



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...

*AGRADECIMIENTOS*

# **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.

*RESUMEN*

# **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.

*ABSTRACT*

# Todo list

■ Write: objectives . . . . .	1
■ Polish: time planning . . . . .	1
■ Write: layout . . . . .	1
■ Write: literature review . . . . .	3
■ Final polish: export all boards again from Miro without watermarks . . . . .	15
■ Take pictures: Images of the radios with computer, flight controller . . . . .	24
■ Write: power management, secondary battery . . . . .	25
■ Final check: link works . . . . .	26
■ Write: setup parameters to enable mavlink on telem2, configure baudrate . . . . .	28
■ Final polish: link to performance section . . . . .	28
■ Write: elaborate on the summary . . . . .	28
■ Write: Describe the actual new diagram, blue vs green arrows, data flow, ... . . . . .	30
■ Write: Add control functionality, describe connection options, link to cli in appendix	35
■ Polish: Link some code from test-camera tool . . . . .	35
■ Draw: hand vectors . . . . .	36
■ Write: add loop code and link to first paragraph . . . . .	37
■ Draw: Include diagram of angles for each pointy gesture ????? . . . . .	37

■ Images: simulators program output image with drawn lines for gestures . . . . .	37
■ Video: hand control in simulator . . . . .	37
■ Images: valid and invalid person detection from simulator camera output . . . . .	38
■ Draw: coordinate system figure . . . . .	38
■ Add some controller and follow code to show and explain . . . . .	39
■ Images: simulator and output, link video of run . . . . .	39
■ Write: PID tools, how they work and code . . . . .	39
■ PID library, Formula and coefficients . . . . .	39
■ Write: safety mechanisms, speed limits on pids, loss of signal, loss of vision, ... . . . . .	39
■ . . . . .	39
■ . . . . .	39
■ Get a better image for this . . . . .	43
■ Save video on GitHub and put link here, fix text to make sense . . . . .	43
■ Image of PX4 console + Unreal with Airsim scene . . . . .	46
■ Image of dronecontrol console + Unreal with Airsim scene + image output running test-camera . . . . .	47
■ video . . . . .	47
■ Videos for github . . . . .	49
■ Discuss velocity graphs as well for every section, mention velocity limits . . . . .	50
■ Controller should end up with (4, 0.1, 0) . . . . .	53
■ link video of test-controller tool . . . . .	56
■ TODO, and rerun . . . . .	57
■ Link video to follow run . . . . .	57

## *TODO LIST*

- check port . . . . . 59
- Edit for new battery . . . . . 60
- Close up image and description of custom connector . . . . . 60
- Check option path on latest OS . . . . . 60
- Fix table . . . . . 60
- Performance analysis with/out peripherals, detection only, detection and control together . . . . . 63
- Make video and link . . . . . 68
- Link video of screen during onboard test-camera flight . . . . . 70
- Check direction of thumb movement . . . . . 71
- Make return home altitude less before flying . . . . . 71
- Video should show side-by-side computer screen with terminal and image output and video from vehicle flying, synced . . . . . 71
- Save performance for flight tests, add measure tables in each section . . . . . 73
- Same recording as hand sec . . . . . 73

*TODO LIST*

# Contents

## List of figures

## List of listings

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	The Dronecontrol project . . . . .	1
1.2	Objectives . . . . .	1
1.3	Time planning . . . . .	1
1.4	Thesis layout . . . . .	1
<b>2</b>	<b>State of the art</b>	<b>3</b>
2.1	Literature review . . . . .	3
2.2	Methodology . . . . .	6
2.2.1	Software . . . . .	6
2.2.2	Hardware . . . . .	10
<b>3</b>	<b>Design and implementation</b>	<b>13</b>
3.1	Development environment and simulation . . . . .	14
3.2	System architecture . . . . .	22

3.2.1	Top level . . . . .	22
3.2.2	Offboard computer configuration . . . . .	24
3.2.3	Onboard computer configuration . . . . .	25
3.3	Software architecture . . . . .	30
3.3.1	Pilot module . . . . .	31
3.3.2	Video source module . . . . .	31
3.3.3	Vision control module . . . . .	33
3.4	Proof of concept: hand-gesture solution . . . . .	35
3.5	Final solution: human following . . . . .	37
3.5.1	Safety mode . . . . .	39
<b>4</b>	<b>Experiments and validation</b>	<b>41</b>
4.1	PX4 SITL simulation and validation . . . . .	41
4.1.1	PX4 SITL validation with AirSim . . . . .	45
4.2	PID controller validation . . . . .	49
4.2.1	Yaw controller . . . . .	50
4.2.2	Forward controller . . . . .	53
4.2.3	PID tuning validation . . . . .	55
4.3	PX4 HITL simulation and validation . . . . .	58
4.3.1	PX4 HITL validation with Raspberry Pi . . . . .	59
4.4	Quadcopter flight tests . . . . .	63
4.4.1	Build process . . . . .	64
4.4.2	Basic tests . . . . .	66

## CONTENTS

4.4.3 Hand gesture control . . . . .	70
4.4.4 Target detecting, tracking and following . . . . .	71
<b>5 Conclusiones y trabajos futuros</b>	<b>75</b>
5.1 Consecución de objetivos . . . . .	75
5.2 Aplicación de lo aprendido . . . . .	75
5.3 Lecciones aprendidas . . . . .	76
5.4 Trabajos futuros . . . . .	76
<b>A Installation process</b>	<b>77</b>
A.1 Installation of PX4 SITL . . . . .	77
A.2 Installation of Dronecontrol . . . . .	77
A.2.1 Installation on a Raspberry Pi 4 . . . . .	77
A.3 Installation of AirSim . . . . .	77
A.4 AirSim configuration file . . . . .	77
A.5 PX4 configuration . . . . .	77
A.6 Command-line interface of the Dronecontrol application . . . . .	77
<b>Referencias</b>	<b>79</b>

**CONTENTS**

# List of Figures

2.1	Main interface for the QGroundControl program . . . . .	8
2.2	Project interface for the Unreal Engine . . . . .	8
2.3	Side views and connector map for the Pixhawk 4 autopilot module. . . . .	11
2.4	Fully assembled X500 kit. . . . .	12
2.5	Raspberry Pi 4 Model B . . . . .	12
3.1	Mavlink messages exchanged between the simulator and the flight stack during simulation. . . . .	14
3.2	High-level overview of how different components interact in the AirSim simulator. . . . .	15
3.3	Network diagram between the different components that interconnect during software-in-the-loop simulation. . . . .	16
3.4	Connection diagram of how the three systems interact with each other during SITL simulation. . . . .	18
3.5	Connection diagram of how the three systems interact with each other during HITL simulation. . . . .	19
3.6	Screenshot from the Unreal Engine environment used for testing the computer vision solutions. . . . .	20
3.7	Top level diagram of the hardware/software interactions . . . . .	22
3.8	The Raspberry Pi 4 microcomputer, with its 40-pin GPIO header marked in red, and its pinout. . . . .	26

3.9	3D model for the camera support designed for the Holybro X500 frame. . . . .	27
3.10	A diagram of the wired connections from the Raspberry Pi 4 to the secondary battery and the flight controller (TELEM2) . . . . .	27
3.11	Onboard configuration connections . . . . .	29
3.12	Structure of the program . . . . .	30
3.13	Diagram of inheritance on the video source classes available to retrieve image data. . . . .	33
3.14	Landmarks extracted from detected hands by the MediaPipe hand solution. . . . .	35
3.15	Vector calculations . . . . .	36
3.16	Landmarks extracted from detected human figures by the MediaPipe Pose solution . . . . .	38
4.1	Outline for the validation process . . . . .	42
4.2	Gazebo simulator (left) and output from the PX4 terminal (right) after PX4's software-in-the-loop mode is started . . . . .	43
4.3	Gazebo simulator (left) and output from the PX4 terminal (right) after the takeoff command has been executed . . . . .	44
4.4	Hand detection algorithm running on images taken from the computer integrated webcam . . . . .	44
4.5	Single frame from the video showing the full execution of the hand-gesture control solution . . . . .	45
4.6	. . . . .	46
4.7	. . . . .	47
4.8	Single frame from the video showing the movement of the drone in response to changes in position of the tracked person . . . . .	48

## LISTOFFIGURES

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. . . . .	49
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. . . . .	50
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. . . . .	50
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. . . . .	51
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. . . . .	51
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. . . . .	52
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. . . . .	53
4.16	Starting position of the simulator for tuning the forward controller. The human model is situated 450 units forward and centered from the vehicle position. . . . .	53
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. . . . .	54
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. . . . .	55
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. . . . .	55
4.20	. . . . .	56
4.21	. . . . .	57
4.22	Pixhawk 4 board connected to the computer running the AirSim simulator and the dronecontrol application . . . . .	58

4.23 Some figures that show these tests . . . . .	59
4.24 Picture of Rpi + power module + battery + pixhawk . . . . .	61
4.25 a) Picture of Rpi raspi-config    b) close-up of pixhawk to pi cable connection	61
4.26 a) RPi desktop with pose output    b) AirSim on Windows . . . . .	62
4.27 Development kit for the Holybro X500. . . . .	64
4.28 Full build annotated with arrows . . . . .	65
4.29 . . . . .	66
4.30 Screenshot from the QGroundControl calibration and setup tools used to configure the vehicle . . . . .	67
4.31 Picture from flight tests . . . . .	68
4.32 a) Terminal output from the test-camera tool running on an offboard computer and b) Output from the camera of the offboard computer towards the vehicle in flight . . . . .	69
4.33 Pose detection algorithm running on images taken during flight . . . . .	70
4.34 Image taken during flight controlled by the hand-gesture solution. Vehicle is taking off . . . . .	72
4.35 Image taken during flight controlled by the hand-gesture solution. Vehicle is moving to the right. . . . .	72
4.36 Image taken during flight controlled by the hand-gesture solution. Vehicle is moving forward . . . . .	73
4.37 Terminal and image output of the dronecontrol follow solution running on the Raspberry Pi . . . . .	74

# List of Listings

3.1 Example of how the communication to the flight stack is established through asyncio and the mavsdk library . . . . .	32
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. . . . .	33

*LIST OF LISTINGS*

# **Chapter 1**

## **Introduction**

### **1.1 The Dronecontrol project**

### **1.2 Objectives**

Write: objectives

### **1.3 Time planning**

Polish: time planning

### **1.4 Thesis layout**

Write: layout



# Chapter 2

## State of the art

### 2.1 Literature review

Write: literature review

1. Ref [10] Indoor Follow Me Drone Tracking through acoustic signals in indoor environments produced by speakers on the drone and received by a mobile device. Discards CV for stability and processing power. Controller on mobile device with three degrees of freedom, pitch and yaw through MPC and roll through PID, with prediction of target's movement.
  - Identification and path following control of an AR.Drone quadrotor [-Link-](#)  
Path following application based on IMC position controllers, external webcam video stream.
  - Tracking a ground moving target with a quadrotor using switching control:  
Nonlinear modeling and control [-Link-](#)  
Tracking of a moving target on ground, embedded camera, use of 2-dimensional images to compute the relative 3-dimensional position and translational velocity of the UAV with respect to the target, switching controllers.
  - A computer vision and control algorithm to follow a human target in a generic environment using a drone [-Link-](#)  
Tracking and following a generic human target, using the HOG classifier, and on local brightness information, using the optical flow algorithm.
  - Framework for autonomous on-board navigation with the AR.Drone [-Link-](#)  
All sensing and computations on-board, three systems to autonomously following several trajectory patterns, visually estimate its position and detecting and following a person.

- A robust real-time embedded vision system on an unmanned rotorcraft for ground target following [-Link-](#)
  
  - UAV path following for constant line-of-sight [-Link-](#)  
Flight path guidance and synchronous camera angles to observe a target, analytic expressions are derived for trajectories required for constant line-of-sight orientation relative to the aircraft.
  
  - Reactive control of autonomous drones [-Link-](#)  
Reactive control that supersedes the time-triggered approach, control decisions are taken only upon recognizing the need to, based on observed changes in the navigation sensors, rate of execution dynamically adapts to the circumstances.
  
  - Correlation filter based visual trackers for person pursuit using a low-cost Quadrotor [-Link-](#)  
Correlation filters for short-term tracking and a redetection strategy based on tracking-learning-detection (TLD), flight experiments in unconstrained environments using human targets and an existing visual servoing controller.
  
  - The Navigation and Control technology inside the AR.Drone micro UAV [-Link-](#)  
Navigation and Control technology embedded in a recently commercialized micro Unmanned Aerial Vehicle (UAV), the AR.Drone.
  
  - Vector field path following for miniature air vehicles [-Link-](#)  
Method for accurate path following for miniature air vehicles is developed, vector-field path-following control laws are developed for straight-line paths and circular arcs and orbits.
  
  - Trajectory tracking for autonomous vehicles: An integrated approach to guidance and control [-Link-](#)
  
  - Trajectory-tracking and path-following of underactuated autonomous vehicles with parametric modeling uncertainty [-Link-](#)
  
  - Combined trajectory tracking and path following: An application to the coordinated control of autonomous marine craft [-Link-](#)  
Good trajectory tracking performance while keeping some of the desired properties normally associated with path following
  
  - Understanding the basis of the kalman filter via a simple and intuitive derivation [lecture notes] [-Link-](#)
- 
2. Ref [12] Computer Vision Based General Object Following for GPS-denied Multicopter Unmanned Vehicles, Ref [13] (6) Vision based GPS-denied Object Tracking and Following for Unmanned Aerial Vehicles Object tracker + IBVS controllers

3. Ref [4] Autonomous Indoor Object Tracking with the Parrot AR.Drone Deformable Object Tracking (CMT) tracker with image-based visual servoing (IBVS), relies directly on image features to compute control values as opposed to position-based servoing
4. Ref [2] Any Object Tracking and Following by a Flying Drone Offboard control solution though WiFi on Parrot AR Drone with computer vision (Training-Learning-Detection), with two PID controllers for forward/backward (scale estimation) and yaw (center position).

## 2.2 Methodology

This section describes the software programs and libraries employed in this project, as well as the hardware used to carry out flight tests for the developed application.

### 2.2.1 Software

#### PX4 autopilot

PX4<sup>1</sup> is a professional open-source autopilot flight stack developed in C++ by developers from industry and academia, and supported by an active world-wide community, it powers all kinds of vehicles from racing and cargo drones through 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 and custom builds made from scratch, as well as many additional kinds of 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<sup>2</sup>, that includes the QGroundControl ground station, Pixhawk hardware, and MAVSDK 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 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<sup>3</sup> is a very 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.

---

<sup>1</sup><https://px4.io/>

<sup>2</sup><https://www.dronecode.org/>

<sup>3</sup><https://mavlink.io/en/>

MAVSDK<sup>4</sup> 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 written in C++. The libraries provides 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 for a ground station or mobile device.

## QGroundControl

QGroundControl<sup>5</sup> is an open-source ground control station for the MAVLink protocol that provides flight control and mission planning for any MAVLink enabled drone. It is designed with focus on ease of use for professional users and developers. QGroundControl connects automatically to flight controllers running the PX4 Autopilot, either through cable or wireless connection to the computer running the ground station, as well as to any 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, take-off and landing. It provides a map interface to keep track of the vehicles GPS location as well as to select target location points for planning flight missions, where the vehicle goes through each planned point at the designated speed and altitude. Figure 2.1 shows the interface of the application on a Windows system.

