






Article

# A Middleware Infrastructure for Programming Vision-Based Applications in UAVs

Pedro Arias-Perez <sup>1</sup>, Jesús Fernández-Conde <sup>2,\*</sup>, David Martín Gómez <sup>3</sup>, José M. Cañas <sup>2</sup>  
and Pascual Campoy <sup>1</sup>

<sup>1</sup> Computer Vision and Aerial Robotics Group, Centre for Automation and Robotics, Universidad Politécnica de Madrid, 28006 Madrid, Spain

<sup>2</sup> Department of Telematic Systems and Computation, Rey Juan Carlos University, Fuenlabrada, 28942 Madrid, Spain

<sup>3</sup> Intelligent Systems Lab, Universidad Carlos III de Madrid, Calle Butarque 15, Leganés, 28911 Madrid, Spain

\* Correspondence: [jesus.fernandez@urjc.es](mailto:jesus.fernandez@urjc.es)

**Abstract:** Unmanned Aerial Vehicles (UAVs) are part of our daily lives with a number of applications in diverse fields. On many occasions, developing these applications can be an arduous or even impossible task for users with a limited knowledge of aerial robotics. This work seeks to provide a middleware programming infrastructure that facilitates this type of process. The presented infrastructure, named DroneWrapper, offers the user the possibility of developing applications abstracting the user from the complexities associated with the aircraft through a simple user programming interface. DroneWrapper is built upon the de facto standard in robot programming, Robot Operating System (ROS), and it has been implemented in Python, following a modular design that facilitates the coupling of various drivers and allows the extension of the functionalities. Along with the infrastructure, several drivers have been developed for different aerial platforms, real and simulated. Two applications have been developed in order to exemplify the use of the infrastructure created: follow-color and follow-person. Both applications use techniques of computer vision, classic (image filtering) or modern (deep learning), to follow a specific-colored object or to follow a person. These two applications have been tested on different aerial platforms, including real and simulated, to validate the scope of the offered solution.

**Keywords:** middleware; aerial robotics; computer vision; deep learning



**Citation:** Arias-Perez, P.; Fernández-Conde, J.; Martín Gómez, D.; Cañas, J.M.; Campoy, P. A Middleware Infrastructure for Programming Vision-Based Applications in UAVs. *Drones* **2022**, *6*, 369. <https://doi.org/10.3390/drones6110369>

Academic Editor: Federico Tombari

Received: 26 October 2022

Accepted: 18 November 2022

Published: 21 November 2022

**Publisher's Note:** MDPI stays neutral with regard to jurisdictional claims in published maps and institutional affiliations.



**Copyright:** © 2022 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (<https://creativecommons.org/licenses/by/4.0/>).

## 1. Introduction

The field of robotics and unmanned aerial vehicles (UAVs) is a sector characterized by strong expansion in recent times, with very relevant growth expectations and increased demand for qualified personnel at the international level for the coming years.

The importance of autonomous flight has been reflected in a flourishing increment in the use of UAVs. UAVs are becoming widely used in both military and civil applications, due to their small size and high maneuverability. The use of UAVs has expanded to almost all areas, particularly in civil applications. In the area of agriculture, the rapid evolution of UAVs has led to precision agriculture applications [1] such as aerial crop monitoring and smart spraying tasks [2]. In the industrial field, UAV developments improve the efficiency of missions such as industrial inspection (e.g., photovoltaic plants) [3], cargo identification and delivery, or logistics, strongly linked to visual SLAM techniques (VSLAM). In addition, UAVs can also be seen in search-and-rescue, topography, or surveillance tasks, among other applications.

Despite continuous technological advances, the development of applications for UAVs is still complex. These types of systems are heterogeneous (very different flight platforms),

with many integrated components (autopilots, control and stabilization subsystems, communications, navigation, perception, etc.), and need to perform correctly in dangerous scenarios, where a single error can lead to the crash of the aircraft or damage to third parties.

For this reason, and due to the large social and economic expansion that this field is undergoing, it has been observed that there is a need for tools that facilitate the development of applications for the extended use of UAVs. These tools should allow abstracting from the complexities associated with an aircraft and even from the development platform itself.

This work pursues that end. It seeks to provide the user with an environment for the programming and navigation of aerial robots, which allows users to focus their efforts on building up only the final application of the UAV, forgetting about the rest of the complexities. This article details the design, implementation, and testing of a middleware infrastructure for the programming and navigation of aerial robots. The design follows a modular structure where the different blocks that make up the system can be replaced for adaptation to the problem. This modularity, in turn, allows the reuse of the code in various circumstances.

The middleware infrastructure developed allows the programming and navigation of UAVs, creating a software solution that offers the user a well-defined Application Programming Interface (API) that facilitates and homogenizes the development of applications.

The proposed middleware infrastructure is intuitive and easy to use (users do not have to be UAV experts), as well as robust and safe, as it is software that works directly with aerial robots. It should be noted that the applications built on top of our middleware infrastructure can be created by users not conscious of the rest of the software involved and even the aircraft used. In addition, the developed infrastructure is focused on users not possessing expertise in UAVs (although they should have some basic notions of robotics).

This middleware is intended for the use of different aerial platforms. Thus, a major objective of the infrastructure is that it should be platform-independent and capable of being used with aircraft of different natures within multi-copters. A variety of multi-copters used during the tests will give a better perspective of the scope of the work carried out.

Among the possible applications, the middleware developed seeks to offer solutions for the creation of autonomous navigation algorithms. Beyond the typical outdoor GPS-based position control, indoor position-based applications (in the absence of GPS) are being explored, such as visual auto-location algorithms (visual SLAM), or visual control applications, not based on position.

In order to validate the infrastructure developed, two different illustrative applications have been implemented, similar to those that may be developed by future users. These applications will make use of the middleware infrastructure developed and will be based on visual reactive control, showing examples of different levels of technical difficulty. We will start from the basic use of the tools collected in this work, progressing to the development of navigation algorithms based on deep learning.

#### *Related Work*

In [4], an architecture and open-source software framework for aerial robotics is presented, named Aerostack. This architecture integrates a complete multi-layered architectural organization to support fully autonomous flights and a versatile software framework for multiple uses. It is compatible with several well-known aerial platforms. The authors mention that there is room for improvement in the intelligent layer and robustness of the platform. Besides, there is no reference to simulated platforms or computer vision algorithms.

A software layer to abstract the users of unmanned aerial vehicles from the specific hardware of the platform and the autopilot interfaces is presented in [5]. The main objective of this aerial vehicle abstraction layer is to simplify the development and testing of higher-level algorithms in aerial robotics by trying to standardize and simplify the interfaces with unmanned aerial vehicles. This UAV abstraction layer can work seamlessly with simulated or real platforms, providing calls to issue standard commands such as taking off,

landing or pose, and velocity controls. A stable version is available for public use. All the code has been implemented in C++ language. Several use cases are presented (multiple drones for media production, autonomous inspection, aerial manipulation), but no modern vision-based applications (using deep learning techniques) are addressed.

A multirotor UAV control and estimation system for supporting replicable research through realistic simulations and real-world experiments is introduced in [6]. A unique multi-frame localization paradigm for estimating the states of a UAV in various frames of reference using multiple sensors simultaneously is proposed. An actively maintained and well-documented open-source implementation, including realistic simulation of UAVs, sensors, and localization systems is provided.

Regarding infrastructures that work only with simulated UAVs, a modular simulation framework is proposed in [7]. The simulator was designed in a modular way, such that different controllers and state estimators can be used interchangeably; incorporating new UAVs is reduced to a few steps. Another multi-rotor UAVs' simulation platform is presented in [8]. It is customizable and open-source. This platform (XTDrone) integrates dynamic models, sensor models, control and state estimation algorithms, and 3D scenes. Users can test their own algorithms, such as SLAM, object detection, motion planning, attitude control, multi-UAV cooperation, and cooperation with other robots on the platform.

Concerning reactive vision-based applications for UAVs, some research works worth mentioning are a robust and accurate vision-based landing technology for drones on moving targets [9], real-time visual object tracking [10], image-based indoor visual tracking of 3D objects [11], a visual tracking system for a drone to follow an omnidirectional mobile robot and land on it when it stops moving [12], and a deep learning network for vision-based UAV recognition [13].

## 2. Materials and Methods

### 2.1. Hardware

In coherence with the objectives of the project, three different UAVs were selected on which to carry out the tests of the developed software. The choice of these UAVs was made to cover a wide range of options, the three quadcopters being very different from each other. Among those selected are real/simulated, proprietary/free, and indoor/outdoor flight UAVs. Covering a wide variety of possibilities means providing the software with high versatility and being more easily extensible to other platforms in the future. The three chosen aircraft are the following: simulated 3DR Iris; DJI Tello; and self-built PX4.

First up is a simulated aircraft, 3DR Iris from 3D Robotics. Using a simulated aircraft in the early stages of development offers great advantages in time and cost savings since errors in the code do not cause damage to the material. The aircraft uses PX4 as an autopilot and is simulated by Gazebo. More details about this aircraft will be given in the software section (see Section 2.2).

The second aircraft selected was the DJI Tello (Figure 1). It is a proprietary aircraft designed for indoor flights due to its small size and weight. It includes a telemetric sensor (optical flow) and a barometer, for odometry calculations, a vision sensor, and a Wi-Fi antenna for communications.

Due to its characteristics, it is appropriate for carrying out the first indoor navigation tests. Its size and weight ensure limited damage to third parties in the event of an accident, and despite its appearance, it is a robust aircraft with great flight performance. In addition, the manufacturer itself offers a development kit (SDK) for programming applications on the drone. However, its use is limited by communication with the ground station since the processing cannot be completed onboard the aircraft.

Thirdly, a self-built drone [14] developed by A. Madridano together with the Intelligent Systems Laboratory of the Carlos III University of Madrid [15] was chosen. The aircraft has a PixHawk v1 controller (manufactured by 3D Robotics, Berkeley, CA, USA), with the PX4 autopilot in its latest stable version. The aircraft has a computer onboard that allows for

on-site information processing. It also includes a GPS antenna, a telemetry receiver, and a Wi-Fi antenna, among other sensors.



