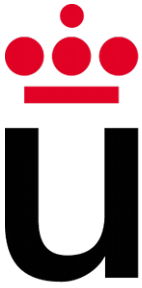


Universidad
Rey Juan Carlos

GRADO EN INGENIERÍA AEROESPACIAL EN AERONAVEGACIÓN

Curso Académico 2022/2023

Trabajo Fin de Grado

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

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

Trabajo Fin de Grado

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

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

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

Presidente:

Secretario:

Vocal:

y habiendo obtenido la siguiente calificación:

Calificación:

Móstoles/Fuenlabrada, a de de 20XX

Aquí normalmente
se inserta una dedicatoria corta

Agradecimientos

Aquí vienen los agradecimientos...

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

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

Resumen

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

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

Lo mejor es escribir el resumen al final.

Abstract

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

Todo list

Contents

List of Figures

List of Listings

Chapter 1

Introduction

1.1 General context

An Unmanned Aerial Vehicle (UAV) is an aircraft able to fly without any pilot or operator on board. They may be operated through remote control by a human operator or with various degrees of autonomy, from sensor-driven control aids provided to the operator to fully autonomous flight through preplanned missions. UAVs have existed since the 20th century and were initially developed as military technology to protect pilots from dangerous missions. However, as the cost of the electronics and sensors decreased and control technologies improved, they became available to a broader public. They are now employed in various civil applications like crop control, search and rescue, or filmmaking and photography. Nevertheless, most commercial solutions are limited to remote operations or rudimentary autonomy, with fully autonomous flight still in the earliest phases. However, there is a growing trend towards using vision-based control solutions, which can offer improved performance and flexibility compared to traditional control methods. Vision-based control solutions use cameras and image processing algorithms to provide real-time feedback and control of the UAV. These solutions are often more flexible and robust than traditional control methods, which are typically based on accelerometers, gyroscopes, and other sensors. Additionally, vision-based control solutions can be implemented on low-cost platforms, making them accessible to a broader range of users. These vision-based guidance systems have only started to appear in recent years as artificial intelligence becomes more widespread and very few fully-developed consumer-ready platforms are available for the general public. Even so, from the essential hardware to build an own quadcopter, a helicopter with four rotors that represents the most common type of UAV, to miniaturized computers that handle complex calculations and open-source software that can be customized to endless applications, all the individual pieces that enable building such a system are readily available.

1.2 The Dronecontrol project

The project presented in this thesis aims to show the options available to design and implement control solutions for the popular PX4 open-source autopilot platform and how to integrate them with detection and tracking computer vision mechanisms to achieve simple vision-based self-guided UAVs. All the necessary code for this project has been written in Python and always runs outside the flight controller board on a companion computer. This allows more processing power to be accessible for any compute-intensive computer vision algorithms, but also ensures that the control solutions are abstracted from the hardware components and not dependent on the specific flight stack employed, to allow applying the results to any available autopilot platform with exposed APIs. Two complete vision-based control solutions have been implemented for two scenarios: keeping the machine driving the computer vision algorithms onboard or offboard the vehicle during flight. The following chapters detail the hardware, libraries, testing environments, and systems used during the development process and how they can be used to develop new control solutions based on available computer vision mechanisms.

1.3 Objectives

The main objective of this project is to demonstrate the available possibilities to develop control solutions for the PX4 autopilot driven by computer vision mechanisms.

More specifically, it aims to:

- Introduce the software and hardware environment of the Dronecode project and the techniques they have made available.
- Show the minimum requirements needed to develop control software for the platform.
- Suggest viable vision-driven control solutions that use those techniques.
- Present a testing process using available tools that can help ensure safety and reliability for any new solutions developed.
- Build a quadcopter and use it to carry out test flights with the developed solutions.

Figure 1.1: Timeline for the development of the project

1.4 Time planning

This project has been carried out over a natural period of one and a half years due to having to coordinate with working a full-time job in parallel. The development has been spread out over three phases, taking up a total of 400 hours of dedication to the project. Each of these phases can be divided into three main tasks: research of the necessary systems, programming of custom solutions and tools, and comprehensive testing on different scenarios. Additionally, during the last two phases, time was allocated to the process of writing the report, which occurred concurrently with the research and development work.

- Phase 1: Proof-of-concept (Oct 21 - Jan 22: 61 hours)
  - Research PX4 (12 hours)
  - Programming hand solution (42 hours)
  - Test standard drone build (7 hours)
- Phase 2: Follow solution (Feb 22 - Aug 22: 138 hours)
  - Research tracking and detection (33 hours)
  - Programming follow solution (80 hours)
  - Test custom hardware (25 hours)
- Phase 3: Hardware (Sep 22 - Feb 23: 65 hours)
  - Research hardware (15 hours)
  - Make test tools and polish code (33 hours)
  - Test custom hardware and integration (17 hours)
- Writing report (133 hours)

1.5 Thesis layout

This section details the structure of this thesis. It is organized into five chapters that reflect the three distinct tasks described in the last section: research, development and testing, along with this introduction and some final conclusions. Specifically:

- In the first chapter, there is a brief introduction to the context in which the project has been developed, as well as the objectives it pursues.
- Chapter 2 presents the technologies and tools employed in this project and the current state-of-the-art for UAV vision-based control solutions.
- The third chapter introduces the simulation environments used to develop the solutions throughout the project and the architecture of the hardware and software used.
- Chapter 4 follows along through the process of testing every part of the system incrementally until reaching the final flight tests.
- The last chapter shows the conclusions drawn from the work and presents ideas for future development.

Chapter 2

State of the art

2.1 Literature review

Vision-based control solutions for UAVs on low-cost platforms have gained significant attention recently due to the increasing demand for cost-effective and efficient aerial applications. This literature review summarizes the existing research in this field and highlights the essential findings and challenges.

A significant amount of research has focused on developing algorithms for real-time image processing, focusing on improving the accuracy and robustness of tracking solutions for UAV applications. In [1], a computer vision algorithm for position measurement and velocity estimation using optical flow is proposed for tracking ground-moving targets. In [2], the focus is to solve the problem of tracking and following a generic human target by a drone in a natural, possibly dark scene 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 as much as possible both the control technics and the computer vision mechanisms of detection and tracking to focus on presenting an easy-to-use platform that can employ already developed algorithms and combine them to make robust systems with a lower threshold of knowledge.

Other studies have already centred on developing lower-cost solutions for more accessible platforms. Many have used the Parrot *AR.Drone* camera-enabled quadcopter that allows controlling through WiFi from an external offboard computer. In [4], [5], [6], and [7], the *AR.Drone* is used for implementing object tracking and following solutions in different environments and conditions with a higher emphasis on the type of trackers employed to achieve a robust visual following mechanism. In [8], [9], and [10], there is a higher focus on exposing the capabilities of the *AR.Drone* as a platform to develop custom control solutions utilizing the navigation and control technology and the low-cost sensors already embedded in the vehicle. The main difference between the platform used in the mentioned research and the PX4 platform that is the target of this thesis is that the *AR.Drone* is offered as-is as a concrete low-cost vehicle for which autonomous guidance and control can be developed, while the PX4 platform is part of an extensive environment that encompasses everything from the minimum essential hardware to the low-level autopilot mechanisms to the high-level control commands and allows complete personalization at each layer of the system.

As the PX4 platform has been available for less than a decade and has only become a widely-recognized standard in the drone industry since 2020, there is still little research on its ecosystem, and fewer specific computer-vision projects have been developed. However, some studies share some of the possibilities that the PX4 software and the Pixhawk flight controllers offer in the field. In [11], aerial images are analyzed in real-time and fed to a

deep-learning architecture to calculate optimal flight paths. In [12], a vision-based precision landing method for quadcopter drones is developed on the Pixhawk flight controller to prevent crashes on landing.

Another vital advantage of the PX4 platform over other available platforms is its compatibility with a wide range of simulators that allow the development of complex control systems in changing environments with less need for expensive, time-consuming, and meticulous testing of the correct integration of the hardware and software components through real-world flight tests of UAVs that traditional development often entails. Some studies have obtained results in this area of the ecosystem. In [13], the Gazebo simulator and the PX4 software are used to develop a system for simulating realistic navigation conditions for obstacle avoidance. In [14], a similar ROS-Gazebo environment serves as the basis for designing fault-tolerant controllers that can recover from rotor failure. In [15], the aim is to integrate a photo-realistic environment simulator with a flight-dynamics simulator to develop full autonomy in the Pixhawk autopilot board. These represent only some of the possible applications of such a complete ecosystem for developing solutions.

In conclusion, vision-based control solutions are still a relatively new field in the broader topic of autonomous guidance and navigation for UAVs. Some older low-cost platforms have been more extensively researched as the basis for developing accessible control systems driven by computer vision. However, for PX4-driven UAVs, there are still many unexplored possibilities when it comes to applying the simulator capabilities of the platform to the development of vision-based control solutions and taking advantage of modern rendering techniques to simulate complex detection and tracking scenarios that reduce the need for demanding real-world flight tests.

2.2 Methodology

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

2.2.1 Software

PX4 autopilot

PX4[1] is a professional open-source autopilot flight stack developed in C++ by developers from industry and academia and supported by an active worldwide community. It powers all kinds of vehicles, from racing and cargo drones to ground vehicles and submersibles. The flight stack software runs on a vehicle controller or flight controller hardware. It supports both Ready-To-Fly vehicles, custom builds made from scratch, and many additional sensors and peripherals, such as distance and obstacle sensors, GPS, camera payloads and onboard computers.

PX4 is a core part of a broader drone platform, the Dronecode Project[2], that includes the QGroundControl ground station, the Pixhawk hardware for flight controllers and the MAVSDK library for integration with companion computers, cameras and other hardware using the MAVLink protocol. PX4 was initially designed to run on Pixhawk series controllers but can now run on Linux computers and other less specific hardware. The software controls the vehicles through flight modes. Flight modes define how the autopilot responds to remote control input and how it manages vehicle movement during fully autonomous flight. The modes provide different types or levels of autopilot assistance to the user, ranging from automation of common tasks like takeoff and landing to mechanisms that make it easier to regain level flight or hold the vehicle to a fixed path or position.

MAVLink and MavSDK

MAVLink[3] is a lightweight messaging protocol for communicating with drones and between onboard drone components. It follows a modern hybrid publish-subscribe and point-to-point design pattern where data streams are published as topics while configuration sub-protocols, such as the mission protocol or parameter protocol, are sent as point-to-point with retransmission. Messages are defined within XML files. Each XML file defines the message set supported by a particular MAVLink system.

[1] https://px4.io/
[2] https://www.dronecode.org/
[3] https://mavlink.io/en/

MAVSDK[4] is a cross-platform collection of libraries for various programming languages to interface with MAVLink systems such as drones, cameras or ground systems. It is primarily written in C++ with wrappers available for, among others, Swift, Python and Java. The Python wrapper is based on a gRPC (Google Remote Procedure Call) client communicating with the gRPC server running C++. The libraries deliver a simple API for managing one or more vehicles, providing programmatic access to vehicle information and telemetry and control over missions, movement and other operations. The libraries can be used onboard a drone on a companion computer or on the ground from a ground station or mobile device.

QGroundControl

QGroundControl[5] is an open-source ground control station from the Dronecode Project that provides flight control and mission planning for any MAVLink-enabled drone. It is designed with a focus on ease of use for professional users and developers. QGroundControl connects automatically to flight controllers running the PX4 Autopilot, both through a cabled or wireless connection, as well as to any local simulator running the PX4 flight stack.

The software enables setting up the initial configuration of a flight controller and calibrating sensors and other peripherals connected to it, editing and keeping track of modified configuration parameters and sending flight commands to the drone, like arming, takeoff and landing. It provides a map interface to keep track of the vehicle's GPS location and select target location points for planning flight missions, where the vehicle goes through each location at the designated speed and altitude. Figure 2.1 shows the application interface on a Windows system.

Unreal Engine and AirSim

Unreal Engine[6] is a 3D computer graphics creation tool best known for its game development capabilities. It was first released in 1998 to develop first-person shooters, but its features have expanded over the years. The engine is now used in all kinds of fields, like film and television, to create virtual sets, real-time rendering and computer-generated animation, and research, especially as a basis for virtual reality tools and developing virtual environments for the design of buildings and vehicles, among others, leveraging the engine's real-time graphic generation. Figure 2.2 shows the program's interface when a project is loaded into the engine.

[4] https://mavsdk.mavlink.io/main/en/index.html
[5] http://qgroundcontrol.com/
[6] https://www.unrealengine.com/en-US

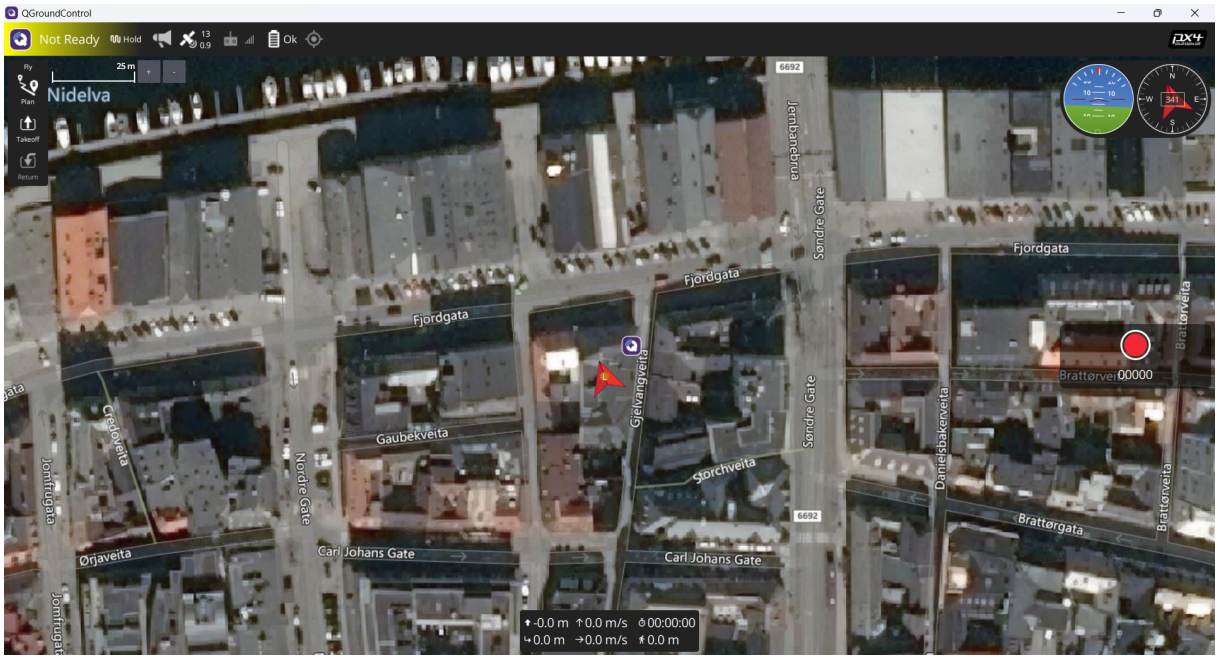


Figure 2.1: Main interface for the QGroundControl program

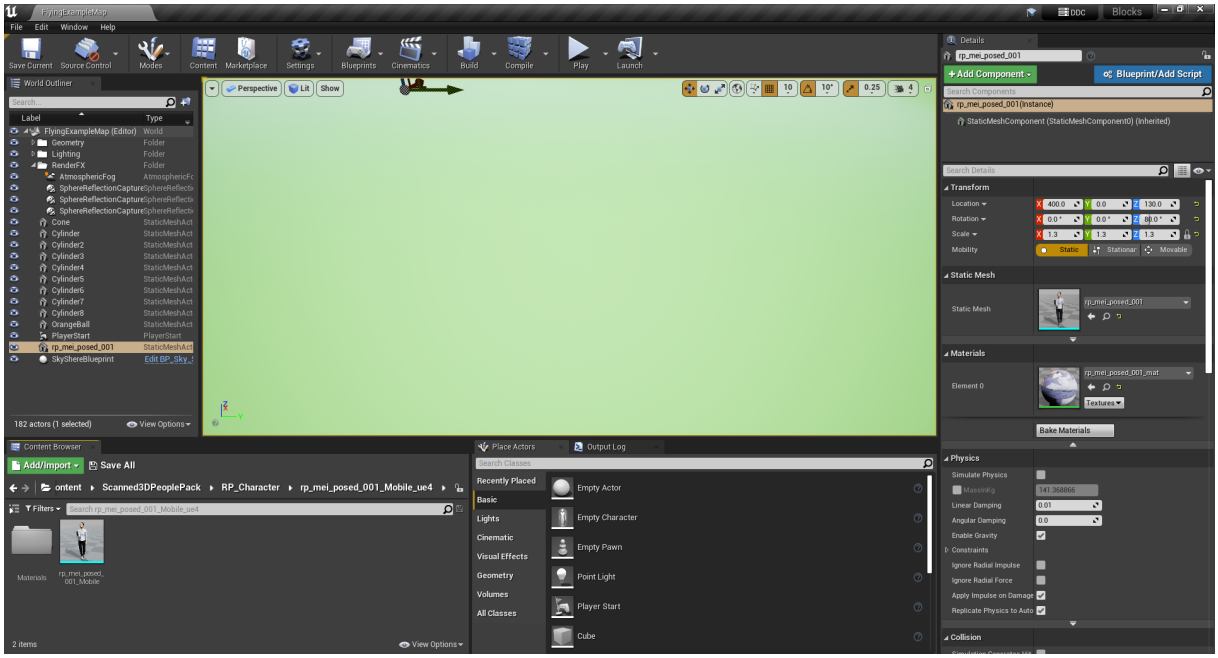


Figure 2.2: Project interface for the Unreal Engine

AirSim[7] is a simulator for drones, cars and more, built on Unreal Engine and developed by Microsoft and released in 2017. It is open-source, cross-platform, and supports software-in-the-loop (SITL) simulation with popular flight controllers such as PX4 and ArduPilot and hardware-in-the-loop (HITL) simulation with PX4 for physically and visually realistic simulations. It is developed as an Unreal Engine plugin that can be incorporated into any existing Unreal environment. Its goal is to develop a platform for AI research to experiment with deep learning, computer vision and reinforcement learning algorithms for autonomous vehicles. For this purpose, AirSim also exposes APIs to retrieve data and control vehicles in a platform-independent way. At the end of 2022, Microsoft announced that they would be releasing a new simulation platform, Project AirSim, in the coming year and subsequently archiving the original 2017 AirSim, which would stop receiving updates but remain available to the public.

MediaPipe

Mediapipe[8] is an open-source project developed by Google that offers cross-platform, customizable machine-learning solutions for live and streaming media. It supports End-to-End acceleration with built-in fast ML inference and accelerated processing even on rudimentary hardware and a unified solution that works across Android, iOS, desktop/cloud, web and IoT. It offers a framework designed specifically for complex perception pipelines, like the real-time perception of human pose, face landmarks and hand tracking that can enable various impactful applications, such as fitness and sports analysis, gesture control and sign language recognition, augmented reality effects and more.

GitHub

GitHub is an online hosting service using the distributed version control system Git for software development. It is currently the most significant source code host, with over 350 million repositories. Both the code and the front website for this project are stored in GitHub.

OpenCV

OpenCV is an open-source library for computer vision, machine learning and image processing that counts with over 2000 algorithms. It can be used to retrieve images or videos from cameras, extract data from them and edit them. It is developed in C++ but offers Python wrappers that take advantage of the general-purpose programming language while

[7] https://microsoft.github.io/AirSim/
[8] https://google.github.io/mediapipe/

maintaining the speed of the underlying computationally intensive C++ code. OpenCV-Python employs the Numpy library, designed for highly optimized numerical operations with a syntax based on the MATLAB style. All the OpenCV array structures are converted in Python to and from Numpy arrays. This makes it easy to work with other libraries that use Numpy, like Matplotlib to plot graphs.

2.2.2 Hardware

Pixhawk 4

The Pixhawk 4 is an advanced autopilot module designed and produced by Holybro[9], a UAV parts and kits manufacturer, in collaboration with the Dronecode Project team. It is based on the Pixhawk-project[10] FMUv5 open hardware design, and it is optimized to run PX4 on the NuttX[11] operating system by the Apache Foundation. The Pixhawk 4 has an integrated accelerometer/gyroscope, a magnetometer, and a barometer, which enables it to drive an unmanned aerial vehicle using the PX4 flight stack natively without any other sensors. Even so, it also includes many connector sockets to extend its features with additional sensors, input/output devices, or a companion computer. Figure 2.3 shows the autopilot module with all its connectors and buttons.

Holybro X500

The Holybro X500[12] is a quadcopter designed by Holybro to use with PX4. It comes as a ready-to-assemble development kit comprising a full carbon-fibre twill frame and the Pixhawk 4 flight controller. The kit also includes a power management board, four motors, a GPS module, an RC receiver, and a telemetry radio. It has a build time of approximately 3 hours and requires no specialized tools. Figure 2.4 shows the final result of the building process.

Raspberry Pi 4B

The Raspberry Pi[13] is a line of single-board computers that stands out thanks to its affordable price, compact size, and maker-friendly design. Model 4B is an improved version of

[9] https://shop.holybro.com/
[10] https://pixhawk.org/
[11] https://nuttx.apache.org/
[12] https://docs.px4.io/main/en/frames_multicopter/holybro_x500_pixhawk4.html
[13] https://www.raspberrypi.com/products/raspberry-pi-4-model-b/

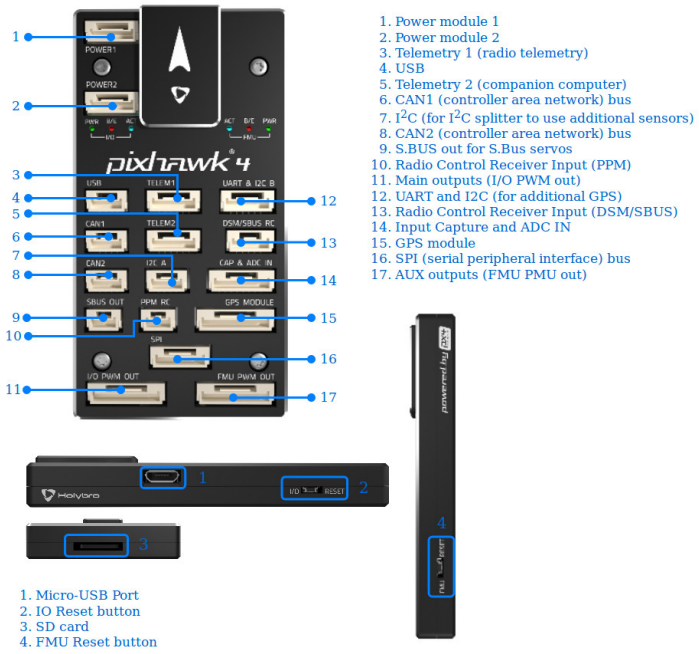


Figure 2.3: Side views and connector map for the Pixhawk 4 autopilot module.

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

its predecessors, significantly increasing processing power, video output, and peripheral connectivity while maintaining the same low price and tiny size offered on past models. This small computer comes as a bare circuit board, without any housing or add-ons, such as a cooling fan or a power button, but it includes USB, HDMI, and Ethernet ports and both Wi-Fi and Bluetooth connectivity, as well as a 40-pin GPIO header, a row of input/output pins that provides direct access for connecting external devices. The Raspberry Pi runs natively Raspbian OS, a free operating system based on Debian optimized for the Pi hardware, but it is compatible with other standard flavours of Linux.

Figure 2.4: Fully assembled X500 kit.

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

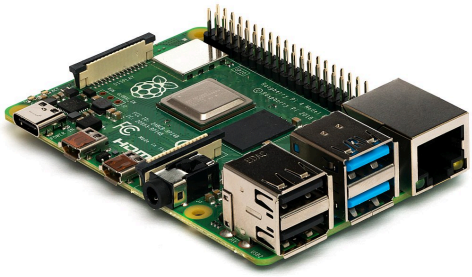


Figure 2.5: Raspberry Pi 4 Model B

Source: Wikimedia Commons [17].

Chapter 3

Design and implementation

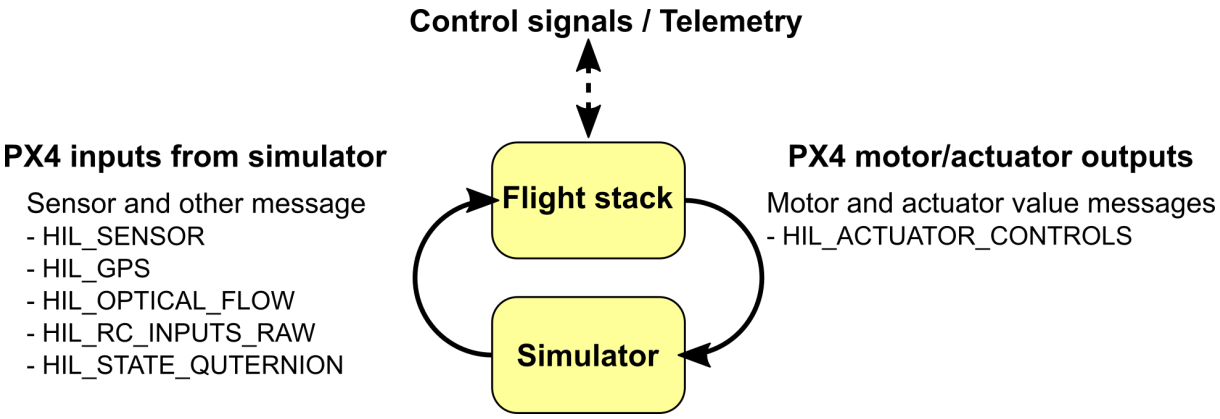


Figure 3.1: MAVLink messages exchanged between the simulator and the flight stack during simulation.

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

3.1 Development environment and simulation

During the process of developing the application, it became necessary to be able to continuously deploy and test the latest version without having to depend on flying the physical vehicle but instead relying on the simulation of the system inside the computer that was at the same time running the developed software. This configuration has the twofold advantage of reducing the development time on the one hand since there is no need to be concerned with the interactions between the different hardware components, and the results can be visualized immediately on the computer screen, and on the other hand, of increasing the safety of the process by executing on the vehicle only software that has already been tested to an acceptable point.

Simulators allow PX4 flight code to control a computer-modelled vehicle in a simulated "world" that can be interacted with in the same ways as with an actual vehicle, using QGroundControl, an offboard API or a radio controller. PX4 supports two different simulation modes: software-in-the-loop (SITL), where the flight stack runs on a non-dedicated computer, and hardware-in-the-loop (HITL), where the simulation firmware executes on an actual flight controller board. Communication into and out of the flight stack uses the MAVLink protocol mentioned in section 2.2.1, which allows exchanging messages between drones, ground control stations, and other MAVLink systems [18]. When the firmware is simulated, a MAVLink server is always started as part of the running software to enable communication with the simulator program, and any other offboard entry point that could be present.