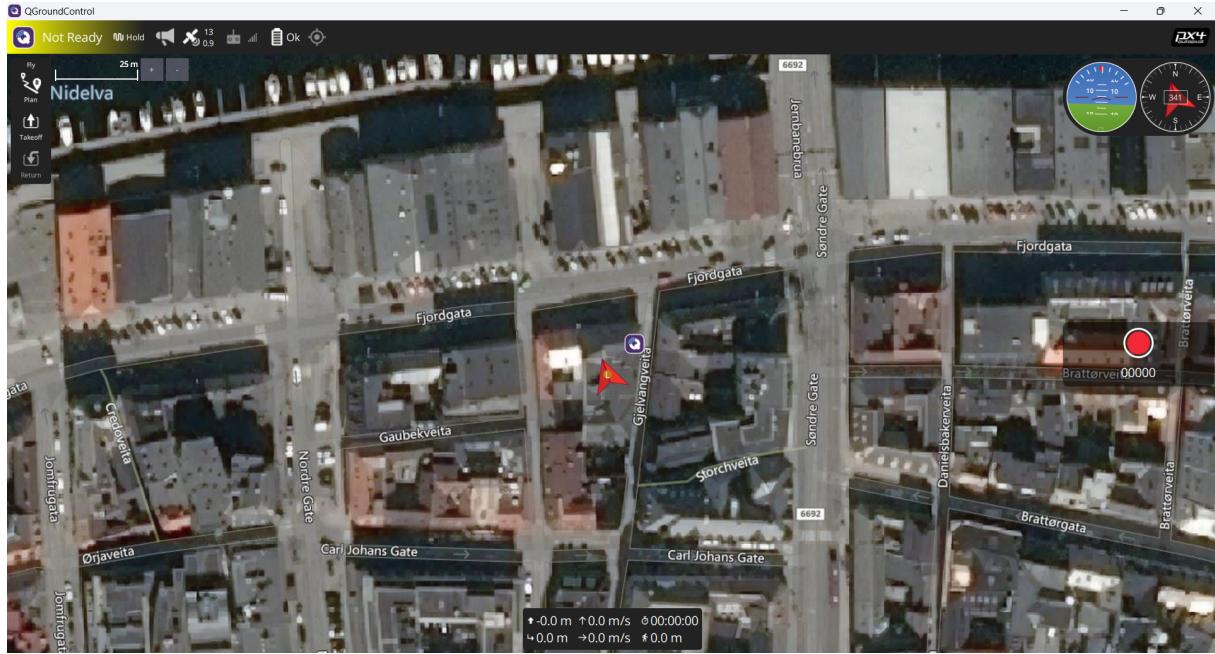
## Unreal Engine

Unreal Engine<sup>6</sup> 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 and the engine is now used on 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, employing the engine's support for real-time graphics. Figure 2.2 shows the program's interface when a project is loaded into the engine.

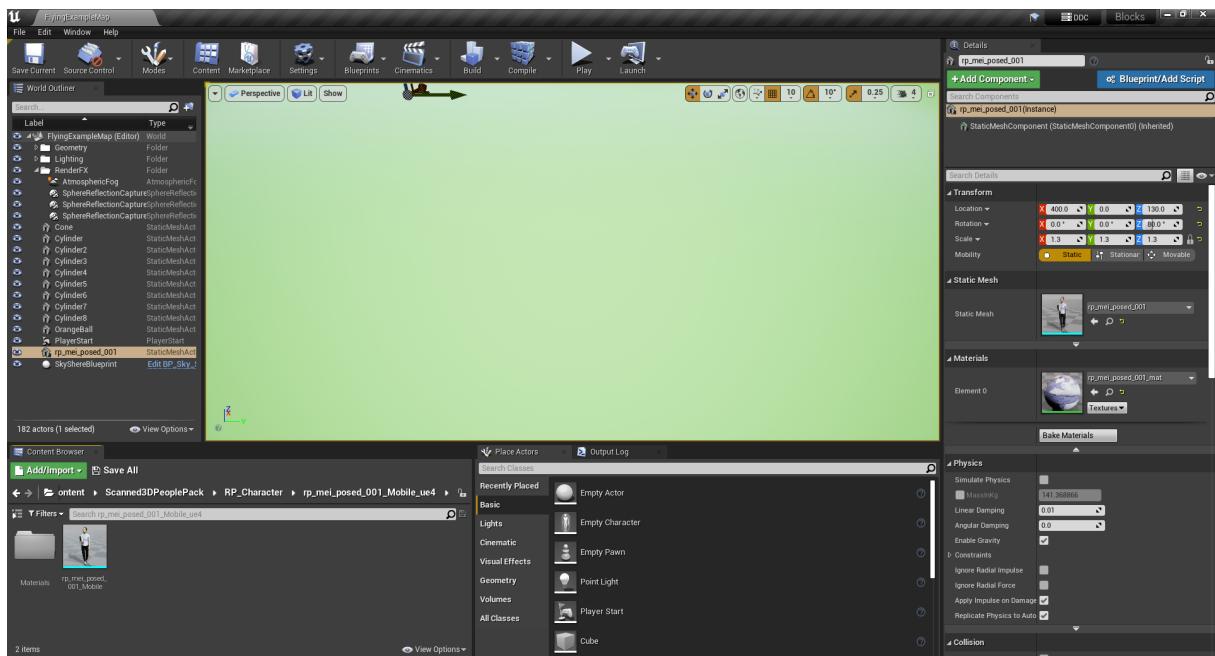
<sup>4</sup><https://mavsdk.mavlink.io/main/en/index.html>

<sup>5</sup><http://qgroundcontrol.com/>

<sup>6</sup><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

AirSim <sup>7</sup> is a simulator for drones, cars and more, built on Unreal Engine and developed by Microsoft. 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) with PX4 for physically and visually realistic simulations. It is developed as an Unreal plugin that can simply be dropped into any 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.

## MediaPipe

Mediapipe <sup>8</sup> 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 processing accelerated even on common 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 real-time perception of human pose, face landmarks and hand tracking that can enable a variety of impactful applications, such as fitness and sport 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 largest source code host with over 350 million repositories.

## 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 has

---

<sup>7</sup><https://microsoft.github.io/AirSim/>

<sup>8</sup><https://google.github.io/mediapipe/>

Python-wrappers available that take advantage of the general purpose programming language while maintaining the speed of the underlying computationally intensive C++ code. OpenCV-Python employs the Numpy library, which is designed for highly optimized numerical operations with syntax based on the MATLAB style. In Python, all the OpenCV array structures are converted to and from Numpy arrays. This makes it easy to work together with other libraries that use Numpy like Matplotlib for plotting graphs.

## 2.2.2 Hardware

### Pixhawk 4

The Pixhawk 4 is an advanced autopilot module designed and produced by Holybro<sup>9</sup>, a UAV parts and kits manufacturer, in collaboration with the Dronecode Project team. It is based on the Pixhawk-project<sup>10</sup> FMUv5 open hardware design, and it is optimized to run PX4 on the NuttX<sup>11</sup> 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 vehicle using the PX4 flight stack natively without any other sensors. It includes as well 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<sup>12</sup> is a quadcopter designed by Holybro to use with PX4. It comes as a ready-to-assemble development kit made up of a full carbon-fiber twill frame and the Pixhawk 4 flight controller. The kit includes as well 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 does not require any specialized tools for the process. Figure 2.4 shows the final result of the building process.

### Raspberry Pi 4

The Raspberry Pi<sup>13</sup> 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 its

---

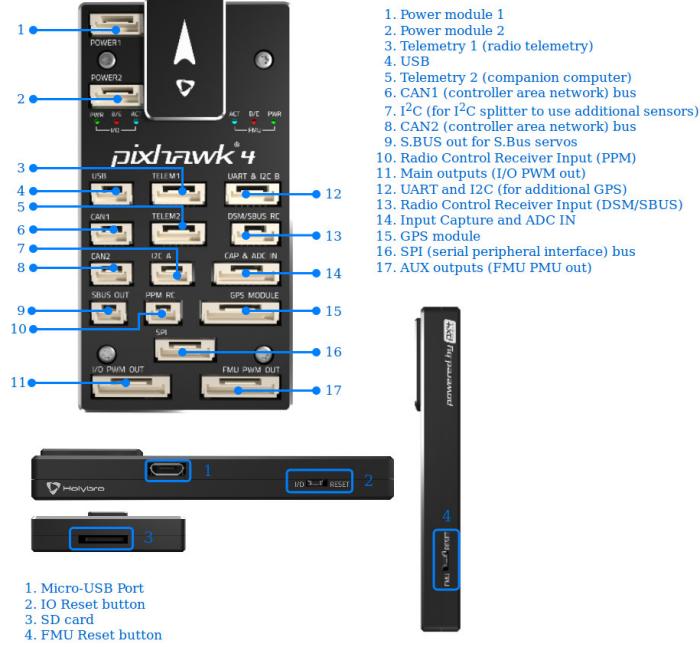
<sup>9</sup><https://shop.holybro.com/>

<sup>10</sup><https://pixhawk.org/>

<sup>11</sup><https://nuttx.apache.org/>

<sup>12</sup>[https://docs.px4.io/main/en/frames\\_multicopter/holybro\\_x500\\_pixhawk4.html](https://docs.px4.io/main/en/frames_multicopter/holybro_x500_pixhawk4.html)

<sup>13</sup><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* [14].

predecessors, getting a significant increase in processing power, enhanced 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* [14].

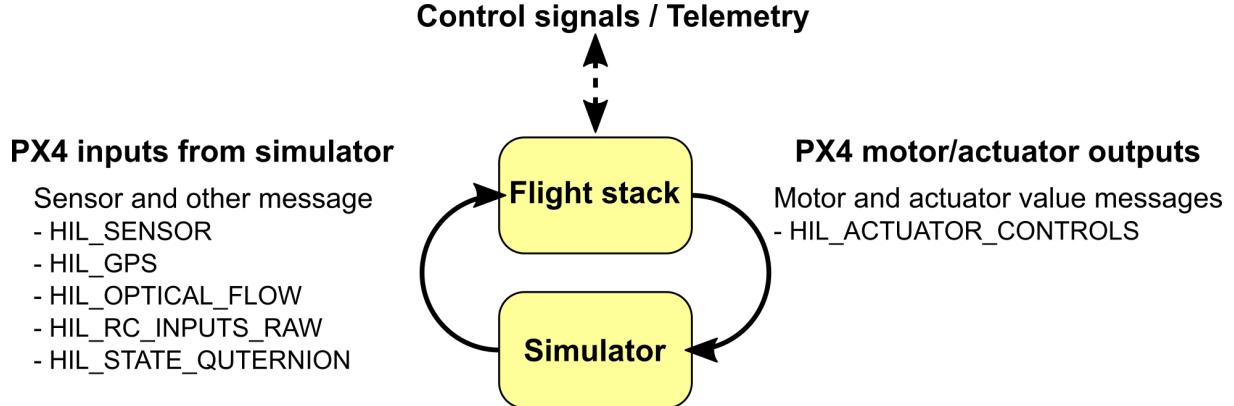


**Figure 2.5:** Raspberry Pi 4 Model B

Source: Wikimedia Commons [1].

# **Chapter 3**

## **Design and implementation**



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

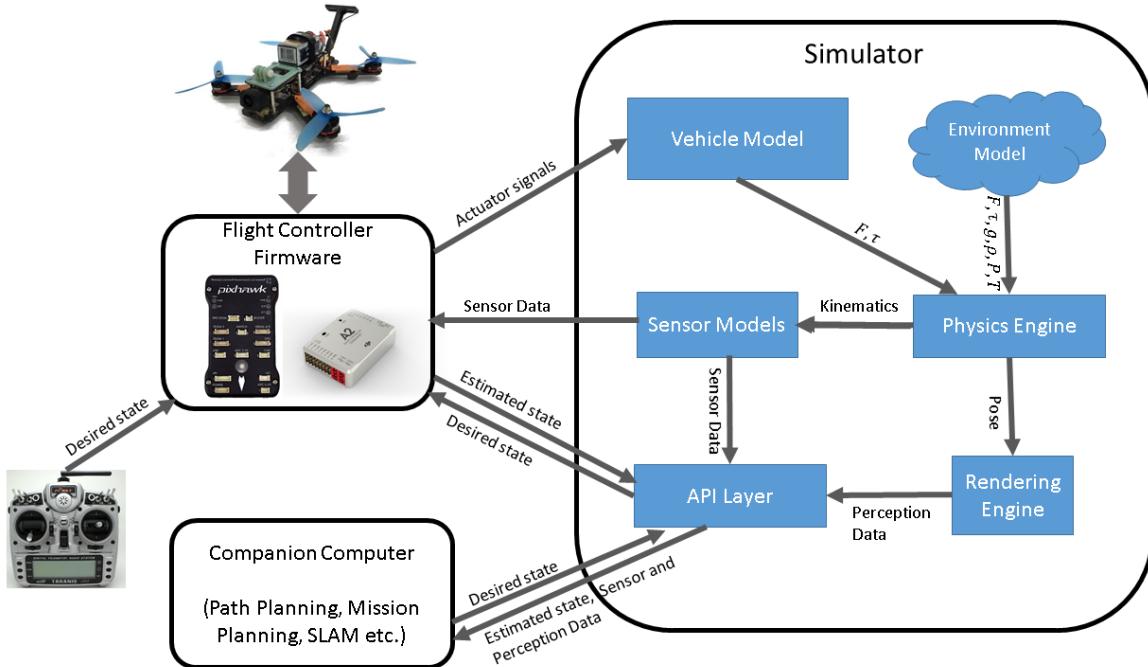
Source: Adapted from *PX4 User Guide* [14].

### 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 increasing the safety of the process by only running on the vehicle software that has already been tested to an acceptable point.

Simulators allow PX4 flight code to control a computer-modeled vehicle in a simulated “world” that can be interacted with in the same ways as with a real vehicle, using QGroundControl, an offboard API or a radio controller/gamepad. PX4 supports two different simulation modes: software-in-the-loop (SITL), where the flight stack runs on an external computer, and hardware-in-the-loop (HITL), where it uses a simulation firmware 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 of messages defined within XML files between drones, ground control stations, and other MAVLink systems [7]. 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



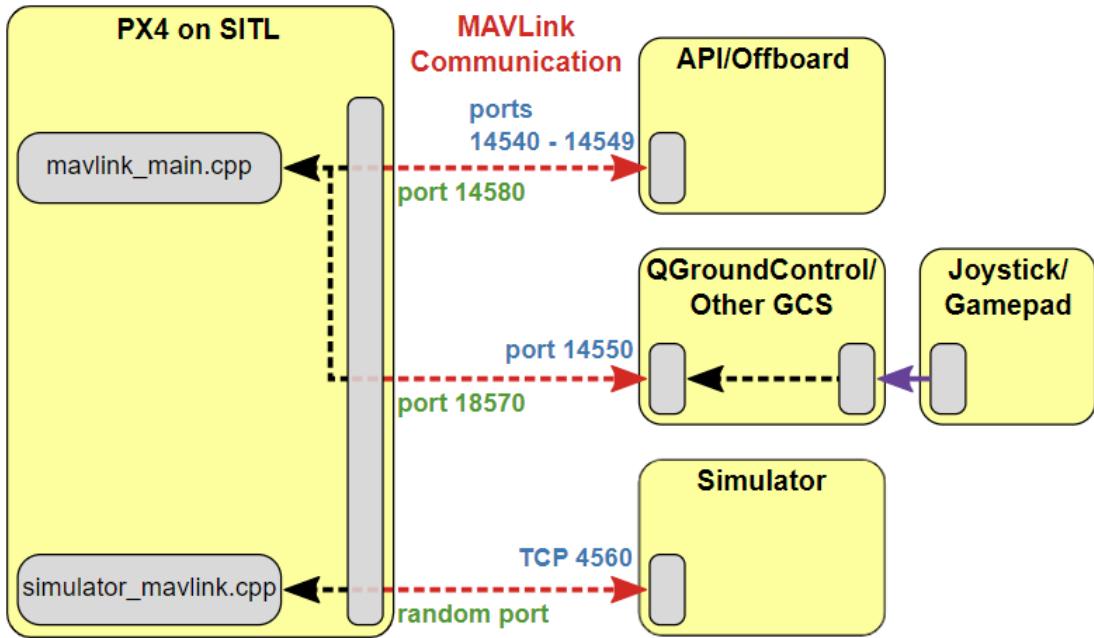
**Figure 3.2:** High-level overview of how different components interact in the AirSim simulator.