**Figure 1.** DJI Tello UAV.

This quadcopter, shown in Figure 2, is intended for use in the later stages of development as it is the most powerful aircraft of the three. Performing the processing on board allows more complex tasks to be carried out outdoors without limitations with communication with the ground station that could otherwise become a bottleneck.

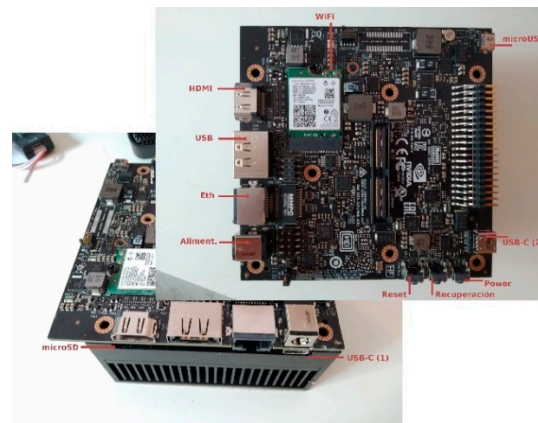


**Figure 2.** Non-commercial UAV.

The increased interest in real-time vision applications has led to the development of low-power embedded devices for integration into mobile robotic systems. Examples of these are devices such as Arduino [16] or Raspberry Pi [17], used in small robots such as PiBot [18] or GoPiGo [19].

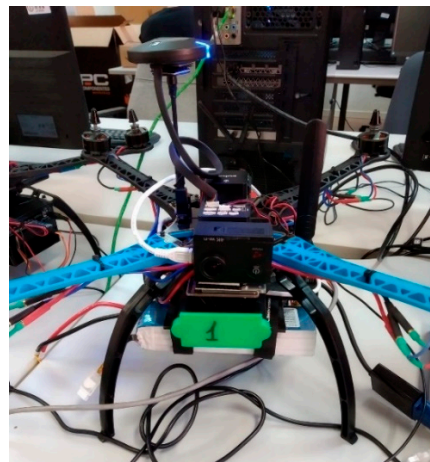
Nevertheless, to execute complex algorithms, such as neural networks, the devices mentioned are not suitable and it is necessary to use specific devices. A viable alternative is to use Jetson devices manufactured by NVIDIA [20]. Each NVIDIA Jetson is a complete system-on-module (SOM) that includes the CPU, GPU, memory, power management, high-speed interfaces, and more, enabling the execution of CUDA [21], a low-level parallel computing library, as well as various toolkits (such as JetPack SDK [22]) designed to optimize processes running on the device.

The size and power consumption of these devices make them ideal systems for embarking on aerial vehicles. There are different models available such as the Jetson Nano, the Jetson TX2, or the Jetson AGX Xavier. In our case, the Jetson AGX Xavier has been selected as part of the aircraft. Figure 3 shows the main elements of the device.



**Figure 3.** NVIDIA Jetson AGX Xavier.

For the development of visual control applications, it is necessary to have a vision sensor onboard the aircraft. Both the simulated aircraft and the Tello have an onboard camera, in the first case a simulated sensor, while in the second, the sensor is integrated into the drone itself. However, for the self-developed drone, it is necessary to select a camera, since the initial design did not include any. Specifically, the selected sensor is a basic Victure AC600 USB camera (manufactured by Govicture, Shenzhen, China). The final assembly of the camera system to the UAV is shown in Figure 4.



**Figure 4.** UAV with camera sensor assembled.

## 2.2. Software

The language chosen for the development of the middleware infrastructure was Python [23]. In addition, for certain aspects of development, C++ was also used [24].

The selected robotic software platform is ROS [25,26]. In addition to being the most widely used platform, it has libraries in different languages that facilitate its use. One of them is rospy [27], a Python client that allows rapid interaction with ROS nodes, services, and parameters.

ROS has different packages that facilitate software development in multiple fields. Within aerial robotics, there is a collection of extensible MAVLink communication nodes for ROS known as MAVROS (Micro Air Vehicles ROS) [28]. This ROS package provides a communication controller for several autopilots with the MAVLink communication protocol, together with a collection of nodes, services, and parameters that ensure correct communication with the aircraft. The MAVROS version used is v1.9.0 (Open Robotics, Mountain View, CA, USA), with MAVLink message collection v2021.3.3.

For image processing, OpenCV (Open-Source Computer Vision) [29], version v3.2.0 (Intel Corporation, Mountain View, CA, USA), was chosen.

Among the drones used, the simulated 3DR Iris was introduced in the previous section. As has been anticipated, the simulation is carried out using the Gazebo simulator (Open Robotics, Mountain View, CA, USA). This open-source simulator is the most used in robotics and artificial vision applications. Due to its open management, it allows the integration of multiple vehicles, worlds, sensors, physics, etc. The version used during the project is Gazebo9. The Iris model simulated in the simulator can be seen in Figure 5. The model includes two plugins that allow one to obtain images (one frontal and one ventral) in a similar way to a camera. The firmware used by the controller is PX4 [30] with version v1.11.3 (Dronecode Foundation, San Francisco, CA, USA), in SITL (“software-in-the-loop” simulation) mode.



**Figure 5.** 3DR Iris simulated UAV in Gazebo.

The embedded device used, the NVIDIA Jetson AGX Xavier, follows highly optimized, embedded design guidelines. A custom version of Ubuntu Linux, called NVIDIA JetPack, is developed and maintained by NVIDIA company (Santa Clara, CA, USA) and is available for download and installation as board firmware [22]. This custom implementation includes low-level interfaces to implement parallel computing operations (CUDA) and various SDK (Software Development Kit) optimizations, such as TensorRT. For the developed system, the version used is JetPack 4.6.

For this work, one of the objectives is the development of a deep learning application in a drone. An already created and widely used neural network, YOLO (You Only Look Once), has been used. YOLO is a state-of-the-art real-time object detection system, designed by Joseph Redmond up to its third version [31]. This system was continued by Alexey Bochkovskiy, creator of the most current versions.

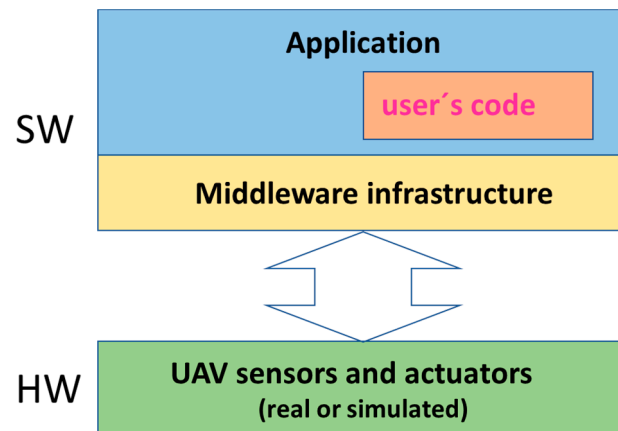
Its approach, very innovative when it was launched, allows it to reach very high execution speeds. With this method, the neural network is applied only once to the image. This network divides the image into regions and predicts bounding boxes and probabilities for each region.

The network version used for this work is YOLOv4 [32]. The default configuration and weights of the YOLOv4 and YOLOv4-tiny versions have been used. These weights were obtained by training with the Microsoft COCO (Common Objects in Context) database [33], which has 80 different classes and more than 300,000 images, and one and a half million labeled objects.

### 3. Middleware Infrastructure Developed: DroneWrapper

The general outline of the problem is presented in Figure 6. Three different layers are distinguished in the scheme. At the bottom is the layer corresponding to the aircraft, while at the top is the user, interested in developing an application to control the aircraft.

In between is the tool developed, which faces the challenge of communicating with the aircraft, generally a complex task, and offering the user a simple interface.



**Figure 6.** Multi-layer scheme of the middleware infrastructure.

To solve the problem of communication with the aircraft, it has been decided to use MAVROS, presented in the previous section. MAVROS establishes an architecture of nodes, topics, and services that allow communication with the aircraft.

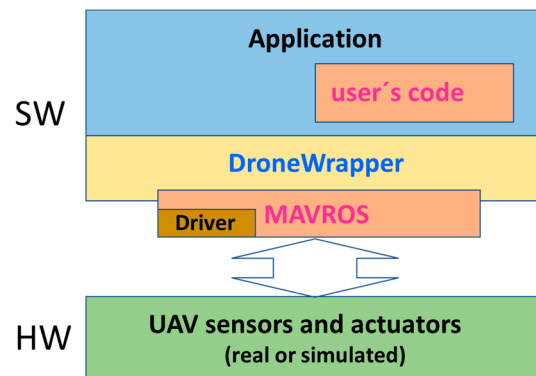
The interaction with the user is solved by offering a ROS package easily importable from Python. The package, called DroneWrapper, provides a user programming interface that allows the control of either physical or simulated aircraft. The package has a DroneWrapper class, with the same name as the package, whose methods provide all kinds of tasks for working with a multi-copter.

It is important to note that the security and robustness objectives are met with MAVROS. Communication has the robustness of ROS, while security is present when using version 2.0 of MAVLink, which allows, among other security aspects, the encryption of messages.

With this design, you can see how communication is achieved through MAVROS and within the presented DroneWrapper package. According to this last scheme, MAVROS has to be able to understand and communicate with both ends of the communication. DroneWrapper does not present any problem, since it has been developed for this reason, but the aircraft can cause some difficulty.

MAVROS supports the main flight controllers such as PX4, present in two of the three aircraft used. However, Tello does not have support from MAVROS, nor from ROS, as it is a private controller. To solve this problem, a communications driver that simulates MAVROS, called Tello Driver, was devised and programmed.