In both SITL and HITL modes, the simulation works according to the feedback loop shown in figure 3.1. The simulator generates the input from the sensors based on its internal world representation and sends it through MAVLink messages to the flight stack running

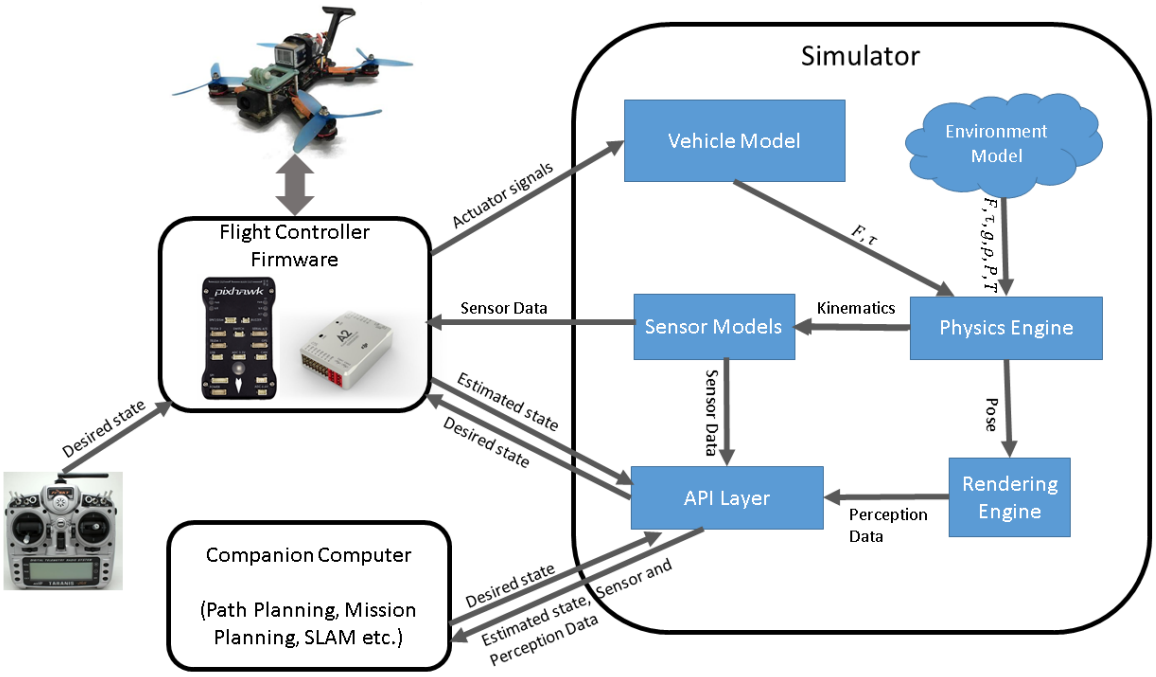


Figure 3.2: High-level overview of how different components interact in the AirSim simulator.

Source: Adapted from *Aerial Informatics and Robotics Platform* [19].

on the same computer using the UDP transport protocol. This, in turn, generates response actuator controls that are fed back into the simulator in the same way to affect the vehicle's position, velocity, and attitude in the simulated world. Simulated communications employ MAVLink messages specific to the mode in use and are not precisely the same as those used during non-simulated flights.

There are many options for simulators supported by PX4, like Gazebo, a robust 3D simulation environment for Linux systems that is particularly suited for testing object avoidance and is commonly used with ROS, or AirSim (2.2.1), a more resource-intensive cross-platform simulator that leverages the Unreal Engine, typically used for game development and animation, to provide physically and visually realistic simulations. For this project, AirSim was chosen because of previous experience with Unreal Engine, the easy availability of visual packages to test computer vision features and its native support for running on Windows machines, which is the operating system running on the computer where the tests will take place. AirSim offers a library called `airlib`[1], available for Python, that can be used to retrieve images taken from a simulated camera from the perspective of the drone in the simulation world. This feature will be necessary when testing the person-recognition utilities used in the program.

A high-level overview of the simulator architecture and how different components in-

[1] https://pypi.org/project/airsim/

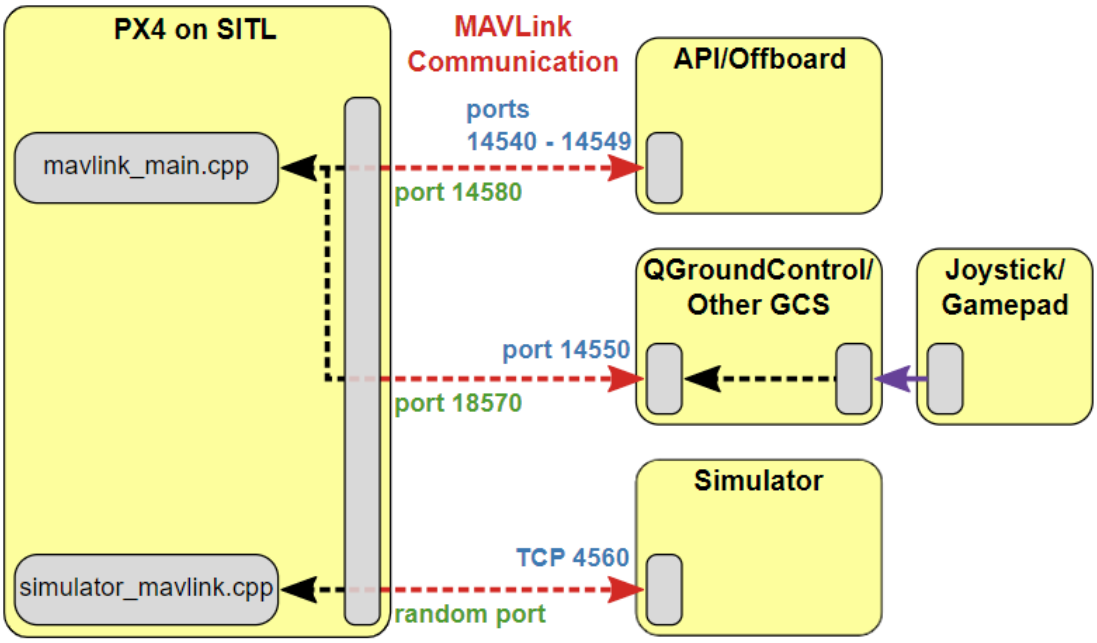


Figure 3.3: Network diagram between the components interconnecting during software-in-the-loop simulation.

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

teract can be seen in figure 3.2. The API layer in the figure inside the simulator environment refers to AirSim's own `airlib` library, which exposes high-level functionality to send control commands to the flight controller directly. However, in order to share the same control code between simulated flight and actual flight without depending on the simulator system, the control commands are sent using the official MavSDK API instead, which allows communication of estimated state, desired state and sensor data directly between the flight controller firmware and any connected systems.

In order to run the software-in-the-loop simulation, the PX4 firmware needs to be built from the source code on the Linux platform where it will run. The build system then sets up all the necessary ports for the MAVLink communication and starts a local instance of the NuttX operating system that would run on the actual flight board.

Figure 3.3 shows how the different parts of the system communicate with each other inside a SITL simulation. PX4 uses commonly established UDP ports for MAVLink communication with ground control stations (e.g. QGroundControl), offboard APIs (e.g. MAVSDK, MAVROS) and simulator APIs (e.g. AirSim, Gazebo). Externally developed applications like Dronecontrol use an offboard API, in this case, MAVSDK, and therefore listen to PX4's remote UDP port 14540 by default. All ports in the range 14540-14549 can be used to connect offboard APIs, for example, when controlling multiple vehicles simultaneously. PX4's remote UDP Port 14550 is used for communication with ground control stations, which

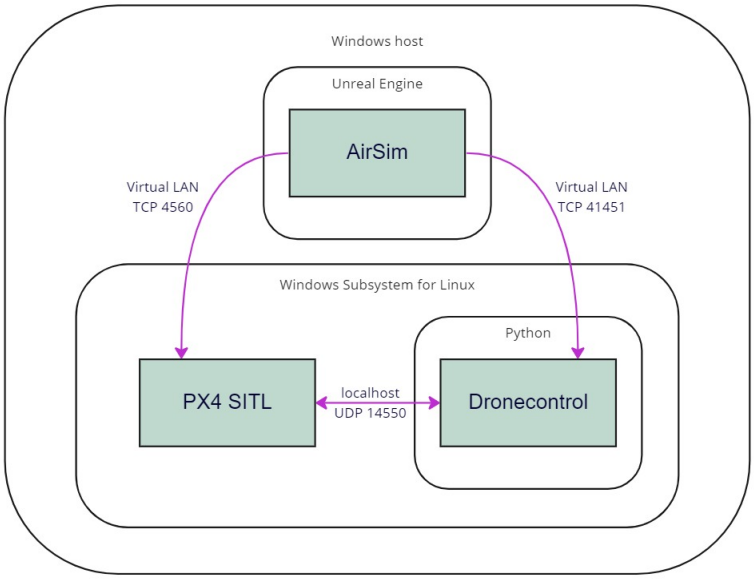


Figure 3.4: Connection diagram of how the three systems interact with each other during SITL simulation.

are expected to listen for connections on this port. QGroundControl listens to this port by default but can be configured to use the others, the same way offboard APIs can use the default ground station port to connect instead. PX4 uses a simulation-specific module to connect to the simulator's local TCP port 4560. Simulators then exchange information with PX4 using the Simulator MAVLink API shown in figure 3.1. PX4 on SITL and the simulator can run either on the same computer or different computers on the same network [20].

Since one of the purposes of using AirSim as a simulator is to run it on a Windows computer and the PX4 software-in-the-loop stack runs best on Linux, it is necessary to run a virtualized Linux OS in parallel on the Windows computer and set up a local network so that the simulator and the flight controller firmware can communicate with each other. The PX4 development team officially supports running the SITL flight stack in Windows through the Windows Subsystem for Linux (WSL2) [21], which allows users to install and run their Ubuntu Development Environment on Windows as if it was running it on a Linux computer. The Windows Subsystem for Linux lets developers run a GNU/Linux operating system (including most command-line tools, utilities, and applications) directly on Windows, unmodified, without the overhead of a traditional virtual machine or dual-boot setup [21]. The complete steps needed to configure the system are detailed in appendix A.1.

The whole set of connections established in the software-in-the-loop simulation is shown in figure 3.4 at the transport layer level. The two systems that run inside the virtualized Linux system through WSL (the simulated flight stack and the Dronecontrol program)

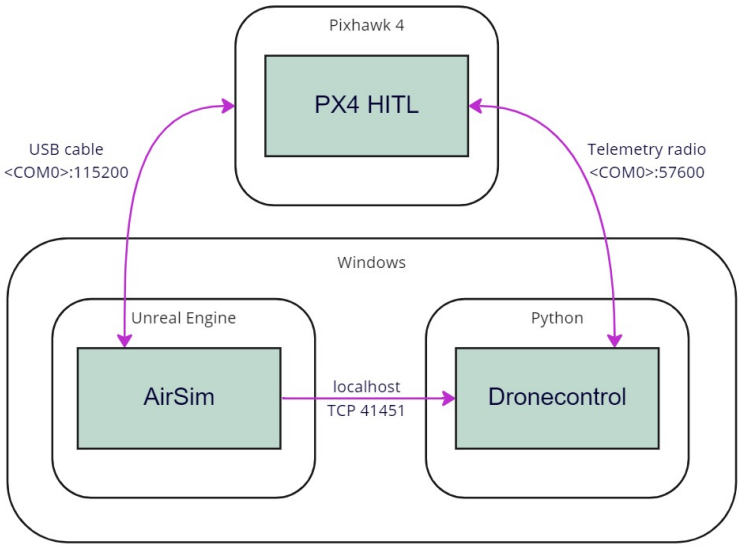


Figure 3.5: Connection diagram of how the three systems interact with each other during HITL simulation.

connect through the localhost network on the defined UDP port, and each of them connects in turn to the AirSim simulator through the virtual Local Area Network established by the Windows Subsystem for Linux to the host Windows computer. The PX4 flight stack on SITL mode connects to the simulator using the TCP port 4560, as defined by PX4 on figure 3.3, and Dronecontrol connects through the AirSim library `airlib`, which uses by default a TCP connection on port 41451.

In the case of hardware-in-the-loop simulation, the main difference with SITL is that the flight stack firmware runs on a physical flight board using a particular configuration. In HITL, all motors/actuators are blocked, but internal software is fully operational. This configuration adds another isolated system, so to simplify this mode of testing, since the WSL environment is no longer needed to run the flight stack, it is possible to move the execution of the Python modules to Windows. This eliminates the need for additional configuration to allow the external flight controller to communicate with the internal WSL network, which is isolated by default from all external USB devices. Furthermore, now that PX4 runs on a separate piece of hardware, it is necessary to establish a separate physical connection to the testing computer for each desired MAVLink channel, either wired, through the micro USB port or an unused telemetry port on the flight controller, or wireless, through a telemetry radio. Then both the simulator and the Python app can communicate with the flight controller independently.

Figure 3.5 shows the chosen connections to execute tests in HITL mode. The Windows machine runs both AirSim in Unreal Engine and Dronecontrol through the Python interpreter. They communicate through TCP in the `localhost` network. The board running

PX4 connects to the simulator through a USB to micro USB cable, which is set up to work with a baudrate of 115200, and to the Python program through a telemetry radio running at a baudrate of 57600, both attached to USB ports on the Windows computer accessible through their COM address.

AirSim testing environment

Unreal Engine is a complex computer graphics generation program. A project in the engine is defined through environments where components like 3D models, cameras, and lighting can be added. AirSim is a plugin made for this engine by Microsoft, and although it also has released a version for the other most extended game engine in the market, Unity, it is experimental and limited in features. The documentation contains all the necessary steps to get Unreal Engine and AirSim running on a Windows, Linux, or macOS operating system. However, Windows is the recommended environment [22]. The source code for the AirSim project includes a built-in Unreal environment that can be used to run tests and contains several 3D shapes like blocks and spheres, as well as a default quadcopter to act as the control vehicle. It is also possible to create custom Unreal environments and run AirSim inside them by adding the built plugin and a custom vehicle to the project.

The environment used in this project for testing the Dronecontrol application is derived from the built-in AirSim environment and can be found on the project's repository[2]. It contains the default quadcopter vehicle, which includes several virtual cameras in order to be able to retrieve images from the vehicle's point of view in the simulated world, with some modifications to the background shapes and colours to provide better contrast to the camera. The main addition to the environment is the 3D model of a human figure, to be used for testing the pose detection and tracking mechanisms in the computer vision solution. This model is part of a free asset library of human models made by Renderpeople [23] obtained from the Unreal Marketplace. Figure 3.6 shows an image of the testing environment.

Add here if any additional changes to AirSim environment

AirSim is compatible with both SITL and HITL simulation modes. However, it must be set up to work with an external PX4 flight controller instead of AirSim's own internal `SimpleFlight` flight stack. Appendix A.3 contains the required settings file for configuring these modes in AirSim, where the *"LocalHostIp"* needs to be exchanged with the IP of the Windows host in the local `vEthernet (WSL)` network to be able to connect between the simulator in Windows and the flight controller inside WSL. This process is detailed further in the AirSim documentation [24] and in appendix A.1. In brief, the Unreal project with AirSim must first be set into play mode, and then the PX4 SITL flight controller must be built and started to attach to an already running simulator. Inside the project in WSL, the

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

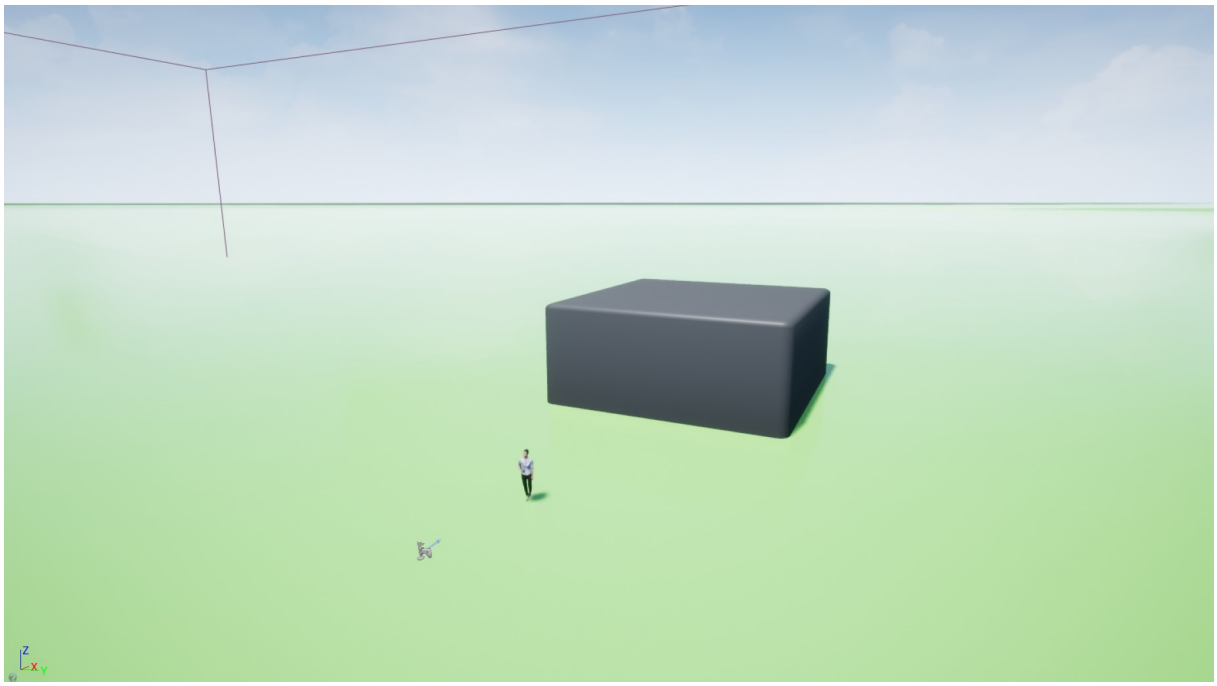


Figure 3.6: Screenshot from the Unreal Engine environment used for testing the computer vision solutions.

flight controller can be run with all the necessary configuration for AirSim by using the provided shortcut script[3]:

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

Once the simulator and the flight controller are connected, an RC controller or QGroundControl can be used to control the vehicle, and other offboard applications like Dronecontrol can be attached to send any desired MAVLink commands.

For HITL, the *"UseSerial"* setting should be set to true in the AirSim configuration, and the Pixhawk board should be connected to a USB port in the computer. Additionally, the board should be set to HITL mode enabled from the QGroundControl safety configuration. Then, the simulator will attach automatically to the board after starting play mode in Unreal. 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] https://github.com/l-gonz/tfg-giaa-dronecontrol/blob/main/simulator.sh

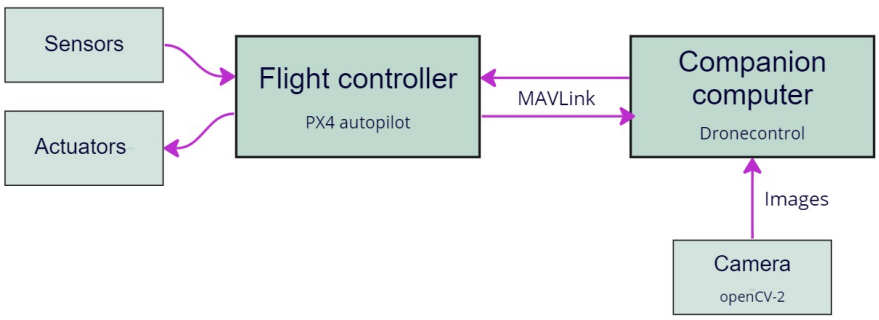


Figure 3.7: Top-level diagram of the hardware/software interactions

3.2 System architecture

3.2.1 Top level

The purpose of the Dronecontrol application is to be able to direct the movement of a UAV through the analysis of the images taken by a camera. Since the processing power needed to analyse the images recorded is superior to that offered by the autopilot flight controller, it becomes necessary to use an additional companion computer that can control the camera and leverage machine learning algorithms to extract useful features from the images and transform them into movement directives for the vehicle.

A top-level diagram of the individual parts that comprise the system is shown in figure 3.7. The main elements are the flight controller, which will run the PX4 autopilot 2.2.1, the companion computer, which will run the developed application, and the camera, which provides the images. The flight controller interfaces directly with the companion computer using the MAVLink protocol described in section 2.2.1 through a wireless radio link or a cabled serial connection between the two. The camera is connected to the companion computer through a USB cable into any available port on the computer. Typically, the type of connection between the flight controller and the computer depends on the system's desired setup. There are two main possibilities. When the images have to be taken from the vehicle's perspective and move with it, the camera, and therefore the companion computer, flies onboard the vehicle next to the flight controller. In this case, using a direct wire connection between the two is the most convenient, as it provides a faster and more stable link. This onboard configuration is detailed in section 3.2.2. The second possibility is that the companion computer acts more like a ground station, directing the vehicle's movement from the ground on an offboard configuration while the flight controller stays onboard the vehicle. This configuration is possible when the camera does not need to move

with the vehicle. Then, it becomes strictly necessary to communicate with the flight controller through a wireless connection, using the pair of telemetry radios provided with the Development Kit of the Holybro X500 (2.2.2). Section 3.2.3 describes the complete setup required for this configuration.

In this project, the flight controller is driven by the PX4 flight stack (2.2.1), and the hardware employed on the autopilot board is optimised for this software. PX4 uses sensors to determine the vehicle state, which is needed for stabilisation and to enable autonomous control. It minimally requires a gyroscope, accelerometer, magnetometer (compass) and barometer. A GPS or other positioning system is also needed to activate all fully automatic flight modes, like planned missions, and some assisted ones, like altitude stabilisation. PX4 uses outputs to control motor speed, flight surfaces like ailerons and flaps, camera triggers, parachutes, grippers, and many other types of payloads. Most PX4 drones rotate the propellers through brushless motors that the flight controller controls via an Electronic Speed Controller (ESC). The ESC converts a signal from the flight controller to an appropriate level of power delivered to the motor. They are most commonly powered by Lithium-Polymer (LiPo) batteries, typically connected to the system using a Power Module or Power Management Board, which provides separate power for the flight controller and the ESCs for the motors. To manually control the vehicle, a Radio Control (RC) system is used. It consists of a remote control unit that uses a transmitter to communicate stick and control positions to a receiver located on the vehicle. More complex RC systems can also receive telemetry information from the autopilot to display for the operator. Other ways of communicating with the autopilot include telemetry radios, which can provide a wireless MAVLink connection between a ground control station and a vehicle running PX4. This makes it possible to tune parameters while a vehicle is in flight, send flight mode commands, inspect telemetry information or change a mission on the fly.

On an actual UAV, the PX4 software runs on a dedicated piece of hardware like the Pixhawk 4 flight controller described in section 2.2.2 that includes all the minimal required sensors for flight as well as interfaces to connect additional actuators and I/O systems (RC, telemetry radio). However, it is also possible to simulate this hardware on a standard Linux computer by building the flight stack from the source code. This process is described in section (3.1).

The Dronecontrol application that runs on the companion computer has been developed using the Python programming language[4]. It offers good advantages for a project of these characteristics because of its high-level, easy-to-use syntax that usually results in a smaller code base than other comparable languages for small projects, its versatility and its support for object-oriented programming. Most importantly, Python is widely used with an official package manager called `pip` that significantly simplifies the use of external libraries and which gathers in its package index[5] thousands of well-tested utilities, including many

[4] https://www.python.org/
[5] https://pypi.org/

machine learning and image processing projects. Moreover, all the necessary libraries for interacting with PX4 through the MAVLink protocol, object detection and tracking, and simulation (MavSDK, OpenCV, AirSim, Mediapipe) offer a version to use with Python. As an interpreted language, it can run without complications in any system with Python installed without compiling separate binaries for different operating systems.

The following sections explore deeper into the differences between the two configurations mentioned before: offboard and onboard companion computer.

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 it can stay on the ground while the vehicle flies. This has the advantage that it allows for a more straightforward configuration without having to be concerned with low-level hardware interactions between the two systems or providing power to the companion computer while in flight. It also facilitates using a more powerful computer for image processing since there is no need to account for any added weight to the vehicle. However, the camera remains connected to the computer on the ground, so the images will not be taken from the drone's perspective in flight, which limits the real-world applications of the system. Other configurations involving a direct connection from a camera to the flight controller and the transmission of its images wirelessly to the companion computer through MAVLink for offboard processing can be feasible with the current technology but fall out of the scope of this project.

In this instance, the wireless link is established through a pair of telemetry radios connecting to a telemetry port on the flight controller and to a USB port in the companion computer. Since the Pixhawk 4 is configured by default to use its `TELEM1` port for this purpose, no additional parameter configuration is needed when using that port. To connect to the board from the companion computer, applications like the QGRoundControl (2.2.1) ground station software can automatically detect a telemetry radio inserted into any USB port on the host computer. Additionally, other applications using the MavSDK (2.2.1) library can establish a connection by specifying the baudrate and the USB serial port address, usually something similar to `/dev/ttyUSB0` on Linux and `COM1` on Windows.

The radio used for the physical tests in this project is the Holybro SiK Telemetry Radio[6]. It is a small, light and inexpensive open-source radio platform that typically allows ranges of more than 300 meters "out of the box" (the range can be extended to several kilometres with a patch antenna on the ground). The radios are offered as 915Mhz (Europe) or 433Mhz (US), so they can be used in different regions and comply with the regulations for

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

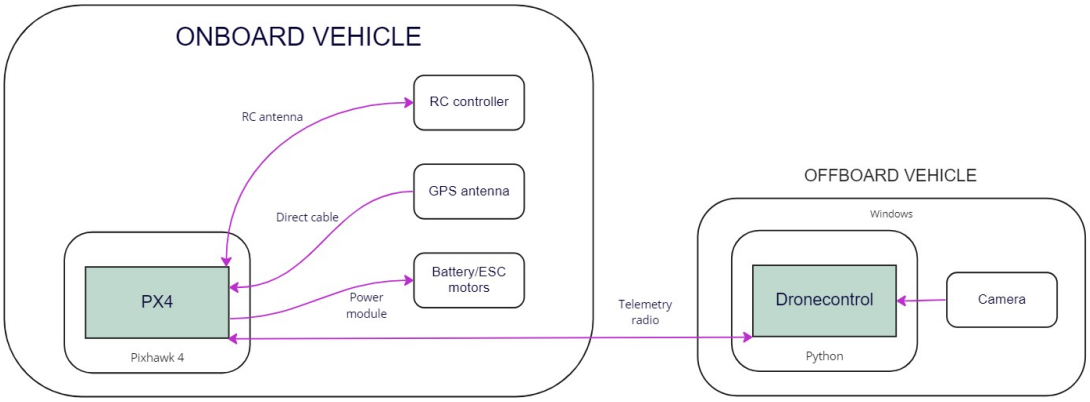


Figure 3.8: Offboard configuration connections

frequency, hopping channels and power levels. They offer 2-way full-duplex communication through an adaptive TDM UART interface, and their antenna allows for an adjustable 100-mW-maximum output power and -117 dBm receive sensitivity. The link is established by default with a baudrate (max bits per second on a serial channel) of 57600, and it can provide air data rates of up to 250 kbps.