Source: Adapted from *Aerial Informatics and Robotics Platform* [16].

on the same computer using the UDP transport protocol, which 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 flight.

There are many options for simulators supported by PX4, like Gazebo, a powerful 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). This more resource-intensive cross-platform simulator 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 as well as for 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 as well a Python library called `airlib`<sup>1</sup> 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.

<sup>1</sup><https://pypi.org/project/airsim/>



**Figure 3.3:** Network diagram between the different components that interconnect during software-in-the-loop simulation.

Source: Adapted from *PX4 User Guide* [14].

Final polish: export all boards again from Miro without watermarks

A high-level overview of the simulator architecture and the way different components interact with each other can be seen in figure 3.2. The API layer present in the Figure inside the simulator environment refers to AirSim's own `airlib` library, which exposes with high-level functionality to send control commands to the flight controller directly. However, in order to share the same control code when simulating flight and in real flight without and not depend 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 the companion computer.

In order to run software-in-the-loop simulation, the PX4 firmware needs to be built from the source code on the Linux platform where it is going to 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 runs 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). External developer application like Dronecontrol use an offboard API, in this case MAVSDK, and therefore listen to PX4's remote UDP

port 14540. All ports in the range 14540-14549 can be used to connect offboard API, for example in the case of controlling multiple vehicles at the same time. PX4's remote UDP Port 14550 is used for communication with ground control stations, which are expected to listen for connections on this port (QGroundControl listens to this port by default). 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 on either the same computer or different computers on the same network.

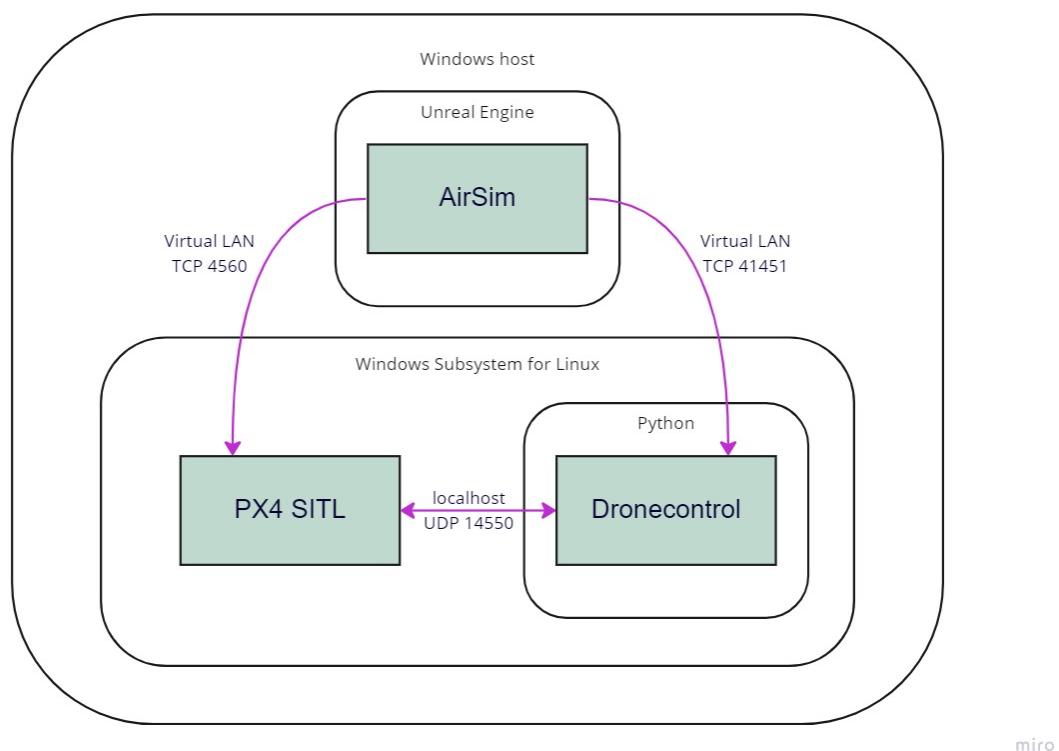
Since the purpose of using AirSim as a simulator is to run it on a Windows computer and the PX4 software-in-the-loop stack runs on Linux, it is necessary to run a virtualized Linux OS in parallel on the Windows computer and setup a local network so that the simulator and the flight controller firmware 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)<sup>2</sup>, 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 environment (including most command-line tools, utilities, and applications) directly on Windows, unmodified, without the overhead of a traditional virtual machine or dualboot setup. The full steps needed for to configure the system are detailed in Section ??.

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) connect through the localhost network on the UDP port defined by PX4 for Offboard APIs 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 which uses by default a TCP connection on port 41451.

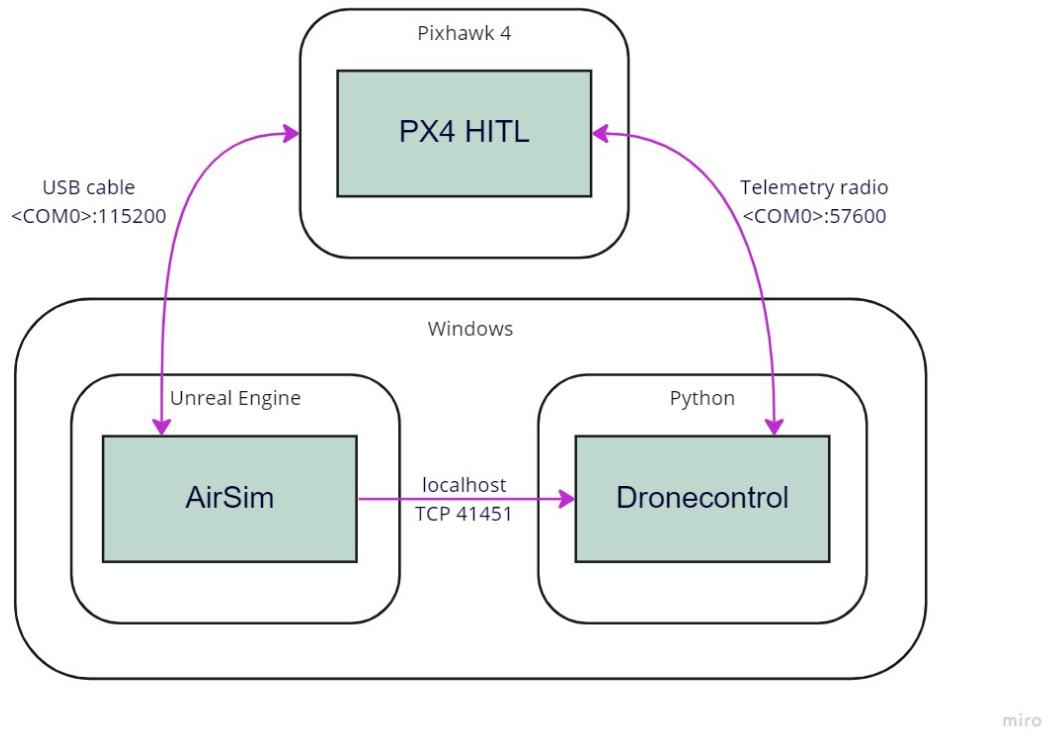
In the case of hardware-in-the-loop simulation, the main difference with SITL is that the flight stack firmware runs on a physical flight board using a special configuration. In HITL all motors/actuators are blocked, but internal software is fully operational. This configuration adds an additional separated operating system and to make this mode of testing simpler, since the WSL environment is no longer needed to run the flight stack, it is possible to move the execution of the Python module to Windows. This eliminates the need to add further configuration to allow the external flight controller to communicate with the internal WSL network, which is by default only accessible by its Windows host computer. Moreover, now that PX4 runs on a separate piece of hardware, it is necessary to establish two separate physical connections to the Windows computer, so that both the simulator and the Python app can communicate through their own channel to the flight controller,

---

<sup>2</sup><https://docs.microsoft.com/en-us/windows/wsl/about>



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



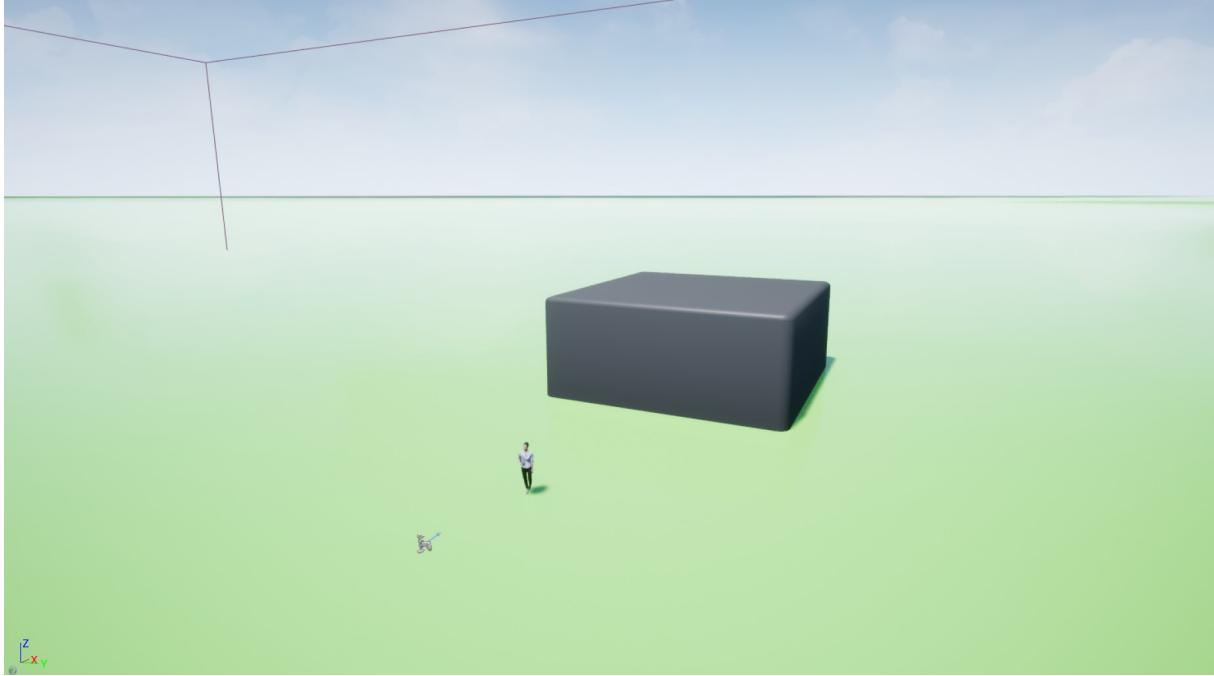
**Figure 3.5:** Connection diagram of how the three systems interact with each other during HITL simulation.

either wired, to the microUSB port or an unused telemetry port on the flight controller, or wireless through a telemetry radio.

Figure 3.5 shows the chosen connections to execute tests in HITL mode. The Windows machine runs both AirSim and the Python interpreter, which communicate through the localhost network using TCP. The board running PX4 connects to the simulator through a USB to microUSB cable, which is setup to work with a baudrate of 115200, and to the developed 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. It works by creating 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 a release for the other main 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, it works best on Windows



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

[11]. 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 block 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 to the project and a custom vehicle.

The environment used in this project for testing the Dronecontrol application is derived from the built-in AirSim environment. 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, and a few of the preset blocks with a green color to give a 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 of the computer vision solution. The model is part of a free asset library of human models made by Renderpeople [15] obtained in the Unreal Marketplace. An image of the testing environment can be seen in figure 3.6.

AirSim is compatible both with SITL and HITL simulation. However, it is necessary to set it up to work with an external PX4 flight controller instead of AirSim's own internal SimpleFlight. Appendix A.4 shows all the required settings for configuring these modes in AirSim. To run the simulator in Windows and the flight controller in WSL, the IP of the Windows host in the vEthernet (WSL) network needs to be provided in the AirSim settings. This process is detailed further in the AirSim documentation [[airsim-doc-wsl](#)]. In brief, first the Unreal project with AirSim has to be set into play mode and then the PX4

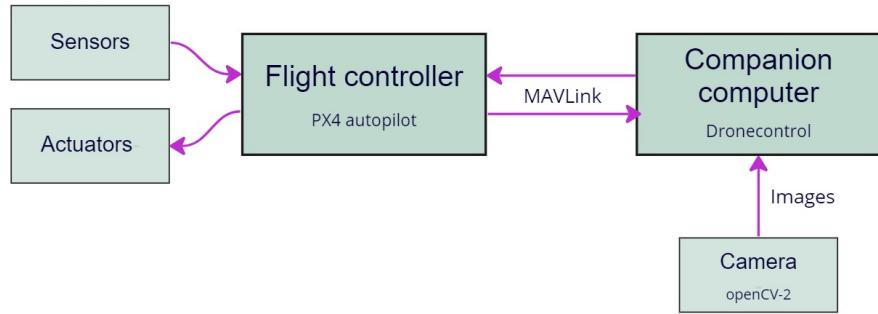
SITL flight controller built and run to attach to an already running simulator. Inside the project in WSL, the flight controller can be run for AirSim by using the provided shortcut script<sup>3</sup>:

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

Once both 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.

---

<sup>3</sup><https://github.com/l-gonz/tfg-giaa-dronecontrol/blob/main/simulator.sh>



miro

**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 Unmanned Aerial Vehicle (UAV) through the analysis of the images taken by a camera. Since the processing power needed to work with the images recorded is superior to that offered by the autopilot flight controller it becomes necessary the use of an additional companion computer that will control the camera and employ machine learning to extract useful features from the images, as well as transform those features 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, that will run the PX4 autopilot 2.2.1, the companion computer, that 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, either through a wireless radio link or a cabled serial connection between the two. The camera is connected to the companion computer through a cable into an USB port on the computer. Typically, the type of connection between the flight controller and the computer depends on the desired setup for the system so in the case where the camera, and therefore the companion computer, flies onboard the vehicle it is most convenient to use a direct wire connection between the two, as it provides a faster and more stable link. This onboard configuration is further detailed in Section 3.2.2. On the other hand, in the case where the companion computer will be acting more like a ground station on an offboard configuration it becomes strictly necessary to communicate with the flight controller through a wireless connection. For this purpose, a pair of telemetry radios provided with the Development Kit of the Holybro X500 (2.2.2)

are used. The complete setup required for this configuration is described in Section 3.2.3.

In this project, the flight controller is driven by the PX4 software (2.2.1) and the hardware employed is optimized for this flight stack. PX4 uses sensors to determine the vehicle state, which is needed for stabilization and to enable autonomous control. It minimally requires a gyroscope, accelerometer, magnetometer (compass) and barometer. A GPS or other positioning system is needed to enable all automatic flight modes, and some assisted ones. PX4 uses outputs to control motor speed, flight surfaces like ailerons and flaps, camera triggers, parachutes, grippers, and many other types of payloads. Many PX4 drones use brushless motors that are driven by the flight controller 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. PX4 drones are mostly commonly powered from Lithium-Polymer (LiPo) batteries. The battery is typically connected to the system using a Power Module or Power Management Board, which provide separate power for the flight controller and to the ESCs for the motors. A Radio Control (RC) system is used to manually control the vehicle. It consists of a remote control unit that uses a transmitter to communicate stick/control positions with a receiver based on the vehicle. Some RC systems can additionally receive telemetry information back from the autopilot. Telemetry Radios 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, inspect telemetry in real-time, change a mission on the fly, etc.

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, etc). However, it is also possible to simulate this hardware on a standard Linux system by building the PX4 source code on a computer with this operating system. This process is described in the development environment section (3.1).