Tello Driver offers, in the same way as MAVROS does, a series of nodes and services that allow communication with DroneWrapper. On the other hand, to communicate with the aircraft, the official Tello SDK [34] is used, which allows the aircraft to be controlled with messages specified by the manufacturer. Following this consideration, the presented scheme is slightly modified. The new design is shown in Figure 7.



**Figure 7.** Design of the DroneWrapper middleware infrastructure.

Just as the Tello needs a particular communications driver, other aircraft may need other specific communication drivers to make use of the DroneWrapper tool. Other peripheral elements may also need drivers to fit into the infrastructure provided. This is the case of the Victure AC60 USB camera used in the self-built aircraft used. To deliver the images through the DroneWrapper interface, another driver has also been developed, called the Victure driver.

Both drivers are presented to the user in the form of ROS packages, which the user can include in the software framework according to their needs.

Before continuing with the implementation details of the different packages, it is necessary to discuss certain aspects of the design for a better understanding of it. In the first place, a commitment has been made to design a main horizontal package that brings together all the common aspects of the infrastructure. On this rests a modular architecture, where the different modules (drivers, as previously introduced) can be included according to the needs of the aircraft or the user.

The design is intended to reflect DroneWrapper as a kind of generic middleware for multi-copters, independent of the specific low-level drivers for each aircraft. For aircraft with PX4 and ArduPilot flight controllers, MavLink and MAVROS are used directly as communication elements. In addition, the DroneWrapper abstracts the fundamental functions such as speed control and position control, as well as data from the usual sensors onboard the aircraft.

### 3.1. DroneWrapper Package

DroneWrapper is organized similarly to a typical ROS package. It is wrapped in a meta-package with other packages, such as Tello Driver. The meta-package aggregates JdeRobot's drone widgets. The code is openly available in the JdeRobot drones/drone\_wrapper repository ([https://github.com/JdeRobot/drones/tree/melodic-devel/drone\\_wrapper](https://github.com/JdeRobot/drones/tree/melodic-devel/drone_wrapper), accessed on 26 October 2022).

DroneWrapper, like any ROS package, uses several tools to maintain communication. These tools are nodes, topics, services, and parameters. Nodes are processes, topics are communication channels between two nodes, services are communication methods on request, and parameters are used to store and manipulate data.

The running schematic of DroneWrapper is shown in Figure 8. The graph shows the topics or message exchange channels. On both sides the different nodes are existing. On the one hand, MAVROS (/mavros in the figure), is in charge of performing communication with the aircraft, and on the other hand, DroneWrapper (/drone), is accessible to the user. The MAVROS node is the standard of the package, which runs as its documentation indicates.



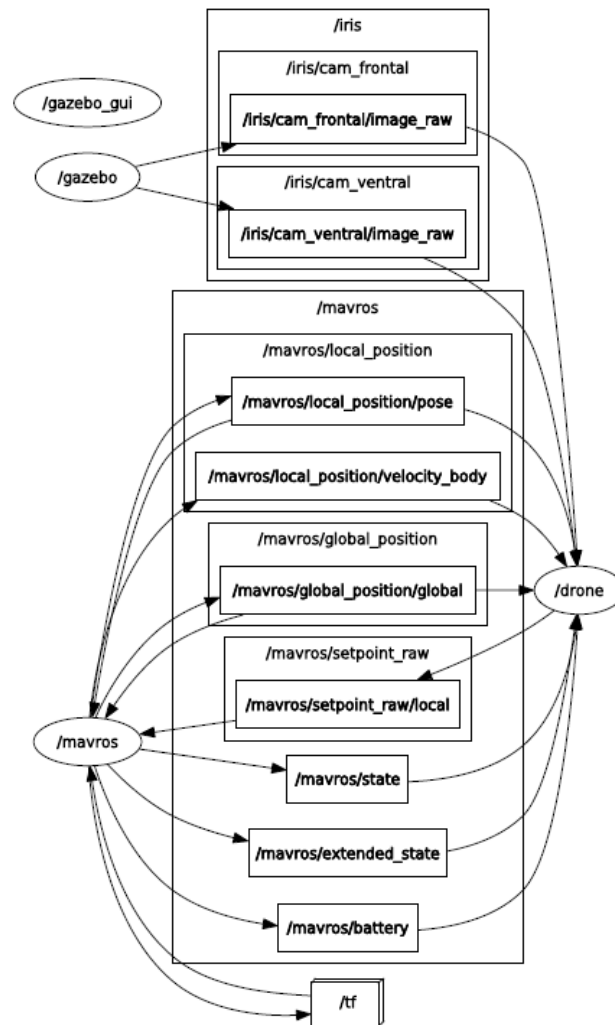


Figure 8. DroneWrapper ROS nodes and topics graph.

About the topics, we distinguish two groups, the publishers of messages and the receivers or subscribers of messages. Note that each topic is either a publisher or a subscriber depending on the node on which attention is focused. The classification is completed on the DroneWrapper node as the package that is intended to be explained in this section.

Among the message publishers (and to which the application subscribes) are eight topics, which send aircraft status information, such as position or battery data, along with camera images (in this case from the plugins when encountering a simulated aircraft).

On the other hand, there is only one subscriber (to whom the application sends messages), who is in charge of sending commands and action orders to the aircraft. Table 1 shows the topics used along with the type of message used.

Table 1. Topics and types of messages used in DroneWrapper.

Group	Topic	Message Type
Publisher	/mavros/state	mavros_msgs/State()
Publisher	/mavros/extended_state	mavros_msgs/ExtendedState()
Publisher	/mavros/local_position/pose	geometry_msgs/PoseStamped()
Publisher	/mavros/local_position/velocity_body	geometry_msgs/TwistStamped()
Publisher	/mavros/global_position/global	sensor_msgs/NavSatFix()
Publisher	/mavros/battery	sensor_msgs/BatteryState()
Publisher	/iris/cam_frontal/image_raw	sensor_msgs/Image()
Publisher	/iris/cam_ventral/image_raw	sensor_msgs/Image()
Subscriber	/mavros/setpoint_raw/local	mavros_msgs/PositionTarget()

Control over the aircraft is achieved through the topic `/mavros/setpoint_raw/local`. It is important to note that for the autopilots to be able to respond correctly to these messages, they need to be in a specific flight mode. In the case of PX4, this flight mode is OFFBOARD.

The `PositionTarget()` message allows the use of different coordinate frames (`coordinate_frame`). The application always uses the same axis, `FRAME_BODY_NED`, which behaves in the same way as a local axis, fixed to the take-off point, and with a North-East-Down (NED) orientation, for position data.

This message also allows different types of control, in position, speed, acceleration, force, and mixed controls, through the last fields of the message. These controls are selected based on the active “`type_mask`” mask. Note that not all masks are valid.

DroneWrapper supports three types of control: position control; speed control; and mixed control based on the speed with fixed flight height. The masks used are illustrated in Table 2.

**Table 2.** DroneWrapper masks and active fields for the different types of control.

Control	Mask	Active Fields
Position	3064	x y z yaw
Velocity	1991	vx vy vz yaw_rate
Mixed	1987	vx vy vz z yaw_rate

In addition to topics, the application makes use of services and parameters. The services are used to launch requests to aircraft of various kinds. These requests take care of arming the aircraft, landing, changing modes, and manipulating parameters. Table 3 lists the services used by DroneWrapper.

**Table 3.** Services and types of messages used in DroneWrapper.

Service	Message Type
<code>/mavros/cmd/arming</code>	<code>mavros_msgs/CommandBool()</code>
<code>/mavros/cmd/land</code>	<code>mavros_msgs/CommandTOL()</code>
<code>/mavros/set_mode</code>	<code>mavros_msgs/SetMode()</code>
<code>/mavros/param/set</code>	<code>mavros_msgs/ParamSet()</code>
<code>/mavros/param/get</code>	<code>mavros_msgs/ParamGet()</code>

Until now, the internal operation of the infrastructure (closest to the aircraft hardware) has been presented. Next, the other end, closer to the user, will be explained. It has already been stated that DroneWrapper is presented to the user as an importable package in Python and with a series of methods (API) that allow operating with the aircraft.

A simple use case is presented in Listing 1. In it, firstly, an object is created that represents the drone and gives access to all the functionalities present in the package. Next, it is ordered to take off and after that, the drone spins around for several seconds. Finally, the drone lands in its current position. It should be noted that the nodes scheme presented in Figure 8 has been obtained with the simulated aircraft and the code shown.

**Listing 1.** DroneWrapper simple use case.

```

1  #!/usr/bin/env python
2  from drone_wrapper import DroneWrapper
3  from time import sleep
4
5  drone = DroneWrapper()
6  drone.takeoff(h=2.5)
7  drone.set_cmd_vel(az=1) # spin
8  sleep(5) #wait for a few seconds
9  drone.land()

```

Finally, the API present in DroneWrapper is shown in Table 4. It includes the methods that allow obtaining information on the sensors and status of the aircraft, the methods to control the aircraft, and the methods to obtain images from the aircraft’s cameras.