Figure 3.8 summarises the connections to establish between the PX4 software, the Dronecontrol application, their hardware platforms, and their respective peripherals both onboard the vehicle and at the ground station for the offboard computer configuration setup.

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. In this case, the connection is made through a serial cable between the telemetry port in the flight controller and a serial port in the companion computer. The camera will then be connected through a cable as well to the companion computer and attached to the frame of the vehicle 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 that reflect the trajectory that it follows. Therefore, it becomes possible to adjust the control loop based on previous reactions of the vehicle to commands and establish a feedback loop to maintain a stabilised output.

Since, in this configuration, the computer running the visual processing algorithm has to fly on board the vehicle, making a good choice when selecting hardware becomes essential. To be able to take into the air, the computer has to be light enough that its weight can

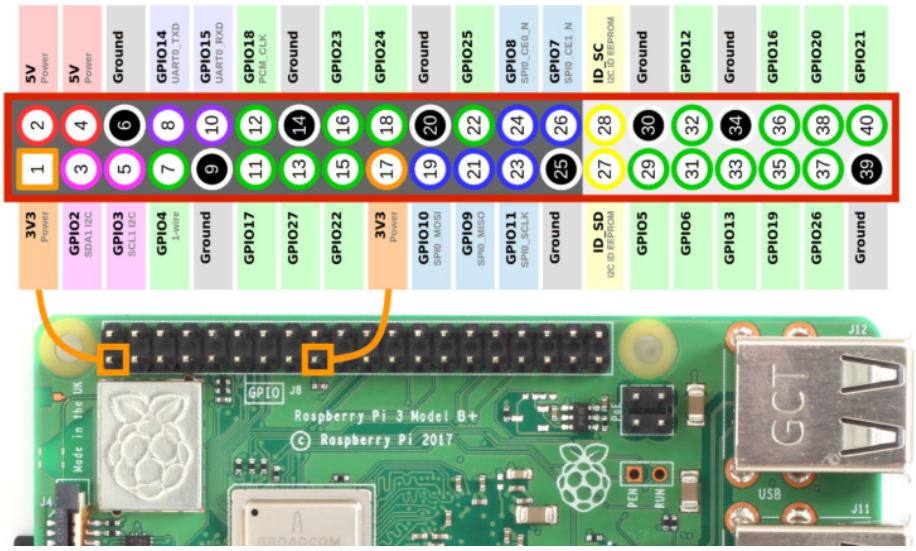


Figure 3.9: The Raspberry Pi 4 microcomputer, with its 40-pin GPIO header marked in red and annotated pinout.

Source: Adapted from *Raspberry Pi GPIO Header with Photo* [25]

be lifted by the propellers while maintaining adequate battery autonomy but also powerful enough that the processor can handle the computer vision algorithms required to extract the necessary features from the images taken from the onboard camera that are to be fed to the control loop.

The Raspberry Pi 4 model chosen for this project and shown in figure 3.9 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's officially supported operating system, called Raspberry Pi OS, is a Debian-based version of Unix optimised for its ARM microcontroller, which simplifies the process of moving from the Ubuntu test environment into the handheld computer. Since this computer is designed to be easy to integrate with hardware projects, it includes a 40-pin GPIO header (highlighted in figure 3.9) that can be programmed for connecting to any number of external devices.

The Raspberry Pi is powered by a 5V input that can be provided either from its USB-C port or through one of two pins in the header dedicated to this purpose (marked as "5v Power" on figure 3.9). In the case of the particular vehicle build used in this project, the power management board supplying the energy (the Holybro PM07[7]) also provides 5V to the flight controller, as well as powering the ESCs to the motors. It counts with two

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

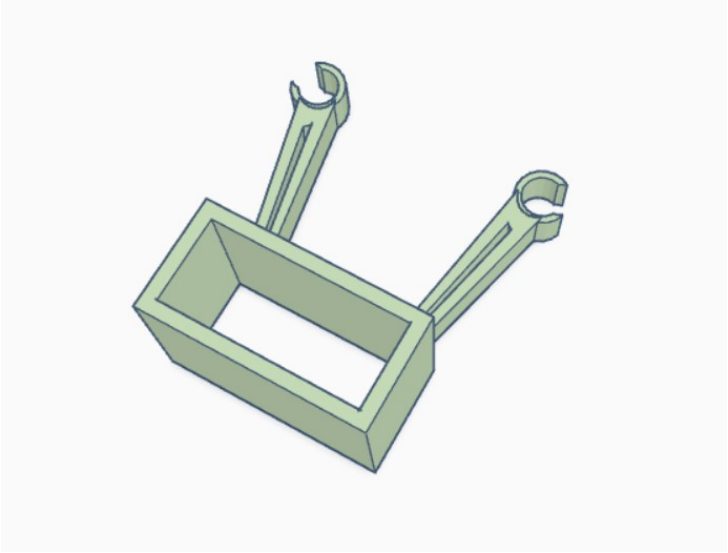


Figure 3.10: 3D model for the camera mount designed for the Holybro X500 frame.

power outputs: one connected to the flight controller's `POWER1` port and another unused one. A first attempt at supplying current to the Raspberry Pi from the main battery was tested initially with a connection between the second, free output on the power module and the powering pins on the GPIO header with a custom connector. However, the consequent power supply ended up being too unstable for the Pi board, resulting in frequent dips in the supplied current that affected the processing capabilities of the companion computer. The devised solution was to add a secondary battery to the vehicle providing power through a USB to USB-C cable. This configuration allows the Raspberry to receive power through its default power input, the regulated USB-C port. The main disadvantage of the solution is that it adds one more piece of equipment of substantial weight that needs to be secured to the vehicle's frame and carried into the air while in flight.

In contrast to the choice of a companion computer, many more possibilities are available regarding the camera that will fly onboard the vehicle. The most important characteristics to focus on are a small weight and simple plug-and-play functionality with the onboard computer. The camera used in the tests carried out and detailed in section 4 is a Logitech C920 1080p webcam[8]. Since the frame of the Holybro X500 is not prepared by default to include an onboard camera, a custom mount has been designed and 3D-printed from PLA plastic to be able to hang the camera from the central rods on the underside of the vehicle frame and ensure that it is attached securely during flight. The 3D model of the mount can be seen in figure 3.10, and the print-ready file can be found in the project repository in GitHub[9].

The second wired connection that needs to be established for this configuration is between the flight controller and the companion computer so the MAVLink messages can be ex-

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

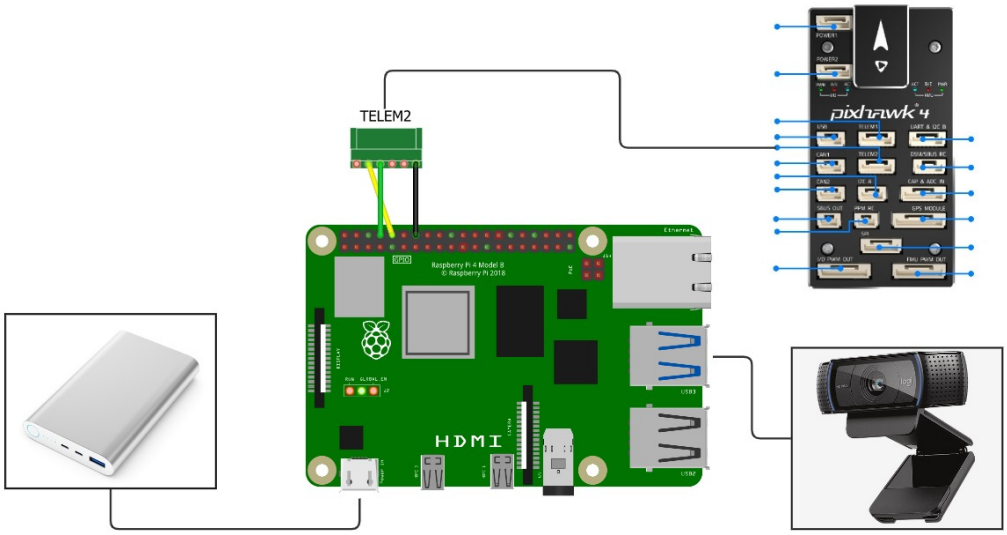


Figure 3.11: A diagram of the wired connections from the Raspberry Pi 4 to the secondary battery and the flight controller (`TELEM2`).

changed. As it is desirable to be able to maintain a wireless link to the vehicle even while the onboard computer is controlling it, the telemetry radio is kept connected to the `TELEM1` port of the flight controller. Then, the companion computer is wired to the secondary telemetry port, `TELEM2`. This port is not configured to be used by default, so it is necessary to modify the configuration of the Pixhawk board either through QGroundControl or the PX4 console to set it up for a companion computer. The main parameter that requires changing in the board is the `MAV_1_CONFIG`, which configures the serial port on which a second instance of MAVLink is run (the primary instance is configured with `MAV_0_CONFIG`, set up for telemetry radios). This parameter defaults to 0 (disabled) and should be set to 102 to map to `TELEM2`. The secondary MAVLink instance is configured for "Onboard mode" by default, which is the appropriate mode for communicating with a companion computer, instead of the "Normal mode" running on the primary MAVLink instance through `TELEM1` to communicate with QGroundControl. Another parameter to consider is the `SER_TEL2_BAUD`, which regulates the baud rate of the `TELEM2` port. The default rate is 921600, which is appropriate for a serial connection. This rate needs to be specified when establishing the connection in the Dronecontrol application through the MAVSDK library. PX4 provides a complete overview of all the available parameters for the board configuration in their documentation [26].

The other end of the connector has three female Dupont wires that go into the TX/RX UART pins of the Raspberry Pi, according to mapping table 3.1 between the six pins in the telemetry connection of the Pixhawk board and the corresponding GPIO pins in the Raspberry Pi header [27] [28]. A diagram of all the connections to the companion computer can be seen 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.

Compared to the default baudrate of 57600 on the telemetry radio link established in the previous section, the wired serial connection works at a baudrate of 921600, which means that data can be transferred up to 16 times faster through this link. However, the main limitation on speed for the program comes from the feature detection process that runs on the images, so a faster link rate will not necessarily mean any increase in the solution's overall performance. In section 4.3.2, different combinations of hardware are compared to analyse any challenges to the program's performance.

The offboard configuration allowed the supervision of the output from the program while the vehicle was flying since the computer stayed stationary on the ground. However, having a screen connected directly to the companion computer is not feasible in this configuration. To fix this situation and be able to monitor the execution during flight, as well as being able to provide input directly to the program, it is possible to make use of the WiFi antenna of the Raspberry Pi to configure a remote desktop and connect to it through another computer serving as a ground station. Then, the camera output and image recognition can be seen in real-time.

Figure 3.12 shows a summary of all the connections present in the onboard configuration between the three pieces of software that interact together: PX4, Dronecontrol and QGroundControl, with their respective hardware platforms and the attached peripherals.

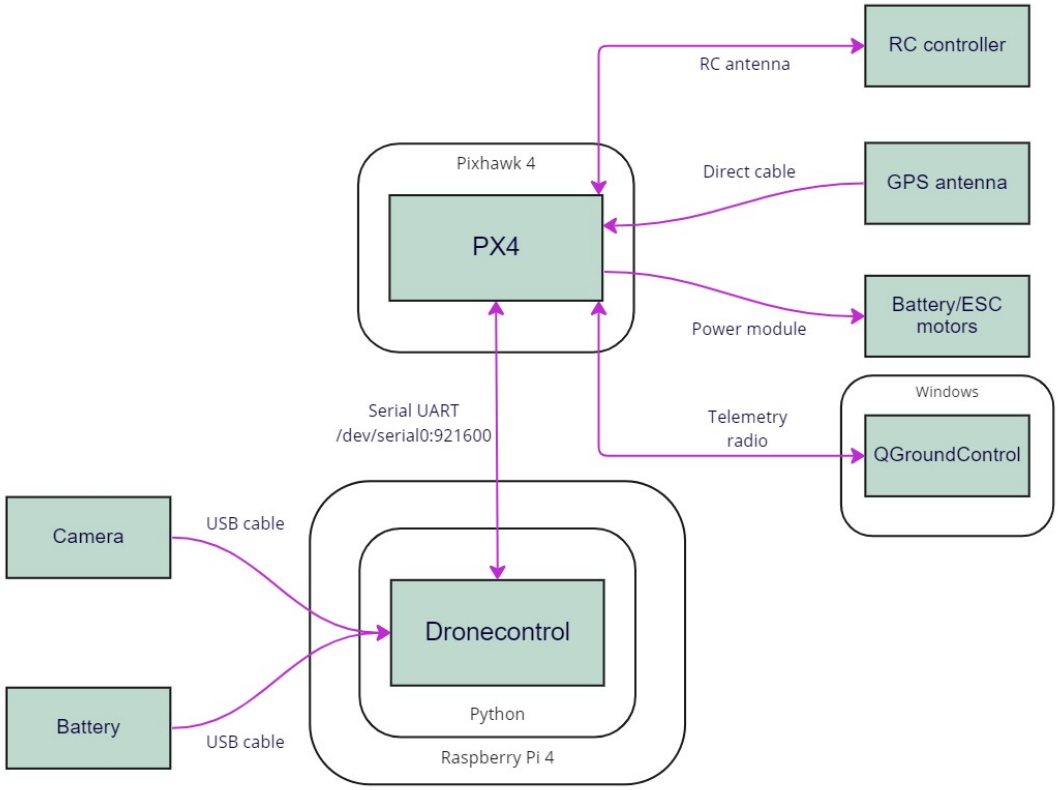


Figure 3.12: Onboard configuration connections

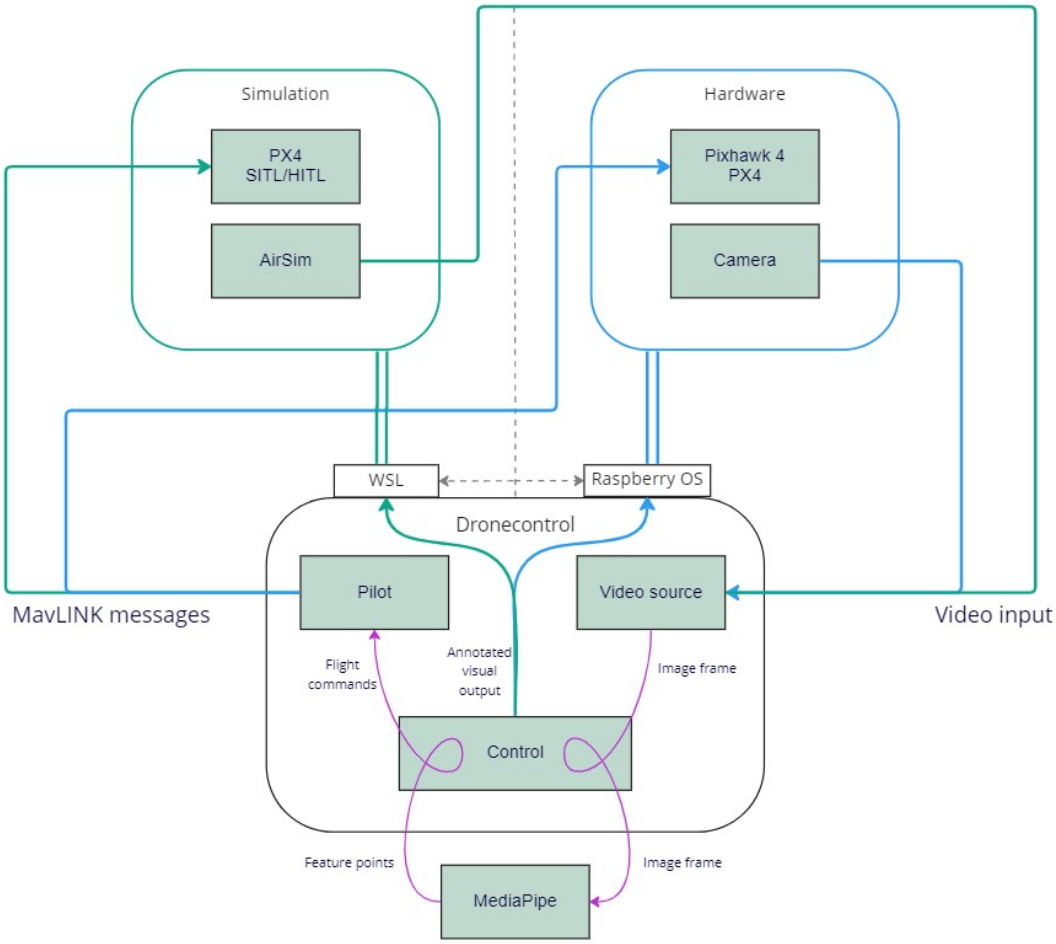


Figure 3.13: Structure of the Dronecontrol application and its interactions with the necessary additional software for running in a simulation (green) or a vehicle (blue).

3.3 Software architecture

Figure 3.13 shows the main modules of the designed software and how it interacts with the external libraries. The application consists of three fundamental parts: the **pilot module**, in charge of sending instructions to the flight controller and receiving back information on position and state through the `mavsdk` library, a **video source module** that handles the retrieval of images from different sources and the necessary processing for image analysis, and a **control module** that directs the interaction between the other two to transform the pixel information first into position points through the `mediapipe` library and then into instructions for the pilot. The upper part of the diagram in figure 3.13 shows the flow of information between the Dronecontrol application and the external systems, with green lines representing its path in a simulated workflow and blue ones indicating the alternative path for the information in a system with actual quadcopter hardware. Purple arrows

show the input/output of each module within the developed application and how they interconnect. Other smaller utilities have also been developed to help test how the systems interact with each other and calibrate different parts of the control behaviour. These are described in sections 3.3.3 and 3.5.1. A user manual with all the options available in the application can be found in Appendix B.

3.3.1 Pilot module

The purpose of the pilot module[10] is to provide access to the rest of the application to send and receive messages from the PX4 controller through the MavSDK library. This library provides a simple asynchronous API for managing one or more vehicles, providing programmatic access to vehicle information and telemetry, and control over missions, movement and other operations. MavSDK utilizes the Python library `asyncio` to run coroutines in parallel while waiting for the messages provided through the MAVLink communication. Therefore all calls to the library have to be written as async functions that await the result of one or more polls to the flight stack.

`asyncio` provides support for writing concurrent code using the `async/await` syntax. It is used as a foundation for multiple Python asynchronous frameworks that provide high-performance network and web servers, database connection libraries or distributed task queues. It provides a set of high-level APIs to run Python coroutines concurrently and have complete control over their execution. The pilot module integrates `mavsdk` and `asyncio` and provides a queue for the control module to send actions to be executed in the vehicle one after another.

Listing 3.1 shows how to establish a connection to a PX4 vehicle through its physical serial address or virtual UDP one and poll for internal information from the flight controller to decide when the system is ready to receive instructions. The MAVSDK library exposes telemetry and other state information through asynchronous generators, which are defined in python as a convenient way to make asynchronous data producers and accessed with the `async for` syntax.

Many of the basic operations that can be executed in the flight controller are implemented in the pilot module, along with error handling and safety checks, like takeoff, landing, return home or manipulating the vehicle flying velocity directly by providing speeds in body coordinates. These actions can be executed directly or added to a queue that is polled periodically to run them in the order that they are added, waiting until the previous action has finished and the vehicle is in the desired state before starting the next. The loop that runs this queue can be seen in listing 3.2. There is a maximum time of 10 seconds that each

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

```
async def connect(self):
    """Connect to mavsdk server.
       Raises a TimeoutError if it is not possible to establish connection.
    """

    if self.serial:
        address = f"serial://{self.serial}"
    else:
        address = f"udp://{self.ip if self.ip else ''}:{self.port}"
    self.log.info("Waiting for drone to connect on address " + address)
    await asyncio.wait_for(self.mav.connect(system_address=address),
    ↪  timeout=self.TIMEOUT)

    async for state in self.mav.core.connection_state():
        if state.is_connected:
            break

    # Wait for drone to have a global position estimate
    async for health in self.mav.telemetry.health():
        if health.is_global_position_ok:
            break

    self.log.info("System ready")
    self.is_ready = True
```

Listing 3.1: Example of how the communication to the flight stack is established through `asyncio` and the MAVSDK library

action can use to run. The loop stops when the asynchronous task it runs on is cancelled with `task.cancel()`, which raises a `CancelledError` exception in the execution thread.

Polish: consider removing code or exchanging for diagrams

3.3.2 Video source module

The objective of the `video_source` module[11] is to provide a collection of classes to retrieve images from different sources, in a way that they can be exchanged for one another without affecting the rest of the application to facilitate testing and be adaptable to running in different environments. Three classes of video sources have been implemented: file, simulator, and camera, which inherit from the same `VideoSource` according to the diagram in figure 3.14.

The `FileSource` class is able to open a video file stored in the companion computer

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

```
async def run_queue(self):
    """
    Run the queue loop.

    Queued actions will be awaited one at a time
    until they are finished.
    The loop will sleep if the queue is empty.
    """
    try:
        while True:
            if len(self.actions) > 0:
                action = self.actions.pop(0)
                self.log.info("Execute action: %s", action.func.__name__)
                try:
                    await asyncio.wait_for(action.func(self, **action.args), timeout=10)
                except asyncio.exceptions.TimeoutError:
                    self.log.warning(f"Time out waiting for {action.func.__name__}")
            else:
                await asyncio.sleep(self.WAIT_TIME)
    except asyncio.exceptions.CancelledError:
        self.log.warning("System stop")
```

Listing 3.2: Loop where the action queue runs on the pilot module. Each action is awaited until it finishes or the timeout time runs out.

and provide images taken frame by frame from it until the video completes. This allows the program to replay the image detection algorithms on videos previously captured by the camera tool described in section 3.3.3. The `CameraSource` class can access a physical camera attached to the computer running the application via USB and provide the captured frames in real time. Both the file and camera sources make use of OpenCV's video capture utilities to take care of the file handling and the camera drivers necessary.

The `SimulatorSource` uses AirSim's Python library `airlib` to communicate with the simulator and retrieve images from a camera object attached to the model of the vehicle in Unreal Engine. It connects automatically through `localhost`, but it can also be initialized with an IP to establish a connection to the simulator running on a different computer on the local network; for example, when the Dronecontrol runs inside WSL and the simulator runs on the host computer.

3.3.3 Vision control module

The control module contains the main logic of the application and is in charge of converting the raw images obtained from the video source into commands for the pilot module. Two different types of control have been implemented. The first one is the proof-of-concept control solution described in section 3.4 that operates in the offboard configuration outlined in

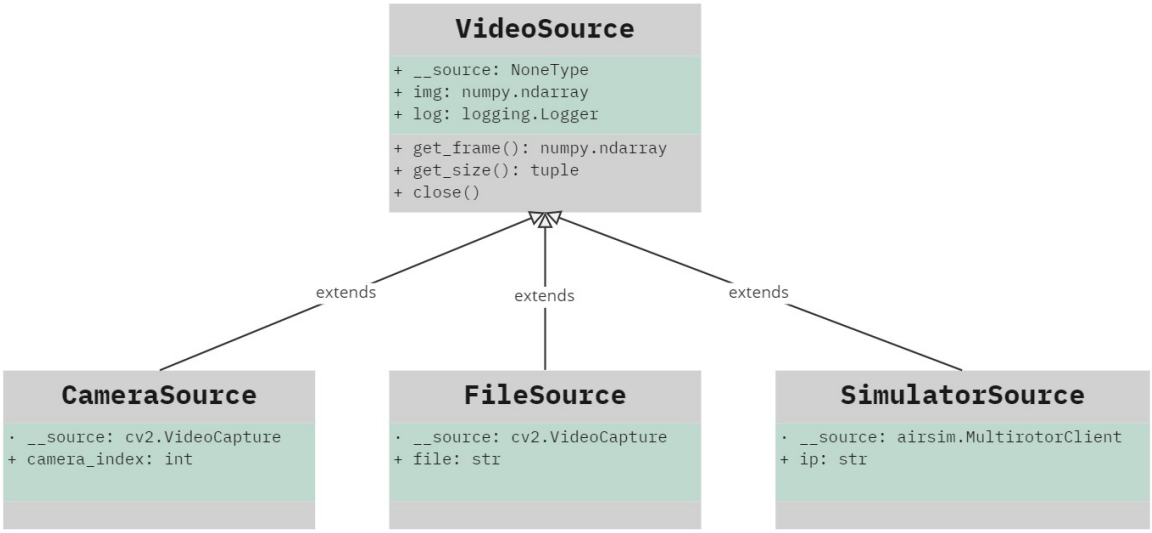


Figure 3.14: Diagram of inheritance on the video source classes available to retrieve image data.

section 3.2.2 and translates predefined hand gestures into commands for the aerial vehicle. The primary purpose is to be able to test the interaction between all the system components in an easily controlled environment thanks to using the more straightforward configuration of situating the controlling computer outside of the vehicle. The second control system consists of a follow solution which attempts to reproduce an environment closer to real-life scenarios, where the control algorithms and the camera are onboard the vehicle. It functions by tracking the location of a person detected in the images obtained from the drone's perspective, which is used to calculate the velocities required to follow said person and maintain it centred in its view.

The process followed in both solutions consists roughly of the same two parts. First, the image is sent to the computer vision and machine learning third-party library (MediaPipe) that extracts the required features from the image in the form of 2D coordinates. Afterwards, various calculations depending on the particular solution are applied to these coordinates to decide which commands are sent to the pilot module. The third-party library used for computer vision in the program is the MediaPipe library described in section 2.2.1, which offers cross-platform, customizable machine-learning solutions appropriate for live and streaming media.

To engage the module in direct control of the vehicle's velocity, it is necessary to use a special flight mode defined by PX4 for this purpose, called Offboard mode[12] (not to be confused with the offboard configuration described in section 3.2.2). Offboard mode primarily controls vehicle movement and attitude and supports only a minimal set of MAVLink messages. This mode requires position or pose/attitude information to be available to the flight

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

controller, for example, through a GPS antenna. In it, the vehicle obeys a position, velocity or attitude setpoint provided over MAVLink by a MAVLink API (i.e. MAVSDK) running on a companion computer and usually connected via serial cable or wifi. The vehicle must receive a stream of setpoint commands at a rate higher than 2Hz before engaging the mode to remain in it. If the message rate falls below 2Hz or the connection is lost, the vehicle will stop and, after a timeout, attempt to land or perform some other failsafe action according to the parameters configured.

Sections 3.4 and 3.5 further explain the control modules used by the two different solutions developed.