The Dronecontrol application that runs on the companion computer has been developed using the Python programming language <sup>4</sup>. It offers good advantages for a project of this 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 support for object-oriented programming. Most importantly, Python is widely used and its official package manager pip greatly simplifies the use of external libraries and which gathers in its package index <sup>5</sup> thousands of standard utilities, including many machine learning and image processing projects and all the required libraries for interacting with PX4 through the MAVLink protocol (MavSDK). As it is an interpreted language, it can run easily in any system with Python installed without the need to compile separate binaries for different operating systems.

---

<sup>4</sup><https://www.python.org/>

<sup>5</sup><https://pypi.org/>

The next sections explore deeper into the differences between the two configurations mentioned before: offboard computer (or ground station) and onboard 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 but that can instead stay on ground while the vehicle flies. This has the advantage that it permits a simpler configuration, without having to be concerned with low-level hardware interactions between the two systems or powering of the companion computer while in flight, as well as allowing the use of a more powerful computer for image processing. However, it also requires that the camera stays connected to computer on the ground so it cannot use images from the perspective of the drone 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 processing can be feasible with the current technology but fall out of the scope of this project.

The wireless link is established in this instance through a pair of telemetry radios that connect to a telemetry port on the flight controller and to a USB port in the companion computer, respectively. Since the Pixhawk 4 is configured by default to used its TELEM1 port for this purpose, no additional configuration is needed when using that port. In the companion computer, applications like the QGroundControl<sup>6</sup> ground station software which is part of the Dronecode Project are able to detect automatically a telemetry radio inserted into any of the USB ports of the host computer. Additionally, other applications using the MavSDK (2.2.1) library can establish connection 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<sup>7</sup>. It is a small, light and inexpensive open source radio platform that typically allows ranges of better than 300m “out of the box” (the range can be extended to several kilometres with the use of a patch antenna on the ground). The radios are offered either as 915Mhz (Europe) or 433Mhz (US) so they can be used in different regions and comply with the regulations for frequency, hopping channels and power levels. They offer 2-way full-duplex communication through an adaptive TDM UART interface and their antenna allows for an adjustable 100-mW-maximum output power and -117 dBm receive sensitivity. The link is established by default with a baudrate of 57600 (max bits per second on a serial channel) and it can provide air data rates of up to 250 kbps.

---

<sup>6</sup><http://qgroundcontrol.com/>

<sup>7</sup><http://www.holybro.com/product/transceiver-telemetry-radio-v3/>

Take pictures: Images of the radios with computer, flight controller

### 3.2.3 Onboard computer configuration

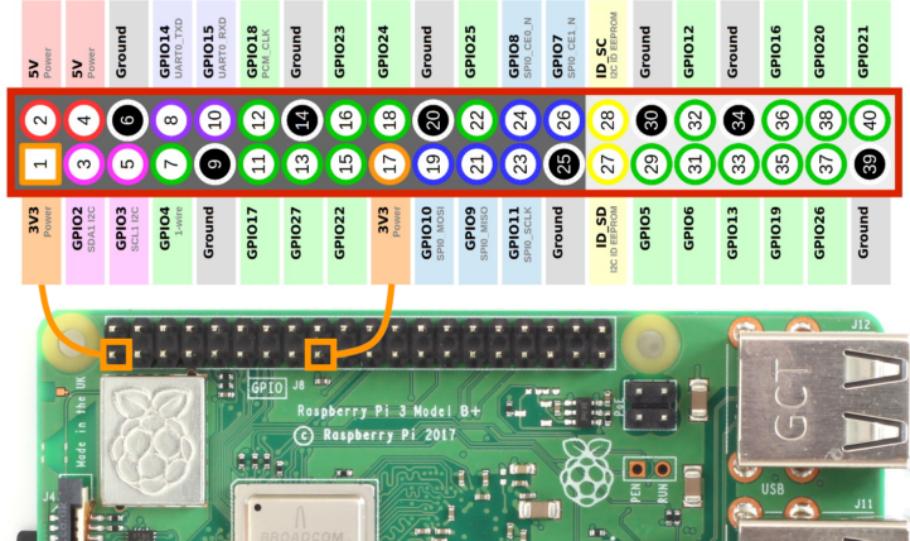
The second way of configuring the interaction between the flight controller and the companion computer consists of integrating both together on board the UAV. In this case, the connection is done through a direct cable between the serial port in the flight controller and a USB port in the companion computer. The camera will then be connected via cable as well to the companion computer and oriented in the vehicle in a way that allows for the best possible perspective during flight. This configuration makes it possible to develop new control solutions based on images taken directly from the vehicle and 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 maintain a feedback loop for a more stable guidance.

Since the computer running the visual processing algorithm now has to fly on board the vehicle, it becomes specially important to make a good choice when selecting hardware. To be able to take into the air, the computer has to be light enough that its weight can be lifted by the propellers while maintaining an 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 feed to the control loop.

The Raspberry Pi 4 model chosen for this project and shown in Figure 3.8 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 important advantages of using such a platform is the easy access to a great amount of manuals, guides and other support available online. In addition, the Raspberry's officially supported operating system, called Raspberry Pi OS, is a Debian-based version of Unix optimized for its ARM microcontroller, simplifying the process of moving from an Ubuntu test environment into real flight experiments. Since this computer is designed to be easy to integrate with hardware projects it includes a 40-pin GPIO header (marked in red in Figure 3.8) that can be programmed for connecting to any number of external devices.

Write: power management, secondary battery

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 (marked as "5v" on figure 3.8. In the case of the particular vehicle build used in this project, the power management



**Figure 3.8:** The Raspberry Pi 4 microcomputer, with its 40-pin GPIO header marked in red, and its pinout.

Source: Adapted from Hawkins [5]

board used (the Holybro PM07<sup>8</sup>) also provides 5V to the flight controller, as well as powering the ESCs to the motors, and counts with two power outputs so one of them can be connected to the flight controller's POWER1 port and the other to the Raspberry Pi's pins through a custom-made connector.

In regards to the camera that will fly onboard the vehicle, there are many possibilities to chose from. The most important characteristics that should be looked for are a small weight and simple plug-and-play interaction with the onboard computer. The camera used for the tests carried out and detailed in section 4 is a Logitech C920 1080p webcam<sup>9</sup>. Since the frame of the Holybro X500 is not prepared by default to include an onboard camera, a custom support 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 custom support can be seen in figure 3.9 and the print-ready file can be found in the project repository in GitHub<sup>10</sup>.

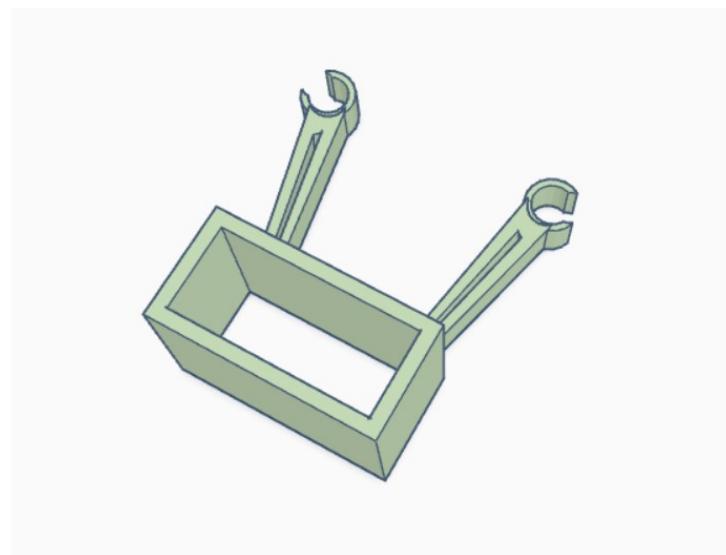
al check:  
works

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 exchanged. As it is desirable to be able to maintain a wireless link to the vehicle even while it is being controlled by the onboard computer, the telemetry radio is kept connected to the

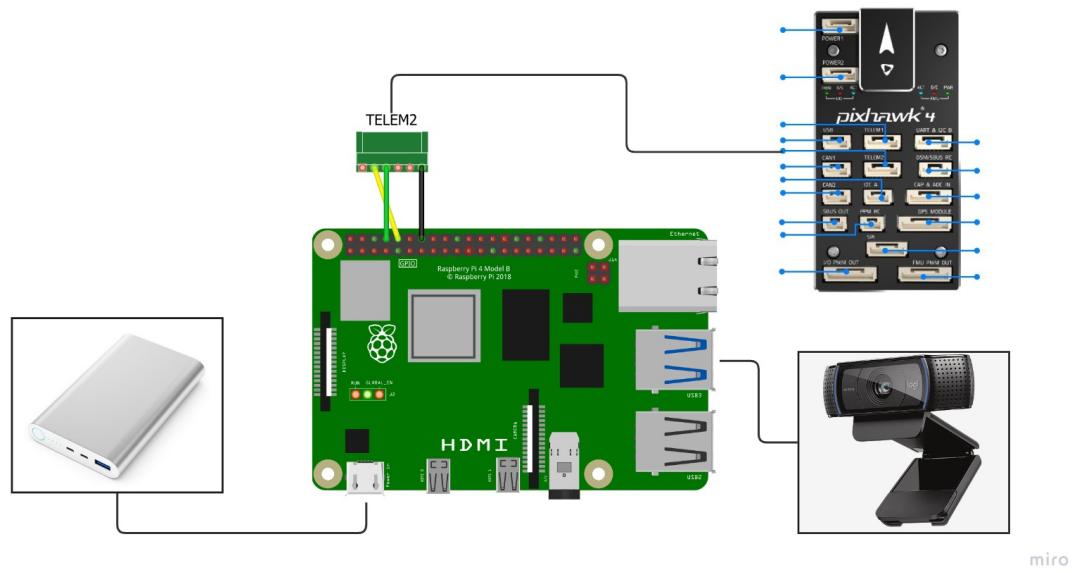
<sup>8</sup><http://www.holybro.com/product/pixhawk-4-power-module-pm07/>

<sup>9</sup><https://www.logitech.com/es-es/products/webcams/c920-pro-hd-webcam.960-001055.html>

<sup>10</sup><https://github.com/l-gonz/tfg-giaa-dronecontrol/blob/main/data/camera-holder.stl>



**Figure 3.9:** 3D model for the camera support designed for the Holybro X500 frame.



**Figure 3.10:** A diagram of the wired connections from the Raspberry Pi 4 to the secondary battery and the flight controller (TELEM2).

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.

TELEM1 port of the flight controller and the companion computer is wired to the TELEM2 port. 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.

Write: setup parameters to enable mavlink on telem2, configure baudrate

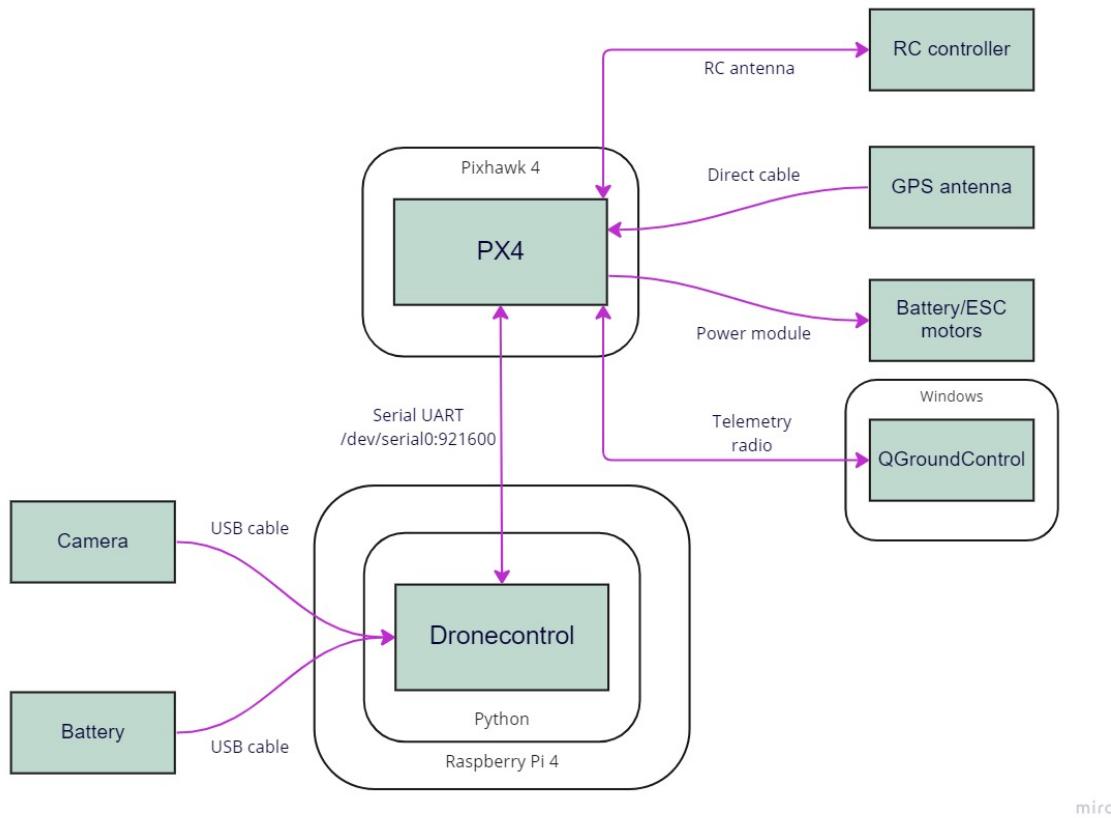
The other end of the connector has three female Dupont wires that go into the TX/RX UART pins of the Raspberry Pi, according to mapping table 3.1 between the 6 pins in the telemetry connection of the Pixhawk board and the corresponding GPIO pins in the Raspberry Pi header [6] [17]. A diagram of all the connections to the companion computer can be seen in figure 3.10.

In comparison with 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 16 times faster through the link.

Final polish: link to performance section

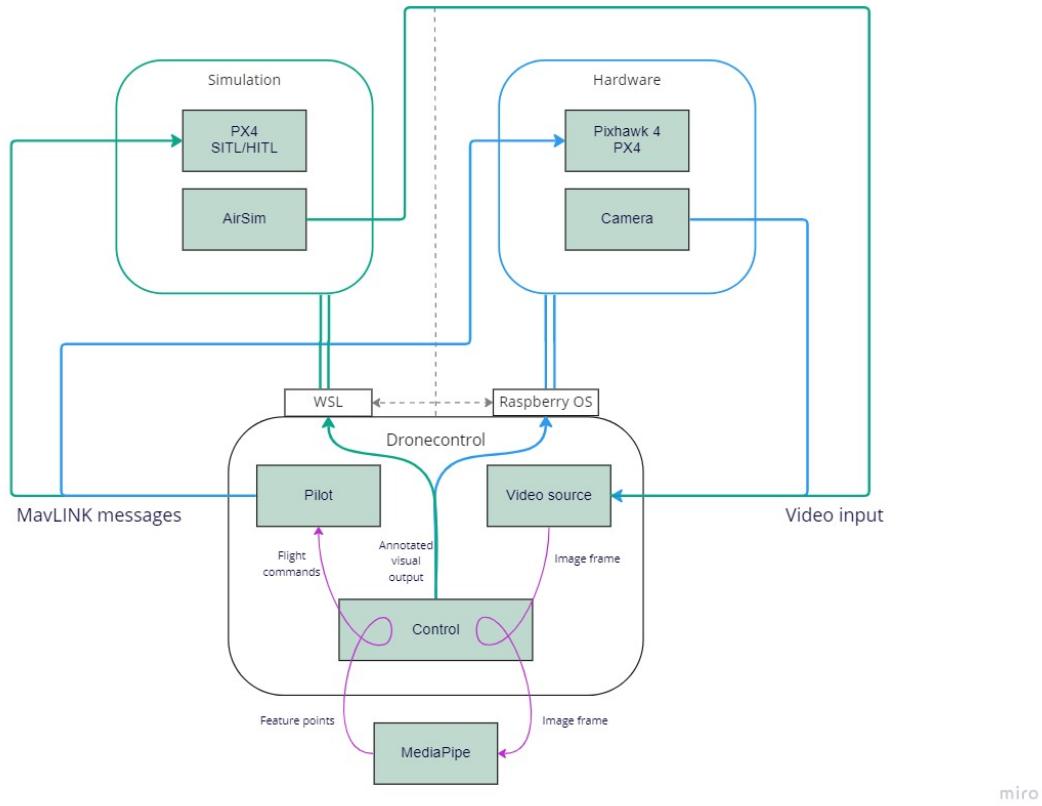
The offboard configuration allowed the supervision of the output from the program while the vehicle was flying, since the computer stayed stationary on the ground, however in this configuration it is not possible to have a screen connected directly to the companion computer. To fix this situation and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

Figure 3.11 shows a summary of all the connections mentioned between the autopilot the companion computer, and their respective peripherals.