**Table 4.** API offered by DroneWrapper.

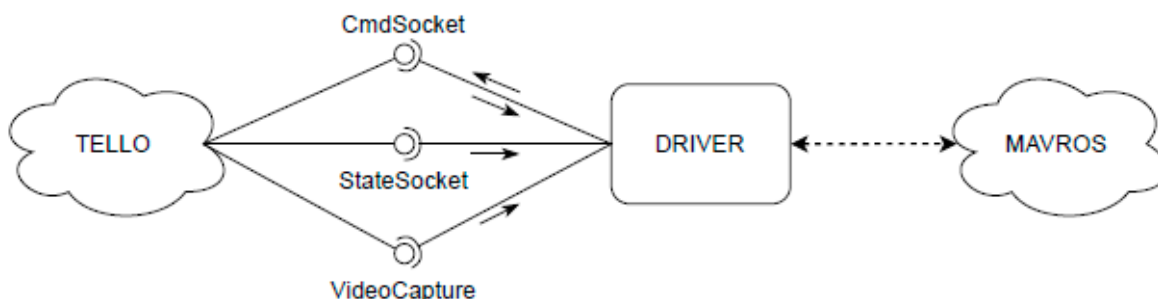
Category	API	Description
Sensors and state	[x, y, z] = get_position()	Return aircraft position (m)
	[vx, vy, vz] = get_velocity()	Return aircraft velocity (m/s)
	rate = get_yaw_rate()	Return aircraft yaw rate (rad/s)
	[r, p, y] = get_orientation()	Return aircraft orientation (rad)
	r = get_roll()	Return aircraft roll angle (rad)
	p = get_pitch()	Return aircraft pitch angle (rad)
	y = get_yaw()	Return aircraft yaw angle (rad)
Control	s = get_landed_state()	Return aircraft landed state (ground, flight, landing)
	takeoff(h)	Take off to height h (m)
	land()	Land in current position
	set_cmd_pos(x, y, z, yaw)	Position control x, y, z (m) and yaw (rad)
	set_cmd_vel(vx, vy, vz, vyaw)	Velocity control vx, vy, vz (m/s) and yaw_rate (rad/s)
Cameras	set_cmd_mix(vx, vy, z, vyaw)	Mixed control vx, vy (m/s), z (m) and yaw_rate (rad/s)
	img = get_frontal_image()	Returns image from frontal camera
	img = get_ventral_image()	Returns image from ventral camera

### 3.2. Tello Driver

In the same way as the previous package, TelloDriver is a ROS package and is organized as such. As with DroneWrapper, the Tello Driver package belongs to the JdeRobot drone meta-package. The code is openly available in the JdeRobot drones/tello\_driver ([https://github.com/JdeRobot/drones/tree/melodic-devel/tello\\_driver](https://github.com/JdeRobot/drones/tree/melodic-devel/tello_driver), accessed on 26 October 2022) repository.

Tello Driver has two main tasks, to communicate with the DroneWrapper and with the Tello aircraft. To clearly show their implementation, both parts will be presented separately, although one part does not make sense in the absence of the other.

In the design section, the use of the Tello SDK for communication with the physical aircraft has been advanced. Following the instructions for its use, the driver makes use of a series of sockets and threads to carry out the communication. The communication architecture is shown in Figure 9.



**Figure 9.** Tello driver communication scheme.

The driver has three sockets. The first one, CmdSocket, is used for sending commands and receiving command responses. It is the only bidirectional one of the three sockets. The second of the sockets, StateSocket, is used to receive status information from the aircraft. Finally, the VideoCapture socket is in charge of receiving the images sent from Tello.

All the information received is handled by three different message handlers, in secondary threads, which are responsible for listening to the information received through the three respective sockets. Instead, commands sent to the aircraft are serviced through the main driver thread.

The Tello driver manages communication handling with MAVROS on the other side. Similar to DroneWrapper, communication is via eight topics, seven publishers, one subscriber, and six services. The communication is almost identical to the one presented with DroneWrapper, as it tries to imitate the standard behavior of MAVROS so that the application does not notice the difference between different drones. The only differences are a new takeoff service, and the absence of one of the two image publishers since Tello only has a camera. Figure 10 illustrates the graph of nodes and topics used.

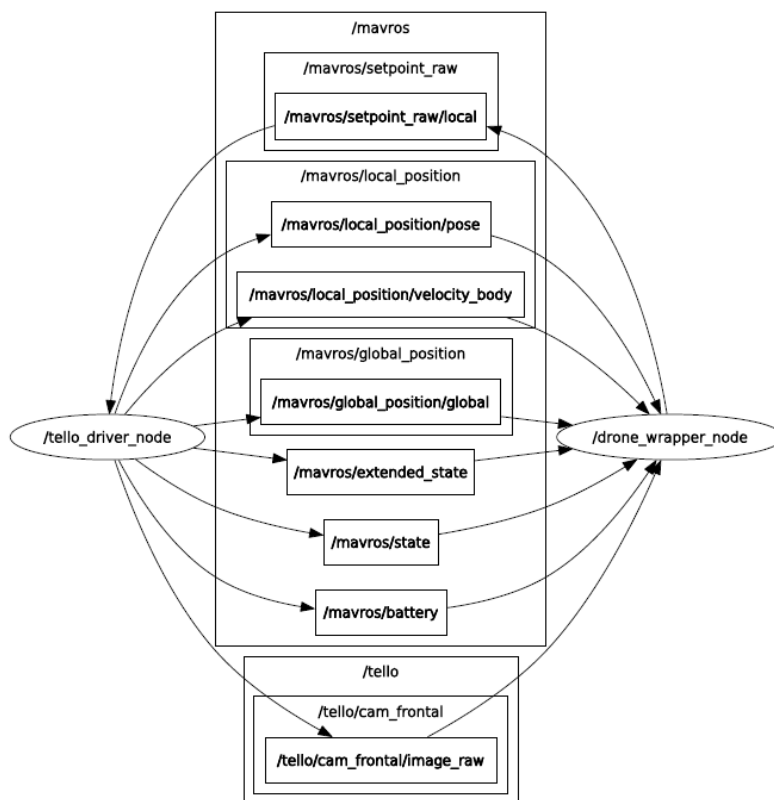


Figure 10. Tello driver ROS nodes and topics graph.

After explaining both ends of the driver, we still need to explain the middle part that converts ROS messages and services into a message language understood by the Tello SDK and vice versa. Sending action commands is executed by sending a text string through the socket previously shown. Depending on the content of the text string, the aircraft interprets one command or another. Although there are many commands accepted in the Tello SDK, the driver only uses the following: command, streamon, streamoff, emergency, takeoff, land, forward x, back x, left x, right x, up x, down x, cw x, ccw x, rc a b c d, battery.

The driver is in charge of translating the commands received by the topic subscriber and through the services into the different messages so that the aircraft performs the ordered task. The response to the command is obtained by one of the handlers and reported through MAVROS.

On the other hand, the state of the aircraft is received through the StateSocket in a text string that resembles this: "data:value;data2:value2; ... ..;\r\n". The set of data sent by Tello are the following (in order of appearance in the text string): pitch, roll, yaw, vgx, vgy, vgz, templ, temp, tof, h, bat, baro, time, agx, agy, agz. These values are processed in the driver, encapsulated in the different ROS messages, and sent through the different topics.

Finally, through the VideoCapture, the different frames that are retransmitted are received through the topic prepared for it.

### 3.3. Victure Camera Driver

The infrastructure-integrated camera driver, Victure driver, in the same way as the two previous packages, has a ROS package structure. The code is publicly available in the repository RoboticsLabURJC/2021-tfm-pedro-arias ([https://github.com/RoboticsLabURJC/2021-tfm-pedro-arias/tree/main/victure\\_driver](https://github.com/RoboticsLabURJC/2021-tfm-pedro-arias/tree/main/victure_driver), accessed on 26 October 2022).

The ROS structure is very simple. The driver consists of a node (`victure_cam`) that reads images from the camera and sends them through a topic (`victure_cam/image_raw`).

Finally, the node graph of the driver is shown in Figure 11. The figure shows the main node together with a test node (“test”), which is responsible for reading about the topic and displaying the received image on the screen.

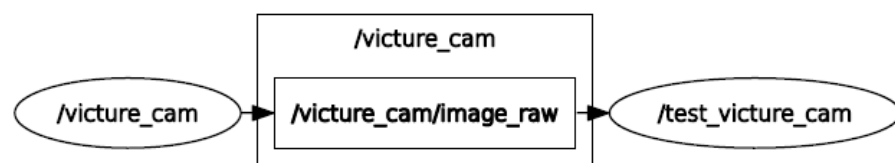


Figure 11. Victure camera driver ROS nodes and topics graph.

## 4. Results

This section includes the two applications developed on the infrastructure presented. Its main objective is to illustrate the use of the infrastructure, including vision management with Deep Learning and speed control that allows for validating its correct operation.

The two developed experiments are shown, “follow-color” and “follow-person” in different sections. In both, the designs devised, their implementation, and the results obtained with the different platforms are explained.

### 4.1. Follow-Color Application

The follow-color application consists of the aircraft following an element with a striking color. The application has been tested on two aircraft, the simulated drone, and the real Tello. The chromatic markers (elements to follow), used for each of the experiments are shown in Figure 12.

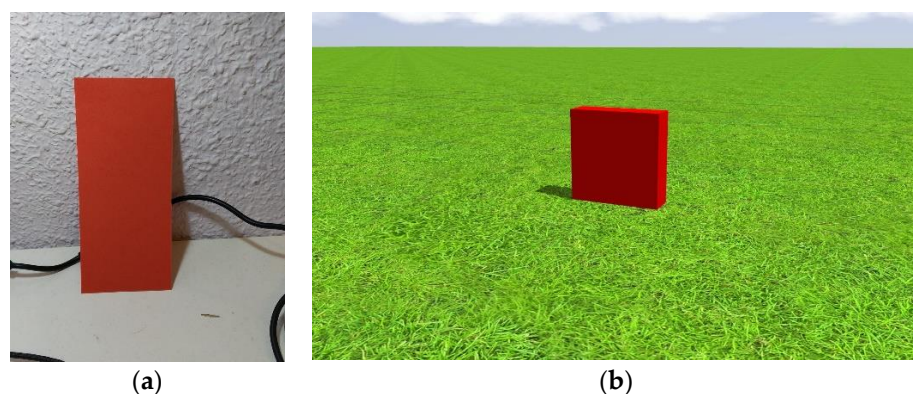


Figure 12. Beacons in the follow-color application. (a) Real beacon; (b) Simulated beacon.

Several Gazebo plugins have been developed on the simulated beacon to give the model movement, either with a predetermined trajectory or teleoperating the object from the keyboard.

The infrastructure allows sharing of the same source code for both the real and the simulated aircraft. However, the characteristics of the aircraft are very different. Using

the same logic for both drones is possible because the application uses configuration files where certain parameters intrinsic to the aircraft are saved.