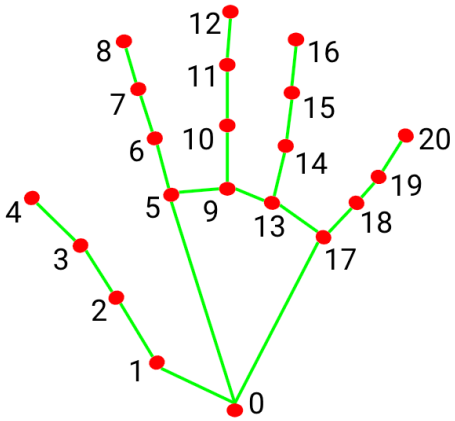
Camera-testing tool

In addition to the main modules, several utilities have been added to the Dronecontrol program to facilitate the development and testing process of the control solutions. The first tool is found on the `test_camera` module[13] and accessed through the command `dronecontrol tools test-camera`. It can be used to test the connection between the computer, the flight stack and the camera without using any self-guided control mechanism, as well as to evaluate the performance of the MediaPipe hand and pose machine learning solution on real-time images. It is also possible to capture images and record video from the live camera feed for later analysis. Through the command-line options, the test tool can be configured to use any of the three available video sources: camera, simulator or any video file; to connect to an optional hardware or simulated PX4 flight controller by specifying a connection string or IP, respectively, to run either in a host computer or a WSL subsystem; Additionally, computer vision can be enabled to process incoming images through the hand or pose recognition software. While the tool is running, the keyboard can be used to send basic commands to a connected vehicle like takeoff, landing or moving along any of its three axes. A full breakdown of all the tool's options can be found in appendix B.

3.4 Proof of concept: hand-gesture solution

The main purpose of this solution is 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 in flight with the minimal setup of a PX4-driven drone with its default components and any computer with a camera.

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



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

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

Source: Adapted from *Hands - mediapipe* [29].

This control module for this solution is the `mapper` module[14]. 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 from MediaPipe [30]. If a hand is detected in the images, 2D coordinates are extracted according to the map in figure 3.15. These landmarks are then converted into discrete gestures like an open palm, closed fist or a specific finger pointing in different directions. When a new gesture is detected, the command assigned to it is queued to the pilot module and executed as soon as the previous commands end their execution.

The conversion between landmarks and gestures is performed by the `gestures` module[15], drawing vectors from the base of each of the fingers to their tips as well as from the base of the hand (wrist feature) to the base of the fingers and using the dot product vector operation to calculate the relative angles between each finger and the base of the hand, as shown in figure 3.16. By comparing the calculated angles to a threshold, it is possible to detect whether each individual finger is extended or folded, as well as the general direction toward which it points. The open hand gesture, for example, can then be defined as all five fingers extended, that is, all five vectors defined by the fingers sharing the same approximate angle with the vector from the base of the hand to that finger.

The full list of gestures detected by the program by calculating these angles is shown in figure 3.17 and is as follows:

- 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.

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

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

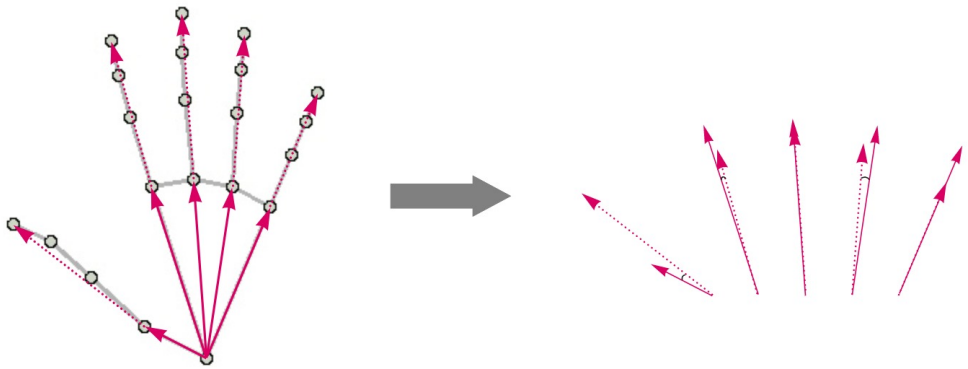


Figure 3.16: Vectors extracted from the detected features to calculate the angles between them and determine their relative positions.

- Open hand: it is detected when all five fingers are extended, as if gesturing stop. In this case, it makes the drone purposefully hold at its current position.
- Fist: it is detected when all five fingers are folded, forming a fist. It makes the drone arm and take off if it is on the ground or land if it is already in the air.
- Backhand: it is detected when the back of the hand is shown towards the camera, with the thumb pointing upwards and the other fingers pointing to the side, and makes the drone go into "return" flight mode, where it climbs to a configured, safe altitude and returns to the last takeoff position.
- Index finger pointing up: it is detected when the index finger is extended and pointing roughly towards the top of the image (± 30 degrees). It makes the drone go into "offboard" mode, where it is possible to receive direct velocity commands. The drone will remain in this mode while this finger is kept extended, and its movement can be controlled by adding one of the following four commands.
- Index finger pointing to the right: same as above but pointing to the right of the image, between 30 and 90 degrees from the top. It makes the drone roll towards its right side at a speed of 1 m/s.
- Index finger pointing to the left: same as above, but as the finger points to the left, so does the drone roll towards its left side.
- Thumb pointing to the right: it is detected when the index finger is extended up (to maintain the drone in offboard mode) and the thumb is folded, 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: same as the previous gesture, but the drone pitches backwards when the thumb points to the left of the screen.

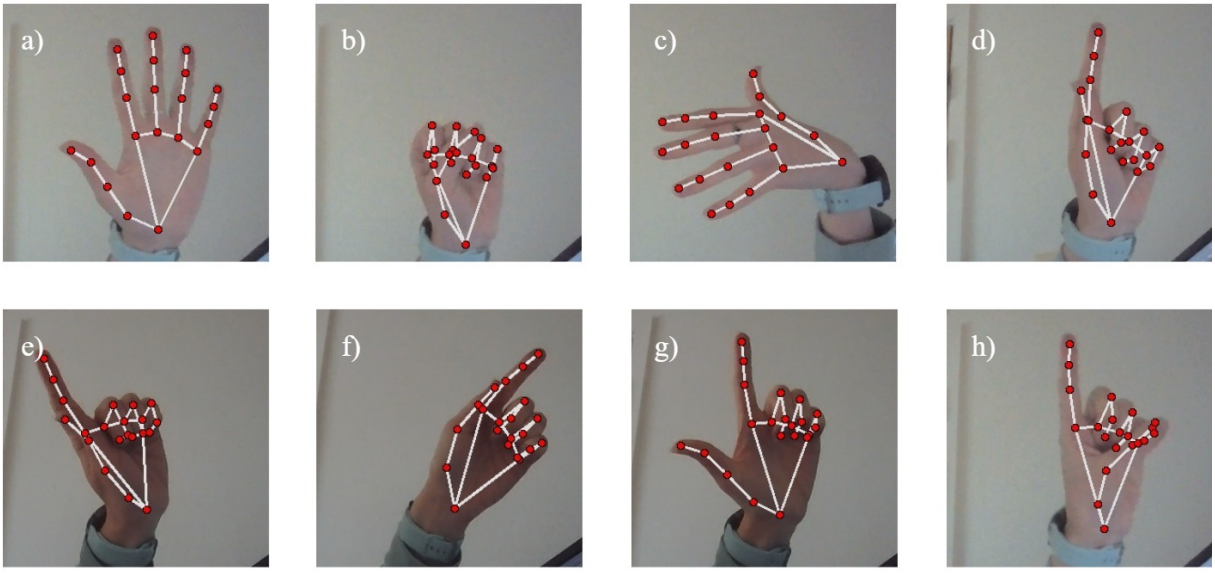


Figure 3.17: Gestures detected by the program to control the drone's movement. a) Open hand, b) fist, c) backhand, d) index point up, e) index point left, f) index point right, g) thumb point left, h) thumb point right

The program execution is outlined in figure 3.18. After all the necessary options have been set, a new thread is started to run the pilot queue loop detailed in 3.3.1, which waits for new commands. On the main thread, the GUI loop runs endlessly until the user quits the program, queuing actions based on gestures calculated from retrieved images and on user input on the keyboard, according to the mapping defined in the `input` module[16].

3.5 Final solution: human following

The intention behind developing a UAV control solution that implements tracking and following of humans is to show how the PX4 open-source development platform and its related projects, MAVLink and MAVSDK, can be used to design complex real-life applications without the need for expensive state-of-the-art proprietary hardware. The only requirements of the follow application are a PX4-enabled flight controller installed in an aerial vehicle, a companion computer of appropriate dimensions that can be mounted on board the vehicle and any camera that can be connected to the companion computer via USB. During the 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 secure shell using the SSH protocol.

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

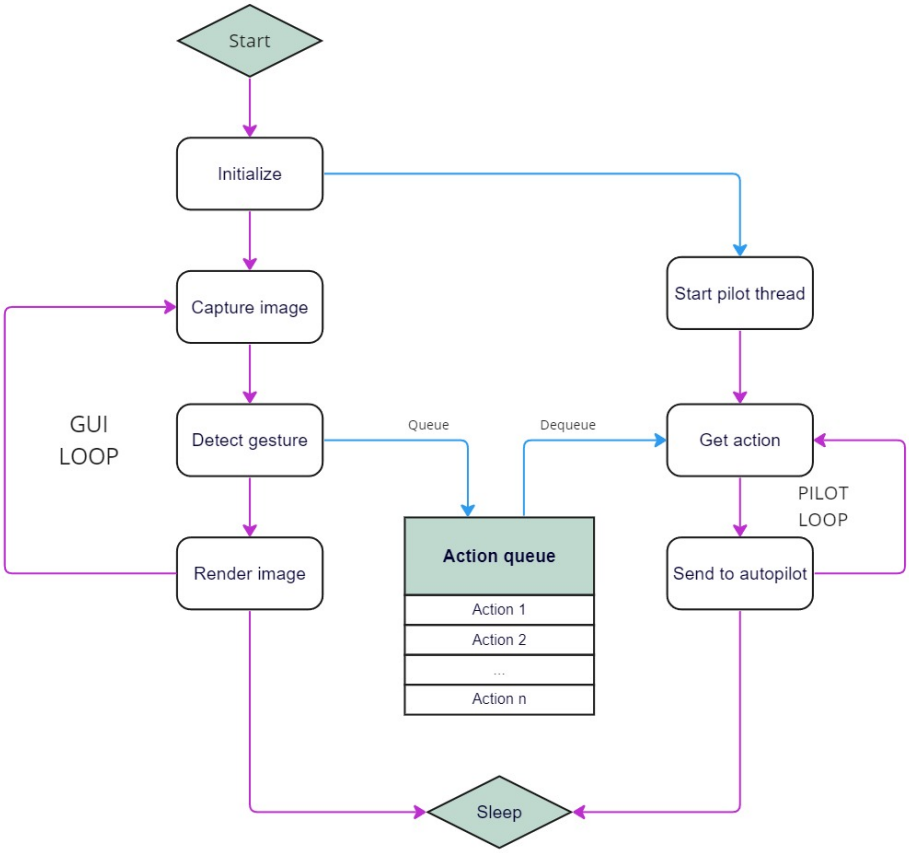
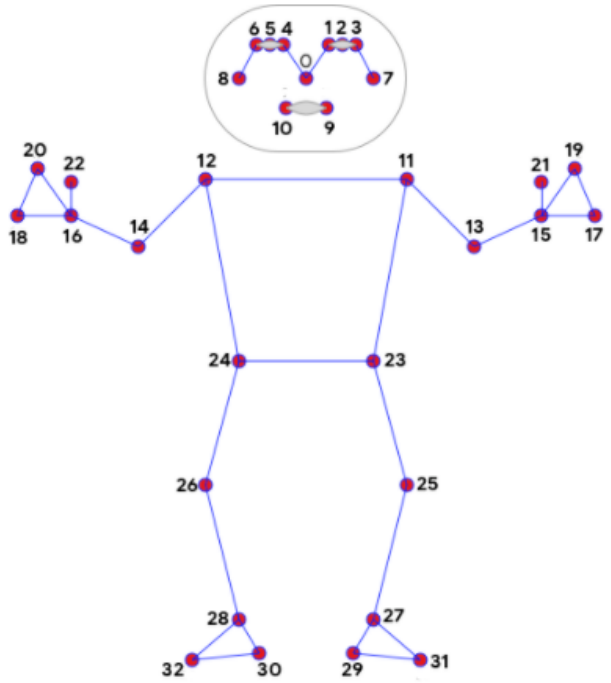


Figure 3.18: Execution flow for the running loop in the hand-gesture control solution.



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

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

Source: Adapted from *Pose - mediapipe* [31]

For safety, the follow mechanism only engages when the flight mode on the vehicle is changed to offboard mode, for example, by activating a configured switch in the RC controller, and stops automatically if the connection to the computer is lost or any of the configured fail-safes are triggered, like low battery, loss of RC or GPS signal or vehicle attitude exceeding a predefined pitch and roll value for longer than a specified time. In this mode, the vehicle will attempt to find a single person in its field of view and follow their movements by changing its yaw and forward velocity to match horizontal movements and distance changes, respectively.

During the program execution and while the offboard mode and the follow mechanism are engaged, the system continuously retrieves images from the offboard camera that are fed into the MediaPipe Pose [32] computer vision library to extract pose landmarks from it in the form of 2D coordinates. Figure 3.19 shows the features extracted by the algorithm and their correspondence to the human body. These coordinates are used to draw a bounding rectangle around the detected person and to validate that the received landmarks match the general pose of a person standing up. Figure 3.20 shows some examples of the validation checks running on the raw landmarks extracted from an image. To prevent unwanted movements, the vehicle will always stop and hover whenever it becomes impossible to detect a person in the image received or its features do not match the expected geometry. After a valid bounding box has been defined around the target person, its position with respect to the camera's field of view is sent to a control mechanism composed

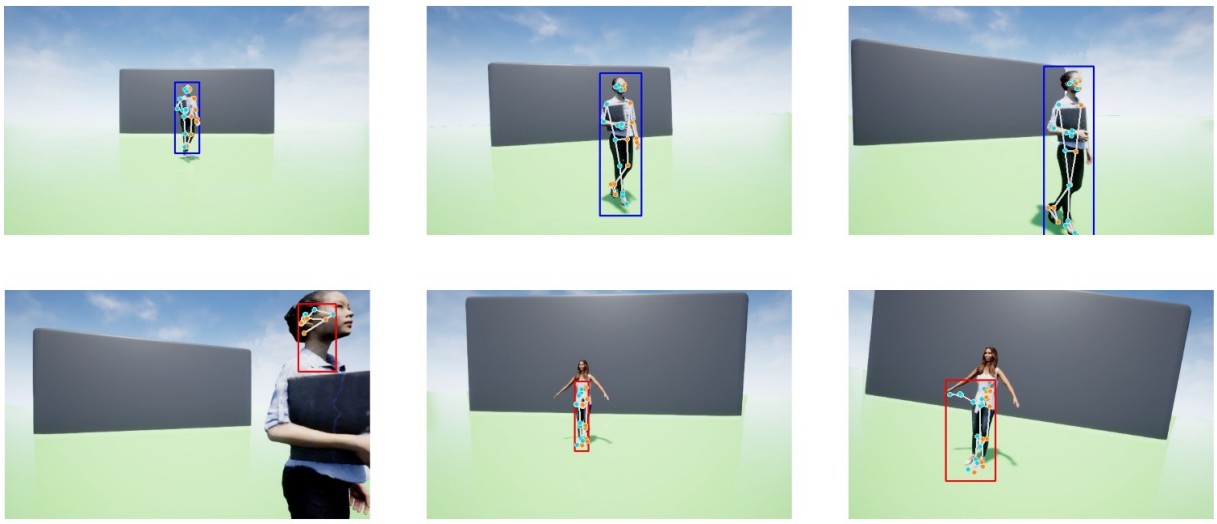


Figure 3.20: Valid versus invalid poses detected by the follow solution

of two independent PID controllers. The theory behind these controllers is explained in section 3.5.1.

The first of the PID controllers is in charge of controlling the yaw velocity of the vehicle to respond to horizontal movements in the x direction of the image. It takes as input the x coordinate of the centre point of the bounding box, and its target is the middle point of the screen. Therefore, this controller will output velocity commands to maintain the bounding box centred horizontally in its field of view. The second PID controller controls the forward velocity of the vehicle to respond to distance changes of the person getting closer or farther away from the drone. It takes as input the height of the calculated bounding box as a percentage of the total height of the image and works to keep it within a value that matches the desired distance to maintain between the person and the vehicle by moving forwards when the height is too low and backwards when it is too high. The exact percentage of the image height covered by an average person at a given distance depends on the camera used, so the target height of the controller needs to be set through trial and error for each separate video source. Figure 3.21 shows how these two inputs are extracted from the coordinates of the bounding box detected around the figure.

A diagram summarizing the structure of the follow module[17] can be seen in figure 3.22. 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, a new image is retrieved, pose features are extracted from it, and a bounding box is calculated. Then, and only if the vehicle is in offboard flight mode, the calculated positions will be fed into the PID controller to get a velocity output to send to the pilot. Keyboard control is also available to send manual commands to the vehicle directly.

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

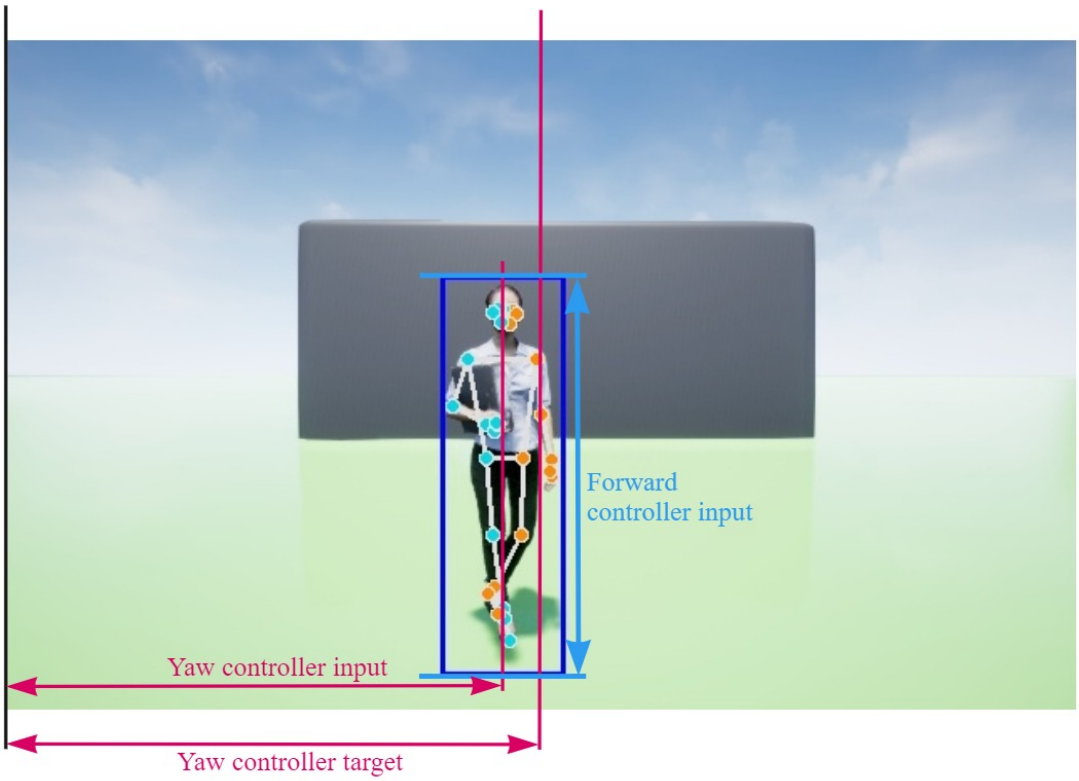


Figure 3.21: Calculation of horizontal position and height of figure from the detected bounding box.

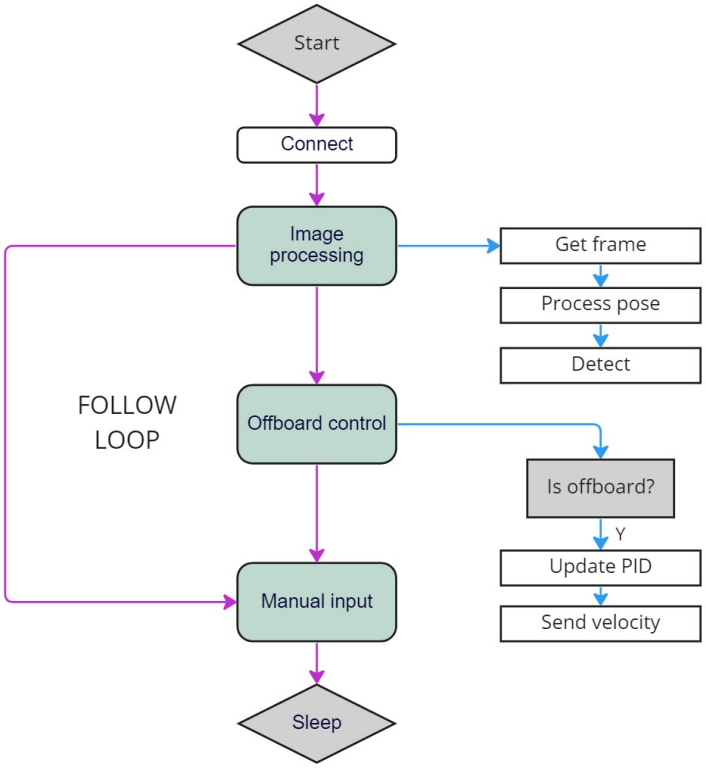


Figure 3.22: Execution flow for the running loop in the follow control solution

3.5.1 PID tools

Two additional utilities have been developed for tuning and measuring the performance of the PID controllers. These tools are used in section 4.2 to set the correct values in the controller coefficients empirically. A proportional-integral-derivative (PID) is a control loop mechanism commonly used in systems requiring continuously modulated control. It works by continuously calculating an error value from the difference between the received input on a chosen process variable, which in this case is the detected position of a person in the frame, and the desired set point for that variable, a position centred in the frame. From the error value, the output for the controller is calculated according to this equation:

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

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

The PID controllers in this project employ the freely-available `simple-pid` library for Python [33], which provides all the necessary error calculations already implemented. It is only necessary to provide the coefficients, or tunings, and the set point for the controller and then call it periodically with an updated input value to receive the output. In the Dronecontrol application, it is the `controller` module[18] the one that interacts with the `simple-pid` library to feed and receive the correct values to the PID controllers calculated from the bounding box around the detected person.

Since the most straightforward way to decide the coefficients for a controller is to test different combinations empirically, a small `tune_controller` tool has been developed to help with the process[19]. With it, it is possible to specify a range of potential coefficients and test the response of the system on the images retrieved from AirSim and a simulated flight controller (either SITL or HITL can be used). For each value to be checked, the simulated person to be followed starts in an offset position from the target centre. Then, the controller is engaged with the test values, and its detected position input and calculated velocity output a plotted in a graph for analysis. Finally, the vehicle returns to the starting position to reset the environment for the next value to be checked. The tool is run with the following command: `dronecontrol tools tune`, and can be started with the option `-yaw` or `-forward` to decide which specific controller should be tuned. The other one is deactivated during the test to visualize the effect of the examined values more easily. Each coefficient can be tested separately by providing fixed values to the other two parameters.

The second tool (`test_controller`[20]) is designed to check the controllers' performance and works slightly differently from the previous one. The underlying process is the same: engage the controller in an offset position and wait until the movement has stopped before resetting and running again. However, this tool aims to display how an already-tuned controller reacts to different position inputs for the camera. This way, the parameter coefficients remain the same for the entire test while the relative position between the vehicle and the person followed varies. During the execution, both the yaw and the forward con-

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

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

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

troller are activated simultaneously to obtain results closer to what will be expected during actual flight. However, the test tool can be run in either yaw or forward mode so that the distances to be tested between the vehicle and the person vary either left to right or forward to backward, respectively. The objective is to measure how the vehicle will react to increasingly bigger differences to the target position to ensure that the movement is stable and that safety is maintained during the entire flight. A complete execution of this tool for both modes is shown in section 4.2.3.

3.5.2 Safety mechanisms

The Dronecontrol application implements a very experimental vision-based guidance system. Therefore, to carry out flight tests with real-life conditions, it is necessary to ensure that both the software implements sufficient safety mechanisms to prevent accidents. Some of these defences employed are part of the developed code, and others are activated simply from the PX4 safety configuration.

In the first place, the Dronecontrol computer vision module prevents accepting as valid input objects that may have been detected wrongly as a person by the MediaPipe algorithm. This is determined by accepting only received feature points that have a minimum resemblance to a standing human figure, where the bounding box's height is always greater than its width and with the features associated with head, shoulder, hip, knee and ankle stacked from top to bottom in that order. If any of these checks fail, or if there is no detection output at all received from the library for the current frame, the vehicle will go into Hold flight mode, discarding queued velocity commands, if any, and making the drone hover over its current position. These checks are implemented in the `image_processing` module[21] for the follow solution.

The second safety mechanism relates to the PID controllers and the pilot module and works by limiting the maximum possible velocity of the vehicle at all points during the execution. This is done both inside the Dronecontrol application by setting up output limits on the PID controllers that prevent the guidance system from sending abnormal velocity commands to the flight controller, and from the PX4 autopilot itself as a fallback in case of issues with the custom MAVLink integration or the companion computer as a whole, by setting the `MPC_XY_VEL_ALL` parameter in the autopilot configuration which limits the overall horizontal velocity of the vehicle.

Other safety configuration options included as part of the PX4 autopilot detect undesired behaviour in the flight conditions and trigger a flight mode change to either Hold (hover) or Return (fly back to takeoff position and land), as documented in *Safety Configuration* (*Failsafes*) | *PX4 User Guide* [34]. Some of the detected failure conditions are:

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

1. Lost connection to companion computer while in Offboard mode.
2. RC transmitter link lost while in manual modes, which can be extended to trigger in Offboard mode.
3. Lost GPS position, when the quality of the PX4 position estimate falls below acceptable levels.
4. Low battery during flight.
5. Vehicle unexpectedly flips.

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

Chapter 4

Experiments and validation

This chapter explains the process of validating each individual piece necessary for the correct operation of the complete system, from testing the first simulation steps to carrying out thorough flight tests on the final guidance solution. Figure 4.1 shows the order in which each step is followed during the validation process.

In the first section, the application is tested in a purely simulated environment, first individually for each part and finally integrating the flight mechanics simulation with providing images to the computer vision detection software and with sending control commands to the flight controller simulation. In the second section, the follow detection and guidance solution is tested thoroughly to guarantee that only safe velocity commands are outputted from the PID controllers for any image input that could be received and that their response is according to the desired movement of the vehicle. In the third section, once the software is guaranteed to operate inside the required safety parameters, the simulation is shifted to the dedicated hardware that will be used for actual flight: the dedicated autopilot board and the companion computer that will be onboard the vehicle; to ensure that all the connections are working as expected and the devices offer the necessary performance. In the fourth and final section, several flight tests are executed with increasing complexity, from the basic controlling of the vehicle with an RC controller to fully autonomous flight with target following.