**Figure 3.11:** Onboard configuration connections

Write: elaborate on the summary



**Figure 3.12:** Structure of the program

### 3.3 Software architecture

Write: Describe the actual new diagram, blue vs green arrows, data flow, ...

Figure 3.12 shows the main modules the designed the software is composed of and how it interacts with the external libraries. The application consists of three basic 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 3.12 shows how the program modules receive input and send instructions to the hardware systems whether these are simulated or not. 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. Appendix A.6 contains the user manual with all the application's options.

### 3.3.1 Pilot module

The purpose of the pilot module 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 standard library `asyncio` to be able 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, distributed task queues, etc; and provides a set of high-level APIs to run Python coroutines concurrently and have full 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) or virtual (UDP) address 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 basic operations that can be executed in the flight controller are implemented in the pilot module 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 runs in a loop executing them in the order 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 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 parallel execution.

### 3.3.2 Video source module

The objective of the video source module 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 running in different

```

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}} {{self.ip}}:{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

environments. There are three classes of video sources implemented: file, simulator and camera, which inherit from the same `VideoSource` base class as shown in Figure 3.13.

The `FileSource` class is able to open a video file stored in the companion computer and provides images taken frame by frame from it until the video ends. This allows the program to replay the image detection algorithms on videos previously captured by the camera tool exposed 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 frames captured in real time. Both the file and camera sources employ OpenCV's video capture utilities to take care of the file handling and the camera driver management capabilities respectively.

The simulator source uses AirSim's Python library to communicate with the simulator and retrieve images from a simulated camera attached to the 3D model of the vehicle in Unreal Engine. It connects automatically through localhost, but it can also be provided with an IP to establish connection to the simulator running on a different computer on the local network, for example when the program runs inside WSL and the simulator runs on its host computer.

```

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.
    """

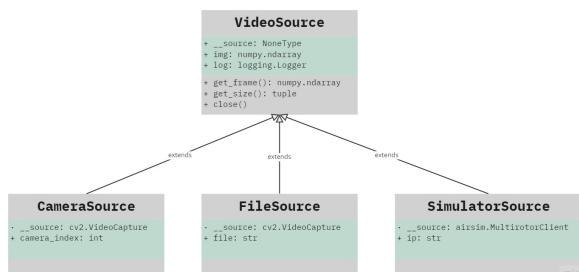
    try:
        while True:
            if len(self.actions) > 0:
                action = self.actions.pop(0)
                self.log.info("Execute action: %s", action.func.__name__)
                try:
                    await asyncio.wait_for(action.func(self, **action.args), timeout=10)
                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.

### 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 a proof-of-concept control solution, described in section 3.4, that runs in offboard mode (see section 3.2.2) and translates some predefined hand gestures into simple commands for the aerial vehicle, of which the main purpose is to be able to test the interaction between all the components of the system in a more easily controlled environment, since it uses the simpler configuration of situating the computer with the controller outside of the vehicle. The second control



**Figure 3.13:** Diagram of inheritance on the video source classes available to retrieve image data.

system consists of a follow solution more applicable to real-life scenarios where the control and the camera is onboard the vehicle and the presence of a person is detected in the images obtained from the perspective of the drone in order to be able to give the flight controller velocity commands to follow said person and maintain it centered 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 that extracts the required features from the image in the form of 2D coordinates. Afterwards a series of 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 ML solutions for live and streaming media, specifically its hand and pose detection solutions.

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 <sup>11</sup> (not to be confused with the offboard configuration described in section 3.2.2). Offboard mode is primarily used for controlling vehicle movement and attitude, and supports only a very limited set of MAVLink messages. This mode requires position or pose/attitude information to be available to the flight controller, e.g. GPS. 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. A stream of setpoint commands must be received by the vehicle at a rate higher than 2Hz prior to engaging the mode and in order to remain in it. If the message rate falls below 2Hz or the connection is lost the vehicle will stop and, after a timeout, the vehicle will attempt to land or perform some other failsafe action according to the parameters configured. In order to hold position while in this mode, the vehicle must receive a stream of setpoints for the current position.

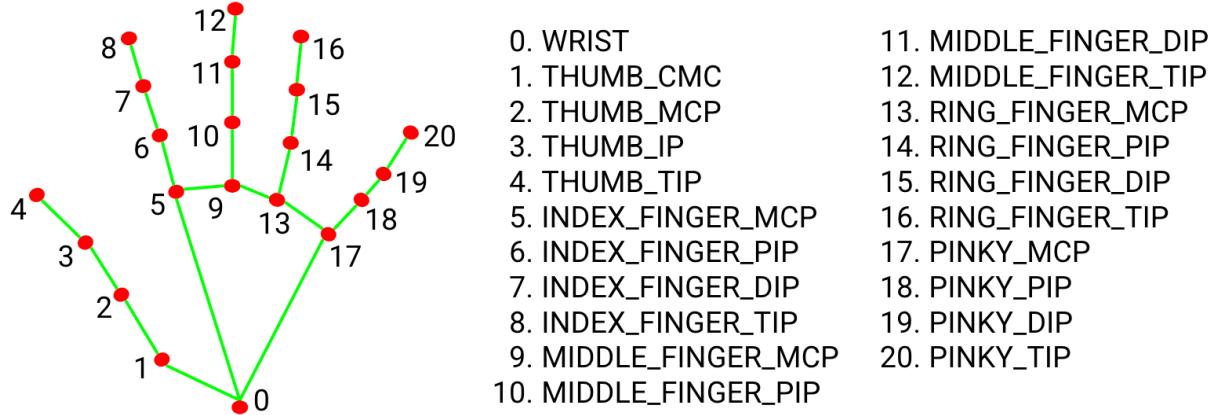
Sections 3.4 and 3.5 offer a more complete explanation of the control module used by the two different solutions developed.

### **Camera-testing tool**

Several additional utilities have been added to the dronecontrol program to facilitate the development and testing process of the two main control solutions. The first tool is accessed through the command `dronecontrol tools test-camera` and can be used for testing the connection between the computer and the camera, as well as the performance of the Mediapipe hand and pose machine learning solution on real time images. It is as well possible to take images and record video from the live camera feed for later analysis and it allows connecting to a physical or simulated flight controller to send basic commands through

---

<sup>11</sup>[https://docs.px4.io/main/en/flight\\_modes/offboard.html#offboard-mode](https://docs.px4.io/main/en/flight_modes/offboard.html#offboard-mode)



**Figure 3.14:** Landmarks extracted from detected hands by the MediaPipe hand solution.

Source: Adapted from *Hands - mediapipe* [8].

keyboard input like takeoff and landing.

Write: Add control functionality, describe connection options, link to cli in appendix

Polish: Link some code from test-camera tool

## 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 real flight, and that all the systems are capable of establishing the required connections with each other. For that reason it is designed to be able to be run in real flight with the minimal setup of a built drone with its default components and any computer with a camera.

This control module 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 [18]. If a hand is detected in the images, 2D coordinates are extracted according to the map shown in figure 3.14. These landmarks are then converted into different discrete gestures like open palm, closed fist or a specific finger pointing 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 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 3.15. By comparing



**Figure 3.15:** Vector calculations

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 it is pointing towards. 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.

#### Draw: hand vectors

The full list of gestures detected by the program by calculating these angles is as follows:

- No hand: happens when no landmarks are able to be extracted from the image. As a safety feature, in this case the vehicle stops whichever previous commands it had in its queue and goes into hold flight mode, where it just hovers in the air maintaining its position.
- Open hand: it is detected when all five fingers are extended, as if gesturing stop, and makes the drone land at its current position.
- Fist: it is detected when all five fingers are folded and makes the drone arm and takeoff. If the drone is already in the air nothing happens.
- Index finger pointing up: it is detected when only the index finger is extended and it is pointing roughly towards the top of the image ( $\pm 30$  degrees) and makes the drone go into offboard mode, where it is possible to receive direct velocity commands.
- Index finger pointing to the right: same as above but pointing to the right of the image and 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 the drone rolls towards its left side.

- Thumb pointing to the right: it is detected when index finger is extended up (to maintain the drone in offboard control) and the thumb is extended 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 backward when the thumb point to the left of the screen.

Write: add loop code and link to first paragraph

Draw: Include diagram of angles for each pointy gesture ?????

Images: simulators program output image with drawn lines for gestures

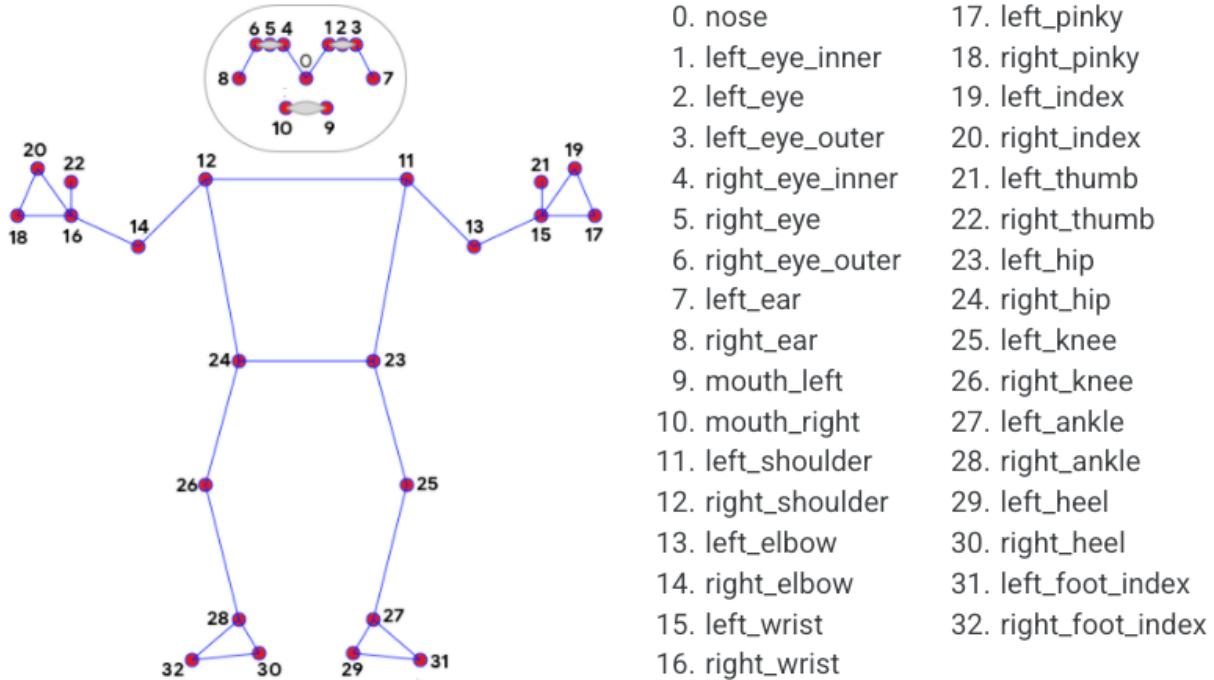
Video: hand control in simulator

## 3.5 Final solution: human following

The intention behind the development of 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 to be able to 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.

For safety, the follow mechanism only engages when the flight mode on the vehicle is changed to offboard mode e.g. by activating a configured switch in the RC controller, and stops automatically if the connection to the computer is lost or any of the available failsafes are triggered, like low battery, loss of RC or GPS signal or vehicle attitude exceeding a pre-defined 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 is engaged, the system continuously retrieves images from the offboard camera that are



**Figure 3.16:** Landmarks extracted from detected human figures by the MediaPipe Pose solution

Source: Adapted from *Pose - mediapipe* [9]

fed into the MediaPipe Pose [3] computer vision library to extract pose landmarks from it in the form of 2D coordinates. Figure 3.16 shows the available extracted features and their correspondence to the human body. These coordinates are used to draw a bounding rectangle around the detected person, as well as for validating that the received landmarks match the general pose of a person standing up. To prevent unwanted movements, the vehicle will always stop and hover every time it becomes impossible to detect a person in the image received or its features do not match the expected geometry.

Images: valid and invalid person detection from simulator camera output

After a valid bounding box has been defined around the target person, its position in respect to the field of view of the camera is sent to a control mechanism composed of two independent PID controllers. The theory behind these controllers is explained in section ??).

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.

Draw: coordinate system figure

It takes as input the x coordinate of the center point of the bounding box and its target

is the middle point of the screen. This controller therefore will output velocity commands aimed to maintain the bounding box centered 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 further away from the drone. It takes as input the height of the calculated bounding box as a percentage of the total height of the field of view 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 that is covered by a person at a given distance depends on the camera used and needs to be obtained empirically for each separate video source.

Add some controller and follow code to show and explain

Images: simulator and output, link video of run

## PID tools

Two additional utilities have been developed for tuning and measuring the performance of the PID controllers.

Write: PID tools, how they work and code

PID library, Formula and coefficients

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

The process of tuning the PID controllers is described in section 4.2

### 3.5.1 Safety mode

Write: safety mechanisms, speed limits on pids, loss of signal, loss of vision, ...