These configuration files store data that allow the generic code to be adjusted to a specific aircraft. During process launch, the data are loaded as ROS parameters, thus allowing easy access to DroneWrapper and follow-color.

The aerial robot application design consists of two parts, perception and control. Perception is responsible for visually detecting the object to be followed, while control sends movement commands to the aircraft in order to follow the object.

The behavior of the aircraft is described next. After takeoff, an infinite iterative loop is started where the perception and control tasks are performed. The perception consists of filtering by color the image obtained by the drone's camera. When something is detected in perception, control comes into action. The control consists of three PID controllers that calculate the speeds commanded to the aircraft. If the perceptual filtering does not obtain any output, the aircraft performs a search algorithm until it finds a new object to follow. This search algorithm consists of turning around on itself at a constant speed.

#### 4.1.1. Perception

Perception is a color filtering of the image using classical techniques. The filtering is performed using the OpenCV computer vision library.

Filtering is generally executed on the HSV spectrum (Hue-Saturation-Value, or Hue-Saturation-Brightness) rather than on the RGB spectrum (Red-Green-Blue). This is because the HSV spectrum represents the color tone (Hue or Tone) in a single value, while the RGB spectrum needs three fields to represent the tone and is much more fragile against lighting changes in the scene, which makes filter design difficult.

The color filtering design consists of four stages:

1. Gaussian blur. Blur over a color image (RGB) to remove pixels spurious using `cv2.GaussianBlur()` and transformation to HSV space;
2. HSV mask. Mask over HSV space via `cv2.inRange()`, join bitwise image and mask (`cv2.bitwise_and()`) and conversion to scaled image of greys;
3. Threshold. Fixed level threshold (value = 150), on an image in the scale of `cv2.threshold()` greys;
4. Segmentation. Grouping by contours on a black and white image for object detection (`cv2.findContours()`).

The applied mask is, by the nature of the HSV space, a combination of two masks. This combination is the sum of both masks ( $\text{mask} = \text{mask1} + \text{mask2}$ ). This occurs due to the angular discontinuity in hue since hue values  $H = 1$  or  $H = 359$  are chromatically similar values even though they are numerically very different. Table 5 reflects the values selected for the mask.

**Table 5.** HSV masks.

Mask	H <sub>min</sub>	S <sub>min</sub>	V <sub>min</sub>	H <sub>max</sub>	S <sub>max</sub>	V <sub>max</sub>
mask1	0	70	50	10	255	255
mask2	340	70	50	359	255	255

These values can be somewhat confusing with their numerical representation. To facilitate its understanding, a graphic representation of the used masks is illustrated in Figure 13.

The process followed during the perception block is shown in Figure 14. As can be seen in the figure, the filtering results in a binary image (black and white). In this image, the detected pixels (in white) are grouped by contours into `cv2.findContours()` objects. In the case of detecting several objects, the object to be tracked is the one with the largest area.

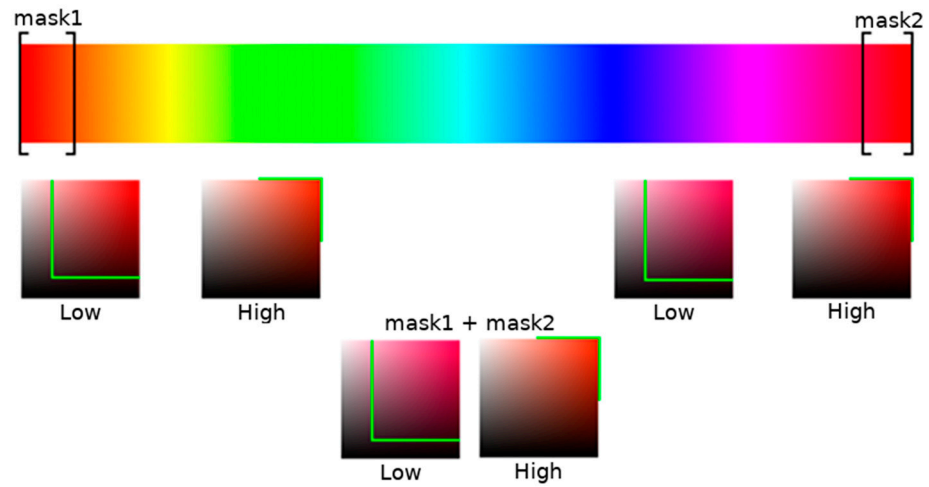


Figure 13. Color filter mask.

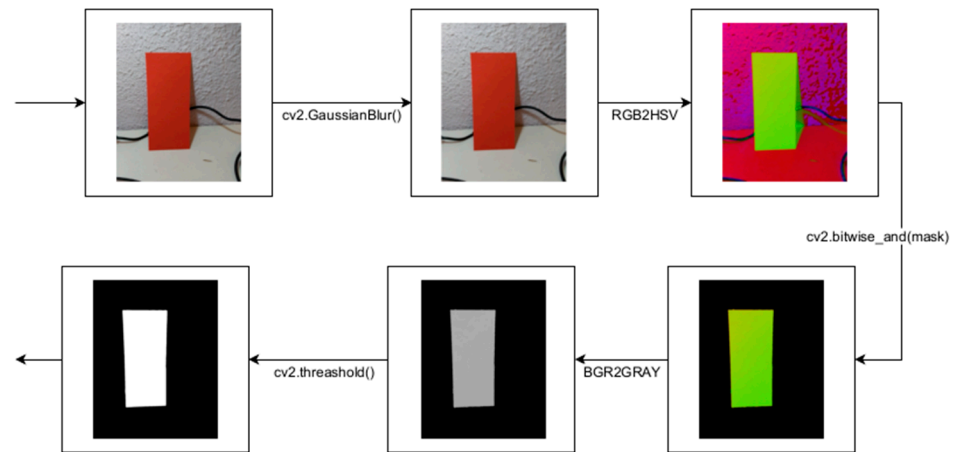


Figure 14. Follow-color perception scheme.

#### 4.1.2. Control

The control block is always executed with a single input, the object to follow. Properties such as the position on the image or the radius of the minimum circle surrounding the contour are extracted from this object. These values are used to calculate the errors or inputs of the controllers, whose outputs are the speeds to command the aircraft. Figure 15 represents a schematic of the control block.

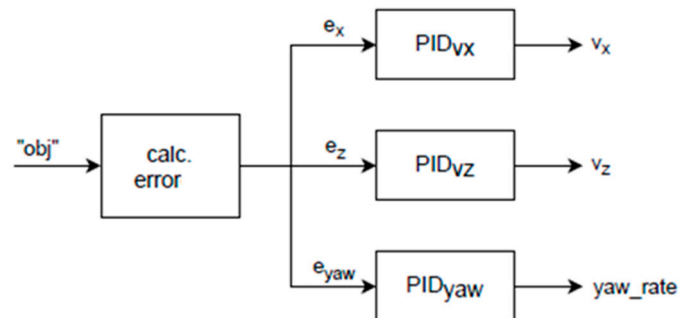


Figure 15. Follow-color control scheme.

Specifically, there are three PID controllers used, one that controls the advance ( $vx$ ), one that controls the height ( $vz$ ), and one that controls the yaw angle ( $yaw\_rate$ ). The inputs for the controllers are calculated according to the following formulas:

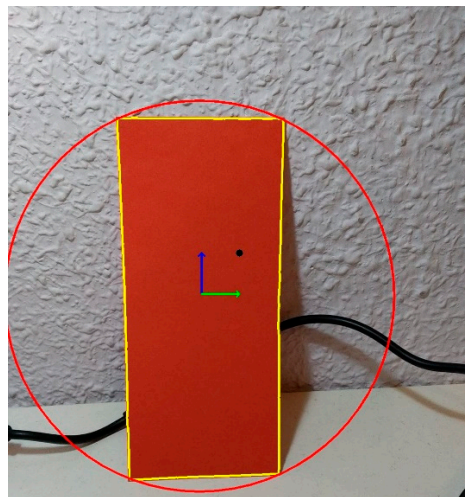
$$e_x = \frac{radius - target\_radius}{target\_radius} \quad (1)$$

$$e_z = c_y - obj_y \quad (2)$$

$$e_{yaw} = c_x - obj_x \quad (3)$$

The height and yaw control (Equations (2) and (3)) are performed according to the position of the centroid of the filtered object ( $obj_x, obj_y$ ) with respect to the center of the image ( $c_x, c_y$ ). The forward control (Equation (1)) is somewhat more complex since it uses the normalized difference of the radius of the minimum enclosing circle to the contour and a reference radius ( $target\_radius = 10$ ). In this fashion, the smaller the object in the image, the faster the UAV will advance, until the observed radius is close to the reference one again. Reference radius is related to a distance to the object of approximately 5 m.

Figure 16 shows the response given by the control block to any input. The arrows indicate the direction the drone will take to correct the existing error, bringing the center of the detected object to the center of the image (black dot). You can also see in the figure the contour of the filtered object together with the minimum circle surrounding the contour.



**Figure 16.** Control block response in follow-color.

The errors obtained are the feed of the controllers that try to reduce these values to zero with their response, which are directly the speed commands that are sent to the drone. The response of the controller depends on its parameters ( $k_P, k_I, k_D$ ). Since the control of the drone depends on its specific intrinsic characteristics for that model, the parameters of the controllers are part of the configuration files.

Controller parameters have been calculated experimentally for both aircraft. The final fit values are shown in Table 6.