4.1 PX4 SITL simulation and validation

Section 3.1 describes the software-in-the-loop simulation mode developed by PX4. Using this mode makes it possible to test and validate the correct operation of each part in the program's architecture. To start, it is necessary to validate that it is possible to use the connection between the simulated flight controller and the Dronecontrol program to send

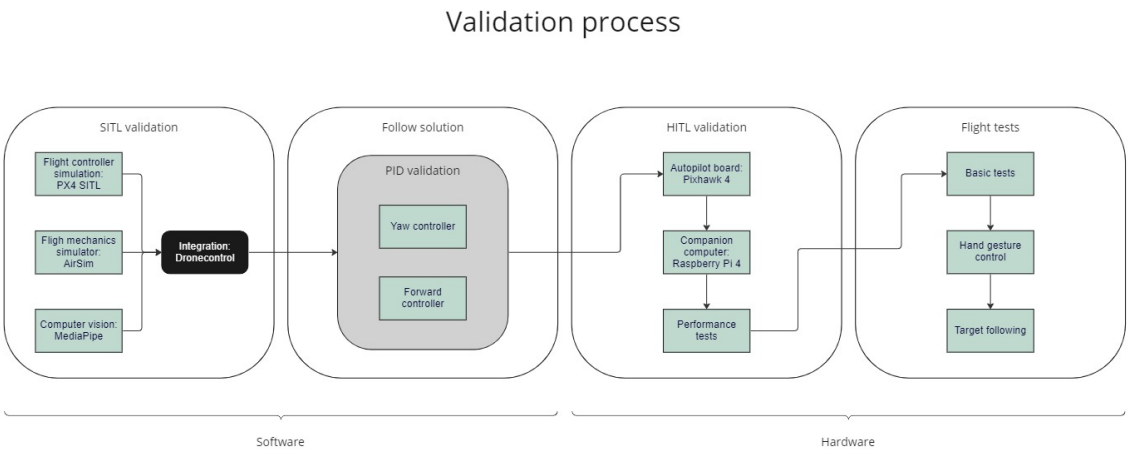


Figure 4.1: Outline for the validation process

commands to the simulated vehicle, as well as to capture images from a connected camera and use them to test the detection algorithm. The full list of validation steps is, therefore:

1. Verify that it is possible to start the simulated flight controller and that it connects to the 3D simulator vehicle.
2. Connect the dronecontrol program to the flight controller and send basic commands.
3. Retrieve images from a camera and run computer vision detection on them.
4. Integrate the last three steps by running the hand-gesture control solution described in section 3.4

For this purpose and to be able to run with a minimal configuration, the Gazebo simulator[1] included with the base PX4 installation will be used as the 3D environment. This simulator works inside Linux on the same computer that runs the SITL PX4. To be able to later transition into using the AirSim simulator instead, which runs in Windows, with minimal changes, for this test, PX4, Gazebo and the Dronecontrol program will run inside the Windows Subsystem for Linux. The complete installation process necessary to run these tests is explained in appendix A.1 and A.2.

Once both parts are installed, PX4 and Gazebo can be started by running the `make px4_sitl gazebo` command inside its installation folder. The result of this command can be seen in figure 4.2, where the left part shows the user interface and 3D world of the Gazebo simulator and the right side contains the PX4 console that can be used for sending

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

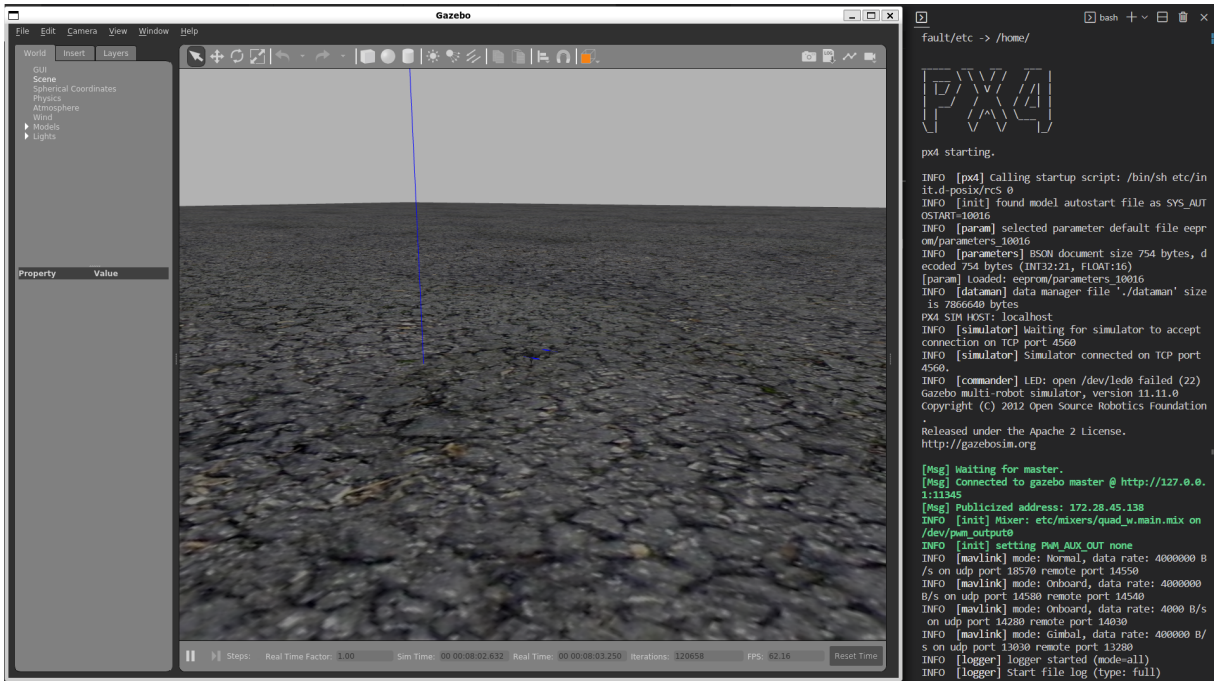


Figure 4.2: Gazebo simulator (left) and output from the PX4 terminal (right) after PX4's software-in-the-loop mode is started

commands and changing the configuration parameters of the simulated flight controller. The most straightforward command to test is takeoff, which is done by sending `commander takeoff` through the console. Figure 4.3 shows the simulator's state after the takeoff command, where the vehicle model has climbed to the default height of 2.5 meters above the ground. The command to land the vehicle again is `commander land`.

The next step is to connect the Dronecontrol application. The camera testing tool described in section 3.3.3 has been developed specifically for this test so that it is possible to establish a connection to PX4 SITL and process images without engaging any of the program's control modules. The commands are then sent to PX4 through keyboard input. For example, the key "T" will make the simulated vehicle take off. 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 joint of the hand present in the captured image and on the right side, the logged information shows when the connection is established and keyboard commands are sent to the simulator.

The entire execution of a test of the hand-gesture-based control solution is shown in this video[2] in this link and an image extracted from it can be seen in figure 4.5.

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

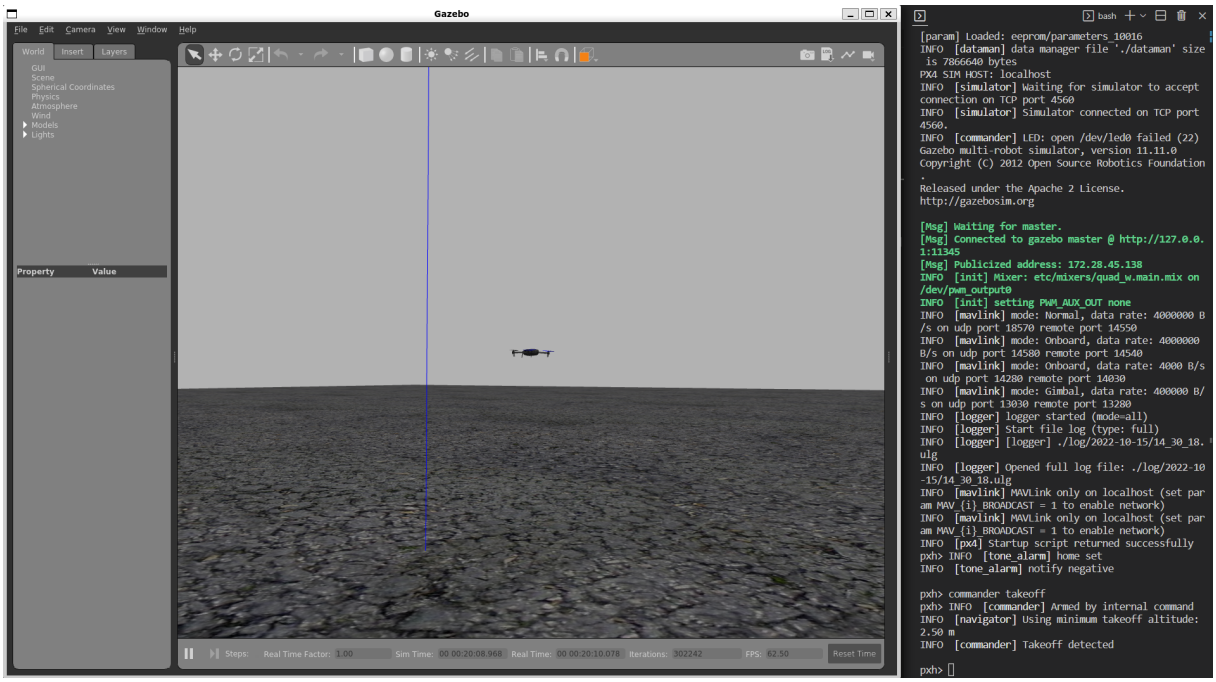


Figure 4.3: Gazebo simulator (left) and output from the PX4 terminal (right) after the takeoff command has been executed

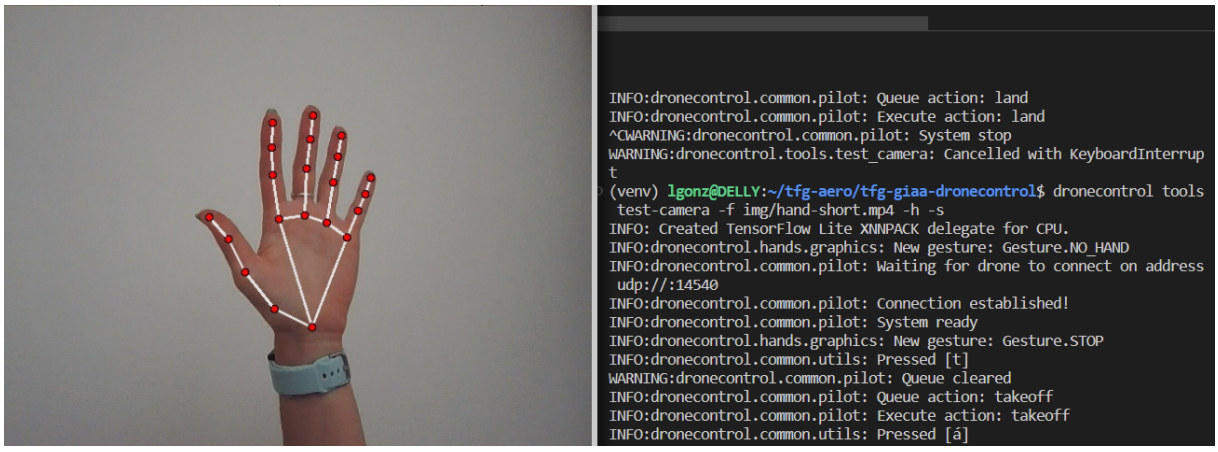


Figure 4.4: Hand detection algorithm running on images taken from the computer integrated webcam

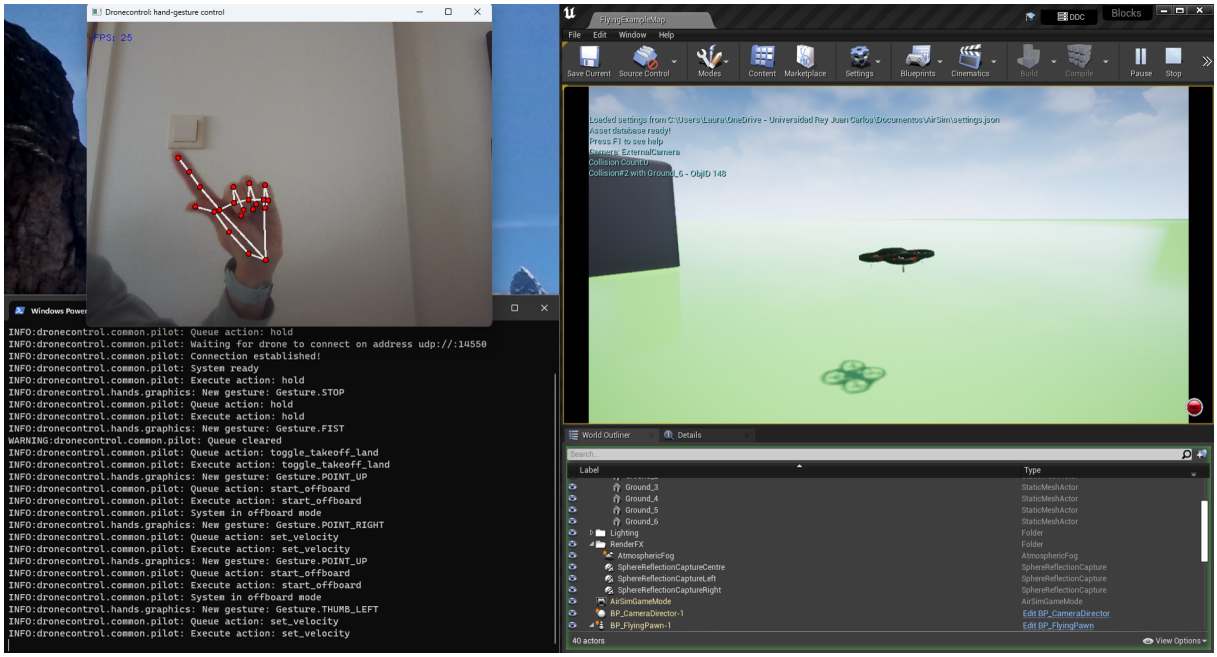


Figure 4.5: Single frame from the video showing the full execution of the hand-gesture control solution

4.1.1 PX4 SITL validation with AirSim

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

1. Verify that the AirSim simulator can start, connect to the PX4 SITL through the WSL virtual network and receive commands from the PX4 terminal.
2. Check that the dronecontrol program can connect to both AirSim through the WSL network to receive images and PX4 through the local network to send movement commands.
3. Test the pose recognition algorithms on the images obtained from the AirSim simulation.
4. Check that the follow solution can control the vehicle's velocity directly in PX4's offboard mode.

In the first place, the AirSim simulator needs to be installed in the Windows host. The complete installation process is described in appendix A.1. There are specific configuration

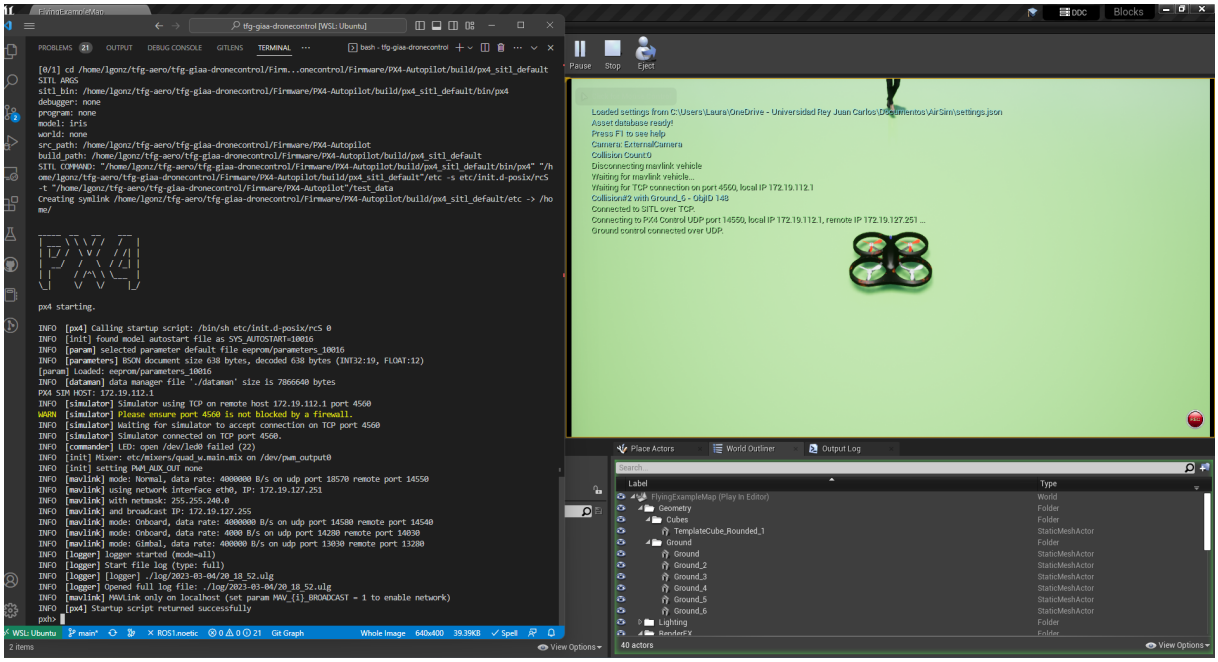


Figure 4.6: AirSim environment connected to PX4 flight stack running in SITL mode

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

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

This starts the software-in-the-loop execution, which attempts to attach to an already running simulator listening on the IP address specified and the TCP port 4560, in this case, AirSim. Therefore, every time either PX4 or AirSim stops its execution, both of them have to be restarted in the specific order of first the AirSim simulator and then the PX4 flight controller. Figure 4.6 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. To test the detection and tracking of human figures from images taken inside the simulator, one can again use the camera testing tool provided with dronecontrol. Figure 4.7 shows the output when the tool is run with a 3D model of a person situated in front of the drone in the

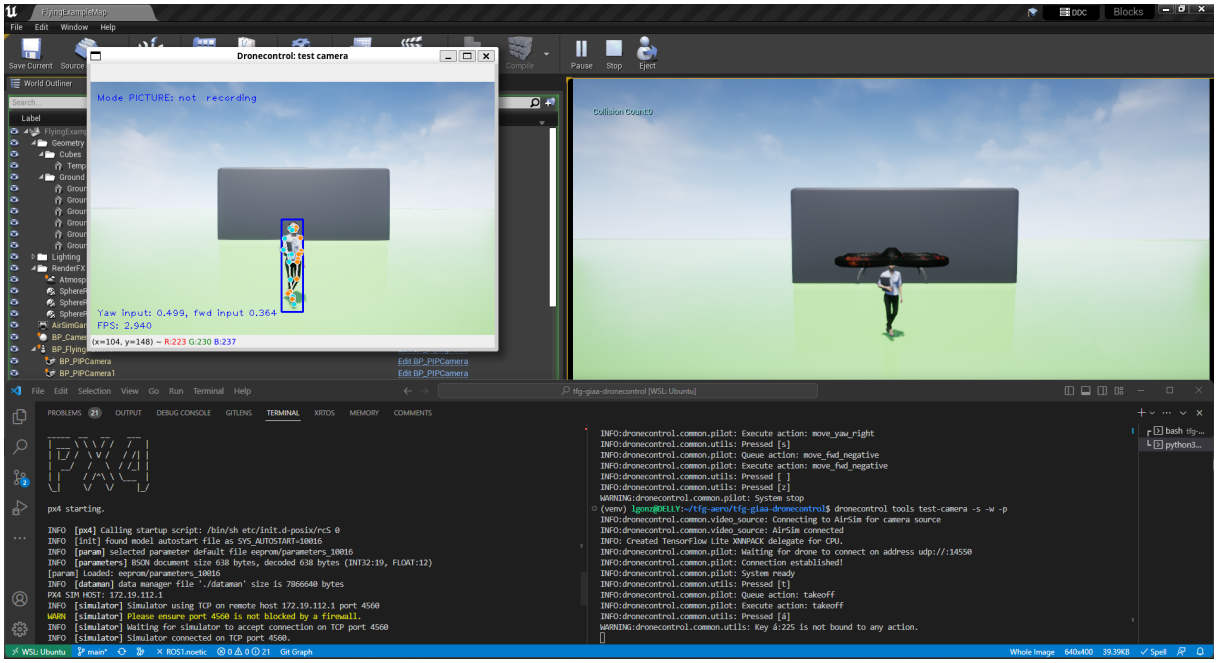


Figure 4.7: AirSim, PX4 and dronecontrol applications running side-by-side and connecting to each other

simulated world and the command:

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

In the image, the computer vision utility detects the main features of the human body and a bounding box is drawn around it. Meanwhile, the logged output from the program shows two calculated positions in the terminal: the x coordinate of the midpoint of the bounding box and the percentage of the image height covered by the height of the bounding box. These two numbers are the inputs for the PID controllers used in the follow solution as described in section 3.5 so that the output can be used to calibrate the distance from which the drone is to follow the person when that control mode is engaged.

Now that the testing environment is working as expected and before it is used to tune the PID controllers in the follow solution to the response of the vehicle's movement in the next section, it is possible to verify that the controllers are capable of reacting to changes in the position of the figure by only enabling their proportional term with an appropriately low magnitude to keep the movement slow and smooth. The drone movement when running the follow control program with values of 10 and 2 on the proportional term of the yaw and forward controllers, respectively, which can be done with the command `dronecontrol follow --sim --yaw-pid (10, 0, 0) --fwd-pid (2, 0, 0)`, is shown in this video[3] and a frame extracted from it can be seen in figure 4.8.

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

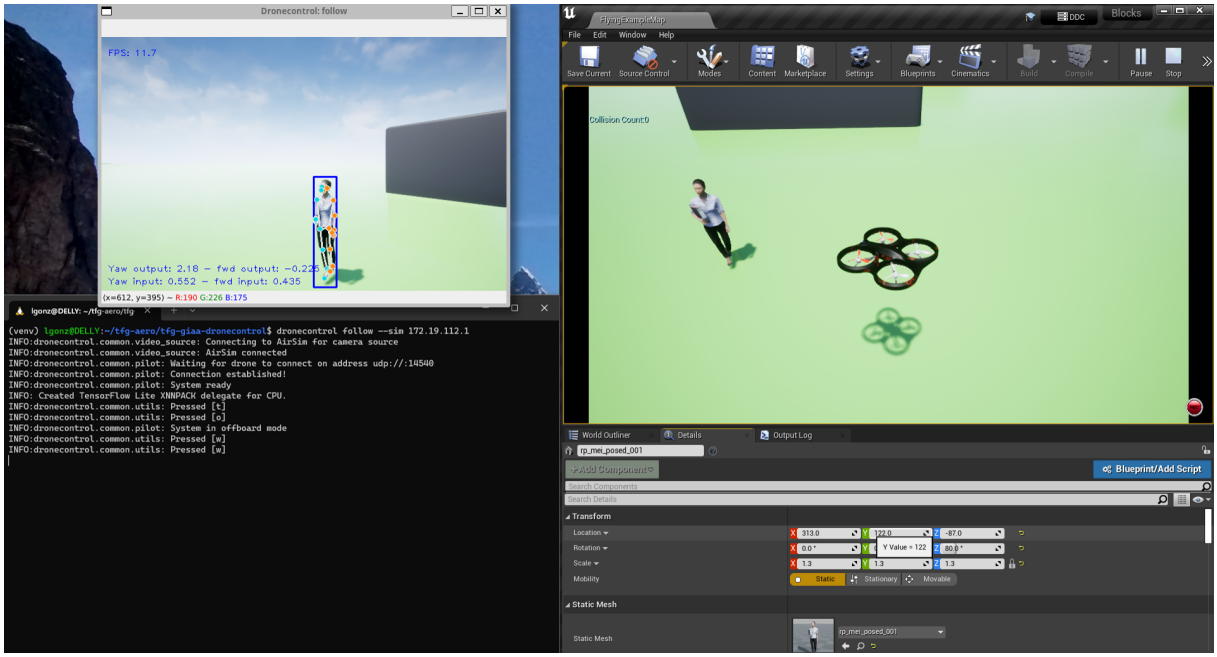


Figure 4.8: Single frame from the video showing the movement of the drone in response to changes in the position of the tracked person

Match text to video

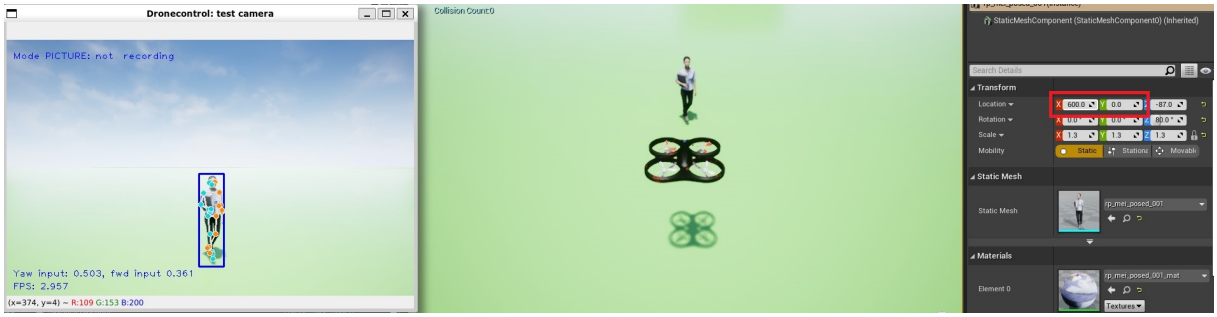


Figure 4.9: Reference position for the yaw and forward PID controllers. From left to right, the panels show the dronecontrol application window, the AirSim simulator world view and the world location of the human model in the simulator. The distance between the vehicle and the person is 600 units in the x direction and 0 units in the y direction.

4.2 PID controller validation

As mentioned before, the velocity controller that is the heart of the person following mechanism is based on two PID controllers. In order to get velocity outputs for the PID controllers that produce a stable movement of the vehicle, it is necessary to tune the parameters of the controllers until the appropriate combination is found. In this section, the value for the coefficients used for the project will be found empirically with the help of the tuning tool described in section 3.5.1 and developed for this purpose. Its performance will be validated with the controller testing tool, both running in the simulated environment with the flight stack in SITL mode so that it is possible to take continuous measures of the response of the PID controllers to step movements of a simulated person present in the 3D world. In the first place, each controller will be tuned independently of the other by allowing the vehicle only one direction of movement at a time. Afterwards, both controllers are engaged simultaneously to measure the joint response to a range of different inputs.

The controllers are calibrated for a reference position of x=0, y=0 for the vehicle and x=600, y=0 for the person in world coordinates of the simulated environment. At these positions, processed images taken from the simulated camera detect the person centred in the field of view and with a height of 36% of the image height. This means that the controllers running in the simulator will have as their target set points 0.5 for the yaw controller and 0.36 for the forward controller. So for any changes in the position of the simulated person, the controllers will send velocity commands to the autopilot to achieve the same relative position between the vehicle and the person as in the reference shown in figure 4.9.