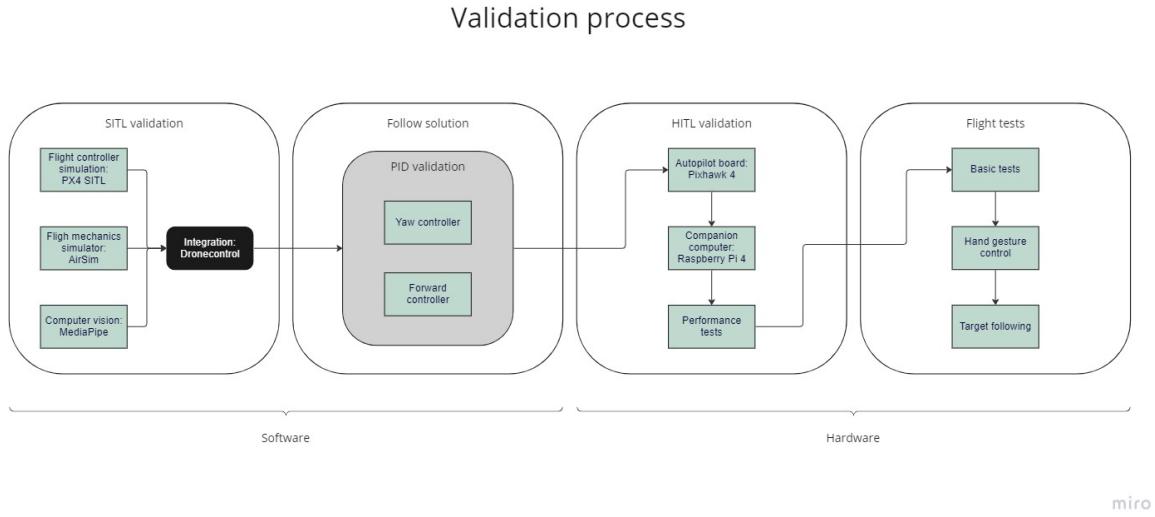
# Chapter 4

## Experiments and validation

This chapter explains the complete process of validating each separate piece necessary for the correct operation of the full system, from testing the first simulation steps to carrying out thorough flight tests on the final guidance solution. Figure 4.1 shows in order each of the steps followed during this validation process. In the first section, the application is tested in a purely simulated environment, first individually on each part and finally integrating the flight mechanics simulation with providing images to the computer vision detection software 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, and 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 real 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. By using this mode, it is possible to test and validate the correct operation of each individual part in the program's architecture. To begin with, it is necessary to validate that it is possible to use the connection between the simulated flight controller and the dronecontrol program to send commands to the simulated vehicle, as well as being able to capture images from a connected camera and use them to test the detection algorithm. The full list of validation



**Figure 4.1:** Outline for the validation process

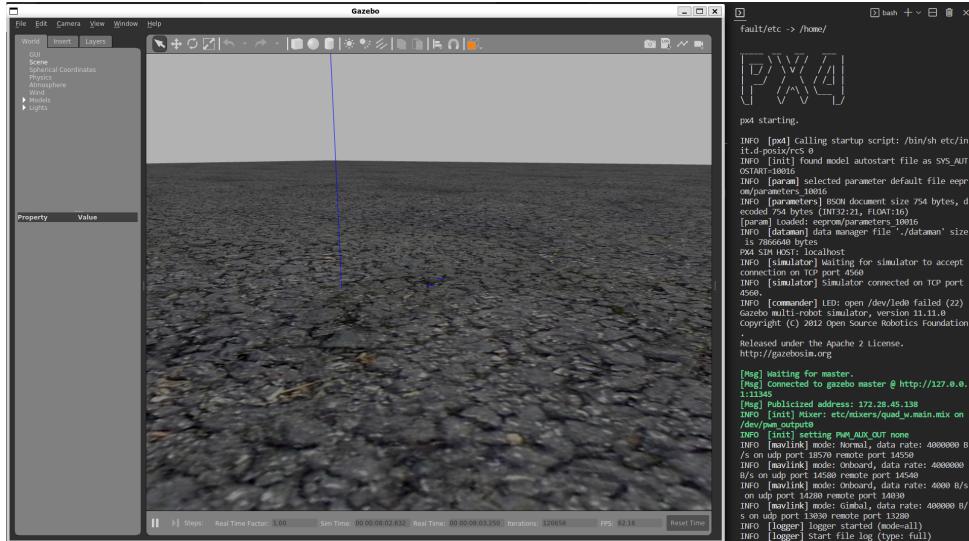
steps is therefore:

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<sup>1</sup> included with the base PX4 installation will be used as the 3D environment. This simulator works inside Linux in 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 commands and changing the configuration parameters of the simulated flight controller. The simplest command

<sup>1</sup><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

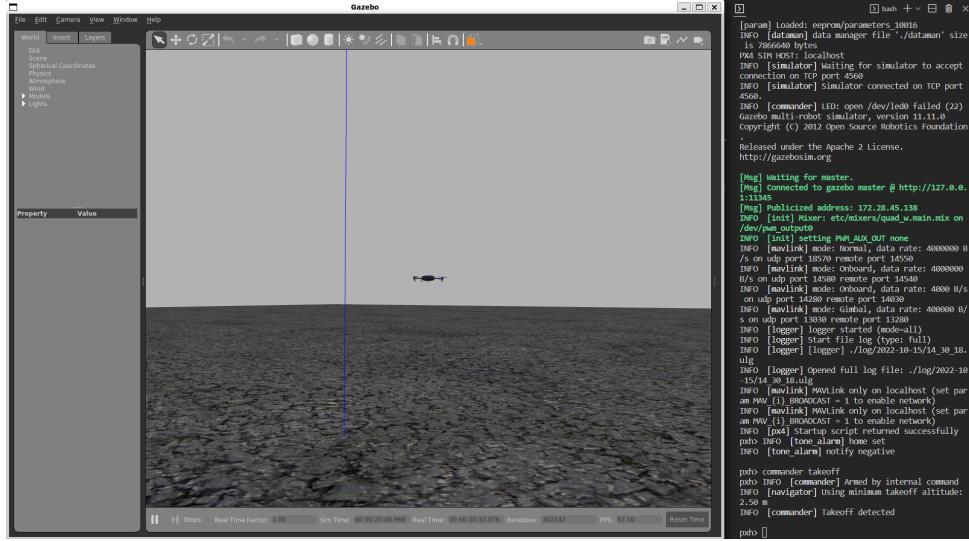
to test is takeoff, which is done by sending `commander takeoff` through the console. Figure 4.3 shows the state of the simulator after the takeoff command, with the vehicle model having climbed to the default height of 2.5 meters above ground. To land the vehicle again, the command is `commander land`.

The next step is to connect the dronecontrol application. The camera testing tool described in section 3.3.3 has been developed 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" can be used to 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.

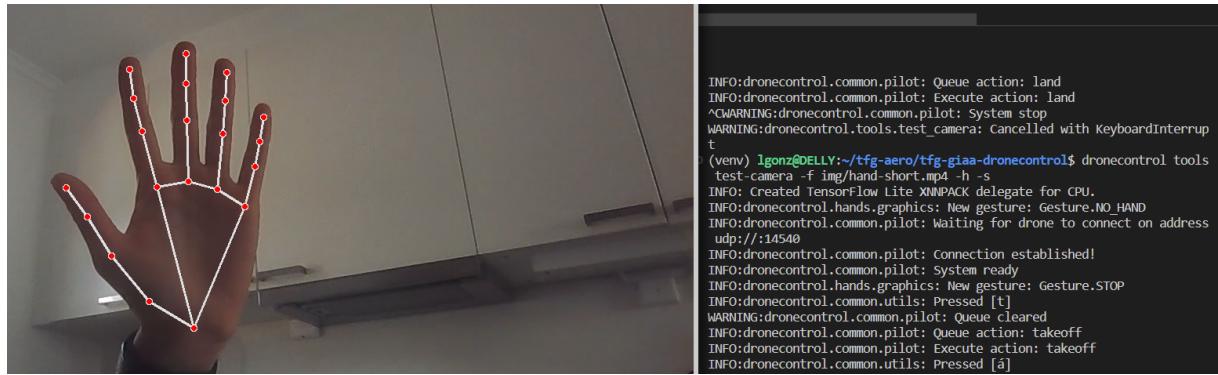
Get a better image for this

The full execution of a test of the hand-gesture based control solution is shown in video in this link and excerpt of it can be seen in figure 4.5.

Save video on GitHub and put link here, fix text to make sense



**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 be able 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 is able to 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 is able to control the velocity of the vehicle directly in PX4's offboard mode.

In the first place, the AirSim simulator needs to be installed in the Windows host. The full installation process is described in appendix A.3. There are specific configuration 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



**Figure 4.6**

a line defining the IP address of the network interface to use. This parameter, along with the full configuration file used for SITL testing can be found on appendix A.4. 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 one of 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.

Image of PX4 console + Unreal with Airsim scene

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 the command `dronecontrol tools test-camera --wsl --sim --pose-detection` and a 3D model of a person is situated in front of the drone in the simulated world. In the image, the computer vision utilities detect the main features of the human body and a



**Figure 4.7**

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 mid point 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.

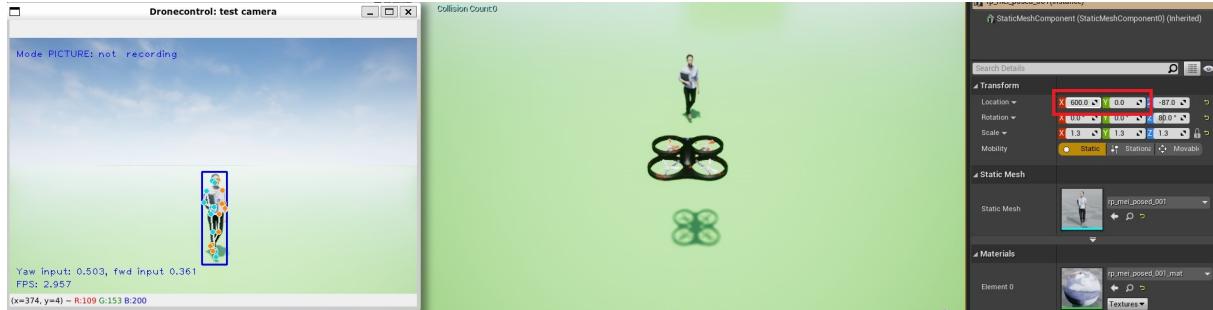
Image of dronecontrol console + Unreal with Airsim scene + image output running test-camera

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 video format in [this link](#) and a frame extracted from it can be seen in figure 4.8.

video



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



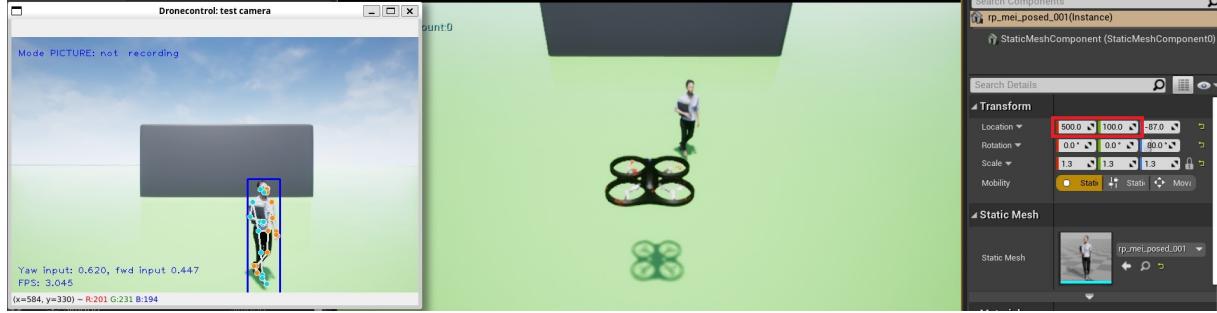
**Figure 4.9:** Reference position for the yaw and forward PID controllers. From left to right the panels show the dronecontrol application window, the AirSim simulator world view and the world location of the human model in the simulator. The distance between the vehicle and the person is 600 units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

## 4.2 PID controller validation

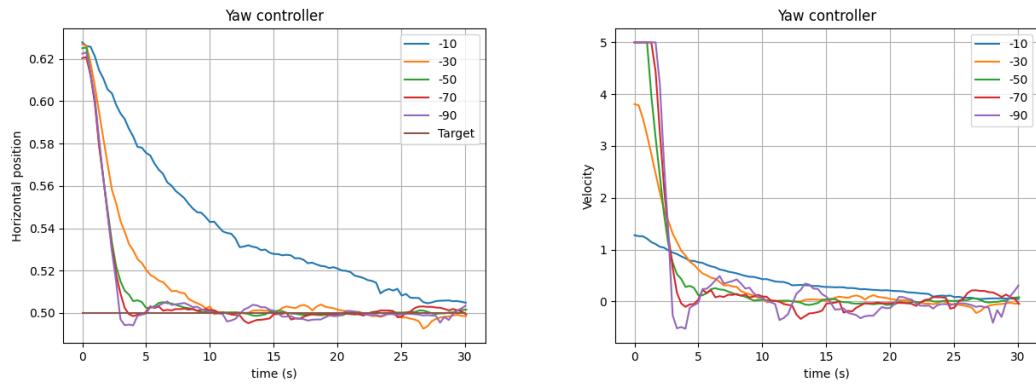
Videos for github

As has been 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 3.3.3 and developed for this purpose, and its performance 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 of the controllers will be tuned independently of the other by allowing the vehicle only one direction of movement at a time. Afterwards, both controllers are engaged at the same time to take measurements of their 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 centered in the field of view and with a height of 36% of the image height. Which 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.

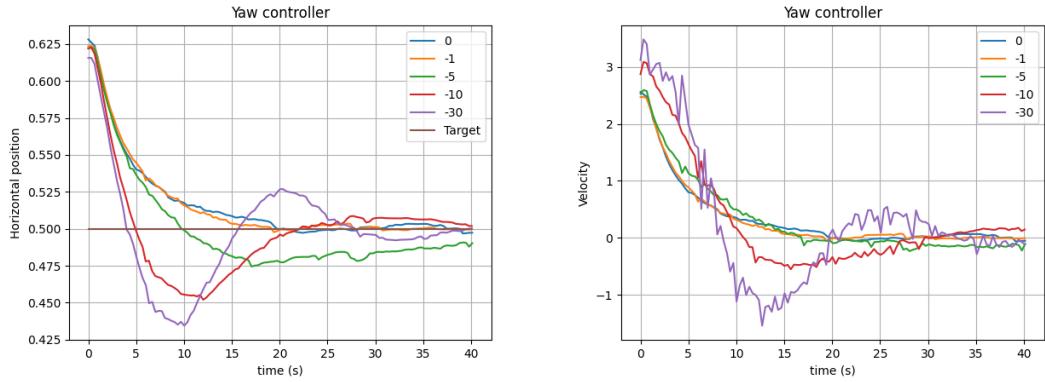


**Figure 4.11:** Variation of (a) input position and (b) output velocity for different values of  $K_P$  and  $K_I = 0, K_D = 0$  while the yaw controller is engaged.

### 4.2.1 Yaw controller

Discuss velocity graphs as well for every section, mention velocity limits

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 produces a rotation of the vehicle to that side. Since the forward controller will not be engaged, 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 will then need to output a positive yaw velocity to center the person in its field of view. Since this offset position to the right produces a negative difference but requires a positive velocity to counteract, 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.



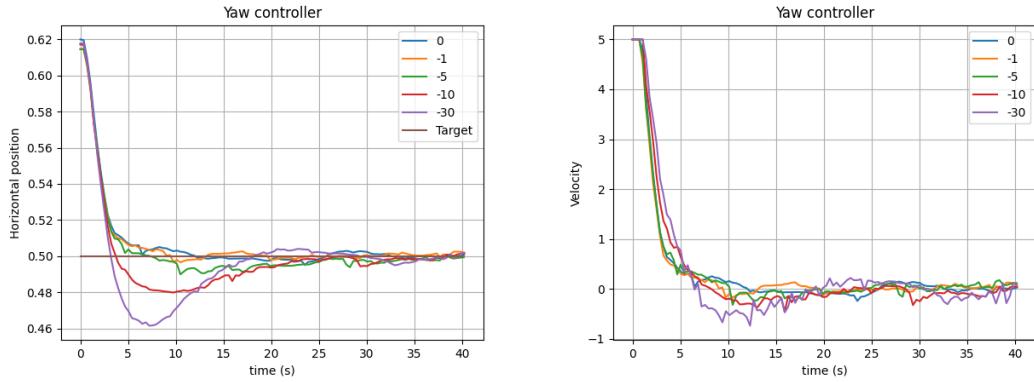
**Figure 4.12:** Variation of (a) input position and (b) output velocity for different values of  $K_I$  and  $K_P = -20, K_D = 0$  while the yaw controller is engaged.

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 and the right graph, the yaw velocity that the controller outputs to the pilot module to reach the target. It can be seen that 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. In 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 without stopping. From the graphs, the best of the values tested is  $K_P = 50$ , where the distance to the target point decreases rapidly but the velocity does not start to increase and decrease widely around 0.