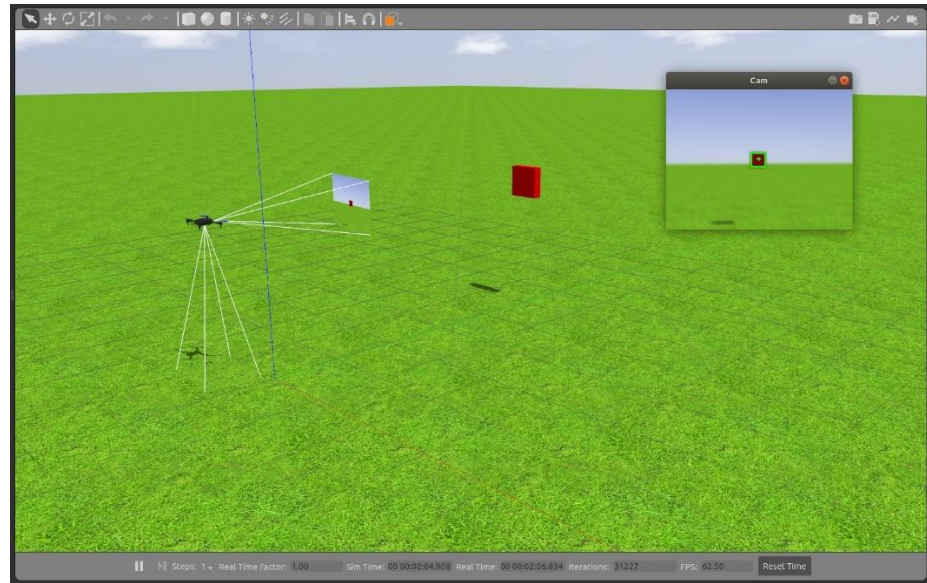
**Table 6.** Parameters for PID controllers.

Controller	Iris $k_P$	Iris $k_I$	Iris $k_D$	Tello $k_P$	Tello $k_I$	Tello $k_D$
vx	0.05	0.0	0.001	0.02	0.0	0.0002
vy	0.0	0.0	0.0	0.0	0.0	0.0
vz	-0.02	0.0	0.001	-0.0015	0.0	0.0
yaw_rate	-0.005	0.0	0.001	0.002	0.0	0.0001



### 4.1.3. Experiments and Results

The results obtained are presented in the form of different tests or experiments. In general, it starts with a simple case and increases its difficulty in successive cases. For follow-color, we have started from a simple experiment with the object to be followed in a static position, in simulation (see Figure 17).



**Figure 17.** Follow-color experiment, static object, simulation mode.

Secondly, the application has been tested with a moving object, first moving the object manually through the developed teleoperation tool, and with smooth movements. Next, we introduced an object with automatic movement and actions that are more abrupt and faster than in the previous case.

After successfully passing the simulated tests, the tests have been conducted on the real Tello drone. Similarly, the experiments carried out have been increasing in difficulty until obtaining a dynamic scenario with fast actions and sudden changes. The different experiments have been collected in different videos, available for viewing in the following playlist: <https://youtube.com/playlist?list=PL2ebURGAzRwusKLBYPJUkFZJ5SHudh6Z>, accessed on 26 October 2022.

### 4.2. Follow-Person Application

The follow-person application consists of following a person with the aircraft. There are several antecedents of similar applications for terrestrial robots [35,36]. The experiment has been carried out on the three available UAVs, the simulated Iris, the Tello, and the self-built PX4. For the simulation, a model belonging to the Ignition Robotics database [37] has been used. Several plugins have been designed on the model to give movement to the person. Figure 18 shows the model used.

The application follows the same design as the previous experiment. The developed infrastructure allows the use of the same source code with different configuration files. Thus, the body of the application is the same for all three aircraft. The application is made up of two blocks: perception and control. Perception is responsible for detecting the person to follow, while control is responsible for commanding the aircraft.

It is important to note that the perception is different between the two applications, detecting a colored object does not entail the same difficulty as detecting a person, while the control is identical between both applications.



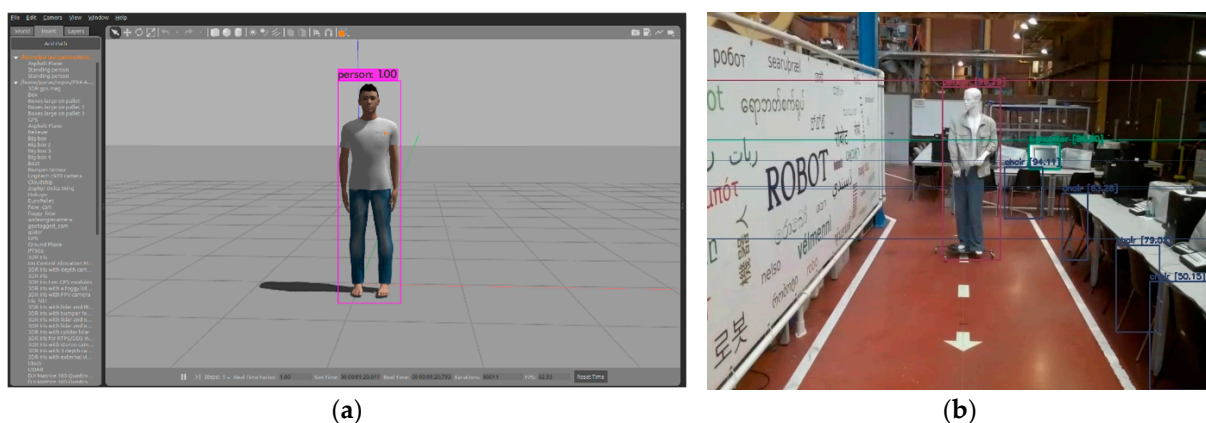
**Figure 18.** Model of a person in the Gazebo simulator. (a) Complete body; (b) Face.

#### 4.2.1. Perception

Perception consists of two parts: robust person detection and person identification. Detection is performed by deep learning, while identification is achieved by spatio-temporal tracking with a finite memory of detection.

The detection consists of a deep neural network, specifically YOLOv4 [32]. Deep learning detection adds robustness to the solution. Detection is reliable in many lighting scenarios, against occlusions or against multiple objects to be detected. However, its main disadvantage is its inference time, which, if not kept limited, slows down the control loops, deteriorating tracking to the point where it becomes impossible.

The operation of the network is simple. The network takes an image and returns a series of detections on it; what happens in between is hidden from the user. The detections consist of a label, a “confidence” (detection probability), and the position of the object within the image (in the form of a bounding box), for each of the detections. These detections are filtered by tag to only obtain the “people” detected. Figure 19 shows the detections completed by the network.



**Figure 19.** Detections in Follow-person. (a) Simulation; (b) Real.

The identification is carried out by storing previous positions of a detection considered as “main”. The selection of the main detection depends on the number of detections. Starting from a situation where there is no main detection, there is no object to track if the number of people detected is zero. In case only one person is detected, they are selected as primary, and in case there is more than one detection, the person with the highest confidence is chosen as primary.

Upon the main detection, the positions of the last centroids on the image are stored in a finite FIFO (First-In-First-Out) queue. The queue length is five, indicating that the last five positions of the identification are saved. On the new detections, their centroids are calculated and compared with the average of the centroids stored in the queue. The detection with the nearest centroid is considered as the object to track. Figure 20 shows the scheme of the identification queue.

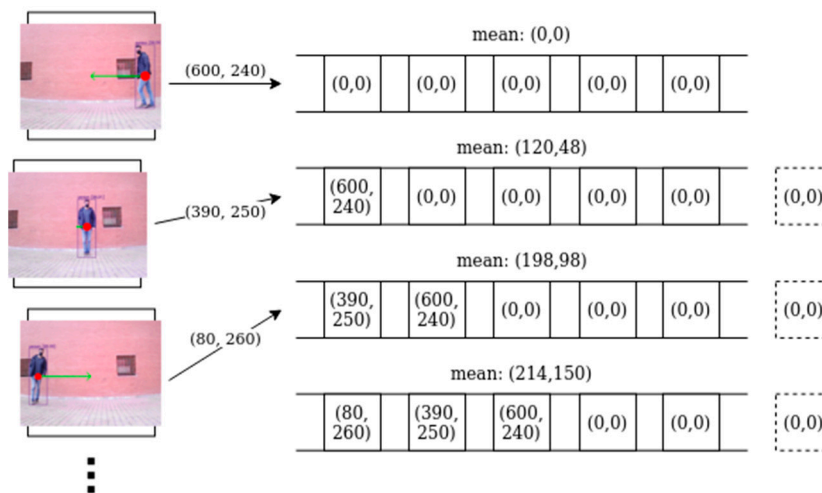


Figure 20. Follow-person identification scheme.

Therefore, the perception block has a single output: the detected person to follow (if one exists). If there is no-one, the aircraft will carry out a search algorithm consisting of circling until it finds a person to follow.

Note that even though perception is focused on detecting people, the application is easily transferable to tracking other objects that the network can detect, see “cars” or “horses”, for example. In addition, the YOLOv4 network has been chosen, but the detection could be carried out with another network, with a simple integration cost in the infrastructure.

#### 4.2.2. Control

The control block is practically identical to the one in the follow-color application (see Figure 15). The block input is unique, the person detected by the perception block. On the bounding box of the detection certain characteristics are calculated that allow estimation of the errors to be corrected by the controllers with their outputs, the new speed orders of the aircraft.

The controllers used are again three PIDs, forward control ( $v_x$ ), height control ( $v_z$ ), and yaw control ( $yaw\_rate$ ). The height and yaw controls are identical, while the forward control is slightly different:

$$e_x = \frac{total\_area}{det\_area} - \frac{total\_area}{target\_area} \tag{4}$$

$$e_z = c_y - obj_y \tag{5}$$

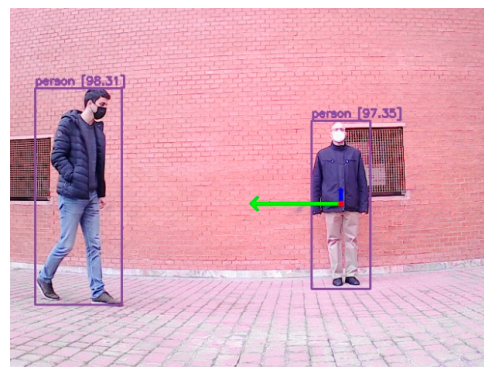
$$e_{yaw} = c_x - obj_x \tag{6}$$

The centroid is calculated on the bounding box of the person. The position of the centroid ( $obj_x, obj_y$ ) is used together with the position of the center of the image ( $c_x, c_y$ ) to compute the height and yaw error (Equations (5) and (6)). The forward error (Equation (4)) is calculated as the difference between the ratio of the two areas. On the one hand, the first ratio is obtained with the total area and the area of the detected object, and on the other hand, the second ratio is calculated with the total area and a reference area. The subtraction

of these two areas ensures an acceptable distance to the detected person. This distance is guaranteed because the person will always occupy a given percentage of the image area. This percentage of occupancy has been calculated experimentally with a value of twenty ( $\text{total\_area}/\text{target\_area} = 20$ ), meaning that the area of the bounding box of the person occupies one-twentieth of the image.

