediapipe (visited on 03/10/2023).

[24] *GitHub - l-gonz/tfg-giaa-dronecontrol*. URL: https://github.com/l-gonz/tfg-giaa-dronecontrol (visited on 16/02/2023).

[25] *GitHub - l-gonz/tfg-giaa-memoria*. URL: https://github.com/l-gonz/tfg-giaa-memoria (visited on 16/02/2023).

[26] *Exploration of Vision-Based Control Solutions for PX4-Driven UAVs*. URL: https://l-gonz.github.io/tfg-giaa-dronecontrol/ (visited on 20/04/2023).

[27] *Holybro - Build your drone from here – Holybro Store*. URL: https://shop.holybro.com/ (visited on 16/02/2023).

[28] *Homepage - Pixhawk*. URL: https://pixhawk.org/ (visited on 16/02/2023).

[29] *Apache NuttX - Home*. URL: https://nuttx.apache.org/ (visited on 16/02/2023).

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

[31] *Buy a Raspberry Pi 4 Model B â Raspberry Pi*. URL: https://www.raspberrypi.com/products/raspberry-pi-4-model-b/ (visited on 16/02/2023).

[32] 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.

[33] *Holybro X500 + Pixhawk4 Build | PX4 User Guide (main)*. URL: https://docs.px4.io/main/en/frames_multicopter/holybro_x500_pixhawk4.html (visited on 16/02/2023).

[34] *Logitech C920 PRO HD Webcam, 1080p Video with Stereo Audio*. URL: https://www.logitech.com/en-gb/products/webcams/c920-pro-hd-webcam.960-001055.html (visited on 16/02/2023).

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

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

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

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

[39] *PyPI · The Python Package Index*. URL: https://pypi.org/ (visited on 16/02/2023).

[40] *SiK Telemetry Radio V3 – Holybro Store*. URL: https://holybro.com/collections/telemetry-radios/products/sik-telemetry-radio-v3 (visited on 12/10/2023).

[41] 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).

[42] *PM07 Quick Start Guide - Holybro Docs*. URL: https://docs.holybro.com/power-module-and-pdb/power-module/pm07-quick-start-guide (visited on 16/02/2023).

[43] *asyncio — Asynchronous I/O — Python 3.11.4 documentation*. URL: https://docs.python.org/3/library/asyncio.html (visited on 13/01/2023).

[44] *Offboard Mode | PX4 User Guide (main)*. URL: https://docs.px4.io/main/en/flight_modes/offboard.html#offboard-mode (visited on 16/02/2023).

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

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

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

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

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

[50] 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).

[51] *Gazebo*. URL: https://gazebosim.org/home (visited on 16/02/2023).

[52] *Practical PID tuning guide*. URL: https://tlk-energy.de/blog-en/practical-pid-tuning-guide (visited on 04/10/2023).

[53] *Measurement Noise Degrades Derivative Action*. URL: https://controlguru.com/measurement-noise-degrades-derivative-action/ (visited on 04/10/2023).

[54] B. Kristiansson and B. Lennartson. 'Robust tuning of PI and PID controllers: using derivative action despite sensor noise'. In: *IEEE Control Systems Magazine* 26.1 (2006), pp. 55–69. DOI: 10.1109/MCS.2006.1580154.

[55] *Radio Control (RC) Setup | PX4 User Guide (main)*. URL: https://docs.px4.io/main/en/config/radio.html (visited on 16/02/2023).

[56] *xrdp by neutrinolabs*. URL: https://www.xrdp.org/ (visited on 10/10/2023).

[57] 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).

[58] *Talking to a PX4 FMU with a RPi via Serial #noUSB - MAVSDK*. URL: https://discuss.px4.io/t/talking-to-a-px4-fmu-with-a-rpi-via-serial-nousb/14119?page=2 (visited on 10/02/2023).

[59] *How to Install Xrdp Server (Remote Desktop) on Raspberry Pi*. URL: https://linuxize.com/post/how-to-install-xrdp-on-raspberry-pi/ (visited on 10/02/2023).