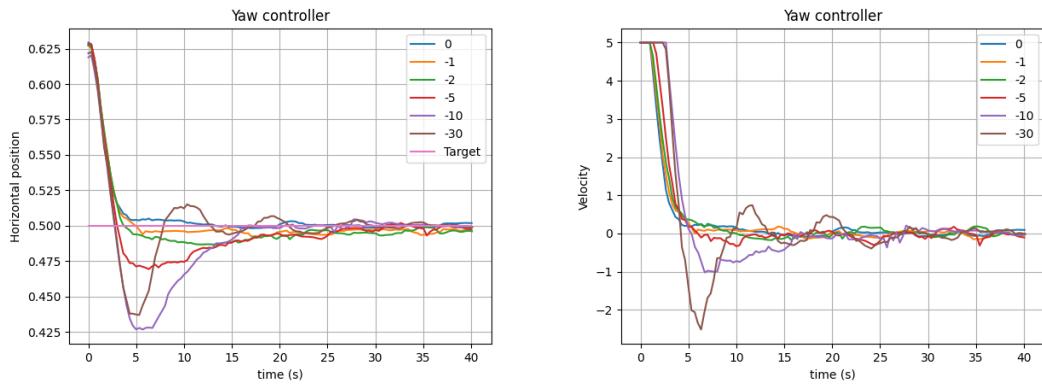
The second step is to find the correct value for  $K_I$ . To do that several values of  $K_I$  will be tested for a low  $K_P$  of -20 and  $K_D = 0$ , so that the effect of the integral part is easier to appreciate. Figure 4.12 shows the evolution of the input and output at the controller for a sample 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 and, for a very large  $K_I$  from around -10 and bigger, the velocity of the vehicle becomes locally unstable with many small 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 than with the proportional part exclusively ( $K_P = -50, K_I = 0, K_D = 0$ ).

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  $K_P = -50$  and both

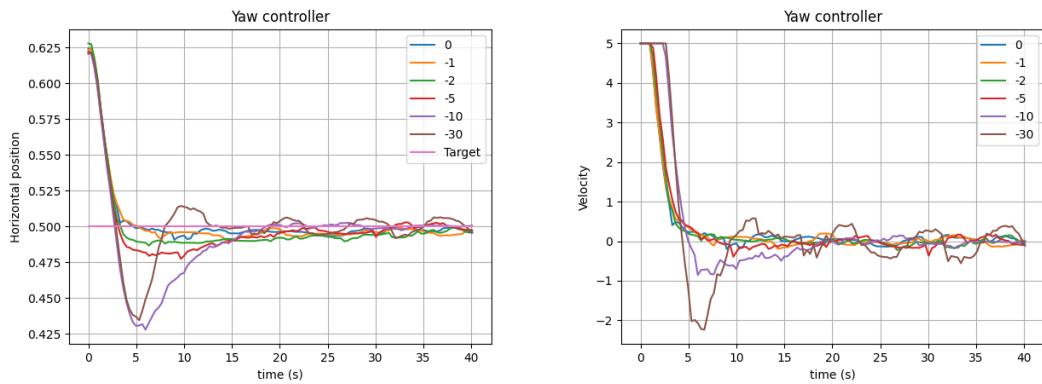


**Figure 4.13:** Variation of (a) input position and (b) output velocity for different values of  $K_I$  and  $K_P = -50, K_D = 0$  while the yaw controller is engaged.

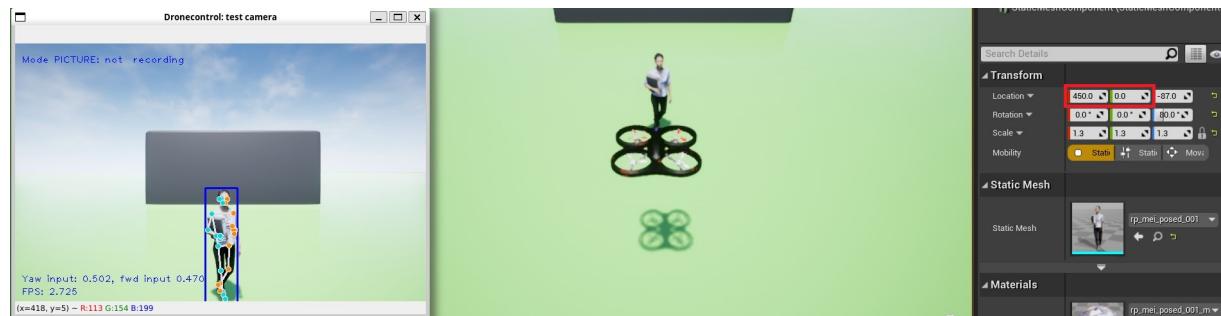


**Figure 4.14:** Variation of (a) input position and (b) output velocity for different values of  $K_D$  and  $K_P = -50, K_I = 0$  while the yaw controller is engaged.

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 shows the evolution of the position detected by the computer vision system and the velocity that 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 best remains on the target position continues to be the one for  $K_D = 0$ . 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 full PID controller.



**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.

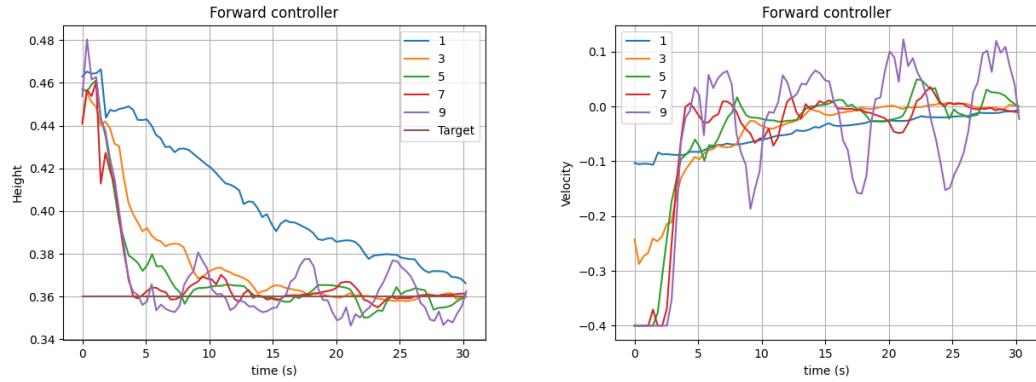


**Figure 4.16:** Starting position of the simulator for tuning the forward controller. The human model is situated 450 units forward and centered from the vehicle position.

### 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 centered 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. 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 and reduces 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).

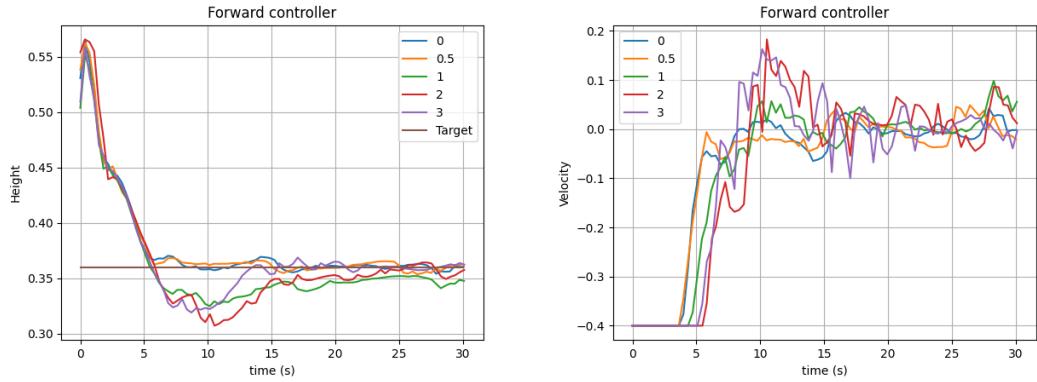
Controller should end up with (4, 0.1, 0)



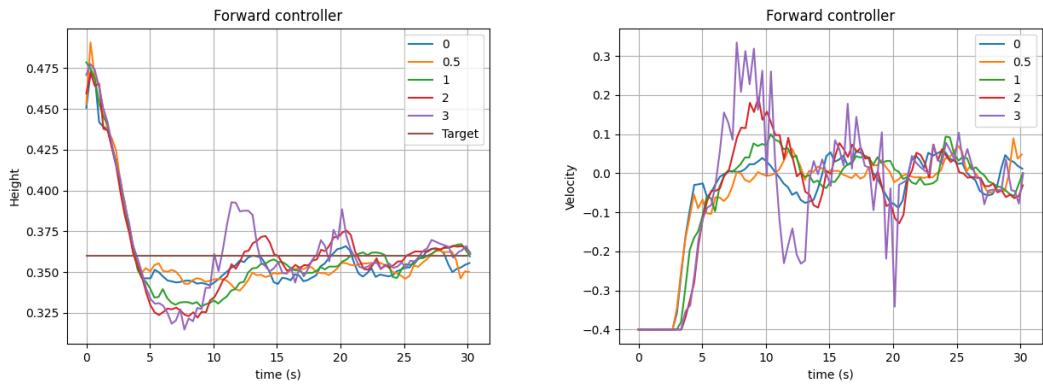
**Figure 4.17:** Variation of (a) input height and (b) output velocity for different values of  $K_P$  and  $K_I = 0$ ,  $K_D = 0$  while the forward controller is engaged.

In general, the forward velocity needs 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 destabilize 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. In the first place, 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 this coefficients are shown in figure 4.17. In comparison with the trajectory described by the yaw controller, the forward controller is in general more unstable since fast movements forward and backward 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 in respect to the human figure. It also creates an effect for the bigger  $K_P$  values tested where small leftover oscillations from the previous test can affect the next one and cause the controller to pull the vehicle in the opposite direction for a second before stabilizing the image and detecting the correct direction of movement it should follow. This is the main reason why the output forward velocity needs to be kept small in the controller. However, even for a value up to  $K_P = 7$  the trajectory described descends rapidly without ending in big oscillations around the target height, so it is a good 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 and 4.19. For the integral part, increasing the coefficient causes the controller to overshoot the target in the first instance, but then causes a stable approximation to the target line. Since for this application overshooting is not desirable, as it can cause safety issues, the chosen value of  $K_I$  will be 0.5, which slightly improves on the performance without overshooting. For the derivative part, high values of  $K_D$  make the system oscillate more, and for very high values from  $K_D = 3$  the changes in forward velocity are fast enough that the camera loses track of the figure before regaining it again, which produces the erratic behaviour that can be seen in the purple line on figure 4.19b. In this case, the value of  $K_D = 0.5$  seems to most closely approximate to the target line with minimal oscillations. The final values for the



**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.



**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.

coefficients for the forward controller will then be  $K_P = 7, K_I = 0.5$  and  $K_D = 0.5$ .

### 4.2.3 PID tuning validation

As a final validation for the tuning obtained previously, in this section the test-controller tool described in section 3.3.3 will be used check the step response of the controllers to different starting distances for the same coefficients. Additionally, for this test both of the controllers will be engaged simultaneously to verify that work well together. In the first instance, the positions will vary in 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 normalized horizontal distance and normalized figure height

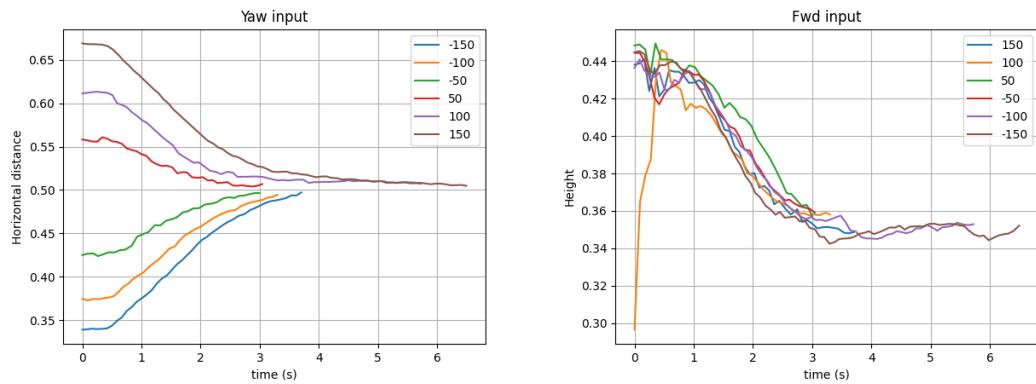


Figure 4.20

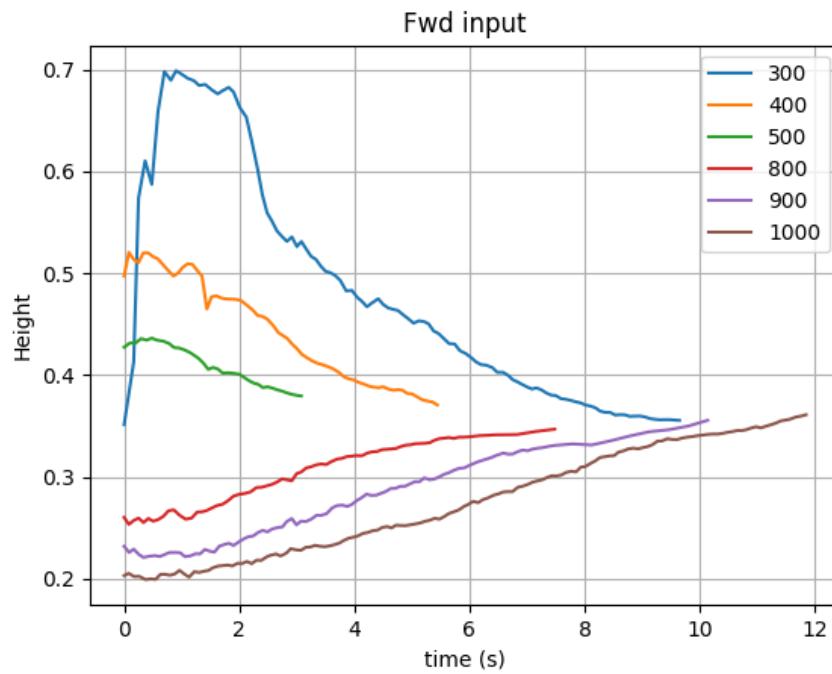
detected by the person recognition algorithm during the time the vehicle takes to reach the target distance from the human figure for each of the positions tested can be seen in figure 4.20. It is considered that the vehicle has reached its target when the error is less than 2% and the output speed at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

video of  
-controller

Since each of the starting positions differ both in distance and orientation to the target, both of the controllers are engaged to reach it. 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 centered in the image received from the camera); and 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 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, 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 low for a bit until it reaches the starting point of the other tests, this occurs because the detection algorithm took longer than usual to identify the person. However, after less than half a second, the detection was stabilized without affecting the time it took to reach the target or its final position. This shows that the controllers can recover from errors in the detection mechanism without affecting the movement of the vehicle.

Additionally, the test-controller tool is 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 in the center of its field of view ( $y$  po-

**Figure 4.21**

sition will remain 0 for the whole process). This means the yaw controller will not need to output any velocity. 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 that there is a very big difference in how the controller reacts to closer positions and to further positions.

TODO, and rerun

- Closer vs further positions - Error in detection for  $x=300$

A video of the full follow solution running with the appropriate tunings for the controllers can be seen in .

[Link video  
follow run](#)



**Figure 4.22:** Pixhawk 4 board connected to the computer running the AirSim simulator and the dronecontrol application

### 4.3 PX4 HITL simulation and validation

The purpose 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.

To activate this new hardware-in-the-loop mode, QGroundControl contains a specific quadcopter HITL airframe configuration that sets up the board with all the required parameters. QGroundControl automatically detects the Pixhawk 4 board when it is connected to the computer through its Micro-USB port. It is as well 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 TELE1 port with a wireless link to its counterpart 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, through which it can automatically connect to the board when it is started. The dronecontrol program will connect through the telemetry radio by specifying the serial port and its baudrate. Figure 4.22 shows all the connections mentioned.



**Figure 4.23:** Some figures that show these tests

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.

With the results matching exactly those obtained in section 4.1.1, and 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<sup>2</sup>. 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. As can be seen in figure 4.23, ....

### 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 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

---

<sup>2</sup><https://docs.px4.io/main/en/config/radio.html>

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.2.1. 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, it is possible to access the Pi's desktop from the ground station computer even during flight. One option to achieve this is XRD<sup>3</sup>, 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. The most common way of providing current to the Pi when mobility is not required is through its USB-C port directly to a wall socket. This provides the recommended 3 amps that it needs to function at maximum capacity. The power module included with the Holybros' X500 development kit has two ports that provide 5 volts with an output current of 3 amps. One of them will therefore be used to power the flight controller and the other to power the Raspberry Pi, as described in section 3.2.3. Figure 4.24 shows the power supply through the battery to the two computers.