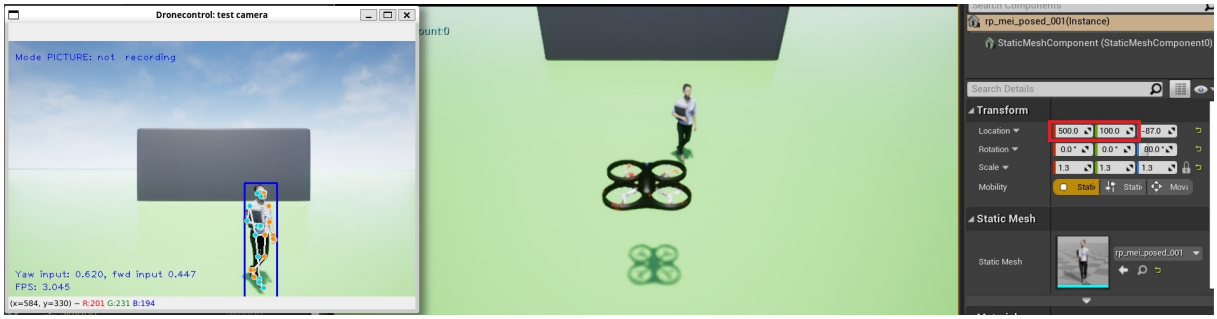


Figure 4.10: Starting position of the simulator for tuning the yaw controller. The human model is situated 500 units forward and 100 units to the right of the vehicle model.

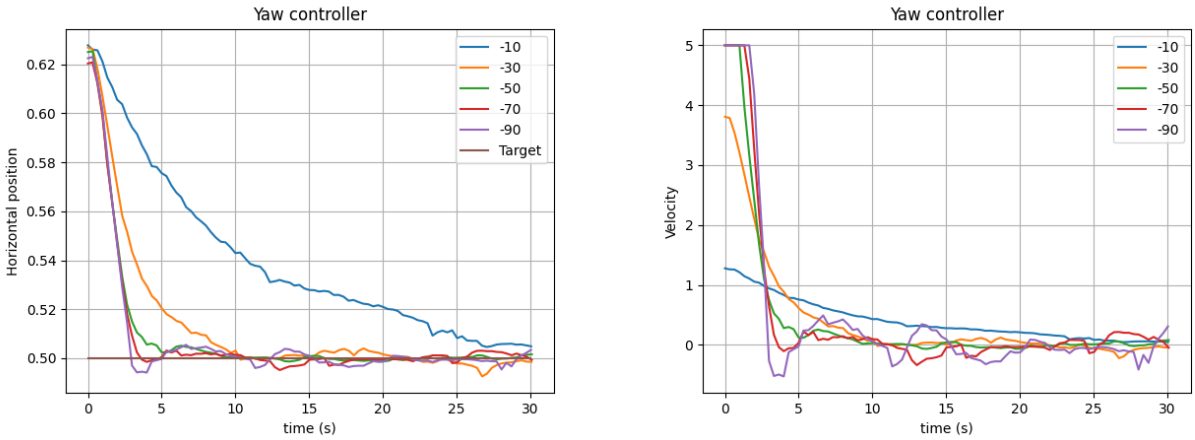


Figure 4.11: Variation of (a) input position and (b) output velocity for different values of K_P and $K_I = 0$, $K_D = 0$ while the yaw controller is engaged.

4.2.1 Yaw controller

To find the correct coefficients for the controller that governs the yaw velocity of the vehicle, the target person is set in a position slightly to the side on its field of view so that when the controller is engaged, it outputs a rotation of the vehicle to that side. Since the forward controller will not be engaged during the test, the target person can be situated closer to the camera to make it easier for the pose detection algorithm to output correct landmarks. Figure 4.10 shows the starting position of the simulated environment before each run, where the 3D model has been situated at $(500, 100)$, that is, 100 units to the right of the reference position. On the left-most panel of figure 4.10, the dronecontrol application shows that the input to the yaw controller is 0.62 at this position. The controller must then output a positive yaw velocity to centre the person in its field of view. This offset position to the right produces an error that is less than zero (set point minus input, $\epsilon = 0.5 - 0.62$). However, it requires a positive velocity to counteract and induce a yaw velocity to the right, so the coefficients for the yaw controller will need to be negative so that an increased horizontal position results in a positive yaw velocity to decrease it, as indicated by equation 3.1.

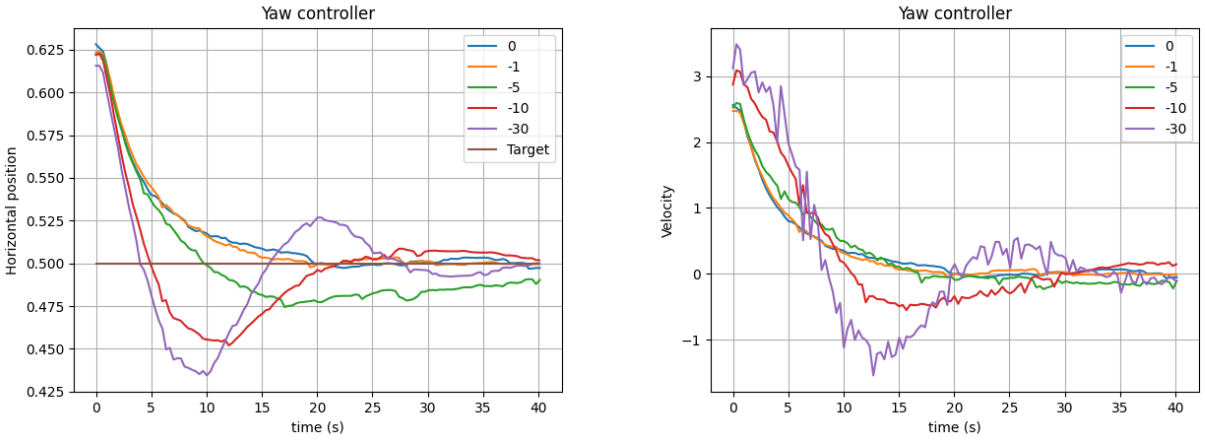


Figure 4.12: Variation of (a) input position and (b) output velocity for different values of K_I and $K_P = -20$, $K_D = 0$ while the yaw controller is engaged.

To tune the controller to its correct coefficients, the first step will be to test different values of K_P while maintaining K_I and K_D at zero. Figure 4.11 shows the output of the tuning program for the five values of K_P tested, from $K_P = -10$ to $K_P = -90$ in increments of 20. The left graph represents the variation of the horizontal position detected by the camera for the first 30 seconds after activating the controller. The right graph shows the yaw velocity that the controller outputs to the pilot module to reach the target. For low values of K_P, the controller makes the vehicle move slowly towards its target so that it takes a long time to reach the midpoint position. On the other hand, for high values of K_P, the controller tries to reach the target too fast, so when it gets close to it, it starts to oscillate around the target. The right side graph also shows well how the output velocity in the yaw controller is limited to 5 degrees per second, so even for very high values of K_P, the vehicle will not rotate faster than that. From the graphs, the best of the values tested is $K_P = 50$, where the distance to the target point decreases rapidly (left graph), but the velocity does not start to increase and decrease widely around 0 (right graph).

The second step is to find the correct value for K_I. To do that, several values of K_I will be tested for a low K_P of -20 and $K_D = 0$ so that the effect of the integral part is easier to appreciate. Figure 4.12 shows the evolution of the input and output at the controller for a sample time of 40 seconds for each of the values tested. For a low $K_I = -1$, the progress toward the target is stable and slightly faster than without any contribution of the integral part. However, when the K_I increases in magnitude, it creates initial oscillations around the target position that fade out as time progresses. For a very large K_I from around -10, the vehicle's velocity becomes locally unstable with many slight variations in its oscillations.

A similar effect can be observed to a lower extent in figure 4.13, where the measurements have been taken for $K_P = -50$ and $K_D = 0$. In this graph, $K_I = -1$ makes the controller reach the target position some 3 seconds faster and with similar oscillations in its velocity than with the proportional part exclusively ($K_P = -50$, $K_I = 0$, $K_D = 0$).

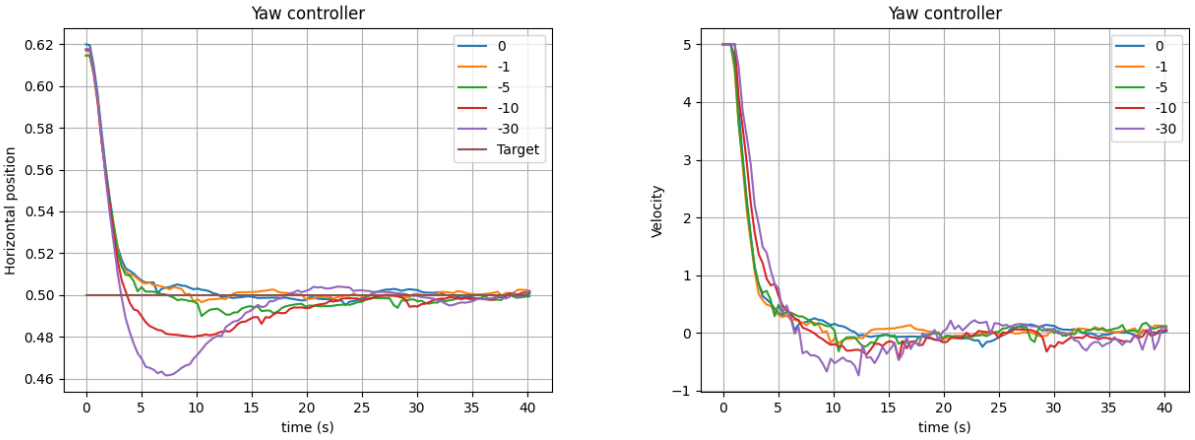


Figure 4.13: Variation of (a) input position and (b) output velocity for different values of K_I and $K_P = -50$, $K_D = 0$ while the yaw controller is engaged.

In the last step, the tuning process will deal with the derivative part of the controller. In this case, several values of K_D have been tested against the chosen $K_P = -50$, both without any integral part and with $K_I = -1$. This will allow seeing the effect that the derivative part has in the controller, as well as validating if the integral part chosen in the last step can work together with the derivative part to make the controller react better to a changed input. Figures 4.14 and 4.15 show the evolution of the position detected by the computer vision system and the velocity the controller outputs for a sample time of 40 seconds for $K_I = 0$ and $K_I = -1$, respectively, with $K_P = -50$. For all the tested values, the iteration with $K_I = -1$ on 4.15 shows a better convergence towards the target positions than their counterpart values of K_D for $K_I = 0$, which indicates that the integral part has been chosen correctly. Furthermore, adding any amount to the derivative part does not produce any visible benefit in the step response of the controller and the curve that first stabilises on the target position continues to be the one for $K_D = 0$, while the velocity graph remains approximately the same between $K_D = 0$ and $K_D = -2$ The final values for the coefficients for the yaw controller will then be $K_P = -50$, $K_I = -1$, and $K_D = 0$, so the controller will, in truth, only be a PI controller and not a complete PID controller.

A recording of the whole tuning process for the yaw controller can be seen in this video[4].

4.2.2 Forward controller

A similar process to the one used for the yaw controller must be followed for the forward controller. The starting position for the tuning process is, in this case, with the figure closer to the vehicle than the reference position and centred in its field of view. Figure 4.16 shows this starting setup with the figure situated at the (450,0) position in the simulated world.

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

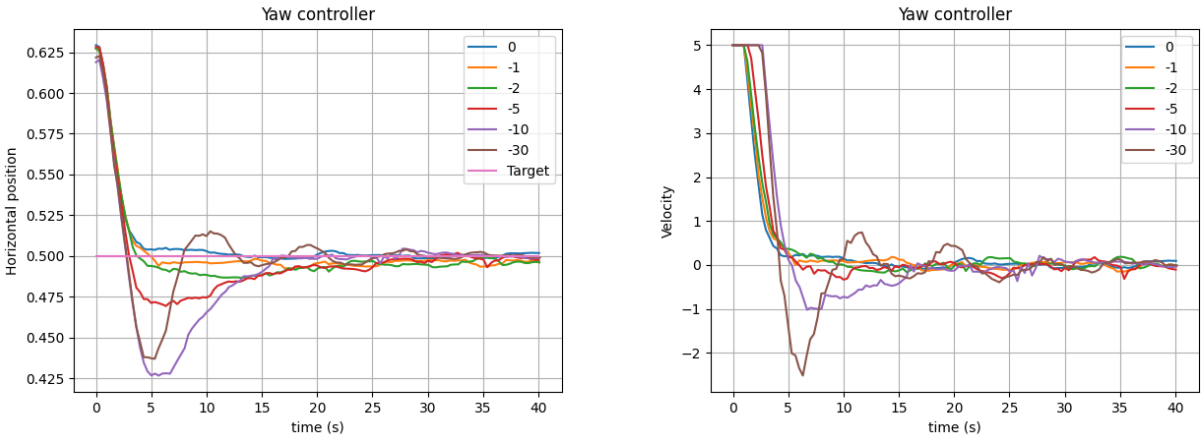


Figure 4.14: Variation of (a) input position and (b) output velocity for different values of K_D and $K_P = -50$, $K_I = 0$ while the yaw controller is engaged.

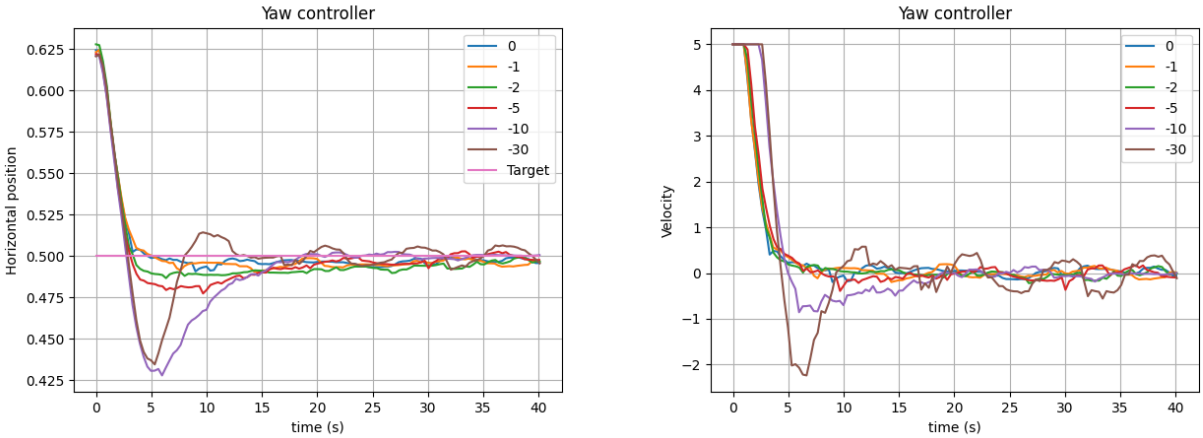


Figure 4.15: Variation of (a) input position and (b) output velocity for different values of K_D and $K_P = -50$, $K_I = -1$ while the yaw controller is engaged.

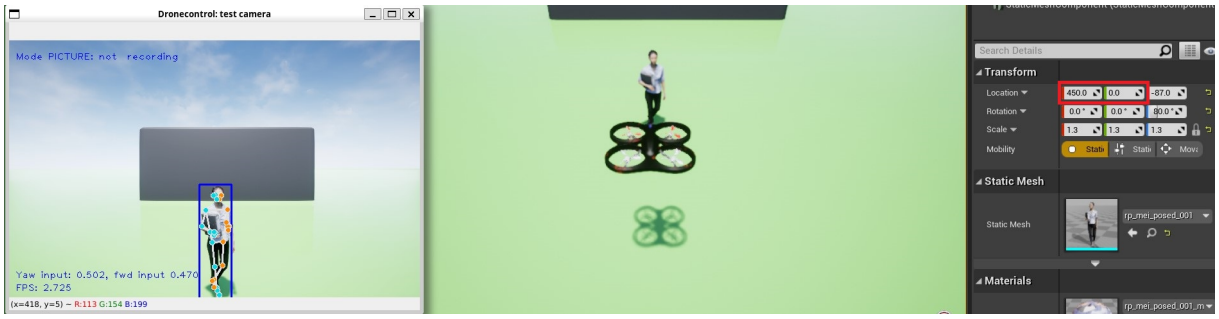


Figure 4.16: Starting position of the simulator for tuning the forward controller. The human model is situated 450 units forward and centred from the vehicle position.

In this position, the input to the forward controller is 0.47, that is, the bounding box around the detected figure takes up 47% of the height of the camera field of view. The response from the controller will therefore need to be a negative forward velocity that brings the vehicle away from the target person to reduce the perceived figure height. Since a negative output velocity reduces the input at the entrance of the controller in a directly proportional manner, the coefficients for this PID controller will need to be positive, contrary to what happened on the yaw controller where the feedback loop was inversely proportional (a positive velocity decreased the input to the yaw controller).

In general, the forward velocity will always need to be smaller than the yaw velocity since it induces a pitch angle in the vehicle that tilts the camera up and down, which can destabilise the camera and cause a loss of sight of the followed figure. To achieve that, the coefficients for the forward controller will be reduced by one order of magnitude. First, values from $K_P = 1$ to $K_P = 9$ in increments of 2 have been tested for $K_I = 0$ and $K_D = 0$ for a sample time of 30 seconds. The curves described by the controller for these coefficients are shown in figure 4.17. Compared with the trajectory described by the yaw controller, the forward controller is generally more unstable since fast forward and backwards movements affect the pose detection algorithm. This is particularly visible at the start of each test as slightly different heights are detected in the image from the camera even though the vehicle is in the same position with respect to the human figure. It also creates an effect, especially for the bigger K_P values tested, where, as the vehicle begins its movement back to its target position, the pose detection mechanism gets a slightly different perspective on the followed person, which increases the detected height slightly so that small spikes of detected differences show in the height graphs even though the velocity graph shows that the vehicle's direction of movement remains the same. This effect is reduced by keeping the output forward velocity small in the controller. In the right graph of figure 4.17, it is also visible for high values of K_P how the output velocity increases enough that the maximum velocity limit is reached on the forward controller and the output is capped to 0.4 $^m/_s$. For a value up to $K_P = 3$, the trajectory described descends rapidly without ending in significant oscillations around the target height, so it is a fair value to keep for the final controller.

The respective tests for K_I and K_D in the forward controller are shown in figures 4.18

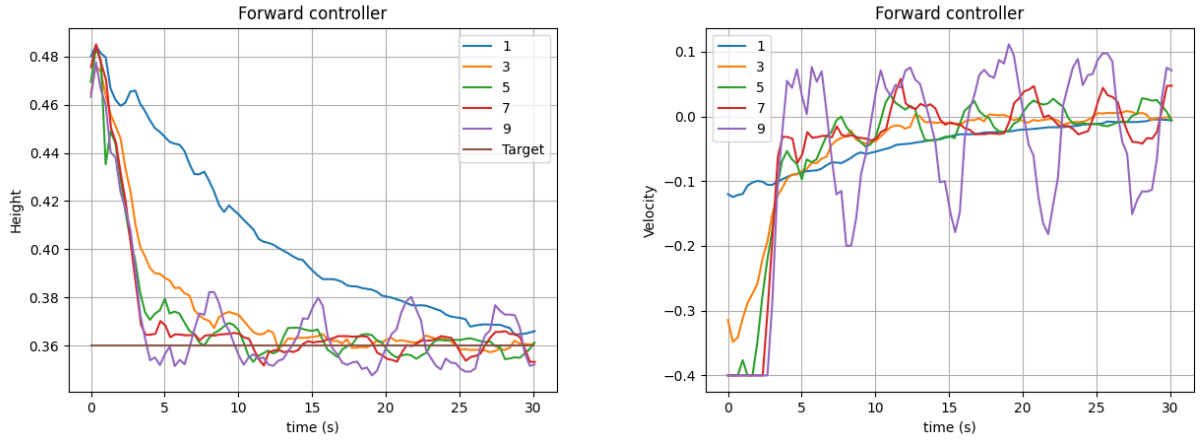


Figure 4.17: Variation of (a) input height and (b) output velocity for different values of K_P and $K_I = 0$, $K_D = 0$ while the forward controller is engaged.

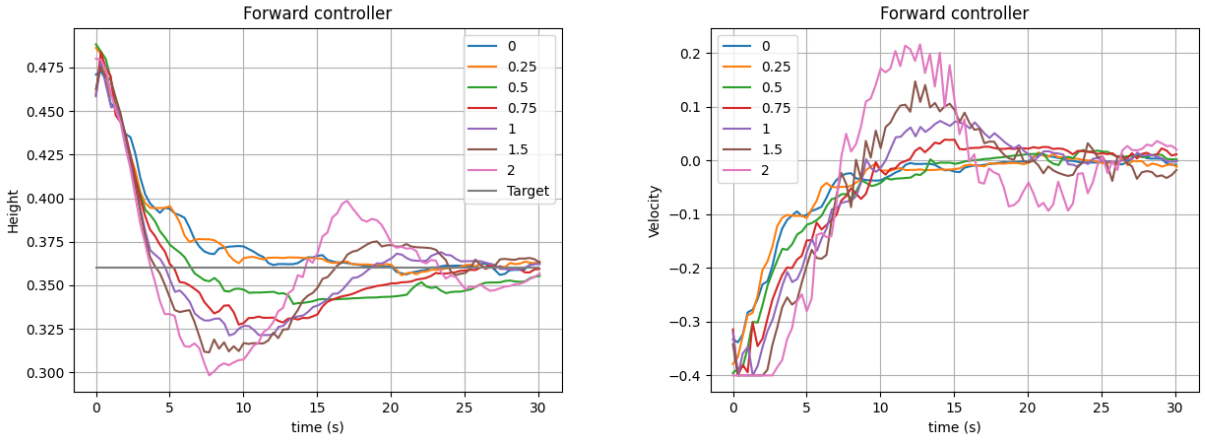


Figure 4.18: Variation of (a) input height and (b) output velocity for different values of K_I and $K_P = 7$, $K_D = 0$ while the forward controller is engaged.

and 4.19. Increasing the coefficient for the integral part causes the controller to overshoot the target initially but then stabilise before starting to approximate to the target position. Since, for this application, overshooting is not desirable, as it can cause safety issues, the chosen value of K_I will be 0. For the derivative part, every value of K_D tested increases the system's oscillations, so it will also be left out of the forward controller. The final values for the coefficients for the forward controller will then be $K_P = 3$, $K_I = 0$, and $K_D = 0$, which makes it only a proportional controller.

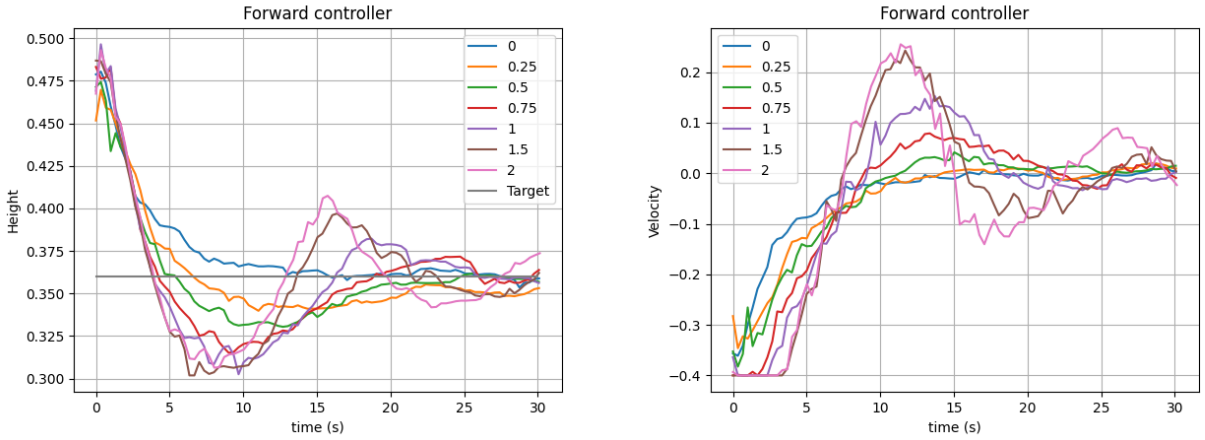


Figure 4.19: Variation of (a) input height and (b) output velocity for different values of K_D and $K_P = 7$, $K_I = 0.5$ while the forward controller is engaged.

4.2.3 PID tuning validation

As a final validation for the tuning obtained previously, the `test-controller` tool described in section 3.5.1 will be used to check the step response of the controllers to different starting distances for the same coefficients. Additionally, for this test, both controllers will be engaged simultaneously to verify that they work well together. Firstly, the positions will vary on the y-axis, that is, the figure will move from left to right in the field of view of the vehicle. The values for the y coordinate to be tested will be between -150 and 150 units in increments of 50. The x coordinate of the figure in the simulated world will remain at $x = 500$.

Plots of the changes in normalised horizontal distance and normalised figure height detected by the person recognition algorithm during the time the vehicle takes to reach the target distance from the human figure for each of the tested positions can be seen in figure 4.20. It is considered that the vehicle has reached its target when the error is less than 2%, and the output speed at the controller is less than 10% of the maximum value. Furthermore, the whole testing process followed with the developed `test-controller` tool can be seen in this video[5].

Since each starting position differs in distance and orientation to the target, both controllers must be engaged to reach the centre. The yaw controller then introduces a negative yaw velocity when the figure is on the left half of the camera's field of view and a positive yaw velocity when the figure is on the right half to reach the target horizontal distance of 0.5 (figure centred in the image received from the camera); Moreover, the forward controller outputs a negative forward velocity to reduce the detected height of the figure from around 0.44 to the target 0.36 since the figure is 100 units closer to the vehicle than the reference

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

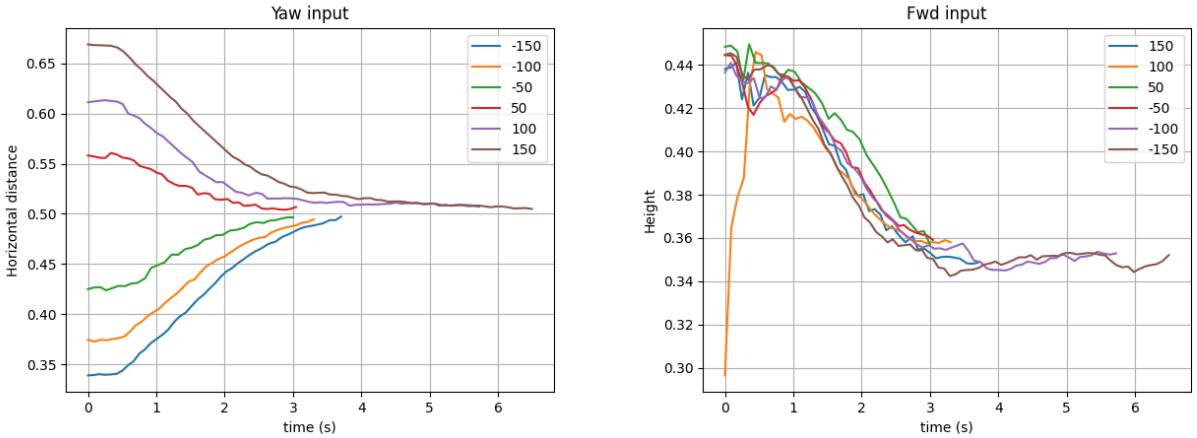


Figure 4.20: Changes over time in detected horizontal position and height as input for the controllers with different starting positions in the y-axis

position $x = 600$.