The network is robust against the relative orientation between the camera and the person (profile view, from behind, etc.) and the position of the person to be detected (standing, sitting, crouching, etc.). In addition, the network also responds correctly to partial detections due to occlusions or similar problems. Using the detection area as a progress control offers a good result in these cases.

Figure 21 shows the response given by the control block to input with several detections. The arrows indicate the direction that the drone will take to correct the existing error.



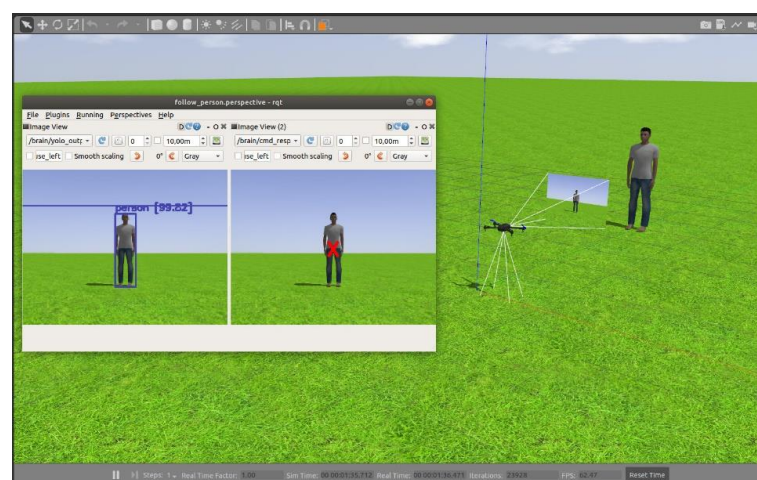
**Figure 21.** Control block response in Follow-person.

#### 4.2.3. Experiments and Results

The results obtained are presented in a similar way to the previous application, through different tests or experiments. We started from a simple simulated case, and after overcoming it, a new case with some added difficulty was tested, until considering that the application is robust enough to be tested on a real aircraft.

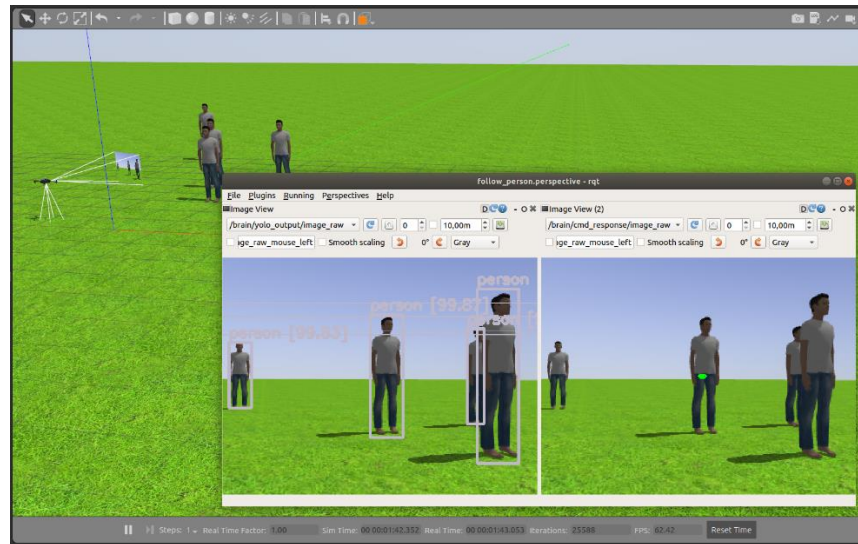
Tests in real aircraft are carried out first on the Tello as it is a smaller aircraft and can be flown indoors. After passing the tests on the smaller drone, the experiments were carried out on the larger aircraft, the PX4 itself. Thus, the application has been tested on the three available platforms.

The experiments have been simulated, starting from a simple experiment with the person to be followed in a static position (see Figure 22).



**Figure 22.** Follow-person, single static model.

Secondly, the model has been given movement, testing the application in a more realistic (and more complex) case. Finally, the application has been tested in an environment with various models of people, moving the models at will (Figure 23).



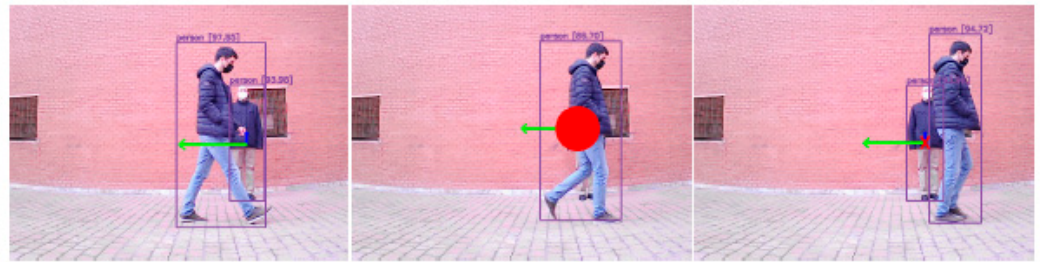
**Figure 23.** Follow-person, multiple models that can be moved at will.

After completing these first simulated tests successfully, the application has been tested with the first of the real UAVs, the Tello. Similarly, we have started from a simple case, with a single static person, a mannequin in this case. Next, the dummy has been replaced by a person, capable of moving in the different control axes, where a correct functioning of the tracking has been observed. We should emphasize that the experiments carried out with the Tello have been carried out indoors, where the traditional use of drones, by GPS position, is not possible.

Finally, the application has been tested outdoors with the largest aircraft. The correct operation of the perception block has been verified with the processing onboard the aircraft. Among the experiments carried out, the operation of the application against occlusions among several people has been tested. There have been two experiments, one where the crossing of the person is behind the element to be followed (see Figure 24) and another more complex one where the crossing is made in front of the person on which it is being crossed (see Figure 25).



**Figure 24.** Follow-person, a person crossing behind the element to be followed.



**Figure 25.** Follow-person, a person crossing in front of the element to be followed.

In Figure 25, it can be seen how detection is lost when the person identified as being followed is hidden from the second person who crosses. However, identification allows for keeping track of the right person when it is detected again.

The tests carried out with real aircraft have been diverse and of varying difficulty. The experiments have been numerous; not only has the object to be followed been provided with movement but it has also been tested with different people and their positions, the number of people in the image, lighting, and environments. Figure 26 shows examples of the different circumstances tested.



**Figure 26.** Examples of the Follow-person application under different circumstances.

The different experiments have been collected in various videos available for viewing in a playlist, that compiles all the tests performed: <https://youtube.com/playlist?list=PL2ebURGAzRwtsXui0GPFuQsIBJotnTIXj>, accessed on 26 October 2022.

## 5. Discussion

This work presents a new infrastructure for UAV programming available to the community. The correct operation of this infrastructure has been demonstrated through the use of three different aircraft and the development of two vision-based different applications.

The developed middleware infrastructure has been designed with a modular architecture based on ROS. The components are divided into programs with various nodes communicating through multiple ROS topics and services. The code has been developed in Python, and the main programs such as DroneWrapper or TelloDriver have an extension of 700 and 600 lines of source code, respectively, not including all the launch, simulation, test files, etc., elaborated during development.

The development of tools for programming multi-copters has been completed with the proposal of the DroneWrapper middleware. The infrastructure based on the MAVROS and MAVLink standards presents speed control as a featured novelty (common in mobile robotics but unusual in aerial robotics). It also incorporates other methods such as obtaining information from vision sensors, thus enabling the construction of visual applications making use of the infrastructure.

The selection of ROS allows for guaranteeing security and robustness. In addition, the chosen design facilitates high usability, since the programming interface offered to the user is straightforward to use.

The use of different UAVs has been accomplished with the use of three different multi-copters, both real and simulated. The different nature of the selected drones allows the horizontality of the infrastructure to be demonstrated.

Finally, the development of different applications has been overcome with two vision-based examples, which have been experimentally validated. The two proposed applications offer different types of technical complexity, being feasible for all types of users, whether they are novices or experts in the field of robotics. They include a perceptive part based on vision, using classic (color filtering) or modern techniques (deep learning), and a reactive control part for the motors, in speed, that uses PID controllers.

## 6. Conclusions

This research work concludes with a working version of the infrastructure published as free software on GitHub, used not only in the proposed applications but also in other free software projects such as Unibotics [38] or BehaviorMetrics [39] by JdeRobot [40]. The source code of the project is located in two different public repositories: part of the code is hosted in JdeRobot/drones (<https://github.com/JdeRobot/drones>, accessed on 26 October 2022), while other parts of the code can be found in RoboticsLabURJC/2021-tfm-pedro-arias (<https://github.com/RoboticsLabURJC/2021-tfm-pedro-arias>, accessed on 26 October 2022).

The final result offers a viable and very complete option for programming applications for UAVs, as has been demonstrated, for all types of users and diverse fields of application.

The proposed infrastructure offers a starting point for many real applications with drones. Despite being a solid product and in operation, there are multiple possibilities for improvement and functionalities with which to provide the software.

For example, the infrastructure can be extended to support new UAVs. It includes the development of new communication drivers with new types of aircraft. Examples of this would be other DJI UAVs, Parrot drones, or the Crazyflie from Bitcraze.