Edit for new battery

Close up image and description of custom connector

A similar connector is used to provide 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 TELE2 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 TELE1 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 is configured to be used as a terminal by default. This can be disabled through the `raspi-config` command-line utility (Advanced config -> Serial -> Disable). 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.

Fix table

<sup>3</sup><http://xrdp.org/>



**Figure 4.24:** Picture of Rpi + power module + battery + pixhawk



**Figure 4.25:** a) Picture of Rpi raspi-config || b) close-up of pixhawk to pi cable connection

Parameter name	Value
MAV_2_CONFIG	TELEM2
SER_TEL2_BAUD	921600

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



**Figure 4.26:** a) RPi desktop with pose output || b) AirSim on Windows

The test camera utility will be used to validate this configuration. At this point, the only physical connection to the ground station running AirSim and/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 4.26. On the left 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 right side, the AirSim simulator shows the movements of the vehicle as it reacts to the input from the flight controller and the companion computer.

The main question to answer now 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 a performance good enough to react to real-time movement.

Performance analysis with/out peripherals, detection only, detection and control together

## 4.4 Quadcopter flight tests

Once the performance and safety of the control algorithms has 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 in order 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 making sure that the vehicle can fly with all the payload through remote control, both of the developed solutions will be tested: first the hand-control solutions to verify that the autopilot can receive flight commands from an offboard computer and second the follow solution to verify 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 QGRound-Control.
4. Fly with custom software from offboard computer with test tool `test-camera` 3.3.3.
5. Fly with test tool `test-camera` from onboard computer.
6. Fly with custom hand-gesture control solution from offboard computer.
7. Fly with custom follow solution from onboard computer.



**Figure 4.27:** Development kit for the Holybro X500.

Source: Adapted from *PX4 User Guide* [14].

#### 4.4.1 Build process

As has been mentioned before, the vehicle used in this project is the Holybro X500, designed specifically to work with PX4. The detailed instructions to build the vehicle from its Development Kit can be found in the PX4 documentation <sup>4</sup>. Figure 4.27 shows all the parts that make up the complete vehicle.

After all the standard parts are put together, the custom additions can be attached as well using the remaining space in the frame. The Raspberry Pi companion computer will sit 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 an acceptable performance. This battery will be located underneath the autopilot, centered in the frame of the vehicle so its weight destabilizes it as little as possible, as can be seen in figure 4.28.

The camera also needs to be attached to the frame of the vehicle in a secure way, with the custom support described in section 3.2.3. The holder attaches to the slide bars underneath the main frame of the vehicle so that the camera and its substantial weight is situated as

---

<sup>4</sup>[https://docs.px4.io/main/en/frames\\_multicopter/holybro\\_x500\\_pixhawk4.html](https://docs.px4.io/main/en/frames_multicopter/holybro_x500_pixhawk4.html)



**Figure 4.28:** Full build annotated with arrows

close to the center 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 centered position to make space for the camera and compensate its weight in the front. Figure 4.29 shows how the underside of the vehicle 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. The QGroundControl 2.2.1 ground station application contains a configuration screen with all the calibration tools needed for the vehicle setup, as 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 to the computer running QGroundControl. Before any flight can be carried out, 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 TELE2, as described in section 3.2.2.



Figure 4.29

#### 4.4.2 Basic tests

##### Baseline flight with factory software

Once the vehicle is fully configured, the RC controller and QGroundControl can be used to test assisted take off 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 ground of the vehicle in the forward/backward and left/right directions relative to the heading the vehicle is facing, and throttle controls speed of ascent and descent. With the sticks centered 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 is working as expected.

Through QGroundControl it is possible to map the different switches in the RC controller to different 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, and one of the switches with three positions will be mapped to the landing/takeoff/position flight modes respectively, so the main autopilot modes can be tested by moving the switch 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.



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



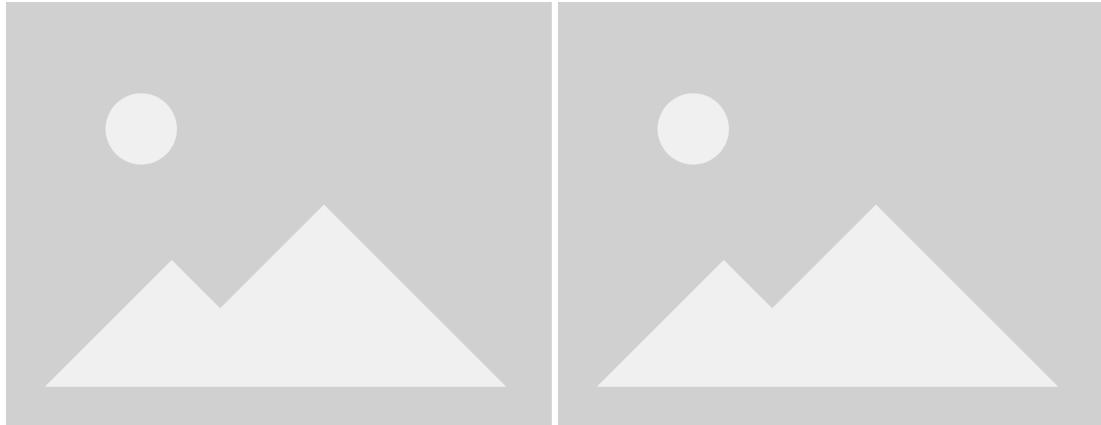
**Figure 4.31:** Picture from flight tests

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 in a computer that has been 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 in 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. Figure 4.31 shows an image from a flight carried out following this steps. The full video of the test can be seen in [link](#).

ake video  
link

### Offboard computer flight with test tool

The second test flight will aim to make certain that the custom software can send takeoff and landing commands through a wireless MAVlink channel from the offboard computer (by using the telemetry radio through the developed test tool). For this flight, the QGroundControl application cannot be connected to the vehicle since the telemetry radio channel will be blocked by the custom tool. The RC controller will be 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 before while sending a takeoff command, the two-way switch of the controller will be mapped for all the tests going forward to the command to kill the power



**Figure 4.32:** a) Terminal output from the test-camera tool running on an offboard computer and b) Output from the camera of the offboard computer towards the vehicle in flight

to the engines. This command can be useful in very 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 a successful connection to the vehicle, the T and L keys in the computer keyboard can be used for takeoff and landing, respectively, and the O key can be used to set the autopilot in offboard flight mode to make it able 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 4.32 displays the output on the terminal window of the computer, where the connection process and the sent velocity commands are shown, and the output on the camera from the offboard computer.

### 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 making sure 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,



**Figure 4.33:** Pose detection algorithm running on images taken during flight

but 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 also possible to activate pose detection on the test images received from this onboard camera. To start the flight test, the main battery and the secondary battery need to be attached, respectively, to the power module and to the Raspberry Pi. After the onboard computer has started up, the easiest way to take control of 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 TELE2 port of the Pixhawk board. The "-p" option is used to enable pose detection in the output images. The terminal and camera output can be seen in figure 4.33 and a video recording the remote desktop during the flight can be seen in [link](#).

ck video of  
en during  
oard test-  
era flight

#### 4.4.3 Hand gesture control

During basic flight tests, all the connections and individual parts of the software were validated in real flight. Now it is time to integrate the piloting system with the image recognition results to test the vision-based control solutions developed. The first solution to be used in flight will be the hand-gesture guidance system, as it is made to run on an off-

board 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 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 webcam of the computer will appear on the screen with an outline over any detected hand. To start controlling the vehicle, an open palm should be shown to the camera. Then, a closed fist will make the drone take off and pointing up with the index finger will start 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 forwards and backwards, respectively. At any point during the test, an open hand will make the vehicle land at the current place and losing sight of the controlling hand will make the vehicle return to the starting position before landing as well.

Check direction of thumb movement

Make return home altitude less before flying

Video should show side-by-side computer screen with terminal and image output and video from vehicle flying, synced

#### 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 is capable of keeping track of and following a moving target during non-simulated flight. The setup will be identical to the third test in section 4.4.2, without the need for 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:

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

Figure ?? shows the process in the terminal 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:** Image taken during flight controlled by the hand-gesture solution. Vehicle is taking off



**Figure 4.35:** Image taken during flight controlled by the hand-gesture solution. Vehicle is moving to the right.



**Figure 4.36:** Image taken during flight controlled by the hand-gesture solution. Vehicle is moving forward

The maximum frames per second managed by the program running on the Pi is around 10. In practice, this means that the person being tracked by the drone has to move quite slowly in order for the camera to not lose sight of them before the autopilot can send the command to the vehicle to move to the previous detected position. At the end of the program run, the average loop time and average runtime for each of the tasks in the main loop is shown in the terminal. From the measures obtained for the test flight carried out, we can see that however, for a proof-of-concept scenario, this is acceptable performance.

Save performance for flight tests, add measure tables in each section

Same recording as hand sec



**Figure 4.37:** Terminal and image output of the dronecontrol follow solution running on the Raspberry Pi

# Chapter 5

## Conclusiones y trabajos futuros

### 5.1 Consecución de objetivos

Esta sección es la sección espejo de las dos primeras del capítulo de objetivos, donde se planteaba el objetivo general y se elaboraban los específicos.

Es aquí donde hay que debatir qué se ha conseguido y qué no. Cuando algo no se ha conseguido, se ha de justificar, en términos de qué problemas se han encontrado y qué medidas se han tomado para mitigar esos problemas.

Y si has llegado hasta aquí, siempre es bueno pasarle el corrector ortográfico, que las erratas quedan fatal en la memoria final. Para eso, en Linux tenemos aspell, que se ejecuta de la siguiente manera desde la línea de *shell*:

```
aspell --lang=es_ES -c memoria.tex
```

### 5.2 Aplicación de lo aprendido

Aquí viene lo que has aprendido durante el Grado/Máster y que has aplicado en el TFG/TFM. Una buena idea es poner las asignaturas más relacionadas y comentar en un párrafo los conocimientos y habilidades puestos en práctica.

1. a
2. b

### **5.3 Lecciones aprendidas**

Aquí viene lo que has aprendido en el Trabajo Fin de Grado/Máster.

1. Aquí viene uno.
2. Aquí viene otro.

### **5.4 Trabajos futuros**

Ningún proyecto ni software se termina, así que aquí vienen ideas y funcionalidades que estaría bien tener implementadas en el futuro.

Es un apartado que sirve para dar ideas de cara a futuros TFGs/TFMs.

# **Appendix A**

## **Installation process**

### **A.1 Installation of PX4 SITL**

### **A.2 Installation of Dronecontrol**

#### **A.2.1 Installation on a Raspberry Pi 4**

### **A.3 Installation of AirSim**

### **A.4 AirSim configuration file**

### **A.5 PX4 configuration**

### **A.6 Command-line interface of the Dronecontrol application**



# Referencias

- [1] 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](https://commons.wikimedia.org/wiki/File:Raspberry_Pi_4_Model_B_-_Side.jpg) (**urlseen** 13/01/2023).
- [2] R. Bartak **and** A. Vykovsky. ?Any object tracking and following by a flying drone? incited By 15: 2016, **pages** 35–41. doi: [10.1109/MICAI.2015.12](https://doi.org/10.1109/MICAI.2015.12).  
URL: <https://www.scopus.com/inward/record.uri?eid=2-s2.0-84987776731&doi=10.1109%5C2fMICAI.2015.12&partnerID=40&md5=fd529a25b1087de90fb4fefdaae0dbf4>.
- [3] Valentin Bazarevsky **and others**. *BlazePose: On-device Real-time Body Pose tracking*. 2020. doi: [10.48550/ARXIV.2006.10204](https://doi.org/10.48550/ARXIV.2006.10204).  
URL: <https://arxiv.org/abs/2006.10204>.
- [4] A. Chakrabarty **and others**. ?Autonomous indoor object tracking with the Parrot AR.Drone? incited By 24: 2016, **pages** 25–30. doi: [10.1109/ICUAS.2016.7502612](https://doi.org/10.1109/ICUAS.2016.7502612).  
URL: <https://www.scopus.com/inward/record.uri?eid=2-s2.0-84979779834&doi=10.1109%5C2fICUAS.2016.7502612&partnerID=40&md5=9c5bd30b53e234b7a03225a07557269a>.
- [5] 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/> (**urlseen** 14/01/2023).
- [6] Holybro. *Pixhawk 4 - Pinouts*.  
URL: <http://www.holybro.com/manual/Pixhawk4-Pinouts.pdf>.
- [7] Anis Koubâa **and others**. ?Micro Air Vehicle Link (MAVlink) in a Nutshell: A Survey? in *IEEE Access*: 7 (2019), **pages** 87658–87680. doi: [10.1109/ACCESS.2019.2924410](https://doi.org/10.1109/ACCESS.2019.2924410).
- [8] Google LLC. *Hands - mediapipe*.  
URL: <https://google.github.io/mediapipe/solutions/hands.html> (**urlseen** 14/01/2023).
- [9] Google LLC. *Pose - mediapipe*.  
URL: <https://google.github.io/mediapipe/solutions/pose.html> (**urlseen** 14/01/2023).

- [10] W. Mao **and others.** ?Indoor follow me drone? incited By 33: 2017, **pages** 345–358.  
DOI: [10.1145/3081333.3081362](https://doi.org/10.1145/3081333.3081362).  
URL: <https://www.scopus.com/inward/record.uri?eid=2-s2.0-85026207294&doi=10.1145%5C2f3081333.3081362&partnerID=40&md5=8b6c37ffaf6a7f02300b1c28b7a6d53d>.
- [11] Microsoft. *Build on Windows - Airsim*.  
URL: [https://microsoft.github.io/AirSim/build\\_windows/](https://microsoft.github.io/AirSim/build_windows/) (**urlseen** 16/01/2023).
- [12] J. Pestana **and others.** ?Computer vision based general object following for GPS-denied multirotor unmanned vehicles? incited By 73: 2014, **pages** 1886–1891.  
DOI: [10.1109/ACC.2014.6858831](https://doi.org/10.1109/ACC.2014.6858831).  
URL: <https://www.scopus.com/inward/record.uri?eid=2-s2.0-84905717349&doi=10.1109%5C2fACC.2014.6858831&partnerID=40&md5=6569abf0d533d8c92a42aedadfb9ce8e>.
- [13] J. Pestana **and others.** ?Vision based GPS-denied Object Tracking and following for unmanned aerial vehicles? incited By 64: 2013. DOI: [10.1109/SSRR.2013.6719359](https://doi.org/10.1109/SSRR.2013.6719359).  
URL: <https://www.scopus.com/inward/record.uri?eid=2-s2.0-84894224375&doi=10.1109%5C2fSSRR.2013.6719359&partnerID=40&md5=b62d374446ced88aebe4e5f875057f52>.
- [14] *PX4 User Guide*. The Linux Foundation. URL: <https://docs.px4.io/main/en/> (**urlseen** 13/01/2023).
- [15] Renderpeople. *Over 4,000 Scanned 3D People Models*.  
URL: <https://renderpeople.com/> (**urlseen** 16/01/2023).
- [16] Shital Shah **and others.** *Aerial Informatics and Robotics Platform*. techreport MSR-TR-2017-9. Microsoft Research, 2017.
- [17] PX4 team. *Pixhawk 4 - PX4 User Guide*. Dronecode Foundation.  
URL: [https://docs.px4.io/main/en/flight\\_controller/pixhawk4.html](https://docs.px4.io/main/en/flight_controller/pixhawk4.html) (**urlseen** 14/01/2023).
- [18] Fan Zhang **and others.** *MediaPipe Hands: On-device Real-time Hand Tracking*. 2020.  
DOI: [10.48550/ARXIV.2006.10214](https://doi.org/10.48550/ARXIV.2006.10214). URL: <https://arxiv.org/abs/2006.10214>.