Figure 4.20a shows that the most time is spent on starting the movement towards the target and, after that, there is not much difference between the time it takes to reach the -50 position and reaching the -150 position, with the former taking 3 seconds and the later taking around 3.6 seconds. In figure 4.20b, all of the trajectories have a very similar graph since the starting distance to the target is practically the same, where the controller makes the vehicle pull backwards so the figure stays far enough, making the detected height decrease. For the $y = 100$ starting position, the initial detected height is very small in the beginning until it reaches the starting point of the other tests. This occurred because the detection algorithm took longer than usual to identify the person in this particular test. However, after less than half a second, the detection was stabilised without affecting much the time it took to reach the target or its final position, showing that the controllers can recover from errors in the detection mechanism without affecting the vehicle's movement.

Additionally, the `test-controller` tool can be run with varying positions in the x-axis, which means that the tests are run with the human figure at different distances closer and further away from the vehicle while maintaining it in the centre of its field of view (y position will remain 0 for the whole process). Therefore, the yaw controller will not need to be considered for this test. The changes over time in the input into the forward controller between each starting position and the target position are shown in figure 4.21. The figure reflects a significant difference in how the controller reacts to positions closer than the target distance or further from it. When the person is very close to the vehicle, there are major differences between detected heights, so the vehicle moves fast away from its position. However, when the person is further away from the vehicle than the target, large differences in distance are associated with minor differences in detected height, which causes the controller to determine a smaller velocity increase than for the same distance difference if the person was closer to the vehicle. Therefore it will take a longer amount of

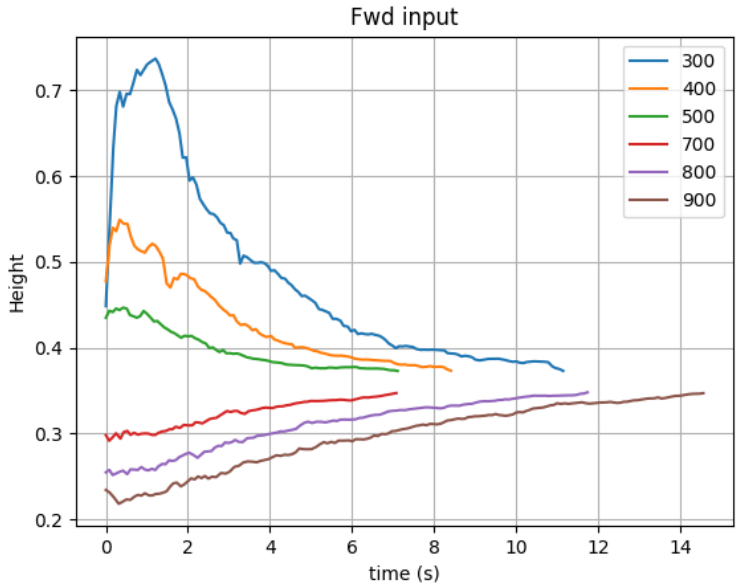


Figure 4.21: Changes over time in detected height as input for the forward controller with different starting positions in the x-axis

time for the vehicle to reach the target. It is worth noting that even when the person is so close to the vehicle that part of it falls outside of its field of view ($x = 300$ in the figure), the pose detection works well enough to interpret where the person should continue outside of the image so that it is still possible for the controller to decide the correct direction of movement.

4.3 PX4 HITL simulation and validation

The aim of this section is to validate the transition from using a simulated version of the flight stack running on Linux (PX4's software-in-the-loop simulation) to using a physical Pixhawk board with simulated input and output to test the flight controller interaction with the developed program. To do so, the aim is to be able to run the follow solution to send control commands through the Pixhawk board and observe the movement of the vehicle in the AirSim simulator in the same manner as in the previous section.

QGroundControl contains a specific quadcopter HITL airframe configuration that sets up the board with all the required parameters to activate this new hardware-in-the-loop mode. It automatically detects the Pixhawk 4 board when connected to the computer through its Micro-USB port. It is also required to make changes to the AirSim configuration for the simulator to work with HITL mode, namely, activating the option to accept connections through serial. To test the complete system configuration for HITL described in section 3.1 and outlined in figure 3.5, the Pixhawk board needs an additional channel of communication to the computer dedicated to the Mavlink exchange with the dronecontrol application. The board will therefore have both a direct cable connection to the computer and a telemetry radio on its `TELEM1` port with a wireless link to its counterparty radio connected to the computer. Since the AirSim simulator requires a higher update rate than the dronecontrol application, it will employ the faster, cabled link. The simulator can automatically connect to the board when it is started once the `UseSerial` option in the AirSim settings is set to true. The Dronecontrol program will connect through the telemetry radio by specifying the serial port and corresponding baudrate when launching the application. Since the flight stack is now running on a physical controller, it is possible to add an RC antenna to the `PPM RC` port of the board to be able to fly the vehicle with a remote control unit after it has been bound to the receiver[6]. By configuring one of the switches in the RC unit to change to PX4's offboard flight mode, additional checks to the safety measures of interrupting autonomous flight on flight mode changes or loss of signal from the RC controller can be carried out. Figure 4.22 shows all the connections mentioned.

After all the necessary connections are set up, the program can be started with the following command: `python -m dronecontrol follow --sim --serial COM[X]:57600`, where the exact COM port will vary depending on the particular USB port to which the telemetry radio is connected.

4.3.1 PX4 HITL validation with Raspberry Pi

The next step in the transition from a fully simulated environment to real flight is to connect the future onboard computer, the Raspberry Pi 4, to the Pixhawk flight controller and

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

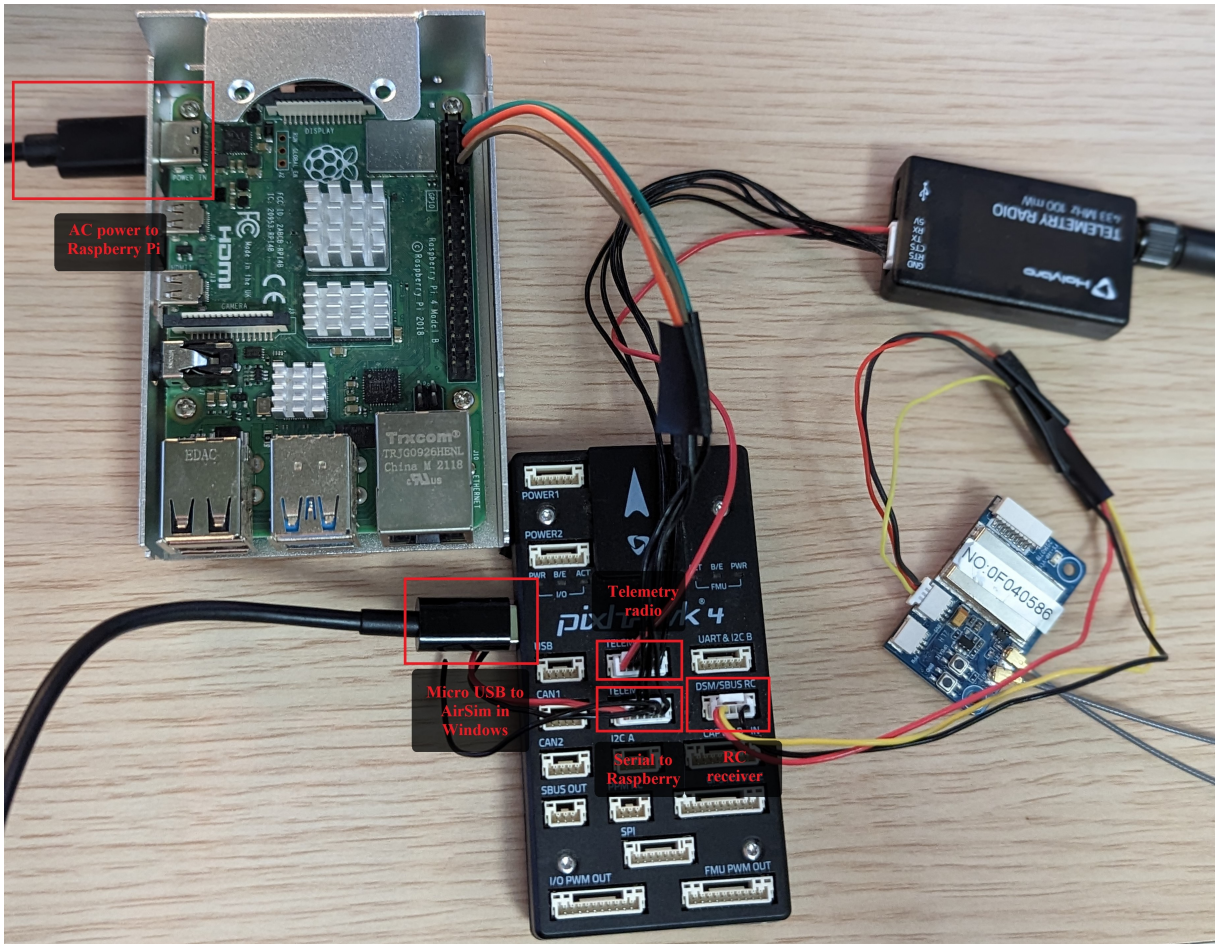


Figure 4.22: Pixhawk 4 board connected to the Raspberry Pi running the dronecontrol application and the Windows computer running the AirSim simulator, with telemetry radio for QGroundControl and RC receiver for manual control

proceed with more realistic tests on the exact hardware that will be controlling the drone. The main characteristics of the Raspberry Pi that have to be ensured to be able to progress further towards autonomous flight are:

1. Capacity to function when power is provided from a battery
2. Stability of serial connection to the Pixhawk board
3. Ability to run the dronecontrol application and all its dependencies
4. Connection to an external camera
5. Performance of the computer vision algorithms with reduced processing power

The complete installation process of all the required libraries and dependencies for the Raspberry Pi is explained in appendix A.2. The most convenient method of controlling the small computer is through a remote desktop connection, where the screen contents and mouse and keyboard input are transmitted through a local network. This way, accessing the Pi's desktop from the ground station computer is possible even during flight. One option to achieve this is XRDP[7], an open-source implementation of a Microsoft Remote Desktop Protocol server compatible with the Raspberry OS.

The first hardware connection to test is the power supply to the Raspberry. As detailed in section 3.2.3, the selected method for powering the companion computer in the onboard configuration is through a secondary battery. The connection will be made with a USB to USB-C cable from the battery to the Raspberry Pi. The intent behind testing the power supply at this point in this process is to verify that it will be suitable for its purpose before it is actually needed to power the computer onboard the vehicle. This power source must provide enough power to both maintain the processor running at an appropriate speed and provide power to the external camera in turn, which will be attached to the Pi's board and extract power from it.

A similar connector provides a channel for the Mavlink communication between the flight controller and the onboard computer. In this case, one end of the connector is attached to the `TELEM2` port in the board and the TX/RX pins are attached to the UART pins on the Raspberry's GPIO header. For this serial connection to work, both the flight controller and the companion computer need some additional configuration. The Pixhawk needs to be configured through QGroundControl to enable a secondary Mavlink channel as, by default, only the `TELEM1` port used by the telemetry radio is configured. The necessary parameters to modify and their values are collected in table 4.1. On the Raspberry side, the serial port comes configured to be used as a terminal instead of as a hardware serial port. This can be fixed through the `raspi-config` command-line utility with the following steps: Interface options -> Serial Port -> Disable login shell, enable serial port hardware.

[7] http://xrdp.org/

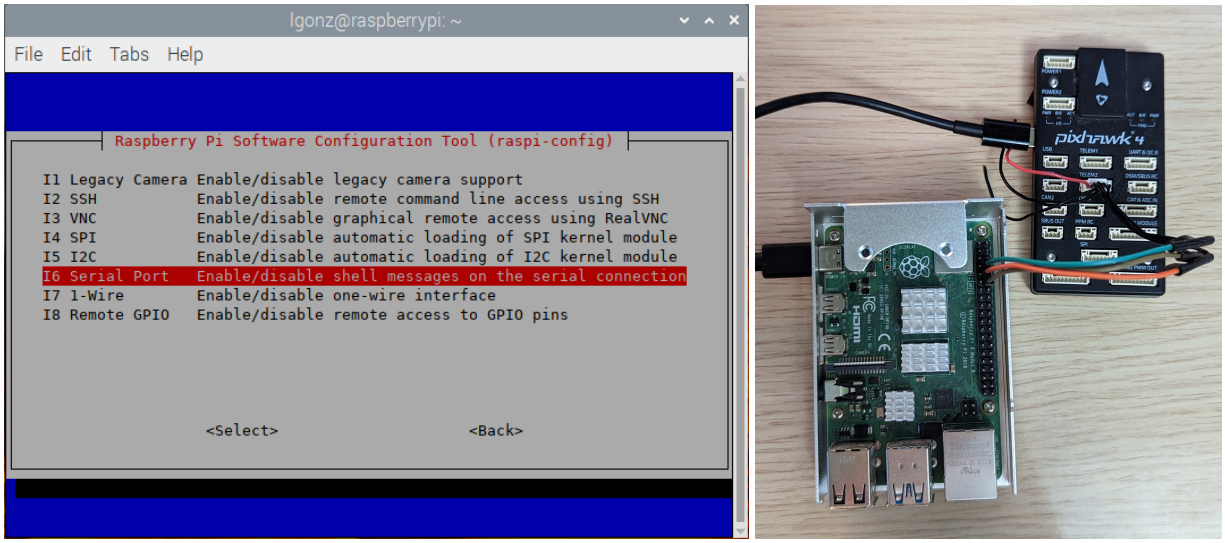


Figure 4.23: a) Picture of Raspberry's `raspi-config` and b) close-up of Pixhawk to Pi cable connection

After that, the `/dev/serial0` address can be used to communicate to the device attached to the UART pins at the baudrate configured through QGroundControl. The `raspi-config` tool and the connector used between the flight controller and the companion computer are shown in figure 4.23.

Parameter name	Value
MAV_1_CONFIG	TELEM2
SER_TEL2_BAUD	921600

Table 4.1: PX4 parameters that need to be configured to enable Mavlink communication through the secondary telemetry port

The test camera utility will be used to validate this configuration. At this point, the only physical connection to the ground station running AirSim or QGroundControl is through the development-only micro-USB port in the Pixhawk. The results from running:

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

can be seen on figure 4.24. On the right side, the remote connection to the Raspberry's desktop shows the output of the dronecontrol program running the pose detection algorithm on the images received from the simulator. On the left side, the AirSim simulator shows the vehicle's movements as it reacts to the input from the flight controller and the companion computer.

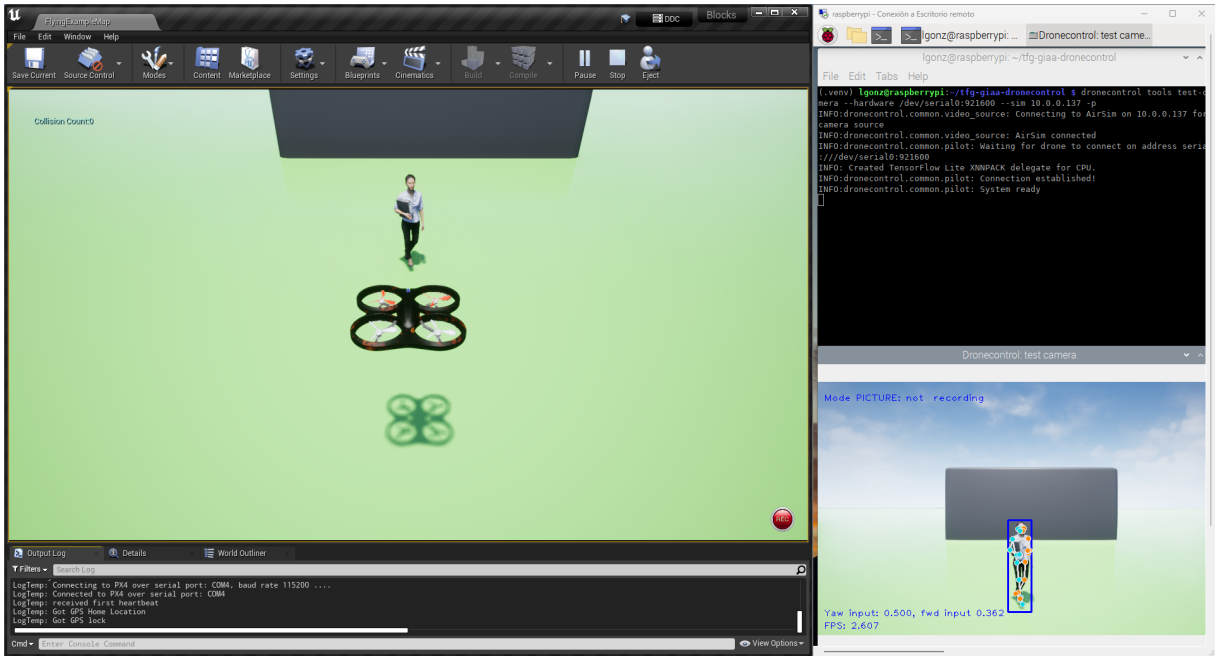


Figure 4.24: Left: AirSim simulator on Windows host, right: RPi desktop with Dronecontrol application and pose output

4.3.2 Performance analysis

The main question left to answer before the vehicle can take to the air with this hardware and software is whether the less powerful processor in the Raspberry Pi 4b, a quad-core ARM Cortex-A72 64-bit SoC running at 1.5GHz, can handle the detection and tracking algorithms with enough performance to get similar results to those obtained with simulated hardware and achieve good reactions to real-time movement. To do that, the average time that the program spends on each task in the running loop can be calculated and analyzed for different scenarios. Then it will be possible to estimate the maximum speed at which the person being followed by the algorithm can move.

From the follow loop presented in section 3.5, it is possible to divide the processing cost into several areas that can be measured independently: image processing, offboard control, manual input, and released thread (sleep).

Figure 4.25 shows the time used for each task on an average run of the follow solution with simulated hardware (PX4 running in SITL mode with AirSim). The time measurements have been taken by calculating the time difference between each statement's start and end and averaging across every iteration of the main loop. The main cost in time of each execution is found in the image processing task, which takes around 89% of the total loop time to run. This is where the most significant differences in performance will come from between the simulated hardware and the solution running in the Pixhawk 4 + Raspberry Pi combination. This image analysis process can be further subdivided to get a finer

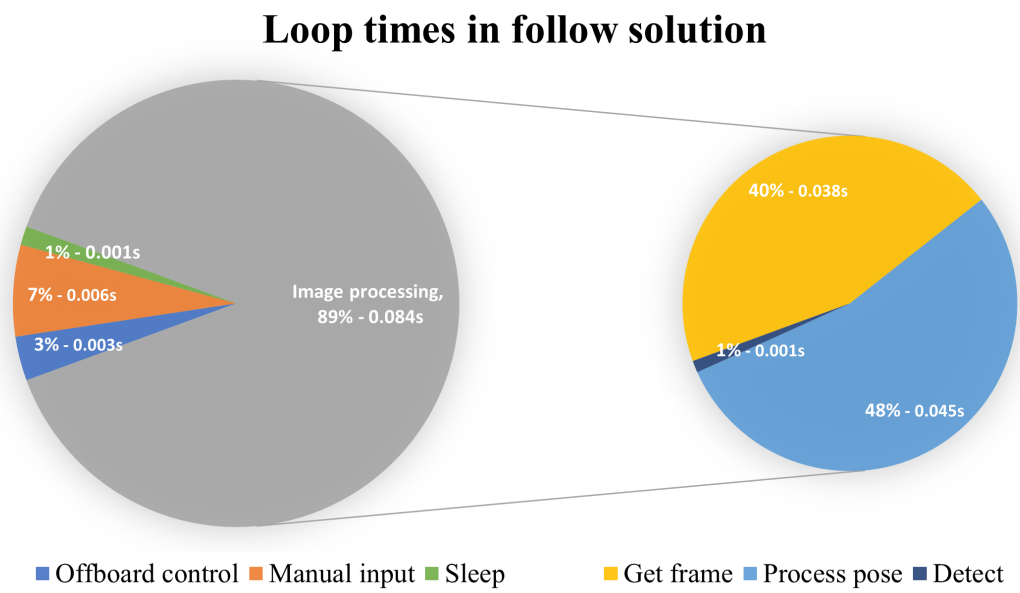


Figure 4.25: Average percentages of a loop spent by each task in the follow solution with their corresponding average absolute times in seconds.

degree of control over how much time each part takes. The three subtasks that makeup image processing are:

1. Get frame: request a new frame from the video source.
2. Process pose: send the frame to MediaPipe library for detection and tracking.
3. Detect: calculate bounding box coordinates and define whether it is a valid pose.

This further division is also shown in figure 4.25, with each of these subtasks taking 40%, 48%, and 1% of the total image processing time, respectively. The total average time for each run of the follow loop is then 0.094 seconds, which results in an average performance of 10.5 FPS (frames-per-second).

Similar measurements have been taken for hardware combinations with different degrees of simulation, running the follow solution with offboard mode enabled and connected to the AirSim simulator. These are:

1. All simulated hardware: PX4 on SITL mode + dronecontrol on standalone computer + images from AirSim simulator video source.
2. Simulated hardware with real images: PX4 on SITL mode + dronecontrol on standalone computer + images from the attached camera as video source.

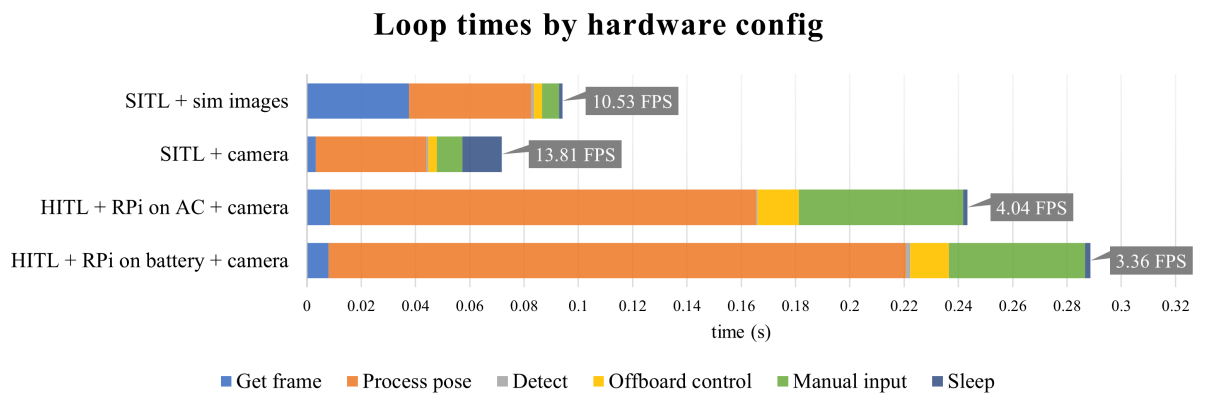


Figure 4.26: Average FPS and time spent on each task per iteration of the follow solution for the different hardware configurations.

3. Test hardware with AC power supply: PX4 on HITL mode on Pixhawk 4 + dronecontrol on Raspberry Pi + images from the attached camera as video source.

4. Test hardware with battery: PX4 on HITL mode on Pixhawk 4 + dronecontrol on Raspberry Pi powered by battery + images from the attached camera as video source.

Figure 4.26 shows the averages of the measurements taken for all the hardware combinations analyzed. In the first test, it can be observed that the retrieval of each new frame from the AirSim simulator takes a much longer time than when an external camera is used. This is because the simulation running on Unreal Engine is also limited in performance to what the computer can offer, which is not a cost that will be reflected when the simulation engine is no longer needed. In the second test, the images from the simulator are replaced with the feed from an external camera connected to the computer. This results in a much lower cost for image retrieval and the best performance of any of the tests, with an average of 14 FPS and peaks of more than 20 FPS. For the third and fourth tests, the image-processing calculations are moved to the onboard Raspberry Pi computer, which increases the time needed for pose-processing by one order of magnitude. There is also a noticeable difference in performance in the Raspberry's processor between powering the computer through the AC power supply and through an external battery, stemming from the fact that the former supplies 3A of current to the board and the latter only 2A.

With these measurements, it is possible to understand how the board will behave during actual flight, with the last test being the closest to the expected flight conditions. It should therefore be possible to sustain a performance of around 3 FPS during flight. With a time between frames of around 0.3 seconds, and since the field of view of the camera used for flight tests is about four meters wide, the person being followed should be able to move at a speed of 3-4 m/s and remain within the view of the drone.

4.4 Quadcopter flight tests

Once the performance and safety of the control algorithms have been validated in the simulated environment, it is possible to begin flight tests and take to the air with a physical drone. In this final section of the validation process, all the previous parts are put together to test how the developed software will perform in a real quadcopter during flight. To do this, first, it will be necessary to build the base vehicle from its development kit and then integrate all the additional pieces needed for this project, like the companion computer and the camera. Then, after ensuring the vehicle can fly with all the payload through remote control, both of the developed solutions will be tested. First, the hand-control solution to verify that the autopilot can receive flight commands from an offboard computer and second, the follow solution to confirm that the companion computer can function as well during flight with its dedicated power supply as it did when it was stationary.

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

1. Build the quadcopter with its basic components.
2. Add custom payload.
3. Fly with remote control and factory autopilot only, monitoring through QGroundControl.
4. Fly with custom software from offboard computer with `test-camera` tool (3.3.3).
5. Fly with `test-camera` tool from onboard computer.
6. Fly with custom hand-gesture control solution from offboard computer.
7. Fly with custom follow solution from onboard computer.

4.4.1 Build process

As mentioned before, the vehicle used in this project is the Holybro X500, a drone designed to work with PX4. The detailed instructions to build the vehicle from its Development Kit can be found in the PX4 documentation[8]. Figure 4.27 shows all the parts that make up the complete vehicle.

After all the standard parts are assembled, the custom additions can also be attached using the remaining space in the frame. The Raspberry Pi companion computer will sit

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

Figure 4.27: Development kit for the Holybro X500.

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

just behind the autopilot to counterbalance the more weighted front of the vehicle where the GPS antenna is located. This location also allows an easy connection between the autopilot and Raspberry's I/O pins with short cables so as not to clutter the frame with wires excessively.

While in flight, the Raspberry Pi will be powered through a dedicated external battery that outputs power through a 2-ampere USB port. This port will be connected with the Raspberry Pi's original power cable to its USB-C power supply socket. As detailed in 4.3.1, this current is enough to power the connected camera and run the developed software with acceptable performance. This battery will be located underneath the autopilot, centred in the vehicle's frame, so its weight destabilizes it as little as possible, as shown in figure 4.28.