In addition, other types of sensors could be incorporated. The infrastructure currently only supports the use of cameras. Other sensors, see for example LiDAR sensors or radio-frequency (RF) beacons, may be useful for some user applications.

Finally, new functionality could be added to the user programming interface. New options should allow the user to perform tasks such as conventional (global) navigation or obtaining more data about the aircraft.

**Author Contributions:** Conceptualization, P.A.-P., D.M.G., J.M.C. and P.C.; Formal analysis, P.A.-P.; Funding acquisition, J.M.C.; Investigation, J.F.-C. and D.M.G.; Methodology, D.M.G., J.M.C. and P.C.; Project administration, J.M.C.; Resources, D.M.G. and P.C.; Software, P.A.-P. and J.F.-C.; Supervision, J.M.C.; Validation, P.A.-P.; Writing—original draft, J.F.-C.; Writing—review and editing, P.A.-P., J.F.-C., D.M.G. and P.C. All authors have read and agreed to the published version of the manuscript.

**Funding:** The research leading to these results has received funding from RoboCity2030-DIH-CM, Madrid Robotics Digital Innovation Hub, S2018/NMT-4331, funded by “Programas de Actividades I+D en la Comunidad de Madrid”, and cofunded by Structural Funds of the EU.

**Institutional Review Board Statement:** Not applicable.

**Informed Consent Statement:** Not applicable.

**Data Availability Statement:** Not applicable.

**Conflicts of Interest:** The authors declare no conflict of interest.

## References

- Kim, J.; Kim, S.; Ju, C.; Son, H.I. Unmanned aerial vehicles in agriculture: A review of perspective of platform, control, and applications. *IEEE Access* **2019**, *7*, 105100–105115. [CrossRef]
- Radoglou-Grammatikis, P.; Sarigiannidis, P.; Lagkas, T.; Moscholios, I. A compilation of UAV applications for precision agriculture. *Comput. Netw.* **2020**, *172*, 107148. [CrossRef]
- Yao, H.; Qin, R.; Chen, X. Unmanned aerial vehicle for remote sensing applications—A review. *Remote Sens.* **2019**, *11*, 1443. [CrossRef]
- Sanchez-Lopez, J.; Fernandez, R.S.; Bayle, H.; Pérez, C.S.; Molina, M.; Pestana, J.; Campoy, P. AEROSTACK: An architecture and open-source software framework for aerial robotics. In Proceedings of the International Conference on Unmanned Aircraft Systems (ICUAS), Arlington, VA, USA, 7–10 June 2016; IEEE: Piscataway, NJ, USA, 2016; pp. 332–341. [CrossRef]
- Real, F.; Torres-González, A.; Ramon, P.; Capitán, J.; Ollero, A. Unmanned aerial vehicle abstraction layer: An abstraction layer to operate unmanned aerial vehicles. *Int. J. Adv. Robot. Syst.* **2020**, *17*, 1729881420925011. [CrossRef]
- Báca, T.; Petrлік, M.; Vrba, M.; Spurný, V.; Penicka, R.; Hert, D.; Saska, M. The MRS UAV System: Pushing the Frontiers of Reproducible Research, Real-world Deployment, and Education with Autonomous Unmanned Aerial Vehicles. *J. Intell. Robot. Syst.* **2021**, *102*, 1–28. [CrossRef]
- Furrer, F.; Burri, M.; Achtelik, M.; Siegart, R. RotorS—A Modular Gazebo MAV Simulator Framework. *Stud. Comput. Intell.* **2016**, *625*, 595–625. [CrossRef]
- Xiao, K.; Tan, S.; Wang, G.; An, X.; Wang, X.; Wang, X. XTDrone: A Customizable Multi-rotor UAVs Simulation Platform. In Proceedings of the 4th International Conference on Robotics and Automation Sciences (ICRAS), Wuhan, China, 12–14 June 2020; IEEE: New York, NY, USA, 2020; pp. 55–61. [CrossRef]
- Assaf, K.; Ben-Moshe, B. A Robust and Accurate Landing Methodology for Drones on Moving Targets. *Drones* **2022**, *6*, 98. [CrossRef]
- Chen, P.; Dang, Y.; Liang, R.; Zhu, W.; He, X. Real-Time Object Tracking on a Drone With Multi-Inertial Sensing Data. *IEEE Trans. Intell. Transp. Syst.* **2018**, *19*, 131–139. [CrossRef]
- Chakrabarty, A.; Morris, R.; Bouyssounouse, X.; Hunt, R. Autonomous indoor object tracking with the Parrot AR.Drone. In Proceedings of the International Conference on Unmanned Aircraft Systems (ICUAS), Arlington, VA, USA, 7–10 June 2016; IEEE: New York, NY, USA, 2016; pp. 25–30. [CrossRef]
- Zou, J.; Dai, X. The Development of a Visual Tracking System for a Drone to Follow an Omnidirectional Mobile Robot. *Drones* **2022**, *6*, 113. [CrossRef]
- Javan, F.D.; Samadzadegan, F.; Gholamshahi, M.; Mahini, F.A. A Modified YOLOv4 Deep Learning Network for Vision-Based UAV Recognition. *Drones* **2022**, *6*, 160. [CrossRef]
- Carrasco, A.M. Arquitectura de Software Para Navegación Autónoma y Coordinada de Enjambres de Drones en Labores de Lucha Contra Incendios Forestales y Urbanos. Ph.D. Thesis, Universidad Carlos III de Madrid, Getafe, Spain, November 2020.
- UCI de Madrid, Laboratorio de Sistemas Inteligentes. Available online: <https://lsi.uc3m.es/> (accessed on 26 October 2022).
- Arduino, A.G. Available online: <https://www.arduino.cc/> (accessed on 26 October 2022).
- Raspberry Pi Foundation. Available online: <https://www.raspberrypi.org/> (accessed on 26 October 2022).
- Vega, J.; Cañas, J.M. PiBot: An open low-cost robotic platform with camera for STEM education. *Electronics* **2018**, *7*, 430. [CrossRef]
- Industries GoPiGo. Available online: <https://www.dexterindustries.com/gopigo3/> (accessed on 26 October 2022).
- NVIDIA Jetson Corporation. Available online: <https://www.nvidia.com/es-es/autonomousmachines/embedded-systems/> (accessed on 20 July 2021).
- NVIDIA CUDA. Available online: <https://developer.nvidia.com/cuda-zone> (accessed on 26 October 2022).
- NVIDIA JetPack SDK. Available online: <https://developer.nvidia.com/embedded/jetpack> (accessed on 26 October 2022).
- Python Software Foundation. Available online: <https://www.python.org/> (accessed on 26 October 2022).
- C++. Available online: <https://www.cplusplus.com/> (accessed on 26 October 2022).
- Mahtani, A.; Sanchez, L.; Fernandez, E.; Martinez, A. *Effective Robotics Programming with ROS*; Packt Publishing Ltd.: Birmingham, UK, 2016.
- Cooney, M.; Yang, C.; Arunesh, S.; Siva, A.P.; David, J. Teaching robotics with robot operating system (ROS): A behavior model perspective. In Proceedings of the Workshop on “Teaching Robotics with ROS”; European Robotics Forum, Tampere, Finland, 13–15 March 2018; Volume 2329, pp. 59–68.



27. Open Source Robotics Foundation. rospy. Available online: <https://wiki.ros.org/rospy> (accessed on 26 October 2022).
28. MAVROS. Available online: <https://wiki.ros.org/mavros> (accessed on 26 October 2022).
29. OpenCV Team. Available online: <https://opencv.org/> (accessed on 26 October 2022).
30. Dronecode Project, PX4. Available online: <https://px4.io/> (accessed on 26 October 2022).
31. Redmon, J.; Farhadi, A. YOLOv3: An Incremental Improvement. *arXiv* **2018**, arXiv:1804.02767.
32. Bochkovskiy, A.; Wang, C.-Y.; Liao, H.-Y.M. YOLOv4: Optimal Speed and Accuracy of Object Detection. *arXiv* **2004**, arXiv:2004.10934.
33. Lin, T.-Y.; Maire, M.; Belongie, S.; Hays, J.; Perona, P.; Ramanan, D.; Dollár, P.; Zitnick, C.L. Microsoft coco: Common objects in context. In Proceedings of the European Conference on Computer Vision, Zurich, Switzerland, 6–12 September 2014; Lecture Notes in Computer Science. Volume 8693, pp. 740–755.
34. DJI, Tello SDK. Available online: <https://dl-cdn.ryzerobotics.com/downloads/Tello/Tello%20SDK%202.0%20User%20Guide.pdf> (accessed on 26 October 2022).
35. Condés, I. Embedded Solution for Person Identification and Tracking with a Robot. Master's Thesis, Universidad Carlos III de Madrid, Leganés, Spain, June 2020. Available online: [https://gsync.urjc.es/jmplaza/students/tfm-deeplearning-person\\_following-nacho\\_condes-2020.pdf](https://gsync.urjc.es/jmplaza/students/tfm-deeplearning-person_following-nacho_condes-2020.pdf) (accessed on 26 October 2022).
36. Condés, I.; Cañas, J.M.; Perdices, E. Embedded Deep Learning Solution for Person Identification and Following with a Robot. In Proceedings of the Workshop of Physical Agents, Madrid, Spain, 19–20 November 2020; Springer: Cham, Switzerland, 2020; pp. 291–304.
37. OpenRobotics. Standing Person. Available online: <https://fuel.ignitionrobotics.org/1.0/OpenRobotics/models/Standing%20person> (accessed on 26 October 2022).
38. JdeRobot. Unibotics. Available online: <https://unibotics.org/> (accessed on 26 October 2022).
39. JdeRobot. BehaviorMetrics. Available online: <https://jderobot.github.io/BehaviorMetrics/> (accessed on 26 October 2022).
40. JdeRobot. JdeRobot. Available online: <https://jderobot.github.io/> (accessed on 26 October 2022).