The camera also needs to be securely attached to the vehicle's frame, with the custom support described in section 3.2.3. The holder attaches to the slide bars underneath the vehicle's main frame so that the camera and its substantial weight are situated as close to the centre of mass as possible behind the GPS platform. The battery powering the engines and the autopilot, which is located on the underside of the carbon frame, is also moved slightly backwards from its centred position to make space for the camera and compensate for its weight in the front. Figure 4.29 shows how the vehicle's underside looks with the camera attached.

After the vehicle has been built, there are additional installation and calibration steps that must be carried out before it can fly, also contained in the guide mentioned above. Any simulation modes previously activated for testing must be deactivated from the Safety section of the vehicle configuration and the `MAV_1_CONFIG` parameter set to `TELEM2`, as described in section 3.2.2. Then all the different sensors present, both embedded on the flight

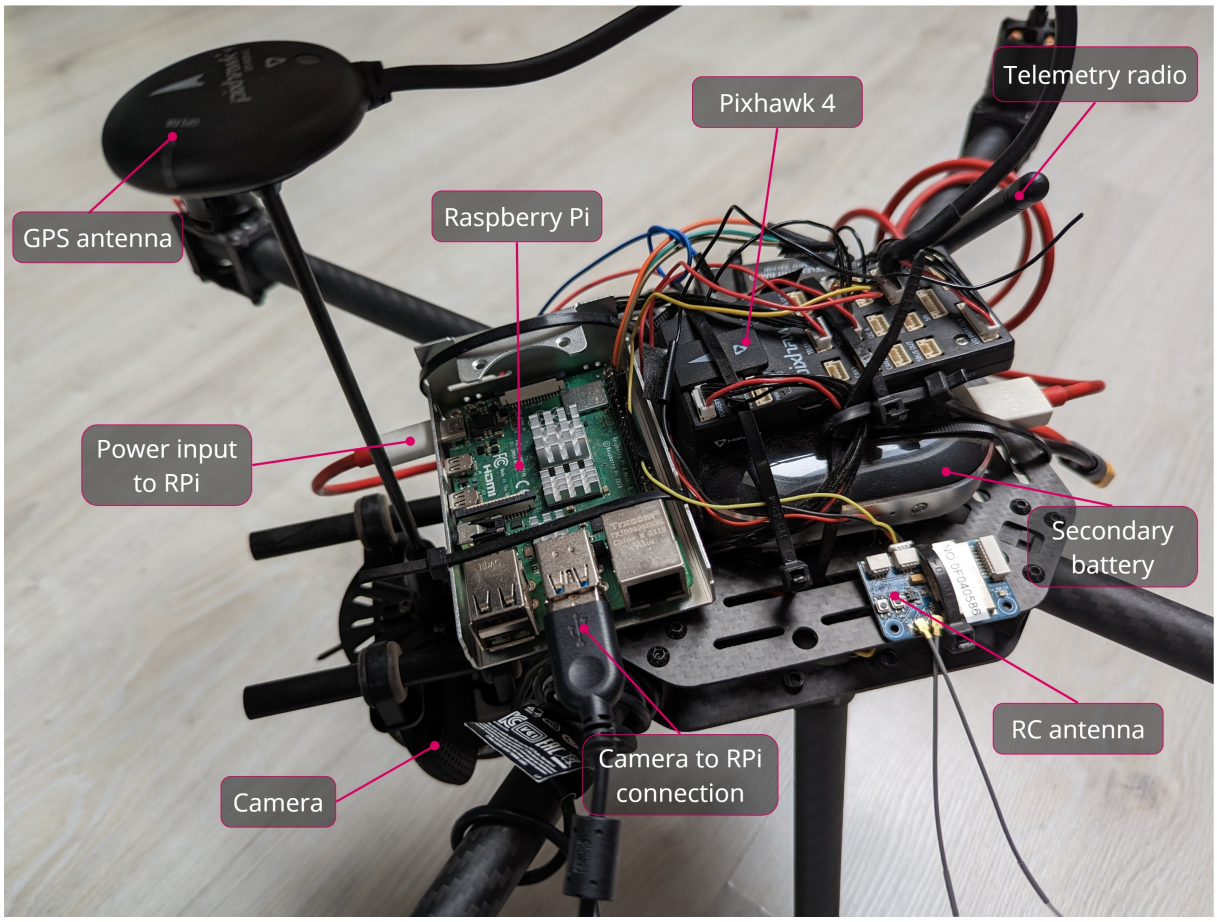


Figure 4.28: Complete build of the quadcopter with the main components highlighted

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

controller board and attached to the outside frame, need to be calibrated for this particular build. The QGroundControl 2.2.1 ground station application contains a configuration screen with all the calibration tools needed for the vehicle setup, shown in figure 4.30. The vehicle can be configured either by connecting the flight controller directly to the computer via the micro-USB port on its side or through a wireless connection by plugging the companion telemetry radio into the computer running QGroundControl.

4.4.2 Initial tests

Baseline flight with factory software

Once the vehicle is fully configured, the RC controller and QGroundControl can be used to test assisted takeoff and landing. At this point, the drone should be able to maintain stable flight while using autopilot-assisted flight modes like Position Mode, where the roll and pitch sticks control the acceleration over the ground of the vehicle in the forward/backward and left/right directions relative to the heading the vehicle is facing. The throttle controls the speed of ascent and descent. With the sticks centred, the vehicle will actively remain locked to a position in 3D space, compensating for wind and other forces. This is the safest manual mode to test that the standard autopilot works as expected.

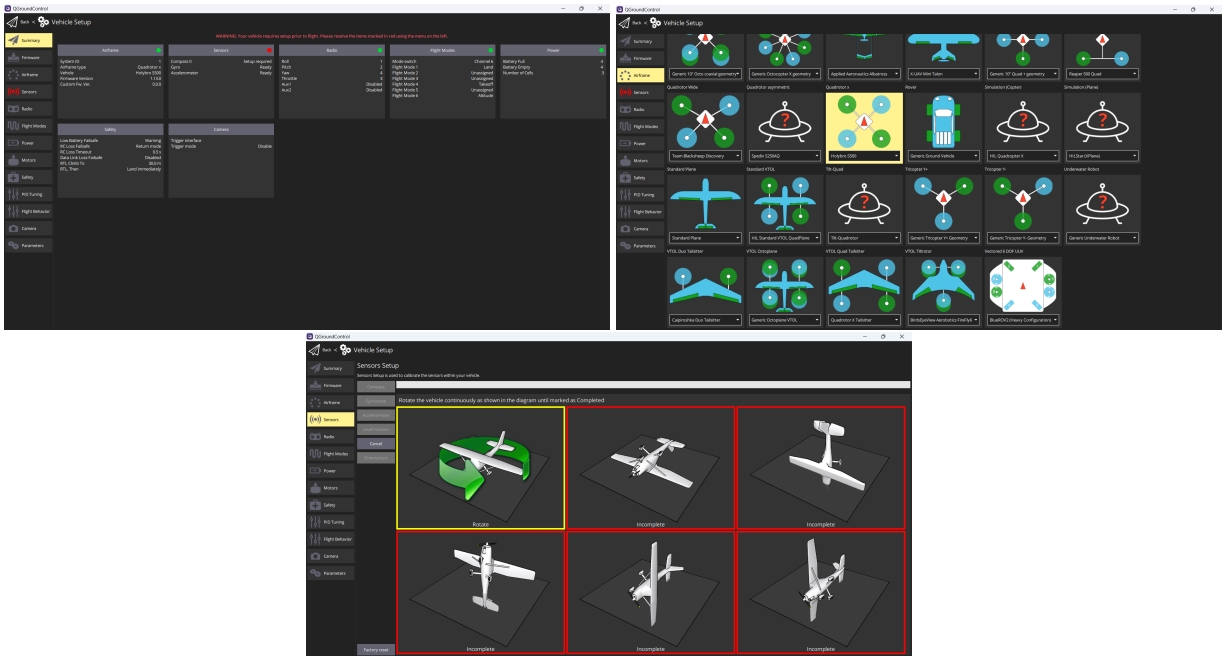


Figure 4.30: Screenshot from the QGroundControl calibration and setup tools used to configure the vehicle

Through QGroundControl it is possible to map the different switches in the RC controller to various autopilot commands. For this test, one of the switches with two positions will be mapped to arm/disarm, which controls whether the engines of the quadcopter can start or not. One of the switches with three positions will be mapped to the landing/takeoff/position flight modes, respectively. The main autopilot modes can be tested by switching between the available positions during flight. This configuration exhausts all the channels available in the RC controller employed. Other flight modes can be set by using the QGroundControl interface directly.

To carry out the flight, first, the main battery is connected to the socket in the power module. This starts up the autopilot, the GPS antenna, the telemetry radio, and the RC receiver. Afterwards, QGroundControl can be started on a computer connected to the second telemetry radio via USB. If everything has worked correctly, the ground station application will automatically connect to the vehicle and situate its position on a satellite map. Turning on the RC controller will likewise make it connect to the vehicle, as long as it has been paired correctly, as indicated in the guide linked in the first step of the build process. Once all the wireless connections have been established, the drone can take off by first switching to the armed state and then switching to the takeoff flight mode. While the drone is in the air, switching to the position flight mode will allow direct control through the joysticks in the controller.

Offboard computer flight with test tool

The second test flight will aim to ensure that the custom software can send takeoff and landing commands through a wireless MAVlink channel from the offboard computer (using the telemetry radio through the developed test tool). For this flight, the QGroundControl application cannot be connected to the vehicle since the Dronecontrol application will block the telemetry radio channel. The RC controller will therefore be used as a backup in case anything goes wrong with the software. At any moment, the controller can switch flight mode and override the input generated from the Dronecontrol application, recovering manual control. Since the Dronecontrol application can now easily arm the vehicle on its own while sending a takeoff command, the two-way switch of the controller will be mapped for all the tests from now on to the command to kill the power to the engines. This command could be helpful in edge cases to protect the vehicle or the surrounding area if the autopilot were to destabilize during takeoff and landing or completely lose control over the vehicle. Now, once the main battery is connected again to the power module, the test tool is run with the following command for a Windows or a Linux machine, respectively:

```
dronecontrol tools test-camera -r COM<X>:57600
```

or

```
dronecontrol tools test-camera -r /dev/ttyUSB0:57600
```

After successfully connecting to the vehicle, the T and L keys in the computer keyboard can be used for takeoff and landing, respectively. The O key can be used to set the autopilot in offboard flight mode to enable it to receive velocity commands. Afterwards, the WASD keys can be used to control the forward and sideways velocity of the vehicle and the QE keys to control its yaw velocity. Figure 4.31 displays the output on the computer's terminal window, where the connection process and the sent velocity commands are shown, and the output on the camera from the offboard computer. A video of the entire process can be found here[9].

Onboard computer flight with test tool

The third and last test flight in this section will ensure that the custom software can send takeoff and landing commands through a cabled MAVlink channel from the onboard computer, as well as ensuring that the onboard camera can obtain a good image of the field of view of the vehicle during flight. For that, the same tool will be used as in the last test, but

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

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

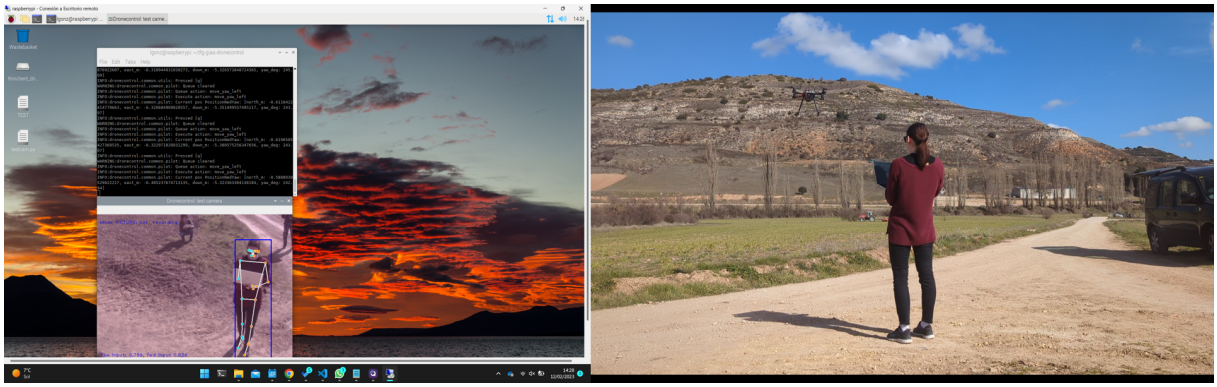


Figure 4.32: Pose detection algorithm running on images taken during flight

in this instance, it will be run on the Raspberry Pi, and the connection will be established through the wired serial link between this onboard computer and the Pixhawk autopilot board. Since the camera connected to the computer sending commands is now looking down on the pilot, it is possible to activate pose detection on the test images received from this onboard camera. To start the flight test, the main and secondary batteries need to be attached, respectively, to the power module and to the Raspberry Pi. After the onboard computer has started, the easiest way to control it is with a remote desktop connection through WiFi, as explained in section 3.1. By this connection, a terminal window can be opened on the desktop, and the following command run:

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

As opposed to the flight using the telemetry radio, in this test, the serial connection runs at a baudrate of 921600, which matches the configured baudrate on the `TELEM2` port of the Pixhawk board. The "-p" option enables pose detection in the output images. A video of the entire process can be found here[10] and an image extracted form it can be seen in figure 4.32.

4.4.3 Hand gesture control

During basic flight tests, all the connections and individual parts of the software were validated in actual flight. Now, it is time to integrate the piloting system with the image recognition results to test the developed vision-based control solutions. The first solution to be used in flight will be the hand-gesture guidance system, which runs on an offboard computer with more available processing resources and no dependence on battery-supplied power to work. The setup will be identical to the second test flight (4.4.2) with the telemetry radio as the serial link and the onboard companion computer turned off. Once the

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

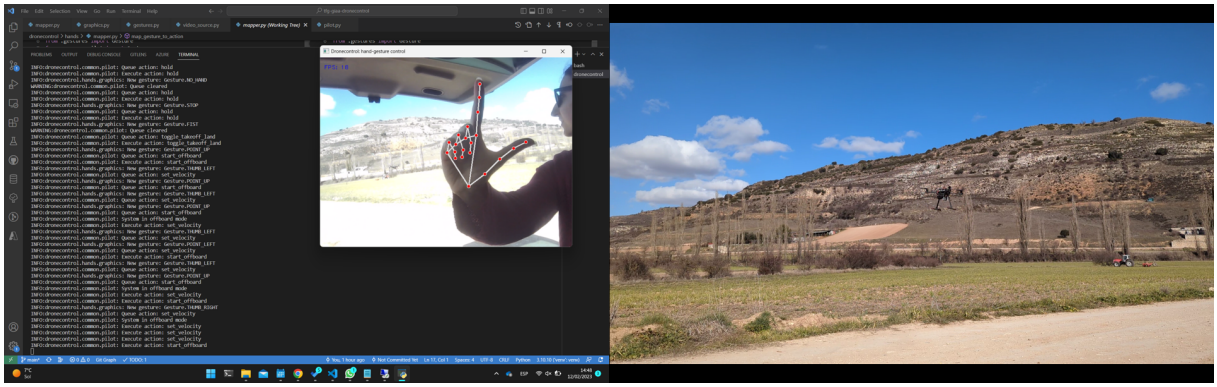


Figure 4.33: Image taken during flight controlled by the hand-gesture solution. The vehicle is moving forward.

autopilot board is powered up, the control solution can be started with the following command:

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

where <device> is the COM port or TTY device the telemetry radio is attached to, depending on the platform.

After the pilot connects, the image from the computer's webcam will appear on the screen with an outline over any detected hand. An open palm should be shown to the camera to start controlling the vehicle. Then, a closed fist will make the drone take off, and pointing up with the index finger will start the offboard flight mode. Afterwards, moving the index finger right or left will make the vehicle mirror the movement, and moving the thumb right or left will make the vehicle move forward and backwards, respectively. At any point during the test, an open hand will make the drone hover at its current place, as will losing sight of the controlling hand. A video of the entire process can be found here[11] and an image extracted form it can be seen in figure 4.33.

4.4.4 Target detecting, tracking and following

Finally, it only remains to test the follow control solution. In this section, the companion computer will be running the follow program, and it will be validated whether it can keep track of and follow a moving target during a non-simulated flight. The setup will be identical to the third test in section 4.4.2, without needing a wireless telemetry connection. The telemetry radio is, therefore, free to be used, for example, to track the vehicle's path through the QGroundControl application on a secondary, offboard computer. The control application will be started with the following command:

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

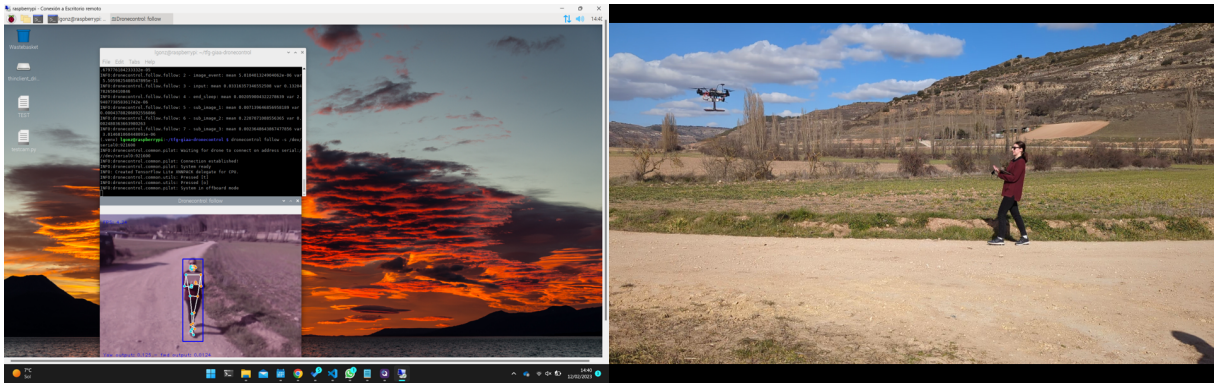


Figure 4.34: Terminal and image output of the dronecontrol follow solution running on the Raspberry Pi

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

This video[12] shows the process of getting the vehicle to takeoff (T key), activating offboard flight mode (O key), and starting movement tracking of the detected figure. Figure 4.34 shows an image extracted from this video.

The maximum frames per second managed by the program running on the Pi is around 6 FPS when the follow mechanism is engaged and around 8 FPS if it is disabled by switching out of offboard flight mode. In practice, this means that the person being tracked by the drone has to move quite slowly for the camera not to lose sight of them before the autopilot can send the command to the vehicle to move to the previously detected position. However, for a proof-of-concept scenario, this is an acceptable performance. At the end of the program run, the average loop time and average runtime for each of the tasks in the main loop are shown in the terminal. From the measures obtained for the test flight carried out, the average frame rate calculates to be 3.58 FPS. If these measurements are compared to those analyzed in section 4.3.2, as shown in figure 4.35, particularly to the test configuration most closely resembling actual flight with the autopilot board running on HITL mode and the companion computer powered by the secondary battery, it is possible to appreciate how close this configuration matches the behaviour during actual flight and therefore validating it as an appropriate environment to test the performance of this type of algorithms.

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

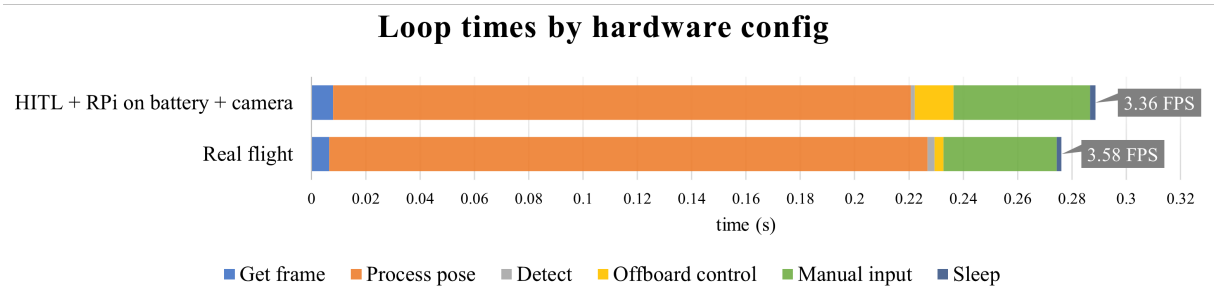


Figure 4.35: Terminal and image output of the dronecontrol follow solution running on the Raspberry Pi

Chapter 5

Conclusions

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

5.1 Evaluation of objectives

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

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

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

Third, it has been demonstrated how the development environment, all the presen-

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

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

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

5.2 Lessons learned

5.2.1 Applied knowledge

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

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

environment developed for the project (autopilot board, companion computer and simulation computer) and making them communicate with each other in a single network.

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

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

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

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

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

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

5.2.2 Acquired knowledge

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

- Python skills: how to organise projects with multiple submodules and create packages from own code, as well as experience in several standard libraries:

  - Asynchronous programming in Python with the `asyncio` library.
  - Matrix and vector handling with the `numpy` library.
  - Plotting customisable graphs in different UI platforms with the `matplotlib` library.

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

5.3 Future work

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

- Integration with more sensors: In this work, only a single camera was used as the input source for the computer vision algorithm. In future work, it would be beneficial

to investigate the integration of other sensors, such as lidar, radar, or multiple camera systems, to improve the performance of the computer vision algorithm.

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

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

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

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

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

Appendix A

Installation manuals

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

A.1 SITL: Development environment

This section describes the process of installing all the necessary applications to set up a development environment to run PX4's SITL simulation in a system similar to the one described in section 3.1. The instructions assume a computer running Windows 10/11 as the operating system and Windows Subsystem for Linux installed. PX4 details the installation steps of their source code for several platforms in their documentation [35], where Ubuntu is the recommended platform. To make it easier, the dronecontrol 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 dronecontrol application.

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

```
./install.sh
```

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

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

To test the installation of dronecontrol, execute:

```
dronecontrol 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 dronecontrol 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 dronecontrol:

```
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/in` 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:

```
dronecontrol tools test-camera -s -c
```

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

A.1.1 Installation of AirSim

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

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

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

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

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

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

```
dronecontrol tools test-camera -s -w
```

The follow control solution can be run from Linux with:

```
dronecontrol follow --sim 172.19.112.1
```

and from Windows with:

```
dronecontrol follow --sim -p 14550
```

A.2 HITL: Installation on a Raspberry Pi 4

Install dronecontrol

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

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

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

A.3 AirSim configuration file

```
{
    ”SettingsVersion”: 1.2,
    ”SimMode”: ”Multirotor”,
    ”ClockType”: ”SteppableClock”,
    ”Vehicles”: {
        ”PX4”: {
            ”VehicleType”: ”PX4Multirotor”,
            ”LockStep”: true,
            ”UseSerial”: ”<true: HITL, false: SITL>”
            ”UseTcp”: true,
            ”TcpPort”: 4560,
            ”ControlIp”: ”remote”,
            ”ControlPortLocal”: 14550,
            ”ControlPortRemote”: 14570,
            ”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
            }
        ]
    }
}
```

Appendix B

Command-line interface of the application

```
Usage: dronecontrol 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: dronecontrol 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: dronecontrol tools test-controller [OPTIONS]

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

```
Usage: dronecontrol 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: dronecontrol follow [OPTIONS]

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

References

[1] J.E. Gomez-Balderas et al. 'Tracking a ground moving target with a quadrotor using switching control: Nonlinear modeling and control'. In: *Journal of Intelligent and Robotic Systems: Theory and Applications* 70.1-4 (2013). cited By 70, 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). cited By 5, pp. 192–202. DOI: 10.1007/978-3-319-42297-8_19.

[3] R. Rysdyk. 'UAV path following for constant line-of-sight'. In: cited By 88. 2003. DOI: 10.2514/6.2003-6626.

[4] R. Bartak and A. Vykovsky. 'Any object tracking and following by a flying drone'. In: cited By 16. 2016, pp. 35–41. DOI: 10.1109/MICAI.2015.12.

[5] A. Chakrabarty et al. 'Autonomous indoor object tracking with the Parrot AR.Drone'. In: cited By 27. 2016, pp. 25–30. DOI: 10.1109/ICUAS.2016.7502612.

[6] J. Pestana et al. 'Vision based GPS-denied Object Tracking and following for unmanned aerial vehicles'. In: cited By 66. 2013. DOI: 10.1109/SSRR.2013.6719359.

[7] K. Haag, S. Dotenco and F. Gallwitz. 'Correlation filter based visual trackers for person pursuit using a low-cost Quadrotor'. In: cited By 15. 2015. DOI: 10.1109/I4CS.2015.7294481.

[8] A. Hernandez et al. 'Identification and path following control of an AR.Drone quadrotor'. In: cited By 45. 2013, pp. 583–588. DOI: 10.1109/ICSTCC.2013.6689022.

[9] 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). cited By 44, pp. 401–412. DOI: 10.1007/s10846-013-9969-5.

[10] P.-J. Bristeau et al. 'The Navigation and Control technology inside the AR.Drone micro UAV'. In: vol. 44. 1 PART 1. cited By 316. 2011, pp. 1477–1484. DOI: 10.3182/20110828-6-IT-1002.02327.

[11] 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). cited By 8. DOI: 10.1016/j.enconman.2022.115217.

[12] R.I. Naufal, N. Karna and S.Y. Shin. 'Vision-based Autonomous Landing System for Quadcopter Drone Using OpenMV'. In: vol. 2022-October. cited By 0. 2022, pp. 1233–1237. DOI: 10.1109/ICTC55196.2022.9952383.

[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). cited By 1, 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: cited By 0. 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: cited By 0. 2022, pp. 430–435. DOI: 10.1109/ICCAIS56082.2022.9990214.

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

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

[18] Anis Koubâa et al. 'Micro Air Vehicle Link (MAVlink) in a Nutshell: A Survey'. In: *IEEE Access* 7 (2019), pp. 87658–87680. DOI: 10.1109/ACCESS.2019.2924410.

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

[20] PX4 team. *Simulation | PX4 User Guide*. Dronecode foundation. URL: https://docs.px4.io/main/en/simulation (visited on 05/04/2023).

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

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

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

[24] AirSim. *PX4 Software-in-Loop with WSL 2*. Microsoft. URL: https://microsoft.github.io/AirSim/px4_sitl_wsl2/ (visited on 17/01/2023).

[25] 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).

[26] PX4 team. Dronecode foundation. URL: https://docs.px4.io/main/en/advanced_config/parameter_reference.html (visited on 17/01/2023).

[27] Holybro. *Pixhawk 4 - Pinouts*. URL: http://www.holybro.com/manual/Pixhawk4-Pinouts.pdf (visited on 14/01/2023).

[28] PX4 team. *Pixhawk 4 - PX4 User Guide*. Dronecode Foundation. URL: https://docs.px4.io/main/en/flight_controller/pixhawk4.html (visited on 14/01/2023).

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

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

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

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

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

[34] 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).

[35] 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).